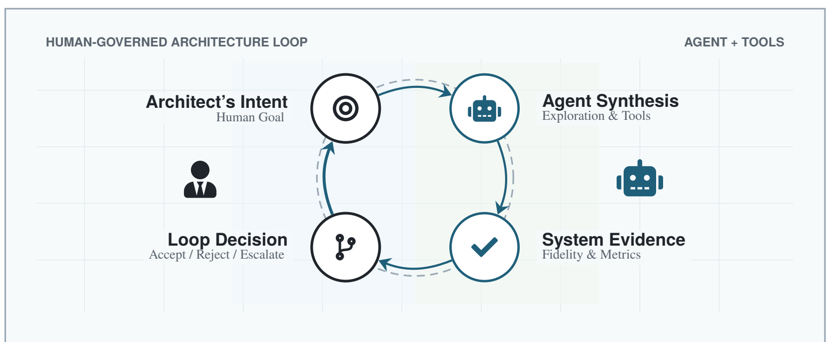
PREVIEW EDITION

Vijay Janapa Reddi

Architecture 2.0

Designing AI-Assisted Loops for Computing Systems



Work in progress

Preview v0.1.0 for the ISCA 2026 Architecture 2.0 workshop.

July 6, 2026

Table of contents

Preface ... ix
- Book structure ... xi
- Who this book is for ... xi
- How to read this book ... xii

Foreword ... xiii

Acknowledgments ... xiv

About the Author ... xv

1 The Architecture 2.0 Moonshot ... 1
- 1.1 Ask What AI Can Do for Architecture ... 3
- 1.2 From Architecture 1.0 to Architecture 2.0 ... 5
- 1.3 The Architecture Moonshot ... 7
- 1.4 Why the Prompt Spans the Stack ... 14
- 1.5 Architecture Development Spans Three Roles ... 17
- 1.6 Efficiency Claims Need Rejectable Loops ... 19
- 1.7 Boundaries of the Argument ... 21
- 1.8 Conclusion ... 22
- 1.9 Open Research Questions ... 23

2 Why Classical Architecture Loops Strain ... 25
- 2.1 Classical Loops Already Use Feedback ... 28
- 2.2 Cadence and Gates Manage Risk ... 29
- 2.3 Architecture Levers Add State ... 31
- 2.4 Specialization and Chiplets Expand Search ... 32
- 2.5 Specialized Hardware Needs a Software Loop ... 35
- 2.6 Software Changes Faster Than Silicon ... 37
- 2.7 Physical Constraints Move Into Architecture ... 39
- 2.8 Engineering Cost Creates the Scissors Gap ... 42
- 2.9 Feedback and Verification Become the Bottleneck ... 45
- 2.10 Architecture Violates Generic AI Assumptions ... 45
- 2.11 AI Helps Only When the Loop Is Designed ... 48
- 2.12 Conclusion ... 48
- 2.13 Open Research Questions ... 49

3 Architectural Claims and Design Loops ... 50
- 3.1 The Architectural Claim Is the Unit of Review ... 53
- 3.2 The Design Loop Is the Unit of Analysis ... 55
- 3.3 Design Spaces Make Claims Meaningful ... 57

3.4 The Architecture 2.0 Ontology 58
3.5 The Compact Framework 59
3.6 Autonomy Is Earned, Not Declared 63
3.7 Intent Defines the Task 65
3.8 Representations and World Models 66
3.9 Tools Become Environments 66
3.10 Agents and Methods Have Roles in a Compound System 67
3.11 Feedback Becomes Evidence 68
3.12 The Design-Loop Card 69
3.13 How the Rest of the Book Uses the Ontology 70
3.14 Conclusion 70
3.15 Open Research Questions 71

4 Representations and World Models 73
4.1 Why Architecture Data Is Not Web Data 74
4.2 Sample Cost Is Architecture Data 80
4.3 Architecture Descriptions as Boundary Objects 83
4.4 Unstructured Design Data and Its Cost 84
4.5 QuArch as a Stress Test 85
4.6 Toward Architecture World Models 87
4.7 Provenance, Coverage, and Negative Traces 89
4.8 When a Representation Becomes Actionable 90
4.9 Conclusion 91
4.10 Open Research Questions 92

5 Tools as Architecture Environments 94
5.1 Tools Shape the Research Question 95
5.2 Interfaces Are Action Boundaries 96
5.3 The Architecture Environment Abstraction 101
5.4 ArchGym as a Worked Example 104
5.5 Interfaces Make Loops Composable 105
5.6 Feedback Latency and Fidelity 106
5.7 Proxy Gaming and the Simulator Case 110
5.8 Building Environments for New Subfields 111
5.9 Environment Validity and Operating Discipline 113
5.10 Conclusion 114
5.11 Open Research Questions 115

6 Method Roles: Generate, Predict, Optimize 117
6.1 Match the Method to the Architecture Task 118
6.2 Hardware Awareness as Staged Capability 122
6.3 Generation: Proposing Candidates and Artifacts 126
6.4 Prediction: Estimating Behavior Before Full Evaluation 127
6.5 Optimization: Learning the Design Space 129
6.6 Sample Efficiency Under Expensive Feedback 130
6.7 Critique, Repair, and Explanation 133

6.8 Choosing a Method Under Constraints 134
6.9 Why No Single Algorithm Wins 136
6.10 Conclusion 137
6.11 Open Research Questions 137

7 Feedback, Evidence, and Trust 139
7.1 Feedback Budget Ledger and Feedback Economics 141
7.2 Fidelity Ladders and Evidence Ledgers 143
7.3 Commitment Levels and Reversibility 145
7.4 Rejection Authority 147
7.5 Proxy Mismatch, Metric Gaming, and Calibration 150
7.6 Security, IP, and Confidentiality Boundaries 151
7.7 Evidence Disputes and the Trust Checklist 154
7.8 Conclusion 156
7.9 Open Research Questions 157

8 Running the Lighthouse Loop 159
8.1 Round One: Generate and Screen on a Proxy 161
8.2 Round Two: Escalate to a Simulation-Stage Estimate 162
8.3 Round Three: Reject on the Envelope 165
8.4 Round Four: Commit at an Honest Level 167
8.5 What the Loop Leaves Behind 170
8.6 Conclusion 170
8.7 Open Research Questions 171

9 Loop Patterns Across the Stack 173
9.1 A Template for Reading the Cases 174
9.2 AI-Assisted Loop Postures 178
9.3 Workload Characterization and Benchmark Construction 178
9.4 Fast Software Loops 180
9.5 Architecture Loops: Accelerators, Memory, and Chiplets 181
9.6 Domain-Specific Architecture and Code Generation 183
9.6.1 Progressive Lowering as the Architecture Loop 187
9.7 Co-Design Loops: Compute, Memory, Network, and Power 188
9.8 Deployment-Facing Architecture Loops: Runtime, Serving, and Datacenter Policy 191
9.9 High-Commitment Loops: RTL, Physical Design, and Verification . . . 193
9.10 The Loop Is Rejection-Bound 194
9.11 What Transfers Across Loops 198
9.12 Conclusion 198
9.13 Open Research Questions 199

10 What the Architect Owns 201
10.1 The Architect Owns the Loop 201
10.2 Return to the Moonshot 202
10.3 Nondelegable Architectural Responsibilities 205

10.4 Extracting and Codifying Tacit Knowledge 208
10.5 The Strongest Objections 210
10.6 Community Infrastructure for Architecture 2.0 211
10.7 Open Research Questions 214
10.8 Long-Horizon Challenge Tasks 215
10.9 From Capability to Standard 217
10.10Loop Invariants as Review Checks 219
10.11Beyond the Current Loop 221
10.12The Architect's Standing Obligation 221
10.13Final Thoughts: Engineering Discipline in a Fast-Moving Field 222

Appendices

A Bootstrapping an Architecture 2.0 Loop 225
A.1 The Architecture Loop Playbook 225
A.2 Choose a Bounded Task 227
A.3 Choose a Representation 228
A.4 Wrap the Environment 228
A.5 Assign the Method Role 228
A.6 Write the Evidence and Rejection Authority 229
A.7 Fill in the Minimal Design-Loop Card 229

B Design-Loop Card and Review Rubric 233
B.1 Why a Card, Not a Paper Summary 233
B.2 The Design-Loop Card Fields 235
B.3 A Machine-Checkable Schema 237
B.4 The Review Rubric 238
B.5 Card Conformance Levels 240
B.6 Adapting the Card for Different Contexts 240
B.6.1 1. For Research Papers (Architecture Claim Card View) 241
B.6.2 2. For Case Studies and Examples (Shareable Evidence Ledger) 241
B.7 Paper-to-Loop Exercise 242
B.8 Lighthouse Card Sketch 242
B.9 Common Failure Modes 244
B.10 Blank Template 244

C Architecture 2.0 Resource Catalog and Links 246
C.1 Use the Catalog as a Loop Checklist 249
C.2 Missing Infrastructure 250
C.3 Live Resource Directory 250
C.4 Architecture 2.0 Framing 251
C.5 The Living Tool Catalog 251
C.6 Architecture 2.0 Templates and Artifacts 251
C.7 Using and Citing the Preview 251
References 252

Preface

Computer architecture has a new question. For decades the field asked what machines should be built for new kinds of computation. Capable AI systems now pose the reverse question: what can those systems do for the practice of architecture itself? The reversal changes the scarce engineering act. When plausible artifacts become cheap to generate, the hard problem is no longer only producing a candidate accelerator, kernel, floorplan, or design report. It is deciding which artifact-backed claim deserves belief, comparison, rejection, escalation, or commitment. This book is about that shift from artifact scarcity to commitment scarcity. The destination is to make AI-assisted architecture claims credible, comparable, and reviewable. The mechanism is to treat the design loop as a first-class architectural object alongside the artifact, creating a structure with visible state, allowed actions, evidence, rejection authority, and commitment boundaries. The artifact still matters. The loop matters because it determines how that artifact is produced, evaluated, rejected, and justified.

> Architecture 2.0 is not a debate about whether we use AI, nor a catalog of what today's agents can do. It is the discipline of governing the design loop so that an AI-produced claim carries the evidence, boundaries, and rejection conditions a human needs to commit to it.

The broader shift is already underway. The Architecture 2.0 foundations article argues why AI methods and agents belong in modern computer system design, especially for computer architecture and hardware/software co-design, and sets out the vision, history, ecosystem, capability horizons, and levels of autonomy that could follow (Janapa Reddi and Yazdanbakhsh, 2025). This book takes up the next question and treats it on its own terms. Suppose an AI-assisted loop proposes a faster accelerator configuration, a lower-energy kernel, or a plausible physical-design move. What state did it inspect? What could reject the result? Who accepts the commitment if the evidence is wrong? Those are the questions this book treats as the practical core of Architecture 2.0.

That credibility question is familiar from a different field. Machine learning systems faced the same problem a decade ago. Claims were everywhere and comparison was hard. The answer was not a better model. It was measurement discipline, requiring shared workloads, defined scenarios, provenance, and rules that made a performance claim mean the same thing across systems. Benchmarking efforts mattered because they turned enthusiasm into evidence. Architecture 2.0 needs the same move one level up. The task is to make AI-assisted architecture claims as credible, comparable, and reviewable as the community learned to make AI-systems claims. One caveat travels with the analogy. Benchmarking earned comparability by fixing tasks, workloads, metrics, and submission rules, so two loop claims are directly comparable only when those match too. When they do not, the design-loop card, the compact loop record this book develops, still makes a claim reviewable and contrastable, which is often the achievable goal at the loop level. Two things earned MLPerf's comparability, and a self-attested card supplies neither: it fixed the output so only the system varied, one level below where the loop now sits,

and it added adversarial peer review, a submission round under shared rules in which competitors could reject each other's claims. In that sense MLPerf is already a worked Architecture 2.0 loop, a versioned workload packet with provenance and an independent rejection authority, and naming that governance is what the loop level still owes.

The central problem sits at the boundary between computer architecture, machine learning systems, benchmarking, and tool-based design. AI methods are powerful, but architecture progress depends on hardware and software interfaces, workload definitions, toolchains, evidence standards, and human judgment. That is why the unit of analysis here is the design loop, not the isolated model.

When the text or a figure draws a single agent, read it as a participant in the loop, not a claim about implementation. That participant might be one model, one tool-using agent, a workflow of models and scripts, or several specialized agents. The same test applies in every case: each participant needs visible state, legal actions, evidence obligations, a rejection path, and an architect-owned commitment boundary.

The argument is therefore data-centric in a specific sense. The limiting question is not only which model or agent is used. It is which parts of architecture work are made observable, including workload traces, design artifacts, tool outputs, constraints, rejected candidates, failed runs, and the provenance that ties feedback to a decision. Data-driven methods become credible only when the data records the design loop, not just its successful endpoints.

What follows is an operating framework, not a catalog. The field is moving quickly, and a catalog of today's agents, tools, and benchmarks would age before it was useful. The durable contribution is a way to describe an architecture design loop, judge its evidence, and decide what the architect still owns. Architecture 2.0 is therefore not primarily a survey of current AI agents for computer architecture. Agents are the forcing function. The foundation is a set of durable principles for making architecture design loops representable, governable, evidence-bearing, rejectable, and improvable as methods become more capable. The faster-moving record of who is doing what belongs with the community now forming around this topic, tracked in the living resource list of Appendix C. The book keeps the parts meant to last.

The recurring artifact is the design-loop card. The card is not a substitute for a paper, benchmark, simulator, or design review. It is the compact record that names the loop behind a claim, specifying the task, representation, environment, method role, feedback budget, evidence, negative traces, rejection authority, and human decision. The book develops those fields chapter by chapter, then collects the card and review rubric in Appendix B.

The book's central act is *computing-system synthesis*: turning architectural intent into defensible computing-system designs for chips, accelerators, memory systems, and the toolchains and workloads around them. Inside the book, *system synthesis* is shorthand for this architecture-level computing-system synthesis. Logic synthesis turns logic into circuits. High-level synthesis turns behavior into hardware. Program synthesis turns specifications into programs. System synthesis operates at the architecture level, where

intent, constraints, representations, tools, feedback, evidence, and human judgment have to be coordinated before a design deserves commitment.

The framework keeps returning to three reader questions. What state is visible to the loop? What actions and tools can change that state? What evidence can reject a result before a person or organization commits to it? Later chapters give those questions reusable names, such as loop contracts, architecture environments, method roles, evidence ledgers, rejection authority, commitment boundaries, and design-loop cards.

Book structure

The book is organized into ten chapters that build the Architecture 2.0 framework:

- **Chapter 1** and **Chapter 2** establish the paradigm shift. They show why classical design loops no longer scale against engineering costs, and why AI assistance requires treating the loop itself as an explicitly designed object.
- **Chapter 3** introduces the core ontology, detailing the shift from artifacts to claims and the compact framework of intent, representations, environments, method roles, and evidence.
- **Chapter 4**, **Chapter 5**, and **Chapter 6** operationalize the framework. They detail how raw data becomes actionable provenance, how ad-hoc tools become trusted environments, and how AI methods map to specific design roles.
- **Chapter 7** and **Chapter 8** focus on evaluation. They build a sequenced ledger for trust, from feedback budgets to rejection authority, and demonstrate a bounded, rejectable turn through the loop.
- **Chapter 9** scales these concepts across the system stack, showing how the loop contract tightens as feedback cost and irreversibility increase.
- **Chapter 10** concludes by identifying the architect's nondelegable responsibilities and the shared community infrastructure the field needs next.

Who this book is for

This is a compact synthesis for readers who already know computer architecture and want a framework for AI-native architectural practice. It assumes fluency in architecture and does not re-teach it. It does not assume a machine-learning or reinforcement-learning background; the ML and RL ideas it borrows are glossed in margin footnotes at first use, so an architect can follow the argument without prior AI training. It is not a survey of today's agents and tools, not a tutorial on building them, and not a forecast of future automation. It is an operating discipline for governing the design loops those tools run inside.

Within that scope, the book serves several readers. A graduate student entering the area should find the vocabulary and the lay of the land. A reviewer should find a way to ask

what a project exposes and what could reject its result, not only what result it reports. Its author should find a way to state the architectural claim so its evidence, boundaries, and rejection conditions are visible. A practitioner should find a way to reason about where an agent may act and where the architect must still decide. If the book succeeds, each of these readers should be able to do something afterward that was harder before. They will be able to name a loop, judge its evidence, and state what remains an architect-owned commitment.

How to read this book

Each chapter is framed by a guiding question and a short *What this chapter gives you* list of the moves you will be able to make. One running example, the lighthouse prompt of Chapter 1, a compact design request for a mobile extended-reality (XR) subsystem, travels through the book as a shared example.

For a quick pass, read the Preface and Chapter 1, then skim the design-loop card in Appendix B. For a one-hour pass, read Chapter 2, Chapter 3, and Chapter 9 to see why the loop becomes a first-class object of design alongside the artifact. For a first project pass, use Appendix A to bound one loop before building a platform. For research review, read the chapters in order; the loop-role resource catalog and living link list in Appendix C show where the framework most needs better examples, tools, and evidence.

AI systems do not remove the architect. They raise what the architect must be good at. The work moves upward, toward intent, representation, evidence standards, rejection authority, and accountability for the final decision. The opportunity is not to wait for a system that designs a computer from a single sentence. It is to make the represented, instrumented, evidence-bearing design loop a first-class part of architectural practice, alongside the artifact it produces, and to build loops worthy of an architect's judgment.

Vijay Janapa Reddi

Foreword

Coming soon.

Acknowledgments

This book grew out of work and conversations across computer architecture, machine learning systems and benchmarking. I am grateful to the students, collaborators, colleagues, and broader research community who pressure-tested the framing, challenged weak claims, and insisted on evidence over enthusiasm. Their questions and examples shaped the emphasis on design loops, evidence standards, rejection, and human architectural judgment throughout the book.

I am especially grateful to Partha Ranganathan, Vice President and Engineering Fellow at Google, who delivered the inaugural keynote for the online Architecture 2.0 workshop.

For feedback, discussions, and challenges that sharpened this work, I am also grateful to Siddharth Garg (New York University), Brian Hirano (Micron), Jenny Huang (NVIDIA), Tushar Krishna (Georgia Institute of Technology), Srivatsan Krishnan (NVIDIA), Benjamin Lee (University of Pennsylvania), Yingyan (Celine) Lin (Georgia Institute of Technology), Jason Lowe-Power (University of California, Davis), Martin Maas (Google DeepMind), Ankita Nayak (Qualcomm), Matt Sinclair (University of Wisconsin-Madison), Srinivas Sridharan (NVIDIA), Zishen Wan (Columbia University), Amir Yazdanbakhsh (Google DeepMind), and Cliff Young (Google DeepMind).

Any errors that remain are my own.

About the Author

Vijay Janapa Reddi is the Gordon McKay Professor of Electrical and Computer Engineering at Harvard University. His work sits at the boundary of computer architecture, runtime systems, edge computing, and machine learning systems, with a recurring focus on how measurements, benchmarks, and shared artifacts turn systems ideas into practice. As a co-founder of MLCommons and one of the architects of the MLPerf benchmarks, he has helped make machine-learning performance a measurable, reproducible engineering claim. Architecture 2.0 grows from a conviction that runs through that work. Good engineering is not about which tools or methods you use but about how well you build, holding efficiency, safety, robustness, and reliability to evidence rather than to assertion. This book carries that discipline into the AI-assisted era.

Chapter 1

The Architecture 2.0 Moonshot

"The purpose of computing is insight, not numbers."
— Richard Hamming, *Numerical Methods for Scientists and Engineers* (1962)

Author's Note: Richard Hamming, a pioneer in computer science, famously emphasized that computing should yield insight rather than just raw data. For us, this quote anchors the book's core premise: AI-assisted architecture isn't about blindly generating billions of candidate designs, but about accelerating our understanding of the design space.

The crux

What would it take for AI to help architects turn intent into a system design that others can check, compare, and reject, not just a plausible answer?

This shift toward designing loops is easiest to state as a rule. A prompt is not yet an architecture claim. It becomes one only when the loop explicitly defines its five-part execution state: what state it saw, what actions it allowed, what alternatives it rejected, what evidence supports the result, and who owns the commitment if the claim is wrong. When plausible artifacts are cheap, that burden of proof, not the artifact, becomes the scarce work. Without treating this loop as a rigorous system-level state machine, we risk building the software equivalent of dark silicon—wasting massive computational resources on unoptimized, unstructured generative runs.

The cost of skipping that proof shows up first where generation is already cheap.

Field note: Buying homes faster than it could say no

Zillow ran an automated home-buying business that priced each purchase from a forecasting model and committed real capital on its output. When the estimates drifted from what houses would actually resell for, nothing in the loop was fast or trusted enough to reject the bad buys before the money was spent. The company wound the business down in 2021 after an inventory writedown of roughly three hundred million dollars and cut about a quarter of its staff (Zillow Group, 2021). Generating an offer was cheap. Trusting one enough to commit was the scarce and ultimately binding step.
Takeaway. When generation gets cheap, the bottleneck moves to trusted rejection, and a loop that cannot reject faster than it commits will scale its mistakes rather than its wins.

Computer architecture is the discipline of turning workload intent, technology constraints, software assumptions, physical limits, and evidence into credible hardware-software systems. It has never been only about which tools you use; it is about how well you build, holding efficiency, safety, robustness, and reliability to evidence rather than to assertion. Architecture 2.0 names the next step in that practice. Architects must design

not only artifacts, but also the design loops that produce, evaluate, reject, and justify those artifacts (Janapa Reddi and Yazdanbakhsh, 2025).

The practical reason architecture needs this discipline is efficiency, but efficiency no longer means a single performance number. The classical quantitative tradition already treated performance, cost, and power as coupled architectural questions (Hennessy and Patterson, 2017). Just as the end of Dennard scaling forced hardware to transition from general-purpose frequency scaling to specialized accelerators, the economic and latency limits of monolithic LLMs force agent systems to transition from raw prompting to structured execution loops. Dark silicon (the share of a chip that must stay powered off to fit its thermal budget), data-movement energy, warehouse scale, and carbon accounting make architectural coupling more difficult (Dennard et al., 1974; Esmaeilzadeh et al., 2011; Horowitz, 2014; Barroso et al., 2019; Gupta et al., 2021). Today, efficiency includes performance, energy, power delivery, reliability, scalability, sustainability, cost, verification burden, engineering effort, and time to credible evidence. Across that range, the design problem is increasingly coupled across hardware, software, tools, and deployment. This multidimensional coupling is exactly what makes naive generation insufficient. An AI-assisted loop must know which constraints, tools, and evidence can change the architectural claim, making structured loops a physical necessity rather than a design metaphor.

It helps to see how unusual this moment is. For roughly fifty years, processor generations improved through a remarkably stable loop. Device scaling delivered faster, cheaper, lower-power transistors on a predictable cadence; the quantitative method turned design choices into measured comparisons; when scaling slowed, microarchitecture, parallelism, and then domain specialization kept the gains coming (Hennessy and Patterson, 2019). That run of specialization is often called a new golden age for computer architecture; its quieter companion is a golden age of complexity, in which every design must navigate an explosion of workloads, software interfaces, and physical constraints. The artifacts changed every generation, but the loop that produced them stayed largely the same: propose, model, measure, and commit. What is new is not another lever inside that loop. It is that the loop itself, the rate at which credible choices can be proposed, evaluated, rejected, and justified, has become the bottleneck. What is missing is an explicit representation of that loop, and that representation is also the state an AI system would need before it can act inside the loop rather than merely emit another candidate. Chapter 2 develops that breakdown in detail; this chapter asks what a redesigned loop would have to be.

The quantitative method made architecture arguments measurable at the artifact level, giving the field a shared vocabulary for performance, cost, and power. Architecture 2.0 keeps that quantitative discipline but moves it one level up. We must now develop metrics and rigorous evaluation for the loop itself. The data, feedback, evidence, and rejection processes that produce the artifact must themselves be represented, measured, and designed.

The need for this explicit representation stems from the growing distance between an architectural intent and a software or hardware artifact that is implemented, tested, and credible enough to use. AI can shorten parts of that distance, but only when the work

between intent and artifact is represented as a loop with explicit state, actions, feedback, evidence, and decision authority. The moonshot is therefore not instant realization. It is a discipline for making the path to realization inspectable enough that humans and AI systems can work inside it.

The purpose of this chapter is to make that shift concrete. We build toward a single concrete design request, not because the request is already solvable, but because it reveals the hidden architecture state that any credible solution would need.

What this chapter gives you

After this chapter you can turn a compact architecture request into an AI-assisted loop contract. That means you can:

- explain why "AI for architecture" means designing the AI-assisted design loop, not prompt-to-chip generation;
- explain why cheap plausible artifacts from AI methods move the scarce work toward credible commitment;
- distinguish the familiar human-carried architecture loop from an explicit, represented Architecture 2.0 AI loop;
- decompose a one-line design request into the architecture state a generative model hides;
- treat efficiency as a multidimensional, loop-level property rather than a single metric;
- separate generation, prediction, and optimization as distinct AI method roles in a design loop.

1.1 Ask What AI Can Do for Architecture

The phrase "AI for architecture" can be read in a shallow way, using a model to generate text, write scripts, summarize papers, or propose configurations. All of those may be useful, but they are not the core shift. The deeper question is how the practice of architecture changes when AI systems can participate inside represented, instrumented, and checked design loops.

That distinction matters because architecture work has always been organized around loops. Architects frame a problem, choose abstractions, construct models or simulators, select workloads, explore alternatives, evaluate results, reject weak candidates, and revise assumptions. For example, an architect weighing a larger L2 cache frames the question (does it help this workload mix without hurting energy?), sets up a cycle-level simulator (like gem5) or a full-system register-transfer-level (RTL) emulator (like FireSim), selects a benchmark suite, sweeps cache sizes, associativities, and replacement policies, reads the resulting miss rates and energy estimates, rejects the configurations that help one workload while hurting another, and revises the design before committing it to RTL.

Tools already participate in this loop. Simulators, compilers, profilers, synthesis tools, spreadsheets, dashboards, and design reviews all mediate architectural judgment.

AI systems become interesting when they can act inside that loop rather than around it. They may generate candidates, call tools, summarize evidence, predict outcomes, search design spaces, critique assumptions, or coordinate subtasks. But participation is credible only if the loop exposes what the system can see, what it can change, how feedback is obtained, what evidence is trusted, and what can reject the result.

This book uses four names carefully. AI systems are participants in the loop, not the loop itself. Methods play bounded roles such as generation, prediction, optimization, critique, repair, verification, explanation, or coordination. Tools become useful architecture environments only when they expose actions, feedback, costs, and failure modes. Architects still own the intent, evidence standard, rejection authority, and final commitment.

Architect's checkpoint: Architect-Owned Commitment

AI methods can act within the design loop, proposing and testing candidates, but the human architect acts as the ultimate decision gate, retaining rejection authority and finalizing any commitment.

The term *participant* is often singular in the prose and figures, but it names an AI-assisted system in the loop. A real implementation may use one generative method, a pipeline of methods, or several specialized tools; the Architecture 2.0 question is still what each participant may see, change, test, reject, and return for architect-owned decision.

Those four roles together define a single discipline, Architecture 2.0, whose central act is architecture-level computing-system synthesis, turning intent into a defensible computing-system design. That act does not mean only logic synthesis or high-level synthesis. The short phrase is simple, but the loop is not. To synthesize a computing-system design credibly, the loop must coordinate constraints, representations, tools, methods, feedback, evidence, and human judgment. The two terms below fix the discipline and describe the bounded role each method plays inside the loop.

Architecture 2.0. Architecture 2.0 is the engineering discipline of building and governing the design loops that produce computing systems, alongside the artifact rather than in place of it, so that AI-assisted architecture claims can be made *credible, reviewable, and contrastable*, and comparable in MLPerf's sense once the field fixes a shared task, workload, metric, and rejection protocol.

Method role. A method role is the bounded job an AI method, model, search procedure, or tool wrapper plays inside the loop: generate, predict, optimize, critique, repair, verify, explain, or coordinate. The role is credible only when the loop defines what the method can read, what it can change, what feedback it receives, and what can reject its output.

That AI-assisted loops can automate bounded parts of producing computing-system design artifacts is the premise of Architecture 2.0, not its achievement. The achievement is a claim another architect can believe, reject if it is wrong, and trace to whoever owns the commitment, which is why the rest of the book is about evidence, rejection, and commitment rather than automation for its own sake.

It helps to place that act on a familiar ladder. The dates are approximate; the point is the widening unit of intent and the widening artifact produced.

- **1980s–1990s (Logic Synthesis):** Logic synthesis turns Boolean and RTL intent into gate-level structure, governed by libraries, physical constraints, and timing feedback (De Micheli, 1994).
- **1990s–2000s (High-Level Synthesis):** High-level synthesis turns algorithmic behavior into RTL candidates, offloading the tactical microarchitecture search from the human to the tool (Coussy and Morawiec, 2008).
- **2000s–2010s (Domain-Specific Synthesis):** Building on deductive program synthesis from the 1970s, domain-specific synthesis turns mathematical kernels and DSL intent into executable code and specialized hardware/software artifacts (Gulwani et al., 2017).
- **2010s–2020s (Agile Hardware Generation):** Agile hardware generation turns parameterized templates (e.g., Chisel, FIRRTL) into bespoke SoC instances, enabling rapid specialization through reusable IP libraries.
- **2020s–2030s (Computing-System Synthesis):** Architecture 2.0 turns workload intent, software contracts, and physical constraints into a defensible, verified system organization. After this point, this book often shortens the phrase to *system synthesis* when the context is clear.

The lower rungs are largely automated; the top rung is what Architecture 2.0 has to learn to support.

The architecture design loop is the object this book will make precise. For now, treat it as the repeated movement from intent to bounded action, feedback, evidence, rejection, revision, and architect-owned commitment.

The framework should be useful in three concrete situations. First, a researcher should be able to describe an AI-for-architecture paper by naming its task, representation, environment, method role, feedback, evidence, and human decision point. Second, a tool builder should be able to ask whether a harness records enough state for another method or team to learn from it. Third, an author or reviewer should be able to ask what would reject the result, not only what result was produced. Later chapters turn those needs into reusable cards, ledgers, and review checks.

1.2 From Architecture 1.0 to Architecture 2.0

To see what changes, it helps to name the practice this framework is moving beyond. Architecture 1.0 is the familiar practice of human-orchestrated artifact design. The architect defines the problem, chooses models, uses tools, interprets feedback, and decides what to build. This practice is not obsolete. It is the foundation on which the field stands.

Architecture 2.0 shifts the emphasis from human-crafted static pipelines to agentic, optimization-driven loops. The architect still owns intent, constraints, abstraction,

evidence standards, rejection, and accountability. But the architect must now also design the loop around the artifact. The architect must decide how tasks are represented, which tools become environments, which method roles are allowed, what feedback budget is available, what evidence is required, and what can say no.

The difference can be seen in a familiar design-space exploration. In Architecture 1.0, an architect might manually script a simulator sweep over cache sizes, associativities, and replacement policies, then inspect the results. In Architecture 2.0, the architect may design a loop in which a method proposes candidates, a surrogate estimates outcomes, a simulator evaluates selected points, a critic flags invalid assumptions, and a human decides whether the evidence is strong enough. The artifact may still be a cache hierarchy. The new contribution is the explicit, inspectable, and rejectable loop that produced it.

Surrogate model: A data-driven approximation used to quickly estimate the results of a computationally expensive simulator.

Figure 1.1 makes the shift explicit. The left loop is the familiar human-carried practice: intent, models, candidates, tool runs, and expert review are coordinated by architectural judgment. The right loop does not remove that judgment. It represents enough loop state, action boundaries, evidence, rejection, and decision authority that bounded AI methods can participate without becoming an uninspectable prompt-to-chip shortcut.

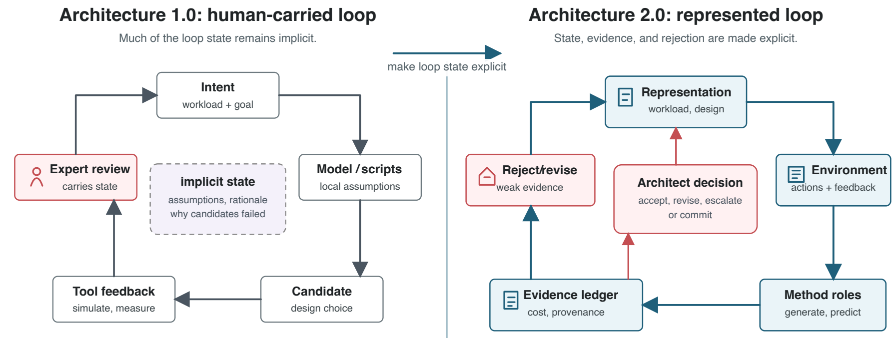


Figure 1.1: **The architecture design loop changes form:** Architecture 2.0 represents loop state, tool interfaces, evidence, rejection, and decision authority rather than leaving them implicit in human-carried practice.

This need for inspectability is why this book centers on *agentic design loops*, meaning loops where AI systems can choose bounded actions, call tools, revise state, and use feedback without owning the commitment. The AI-assisted participant, whether singular or a coordinated set of role-specific methods, is not the whole system. The governed loop is the system around the artifact. The automated parts must be embedded in representations, environments, evidence ledgers, and human decision points.

Design principle: Design the AI-assisted loop, not only the artifact

The key move is to design the AI-assisted loop, not only the thing it produces. A single artifact, a generated design from an AI model, a candidate configuration, or a plausible answer, is not the contribution. The durable object is the AI loop that exposes state, action, feedback, rejection, and decision, because that is what another architect can inspect, compare, reject, and improve before an AI method's output crosses a commitment boundary.

To see why the loop itself must be designed, work backward from the systems tradition. A useful AI-assisted system should not merely know how to call a simulator, propose RTL, or summarize papers. It should carry the habits that make systems engineering credible: suspect the bottleneck, account for memory movement and physical limits, compare tradeoffs numerically, preserve provenance, expose failure modes, and ask what evidence is strong enough for the next commitment. Those habits do not appear automatically because a model is capable. They have to be designed into the loop.

1.3 The Architecture Moonshot

Before introducing a concrete design prompt, we must fix two terms: *architecture* and *architect*. In this book, architecture does not mean only microarchitecture, a block diagram, or a chip artifact.

Architecture. Architecture is the hardware-software contract and system organization that turn workload intent and technology constraints into a defensible system design. It includes ISA and microarchitecture, memory and interconnect, accelerators and chiplets, compiler/runtime interfaces, physical-design constraints, verification, deployment, and the evidence used to justify choices.

Architect. The architect is the human who owns the intent, frames the problem, chooses the abstractions, sets the evidence standard, and retains rejection authority and final accountability for the committed design. AI methods act inside the loop; the architect owns it.

With those terms fixed, we ground the Architecture 2.0 framework in a running prompt sequence, building toward the lighthouse prompt this book traces throughout.

Sandbox Prompt: Design a matrix multiplication systolic array in Chisel and integrate it into a full SoC via a standard NoC (e.g., TileLink or AXI) with DMA engines. Verify its correctness against a software model, sweep its dimensions to find the optimal system-level tradeoff between latency, area, and memory stalls for a 16×16 workload, and produce a reviewable evidence ledger explaining why suboptimal variants were rejected.

While a sandbox prompt is excellent for learning to code an agentic loop locally (and we encourage students to start there), evaluating the full scale of Architecture 2.0 requires a moonshot. Our primary running example pushes the limit. It asks for a compute subsystem serving real-time mobile extended reality (XR), under a thermal-design-power (TDP) budget in a low-power (LP) mobile process class, and it asks for a report rather than a chip:

Lighthouse Prompt: Design a low-power, 64-bit RISC-V-based compute subsystem for an XRBench-class real-time mobile XR workload. Realize it as a vector-capable CPU, accelerator, or SoC block under a 3 W TDP target in a 3 nm-class LP mobile process, and return a design-space report with evidence and rejected alternatives.

The accelerator option spans fixed-function, reconfigurable-spatial (CGRA), and processing-in-memory (PIM) realizations, and a full program would also have to decide chiplet scale-out (die-to-die PHY, core-disable yield binning). The running example defers those so that one canonical sentence can travel through the book verbatim.

This sentence is quoted, fragment by fragment, throughout the book, so it is worth reading once in full. Figure 1.2 keeps it visible as an object of analysis. The top panel restates the prompt; the middle panel names the architectural obligations embedded in that request; and the bottom panel shows the loop turn that would be needed before an AI system's answer deserved architectural trust: represent the task, act through bounded tools and methods, gather evidence, reject weak outputs, decide what to commit, and revise the next turn. The point is not "type a prompt and get a chip." The point is to expose what a credible loop would need to know.

Lighthouse prompt. The lighthouse prompt is this compact Architecture 2.0 design request, used throughout the book: a request for a low-power, RISC-V-based mobile XR compute subsystem that must be answered with a design-space report carrying evidence and rejected alternatives.

Lighthouse prompt: Read the prompt as an AI-assisted loop request

Context. The lighthouse prompt is not a miniature product specification. It is a compact way to expose the state a credible AI-assisted architecture loop would have to carry.

In the Lighthouse prompt. "64-bit RISC-V-based compute subsystem" sets the ISA/ABI and software contract the method must respect. "XRBench-class real-time mobile XR workload" sets the workload scope the AI method must optimize. "vector-capable CPU, accelerator, or SoC block" sets the compute organization bounds for the method's action space. "3 W TDP target in a 3 nm-class LP mobile process" sets the physical-design envelope that can reject a generated proposal, and "design-space report with evidence and rejected alternatives" sets the evidence standard the AI loop owes the human architect. Those fragments force the AI-assisted loop to carry memory, data movement, compiler, runtime, reliability, and verification state rather than treating the prompt as a simple hardware generation task.

Takeaway. Treat the prompt as a recurring stress test. If a proposed AI-assisted loop cannot say what it represents, what it may change, what evidence it records, what it rejects, and who decides, it has not yet answered the prompt.

Finally, while our examples primarily trace the mobile XR prompt to keep node-level concepts concrete, modern architecture is distributed hardware-software co-design. The most extreme version of our lighthouse prompt scales to the datacenter:

Datacenter-Scale Lighthouse Prompt: Design a scale-out TPU supercomputer topology for training 10T-parameter mixture-of-experts[1] models. Ensure the tensor compiler (XLA/MLIR)

[1] A mixture-of-experts (MoE) model selectively routes each input to a small, specialized subset of its neural network weights, reducing compute cost per token while maintaining a large total parameter count.

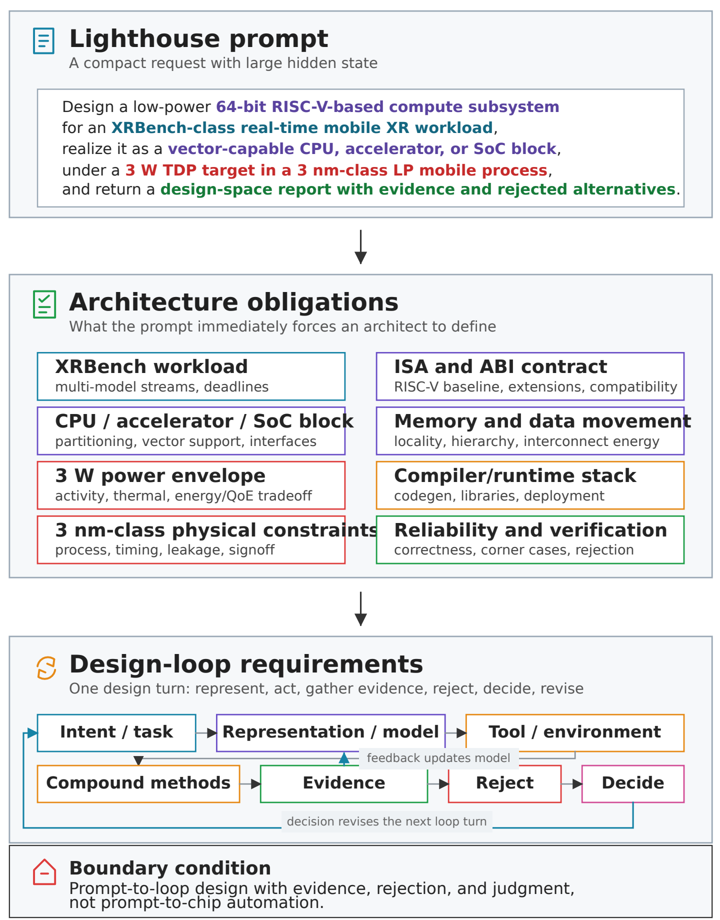


Figure 1.2: **The lighthouse prompt exposes hidden loop state:** It is useful because it surfaces the loop state, evidence, rejection, and decision authority a credible architecture answer would need.

> can discover an efficient mapping, bound tail latency across 10,000 optical links, co-design network topology with deadlock-free adaptive routing and virtual channels, define recovery targets and rejection gates for synchronous faults, and tolerate cloud multi-tenancy overheads (cold starts, cache pollution, memory disaggregation latencies). Crucially, the loop must explicitly measure and enforce software ecosystem preservation—rejecting any topology or ISA change that breaks compatibility with the dominant TPU software stack (XLA, JAX and PyTorch/XLA, and the collective-communication libraries).

Just like the mobile XR prompt, this datacenter prompt forces an AI-assisted loop to carry specific architectural state: “10T-parameter mixture-of-experts” sets the memory capacity and bandwidth bounds; “XLA/MLIR mapping” defines the software contract and

rejection authority for tensor placement; “network topology with deadlock-free adaptive routing” requires the loop to explicitly represent virtual channels and routing protocols as hard rejection gates to prevent bursty MoE AllToAll traffic deadlocks; and “cloud multi-tenancy overheads” forces the evaluation of general-purpose serverless latencies alongside AI training throughput. Furthermore, by elevating ecosystem compatibility to a first-class rejection authority, the AI agent is forced to respect the software moat—ensuring it is designing a usable data center platform, not just drawing a mathematically optimal but useless network graph. The evidence ledger for this prompt must record not just the winning topology, but why alternative routing schemes or fault-tolerance mechanisms failed under scale.

The prompt is deliberately extreme. No architect, and no model, can turn that sentence into a verified subsystem today, and that is the point. Pushing the request past what is currently feasible is a forcing function. It exposes the hidden architectural state, evidence, and decisions a credible design loop would have to handle, rather than letting a plausible-sounding answer conceal them.

With that concrete prompt in view, the word *moonshot* can be used precisely. The term should not mean a prediction that the field can already automate architecture end to end. X, the Moonshot Factory, frames a moonshot as the intersection of a huge problem, a radical solution, and a breakthrough technology that makes the solution plausible enough to pursue (X, The Moonshot Factory, 2025). This book adapts that structure to computer architecture.

Use the moonshot analogy only to name a loop requirement. A hard target becomes useful when it defines shared state, instruments, feedback, rejection rules, and commitment boundaries. Apollo, the Human Genome Project, DARPA’s Grand Challenge, and AlphaFold matter here as examples of targets that created shared tasks, instruments, datasets, tests, and evidence standards, not as a claim that architecture should imitate their politics, budgets, or technical domains (National Aeronautics and Space Administration, 2008; National Human Genome Research Institute, 2025; Defense Advanced Research Projects Agency, 2014; Jumper et al., 2021).

The common pattern is not that a moonshot is large or fashionable. It is that the target organizes a community around a hard problem, a different way of working, and an enabling technical shift.

There is a sharper lesson here than the analogy first suggests. Each example is remembered as a singular achievement, a flag on the Moon, a finished genome, a solved protein structure. But the durable contribution was rarely the single result. It was the shared task, the instruments, the methods, and the evidence standards that turned an exceptional effort into a process others could run. Architecture 2.0 takes that stance. The goal is not to celebrate one impressive design that a generative method happens to emit. It is to engineer the design and discovery process itself, the loop, the representations, the instruments, and the evidence, so that credible architecture results can be produced, checked, and reproduced rather than admired as isolated showcases. Computer architecture has done exactly this before, more than once.

The Mead and Conway VLSI design methodology turned chip design into a structured, reusable discipline with shared abstractions and design rules (Mead and Conway, 1980), and the reduced-instruction-set program reshaped the hardware/software contract around quantitative evidence (Patterson and Ditzel, 1980). Both changed the loop, not just the artifact. Architecture 2.0 belongs in that lineage. The next shift is in the design loop itself.

> **Architecture moonshot.** An architecture moonshot is an aspirational target at the intersection of three conditions: a grand architecture challenge that ordinary ad hoc coordination cannot evaluate credibly enough, a radical design-loop target that changes how architectural work is represented and evaluated, and an enabling AI/data/tool breakthrough that makes the target technically plausible enough to study without pretending it is solved.

Figure 1.3 shows that pattern in the architecture vocabulary used here. The phrase is deliberately about loop capacity, not about dismissing classical architecture practice. Classical architecture already has models, measurements, review, and signoff discipline. The moonshot is to make enough of that loop explicit, tool-connected, and evidence-bearing that AI methods can participate without hiding what would reject their output.

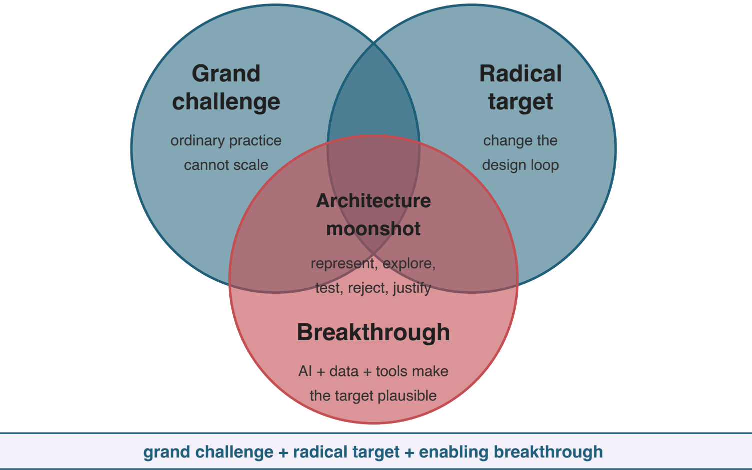


Figure 1.3: **A moonshot needs three conditions at once:** An architecture moonshot is not just a big challenge, a radical target, or a promising technology. It sits at the intersection of a grand architecture challenge, a design-loop target that changes how work is represented and evaluated, and an enabling AI/data/tool breakthrough that makes the target plausible enough to study.

This architecture-specific moonshot is worth naming because the pressure is real but the solution is not yet settled. The grand challenge is that hardware/software systems now span workloads, software stacks, ISAs, microarchitecture, accelerators, memory systems, EDA, physical constraints, verification, and deployment. The radical target

is not to automate architects away; it is to design interoperable loops that represent these choices, call the right tools, preserve evidence, record failures, and give humans rejection authority across ISA, microarchitecture, compiler, RTL, physical-design, and deployment boundaries. The enabling shift is the arrival of AI methods, architecture datasets, executable environments, and tool interfaces that make pieces of those loops plausible enough to study. Publicly reported examples already include reinforcement learning systems for physical-design exploration and learned circuit-generation loops with silicon evidence (Synopsys, 2023; Roy et al., 2021). Chapter 9 returns to those examples as loop-pattern cases. These examples show bounded loop pieces with evidence, not end-to-end architecture autonomy. The moonshot is not that these pieces exist; it is assembling them into a family of represented, evidence-bearing loops that a human can still govern.

Reinforcement learning: A machine learning method in which an agent learns to choose actions that maximize a cumulative reward signal from its environment, often used to navigate complex design spaces (Sutton and Barto, 2018).

Returning to the lighthouse prompt, each fragment sets a specific constraint. XRBench, a benchmark suite for mobile XR workloads, gives the prompt a workload anchor rather than a vague application label (Kwon et al., 2023). Real-time mobile XR stresses latency, energy, memory movement, model concurrency, sensing, graphics, and deployment constraints. End-to-end XR systems research has shown how tightly those stages couple across the full perception-to-display pipeline, which is what makes mobile XR a demanding architecture target rather than a single kernel to optimize (Huzaifa et al., 2021). A 64-bit RISC-V contract (an open standard instruction set architecture) gives the design an ISA boundary. Vector capability makes the compute organization concrete but does not decide whether the realization should be a CPU extension, accelerator, or SoC block. The 3 W TDP target and 3 nm-class LP mobile process assumption force the prompt into a contemporary technology envelope. The node is intentionally stated as a class rather than as a named foundry PDK (process design kit); current mobile SoCs are publicly described in 3-nanometer-class technology, but a credible architecture loop must still state which process, libraries, voltage assumptions, and signoff path it actually uses (Apple, 2024a,b). The requested deliverable is not merely a design. It is a design-space report with evidence and rejected alternatives.

Another way to read the prompt's requirements is through the familiar foundation-model stack. In the generic version, many kinds of inputs feed a central foundation model, and many downstream applications fan out on the other side. The architecture analogy is different. The left side includes workload traces, specifications, RTL or IP blocks, simulator configurations, process and library assumptions, verification logs, papers, and prior designs. The middle should not be read as a single language model or as a current capability claim. It is a placeholder for a represented design loop connected to tools, constraints, evidence, and rejection. The right side is not "a chip" as a single output. It includes ISA proposals, microarchitecture sketches, accelerator or SoC partitioning choices, RTL and testbench fragments, design-space reports, verification packages, and deployment decisions. Figure 1.4 should be read in two steps. Panel A shows the generic foundation-model pattern. Panel B translates each part of the pattern into architecture: the inputs become design artifacts and evidence, the middle becomes a represented

and tool-connected design loop, and the outputs become architecture deliverables with different commitment levels.

Foundation model: A large-scale AI model trained on a vast quantity of data that can be adapted to a wide range of downstream tasks.

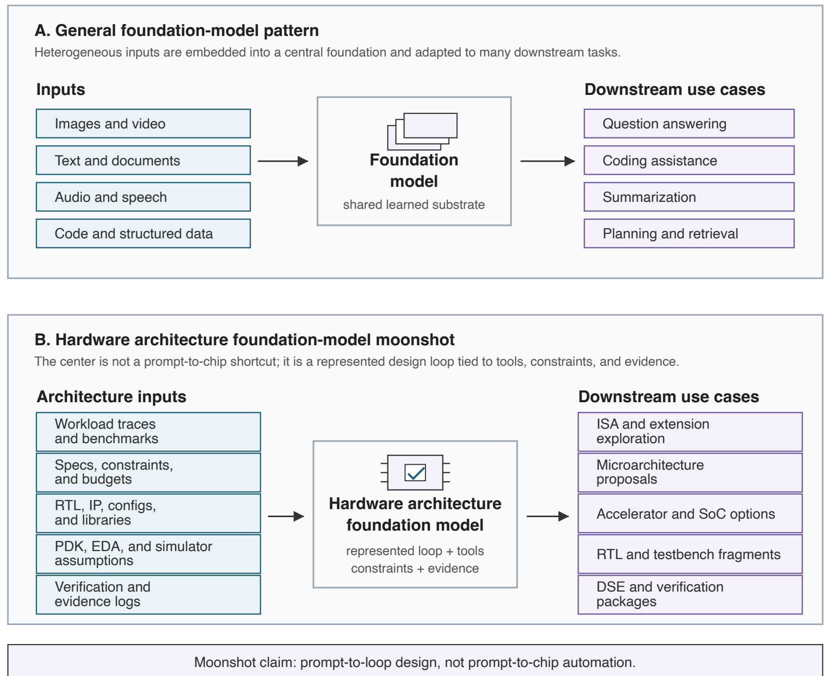


Figure 1.4: **A foundation-model analogy for architecture loops, not a prompt-to-chip shortcut:** The center is a represented design loop tied to architecture artifacts, tools, constraints, evidence, and rejection.

Read Panel B as a loop substrate, not as a single model call. For the lighthouse prompt, one loop turn might ingest an XRBench trace slice and a software pipeline description, generate vector-width, local-memory, and CPU/accelerator partition candidates, run a cheap performance and power proxy, reject candidates that miss the latency or 3 W envelope, and preserve the rejected alternatives as evidence for the next turn. A higher-commitment turn would replace some of that proxy evidence with simulator runs, compiler output, RTL fragments, or physical-design feedback. The foundation-model analogy is useful only if it points to that maintained architecture state and evidence path.

Architect's checkpoint: Higher-Commitment Decision Gate

Before a design loop transitions from proxy evidence to expensive simulator or physical-design feedback, the architect must evaluate the retained evidence and authorize the higher-commitment turn. Automation does not cross this boundary autonomously.

The prompt should be treated as a moonshot, not as a current capability claim. A present-day generative method may draft a plausible answer. It may produce a list of architectural choices, cite related ideas, or generate code fragments. That is not enough. The useful question is what loop would be required before such an answer deserved architectural trust. The book uses this prompt as a spine rather than as the only example. Later chapters zoom into facets of the same request: memory and data movement, software drift, chiplet partitioning, verification, physical design, and deployment. Additional examples appear only when a facet needs a more specific loop.

1.4 Why the Prompt Spans the Stack

The prompt is architectural because each phrase creates obligations beyond the surface words. A credible loop must track workload behavior, software contracts, hardware organization, physical feasibility, evidence, and rejection paths together. Table 1.1 keeps that obligation compact. The table is not meant to exhaust every subtask. It is a reader's checklist for why the prompt crosses boundaries that architecture cannot ignore.

Read the table as a stack of obligations rather than as a shopping list. The workload phrase says what behavior the design must serve. The ISA and compute phrases say what boundary the hardware/software interface must expose. The power and process phrase says what physical world can reject the idea. The report phrase says what kind of evidence the loop owes the architect before the answer deserves trust. Each row therefore names both a design decision and a way for the loop to be wrong.

Table 1.1: **Prompt fragments create architecture obligations:** The lighthouse prompt maps to decisions about workload definition, software and ISA contracts, hardware organization, physical feasibility, and evidence.

Prompt fragment	Architectural decisions	Evidence or rejection need
XRBench mobile XR	Workload slice, input distribution, quality-of-service target, latency deadline, memory traffic, software pipeline, and drift assumptions.	Trace provenance, benchmark version, statistically rigorous phase clustering (e.g., SimPoint), and rejection when results over-optimize a single hot loop or miss real-time behavior.

Prompt fragment	Architectural decisions	Evidence or rejection need
64-bit RISC-V with vector or accelerator option	ISA boundary, custom extension policy, programming model, compiler/runtime path, library support, and software compatibility.	Correctness, toolchain support, generated-code evidence, portability checks, and rejection of unsupported software semantics.
Compute subsystem	CPU, fixed-function accelerator, reconfigurable spatial array (e.g., CGRA or Reconfigurable Dataflow Architecture/RDA), or Processing-in-Memory (PIM) data-centric block; includes SIMT state (TLP, register pressure), memory hierarchy, and SoC integration.	Design-space comparison and rejection of candidates that only win by moving cost, bandwidth, energy, or complexity elsewhere, or those lacking hardware security isolation (e.g., timing channels, GLIFT).
3 W, 3 nm-class low-power envelope	Power, voltage/frequency, thermal, area, EDA constraints, and dynamic power management interfaces (e.g., Dynamic Voltage and Frequency Scaling or DVFS, power domains, thermal sensors) for the OS/firmware.	Power-model provenance, physical RTL limits (IR drop, electromigration, timing closure), sensitivity analysis, and explicit rejection gates for DRAM read-disturbance faults (e.g., RowHammer, worsened when a tight power budget reduces refresh) and runtime throttling failures.
Design-space report with evidence and rejected alternatives	Alternatives, Pareto fronts (designs that nothing else beats on every objective at once), assumptions, uncertainty, verification plan, rejected or failed candidates, and human decision points.	Evidence ledger: connected measurements, assumptions, checks, rejections, coverage, reproducible artifacts, and explicit rejection authority before higher commitment.

The table gives the first concrete version of a loop contract. It says what state must be represented, which parts of the design space can be exposed to methods, what feedback matters, and what can reject a result before commitment. Later chapters make each field more precise, but the contract begins here. A weak lighthouse answer can be rejected immediately if it cannot say which workload slice, software contract, evidence level, and human decision owner its design claim depends on. The terms below are signposts for that later precision; the only idea needed now is simple. A loop should say what it may change (the part of the design space the architect exposes to methods, narrower than the full space), what evidence it owes, what can reject it, and who decides.

Loop contract. A loop contract is the explicit agreement a design loop makes visible before a method acts: the task, represented state, permitted actions, architecture environment, feedback budget, evidence standard, rejection authority, and human decision owner.

Engineer move: Fill in an AI-assisted loop contract

A compact request becomes an AI-assisted loop contract by answering seven questions. For the lighthouse prompt the answers are the entries in Table 1.1; for a new request, fill them in.

1. **Workload or scenario.** Which workload slice, input distribution, and quality-of-service target is the AI-assisted loop actually optimizing?
2. **Interface or software contract.** Which ISA, programming model, compiler/runtime path, and compatibility must the method respect?
3. **Legal action space.** Which design choices may the AI method change, and which are fixed or deferred to a later turn?
4. **Environment or tool path.** Through which simulator, flow, or harness does the AI loop act and observe?
5. **Feedback budget.** How many evaluations, at what cost, latency, and fidelity, can the AI method afford? For example, the loop must explicitly distinguish between a millisecond analytical proxy and a three-day place-and-route (P&R) run.
6. **Evidence and rejection gate.** What evidence does an AI claim owe, and what can reject it before human commitment?
7. **Human commitment boundary.** What is the strongest claim the evidence licenses, and who owns the final architectural commitment?

A request that cannot answer these is not yet a credible AI-assisted loop; it is a wish, and the architect can reject it on that basis.

Design-loop card. A design-loop card is a compact review artifact that records the intent, task, design space, representation, environment, method role, feedback budget, evidence, rejected and failed candidates, rejection authority, commitment boundary, and human decision for a loop.

The rest of the vocabulary in the loop contract is a signpost here, not a full definition stack. Later chapters give sharper treatment to evidence ledgers, feedback budgets, rejection authority, and commitment boundaries. For now, the point is that those terms turn the lighthouse prompt from a request for an answer into a reviewable contract for a loop.

Design principle: Make AI synthesis obligations explicit

System synthesis is credible only when the AI-assisted loop exposes the obligations that make a result reviewable: what state it represents, which actions are bounded, which environment returns feedback, which AI method role acts, what feedback budget it spends, what evidence ledger it preserves, and what can reject the claim before human commitment.

These obligations are the chapter-level preview of the design-loop card, not a competing schema. Represented state maps to the card's intent, task, and representation fields; bounded action maps to design space and method role; architecture environment and feedback budget map directly; evidence ledger maps to evidence and negative traces;

and rejection authority carries the commitment boundary and human decision fields that keep the loop accountable.

No single model can make these obligations disappear. The table is deliberately compressed; each row expands into many implementation and evidence questions. The “3 W, 3 nm-class” row, for example, reaches down into RTL, synthesis, floorplanning, timing, IR drop, leakage, thermal behavior, and signoff. The “RISC-V with vector or accelerator option” row reaches sideways into compilers, runtimes, libraries, generated code, and portability. Architecture development therefore means proposing artifacts, predicting consequences, optimizing under constraints, and rejecting weak evidence across changing fidelity levels. This is why the moonshot is a computer architecture problem rather than prompt engineering. The loop has to carry architectural state across the stack.

1.5 Architecture Development Spans Three Roles

These obligations of the lighthouse prompt also clarify what the word *development* has to cover. Architecture development is not one AI task. It is a loop in which different roles produce different kinds of architectural work.

This chapter emphasizes generation, prediction, and optimization because they are the development roles most easily mistaken for the whole story. Later chapters add critique, repair, verification, explanation, and coordination as the checking and governance roles that keep automated participation credible.

Generation proposes objects the architect can inspect: an ISA extension, microarchitecture sketch, accelerator interface, memory hierarchy option, RTL fragment, testbench, benchmark harness, or design-space report. Prediction estimates what those objects would do before every expensive evaluation: latency, energy, memory traffic, timing risk, compiler support, verification burden, or deployment behavior. Optimization searches among alternatives: which cache shape, vector width, dataflow, voltage/frequency policy, chiplet partition, or compiler schedule best satisfies the objective and constraints.

The lighthouse prompt needs all three. A generator might propose a vector extension for XR kernels, but prediction has to estimate whether the extension actually improves latency and energy under the mobile power envelope. Optimization then has to compare that extension against an accelerator, a tighter memory hierarchy, or a software/runtime change. None of those steps is credible without rejection: a compiler may not generate valid code, a power model may be out of support, a timing check may fail, or a workload slice may not represent the intended XR behavior. Crucially, when an EDA tool rejects a candidate, it does not return a simple scalar loss; it returns structural feedback like thousands of timing violation paths, which the loop must parse and use to guide the next generation.

Those roles overlap, but none is sufficient alone. The process closely parallels machine learning training: generation proposes a candidate (like a forward pass), prediction

evaluates its quality (like a loss function), and optimization searches for a better candidate (like backpropagation). At the center, systems are synthesized in a closed loop. Generated candidates are predicted, optimized, checked, rejected, and revised under explicit evidence standards.

Figure 1.5 visualizes this distinction. Its purpose is not to introduce three disconnected topics. It shows why an architecture loop needs all three roles at once: generation without prediction produces unsupported artifacts, prediction without optimization does not search the space, and optimization without generation and evidence can overfit a proxy. Chapter 6 returns to the methods in detail; here they establish that Architecture 2.0 is about the loop among these roles, not only about producing candidate designs.

Overfitting: In machine learning, fitting the noise and idiosyncrasies of a training sample so that performance fails to generalize to new data. An optimizer can *overfit a proxy* in the same way, raising the proxy's measured score by exploiting its quirks rather than improving the true objective.

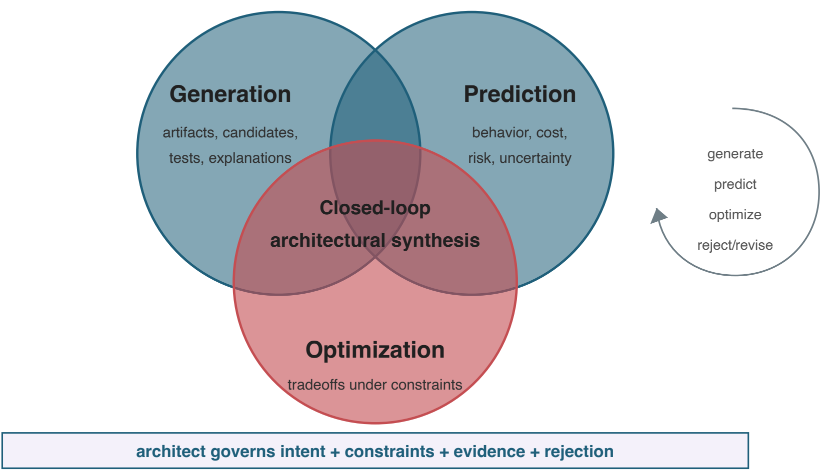


Figure 1.5: **Architecture development is broader than generation:** Generation proposes artifacts and candidates, prediction estimates behavior and risk, and optimization searches tradeoffs under constraints. Architecture 2.0 is concerned with the closed loop in the middle, governed by evidence, rejection, and human architectural judgment.

The figure shifts the reader from a list of AI capabilities to a loop contract: each method role is useful only when its output is checked by the other roles and by evidence strong enough for the next commitment boundary.

1.6 Efficiency Claims Need Rejectable Loops

Naming the roles inside the loop does not yet say what the loop is optimizing for. Architecture still cares about efficiency. The discipline turns scarce resources into useful work through durable hardware/software interfaces. But the loop optimizes for credible efficiency claims, not raw artifact output. A design that is faster but consumes too much power, is impossible to verify, or depends on fragile software assumptions has not really solved the architectural problem. If Architecture 2.0 only made architects produce more artifacts, it would not be enough. The goal is to produce better, more credible, and more efficient systems under rising complexity.

The hard shift is not from performance to a single new metric called power. It is that efficiency itself is becoming more multidimensional. Classical computer architecture made performance quantitative, but it also treated cost and power as first-class constraints (Hennessy and Patterson, 2017). Dennard-style scaling once made it easier to improve performance while keeping power density manageable (Dennard et al., 1974). As that story weakened, dark silicon and the limits of multicore scaling pushed the field toward specialization (Borkar and Chien, 2011; Esmaeilzadeh et al., 2011; Hennessy and Patterson, 2019). Data-movement energy made arithmetic alone an insufficient efficiency story (Horowitz, 2014). Warehouse-scale operation expanded the boundary to power delivery, utilization, networking, operations, and total cost of ownership (Barroso et al., 2019). Sustainability adds another layer because carbon depends on operational energy, hardware manufacturing and infrastructure, utilization, geography, and lifetime (Gupta et al., 2021). For an AI-assisted architecture loop, each efficiency dimension becomes represented state, an evidence requirement, and a possible rejection condition.

Given this multidimensional efficiency, the question is not whether traditional architecture methods suddenly stop working. Many still work extremely well when the workload, abstraction, feedback path, and commitment level are bounded. The harder question is where the classical loop becomes too slow, too implicit, or too expensive to manage the coupled objectives. Architecture 2.0 should be understood as a way to make that boundary explicit: which parts can still be handled by familiar models, scripts, simulation, and expert review, and which parts need more explicit state, tool feedback, evidence ledger entries, rejected alternatives, or AI-assisted search.

To make these boundaries explicit, a credible loop must also record what failed as well as what worked, not only the winners. Just as machine learning training relies on checkpoints to recover from divergent states, an AI-assisted design loop requires checkpointing the design state so it can roll back from failed architectural paths. That record keeps later methods, reviewers, and architects from re-exploring ground the loop already ruled out. Later chapters give it a sharper name and make it part of the evidence ledger.

Modern benchmark suites show this same pressure toward multidimensional efficiency. MLPerf, the community benchmark effort for making machine-learning performance claims reproducible across systems, treats deployment scenario, latency, throughput, accuracy, and power as part of the comparison rather than as one scalar claim (Mattson

et al., 2020). Chapter 2 develops that case with the numbers; the architectural lesson here is that efficiency is not one number. It is a structured claim about useful work under constraints. Useful work must be evaluated against the relevant scarce resource: latency, throughput, energy, power, area, dollar cost, carbon, reliability, verification effort, engineering time, or risk.

A compact way to write the point is that every efficiency claim has a design, workload, scenario, and resource denominator:

$$\mathrm{Eff}_r(d, w, s) = \frac{\mathrm{UsefulWork}(d, w, s)}{\mathrm{Resource}_r(d, w, s)}.$$

Here, d is the design, w is the workload, s is the deployment or evaluation scenario, and r may be time, energy, power, area, dollar cost, carbon, validation effort, or another scarce resource. In an AI-assisted design loop, d, w, s, and r are represented state, not only notation. Changing any one changes what evidence can support the claim and what should reject it. The equation is simple on purpose. It prevents the loop from treating a faster design as efficient if the useful work, scenario, or resource denominator has quietly changed, and because the denominator is stated openly, it also lets a second architect compare two competing designs on equal footing rather than trusting whichever number looks larger.

Table 1.2 summarizes the dimensions that this chapter will treat as part of efficiency. The rows are not separate goals to optimize independently. They are coupled obligations that a design loop must represent and test.

Table 1.2: **Efficiency is becoming multidimensional:** Performance, power, reliability, scalability, sustainability, and evidence cost are increasingly coupled. The architecture question is which parts traditional loops can still handle and which parts need more explicit state, feedback, and rejection.

Dimension	Efficiency question	Why it complicates the loop
Performance	How much useful work is delivered per unit time, latency budget, or service-level target?	The answer depends on workload selection, scenario, software stack, and whether the measured behavior matches the deployment claim.
Power and energy	How much useful work is delivered per watt, joule, thermal budget, or battery envelope?	The loop must model activity, data movement, voltage/frequency choices, thermal constraints, and fidelity gaps between estimates and signoff.

Dimension	Efficiency question	Why it complicates the loop
Reliability and correctness	How much useful work survives faults, corner cases, nondeterminism, and validation?	A faster candidate is not efficient if it spends its savings on fragility, debug burden, or invalid software and hardware assumptions.
Scalability and cost	How much useful work is delivered per dollar, rack, network hop, operator action, or unit of capacity?	Local wins can shift cost to memory, network, power delivery, utilization, operations, or total cost of ownership.
Sustainability	How much useful work is delivered per unit of operational and embodied environmental footprint?	Carbon depends on hardware lifetime, manufacturing, energy mix, utilization, and where and when computation runs.
Evidence and engineering effort	How much credible evidence is obtained per simulation, experiment, verification run, or engineer-hour?	A loop that generates more candidates can still be inefficient if it consumes scarce feedback, hides failures, or produces evidence that cannot reject outputs.

The word efficient should therefore be read broadly. A candidate that improves simulated performance while increasing verification burden may not be efficient. A candidate that reduces energy but requires fragile software assumptions may not be efficient. A candidate that looks good under a proxy metric but fails under a more faithful workload may not be efficient. Architecture 2.0 should treat efficiency as a loop property, not only an artifact property.

The lighthouse prompt makes this concrete. The requested subsystem must support a workload class, meet a power envelope, fit a technology assumption, interact with software, and produce evidence. The design loop must reason about tradeoffs among energy, latency, memory traffic, programmability, verification, and deployment risk. A single scalar objective may be useful inside the loop, but it cannot be the whole architectural judgment.

1.7 Boundaries of the Argument

Having set credible efficiency claims as the loop-level target, it is worth being equally clear about what this book is not trying to do. The goal is not to produce a paper catalog. The field is moving too quickly for a catalog to remain useful for long. The goal is to give readers a framework that can organize current work and still be useful as models, tools, and benchmarks change.

Nor is the goal to make a product forecast. The book does not claim that a particular model, tool harness, simulator, EDA flow, or benchmark will define the field. Those will

evolve. The durable question is what must be represented, measured, checked, rejected, and decided.

Nor is this a tool manual. Tools matter deeply, but the focus is the architecture of the design loop rather than installation instructions or process recipes. Appendix A gives a compact bootstrap path. Appendix B gives the design-loop card and rubric.

It is also not a claim that AI systems replace architects. The opposite is closer to the book's argument. As design loops become more automated, the architect's responsibility moves upward. The architect must frame the task, choose representations, define environments, set evidence standards, inspect rejected and failed candidates, maintain rejection authority, and own the final commitment. Chapter 4 gives those records a sharper name and shows why they matter as loop state.

Failure mode: What not to claim

Architecture 2.0 should not be read as push-button chip design, replacement of architects, or a claim that today's generative methods define the field. The defensible claim is narrower and stronger. Architecture can move toward synthesizing systems under governance when design loops expose state, actions, feedback, evidence, rejection authority, and architect-owned commitment.

The rest of the book follows the loop exposed by the moonshot. Chapter 2 explains why the classical architecture loop strains as specialization, chiplets, software velocity, data movement, EDA constraints, reliability expectations, sustainability pressure, and verification burden grow together. Chapter 3 names the ontology of the new loop. Chapter 4 asks what architecture data must represent, including the world models introduced by the ontology. Chapter 5 turns tools into environments with actions, observations, feedback, and constraints. Chapter 6 separates generation, prediction, optimization, critique, and repair as method roles. Chapter 7 defines feedback, verification, and trust. Chapter 8 runs one loop end to end on the lighthouse prompt. Chapter 9 compares loop patterns across the stack. Chapter 10 returns to what the architect owns, then turns the framework into long-horizon challenge tasks and a research agenda. The appendices then equip the reader: a bootstrap path for a first loop, the design-loop card and review rubric, and a loop-role resource catalog with living links.

World model: A predictive model of an environment used to anticipate the consequences of actions. The seminal sense is a *learned* latent model (Ha and Schmidhuber, 2018); this book uses the term more broadly, covering simulators, surrogates, cost models, and design rules that encode what the loop expects to happen.

1.8 Conclusion

This chapter asked what it would take for AI to help an architect turn intent into a system design that others can check, compare, and reject, rather than settle for a plausible answer. The moonshot is not a faster generator. It is a shift in the unit of work, from the

artifact to the loop that produces, evaluates, rejects, and justifies it. A prompt becomes an architecture claim only when the loop makes its state explicit, what it saw, what actions it allowed, what alternatives it rejected, what evidence supports the result, and who owns the commitment if the claim is wrong.

That shift is forced by efficiency, now read as a coupled budget across performance, energy, power delivery, reliability, sustainability, cost, verification burden, and time to credible evidence, rather than a single number. When plausible artifacts are cheap, the scarce work is the burden of proof, and the rate at which credible choices can be proposed, checked, rejected, and justified becomes the real bottleneck. Designing that loop is the discipline the book names.

None of this removes the architect. As the loop automates, responsibility moves upward. The architect frames the task, chooses representations, defines environments, sets evidence standards, inspects what failed, keeps the authority to reject, and owns the final commitment. Architecture 2.0 is the modest but demanding claim that architecture can synthesize systems under governance, once its design loops expose state, actions, feedback, evidence, rejection authority, and human-owned commitment.

1.9 Open Research Questions

These open research questions represent unsettled, forward-looking directions that push the boundaries of current architecture research. Carry them as conceptual challenges to consider as subsequent chapters formalize the design loop.

1. **Formalizing Reproducibility and Provenance in Non-Deterministic AI Generators.** How can the computer architecture community construct a formal framework for verifying hardware designs synthesized by stochastic, non-stationary AI models? This requires establishing cryptographic or formal bounds on provenance such that a generated microarchitectural claim remains rigorously reproducible, reviewable, and defensible even when the underlying foundational models shift or their weights remain proprietary (see the discussion on “Why the Prompt Spans the Stack” in Section 1.4).

2. **Adversarial Red-Teaming[2] for Autonomous Hardware Generation Loops.** How do we design a systematic theory of adversarial workloads and edge-case constraints designed exclusively to stress-test the validation mechanisms of AI-assisted design loops? Addressing this demands the creation of novel, mathematically grounded “red-team” benchmarks that intentionally induce compounding errors across the hardware-software stack, forcing agents to propose plausible yet physically or logically flawed systems to evaluate the robustness of their rejection authorities (see the discussion on “Efficiency Claims Need Rejectable Loops” in Section 1.6).

3. **Establishing a Unified Currency for Multi-Fidelity Feedback Budgets.** As AI agents query a mix of rapid surrogates and highly accurate but computationally expensive cycle-accurate simulators, how do we formalize an economic and information-

[2] Red-teaming in AI involves proactively probing a system to discover vulnerabilities, biases, or unexpected failure modes.

theoretic model of "evidence cost"? A thesis in this space must derive a rigorous optimization framework that dynamically balances financial cost, latency, and predictive uncertainty to establish a standardized currency for the feedback budget in automated design loops (see the discussion on "Architecture Development Spans Three Roles" in Section 1.5).

4. **Federated Ontologies for Architectural Rejection and Failure Modes.** How can the community synthesize a unified, machine-readable taxonomy of failure modes that structurally encodes *why* an AI-generated architecture candidate was rejected by a simulator, compiler, or physical design tool? Formulating this shared infrastructure would prevent independent AI-assisted systems from repeating foundational mistakes, requiring research at the intersection of knowledge representation, compiler feedback loops, and hardware design verification (see the discussion on "From Architecture 1.0 to Architecture 2.0" in Section 1.2).

What to carry forward

- **Reader test:** For any AI architecture result, can you say what state was represented, what method role acted, what evidence was preserved, what could reject it, and where the commitment boundary stays human-owned?
- **Up next:** Cheap generation makes that reading habit matter more, not less; the next chapter turns it into a pressure test by asking why the classical loop no longer scales.

Chapter 2

Why Classical Architecture Loops Strain

> *"Feedback is the control of a system by reinserting into the system the results of its performance. If these results are merely used as numerical data for the criticism of the system and its regulation, we have the simple feedback of the control engineers."*
>
> — Norbert Wiener, *Cybernetics* (1948)

To understand how to design that loop, we must first make the pressure it faces explicit.

The crux

When architecture choices grow faster than trusted feedback, what must the loop record so AI helps rather than just adding more candidates?

Author's Note: Norbert Wiener, the founding father of cybernetics, established that true control requires feedback. In our context, this means that computer architecture is fundamentally a feedback loop, and our current crisis is simply that manual loops have become too slow to function effectively.

Chapter 1 named Architecture 2.0 as a discipline for designing the design loop itself. It then made the claim concrete with a compact lighthouse prompt: design a low-power, RISC-V-based compute subsystem for real-time mobile XR under a 3 W, 3 nm-class mobile envelope, and return a design-space report with evidence and rejected alternatives. The prompt is intentionally small. The architecture state it implies is not.

This chapter explains why that state cannot be handled by merely asking a larger model to produce a larger answer. Computer architects already use models, simulators, benchmarks, profilers, spreadsheets, compilers, RTL flows, EDA tools, and expert reviews. The problem is not the absence of tools. The problem is that the design loop that coordinates those tools is under pressure. The space to search, the constraints to satisfy, and the evidence required to trust a result now grow faster than manual coordination, review, and verification capacity.

The claim is not that the old loop is obsolete. It is that the old loop must become a first-class object of design alongside the artifact it produces.

Scissors gap. The scissors gap is the widening gap between the rate at which design choices, constraints, and evidence demands expand and the rate at which a team can obtain trusted feedback, reject weak candidates, and commit responsibly.

The gap opens when the number of plausible actions, constraints, feedback sources, and evidence requirements grows faster than the loop's ability to evaluate, reject, revise, and commit architecture candidates credibly. That is the visible failure mode. Trusted feedback cannot keep up with the choices and evidence demands the loop has created. For automation, the scissors gap sets the safe operating boundary. The loop can delegate only as much search, prediction, and synthesis as it can review and reject.

Architect's checkpoint: The Delegation Gate

An automated method can only be delegated tasks where the loop has explicit, trusted rejection checks. If you cannot reject a bad candidate, you cannot safely delegate its generation.

The practical diagnostic is therefore not "where can we add a model?" It is "which part of the loop cannot keep up?" A loop may be representation-bound, action-bound, feedback-bound, rejection-bound, or commitment-bound. Each names a different part of the loop that saturates first, and each shows a different symptom, needs a different record to confirm, and calls for a different first fix (Table 2.1).

Table 2.1: **A diagnostic for which part of the loop cannot keep up:** each bound shows a different symptom, needs a different record to confirm, and has a different first fix.

Bound	Symptom	Lighthouse example	Record to inspect	First fix
Representation-bound	The loop cannot encode the state that decides the outcome, so it proposes candidates the environment later rejects.	It ranks XR compute organizations without representing memory-movement cost, so a later estimate overturns the proxy order.	The state schema: which fields the loop can read and write.	Add the missing fields (workload, interface, memory-traffic) to the representation.
Action-bound	The legal action set is too narrow, or too loose, to reach the candidate that would clear the gates.	The loop can retune parameters but cannot change the accelerator interface, so it never reaches the organization that would meet both the real-time deadline and the power envelope.	The action schema: which moves are legal, and recorded.	Widen or tighten the legal action set and record it in the loop contract.
Feedback-bound	Trusted feedback is too scarce or too expensive to keep up with the candidates generated.	Only a few simulation-stage estimates are affordable, so most candidates are dropped or committed unjudged.	The feedback budget and each check's cost and fidelity.	Add cheaper proxies and a ladder of feedback fidelities so scarce feedback is spent where it changes a decision.
Rejection-bound	The loop generates faster than it can reject, so weak candidates survive to expensive stages.	Lacking a gate, an accelerator that leads a cheap proxy ranking is not rejected until an expensive check exposes its data-movement cost.	The rejection gates and the negative-trace log.	Add explicit gates and record every rejection with its reason.

Bound	Symptom	Lighthouse example	Record to inspect	First fix
Commitment-bound	The loop cannot say what evidence licenses which commitment, so it over- or under-commits.	A candidate clears both gates, but nothing states whether that authorizes more exploration or an implementation commitment.	The commitment boundary and the evidence level attached to it.	Set a commitment level the current evidence supports, and name what would raise it.

Section 9.10 turns the rejection bound into a quantitative limit, but the pressure starts here. A loop that can generate faster than it can reject only widens the gap.

Because this failure mode is the central problem, the remainder of this chapter is not a general technology-trend survey. The pressures collapse into two fundamental limits: the Physical Wall (specialization, chiplets, memory movement, EDA) and the Verification Wall (software velocity, verification burden). They matter here specifically because they create that common failure mode. The fundamental economic logic of the loop is: generate cheap, reject expensive. The architecture team can imagine more candidates than it can evaluate, reject, and justify. Architecture 2.0 begins by making that failure mode explicit.

Read each section as a pressure test on one part of the loop. Cadence exposes gates and commitment policy. Architecture levers expand the state the loop must carry. Specialization and chiplets (multiple smaller dies integrated in one package) multiply actions. Software drift changes the workload contract. Physical constraints create early rejection conditions. Engineering cost makes feedback scarce. Generic AI assumptions fail because architecture work is not a cheap-label pipeline. Together, those pressures explain why the loop becomes a first-class architectural object alongside the artifact.

What this chapter gives you

After this chapter you can use the scissors gap to audit whether an AI-assisted architecture loop has enough trusted feedback to reject and commit responsibly. That means you can:

- recognize the scissors gap between choices and trusted feedback in a real architecture setting;
- explain why the bottleneck is trusted feedback, not idea generation;
- diagnose whether a loop is limited by representation, action validity, feedback, rejection, or commitment;
- point to the design and verification costs that make high-fidelity feedback scarce;
- connect each pressure source to loop state that can no longer remain implicit;
- identify where generic AI assumptions break at architecture boundaries;
- distinguish bounded AI roles from architect-owned commitment decisions.

2.1 Classical Loops Already Use Feedback

That first-class architectural object did not arrive out of nowhere; computer architecture has always been a loop. A typical loop begins with an intent: improve latency, reduce energy, raise throughput, support a workload, or fit a system into a power and cost envelope. The architect then chooses an abstraction, builds or selects a model, runs an analysis or simulation, studies the result, revises the design, and repeats. Eventually the work crosses into implementation, validation, verification, and signoff. That basic pattern is visible in textbook architecture practice and in industrial design loops (Hennessy and Patterson, 2017).

A traditional SPEC CPU-style study makes the loop concrete. An architect might choose a subset of SPEC CPU workloads, propose a cache hierarchy or branch predictor change, run a simulator or performance model, inspect IPC, miss rates, branch-misprediction rates, area and power proxies, reject candidates that help one workload while hurting others, and repeat. Human judgment enters repeatedly: selecting workloads, deciding which proxy is credible, noticing unexpected behavior, choosing when a candidate is worth deeper analysis, and deciding which risk to accept. SPEC CPU 2017, for example, was designed as an industry-standardized suite for compute-intensive performance, stressing processor, memory subsystem, and compiler behavior (Standard Performance Evaluation Corporation, 2017). The important point is not the specific suite version. It is the loop discipline: a bounded workload set, an explicit model, comparable metrics, rejection of weak candidates, and expert judgment about whether the evidence is strong enough.

Chapter 1 made the Architecture 1.0 to Architecture 2.0 loop shift visible in Figure 1.1. The point here is why that shift becomes necessary. Classical architecture loops already have intent, models, candidates, tool runs, and expert review. They strain when the state, action boundaries, evidence, rejection, and decision authority become too large to remain mostly implicit.

For the lighthouse prompt, the difference is not that Architecture 2.0 invents review discipline. The difference is what the loop must record before a result can be trusted. Table 2.2 summarizes that delta.

Table 2.2: **Architecture 2.0 makes review state inspectable:** Disciplined architects already review results. The new requirement is that the loop records the state, feedback, rejection, and commitment information needed for bounded methods to participate without hiding the architectural judgment.

Classical review might record	Architecture 2.0 additionally requires
Candidate design and measured result.	The represented task, workload slice, assumptions, and candidate provenance.
Tool output or benchmark score.	Tool version, heuristics, random seeds, feedback fidelity, uncertainty, cost, and what the feedback is allowed to reject.

Classical review might record	Architecture 2.0 additionally requires
Expert judgment about whether the result looks plausible.	Explicit rejection authority, failed or rejected candidates, and the commitment level the evidence supports.

This loop is powerful because it does not require perfect automation. It combines formal models, approximations, domain knowledge, and review. It also rests on an unstated balance: the number of choices, the cost of evaluation, and the evidence needed to make progress must remain within the capacity of the team and tools. When that balance holds, the loop works. When it breaks, the team may still generate ideas, but it cannot evaluate and reject them fast enough to make credible progress. The loop contract of Chapter 1 is that balance written down and made visible.

2.2 Cadence and Gates Manage Risk

One way industrial practice has long held that contract together is by treating cadence itself as a loop policy. It limits which action class may change at a commitment boundary and defines the evidence needed to advance. Intel's tick-tock model is the cleanest familiar example. A "tick" moved a known microarchitecture to a new process technology; a "tock" introduced a new microarchitecture on a more mature process. The point was not that product development was literally two steps. The point was risk isolation. Avoid changing every hard thing at once, preserve a cadence, and let evidence from one step inform the next. When node transitions lengthened, Intel described a move toward process-architecture-optimization, explicitly using longer-lived 14 nm and 10 nm process technologies while further optimizing products and processes to maintain product cadence (Intel Corporation, 2016).

That history is useful for Architecture 2.0 because it shows that a design loop is not only a sequence of tools. It is also a policy for what is allowed to change, which evidence is strong enough to advance commitment, and how the organization reacts when feedback latency changes. Tick-tock separated process risk from microarchitecture risk. Process-architecture-optimization added an explicit optimization phase when process shrinks no longer arrived on the old schedule. Instead of every generation requiring both a new process step and a new architecture step, the loop could spend another cycle improving products, libraries, physical implementation, frequency, power, and yield on a known process. In other words, the cadence changed because the feedback and commitment costs of the physical process changed. Architecture 2.0 generalizes the same lesson. If AI methods increase the rate at which candidates are proposed, the loop must become stricter about change scope, evidence gates, rejection authority, and what kind of optimization is being performed.

Adjacent systems practices offer compatible lessons. EDA timing closure shows that a late tool can reject an early abstraction. The source-backed examples later in this

chapter make the pattern concrete. Autotuning treats measurements as samples from a costly space, and benchmark governance depends on maintained rules and comparability contracts. A loop without observability, a decision policy, and escalation gates is not a credible loop. These analogies should not displace computer architecture. They help name the reusable loop properties architects already care about: cadence, state, feedback, gates, rejection, and commitment.

Autotuning: The process of automatically searching for optimal program or system parameters.

This is not the first time architecture has had to redesign its loop. Read the historical rows in Table 2.3 as examples of explicit action schemas, environment access, workload records, and rejection gates, not as a history survey. Each row names one kind of state, rule, or rejection path that would have to be exposed before an automated loop could help without weakening the evidence standard. The point is not nostalgia. The point is that many ideas that once looked unmanageable became tractable only after the field made some part of the loop explicit: the interface, the rules, the workload, the tool contract, the evidence gate, or the software path.

Table 2.3: **Architecture progress often comes from redesigning the loop:** Historical shifts became durable when they exposed a representation, interface, tool path, benchmark, or evidence gate that let the community coordinate work.

Shift	What the loop made explicit	Architecture 2.0 lesson
System/360 compatibility	A stable ISA contract separated architecture from implementation across a product family (Amdahl et al., 1964).	Architecture is a durable interface and commitment policy, not only a circuit or microarchitecture.
Mead–Conway VLSI and MOSIS	Design rules, layout abstractions, and fabrication access turned custom-chip design into a shareable and reusable loop (Mead and Conway, 1980; USC Information Sciences Institute, 2025).	Representation and access to feedback can change who can participate in architecture work.
RISC	Workload, compiler, VLSI, and the quantitative Iron Law of Performance became the explicit architectural argument (Patterson and Ditzel, 1980).	Evidence can reject attractive complexity when the full loop cost is visible.
SPEC-style benchmarking	Workload selection, run rules, reporting conventions, and comparability became community infrastructure (Standard Performance Evaluation Corporation, 2017).	Benchmarks are loop governance, not just input programs.
Logic synthesis and timing closure	HDL, libraries, constraints, and timing reports gave downstream tools authority to reject upstream choices (De Micheli, 1994).	Implementation feedback belongs in the architecture loop before commitment.

Shift	What the loop made explicit	Architecture 2.0 lesson
CUDA-style GPU programming	Kernels, thread hierarchies, memory spaces, libraries, and toolchains made specialized hardware programmable (Nickolls et al., 2008).	Specialized hardware succeeds only when the software loop is designed with it.

These examples should make the Architecture 2.0 claim less exotic. The field has repeatedly advanced by turning tacit craft into explicit loop structure. The new challenge is that AI methods can propose, search, summarize, and optimize at a scale that makes the loop state itself the bottleneck.

2.3 Architecture Levers Add State

That loop state did not appear all at once. It accumulated as architecture advanced by adding levers. For decades, technology scaling made the same basic design style better by providing smaller, faster, cheaper, and more energy-efficient transistors. Dennard scaling gave architects a favorable energy story as devices shrank (Dennard et al., 1974). As that story weakened, the field leaned harder on microarchitecture, instruction-level parallelism, caches, speculation, vector units, multicore, accelerators, specialization, and system-level optimization. The result is not a simple failure narrative. It is an accumulation of levers.

The accumulation matters because each lever creates both opportunity and obligation. Better microarchitecture adds policies and corner cases. Multicore adds coherence, synchronization, memory ordering, and workload partitioning. Specialization improves efficiency when the workload and software stack are understood, but it adds interfaces, data movement, programmability, and verification burden. Chiplets and heterogeneous integration promise modularity and scaling beyond a monolithic die, but they add partitioning, die-to-die interfaces, package-level constraints, test, yield, thermal coupling, and supply-chain questions.

Figure 2.1 summarizes the first part of the point. The field keeps adding levers because efficiency still matters. But the same moves that recover efficiency also increase the burden of representing the design state and producing trusted feedback.

While Figure 2.1 illustrates a broad accumulation of design levers, the end of Dennard-style scaling and the limits of multicore scaling specifically explain why specialization became so central (Borkar and Chien, 2011; Esmaeilzadeh et al., 2011). Hennessy and Patterson's Turing Award lecture framed this moment as a new golden age for architecture, driven by domain-specific hardware/software co-design, open architectures, and agile hardware development (Hennessy and Patterson, 2019). Architecture 2.0 should be read in that lineage, but with an explicit connection to the RISC revolution that preceded it. RISC was not just a simplified instruction set; it was the realization

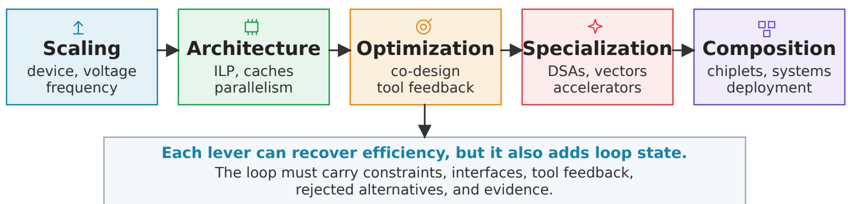


Figure 2.1: **Architecture levers add state:** Scaling, microarchitecture, optimization, specialization, and composition create new opportunities for efficiency, but each also adds constraints, interfaces, tool feedback, rejected alternatives, and evidence that the loop must carry.

that making an architectural claim required a compiler, a benchmark, and a simulator. Architecture 1.0 gave us the tools to measure IPC and execution time, anchoring the field to the Iron Law of Performance (Time = $\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$).

Architecture 2.0 does not break the Iron Law; it uses the AI loop as an active co-designer to negotiate across the hardware-software boundary (compiler, ISA, microarchitecture), much like the early RISC teams did. The loop-level analogue is not the Iron Law but Amdahl's. Chapter 9 shows that a design loop's throughput is set by how cheaply it can reject, not by how many candidates it proposes, and derives that as a re-coordinatization of Amdahl's law with trusted rejection as the serial fraction. Architecture 2.0 elevates the quantitative method from comparing candidate artifacts to designing the data, feedback, and evidence loops that make a larger, more coupled design space tractable. In that sense, the loop becomes a first-class architecture object: something to represent, instrument, test, reject, and improve.

That is the architectural consequence for Architecture 2.0. The new golden age gives the field more architectural levers; Architecture 2.0 asks how to govern the loop that uses them. Without that layer, AI assistance can only make the design space larger. With it, AI methods can be assigned bounded roles in search, evidence, rejection, and revision.

2.4 Specialization and Chiplets Expand Search

Of those bounded roles, search is the one most immediately stressed by the shift to Architecture 2.0, driven largely by the rise of specialization. Specialization is attractive because efficiency claims are multidimensional, the point Chapter 1 made explicit. The binding constraint differs across scales, from a battery- and thermally-limited mobile part to a warehouse-scale system bounded by power delivery and total cost of ownership (TCO) (Barroso et al., 2019). What this chapter adds is the consequence for the loop.

Specialization does not just change which metric matters, it multiplies the decisions the loop must evaluate.

Specialization increases the number of architectural decisions because the architect must decide what to specialize, where to specialize it, and how it communicates with the rest of the system. A low-power XR subsystem is not just a choice between CPU and accelerator. It raises questions about vector length, memory hierarchy, local buffers, compression, dataflow, quantization, runtime scheduling, compiler support, sensor streams, display deadlines, thermal behavior, and fallback modes.

Chiplets compound the effect. They make it possible to compose systems from multiple dies and to mix process technologies, IP blocks, and memory technologies. But a chiplet system is not simply a bigger board-level system inside a package. The package changes latency, bandwidth, energy, thermal coupling, test, repair, physical constraints, and business boundaries. Open standards such as Universal Chiplet Interconnect Express (UCIe) aim to make die-to-die integration more composable by standardizing interface layers, protocols, software models, and compliance expectations (UCIe Consortium, 2026). That standardization is valuable, but it also makes the architecture question more explicit: what should be partitioned, through which interface, under which evidence standard?

The combinatorics are easy to understate. The following arithmetic is illustrative, not a measurement. Suppose a team is exploring only a narrow slice of the lighthouse prompt: an accelerator and memory subsystem for one workload family from XRBench (Kwon et al., 2023), a benchmark suite for extended reality systems. Even if it considers five compute organizations, four vector or accelerator interface choices, six memory hierarchy choices, four interconnect choices, three voltage/frequency policies, three compiler/runtime policies, and three verification or fidelity levels, the crude product is already:

$$5 \times 4 \times 6 \times 4 \times 3 \times 3 \times 3$$

That is 12,960 candidate loop states before workload versions, process corners, thermal constraints, reliability cases, and rejected configurations are counted. This number is intentionally conservative. More realistic architecture-adjacent loops quickly become much larger, or much slower, even before final silicon evidence is involved.

The product is also not just a counting problem. If each surviving state needs even a cheap analytical model, a simulator run, a synthesis check, or a human review, the loop is immediately limited by feedback cost and fidelity. Cheap models are essential, but they move the architecture problem rather than removing it. The loop must know when a proxy is good enough, when uncertainty is too wide, and when to escalate to stronger evidence. Chapter 4 turns that intuition into sample-cost and simulation-time representations, Chapter 5 separates feedback regimes by latency, fidelity, and rejection authority, and Chapter 6 returns to this same count in an evidence-gap plot that compares candidate scale with affordable high-fidelity samples.

In the Lighthouse slice, the same count is not search-space trivia. The CPU/accelerator/SoC choice also fixes interfaces, memory paths, compiler/runtime assumptions, physical constraints, and an evidence schedule: which states get cheap proxies, which get simulation, which escalate, and which rejected regions are preserved.

Engineer move: Triage a design-space slice before you simulate it

1. **Situation.** A 12,960-state slice of the design space faces a fixed simulator-hour budget; not every state can be evaluated at full fidelity.
2. **Architecture decision.** Which states get a cheap proxy, which get a cycle-level simulation, which escalate to stronger tools, and which regions to reject outright.
3. **Bound the loop.** Fix the slice, the per-week simulator budget, and the metrics that decide the question (latency, energy, power).
4. **Method role.** Use the AI system as a *searcher and predictor*. Rank states by a cheap proxy and propose the next most informative simulation. It proposes; it does not decide.
5. **Evidence path and escalation.** Proxy-rank all states, simulate only the top candidates with logged provenance, and escalate to synthesis only after a candidate survives the cycle-level check.
6. **Negative trace.** Record rejected regions with their reason (infeasible power, dominated latency) so the loop does not re-explore the same dead ends.
7. **Architect signs off.** A human approves which surviving candidate has evidence strong enough for the next commitment, and owns the schedule risk if the proxy mis-ranked.

Beyond the lighthouse example, Table 2.4 grounds this combinatorial pressure in other concrete architecture settings. The examples are not meant to be a single measurement scale; they show that mapping, design-space exploration (DSE), physical design, and software tuning each impose a different kind of loop burden. Timeloop, an analytical framework for evaluating and mapping deep-neural-network (DNN) accelerators, makes the mapping case especially concrete. For a single convolutional-neural-network (CNN) layer with seven nested loop dimensions, mapped onto a four-level memory hierarchy, the number of legal mappings is astronomically large. Before hitting the raw math, consider what is being multiplied: for every level of the memory hierarchy, we must permute the order of the seven loop dimensions, and for every tensor, we must decide whether it bypasses each memory level. The exact expression matters less than its size. The unconstrained mapspace contains $(7!)^4 \times (2^4)^3$ arrangements before co-factor choices (such as tiling sizes) are even counted. The $(7!)^4$ factor counts loop-order permutations of the seven nested loop dimensions (e.g., batch, output channels, input channels, and spatial dimensions) at each of the four memory levels, and $(2^4)^3$ counts the level-bypass choices that decide which of the three tensors (inputs, weights, outputs) skip any of the four memory levels (Parashar et al., 2019).

Table 2.4: **The scissors gap has source-backed scale anchors:** Mapping, accelerator DSE, floorplanning, and tensor-program tuning all show candidate scale, validity, feedback cost, and evidence standards growing together.

Loop example	Scale anchor	Loop lesson
DNN accelerator mapping	Timeloop exposes loop permutations, factorization choices, and level-bypass alternatives.	Mapping is itself a combinatorial problem; architecture evaluation depends on the mapper and its constraints.
DNN accelerator DSE	MAESTRO, an analytical cost model for estimating DNN dataflow performance, reports 480M candidates, 2.5M valid designs, and 0.17M designs/s (Kwon et al., 2019).	Validity and pruning shape what evidence can be trusted.
TPU-block floorplanning	Learned floorplanning reported a months-to-hours loop compression; later work challenged baselines and reproducibility (Mirhoseini et al., 2021; Cheng et al., 2023b).	High-fidelity tool feedback and rejection authority determine whether generation is credible.
Tensor-program tuning	AutoTVM, a machine learning-based tensor program optimizer, describes tensor-operator search spaces on the order of billions of possible implementations for a single GPU operator (Chen et al., 2018).	The software side of specialization also has a large hardware-dependent loop.

The exact size of any one space is not the point. These examples differ in task, fidelity, and tool chain, but they share the same pressure pattern. Candidate count, validity, feedback cost, and evidence standards grow together. More candidates are useful only if the loop can evaluate, explain, and reject them (a dynamic we will formalize as the scissors gap in Figure 2.5).

2.5 Specialized Hardware Needs a Software Loop

The scissors gap is not solely a hardware problem. Specialization also exposes a software obligation. It is one thing to build an accelerator, vector unit, memory hierarchy, or chiplet partition that looks efficient in isolation. It is another thing to let programmers, compilers, runtimes, libraries, and deployment systems use it without destroying the efficiency claim through data movement, synchronization, code-generation overhead, or maintenance burden.

The historical examples in Table 2.3 show the same pattern. RISC depended on a compiler story. CUDA made GPU specialization useful by making the programming and toolchain loop explicit (Nickolls et al., 2008). The tensor-program row in Table 2.4 pushes the point further. Billion-scale operator search is already a software-side obligation of specialization. Systems such as Halide and MLIR, advanced compiler infrastructures used to define and schedule domain-specific operations, make scheduling, lowering, and intermediate representations central parts of the performance loop (Ragan-Kelley et al., 2017; Lattner et al., 2020).

Field note: The architecture bet the compiler never paid: Itanium

Itanium's EPIC architecture rested on one claim, that a compiler could statically expose enough instruction-level parallelism to feed a very wide machine (Schlansker and Rau, 2000). Written as a design-loop card, that bet is a rejection gate waiting to fire.

- **State seen:** a wide static-issue target whose payoff needs sustained ILP close to the hardware's issue width, running general-purpose code with unpredictable control flow and memory aliasing.
- **Action allowed:** widen the machine and delegate parallelism extraction to the compiler.
- **Evidence required to commit:** compiler-*achieved* ILP measured on representative branchy workloads, not the ILP the machine could retire in principle.
- **Rejection rule:** reject the wide-static bet if achieved ILP stays far below the issue width. The machine was built to issue on the order of six operations per cycle, against sustained integer ILP that real compilers struggled to push past one or two.
- **Commitment owner:** the architects who signed the tapeout.

The rejecting evidence was gatherable before commit by building the compiler, running real control flow, and measuring the ILP it actually extracts. Evaluated that way, the gate fires early and the bet is rejected in favor of a narrower or out-of-order organization. Instead the loop committed silicon on the assumed capability, and the part underdelivered against simpler rivals for a decade. Intel's Larrabee made the same shape of bet a decade later, that software rasterization on many x86 cores would match fixed-function GPUs (Seiler et al., 2008), and Transmeta's Crusoe bet that code-morphing software would let a simple core match x86 (Dehnert et al., 2003). Both rested on a software capability asserted rather than measured.
That is the whole difference. Architecture 1.0 can name this failure in hindsight.
Takeaway. Architecture 2.0 makes the compiler-ILP evidence a gate the bet must clear before commit, so an architecture can no longer be committed on a software capability that was asserted rather than measured.

For the lighthouse prompt, this means that a "64-bit RISC-V compute subsystem" cannot be judged by hardware structure alone. If the answer proposes a vector extension, custom accelerator, memory-local dataflow, or chiplet boundary, the loop must also represent how code reaches that mechanism, what compiler or runtime assumptions are required, which libraries or kernels use it, and which tests reject a design that is efficient only

in a hand-written kernel. The software path is not downstream polish. It is part of the architectural claim. For an AI assistant, a hardware candidate is incomplete unless it also names the software contract and the tests that can reject unsupported semantics.

Lighthouse prompt: RISC-V is a software contract

Context. The ISA phrase fixes more than an instruction encoding. It names the boundary where AI-generated hardware choices become visible to compilers, runtimes, libraries, operating systems, and compatibility tests.
In the Lighthouse prompt. "64-bit RISC-V-based" and "vector-capable CPU, accelerator, or SoC block" make the software path part of the claim. An AI-assisted loop must explain how code reaches the mechanism, what ABI, memory-model, and toolchain assumptions it preserves, and what tests reject unsupported semantics.
Takeaway. The ISA is part of the evidence path for whether the AI-proposed subsystem can actually be used, not a label on the hardware box.

2.6 Software Changes Faster Than Silicon

Even when that software contract is perfectly specified, specialization still depends on stable enough targets. But modern software stacks move quickly. AI models change. Precision formats (e.g., FP8, INT4, block-scale formats[3]) and sparsity patterns (the structured or unstructured zeros in weights or activations that hardware can skip) mutate wildly, meaning the hardware must place bets on data types years in advance. Compiler passes change. Kernel libraries change. Runtimes, serving systems, quantization formats, batching strategies, fleet policies, and benchmark versions change. The hardware design cycle does not move at the same pace.

[3] Block-scale formats share a single exponent across a block of low-precision mantissas to maintain dynamic range with minimal memory footprint.

Generative AI sharpens this mismatch rather than easing it. As models lower the marginal cost of producing code, the volume and churn of software rise rather than fall, a software-era instance of Jevons' paradox. The shift is already visible in practice. By the mid-2020s, AI coding assistants had been adopted at scale. The 2024 Stack Overflow Developer Survey reported that a majority of professional developers were already using AI tools in their workflow (Stack Overflow, 2024). The design target stops being a fixed workload an architect samples once and becomes the output of its own fast, semi-autonomous loop, retrained and regenerated faster than a hardware team can respond. A silicon program measured in years is then committed against a snapshot the software has already moved past. The architect cannot answer this by demanding faster silicon alone, because the fabrication cycle and physical closure set a floor on how short the hardware loop can be.

Jevons' paradox: From nineteenth-century resource economics, the observation that making a resource cheaper to use can raise its total consumption instead of lowering it.

This forces a tradeoff the classical loop could often ignore. Peak efficiency comes from specialization, which is brittle when the workload drifts, while generality preserves

agility at a cost in performance per watt. Under fast software churn, committing to a specialized design is a bet on workload stability, and that bet is increasingly made without evidence that the stability holds. The Architecture 2.0 response is not to abandon specialization but to make the bet explicit. Bound the workload the design serves, measure how fast it is drifting, and specialize only with evidence that the target will hold long enough to repay the silicon. That repayment includes the embodied carbon manufactured into the part up front, an irreversible commitment the holding workload must amortize, so a candidate that is low-energy in operation can still fail a carbon gate if the workload it bet on moves. The artifact remains the commitment target; the loop carries the evidence about whether the workload bet still holds.

The scale of the target workload is not static either. For the loop, the important fact is not the compute-growth curve itself; it is that workload records expire. The compute behind notable AI systems has grown by roughly twenty orders of magnitude in nearly seven decades (Figure 2.2), so the workloads an architecture must serve can shift faster than a silicon program can absorb. A lighthouse loop should version model, compiler, runtime, benchmark, trace, and deployment snapshots, then trigger re-evaluation when drift crosses a commitment boundary.

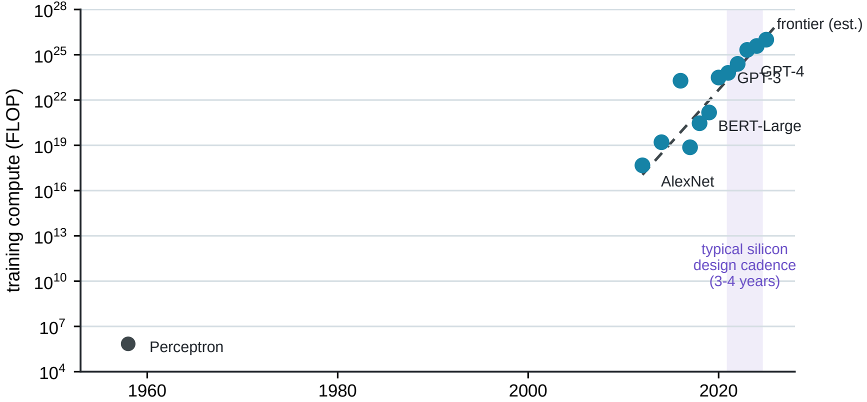


Figure 2.2: **AI compute demand is a fast-moving target:** Training compute for notable AI systems has grown by roughly twenty orders of magnitude since the Perceptron, with a sharp acceleration in the deep-learning era after 2012 and continued frontier growth well past GPT-4 (Sevilla et al., 2022). Points are notable-model training-compute estimates from Epoch AI's database; the dashed line is the deep-learning-era trend and the 2025 frontier point is an order-of-magnitude estimate. The architecture that serves these systems is designed against workloads whose scale and shape can change faster than a silicon program can respond.

This growing need to track moving workloads reflects a broader breakdown of abstractions, a shift that Compiler 2.0 offers as a useful adjacent warning. Amarasinghe's framing

is that compilers originally made hardware disappear for programmers, but multicore processors, vector instructions, accelerators, and heterogeneous systems have pushed more performance burden back onto programmers (Amarasinghe, 2020, 2026). The same pattern appears at the architecture level. Abstractions still matter, but the design loop must now expose more of the workload, software, hardware, and physical state that earlier abstractions could hide.

To manage this rapidly changing software state, the community has had to formalize its workload agreements, making MLPerf a useful example. It was built to create common, reproducible machine-learning system benchmarks across a rapidly changing field (Mattson et al., 2020). MLPerf Inference sharpened the deployment-facing version of that problem. The paper reports more than 100 organizations building ML inference chips, systems spanning at least three orders of magnitude in power and five orders of magnitude in performance, and more than 600 reproducible measurements from 14 organizations in the first submission round (Reddi et al., 2020). The lesson is not only that benchmarks need rules. It is that a benchmark must encode scenarios, latency constraints, accuracy targets, software stacks, and comparability rules before performance numbers mean the same thing across systems (Reddi et al., 2021). That is also the challenge for architecture. A benchmark is not a fixed oracle. It is a maintained agreement about what evidence should count for a class of systems. The loop artifact is a versioned workload packet: traces, scenario constraints, acceptance tests, and expiration or review triggers.

For the lighthouse prompt, the workload is not merely “XR.” It is a moving bundle of sensing, perception, graphics, display, interaction, latency, quality-of-experience, and energy constraints. Even with XRBench (Kwon et al., 2023), a credible architecture loop must still decide which traces, model versions, deadlines, input distributions, and quality targets matter. If the software stack changes faster than the hardware loop can absorb, the design may optimize yesterday’s workload.

2.7 Physical Constraints Move Into Architecture

Drifting workloads are only one source of pressure, pushing down from the software stack; physical limits push up on the loop from below just as hard. Architecture does not sit above physical reality. It is the layer where software intent, hardware mechanisms, and physical constraints become one design problem.

For example, data-movement estimates serve as early physical constraints, providing cheap rejection evidence rather than merely confirming that memory is expensive. Moving data through the memory hierarchy often costs far more energy than arithmetic, and Horowitz’s widely used energy estimates made this point concrete for a generation of architects (Horowitz, 2014). That changes what architecture work means. A design loop cannot only ask which compute block is fastest. It must ask where the data lives, how often it moves, who schedules it, what locality exists, what precision is acceptable, and what the software stack can express.

A useful architecture-level decomposition is

$$E_{\text{system}} = E_{\text{compute}} + E_{\text{memory}} + E_{\text{interconnect}} + E_{\text{control}} + E_{\text{leakage}}.$$

This is not a circuit-level energy model. Use this decomposition as loop state. If any term is unrepresented, stale, or estimated outside its validity regime, the candidate should not cross the next commitment boundary. A candidate that reduces arithmetic but increases memory movement, interconnect traffic, control overhead, or leakage has not necessarily improved the system, a trade-off illustrated by Figure 2.3.

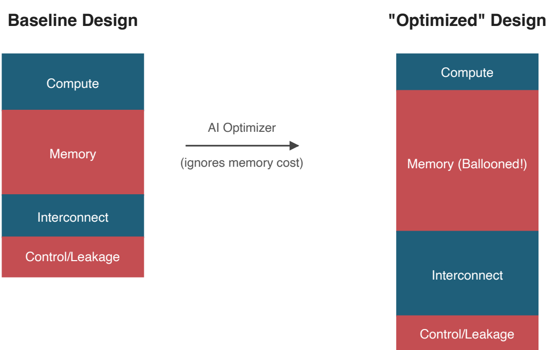


Figure 2.3: **The Waterbed Effect:** Optimizing one term of the physical energy constraint often causes others to balloon. A naive AI optimization might squeeze down compute energy while massively inflating memory and interconnect costs if it fails to represent data movement.

Interconnect costs have the same character as memory movement. On-chip networks, package links, memory interfaces, collectives, and host-device protocols define what a design can actually sustain. EDA and physical-design constraints also move upward. Timing, placement, routing, IR drop, thermal behavior, power delivery networks (PDN), leakage, signoff, and test are not late implementation details when they can overturn an architectural choice. Power and thermal density, for instance, can flag a dense accelerator mapping as risky early, and IR-drop and thermal signoff confirm or clear it later, once a placed netlist and current profile exist. Furthermore, because EDA tools rely on high-variance heuristics, a failed timing closure might just be a bad random seed, not a bad architecture. For a design loop, this means that a simulator score or model prediction is not enough. The loop needs a path from low-fidelity estimates to stronger evidence, and it needs rules for when physical constraints—or tool limitations—reject an otherwise promising candidate.

The Architecture 2.0 move is to make those physical assumptions inspectable before the loop delegates work. A generic generator can propose a faster block or a clever dataflow; an architecture loop must say which power model, memory traffic model, placement assumption, timing margin, and escalation rule make that proposal credible. Without

that state, AI merely produces more candidates for a later physical-design step to reject. With that state, physical reality becomes an early design constraint, not a late surprise.

The important consequence is not that every early idea needs signoff-quality evidence. It is that the loop must know which physical assumptions are being made, what evidence would overturn them, and when to escalate from a proxy to stronger feedback. Otherwise the apparent speedup from AI-generated candidates is paid back later as discarded work.

The order-of-magnitude spread in Figure 2.4 is not something to memorize or treat as a current-node prediction. The architectural use is simpler. Local arithmetic and memory movement live on very different energy scales, so a loop that optimizes only arithmetic can improve the wrong thing. Advanced-node designs do not remove this lesson; if anything, the gap between local logic and moving data, driving wires, and feeding memory systems is one reason locality remains an architectural problem rather than a solved device-scaling detail. For the 3 nm-class lighthouse prompt, the loop would need a fresh power model before making a design decision.

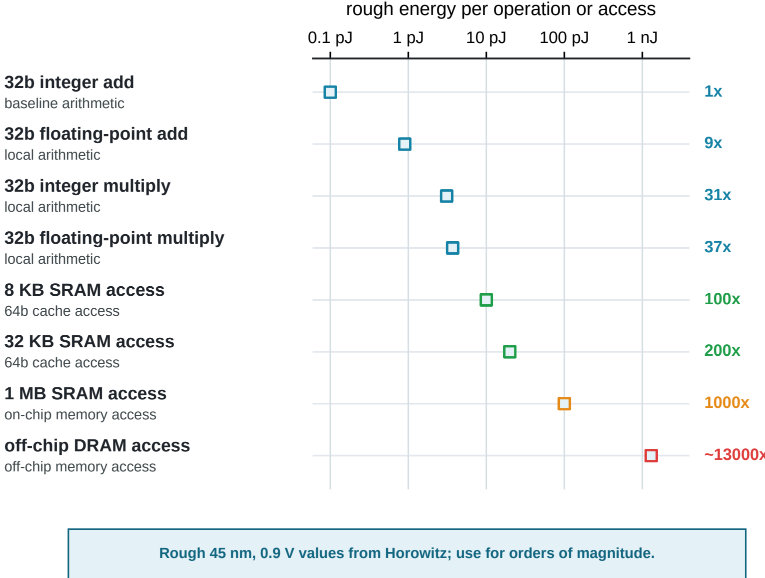


Figure 2.4: **Data movement can dominate arithmetic energy:** Rough Horowitz 45 nm energy values show why architecture loops must represent locality, buffering, precision, scheduling, and memory hierarchy rather than counting arithmetic alone. The values are order-of-magnitude interpretive anchors, not current-node device estimates.

The plot changes the section's claim from "data movement is expensive" to "locality is an early rejection condition". A loop that cannot represent memory movement cannot support a power or efficiency commitment.

2.8 Engineering Cost Creates the Scissors Gap

Each of these pressures, physical reality included, lands on the same fault line, a scissors gap. On one blade are design choices, workload variants, tool outputs, cross-layer assumptions, simulator hours, verification cases, EDA reports, physical constraints, and deployment signals. On the other blade are human attention, expert review time, tool budget, schedule, and verification capacity. The first blade rises quickly. The second does not.

Figure 2.5 makes the metaphor explicit. The upper blade is not only candidate count; it is the coupled burden of choices, constraints, evidence, software paths, and physical feasibility. The lower blade is not the ability to think; it is the slower-growing capacity to evaluate, review, reject, and commit with confidence.

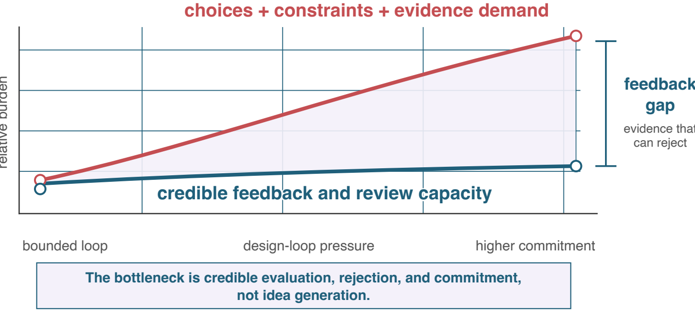


Figure 2.5: **The scissors gap is a feedback gap:** Architecture pressure rises when design choices, constraints, and evidence demand grow faster than unaided manual coordination and verification capacity. The widening region is the gap that Architecture 2.0 tries to make explicit and manageable.

As a silicon-facing scale anchor, the gap is also an engineering-cost problem. Public estimates vary, and they should not be treated as universal accounting rules, but their scale is useful. The Semiconductor Industry Association reports that the cost of designing a latest-node chip rose from about $30M for a 65 nm chip in 2006 to more than $540M for a 5 nm chip in 2020, a greater than 18× increase (Semiconductor Industry Association, 2026). A McKinsey analysis gives a similar order of magnitude, estimating roughly $175M for a 10 nm design, $300M for a 7 nm design, and $540M for a 5 nm design when validation, IP qualification, and related development costs are included (Bauer et al., 2020). These are not only mask or wafer costs. They are costs of architecture, design, validation, verification, IP, tools, and people.

Verification makes the people cost visible. In a summary of the 2022 Wilson Research Group functional-verification study, Foster reports that demand for IC/ASIC verification engineers grew faster than demand for design engineers from 2007 to 2022; the same summary reports that mean peak staffing is roughly one verification engineer per

design engineer across most market segments, that processor projects can reach a 5-to-1 verification-to-design ratio, and that design engineers spent 49 percent of their time in verification in 2022 (Foster, 2022). A later 2024 Wilson Research Group IC/ASIC report makes the commitment risk visible from another angle. It reports first-silicon success at 14 percent, the lowest level in two decades (Foster, 2025). This is why feedback and rejection are central to Architecture 2.0. Each invalid candidate consumes scarce engineering capacity and commitment risk, not just compute cycles. This is the economic reason rejection, not generation, must pace automated search. Generated candidates are cheap only until they consume scarce verification, tool, and review capacity. ::: {.callout-field-note title="The bug five entries wide: Pentium FDIV"} The 1994 Pentium FDIV bug cost Intel roughly $475M because five lookup-table entries were wrong, and no cheap check caught it before silicon.

Takeaway. The missing rejection, an automated gate that could have caught the error early, not the missing entries, set the price. ::: Figure 2.6 turns the public dollar estimates into a scale check. It should not be read as a universal cost curve. Different products, IP reuse strategies, node maturities, organizations, and accounting boundaries produce different numbers. The robust point is simpler. As the loop moves toward leading-edge implementation, feedback and commitment consume real engineering budgets, not only simulator cycles.

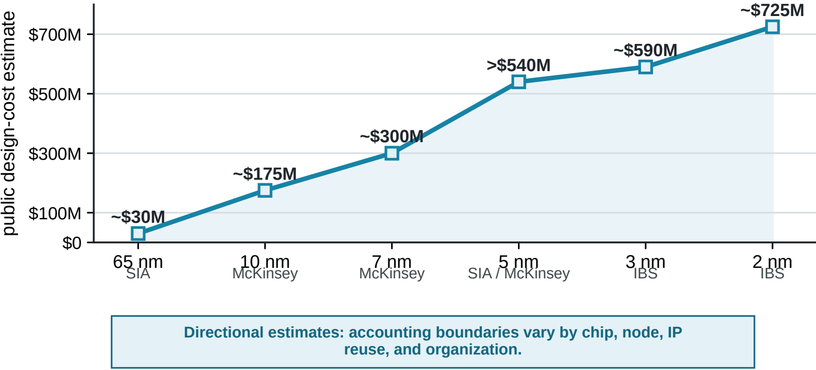


Figure 2.6: **Leading-node design-cost estimates make the scissors concrete:** Public SIA and McKinsey estimates (Semiconductor Industry Association, 2026; Bauer et al., 2020) put latest-node chip design costs from tens of millions of dollars at 65 nm to more than $500M at 5 nm, with IBS estimates extending the trend to roughly $725M at 2 nm (International Business Strategies, 2024). The accounting boundaries vary, but the scale makes feedback, validation, IP, tools, and human engineering effort part of the architecture loop.

The cost plot is not an indictment of architects. It says the unit of work has changed. The bottleneck is trusted feedback. That means preserving invalid candidates, proxy failures, assumptions, and the decision a human can responsibly commit to. Without that record,

teams rerun old mistakes, rediscover invalid regions, and mistake more output for more architectural progress.

This bottleneck is not only a diagnosis. It has a bound. The diagnostic table earlier in this chapter asked which part of one loop saturates first; Chapter 9 makes the stronger, loop-wide claim precise. Whichever part saturates, end-to-end throughput is set by how cheaply the loop can reject, not by how much it can generate. Generating proposals faster than trusted feedback cannot keep up widens the scissors gap rather than closing it. Just as Amdahl's Law dictates that sequential code bottlenecks multicore speedups, the Architecture 2.0 loop has its own Amdahl's Law. No matter how fast the generative AI proposes candidates, the throughput of the design loop is bounded by the serial fraction of trusted, independent rejection capacity (see Figure 2.7). That capacity has two parts, and they age differently. How *fast* the loop can reject is a throughput limit, and cheap, independent, increasingly automated rejectors keep raising it. Who is *accountable* for the commitment if a claim is wrong is not a speed limit at all; it is a governance floor that stays human even as verification gets faster. Chapter 9 makes the throughput bound precise and shows how much of it an automated rejector can discharge before it reaches what only accountable judgment can clear.

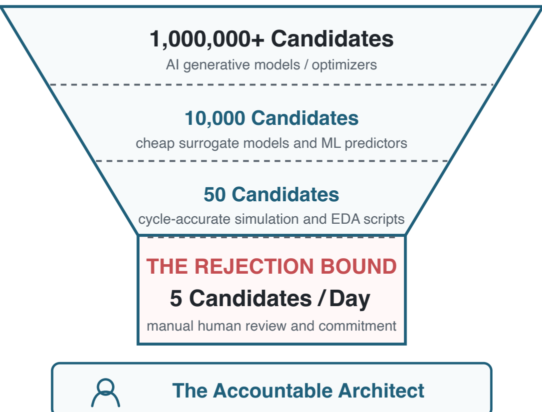


Figure 2.7: **The Rejection Bottleneck:** No matter how many candidates an AI loop can generate (millions) or how fast surrogate models can screen them (thousands), the total throughput of an architectural exploration loop is bottlenecked by the serial capacity of trusted, independent rejection, illustrated here by the human-review floor that only accountable judgment can clear.

2.9 Feedback and Verification Become the Bottleneck

That human review and rejection capacity is so strictly bounded because architecture feedback has uneven cost and uneven authority. A spreadsheet model is cheap but weak. A simulator may be more informative but slow or biased. A synthesis result exposes more implementation reality but depends on tool settings and constraints. Physical design and signoff are stronger still, but expensive and late. Silicon and deployment telemetry are authoritative in different ways, but they arrive after major commitments.

This feedback-regime structure makes naive autonomy dangerous. An optimizer that targets a cheap proxy may move quickly in the wrong direction. A search method that reports a Pareto frontier may hide invalid configurations, failed tool runs, or assumptions that would not survive signoff. A generated RTL fragment may look plausible but fail under verification or integration. Feedback becomes evidence only when its fidelity and provenance are clear, the discipline that Chapter 7 develops in full.

This is why Architecture 2.0 does not begin with autonomy. It begins with the loop. The loop must say what can be changed, what can be observed, what can reject a candidate, what evidence is strong enough for the next commitment, and what remains a human decision.

Architect's checkpoint: The Automation Gate

When does an automated proxy ranking escalate to a human architectural decision? The loop must explicitly define which metrics, constraints, and failures require human signoff before the automated method can move a design to the next commitment level.

This discipline of designing the loop forces us to elevate feedback above generation, forming a core design principle for Architecture 2.0.

Design principle: Treat feedback as the bottleneck

More AI-generated candidates do not help a loop that cannot evaluate, reject, and justify them. Design for the rate of trusted feedback first, and add AI generation only as fast as human or automated rejection can keep up.

2.10 Architecture Violates Generic AI Assumptions

Designing a loop around the rate of trusted feedback is difficult exactly because architecture is not a standard machine-learning problem. Many successful AI systems are built in domains that generic machine-learning workflows take for granted, domains with abundant data, cheap feedback, stable labels, and clear losses. Those are exactly the conditions a design loop would want, and computer architecture violates them at almost every boundary. Data are often proprietary, incomplete, stale, or missing the rejected

and failed candidates that explain why a design space has certain boundaries. Chapter 4 gives those records a sharper name and makes them part of loop state. Feedback ranges from a quick proxy to a simulation, synthesis run, physical-design report, expert review, emulation result, or silicon measurement, each with different latency, cost, fidelity, and authority. The action space is also unusual. Many generated configurations are not merely low quality; they are illegal, unsupported by tools, unverifiable, or incompatible with software and physical constraints.

The result is not a simple lack of data. It is a mismatch between generic AI assumptions and architecture-loop requirements. Architecture loops need representations that carry constraints, provenance, feedback cost, uncertainty, and rejection conditions. They also need methods that understand when a proxy result is only a proxy and when a decision is moving toward a higher commitment level.

A conventional machine-learning workflow is still useful as a foil. Students often learn a pipeline that runs from data collection to preprocessing, training, validation, deployment, and monitoring. Figure 2.8 keeps that familiar picture but shows why the architecture version cannot be a simple pipeline. Every step must carry validity constraints, tool costs, provenance, drift, rejected alternatives, and a commitment boundary.

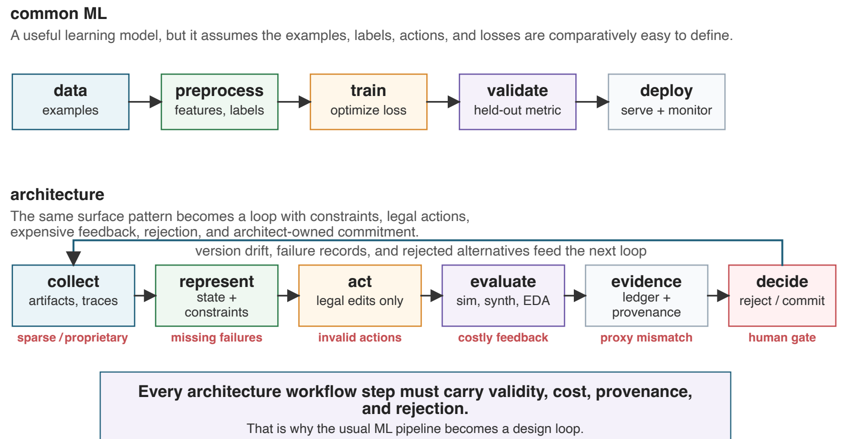


Figure 2.8: **Architecture turns the ML pipeline into a constrained loop:** A generic ML workflow can be taught as data, preprocessing, training, validation, and deployment. In architecture, the corresponding workflow must also represent constraints, legal actions, expensive tool feedback, an evidence ledger, failed and rejected candidates, drift, rejection authority, and a commitment boundary.

The mismatch has several concrete forms. Data are not just examples; they are design artifacts with permissions, tool versions, and missing failures. Feedback is not just a label; it spans regimes such as proxies, simulation, synthesis, physical design, expert

review, emulation, and silicon. Actions are not just tokens; they are edits to configuration spaces, software interfaces, RTL, constraints, and deployment policies, many of which can be invalid. Losses are not just scalar rewards; they are multiobjective efficiency claims under performance, power, area, cost, reliability, sustainability, and verification burden. Table 2.5 gives the checklist version of that argument.

Table 2.5: **Generic AI assumptions break at architecture boundaries:** Architecture loops must represent proprietary or incomplete data, expensive feedback, invalid actions, drifting workloads, multiobjective efficiency, proxy mismatch, and rejected or failed candidates.

Common AI assumption	Architecture violation	Loop implication
Abundant labeled data	Many labels are proprietary, expensive, stale, or not recorded.	The loop must build source records, provenance, and reusable failure records.
Cheap feedback	Simulation, synthesis, EDA, emulation, and review can be slow or scarce.	Methods must be sample-efficient and aware of feedback budgets.
Stable distribution	Workloads, software stacks, compilers, models, and deployment policies drift.	Benchmarks and representations must be versioned and revisited.
Valid actions are easy to define	Many generated configurations are invalid, unsafe, unverifiable, or unsupported by tools.	Environments need action schemas, constraint checks, and rejection authority.
Reward is clear	Efficiency mixes performance, energy, area, cost, reliability, sustainability, verification burden, and risk.	Objectives must be explicit, multiobjective, and tied to human decisions.
Proxy metrics are enough	Proxy wins can vanish at higher fidelity or under different workloads.	Evidence needs fidelity-regime checks and sensitivity checks.
Failures are just bad samples	Failed runs, rejected candidates, and invalid states describe the design space.	Failure records should be preserved as architecture data.

This mismatch does not make AI irrelevant. It makes representation and loop design central. Architecture needs generation, prediction, optimization, critique, retrieval, and tool use. But those capabilities must operate inside an environment that knows what actions are legal, what feedback means, and who can say no.

2.11 AI Helps Only When the Loop Is Designed

If the environment can enforce those boundaries, AI becomes profoundly important because the classical loop is under pressure. It can help summarize tool outputs, propose candidates, search spaces, predict costs, construct tests, critique assumptions, retrieve prior designs, and coordinate subtasks. In domains with expensive feedback and large design spaces, even partial improvements in search, triage, and explanation can matter.

But AI is not sufficient because architecture is not only generation. The architectural problem is to produce a credible system artifact under constraints. That requires state, tools, evidence, rejection, and commitment. A model that proposes a design without exposing its assumptions has not solved the architecture problem. A loop that finds a better proxy score without recording rejected and failed candidates has not produced trusted evidence. A system that cannot say what rejects its own output cannot be given high-commitment authority.

The right conclusion is therefore narrower and stronger than generic AI optimism. We should not merely search larger design spaces. We should design loops that learn, record, reject, and justify architecture work.

2.12 Conclusion

This chapter asked what the loop must record, when architecture choices grow faster than trusted feedback, for AI to help rather than merely add candidates. The pressure is structural, not a matter of taste. Two walls converge on the classical loop. One is physical, built from specialization, heterogeneous integration, and data movement. The other is verification, built from software velocity and validation burden. Between them opens a scissors gap, in which generative methods raise the rate of plausible proposals far faster than any tool raises the rate of trusted rejection.

That reframes the problem. The scarce resource in modern architecture is not ideas but trusted feedback, the ability to evaluate, reject, revise, and commit at the fidelity a decision demands. An AI that widens the design space without widening rejection capacity makes the gap worse, not better. This is why feedback, not generation, sets the pace of progress, and why the chapter treats it as the bottleneck to design around.

AI assistance earns its place only inside a loop built to absorb it, one that records state, preserves failed and rejected candidates as evidence, names what can say no, and keeps a human accountable for each commitment. The goal is not a larger search space but a loop disciplined enough that more proposals become more progress instead of more noise.

2.13 Open Research Questions

The strain on classical design loops reveals that AI in architecture is not merely a generation problem, but a systems-design challenge. The following open questions highlight the research needed to make AI-assisted architecture reliable, verifiable, and capable of handling exploding design spaces.

For the strained-loop regime:

1. **The Minimum Inspectable Loop-State:** How can we formally define the minimal state record—spanning task intent, action schemas, candidate provenance, and rejected alternatives—required to make an AI-generated architecture fully reviewable? As the scissors gap widens, this challenges the community to standardize the "black box" of AI generation so that human architects can securely audit and commit to designs even when high-fidelity samples are prohibitively scarce.
2. **Representing Failure and Uncertainty as Reusable Evidence:** In the absence of abundant, open-source architectural data, how can a design loop safely internalize proprietary constraints, failed candidates, and tool-version drift? This requires pioneering new data representations that treat rejected designs and physical constraint violations (see the discussion on "Physical Constraints Move Into Architecture" in Section 2.7) not as wasted effort, but as mathematically rigorous negative evidence that prevents generative models from repeatedly exploring invalid regions.
3. **Provable Rejection Authority in the Scissors Gap:** How can we construct a hierarchy of cheap, high-confidence rejection proxies that filter invalid AI-generated candidates before they consume validation cycles? Because feedback and validation are the ultimate limits on throughput (see the discussion on "Feedback and Verification Become the Bottleneck" in Section 2.9), discovering provably safe triage rules that minimize both false acceptances (which waste time) and false rejections (which discard optimal designs) is critical to scaling automated exploration.
4. **Versioning the Validity Horizon Against Software Drift:** How must architectural claims be packaged to support reproducibility when software, models, and workloads mutate faster than the silicon design cycle (Section 2.6)? This invites the creation of dynamic, versioned workload snapshots and automated contracts between hardware commitments and software targets, so an AI-specialized architecture can trigger re-evaluation before a deployment bet becomes obsolete.

What to carry forward

- **Reader test:** When candidate scale and feedback cost grow together, can you say why trusted feedback, not idea generation, is the bottleneck, and what loop state the scissors gap demands?
- **Up next:** AI helps only when the loop records state, failures, evidence, rejection authority, and each commitment's human owner; the discussion on "Architectural Claims and Design Loops" (Chapter 3) names the minimum loop state that makes that pressure inspectable.

Chapter 3

Architectural Claims and Design Loops

"All models are wrong, but some are useful."

— George E. P. Box, *Robustness in the Strategy of Scientific Model Building* (1979)

Author's Note: George E. P. Box, a British statistician who made major contributions to time-series analysis, famously noted that while all models are flawed, the good ones are practically useful. For us, this perfectly captures the role of proxies and surrogate models. While fast proxies are inherently inaccurate, they are absolutely essential for accelerating an AI agent's search.

The crux

What does a design loop need to make explicit before an AI output can be accepted or rejected as an architectural claim?

Computer architecture has always depended on disciplined abstraction. An architect rarely reasons directly from every transistor to every application behavior. The field instead builds models, simulators, workload characterizations, cost estimates, design rules, and review practices that make large design spaces tractable. That quantitative tradition is central to modern architecture practice (Hennessy and Patterson, 2017). It also explains why Architecture 2.0 should not be framed as a sudden break from the past. The continuity is the *existence* of the loop; the field has always designed through loops of abstraction, measurement, feedback, and judgment. The discontinuity is its *automation*. Historically, humans drove this loop; in Architecture 2.0, AI systems can execute steps of it programmatically, acting inside the loop while the architect owns the commitment. That automation is why the failure mode changes from slow human drift to high-velocity, catastrophic failure when the verification and rejection mechanisms collapse.

What becomes first-class alongside the artifact is the loop that produces and tests claims about it. In Architecture 1.0, the architect uses tools to design artifacts: an ISA extension, a cache hierarchy, an accelerator, a memory system, a chiplet partition, a compiler policy, or a system configuration. But an artifact matters because it supports an architectural claim. These claims collapse into two orthogonal system axes: Pareto efficiency (optimizing latency, energy, correctness, and useful work along an existing frontier) and boundary expansion (shifting the frontier to make previously impossible systems possible under a workload and set of constraints). In Architecture 2.0, the architect must also design the loop that produces, tests, rejects, and revises those claims. The loop itself must be formalized as a five-part execution state: what state it saw, what actions it allowed, what alternatives it rejected, what evidence supports the result, and who owns the commitment if the claim is wrong (Janapa Reddi and Yazdanbakhsh, 2025). Without those pieces, an AI system may still produce plausible text, code, or configurations, but it is not participating in architecture work in a way the field should trust.

Making a design process explicit is what mature engineering fields do once the cost of an undetected error grows. Aviation did not become safe by trusting more skilled pilots; it formalized the process with checklists, assurance cases (Kelly and Weaver, 2004), and certification standards that make the evidence for a safety claim auditable. Software operations did not become reliable by hiring more careful engineers; DevOps and site-reliability engineering, disciplines focused on automating and monitoring software deployment, made deployment, monitoring, and rollback explicit, with error budgets and runbooks that state what evidence justifies a change (Beyer et al., 2016). The common pattern is a move from individual judgment to a represented process with explicit evidence and rejection rules. Architecture 2.0 makes the same move for computing-system design. The alternative is not creative freedom; it is an ad hoc loop whose assumptions, rejected alternatives, and evidence live only in people's heads, which is exactly the state that stops scaling when the design space and the verification burden grow together.

This chapter gives the reusable language for that shift. The goal is not to classify every paper or tool. A taxonomy of current systems will age quickly. The more durable contribution is a consistent way to state the architectural claim being made, together with an ontology of the entities and relationships that must exist before AI systems can act inside the architecture design loop credibly.

The ontology has to earn its space by being useful. A researcher should be able to cite it when explaining the structure of an Architecture 2.0 contribution. A reviewer should be able to use it to ask what state, action, feedback, evidence, and rejection authority a paper exposes. A tool builder should be able to use it as a checklist for a harness or environment. An author should be able to use it to state the claim and loop for a concrete design problem. If the ontology cannot support those uses, it is only vocabulary. Its test is whether it exposes enough state for a generative method to act within bounds and enough evidence for a human architect to reject or commit the result.

The deeper reason we need such an ontology is that architecture reasoning still lacks a durable structured layer above RTL. Below RTL, design flows already have artifacts that tools can parse, transform, reject, and sign off. Above RTL, much of the reasoning that decides *what* should be built still lives in whiteboards, spreadsheets, scripts, slides, review memories, and natural-language specs. The design-loop card, loop contract, environment contract (Chapter 5), and evidence ledger are not paperwork around the real work. They are candidate structured abstractions for the part of architecture practice that has not yet been made inspectable.

The hierarchy in this chapter is simple. The artifact is what may eventually be built. The architectural claim is what a reviewer accepts or rejects about that artifact. The design loop is the process that tests the claim. The ontology names the state and relationships the loop must expose. The design-loop card is the compact record that makes the claim and loop reviewable. The loop contract from Chapter 1 is the same object seen before action: it states the task, state, actions, feedback, evidence standard, and decision owner the loop promises to expose, and the card records how the loop kept that promise.

Figure 3.1 places that layer between tacit reasoning and implementation flows. Its purpose is not to add ceremony. It is to make the part of architecture reasoning above RTL explicit enough that tools, automated participants, reviewers, and architects can all see the same loop state.

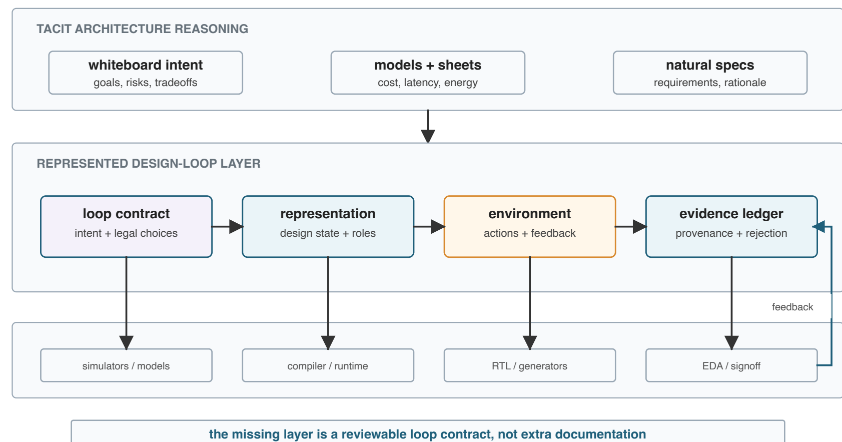


Figure 3.1: **Architecture 2.0 needs a structured layer above RTL:** Whiteboards, spreadsheets, and natural-language specs still carry much of the reasoning that decides what should be built. The loop contract, represented state, environment contract, and evidence ledger turn that reasoning into a reviewable layer that can connect to simulators, compilers, RTL, EDA, and software evidence.

> What this chapter gives you
>
> After this chapter you can turn an AI prompt, paper, tool, or project into a reviewable design-loop card. That means you can:
>
> - **write an AI-generated architectural claim as a reviewable object:** workload, baseline, design space, objectives, constraints, evidence, rejection authority, commitment boundary, and human decision;
> - audit an AI-assisted paper, tool, or project by naming the loop state, legal actions, environment feedback, evidence ledger, rejection and escalation rules, and human-owned commitment it exposes;
> - explain how artifact, claim, loop, ontology, checklist, and design-loop card fit together in an AI-assisted workflow;
> - distinguish an ontology from a taxonomy and say why ontology comes first;
> - judge how much autonomy an AI-driven loop has actually earned.

3.1 The Architectural Claim Is the Unit of Review

Asking whether a generative model[4] can design hardware is too coarse a framing. The more architecture-native approach focuses on identifying the specific claim being made and determining the evidence required to make that claim credible. Architects rarely accept an artifact by itself. They accept or reject a claim about that artifact relative to a workload, baseline, design space, objectives, constraints, and evidence. The definition below names that review object so the rest of the chapter can ask what state a loop must carry before an AI-assisted result can be accepted or rejected.

[4] A class of AI models designed to generate new data, such as text or configurations, that resembles its training data.

> **Architectural claim.** An architectural claim is a statement that a proposed artifact, method, or loop improves, preserves, or explains a hardware/software behavior for a specified workload or scenario relative to a baseline, under explicit objectives, constraints, evidence, rejection conditions, and decision authority.

The clause "for a specified workload or scenario" is doing real work, and a neighboring field paid to learn it.

Field note: Valid for the last rocket, not this one

The Ariane 5 rocket reused an inertial-reference software component that had flown correctly on Ariane 4. Its correctness claim was real, but it was a claim about Ariane 4's flight envelope. On Ariane 5's different, higher-velocity trajectory a value overflowed a conversion the older profile never reached, both channels shut down, and the vehicle self-destructed about forty seconds after launch (Lions, 1996). The component was not broken. Its evidence simply did not cover the scenario it was now committed to.
Takeaway. A claim carries the scenario it was verified under. Reusing a cheap, trusted component still means re-checking its evidence against the new workload, because the claim, not the code, is what has to hold.

A plain-language version might say, this candidate improves useful work for this workload, relative to this baseline, inside this design space, under these constraints, using this evidence, and with this rejection authority. A compact way to write the same review object is

$$C = \langle W, B, \mathcal{D}, \mathbf{J}, \mathcal{K}, E, R, M, H \rangle.$$

Here, W is the workload or scenario, B is the baseline, $\mathcal{D}$ is the legal design space, $\mathbf{J}$ is the objective vector, $\mathcal{K}$ is the constraint set, E is the evidence, R is the rejection authority, M is the commitment boundary, and H is the human or organizational decision authority. This tuple is the claim slice of the design-loop card, not a second schema. The notation prevents a generated artifact from masquerading as an architectural result before the comparison, constraints, evidence, rejection authority, and commitment boundary are visible.

In card terms, W and B anchor the task and evidence comparison, $\mathcal{D}$ is the design space, $\mathbf{J}$ and $\mathcal{K}$ belong in the representation and evidence standard, E is the evidence ledger, R

is the rejection authority, M is the commitment boundary, and H is the human decision owner.

Table 3.1 turns the tuple into a reader checklist for the lighthouse prompt. The important point is that the prompt's compact wording hides a large amount of architectural state. A credible answer must expose that state before the reader can judge whether the result deserves trust.

Table 3.1: **An architectural claim needs more than an artifact:** The lighthouse prompt becomes reviewable only when the workload, baseline, design space, objectives, constraints, evidence, rejection authority, commitment boundary, and human decision are explicit.

Claim field	Reader question	Lighthouse instance
Workload or scenario	What behavior is the design supposed to serve?	XRBench-class real-time mobile XR workloads, with latency, sensing, graphics, and interaction requirements.
Baseline	Compared to what architecture, software stack, or prior result?	A scalar CPU-only baseline, a vector-capable CPU, an accelerator baseline, or an existing mobile XR subsystem.
Design space	What choices are legal, and which regions are invalid?	RISC-V ISA options, vector width, CPU/accelerator partitioning, memory hierarchy, clocking, compiler/runtime path, and tool-flow limits.
Objective vector	What counts as improvement, and what tradeoffs matter?	Throughput, tail latency, energy, area, programmability, verification burden, and evidence cost under the 3 W target.
Constraints	What cannot be violated even if a metric improves?	ISA compatibility, correctness, thermal limits, process assumptions, package limits, software compatibility, and 3 nm-class low-power envelope.
Evidence	What supports the claim at the required commitment level?	Workload traces, simulations, power model, sensitivity checks, rejected candidates, tool logs, and comparison against baselines.
Rejection authority	What observation, tool, review, or rule can invalidate or weaken the result?	Missed latency target, power envelope violation, invalid RTL/configuration, compiler failure, simulator mismatch, or weak coverage.
Commitment boundary	What claim is the evidence strong enough to support, and what remains uncommitted?	Exploration, RTL study, implementation, deployment, or silicon-facing commitment, with stronger evidence required at each boundary.

Claim field	Reader question	Lighthouse instance
Human decision	Who can accept, revise, escalate, or commit the claim?	The architect or review process that owns assumptions, evidence thresholds, risk, and final commitment.

This schema also clarifies what AI systems are being asked to do. Generation can propose artifacts inside $\mathcal{D}$. Prediction can estimate components of $\mathbf{J}$ before expensive feedback. Optimization can search tradeoffs under $\mathcal{K}$. Critique and verification can apply R. The architectural result is not any one of those operations. It is the claim that survives the loop.

Design principle: State the claim as a review object

An AI-generated architectural result is credible only when the loop names its workload, baseline, design space, objective, constraints, evidence, rejection rule, and decision owner. Until it states those, the AI output is an artifact, not an architectural claim.

3.2 The Design Loop Is the Unit of Analysis

Once the claim is explicit, the architect must define the design loop that can test it. That shift matters because architecture work is not a single act of generation. It is a repeated process of framing a problem, choosing abstractions, exploring alternatives, measuring candidates, rejecting weak results, revising assumptions, and deciding when evidence is strong enough to commit.

> **Architecture design loop.** An architecture design loop is the repeated process that carries architecture state through bounded actions, feedback, evidence, rejection, revision, and architect-owned commitment until it produces an artifact or a revised loop.

For Architecture 2.0, this is the loop an AI system enters. If its state, actions, feedback, and stopping rules are implicit, the system is only producing outputs, not participating credibly in architecture work.

For the lighthouse prompt, the distinction is immediate. A request for a low-power, 64-bit RISC-V-based compute subsystem, using the open standard instruction set architecture, for XRBench-class mobile XR, a mixed-reality workload suite, under a 3 W, 3 nm-class low-power mobile envelope sounds compact. But the prompt does not define the design loop. It does not say which workload traces are authoritative, which vector operations matter, which memory hierarchy is admissible, which software stack must run, which simulator is trusted, which power model applies, which process assumptions are available, which alternatives must be considered, or what evidence is enough to reject a candidate.

These are not details to add after a generative method responds. They are the architecture problem.

In practice, the loop has at least the following elements. It has a state: what is known about the workload, design, tools, constraints, and prior evidence. It has actions: what can be changed, generated, queried, tuned, or tested. It has observations: what the loop can see after an action. It has objectives and constraints: what counts as progress and what is not allowed. It has a feedback path: the measurement, simulation, synthesis report, trace, review, or deployment signal returned by the environment. It has stopping and escalation rules. It has decisions: accept, reject, revise, or request stronger evidence. It has artifacts: reports, configurations, design descriptions, plots, RTL fragments, benchmarks, or implementation plans.

A compact way to write the loop is

$$s_{t+1} = \text{Update}(s_t, a_t, o_t, e_t, d_t).$$

Here, s_t is the represented architecture state, a_t is the bounded action taken by the loop, o_t is the observation returned by the environment, e_t is the evidence ledger entry that makes the observation auditable, and d_t is the human or policy decision to accept, reject, revise, or escalate. The equation is not claiming that every architecture loop is a Markov decision process. It is a bookkeeping discipline. If a loop cannot say how actions, feedback, evidence, and decisions update state, then it is not yet represented well enough for credible AI-mediated architecture work. Figure 3.2 visualizes how these pieces form a unified design loop.

Markov decision process (MDP): A mathematical framework for modeling decision-making where outcomes are partly random and partly under the control of a decision maker; it forms the foundation of reinforcement learning.

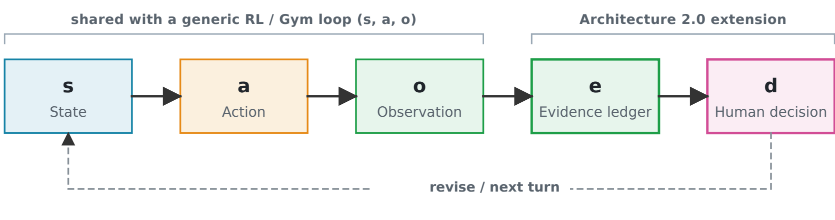


Figure 3.2: **The Architecture 2.0 loop tuple:** The design-loop update s, a, o, e, d extends a generic reinforcement-learning loop's state, action, and observation with an evidence ledger and a human decision, replacing the scalar reward the environment must never compute.

Architecture 2.0 uses that loop as the unit of analysis. A model is one participant in the loop. It may generate candidates, summarize evidence, predict outcomes, call tools, critique assumptions, or coordinate subtasks. But the credibility of the result comes from the whole loop, not from the model in isolation.

The cross-layer reach of architectural decisions is another reason the loop, not the model, is the unit of analysis. An architecture decision moves through the whole stack at once. A choice that is legal at the microarchitecture level can still force a new compiler pass above it or trip a timing and routing bottleneck below it. The model that proposes the choice sees only its own layer, while the consequences ripple up into software and down into physical design. Representing those layers as intertwined, rather than as isolated abstractions, is part of what the loop has to carry, because that cross-layer state is what lets a reviewer reject a candidate that looks valid in one layer but breaks another the model never saw.

3.3 Design Spaces Make Claims Meaningful

An architectural claim is meaningful only relative to a design space. A system that reports "the best" candidate without exposing the alternatives, invalid regions, baseline, and tradeoffs has not made an architecture result easy to review. It has hidden the comparison that gives the result meaning.

In architecture, the design space is not the set of all strings a model might emit. It is a constrained set of legal choices:

$$\mathcal{D} = \{x \in X \mid x \text{ is valid under the task, tool chain, and constraints}\}.$$

Here, X is that unconstrained space of candidate artifacts, everything a method could emit. The validity conditions may include ISA compatibility, memory semantics, software support, timing assumptions, power limits, package constraints, verification requirements, and deployment policy. A candidate outside $\mathcal{D}$ is not a bold design. It is an invalid action unless the loop explicitly revises the design space and records why.

The lighthouse prompt makes this concrete. A 64-bit RISC-V-based mobile XR subsystem might vary vector width, cache and memory hierarchy, accelerator partitioning, dataflow, clocking, voltage assumptions, compiler/runtime support, and verification scope. Some choices are legal but unattractive. Some are attractive under a proxy but fail at higher fidelity. Some violate the power envelope, process assumptions, software contract, or workload coverage. An Architecture 2.0 loop must represent those distinctions. Otherwise it cannot know whether it is improving a design or exploiting a hole in the problem statement.

The design space is also where multiobjective efficiency enters the ontology. The objective is rarely a scalar reward. It is a vector of performance, energy, latency, area, reliability, programmability, verification burden, cost, and evidence requirements. A

design-space report is therefore an evidence object. It should show what was explored, what was rejected, what tradeoffs remain, and what the architect must still decide.

3.4 The Architecture 2.0 Ontology

A method label is not enough to review an architecture claim. A paper can say it uses an LLM, reinforcement learning, Bayesian optimization, or a surrogate model and still leave the important state invisible: what task was bounded, what actions were legal, what feedback was used, what failed, and who accepted the commitment. A taxonomy groups things. It can list tasks, methods, benchmarks, tools, system architectures, or evaluation settings. Taxonomies are useful, and this chapter will use them where they help a reader make decisions. But a taxonomy is not enough for a field that is still moving. Model interfaces will change. Method harnesses will change. Benchmarks will change. Electronic Design Automation (EDA) flows and simulator stacks will change. If the book is organized only around today's artifacts, it will age with them.

Bayesian optimization: A strategy for optimizing expensive black-box functions, where each candidate evaluation (such as a cycle-accurate simulation) is costly.

Architecture 2.0 ontology. The Architecture 2.0 ontology names the entities and relationships that must exist for AI-mediated architecture work to be represented, acted on, evaluated, rejected, and committed by a human architect.

Two of those entities matter enough to name now, because the rest of the book leans on them: the world model the loop reasons with, and the evidence ledger it reasons from.

World model. A world model is the loop's belief about how architecture actions change outcomes, whether that belief lives in a simulator, a learned surrogate, a cost model, or design rules. This chapter uses the term only at that working level; Chapter 4 gives the canonical definition and shows what such a model must encode, scope, and keep credible.

Both entities are critical, but while the world model focuses on prediction and belief, the evidence ledger serves as the historical record of truth.

Evidence ledger. An evidence ledger is the durable record that ties a candidate to the feedback that evaluated it, the constraints it faced, and the decision that accepted or rejected it. To flatten the learning curve, consider these terms as the automated equivalent of a scientific peer-review loop or a Design Space Exploration (DSE) review board: a *rejection gate* is analogous to a reviewer rejecting a paper for a methodological flaw, and an *evidence ledger* is a multidimensional record capturing the Pareto tradeoffs and justifying exactly *why* alternative architectures were rejected during exploration.

Read the chapter as a chain of loop obligations. The claim states what is at stake; the ontology names the state, action, feedback, evidence, and decision relationships; the checklist tests whether an automated system can act safely; and the design-loop card records the contract for review. Within this framework, architecture environments act as the tool-connected settings that define legal actions, observable feedback, costs, failure modes, provenance, and invalid-action behavior; Chapter 5 gives the canonical definition.

Zooming out, the ontology establishes the fundamental requirements for AI-mediated architecture work to be credible by defining which entities must exist and how they must relate. The important pieces are not only the nouns. They are the relationships. Intent constrains tasks. Tasks determine what must be represented. Representation limits what the loop can observe and modify. The world model encodes beliefs about how actions change outcomes. Tools and environments define valid actions and measurable feedback. Methods are selected for the task, representation, and feedback budget. Feedback becomes evidence only when fidelity, provenance, uncertainty, and relevance are understood. Human decisions accept, reject, revise, or escalate the result. This is why ontology should precede taxonomy. Rather than immediately classifying whether a paper uses an LLM[5], reinforcement learning[6], Bayesian optimization, a surrogate model, or a simulator wrapper, the analysis must first identify the exposed loop. The evaluator must determine the task, the represented state, the legal actions, the environment returning feedback, the feedback budget, the supporting evidence, the rejection conditions, and the remaining architectural decisions. Once those components are clear, a taxonomy of methods becomes useful. Before that, method labels can hide more than they reveal.

The practical implication is not to build a chip-specific language model[7] first. A field becomes AI-addressable only when its objects of work, legal actions, feedback, evidence, rejection rules, and commitment boundaries are represented well enough for methods to act and for experts to judge. For architecture, those objects are not only papers or text. They include workloads, design states, tool configurations, invalid actions, failure records, fidelity levels, and commitment decisions. That is why this book starts with an ontology of the design loop rather than a catalog of current models.

[5] Large Language Model, a neural network trained on vast amounts of text to understand and generate human-like language.

[6] A machine learning training method based on rewarding desired behaviors and punishing negative ones.

[7] A statistical model trained to predict the next word or token in a sequence, forming the foundation of modern AI text generation.

3.5 The Compact Framework

Figure 3.3 asks what has to become explicit before an AI method can participate in architecture work. Read it left to right as a chain of obligations: intent bounds the task, representation and world model bound what can be known, architecture environments return feedback, and the evidence ledger plus human decision determine commitment.

The change is that the loop, not the model, becomes the review object; missing links are reasons to withhold commitment rather than details to fill in later.

For practical use, this book compresses the ontology into *five framework elements*. They group the same twelve fields the design-loop card records: intent, task, design space, representation (which carries the world model), environment, method role, feedback budget, evidence, negative traces, rejection authority, commitment boundary, and human decision. These elements name what a loop is *made of*, distinct from the *five-part execution state* of Chapter 1, which names what one turn *records*. There is one artifact, the twelve-field design-loop card; every other list in this book, the five framework elements, the five-part execution state, the nine-field claim tuple, and the seven-question loop contract, is a view of it (Appendix B), not a competing schema.

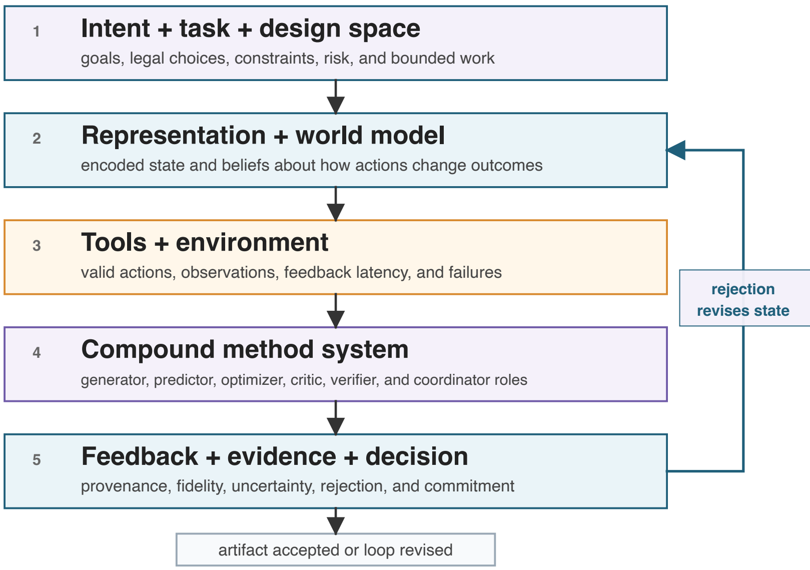


Figure 3.3: **The Architecture 2.0 ontology chain makes the loop explicit:** Intent, task, and design space define what work the loop is allowed to do; representation and world model define what the loop can know; tools and environments define valid action and feedback; compound methods act inside the loop; evidence and human decision determine whether an artifact is accepted, rejected, or used to revise the loop.

First, there is *task, intent, and design space*. Intent states what the architect or organization is trying to achieve, what constraints matter, what risks are acceptable, and what cost of failure is tolerable. The task is the bounded work unit that can be assigned, repeated, measured, or decomposed. The design space states which choices are legal, which regions are invalid, and which tradeoffs the loop is allowed to explore.

Second, there is *representation and world model*. A representation is the encoded design state: specifications, workload traces, architecture descriptions, graphs, RTL, compiler IR, simulator configurations, EDA reports, benchmark metadata, tool logs, design documents, or review notes. A world model is the loop's belief about how architecture actions change outcomes. It may be explicit, learned, simulator-backed, symbolic, statistical, or partly implicit in tools.

Third, there is the *architecture environment*. Tools become environments when they define actions, observations, constraints, costs, rewards or objectives, latency, provenance, and invalid-action behavior. An architecture simulator is not merely a measurement device in such a loop. It is part of the state transition and feedback system.

Fourth, there are *method roles*. The credible unit is rarely a single model. It is a composition of roles: generator, predictor, optimizer, searcher, critic, verifier, planner,

tool caller, and coordinator. Some roles may be played by language models, some by search algorithms, some by learned surrogates, some by scripts, some by formal tools, and some by humans.

Fifth, there is *feedback, evidence ledger, and human decision*. Feedback is any signal returned by the loop. Evidence becomes useful when it is tied to provenance, fidelity, assumptions, uncertainty, coverage, rejection, and a decision. The decision is where the architect accepts, rejects, revises, or escalates the result.

Table 3.2 gives the checklist version. It is the question a reader should be able to ask of a paper, benchmark, tool, or internal loop before trusting an Architecture 2.0 claim.

Table 3.2: **The framework becomes a checklist when each loop state is explicit:** A project is easier to review when it names the task, representation, architecture environment, method roles, feedback, evidence ledger, rejection authority, and human decision before claiming autonomy or architectural progress.

Framework piece	Reader question	Lighthouse instance
Task, intent, and design space	What architectural objective is being pursued, under what constraints, risk, and legal choices?	Improve mobile XR efficiency within a 3 W, 3 nm-class low-power mobile envelope while exploring legal RISC-V, vector, memory, accelerator, and software-stack choices.
Representation and world model	What state is encoded, and what does the loop believe about how actions change outcomes?	Workload traces, parameters, compiler assumptions, power model, memory behavior, and constraints.
Architecture environment	What actions are legal, what feedback is returned, and what failures are observable?	Simulator, cost model, workload harness, and invalid-configuration checks.
Method roles	Which roles generate, predict, search, critique, verify, call tools, or coordinate?	Candidate generator, surrogate or search method, verifier, evidence writer, coordinator, and human reviewer.
Feedback, evidence ledger, and human decision	What supports the claim, what can reject it, and what remains an architect-owned commitment?	Pareto evidence, sensitivity checks, failure records, rejection authority, and final architectural judgment.

This checklist is intentionally stricter than many current demonstrations. A system can be a useful demonstration while still not being a credible Architecture 2.0 loop; unanswered rows identify where a method lacks action bounds, evidence standards, rejection authority, or a human commit gate.

The practical artifact is the design-loop card introduced later in this chapter and expanded in Appendix B. The card is not a new concept on top of the framework. It is the same

framework compressed into a reusable review object, with the five framework pieces serving as grouped views of the card fields. Appendix B also gives the card a machine-checkable schema and four conformance levels, from context-only to an independently rejectable loop, so “we used the card” becomes a claim a tool and a reviewer can check rather than a gesture.

Beyond structuring the review process, the checklist also keeps the vocabulary from drifting into generic AI language. Words such as state, action, observation, environment, reward, and critic are useful only after they are translated into architecture objects. Table 3.3 gives the translation rule. If a paper says an automated method acts in an environment, the reader should be able to name the architecture state it reads, the action it is allowed to take, the tool feedback it observes, and the authority that can reject the result.

Table 3.3: **AI loop terms need architecture translations:** Architecture 2.0 uses generic loop vocabulary only when each term is grounded in concrete hardware/software design objects, tool outputs, and rejection mechanisms.

Generic term	Architecture translation	Example artifacts or observations	What can reject it
State	Workload, design, software, tool, constraint, and evidence state.	Traces, configs, RTL, compiler IR, simulator stats, EDA reports, review notes.	Missing provenance or hidden assumptions.
Action	Legal architecture, compiler, runtime, or tool-flow change.	Change cache size, vector width, mapping, schedule, constraint, partition, or test.	Invalid parameter, noncompilable code, nonsynthesizable RTL, or policy violation.
Observation	Feedback returned by a tool, benchmark, review, or deployment path.	Latency, energy, area, timing, congestion, warnings, failures, telemetry.	Wrong workload, stale tool version, simulator mismatch, or weak fidelity.
Environment	Tool-connected harness that defines legal actions and feedback.	Simulator wrapper, compiler pipeline, RTL flow, EDA stage, benchmark harness.	Unmodeled constraints, nondeterminism, incomplete logging, or invalid actions.
Objective	Explicit architecture tradeoff, not a generic reward.	Performance, power, and area; tail latency; power envelope; reliability; carbon; cost; evidence budget.	Proxy gaming, lost Pareto tradeoff, or missing human decision rule.

Generic term	Architecture translation	Example artifacts or observations	What can reject it
Critic/verifier	Independent check that can challenge or reject a claim.	Tests, formal checks, baseline replay, cross-simulator comparison, signoff review.	Unsupported claim, failed check, counterexample, or insufficient evidence.

The five pieces are not a pipeline that runs once. They form a loop. A failed simulation may revise the representation. A weak benchmark result may revise the task. A provenance problem may invalidate the evidence. A human rejection may change the environment, not merely reject a candidate. Architecture 2.0 is therefore not only about adding AI into existing work. It is about designing the loop so that AI participation is bounded, observable, and accountable.

Architect's checkpoint: The Loop Revision Gate

When an AI method fails or produces an invalid result, the architect must make a decision at the loop boundary. Does the failure simply reject the candidate, or does it require revising the environment, design space, or evidence standard to keep the method bounded?

3.6 Autonomy Is Earned, Not Declared

The first stress test for the framework is autonomy. Evaluating Architecture 2.0 systems simply as autonomous or non-autonomous lacks sufficient nuance. Autonomy is not a personality trait of a model. It is a property of a bounded loop, and broader autonomy must be earned by stronger evidence.

Figure 3.4 shows four stages of allowed loop authority. The point is not that the automated system gradually replaces the human architect. The point is that each stage grants the AI participant a larger role only when the loop also defines the allowed action space, feedback budget, evidence standard, rollback or escalation path, and architect-owned commitment boundary. The human and the automated system are both visible because Architecture 2.0 is a shared loop with asymmetric responsibility. The AI participant may act inside the loop, but the architect owns the boundary conditions. The figure draws one automated participant to keep the contract legible. The same autonomy test applies when the implementation uses several AI-assisted systems. Every generated proposal, tool call, critique, repair, verification, or coordination step must still be bound to the loop's state, allowed actions, evidence obligations, rejection path, and architect-owned commitment boundary.

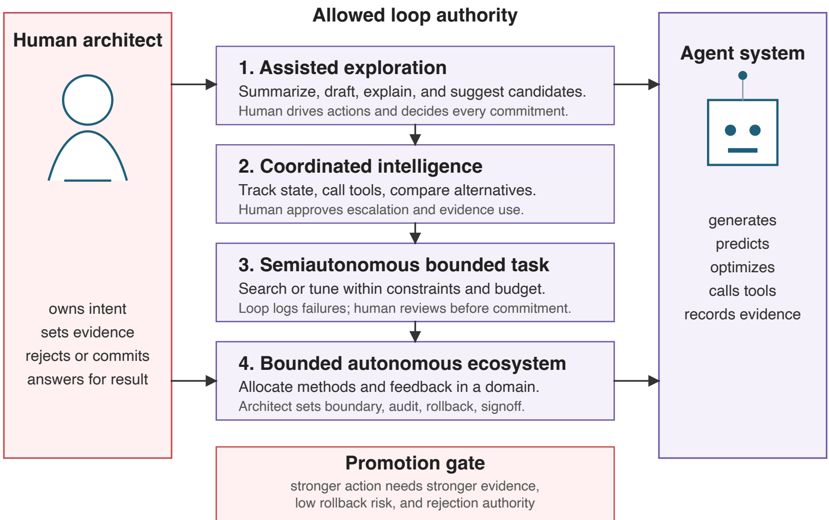


Figure 3.4: **Autonomy is earned by the loop:** Higher autonomy is a property of a bounded loop with explicit actions, feedback, evidence, rollback or escalation paths, rejection authority, and architect-owned commitment boundaries.

At the lowest level, AI systems support assisted exploration. They summarize prior work, draft experiment scripts, explain tool output, suggest candidate parameters, or help prepare design reviews. The architect still directly drives the loop.

At the next level, AI systems provide coordinated intelligence. A model or agent can call tools, track state, propose alternatives, compare candidates, and route work among specialized components. The loop becomes more explicit, but human approval remains frequent.

At a higher level, semiautonomous human-in-the-loop systems can perform bounded subtasks: search a design space, tune a configuration, generate a benchmark variant, build a surrogate, or identify invalid candidates. These systems need clear action spaces, feedback budgets, logging, and rejection authority.

The strongest level is bounded autonomous ecosystems. Here, agents can adapt parts of the loop, choose among methods, allocate feedback budget, and revise representations within a constrained domain. Even then, autonomy is bounded by commitment cost, evidence standards, and human accountability.

The stage of autonomy depends on architecture-specific risk. A compiler flag that can be rolled back after telemetry is not the same as an RTL change that affects timing closure. A simulator configuration is not the same as a mask-level choice. A benchmark-generation

loop is not the same as a signoff loop. The more irreversible the action, the stronger the evidence and rejection authority must be.

Architect's checkpoint

Before granting an AI-driven loop a higher autonomy stage, confirm it defines:
- the allowed action space, and which actions are illegal;
- the feedback budget and evidence standard the stage requires;
- the rollback or escalation path when a candidate fails;
- the rejection authority that can still say no;
- the architect-owned commitment boundary the automated participant may not cross.

If any of these is missing, the autonomy is declared, not earned.

With that stress test in place, the rest of the chapter walks through the framework pieces in order.

3.7 Intent Defines the Task

Architecture tasks do not appear naturally. They are carved out of messy intent. A product goal such as "improve mobile XR efficiency" is not yet a task. It must be translated into bounded work: characterize the workload, choose a candidate ISA extension, compare vector and accelerator organizations, estimate memory traffic, build a power model, explore clock and voltage points, evaluate compiler support, or prepare a design-space report.

This translation is architectural judgment. It decides what is in scope, what is out of scope, what can be measured, and what cost of being wrong is acceptable. It also decides how ambitious an AI-assisted loop can be. A loop that critiques a design report needs different state and evidence than a loop that edits RTL. A loop that predicts energy needs different calibration than a loop that generates workload questions. A loop that searches an accelerator tiling space needs different invalid-action semantics than a loop that proposes chiplet partitionings.

To ground this translation of intent into tasks, this book treats several task families as recurring: design-space exploration, workload characterization, generation, prediction, optimization, critique, verification, and benchmark construction. The list is not meant to be exhaustive. Its purpose is to remind the reader that "use AI" is never a task. The task must be bounded before the method can be chosen, and each family becomes an Architecture 2.0 task only after its action space, feedback budget, evidence standard, rejection rule, and owner are named. These are not content categories; they are recurring loop shapes that differ in represented state, legal actions, feedback cost, invalid-action semantics, escalation rules, and ownership.

3.8 Representations and World Models

Representation is the first hard problem because it determines what the loop can see. Architecture knowledge lives in many forms: natural-language specifications, ISA documents, traces, graphs, simulator configurations, RTL, compiler IR, EDA reports, design reviews, benchmark metadata, spreadsheets, scripts, and plots. Much of the most important state is implicit. It may live in default flags, workload selection, tuned scripts, undocumented assumptions, or the memory of the architect who knows why one experiment was abandoned.

AI systems are brittle around hidden state. If a constraint is not represented, the automated optimizer may violate it. If a simulator option is undocumented, a result may not be replayable. If rejected candidates are missing, a method may relearn known failures. If benchmark provenance is unclear, a comparison may be misleading.

A world model is different from a representation. The representation says what is encoded. The world model says what the loop believes will happen when an action is taken. A simulator embodies one kind of world model. A learned surrogate embodies another. A set of design rules, expert heuristics, or calibrated equations can also function as a world model. None is automatically true. Each has a scope, fidelity, uncertainty, and failure mode.

Distinguishing between representations and world models helps clarify what we are testing. For example, QuArch, an architecture-centric benchmark dataset, is useful as a boundary test. It can assess whether a model knows architecture concepts, but not whether an automated participant has the represented workload, tool, action, evidence, and rejection state needed to synthesize or reject a candidate (Prakash et al., 2025b). That distinction is the bridge to Chapter 4.

3.9 Tools Become Environments

Architecture tools become Architecture 2.0 environments when they define how an automated participant or method can act. A simulator, compiler, profiler, RTL tool, EDA flow, runtime system, or fleet telemetry pipeline does more than return a number. It defines which actions are legal, how long feedback takes, what observations are available, what costs are incurred, what provenance is recorded, and what failure means.

Figure 3.5 turns the distinction into a visual test. A wrapper connects a method to a tool, while an environment exposes the contract that lets the loop audit action, feedback, failure, and rejection.

This is why wrapping tools is not mere engineering plumbing. The wrapper defines the research question. If the action space permits invalid configurations, the loop needs invalid-action semantics. If the observation schema hides memory traffic, the loop cannot reason about data movement. If the reward combines performance and energy without preserving the separate components, the method may optimize a proxy that the architect

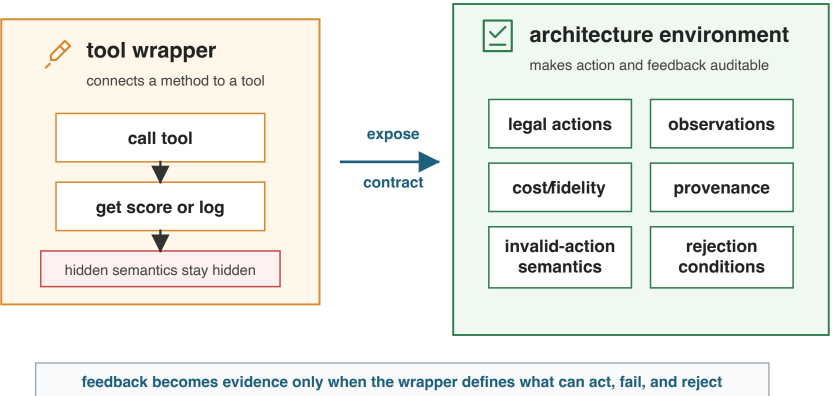


Figure 3.5: **Tools become environments when the wrapper exposes a contract:** A tool wrapper is useful only when it returns more than a score. It must define legal actions, observations, cost and fidelity, provenance, invalid-action semantics, and rejection conditions so feedback can become architecture evidence.

cannot audit. If the environment does not log tool versions, seeds, workload revisions, and failed runs, the feedback may not become evidence.

Building on this need for principled tool wrappers, ArchGym, an open-source evaluation environment, is an important example because it treats the connection between search algorithms and architecture simulators as a first-class interface (Krishnan et al., 2023). Its durable lesson is that a simulator wrapper becomes architectural only when it exposes legal actions, invalid-action semantics, feedback cost, provenance, comparable baselines, and rejection conditions. Chapter 5 expands this point and asks what such environments can and cannot prove.

3.10 Agents and Methods Have Roles in a Compound System

The word "agent" can hide too much. In credible Architecture 2.0 systems, there may be several roles rather than one monolithic actor. A generator proposes candidates. A predictor estimates behavior before expensive evaluation. An optimizer chooses what to try next. A critic challenges assumptions. A verifier checks constraints. A planner decomposes work. A tool caller executes actions. A coordinator tracks state, provenance, and dependencies. A human architect sets intent and decides what evidence is enough.

These roles can be implemented by different mechanisms. A language model may draft an architecture description or critique a result. Bayesian optimization may choose the next candidate. Reinforcement learning may learn a policy for a bounded environment.

A surrogate model may estimate energy or latency. A formal tool may reject invalid behavior. A script may maintain the experiment ledger. The critical evaluation metric is not which method is fashionable, but rather the specific role the method plays in the loop, the state it consumes, the action it takes, the feedback it receives, and the evidence that can reject its output.

This role-based view is also more faithful to architecture practice. Even before AI systems entered the discussion, architects already worked through compound systems: simulators, models, scripts, profilers, spreadsheets, benchmarks, reviews, and signoff processes. Architecture 2.0 makes that compound structure explicit and asks where AI systems can participate without erasing accountability.

3.11 Feedback Becomes Evidence

Feedback is not evidence by default. A simulator result, benchmark score, synthesis report, generated explanation, or model confidence value is feedback. It becomes evidence only when it is tied to a claim, a decision, and a provenance trail.

Lighthouse prompt: A 3 W claim needs evidence, not a number

Context. A reported power result is feedback. It becomes evidence only when the loop records enough state for a reviewer to judge what the number means.
In the Lighthouse prompt. If a generative method claims that a "vector-capable CPU, accelerator, or SoC block" for the "XRBench-class real-time mobile XR workload" meets the "3 W TDP target in a 3 nm-class LP mobile process," the loop must record the workload, input distribution, memory traffic, power model, process assumptions, compiled software stack, rejected alternatives, uncertainty, and rejection rule.
Evidence rule. The same power number has different authority as a proxy estimate, cycle-level simulation, synthesis result, post-layout estimate, or silicon measurement. A proxy estimate may support exploration; post-layout or silicon evidence is needed before stronger implementation or deployment commitments.
Takeaway. The design-space report must say which evidence level supports the claim and which commitment boundary it has not crossed.

Evidence also includes negative information. Rejected candidates, failed simulator runs, invalid configurations, proxy wins that disappear at higher fidelity, and assumptions that had to be abandoned are not waste. They are architecture data. They tell the loop where not to go and tell the human reviewer why a surviving candidate deserves attention.

This distinction between feedback and evidence is one of the main safeguards against hype. Architecture 2.0 is not credible because a generative method can produce outputs quickly. It is credible only when the loop can explain why an output should be believed, what evidence would overturn it, and who has authority to say no.

Architect's checkpoint: The Feedback vs. Evidence Gate

Before committing a model-generated candidate, the architect must evaluate the decision gate: - Does the loop provide a durable evidence ledger, or just a feedback score? - What specific evidence would overturn this claim? - Who holds the authority to say no? If these are undefined, the AI output cannot pass the commitment boundary.

3.12 The Design-Loop Card

The ontology becomes operational through a design-loop card. The card is the practical payload of the ontology, a compact way to describe a paper, project, tool, benchmark, or internal loop. It asks for the loop, not only the result.

Figure 3.6 shows a compact example for the lighthouse prompt. The point is not that the card completes the design. The point is that it exposes the state a credible loop must carry before any generated candidate should be trusted.

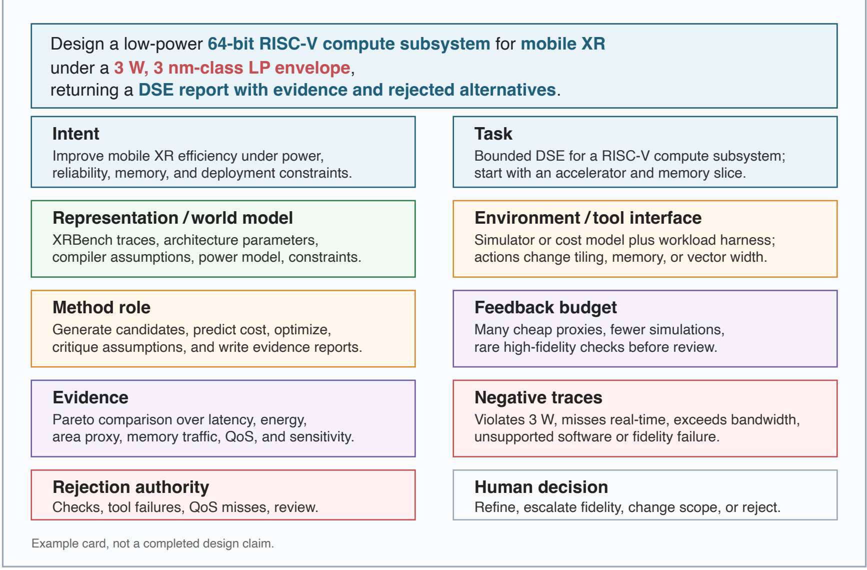


Figure 3.6: **A filled design-loop card turns a prompt into reviewable state:** The lighthouse prompt becomes explicit loop state: intent, task, design space, representation, environment, method role, feedback budget, evidence, failure records, rejection authority, commitment boundary, and human decision.

The card is deliberately simple. Its purpose is not to create paperwork. Its purpose is to reveal whether a claimed Architecture 2.0 contribution exposes the loop that makes it credible. A paper that reports a better search result but hides its feedback budget, rejected candidates, or environment validity is hard to compare. A tool that produces designs but cannot say what rejects them is hard to trust. A benchmark that measures model accuracy but not architecture-relevant reasoning may be useful, but it should not be mistaken for a complete design-loop evaluation. Missing card fields are review outcomes, not formatting problems: ask for evidence, escalate fidelity, reject the claim, or narrow the commitment until the loop can support what it reports.

Appendix B gives the full 12-field card and review rubric. The important point here is that every major Architecture 2.0 claim should be able to expose its loop.

3.13 How the Rest of the Book Uses the Ontology

The remaining chapters unpack the ontology as missing conditions for credible AI participation. Chapter 4 asks what architecture data must encode before AI systems can reason about it. Chapter 5 asks how simulators, compilers, EDA flows, benchmarks, and deployment systems become action settings. Chapter 6 asks which method roles are valid under a feedback budget. Chapter 7 asks when feedback becomes evidence strong enough for rejection and commitment. Chapter 8 runs one loop end to end on the lighthouse prompt. Chapter 9 applies the framework across loop patterns in software, architecture DSE, co-design, systems, and high-commitment silicon-facing work. Chapter 10 returns to the architect: what remains nondelegable, what the community must build, and what it would mean for Architecture 2.0 to become a discipline rather than a collection of demonstrations.

3.14 Conclusion

This chapter asked what a design loop must make explicit before an AI output can be accepted or rejected as an architectural claim. The continuity with classical practice is the loop itself, since architecture has always advanced through cycles of abstraction, measurement, feedback, and judgment. The discontinuity is automation, and with it a change in the failure mode, from slow human drift to fast, confident, catastrophic error when verification and rejection go missing.

The ontology answers that question by naming the minimum a claim must expose to be reviewable. A claim has to show the task and state it represents, the environment it acts in, the method roles that produced it, the feedback turned into an evidence ledger, the authority that can reject it, and the commitment boundary that stays human-owned. Stated as a review object rather than a result, a claim can be compared, reproduced, and contested by someone who never ran the loop. The design-loop card is simply that object made routine.

The ontology guarantees no correctness, and that is the point. It does not pick winning models or tools. It exposes what must be represented, measured, checked, rejected, and decided, which is exactly why it can outlast whichever models and tools are current. A claim that cannot expose the loop that produced it is not yet an architectural claim, however good its numbers look.

3.15 Open Research Questions

The ontology and framework proposed here map the known requirements for Architecture 2.0, but establishing them as standard practice exposes several unsettled research directions. The following open questions push beyond the current conceptualization to explore how these loops can be formalized, enforced, and scaled into rigorous, thesis-level challenges:

1. **How can the design-loop card be elevated from a static review checklist into an executable, zero-knowledge verifiable schema for multi-organizational architecture verification?** While the design-loop card (Appendix B) provides a structured review object, establishing it as a universal protocol requires mechanisms for automated systems to cryptographically prove that a candidate survived the exact claimed environment. A thesis-level challenge is designing executable schemas where constraints, baseline comparisons, and evidence ledgers can be independently verified without exposing proprietary RTL or toolchains.
2. **What are the foundational limits of wrapping legacy, opaque simulation infrastructure into formal Architecture 2.0 environments?** Retrofitting decades of tacit, black-box EDA tools and simulators into formal environments (see the discussion on "Tools as Architecture Environments" in Chapter 5) exposes severe friction. Research must determine how to mathematically bound the invalid-action semantics of these legacy tools so that high-speed generative optimizers cannot silently exploit abstraction inaccuracies to fabricate false Pareto frontiers.
3. **How can the "dark matter" of architectural exploration—rejected candidates, failed timing closures, and proxy-fidelity mismatches—be formalized into a causal world model?** Current AI methods often discard the failure traces that teach human architects the true boundaries of a design space. A major open question is how to continuously extract and encode these negative paths into a rigorous representation of architecture state (see the discussion on "The Architecture 2.0 Ontology" in Section 3.4) that actively regularizes generative models against out-of-bounds hallucinations[8] and causal errors.
4. **Can formal, mechanically-checked protocols enforce human-in-the-loop commitment boundaries in high-velocity autonomous design ecosystems?** As automated loops scale toward bounded autonomous ecosystems (Figure 3.4), policy-based rejection becomes insufficient. A critical systems challenge is engineering strict, verifiable commit-gate protocols that isolate generation from deployment, ensuring

[8] Instances where an AI model generates false, fabricated, or nonsensical information confidently.

that no architectural claim can cross a fidelity boundary without unforgeable, explicit human authorization.

What to carry forward

- **Reader test:** Can you review an AI-generated architectural claim through the loop that produced it: its environment, method roles, evidence ledger, rejection authority, and commitment boundary?
- **Up next:** The next chapter asks what representations and world models must encode before an AI-assisted loop can synthesize systems credibly.

Chapter 4

Representations and World Models

"The limits of my language mean the limits of my world."
— Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1922)

The crux

What must architecture data record before an automated optimizer can act on it and a human architect can audit the result?

Author's Note: Ludwig Wittgenstein, one of the most important philosophers of the 20th century, argued that language constrains what we can possibly conceive. For us, this means an AI agent can only optimize what it can represent; if a hardware description language lacks the vocabulary for a novel concept, the generative AI remains completely blind to it.

Chapter 3 defined the ontology. This chapter deepens representation and world model, the piece that determines what every later piece can see. That ordering is deliberate. It is tempting to begin Architecture 2.0 by asking which model, agent framework, or optimization method to use. But a method can only act on what the loop can represent. *If the relevant state is invisible, a more capable model may simply move faster in the wrong direction.*

Agent framework: A software structure that defines how autonomous systems perceive their environment, reason, and take actions to achieve goals.

Fundamentally, this is a low-resource loop-state problem. The scarce resource is not examples in the abstract, but records that bind intent, workload, tool state, actions tried, failures, accepted evidence, and architect decisions. Architecture does not need data in the same way a web language model[9] needs text. The durable question is therefore not only how to collect more architecture data. It is how to represent scarce architecture data so that a loop can act on it and a human can audit it. The architectural risk is not just small data; it is that a generative method may infer authority from public text where the actionable state was never recorded.

[9] A statistical model trained to predict the next word or token in a sequence, forming the foundation of modern AI text generation.

What this chapter gives you

After this chapter you can turn an architecture prompt, result, or artifact into an actionable loop-state record for automation. That means you can:

- explain why architecture data is not web data: sparse, tool-bound, and full of state the final paper omits;
- treat a feedback event as a sample that carries cost, fidelity, and provenance;
- spot undocumented design state and the missing negative traces in an AI-assisted loop;
- identify when the human architect must accept evidence, reject candidates, escalate fidelity, and own residual risk;

- tell the optimizer's architecture representation apart from its world model.

4.1 Why Architecture Data Is Not Web Data

To understand why this loop-state record is necessary, we must first contrast architecture with domains that have thrived on scraped data. Many successful AI systems benefit from abundant public text, images, code, logs, or interaction traces. Architecture work is different. Some useful material is public: papers, textbooks, manuals, open-source tools, benchmark descriptions, and selected design artifacts. But much of the state that makes an architecture decision meaningful is not public, not standardized, and not preserved in the final paper. Some of it is tacit knowledge: the design-review habit that recognizes a fragile assumption, the instinct that a simulator result is outside its calibrated regime, the memory of why a workload slice was excluded, or the experience that a supposedly local change will become a verification problem later.

The missing state matters. Workload traces may be proprietary or too large to share. Simulator configurations may live in scripts rather than in the paper. EDA reports may be confidential or tied to licensed process assumptions. Labels may require expert judgment. Negative results are rarely published. High-fidelity measurements may be slow, expensive, or unavailable until late in the design process. The cost of a wrong action is also different. A weak answer in a question-answering task may be corrected immediately. A weak architecture proposal can consume weeks of simulation, mislead a design review, or push effort toward a candidate that cannot survive synthesis, timing, power, or software integration.

Lighthouse prompt: The AI prompt is not the loop state

Context. This chapter's representation question is visible in the first noun phrase of the lighthouse prompt. A user sentence can name a workload and technology target, but it does not yet represent the design state the loop can act on.
In the Lighthouse prompt. "64-bit RISC-V-based compute subsystem" and "XRBench-class real-time mobile XR workload" cannot be answered from public text alone. XRBench gives a workload anchor (Kwon et al., 2023), and RISC-V gives a software-contract anchor.
Representation. The AI-assisted loop must also record the scenario, input distribution, frame deadline, candidate compute organization, SoC boundary, software stack, ISA and ABI assumptions, compiler choices, memory behavior, power model, process assumptions, verification and reliability status, deployment context, and rejected alternatives.
Takeaway. Treat the user's prompt as an index into missing loop state. An automated design-space report is credible only when the representation carries enough provenance, coverage, and negative traces for a human architect to audit.

This is why internet-scale recipes transfer poorly if they are used naively. Architecture data is sparse, expensive, tool-bound, and full of hidden constraints. Table 4.1 makes the contrast explicit.

Table 4.1: **Architecture data breaks the web-data assumptions:** Internet-scale recipes fail in architecture because the data regime differs fundamentally in scale, structure, cost, and visibility.

Dimension	**Web Data (Internet-Scale AI)**	**Architecture Data (Architecture 2.0)**
Abundance	Hundreds of trillions of tokens.	Sparse; bound by slow simulation and engineering time.
Format	Mostly unstructured text, code, and images.	Heavily state-based: tool flags, workloads, constraints, negative traces.
Visibility	Publicly accessible (scraped corpora).	Often proprietary, tacit, or lost after publication.
Feedback Cost	Cheap (e.g., fast automated grading or immediate user correction).	Expensive (hours/days of cycle-level simulation or physical synthesis).
Cost of Failure	Low (a poor chat response is easily ignored).	High (a poor design choice wastes compute budgets or fails timing).

The right response is not despair. It is representation design. Make the state explicit enough that methods can act within the boundaries of what is known, what is assumed, and what must still be checked. Representation design has two halves: the representation that records that state, and the world model that predicts what acting on it will do.

Architecture representation. An architecture representation structures the loop into three state schemas that a method can read, compare, change, replay, and audit:

- **Loop State:** Workload, design state, and constraints (the system inputs/outputs defining the *state seen* and *actions allowed*).
- **Execution History:** Provenance, feedback, and negative traces (the runtime telemetry proving *why* the loop navigated there, mapping to *evidence* and *alternatives rejected*).
- **Meta-State:** Assumptions, uncertainty, and tool state (the boundary conditions of the loop itself).

This representation records what the loop knows, but the loop also needs a mechanism to predict the outcome of its actions.

Architecture world model. An architecture world model is a *predictive design-space surrogate* that estimates physical reality. It embodies the loop's belief about what will happen when an architecture action is taken: what a simulator, surrogate, rule system, or constraint model predicts, permits, rejects, or leaves uncertain. Human review is not part of the world model; it is an external rejection authority that acts on the model's output.

The boundary test in Table 4.2 is simple. A representation records the state a loop can read and replay. A world model predicts or constrains the consequences of acting on that state. The same artifact can contribute to both, but the loop must know which role it is playing.

Table 4.2: **Representation records state; world models predict consequences:** The loop needs both, and it must not confuse replayable artifacts with evidence that an action will remain valid under new workloads, tools, or fidelity levels.

Example	Representation role	World-model role
Simulator configuration	Records flags, model version, workload slice, seed, and command.	Defines what behavior the simulator can predict and where its calibration fails.
Candidate parameter table	Records legal choices and candidate values.	Feeds a surrogate or rule system that predicts latency, energy, area, or invalidity.
Constraint file	Records declared limits and assumptions.	Rejects actions that violate timing, power, interface, or policy boundaries.
Failed run log	Records what happened, with provenance.	Updates the loop's belief about invalid regions and escalation rules.

To see why we cannot simply extract this state from internet-scale text, the token count makes the contrast sharper. A broad corpus of public computer-architecture knowledge is only on the order of 10^9 tokens. The architecture-relevant subset of the Wikipedia corpus sits at that scale, and a back-of-the-envelope cross-check from roughly fifty years of the literature agrees, at about 100,000 papers of 10,000 tokens each:

$$T_{\text{arch-text}} \approx N_{\text{artifacts}} \times \bar{t}_{\text{artifact}} \approx 10^5 \times 10^4 \approx 10^9 \text{ tokens.}$$

That sounds large, but it is small by web-scale pretraining standards. More importantly, it is incomplete in the wrong way. Papers preserve accepted claims far better than failed configurations, simulator flags, workload revisions, EDA reports, review arguments, and rejected alternatives. Architecture 2.0 therefore cannot treat "all architecture text" as the dataset. The dataset must include the loop state that made the text credible, and it must expose enough tacit judgment to make assumptions, exclusions, and rejection decisions inspectable. The token count matters only because it limits what a generative method can infer about action authority. Text may retrieve concepts, but it cannot recover which candidate was legal, what feedback was observed, why alternatives were rejected, or who accepted the risk.

Pretraining: The computationally intensive initial phase of training a large machine learning model on a massive, general dataset before fine-tuning it for specific tasks.

Both are order-of-magnitude anchors, not a measured corpus inventory. The transparent literature calculation and the architecture-relevant Wikipedia corpus land at the same 10^9-token scale. Table 4.3 shows why the exact token count is not the main claim. The important scarcity is that the most useful architecture data are often not public, not textual, or not preserved as reusable loop state.

Table 4.3: **Architecture data scarcity is about missing loop state, not only token count:** Public text is useful, but the decisive records are often provenance, negative traces, tool settings, and commitment decisions.

Data layer	Rough public visibility	What is missing for Architecture 2.0
Published papers, manuals, and tutorials	Public and mostly textual.	Failed runs, rejected alternatives, exact workload slices, scripts, seeds, and review rationale.
Open-source RTL, simulators, compilers, and benchmarks	Public but heterogeneous and version-sensitive.	Tool settings, run provenance, invalid configurations, and cross-tool disagreement.
Commercial EDA reports, PDK assumptions, and signoff context	Often private or redacted.	Timing, power, physical, waiver, and closure evidence tied to the claim.
Design reviews and engineer memory	Usually tacit.	Why an experiment was abandoned, which proxy was distrusted, and who accepted risk.

To visualize just how small the public-text estimate is, Figure 4.1 puts the architecture token count on the same log-scale axis as three public AI-data anchors: the 1.4 trillion tokens used to train DeepMind's Chinchilla model, Meta's report that Llama 3.1 405B was trained on more than 15 trillion tokens, and a public human-text-stock estimate on the order of hundreds of trillions of tokens (Hoffmann et al., 2022; Meta AI, 2024; Villalobos et al., 2024). The point is not that architecture should race to scrape the web. The point is that even a generous public architecture-text estimate is tiny by modern pretraining standards, while the most important missing records are not ordinary text at all.

Table 4.4 makes the receipt explicit. It is not a measured corpus inventory. It is a scale check that separates the easy counts from the hard missing records. The multiplication that gives 10^9 tokens is deliberately simple; the point is that even a generous public-text corpus is small, and the public surfaces we can count most easily are not the same thing as architecture loop state. Treat the counts themselves as a release-time snapshot, not a durable benchmark.

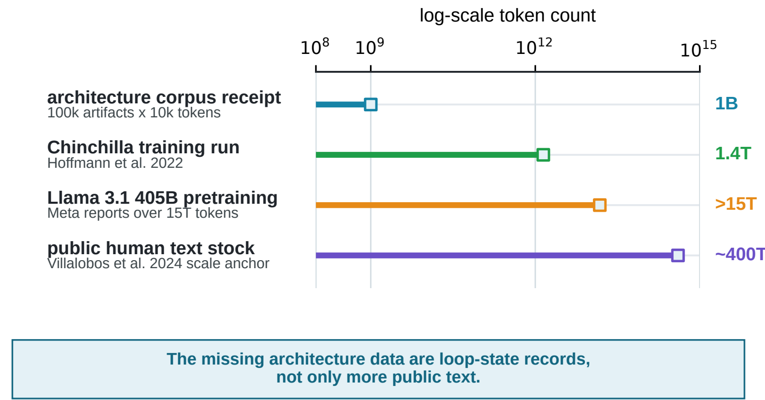


Figure 4.1: **Architecture text is small by web-scale standards:** The missing architecture data are reusable loop-state records, not just more public text.

Table 4.4: **The architecture-data receipt is an assumption log, not a corpus claim:** The current (10^9)-token estimate comes from transparent back-of-the-envelope assumptions, while the available mining signals are artifact proxies. GitHub counts are single Search API snapshots and drift over time.

Receipt	Current value	What it supports	Caveat
Public architecture, systems, and EDA artifacts	10^5 paper-equivalent artifacts	Order-of-magnitude basis for the 10^9-token scale check.	Assumption for intuition; not a measured corpus boundary.
Tokens per paper-equivalent artifact	10^4 tokens per artifact	Transparent multiplication: $10^5 \times 10^4 = 10^9$.	Tokenization and artifact length vary; manuals and specifications can be much larger.
DBLP title pilot	1,015 title records from selected ISCA, MICRO, HPCA, and ASPLOS years	Shows that public metadata is easy to collect and useful for trajectory signals.	Title-only pilot; not full text, artifacts, tool state, or design-loop evidence.
GitHub RTL language proxy	141,288 Verilog repositories; 43,371 SystemVerilog repositories	Shows a large public RTL-adjacent surface that could seed artifact mining.	Broad language counts include small, toy, forked, stale, and non-architecture repositories.

Receipt	Current value	What it supports	Caveat
GitHub RTL keyword proxy	4,292 Verilog+“rtl” repositories; 1,657 SystemVerilog+“rtl” repositories	Gives a narrower proxy for RTL-oriented repositories.	Keyword-sensitive and unvalidated; it is still not a count of usable architecture design examples.
GitHub topic proxy	353 computer-architecture-tagged Verilog repositories; 1,817 FPGA-tagged Verilog repositories	Shows that curated labels are much smaller than broad language counts.	Topic labels are voluntary, incomplete, and uneven across projects.
Missing loop-state records	Not counted	The highest-value architecture data are traces, configs, logs, reports, reviews, negative results, and rejected candidates.	Much of this state is private, tacit, uncodified, or discarded before publication.

The useful lesson from the receipt is the mismatch. A title corpus can help map topic drift, but it cannot recover simulator flags, failed configurations, or review arguments. A Verilog repository count can show that public RTL-like material exists, but it does not say whether the repository contains a well-specified architecture decision, a reusable testbench, a valid workload, or evidence that rejected alternatives were considered. Architecture 2.0 therefore needs corpus building only when artifacts are converted into loop records: candidate, workload, tool version, legal and invalid actions, feedback, provenance, rejected alternatives, and decision owner.

Domain adaptation matters here only if it helps construct or retrieve loop records; vocabulary and retrieval gains do not grant an automated optimizer action authority. The medical AI lineage is a useful comparison because it shows both the power and the limit of domain adaptation. BioBERT adapted a general language model to biomedical text; ClinicalBERT adapted representations to clinical notes; Med-BERT adapted BERT-style pretraining to structured electronic health records rather than ordinary prose (Lee et al., 2020; Huang et al., 2019; Rasmy et al., 2021). Those systems mattered because they treated the domain’s data format as a first-order problem. Chip design has its own instance. ChipNeMo adapts language models to hardware-design text with a custom tokenizer, domain-continued pretraining, and retrieval over internal corpora, then applies them to engineering question-answering, electronic-design-automation script drafting, and bug-report summarization (Liu et al., 2023). It demonstrates that domain adaptation can improve text-facing hardware tasks, and it illustrates the limit this chapter presses. It maps text to text, so it addresses the data-ingestion problem, not action authority in a closed design loop. Architecture must do the same domain work, but the representation

burden is broader. A compute-subsystem design loop needs not only domain terms, but also executable tool state, workload provenance, constraints, multi-fidelity feedback, rejected alternatives, and decision authority. A paper corpus can bootstrap knowledge; it cannot by itself represent the design loop.

Domain adaptation: A machine learning technique where a model trained on one data distribution is adjusted to perform well on a different, specific target domain (Pan and Yang, 2010).

Domain-continued pretraining: Further training a foundational model on a large corpus of domain-specific text to improve its specialized vocabulary and understanding.

Moving from training text to evaluation, benchmark lineages such as SQuAD and GLUE offer a related lesson. Shared, scoreable examples can move a whole field. The Architecture 2.0 lesson is not to seek a cheap labeled row, but to ask what a shared evaluation object must record before an automated optimizer can act. Architecture needs shared evaluation objects too, but their purpose is not only scoring models; it is constraining what an automated optimizer may try next, what evidence can update the loop, and what a human reviewer can reject. An architecture "example" is rarely just a cheap labeled row. It may be a simulator run, a synthesis run, a physical-design report, a workload trace, a failed configuration, or an expert review tied to a specific fidelity level. The cost of a sample is therefore part of the representation problem, not an afterthought.

SQuAD and GLUE: Standard benchmark datasets used to evaluate the performance of natural language processing models, where SQuAD is the Stanford Question Answering Dataset (Rajpurkar et al., 2016) and GLUE is the General Language Understanding Evaluation (Wang et al., 2018).

4.2 Sample Cost Is Architecture Data

Because this cost dictates how an optimizer must treat an evaluation point, we have to move away from the web-centric view of a sample. In many benchmark settings, a sample is treated merely as an input-output pair: a question and answer, an image and label, a prompt and reference response. In an architecture design loop, a sample is better understood as a feedback event that changes what the loop believes. It might be a cycle-level simulation, a compiler report, a failed synthesis run, a power estimate, a rejected floorplan, an expert review, or a silicon measurement. Each event has a cost, fidelity, provenance, and commitment level.

Architecture sample. An architecture sample is any feedback event that changes the loop's belief about a design candidate, including its cost, fidelity, provenance, assumptions, rejected-space coverage, and commitment level.

For an automated design loop, sample cost is part of the action policy. It determines which evaluations are cheap enough to explore, which require escalation, and which decisions should remain human-owned.

Architect's checkpoint: The Cost of Fidelity

Before the loop consumes days of compute on a cycle-accurate simulation or physical synthesis run, it must hit a fidelity gate. The architect must own the decision to authorize expensive, high-fidelity samples, balancing the exploration budget against the cost of the feedback event.

A useful representation should therefore record the burden and evidence context of feedback, not only the feedback value. A sample is best written as a vector of incommensurable dimensions, not a single number, since they do not share units:

$$M_{\text{sample}} = \langle C_{\text{setup}}, C_{\text{tool}}, C_{\text{license}}, C_{\text{compute}}, C_{\text{human}}, F_{\text{fidelity}}, P_{\text{provenance}}, K_{\text{coverage}}, C_{\text{opportunity}}, C_{\text{risk}} \rangle.$$

The vector does not collapse into one currency; its dimensions carry different units. It is a reminder that an architecture sample carries hidden state. A simulator point may require setup time, tool availability, license access, calibration work, human triage, provenance, coverage context, and the opportunity cost of not evaluating another candidate. A post-layout result may carry higher fidelity but also higher latency and risk. A field deployment measurement may be authoritative for one population and irrelevant for another.

Concrete architecture tools span this range. Analytical mapping and dataflow models are designed for broad design-space exploration (Parashar et al., 2019; Kwon et al., 2019). Physical-design and verification flows expose the other end of the spectrum, where feedback can take hours or days and the human and engineering cost becomes part of the sample itself (Mirhoseini et al., 2021; Semiconductor Industry Association, 2026; Bauer et al., 2020; Foster, 2022). Table 4.5 is therefore not a tool taxonomy. It is a representation checklist. If the row changes, the loop must record different state.

Table 4.5: **Architecture samples carry cost, fidelity, and commitment:** A feedback event is useful only when the representation records what it cost, what assumptions it used, and what future decision it can support or reject.

Feedback source	Latency / cost intuition	What it exposes	Record for reuse
Analytical model or mapper	Milliseconds to seconds; low direct cost; high model-risk exposure.	Useful for pruning and sensitivity checks, not final evidence.	Model assumptions, workload slice, constraints, and proxy-validity notes.
Trace, profile, or replay	Seconds to hours depending on capture and replay setup.	Workload provenance is part of the sample.	Trace version, sampling policy, software stack, and filtering choices.

Feedback source	Latency / cost intuition	What it exposes	Record for reuse
Cycle-level simulation	Minutes to days depending on model detail and target workload.	Simulator evidence is scoped by abstraction, calibration, and unsupported states.	Simulator version, configuration, seeds, workload revision, and calibration notes.
RTL, gate, or EDA feedback	Hours to days when synthesis, timing closure, power analysis, or physical DRC/LVS feedback enters the loop.	High-fidelity samples are scarce and multiobjective.	Tool versions, constraints, process assumptions, waived warnings, and rejected candidates.
FPGA, emulation, or prototype	High setup and shared-resource cost; high throughput once mapped.	Speed changes observability and debugging semantics, not only wall-clock time.	Mapping constraints, observability limits, debug hooks, and queue/resource state.
Silicon or field telemetry	Weeks to years and high commitment.	Authoritative measurements still require context and human decision authority.	Population, deployment version, rollback policy, incident context, and decision owner.

For tools like simulators, the timing side of this cost is multiplicative. A simulator is not slow in the abstract; it is slow relative to the target cycles the workload demands. For a target clock f_{target}, workload duration T_{workload}, and simulation throughput R_{sim}, the wall time is

$$T_{\text{wall}} = \frac{N_{\text{cycles}}}{R_{\text{sim}}} = \frac{f_{\text{target}} \times T_{\text{workload}}}{R_{\text{sim}}}.$$

Table 4.6 gives the intuition for a 1 GHz target. Read each row as a span of target execution time, each column as a simulator throughput, and each cell as the wall-clock time to simulate that span. One second of a 1 GHz target takes about 11.6 days at 1 kcycle/s but only 10 s at 100 Mcycle/s. The rates are illustrative, but the multiplication is the point. A loop can afford many cheap proxy samples, fewer cycle-level samples, and very few high-fidelity samples unless it has a disciplined plan for escalation and rejection.

Table 4.6: Target cycles turn simulator throughput into wall-clock pressure: even simple workloads become expensive when target execution time is multiplied by target frequency and divided by simulation throughput. Wall-clock values are computed from the cycle count and rate.

Target workload at 1 GHz	Target cycles	1 kcycle/s	100 kcycle/s	10 Mcycle/s	100 Mcycle/s
1 ms	10^6	16.7 min	10 s	0.1 s	0.01 s
1 s	10^9	11.6 days	2.8 h	1.7 min	10 s
1 min	6×10^{10}	1.9 years	6.9 days	1.7 h	10 min
1 h	3.6×10^{12}	114 years	1.1 years	4.2 days	10 h

This cost structure changes how we should think about design spaces. A small co-design exercise with five workload slices, four architecture configurations, and six compiler or mapping settings has $5 \times 4 \times 6 = 120$ candidates and can sometimes be enumerated. A physical-design, compiler, mapping, or chiplet-integration space may be combinatorial, sequential, tool-bound, and partially invalid. Learning-assisted chip placement makes the point concrete. The design problem can be formulated as a learning problem, but the value of each sample depends on the representation of macros, nets, constraints, tool feedback, and placement validity (Mirhoseini et al., 2021); that result was later contested on its baselines and reproducibility (Chapter 7) (Cheng et al., 2023b), which only sharpens the point. The architecture lesson is broader than placement. When samples are expensive, the loop record must capture what each sample cost, what region of the space it informs, what it rules out, and how it should constrain the next optimizer action.

Chapter 6 returns to sample efficiency from the method side. The representation lesson comes first. If cost, fidelity, and rejected-space coverage are not recorded, a later optimizer cannot know whether it is learning the design space or merely collecting disconnected measurements.

4.3 Architecture Descriptions as Boundary Objects

All of that recorded state has to live somewhere the loop and the architect can both read, and that artifact, the architecture description, is a boundary object. In the sense introduced by Star and Griesemer, a boundary object is shared across communities while still supporting their different local uses (Star and Griesemer, 1989). It sits between human intent and tool action. It must be readable enough for architects to inspect, precise enough for tools to execute, and structured enough for automated optimizers or methods to modify without breaking the design contract.

At minimum, an architecture description should make the action contract explicit: what is being described, what can change, what must not change, what evidence can update the record, and which tools can consume it. For a memory hierarchy, this may include cache sizes, associativity, replacement policy, prefetching, coherence assumptions, bandwidth, latency, and workload mix. For an accelerator or compute subsystem, it may include supported operations, data layout, local storage, vector width, dataflow, quantization, compiler/runtime assumptions, and fallback behavior. For an optimizer-facing architecture description, the minimum is not only fields but permissions: which fields are mutable, which are read-only constraints, which tool observations can update them, and which violations trigger rejection or escalation.

The important point is not that every representation must be one universal schema. Different loops need different representations. A paper-reading loop, a simulator-driven design-space exploration loop, an RTL-generation loop, and a post-silicon telemetry loop should not have identical records. But they do need explicit invariants. What fields must be present? Which fields can a method change? Which fields are read-only constraints? Which assumptions travel with a result? Which tool versions, seeds, and workload revisions are required for replay?

Without those boundaries, a representation becomes a prompt-shaped anecdote. It may sound plausible, but it cannot safely drive action.

4.4 Unstructured Design Data and Its Cost

When those boundaries are left implicit, the missing structure does not stay free. Architecture teams accumulate undocumented design state, the same kind of hidden cost that software teams call technical debt when complex data dependencies and implicit assumptions go unmanaged (Sculley et al., 2015). The cost appears whenever important design state exists but is not captured in a durable, inspectable form. It may live in shell scripts, simulator flags, spreadsheet formulas, plotting notebooks, benchmark directories, issue threads, email, review slides, or the memory of the person who knows why one candidate was rejected. Some of this state is tacit rather than textual: what an experienced architect chooses not to try, which proxy result they distrust, which corner case they ask about in review, and which risk they refuse to delegate.

This gap is manageable when a small team manually coordinates the loop. It becomes a technical failure when an AI system acts inside the loop. If a constraint is implicit, the method may violate it. If a simulator flag is hidden, the result may not be replayable. If rejected candidates are missing, the search may rediscover known failures. If workload provenance is unclear, the loop may optimize for the wrong distribution. If plots preserve only the winning candidate, the evidence trail cannot explain why alternatives were discarded. ::: {.callout-field-note title="The result no one could rerun"} Consider AI-generated chip placement. A reinforcement-learning method was reported to place macros as well as or better than human experts, and said to have run on production silicon (Mirhoseini et al., 2021); independent groups then could not reproduce the advantage,

because the code and pretraining inputs needed to rerun it were never released (Cheng et al., 2023a; Markov, 2023), and the authors answered that the critics had not run the method as described (Goldie et al., 2024). Years later the claim can be neither confirmed nor rejected. The lesson is not who is right; it is that a result no one can rerun leaves the loop with an anecdote instead of evidence.

Takeaway. Enforce tool and environment provenance as the run happens, so the loop leaves a replayable receipt rather than a dispute. ::: To show where this missing information typically hides, Table 4.7 gives common sources of undocumented design state. The point is not to document everything for its own sake. The point is to capture enough state that a loop can compare, replay, reject, and revise.

Table 4.7: **Architecture artifacts become reusable loop data when they are representation records:** Each artifact should carry enough provenance, assumptions, constraints, and validity information for a loop to act on it and for a reviewer to audit it.

Artifact	What it enables	What it often hides	Failure mode
Paper or plot	Claim, result, and comparison.	Tool flags, failed candidates, tuning history.	Reproduce only the story, not the loop.
Workload trace	Concrete input behavior and measurements.	Coverage, versioning, sampling policy, privacy filters.	Optimize for an unrepresentative slice.
Simulator config	Replayable model setting.	Defaults, unsupported states, calibration limits.	Trust a number outside its valid scope.
RTL or EDA report	Implementation-facing feedback.	Process assumptions, constraints, waived warnings.	Accept an artifact that cannot close.
Review notes	Human judgment and rationale.	Tacit assumptions and discarded alternatives.	Lose why a decision was made.
Rejected candidate	Search boundary and negative evidence.	Why it failed and at what fidelity.	Rediscover known dead ends.

4.5 QuArch as a Stress Test

The artifacts in Table 4.7 are internal and often unstructured. Public benchmarks sit at the opposite extreme, well structured but paper-bound. QuArch, a question-answering benchmark that turns the architecture literature into a structured, expert-validated evaluation object (Prakash et al., 2025b), is a useful stress test for this chapter precisely because it exposes what that public structure still leaves out. It bridges two representation

layers, architecture knowledge that can be asked about in text and architecture state that must be carried by a loop before an automated optimizer can act. The later QuArch reasoning benchmark makes the same point more explicit by organizing 2,671 expert-validated questions around recall, analysis, design, and implementation competencies (Prakash et al., 2025a). QuArch can ask whether a model recalls concepts, tracks architectural relationships, reasons over published claims, and avoids obvious domain mistakes. That is valuable. A field cannot build credible AI-assisted systems if those systems lack basic architectural knowledge.

Figure 4.2 shows why that achievement is necessary but not sufficient. Paper-derived questions test one layer of architecture data, while action and rejection require loop-state records that papers often omit.

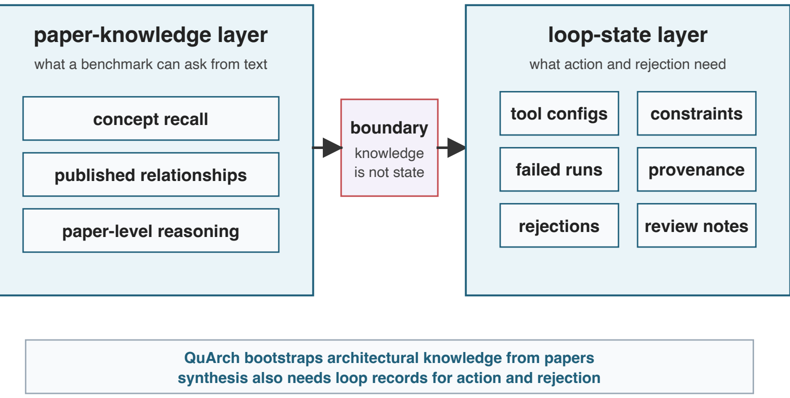


Figure 4.2: **QuArch is a boundary test, not the whole data layer:** Question-answering benchmarks can test architectural knowledge from papers, but Architecture 2.0 also needs loop records: tool configurations, failed runs, constraints, provenance, rejections, and review notes.

As this boundary test highlights, the limit of paper-derived datasets like QuArch is that papers preserve accepted claims far better than they preserve design-loop state. They rarely contain every simulator configuration, rejected candidate, failed run, hidden constraint, calibration choice, or review argument. A model that answers questions over papers may know what a concept means and still lack the state needed to act inside a design loop. It may summarize a memory-system paper, but it does not necessarily know which candidates failed, which simulator flags were decisive, which workload slices were excluded, or which result would cause a human architect to reject the next proposal.

The lesson is not that question-answering datasets are insufficient and therefore unimportant. The lesson is that they occupy one layer. They help represent architectural knowledge.

Architecture 2.0 also needs representations of experiments, tools, constraints, provenance, and negative traces. A reader should see QuArch as one example of bootstrapping the data layer, not as the whole data layer.

4.6 Toward Architecture World Models

If the data layer described above—including both paper-derived knowledge and loop-state records—constitutes the representation, the loop still needs a mechanism to act on it. Representation records what the loop knows; the world model, the second half of the pairing, is what turns that record into predicted consequences. The world-model idea becomes architecture-specific when it is tied to tool behavior, cost, constraints, invalid-action rules, uncertainty, and decision policies, much like reinforcement learning agents use latent world models[10] to plan and simulate outcomes before acting.

[10] Internal, compressed representations of an environment used by an AI to simulate and plan actions without interacting with the real world.

This distinction matters. A simulator configuration is part of the representation. The simulator's behavior, scope, calibration, and failure modes are part of the world model. A table of candidate parameters is representation. A surrogate that predicts latency or energy from those parameters is a world model. A set of design rules, expert heuristics, or physical constraints can also act as a world model.

Figure 4.3 gives the basic structure. A representation record contains workload traces, architecture descriptions, tool configurations, logs, constraints, and objectives. A world model contains state, action spaces, dynamics, costs, constraints, invalid-action rules, uncertainty, and decision policies. Tools return feedback; evidence updates both the representation and the world model.

For the lighthouse prompt, a small world model might be as simple as Table 4.8. It is not a full simulator. It is the part of the loop's belief that says which actions are meaningful, what they are expected to change, and what evidence can overturn the prediction.

Table 4.8: **An architecture world model is a scoped belief about action and consequence:** Even a small sketch should name state, legal actions, predicted outcomes, uncertainty, invalid actions, and escalation triggers.

World-model field	XRBench/RISC-V sketch
State	Workload slice, frame deadline, software path, candidate compute organization, memory traffic, and power envelope.
Legal action	Change vector width, local memory size, CPU/accelerator partition, or data layout inside the declared software contract.
Predicted transition	Estimate latency, memory traffic, energy, area pressure, and compiler/runtime feasibility.
Uncertainty and calibration	State whether the estimate comes from an analytic proxy, calibrated simulator, prior measurements, or expert rule.

World-model field	XRBench/RISC-V sketch
Invalid-action rule	Reject candidates that break software compatibility, exceed the action bounds, fail correctness, or lack provenance.
Escalation trigger	Move to stronger evidence when a candidate nears the 3 W target, misses the frame deadline, or wins only under a weak proxy.

The type of world model matters only through the loop contract it supports: what actions it can evaluate, what uncertainty it reports, what invalid moves it rejects, when it escalates fidelity, and which decisions remain human-owned. A simulator-backed world model uses a tool as the transition and feedback mechanism. A learned surrogate world model predicts outcomes from prior evaluations. A symbolic or constraint-based world model encodes invalid configurations, design rules, or physical limits. A hybrid world model combines these pieces: a simulator for selected candidates, a learned predictor for cheap screening, a rule system for invalid actions, and an escalation rule that routes high-commitment decisions out to human review.

No world model is automatically credible. Each has a scope. Each has uncertainty. Each can be wrong under distribution shift, new workloads, different software stacks, tool changes, or higher-fidelity evaluation. The goal is not to pretend the world model is truth. The goal is to make its assumptions explicit enough that the loop can decide when to trust it, when to escalate fidelity, and when to reject its advice. The lighthouse prompt makes that discipline concrete. An analytic proxy predicts that a wider-vector candidate meets the 3 W envelope, but the margin to the envelope is smaller than the proxy's known error band, so the escalation trigger from Table 4.8 fires. A calibrated simulator then shows the candidate misses the frame deadline on the real XRBench slice, and the architect records the rejection with its fidelity level and reason. The prediction, the escalation, and the rejection are all part of the world model, and each is auditable. An automated optimizer acted on the belief, and a human could see why it was overturned.

Distribution shift: A situation where the data a model encounters during deployment differs significantly from the data it was trained on, often degrading performance (Quinonero-Candela et al., 2009).

Architect's checkpoint: Escalate or Reject

When the world model proposes a candidate whose margin of safety is smaller than the model's known error band, the loop must pause. The architect decides whether to escalate to a higher-fidelity tool (e.g., a cycle-level simulator) or reject the candidate outright based on risk. This gate ensures the optimizer does not commit to decisions based on uncalibrated proxy estimates.

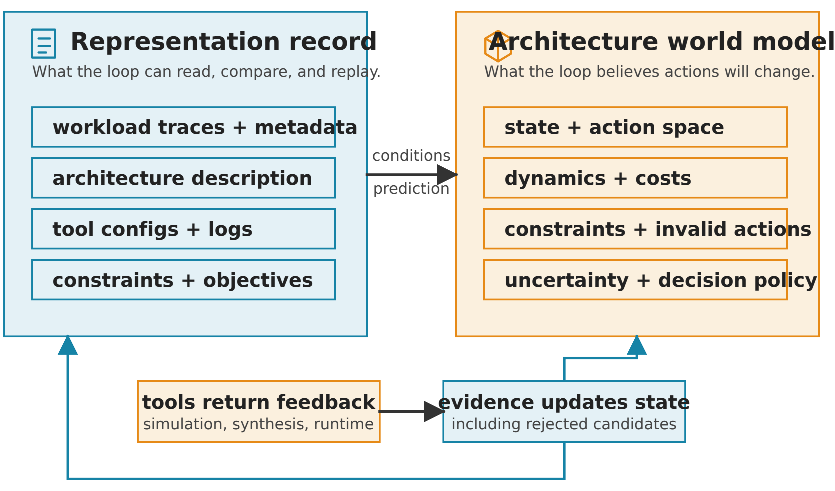


Figure 4.3: **A world model connects artifacts to valid architectural action:** A representation record captures what the loop can read, compare, and replay; a world model captures what the loop believes actions will change. Feedback becomes reusable only when provenance, coverage, and negative traces are recorded.

4.7 Provenance, Coverage, and Negative Traces

Knowing when to reject the world model's advice depends on records most teams are quick to discard. The most distinctive architecture data may be the data the field usually throws away. Published work tends to preserve accepted artifacts. Design loops also need rejected alternatives. A failed simulator run, invalid configuration, proxy win that fails at higher fidelity, abandoned floorplan, or unsupported-software path is not merely noise. It shows the loop where the boundary lies.

> **Venues for negative results:** The wider computing-systems community has begun to build places to publish exactly these records, such as the NOPE workshop (Negative results, Opportunities, Perspectives, and Experiences) held with ASPLOS. The field needs more of them, so that failed runs, rejected candidates, and the assumptions behind them accumulate as shared evidence instead of being discarded before publication.

> **Negative traces.** Negative traces are recorded failed, rejected, invalid, or abandoned candidates together with the reason they did not support the claim. They matter because a loop that records only winners cannot inform the next method, reviewer, or architect which assumptions, tools, workloads, or regions of the design space were already ruled out.

Negative traces matter because architecture action spaces are full of invalid or misleading moves. A generated RTL fragment may be syntactically plausible but fail timing or violate an interface. A design-space search may find a candidate that looks good under a proxy but fails under a better power model. A benchmark result may improve because

the workload slice is too narrow. A chiplet partition may appear modular but introduce unacceptable latency, thermal coupling, test complexity, or supply-chain risk.

Table 4.9 turns this point into a data schema. The purpose is to record what failed, what boundary it exposed, and how that evidence should change the next action in the loop.

Table 4.9: **Negative traces are architecture data:** Failed builds, invalid candidates, missed constraints, and rejected alternatives define the boundary of the design space and should be preserved rather than discarded.

Negative trace	What it records	What the loop learns
Synthesis or timing violation	Constraints, process, parameters, failing setup/hold paths.	Exclude this macro or parameter combination.
Unroutable or PDN failing netlist	Placement, density, pin layout, congestion map, IR drop.	Add congestion-aware spacing or reshape the action space.
Proxy win that fails fidelity	Cheap metric improves, but stronger evaluation rejects it.	Calibrate proxies and require sensitivity checks.
Tool failure or crash	Simulator, synthesis, compiler, or harness cannot complete.	Separate design failure from environment failure.
Coverage gap	Workload, input, scenario, or architecture class was not represented.	Mark the evidence boundary before committing.
Rejected design rationale	Human review rejects a candidate for risk, maintainability, schedule, or integration.	Preserve architect judgment as training and audit data.

Negative traces require provenance—an exact, replayable record of the environment that produced the result. A failed result without context is not very useful. The minimum reusable record includes the candidate identifier, workload slice, software stack, tool version, seed, parameters, constraints, fidelity level, failure reason, coverage boundary, and decision owner. The loop also needs coverage. What part of the design space, workload space, or evidence regime does this trace represent? Without provenance and coverage, negative traces become a pile of anecdotes. With them, they become architecture data.

4.8 When a Representation Becomes Actionable

Collecting this architecture data—from high-fidelity negative traces to structured constraints and world-model escalation triggers—ultimately serves a single goal, enabling automation. A representation becomes actionable when it can safely support loop

operations. It should define valid actions, expose relevant observations, carry constraints and objectives, record provenance, support replay, preserve uncertainty, separate feedback from evidence, and record what rejects a candidate. It should also say what remains a human decision.

This is a higher bar than asking whether a model can read it. A representation that can be summarized may still be useless for design. A representation that can be searched may still hide invalid actions. A representation that produces a score may still lack provenance. A representation that captures winning results but not rejected alternatives may give the loop a biased view of the space.

The same test applies to abstractions. A useful Architecture 2.0 abstraction does not merely hide detail or provide nicer names. It creates a surface where valid actions, feedback, rejection, replay, or review can happen earlier, cheaper, or with clearer authority. An abstraction that cannot return feedback is merely notation; it may help people talk, but it has not yet made the loop more capable.

For the lighthouse prompt, an actionable representation would not merely contain the prompt text. It would include the XRBench scenario, workload metadata, architecture parameters, software assumptions, power and process constraints, tool configurations, candidate set, feedback budget, evidence regime, negative traces, and rejection authority. Only then can the loop ask a bounded question: which candidate should be evaluated next, which evidence is strong enough, and which decision still belongs to the architect?

> **Design principle: Make architecture work legible to the automated loop**
>
> A representation earns trust only when it records provenance, assumptions, costs, failures, and negative traces, not only the successful endpoints. What the loop cannot see, it cannot reason about, and what the human architect cannot see, they cannot safely reject.

4.9 Conclusion

This chapter asked what architecture data must record before an automated optimizer can act on it and a human architect can audit the result. Representation is the first hard problem because a method can only act on what the loop can see. Get it wrong and a more capable model simply moves faster in the wrong direction. Get it right and the loop gains a surface where valid actions, feedback, rejection, replay, and review can actually happen.

Legibility is the bar, and it sits higher than readability. A representation is actionable only when it defines valid actions, exposes relevant observations, carries constraints and objectives, records provenance, supports replay, preserves uncertainty, separates feedback from evidence, records what can reject a candidate, and says what remains a human

decision. A representation that captures winning results but not rejected alternatives, or scores but not provenance, hands the loop a biased and unauditable view of the space.

The durable rule is to make architecture work legible to the automated loop, which means recording provenance, assumptions, costs, failures, and negative traces, not only the successful endpoints. What the loop cannot see, it cannot reason about, and what the architect cannot see, they cannot safely reject. World models, structured design data, and negative-trace corpora all serve that one end, turning scattered architecture experience into state a loop can read, replay, compare, and revise.

4.10 Open Research Questions

As we have seen, the gap between public architecture knowledge and actionable loop state is significant. This gap leaves several unsettled research directions for the community to explore. Solving these requires moving beyond static data to dynamic, fidelity-aware representations that can safely guide an automated optimizer.

1. **What is the formal semantics of an auditable architecture proposal?** While representations must act as boundary objects between human intent and tool execution (see the discussion on "Architecture Descriptions as Boundary Objects" in Section 4.3), we lack a rigorous calculus for optimizer proposals. A thesis-level challenge is to define a canonical, machine-readable schema for loop-state records that supports independent replay, checking, and safe rejection of an AI-generated architecture candidate using its attached provenance.
2. **Can world models discover and navigate fidelity frontiers?** Building on the escalation triggers defined in our world model sketch (Table 4.8), future research must move beyond static rule-based thresholds. An open challenge is to develop predictive surrogates that learn their own calibration boundaries and route high-risk design evaluations to cycle-accurate simulation or physical synthesis when the loop's evidence boundary is exceeded, thereby managing the trade-off between sample cost and prediction risk.
3. **What is the information-theoretic limit of generalization in AI-driven design?** As emphasized by treating sample cost as architecture data (see the discussion on "Sample Cost Is Architecture Data" in Section 4.2), workload provenance is critical loop state. Yet, we lack rigorous methods to detect when an automated optimizer has overfit to a specific execution phase or synthetic input distribution. A major research direction is establishing formal generalization bounds[11] for architecture agents, helping prevent silent hardware failures under distribution shifts toward architecture world models (see the discussion on "Toward Architecture World Models" in Section 4.6).
4. **How do we formulate representation learning over negative architecture traces?** As established in our discussion of provenance, coverage, and negative traces (see the discussion on "Provenance, Coverage, and Negative Traces" in Section 4.7),

[11] Theoretical limits on how well a machine learning model's performance on its training data will apply to unseen, out-of-distribution data.

rejected candidates, missed timing constraints, and invalid parameters define the true boundary of the design space (Table 4.9). However, current generative models are primarily trained on successful artifacts. A critical problem for the ML-for-systems community is designing novel loss functions[12] or preference-optimization pipelines[13] that force models to learn from high-fidelity negative records, ensuring they actively avoid—rather than rediscover—known failure modes.

[12] Mathematical functions used to measure the difference between an AI model's predicted output and the actual target, which the training process seeks to minimize.

[13] Techniques like Reinforcement Learning from Human Feedback (RLHF) or Direct Preference Optimization (DPO) used to align a model's outputs with human preferences.

What to carry forward

- **Reader test:** Could an automated optimizer replay both the winning and the rejected candidates six months later, compare the evidence boundaries, see who could reject the result, and read what commitment the evidence supports?
- **Up next:** Once that state exists, tools can become environments that act on it and return interpretable feedback. That is what representation buys in Architecture 2.0, constrained automated action under human-auditable authority.

Chapter 5

Tools as Architecture Environments

"We shape our tools and thereafter our tools shape us."
— John Culkin, Media Scholar

Author's Note: John Culkin, an influential American media scholar, observed that our tools ultimately reshape our own behavior. In Architecture 2.0, Culkin's "us" expands beyond the human architect; by wrapping existing EDA tools into programmatic environments for AI agents, we are fundamentally reshaping both the automated loop itself and what is possible to build.

The crux

What turns a tool flow into an environment a generative model can act in and a reviewer can trust?

Chapter 4 argued that a loop can only act on what it represents. This chapter asks where the loop acts. In Architecture 1.0, tools often sit behind the architect: simulators, scripts, compilers, profilers, spreadsheets, RTL flows, EDA tools, dashboards, and deployment logs are the means by which a human expert gathers evidence. In Architecture 2.0, those same tools must become explicit environments when they sit inside AI-assisted design loops. They define what actions are legal, what observations are returned, how expensive feedback is, which assumptions are baked in, what state is logged, what can reject a candidate, and what commitment the environment can support before the loop spends more effort.

Architecture environment. An architecture environment is the tool-facing part of the loop contract made executable. It maps the five-part execution state directly to the toolchain: it specifies the legal actions (*actions allowed*), observations and feedback costs (*state seen*), logged state and provenance (*evidence*), rejection authority (*alternatives rejected*), and commitment boundaries (*ownership*) for the tools a design loop acts through.

To keep these boundaries clear, we must use the tool vocabulary carefully. A *tool* performs an operation. A *wrapper* calls that tool. An *environment contract* states what actions are legal, what observations return, what costs and failures mean, and what evidence is logged. A *harness* is the maintained runnable object around that contract: workloads, versions, scripts, provenance, invalid-action records, and review state.

This shift from simple wrappers to formal environment contracts is easy to understate. A tool wrapper is not plumbing. It is a research claim about the architecture problem. It decides which parts of the design space are visible, which constraints are enforceable, which metrics are trusted, which failures are recorded, and which actions are silently impossible. A weak environment can make a strong method look useful by hiding the hard cases. A disciplined environment can make a modest method valuable by making the task bounded, repeatable, and rejectable.

The same contract is what makes comparison meaningful. Two AI-assisted methods can be compared only when they act under compatible workload definitions, action schemas,

feedback budgets, invalid-action rules, provenance records, and commitment boundaries. Otherwise a result may compare private worlds rather than architecture methods.

The lighthouse prompt makes the point concrete. "Design a low-power, 64-bit RISC-V-based compute subsystem for an XRBench real-time mobile XR workload under a 3 W, 3 nm-class low-power mobile envelope" is not one call to one tool. It needs a workload harness, software stack, ISA and vector assumptions, candidate architecture representation, simulator or estimator, power model, compiler/runtime interface, validity checks, provenance log, and a way to record rejected alternatives. If any of those pieces are implicit, the prompt is not yet an Architecture 2.0 loop. It is only a sentence.

> What this chapter gives you
>
> After this chapter you can turn a tool path into a reviewable environment contract. That means you can:
> - turn a tool into an environment with an explicit action, observation, cost, rejection, and commitment contract;
> - read a result by naming the environment that produced it;
> - explain why environment contracts make AI-method and paper comparisons meaningful;
> - reason about feedback latency and fidelity as an economy of evidence;
> - detect proxy mismatch and proxy gaming, simulators included, before they mislead the loop.

5.1 Tools Shape the Research Question

Computer architects have always used tools to reason quantitatively about systems. Simulators, analytic models, profilers, compilers, RTL generators, EDA flows, and measurement systems make design spaces tractable. They also shape the questions that can be asked. A simulator with a particular memory model makes some cache questions natural and others awkward. A compiler pass that exposes one schedule representation and hides another constrains what an optimizer can change. An EDA flow that returns timing and power after hours of work makes feedback precious. A runtime telemetry system that reports aggregate utilization but not per-request interference makes some deployment claims difficult to support. When tools are used by a human expert, some of these limits can remain tacit. For an AI-assisted loop, tacit limits become hidden state. The loop can act safely only on constraints, costs, and rejection rules the environment exposes.

The usual way to describe these tacit boundaries is to say that tools have limitations. That is true, but too weak. Tools do not merely limit architecture work; they define its observable world. They decide what state exists for the loop, what actions can be applied to that state, and what feedback comes back. The classic quantitative tradition in computer architecture shifted the field from intuition to rigorous simulation and

measurement (Hennessy and Patterson, 2017). Architecture 2.0 extends that tradition, asking for one more layer of explicitness. Where the first era measured systems, the second era makes the measurement environment itself a programmable artifact. The tool interface itself must be part of the design object.

This level of explicitness matters because generative and learning-based methods are literal about interfaces. A human architect can sometimes infer that a simulator result is out of distribution, that a benchmark run used a stale configuration, or that a reported improvement is not meaningful because the compiler changed. A method acting through an environment will not infer those facts unless the environment represents them. The environment must expose enough state for useful action and enough constraints for safe rejection.

The right question is therefore not, “Which tool did the paper use?” The better question is, “What environment did the loop define?” That question forces the paper or project to name its workload distribution, action schema, observation schema, feedback latency, validity constraints, cost model, provenance record, and rejection authority. Once those are visible, method claims become easier to compare.

5.2 Interfaces Are Action Boundaries

To translate these environments into practice, what makes a tool’s action and observation schema concrete is the interface it exposes. Architecture is often described as the boundary between hardware and software. For Architecture 2.0, that statement has an operational meaning. Interfaces are where actions become legal or illegal, observations become meaningful or misleading, and evidence becomes portable or trapped inside one tool script. An ISA, compiler IR, Domain-Specific Language (DSL) like SLICC for cache coherence, memory model, accelerator runtime, simulator API, benchmark harness, EDA handoff, or telemetry schema is not just a convenience for implementation. It defines what a loop can change and what can reject the change. AI agents must therefore interface with these DSLs; emitting monolithic C++ or raw RTL defeats the modularity required for a human architect to review and maintain the logic. However, an AI agent mutating a concurrency DSL like SLICC cannot rely on a cycle-level simulator as its sole rejection authority. The design loop must explicitly integrate formal model checking (e.g., Murphi) or state-space exploration to catch latent race conditions, livelocks, and deadlocks before commitment.

This is why tool interfaces belong in the architecture argument rather than in an appendix. A generator that emits a schedule must know the schedule language. A search method that changes memory hierarchy parameters must know which combinations the simulator accepts and which violate a software-visible contract. A critique system that reads a synthesis report must know which warnings are fatal, which are informational, and which require a higher-fidelity check. The interface is the boundary where method capability meets architectural validity.

Table 5.1 lists the interfaces that a credible loop often has to expose. The table is not a complete taxonomy. Its claim is that every interface has two jobs: it makes some actions possible, and it defines what evidence those actions can produce. If either side is hidden, an automated optimizer can appear capable while acting outside the architecture problem the human intended to solve.

Table 5.1: **Architecture interfaces define action and evidence boundaries:** Each interface tells the loop what can be changed, what can be observed, what feedback is meaningful, and what can reject an invalid action.

Interface	What it makes actionable	Evidence it makes interpretable	Failure if hidden
ISA and vector contract	Instructions, registers, vector length, exceptions, privilege, and binary compatibility.	Correct execution, portability, software-visible behavior, and compatibility tests.	The loop proposes a microarchitecture that software cannot legally target.
Compiler IR and schedule representation	Lowering choices, tiling, fusion, layout, vectorization, and target-specific code generation.	Compiler success, generated code, performance counters, and optimization provenance.	Performance is attributed to hardware while the software contract changed.
Memory and coherence model	Ordering, sharing, cacheability, consistency, DMA, and synchronization assumptions.	Correctness tests, contention behavior, latency, bandwidth, and race or ordering failures.	A candidate looks fast because it violated the program's memory assumptions.
Accelerator or runtime API	Invocation, data movement, synchronization, library calls, queues, and resource ownership.	End-to-end latency, overhead, utilization, portability, and software integration cost.	Specialized hardware is efficient in isolation but unusable in the system.
Simulator or environment API	Legal parameters, workload inputs, observations, errors, seeds, and fidelity levels.	Comparable runs, replayable experiments, invalid-action records, and feedback cost.	The method optimizes simulator quirks or incomparable configurations.
EDA handoff and constraints	RTL, clocks, floorplan hints, timing constraints, power intent, physical limits, and signoff checks.	Timing, area, power, congestion, rule violations, and implementation feasibility.	A candidate survives architectural simulation but fails physical reality.
Benchmark harness	Inputs, versions, metrics, splits, data-leakage rules, and submission constraints.	Coverage, reproducibility, benchmark validity, and claim scope.	The loop overfits a stale or leaky benchmark slice.

Interface	What it makes actionable	Evidence it makes interpretable	Failure if hidden
Telemetry and deployment schema	Live workload mix, service-level objectives (SLOs), counters, interference, rollout state, and drift signals.	Field behavior, regressions, rollback triggers, and post-deployment calibration.	Production evidence is rich but too confounded to support the architecture claim.

The EDA handoff row is worth making concrete, because it is where a wrapper does the most work. Validity comes in two tiers, and honest environments separate them. *Schema legality* is cheap and checkable up front, asking whether the parameters, clocks, and interfaces are well formed. *Physical feasibility* is late and expensive, because routability, timing closure, congestion, and IR drop often surface only at synthesis or place-and-route. A late failure must map back to the earlier action that caused it, not read as a contract violation.

Wrapping a commercial flow as an environment is therefore not a thin API. The wrapper has to emit tool-specific scripts, parse unstructured reports, survive timeouts and run-to-run nondeterminism, and attach the fidelity or confidence label itself, because the tool emits metrics, warnings, and logs, not a fidelity level. To stay auditable, its provenance record must carry more than a bare number. It records the tool and host version, thread count, license state, run-to-run variance, warnings and waivers, and a hash of the report the number came from. Two "identical" runs that differ on any of these are not comparable, and a loop that hides the difference will trust a result it should have rejected.

Lighthouse prompt: One prompt, many interfaces

Context. The lighthouse request crosses multiple interfaces at once, so no single simulator API or wrapper is the environment.
In the Lighthouse prompt. "64-bit RISC-V-based" is an ISA and ABI/software contract. The fragment "vector-capable CPU, accelerator, or SoC block" is an action boundary over compute organization and integration. "XRBench-class real-time mobile XR workload" fixes the workload harness and quality-of-service target. "3 W TDP target in a 3 nm-class LP mobile process" invokes power and physical-design constraints.
Boundary. When the optimizer proposes a SoC block instead of an isolated accelerator, it changes the action space. The environment must now expose memory attachment, coherence, interrupts, DMA, runtime ownership, verification scope, and the evidence needed to show that the subsystem works in the larger system, otherwise the loop will optimize blindly.
Takeaway. The environment for this prompt is a bundle of coherent interfaces. An AI-assisted loop can change vector width, local memory, data layout, or accelerator interface only inside contracts that can reject software-incompatible, physically implausible, or integration-breaking candidates.

In practice, concrete tools instantiate this bundle of interfaces at different fidelity levels. The point is not that Architecture 2.0 requires one canonical simulator or one vendor flow. The point is that a loop must say which environment it is acting in, what that environment can observe, and what authority its feedback has. Table 5.2 gives a compact set of examples.

Table 5.2: **Concrete tools instantiate the environment contract:** A credible loop should name the simulator, RTL flow, emulation path, EDA stage, or deployment harness it is acting through, because each one exposes different actions, feedback, cost, and rejection authority.

Environment instance	Action boundary	Feedback and evidence	Loop lesson
gem5-style cycle or full-system simulation (Binkert et al., 2011)	Core, cache, memory-system, ISA, and workload configuration.	Statistics, traces, timing-model behavior, simulator warnings, and scoped performance comparisons.	Strong for controlled DSE; bounded by model fidelity and configuration state.
Verilator-style compiled RTL simulation (Veripool, 2026)	RTL modules, test benches, assertions, generated C++/SystemC models, and debug hooks.	Functional behavior, waveform/debug evidence, assertion failures, and implementation-adjacent traces.	Moves closer to implementation but narrows throughput and increases debug cost.
FireSim-style FPGA-accelerated simulation (Karandikar et al., 2018)	RTL target, workload image, network model, FPGA mapping, and runtime configuration.	Faster cycle-exact feedback, workload-scale behavior, instrumented counters, and deployment-like experiments.	Speed changes sample economics, but setup and observability become part of the evidence.
Synthesis, place-and-route, and signoff flow (Mirhoseini et al., 2021; Semiconductor Industry Association, 2026; Bauer et al., 2020)	RTL, constraints, floorplan hints, clocks, power intent, libraries, and process assumptions.	Area, timing, power, congestion, rule violations, waived warnings, and closure failures.	High-fidelity samples are scarce; use them as rejection gates, not blind search targets.

For each row, the Architecture 2.0 object is not the tool name but the maintained action/observation record: workload version, legal edits, failed runs, waivers, escalation gate, and human owner.

In practice these environments are rarely used in isolation. Open frameworks integrate them. Chipyard, an open-source framework for generating and evaluating RISC-V systems-on-chip, wraps a configuration system, Chisel-based RTL generators such as the Rocket and BOOM cores, and a FIRRTL compiler path together with the simulation, synthesis, and FPGA-emulation backends the table lists, so one configuration can be carried reproducibly from a generated design to each evaluation path (Amid et al., 2020). Figure 5.1 shows that integration. For an AI-assisted loop, the useful property is not the framework name but where the loop actually runs. The automated optimizer acts at the configuration boundary, editing one configuration under a set of legal actions; each backend, whether simulation, synthesis, or FPGA emulation, returns a receipt; and those receipts, together with invalid-action logs, escalation thresholds, and human decision points, are the loop contract that keeps a generated configuration from becoming an unowned commitment.

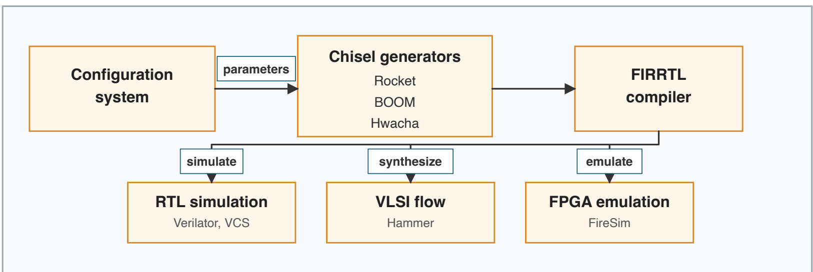


Figure 5.1: **An integrated environment ties the backends together:** A reproducible SoC generation framework connects a configuration system, RTL generators, and a compiler to simulation, synthesis, and FPGA-emulation backends, so one configuration reaches every evaluation path.

However, even the broad categories in Table 5.2 hide internal ladders of fidelity. Post-synthesis timing, for example, is often a useful surrogate for post-route timing, but routing congestion, IR drop, clocking, and signoff checks can still reject a candidate that looked acceptable at synthesis. The lesson is not that every loop must begin with the strongest tool. It is that the loop must know which feedback is a proxy and which later check has authority to overturn it. For an AI-assisted loop, that later authority is what decides whether a candidate earns another sample, an escalation, or termination.

Physical-design environments make this staged contract particularly visible. A loop should not report “EDA feedback” as if all tool stages had the same authority. Table 5.3 separates common stages by what they can and cannot reject.

Table 5.3: EDA feedback has stage-specific authority: From logic synthesis through signoff, each physical-design stage returns different evidence and should be allowed to reject different claims.

EDA stage	Feedback returned	What it can reject	What it cannot prove alone
Logic synthesis	Mapped gates, area proxy, timing estimates, constraint warnings, and early power estimates.	Nonsynthesizable RTL, impossible constraints, obvious area/timing problems.	Routability, final timing, IR drop, clocking, and post-layout power.
Floorplanning and placement	Cell locations, utilization, congestion hints, timing pressure, and macro or memory placement effects.	Floorplans that create severe congestion, timing pressure, or integration problems.	Final routed timing, signoff power, manufacturability, and workload-level benefit.
Clock, routing, and power closure	Routed timing, clock behavior, congestion, design-rule checks, power integrity, and closure failures.	Candidates whose physical effects invalidate earlier architecture estimates.	Product commitment without verification, workload coverage, and architecture review.
Signoff and review	Tool signoff reports, waivers, residual risks, review decisions, and ownership records.	Unsupported claims, unacceptable waived warnings, and evidence gaps before commitment.	Generality beyond the stated workload, process, tool version, and design context.

This arrangement is not hypothetical. Some commercial physical-implementation flows already treat synthesis and place-and-route as a search environment: tool directives and floorplan parameters become actions, full tool runs become transitions, and reports become observations about power, performance, area, and closure (Synopsys, 2023; Cadence Design Systems, 2021). The durable point is not which vendor leads, since the products will be renamed and replaced. It is that an EDA flow becomes an environment only when those directives are an action schema, reports are observations with provenance, closure and signoff are rejection authorities, and the architect owns the gates the loop may not cross.

5.3 The Architecture Environment Abstraction

To formalize these interfaces across different tools, we need an abstraction. The environment has an executable core, but it is not only software. It receives a proposed action, checks whether that action is meaningful, calls one or more tools, collects observations, logs provenance, and returns feedback that can become evidence. It can also enforce or record declarative interface contracts: ISA rules, memory models, telemetry schemas,

benchmark rules, and EDA constraints. As illustrated in Figure 5.2, an architecture environment fundamentally reshapes this relationship. Rather than just taking inputs and returning outputs, it wraps the underlying tool with a strict contract that exposes legal actions, bounds observations, enforces physical costs, and explicitly defines rejection paths for invalid states.

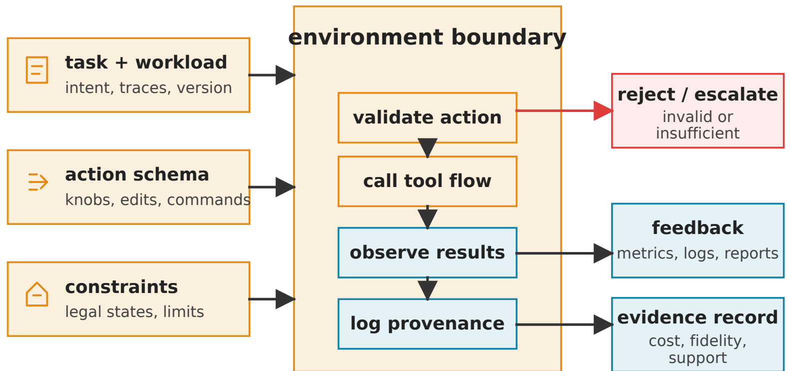


Figure 5.2: **A tool becomes an environment when its contract is explicit:** A credible environment defines workload state, legal actions, observations, constraints, cost, provenance, feedback, invalid-action semantics, and rejection paths.

The figure is a loop contract, not a software architecture diagram. Each component is useful only after the environment names the action it permits, the feedback it returns, and the rejection authority attached to that feedback. That is why two projects can use the same simulator but define different architecture environments.

Fundamentally, this abstraction can be described as a contract. The environment is literal. It is the toolchain exposed as an executable sandbox. The environment does not compute a scalar reinforcement-learning reward. It emits raw observations and cost telemetry—such as cycle counts, gate area, and test failures. The loop's rejection authority evaluates this multi-dimensional telemetry against our claims. If we let tools compute a scalar reward directly, we invite reward hacking[14] where the loop optimizes for simulator bugs instead of physical reality. The contract requires that the loop make a small set of obligations explicit. Table 5.4 lists the main fields.

[14] Reward hacking occurs when an AI agent finds an unintended shortcut to maximize its reward signal without solving the underlying problem.

Table 5.4: **An environment contract makes tool use auditable:** The loop should state its action schema, observations, costs, failures, provenance, and human-review boundaries before methods operate inside it.

Field	What it defines	Question it answers
Workload distribution	Inputs, traces, benchmark versions, software stack, and operating scenarios.	What behavior is the design supposed to serve?
Action schema	Parameters, edits, configurations, generated artifacts, or commands the loop may propose.	What can the method actually change?
Observation schema	Metrics, traces, logs, reports, errors, and artifacts returned after an action.	What can the loop observe after acting?
Constraints and validity	Type checks, feasibility rules, physical/manufacturing limits, software ecosystem compatibility, and invalid-action handling.	What makes a candidate illegal before performance is considered?
Feedback budget	Cost, latency, fidelity, determinism, parallel scalability, and sample limits for each source of feedback.	How much evidence can the loop afford?
Provenance record	Tool versions, seeds, inputs, configuration files, assumptions, and artifact hashes.	Could a human replay or audit the result?
Rejection authority	Conditions that stop, revise, or escalate a candidate.	What can say no?

The contract is deliberately broader than reinforcement learning terminology. Actions, observations, and rewards are useful terms, but architecture work also needs constraints, invalid-action semantics, provenance, and human decision points. A loop that proposes a cache size, vector width, chiplet partition, compiler flag, or RTL edit needs to know not only whether a score improved, but whether the candidate is legal, reproducible, comparable, and worth committing to a higher-fidelity stage.

Applying this contract to the lighthouse prompt, the environment might expose actions such as changing vector width, local memory size, issue width, cache configuration, accelerator interface, or data layout. It might return observations such as latency, throughput, estimated power, memory traffic, area proxy, simulator warnings, compiler failures, and rejected workloads. It should also return cost: how long the run took, which fidelity level was used, and how much confidence the loop should place in the feedback.

Without that cost and provenance, the loop cannot reason about sample efficiency or evidence.

5.4 ArchGym as a Worked Example

ArchGym, which frames architecture design as a gymnasium for machine-learning-assisted design where automated methods interact with architecture environments through defined interfaces (Krishnan et al., 2023), makes the environment idea concrete in the architecture domain. The name is deliberate. It borrows from OpenAI Gym, which standardized the agent-environment interface in reinforcement learning. Just as one learning algorithm can drive many Gym environments, an ArchGym user can swap a memory-controller simulator for an accelerator simulator while keeping the same optimization or learning method intact, which is exactly the composability a shared environment is supposed to buy. The Architecture 2.0 gymnasium essay makes the same broader argument. The field needs data-centric environments where architecture tasks, feedback, and evaluation are exposed systematically (Janapa Reddi and Yazdanbakhsh, 2023).

For our purposes, the important lesson is not that every project should use ArchGym literally. The lesson is that a shared environment changes the research question. Instead of asking whether one optimizer beat another under a private script, the community can ask which task was defined, which action space was exposed, what feedback was available, what workloads were used, and which tools or methods were compared under the same conditions. That makes method claims less anecdotal. The comparison is credible only when every AI-assisted system acts against the same represented state, receives the same feedback records, and accepts the same invalid-action, escalation, and human-review rules.

Table 5.5 reads an ArchGym-style environment as a mini-contract.

Table 5.5: **A worked environment example is a contract, not only a benchmark:** ArchGym is useful for Architecture 2.0 because it makes the task, action space, feedback, comparison setting, and limits discussable.

Contract field	ArchGym-style instance
Task	Search or tune a bounded architecture design space under a declared workload and metric.
Action space	Architecture parameters, simulator knobs, or mapping choices the automated method is allowed to change.
Observation and reward	Metrics returned by the simulator or tool path, with the reward derived from those observations.
Workload and tool path	Benchmark inputs, simulator configuration, and versioned evaluation scripts.

Contract field	ArchGym-style instance
Invalid-action behavior	Rules for rejected parameters, failed runs, timeouts, or unsupported configurations.
Feedback cost	Approximate turnaround and sample budget for each evaluation regime.
Rejection and commitment boundary	Independent checks that can reject a result and the strongest claim level the environment can support.
Limits	Simplified simulators, cleaner action spaces, and missing industrial constraints can bound what the result proves.

ArchGym also shows why the environment chapter cannot be collapsed into the methods chapter. Once an environment is defined, many methods can interact with it: Bayesian optimization, reinforcement learning, evolutionary search, surrogate-guided exploration, random search, heuristic search, or a human designer using the environment as an instrumented assistant. The environment is the common ground on which method comparisons become meaningful.

Evolutionary search: In this chapter, a way to propose candidate actions by mutation, crossover, and selection, useful only when the environment can reject invalid or weak candidates.

Surrogate-guided exploration: In this chapter, using a fast predictor to screen candidate actions while preserving the stronger feedback needed to reject proxy wins.

At the same time, ArchGym should not be treated as if it solves the whole Architecture 2.0 problem. A gym can still use simplified simulators. It may not include proprietary physical-design constraints, confidential workloads, tool-license behavior, workload drift, negative trace capture, or deployment telemetry. Its action spaces may be cleaner than industrial design spaces. Its feedback may be faster and more standardized than the feedback available in late-stage silicon work. Those limitations are not reasons to dismiss the environment pattern. They are reasons to make environment validity a first-class concern.

5.5 Interfaces Make Loops Composable

Validity is not the only reason to insist on an explicit interface. A single tool wrapper can support one experiment. A disciplined interface can support a research ecosystem. The difference is composability. If action schemas, observation schemas, workloads, provenance records, and validity rules are explicit, then generators, predictors, optimizers, critics, verifiers, and human reviewers can be swapped or combined without rewriting the whole loop.

This is how architecture environments can become community infrastructure. As Chapter 2 argued for benchmark governance, useful infrastructure such as MLPerf does more than name workloads. It creates rules, versions, metrics, submissions, and

comparison conventions that help a community interpret results (Mattson et al., 2020). Benchmark governance governs comparisons; environment governance must also govern what AI-assisted systems may try, which actions are invalid, what feedback means, and what can stop the loop. A useful environment should not merely publish a script; it should publish the contract under which actions are legal, observations are valid, and evidence can be compared. Keeping those contracts valid after the first paper is the operating discipline Section 5.9 develops.

This transition from single-use wrappers to shared community infrastructure is why our opening vocabulary reserved the word harness. A wrapper calls a tool. A harness preserves the contract around the tool: task, workload, action schema, observation schema, cost, provenance, invalid-action semantics, negative traces, and review status. A multi-participant harness adds role boundaries: which component may propose, which may execute, which may critique, which may verify, and which human decision is required before escalation. The distinction matters because a wrapper can automate one experiment, but a harness can accumulate reusable knowledge about a design space.

A minimal implementation receipt for such a harness might include:

> task identifier; workload version; input distribution; action fields; read-only constraints; tool commands; observation fields; fidelity level; runtime cost; random seed; tool versions; generated artifacts; failure status; rejection reason; and human decision.

That list is intentionally mundane. Mundane records are what make loops auditable. If a method proposes an architecture candidate, the environment should preserve not only the winning score but also the command that produced it, the workload revision, the tool version, the failed alternatives, the warnings, and the conditions under which the candidate would be rejected.

This focus on composability also changes how we interpret AI-assisted systems. A compound design system may have a planner, code generator, simulator caller, surrogate model, evidence critic, and human reviewer. Those components can coordinate only if the environment gives them a shared state representation and stable interfaces. If those components are implemented as several agents, the need for an environment contract does not shrink; it grows. The harness has to record which component read which state, which action it took, which feedback it used, and which gate could reject its output. Without that, an “agent” is merely a wrapper around a pile of scripts. With it, the loop can become an inspectable system.

5.6 Feedback Latency and Fidelity

Once an environment is structured to support these composable agents, the hardest design choice is often not the action schema. It is the feedback regime. Architecture feedback ranges from cheap and weak to slow and authoritative. A simple analytic proxy may return in milliseconds. A cycle-level simulation may take minutes or hours, because a detailed simulator typically runs many orders of magnitude slower than the hardware it models, so seconds of target execution become hours of wall-clock time.

Synthesis, place and route, or signoff of a single block are commonly hours-to-days jobs. Hardware-in-the-loop, deployment telemetry, and silicon evidence may arrive only after substantial commitment.

This is why more capable AI does not remove the environment problem. In many architecture loops, the model call is fast and the feedback is slow. A method may propose hundreds of candidates before a simulator, compiler, synthesis flow, verification run, or human review can return decision-grade evidence. The environment therefore becomes the pacing item. It must decide which actions are worth spending feedback on, which cheap checks can reject early, and when the loop should escalate to a stronger but scarcer source of evidence.

Architect's checkpoint: The Escalation Gate

When an AI method proposes a candidate, do not automatically grant it a high-fidelity evaluation. The environment must enforce an escalation gate that asks:

- Did this candidate survive all cheap proxy and validity checks?
- Is the potential improvement worth the cost of a scarce simulator or synthesis sample?
- If the high-fidelity check fails, can the environment map that failure back to a specific tool action?

This dynamic creates an economy of evidence. Cheap feedback buys breadth, pruning, and parallel scaling; expensive feedback buys stronger rejection authority. As visualized in Figure 5.3, this economy dictates the fundamental pressure of the design loop. As feedback moves toward implementation and deployment, the loop usually spends exponentially more time per sample, gains less freedom to explore, and needs dramatically clearer justification for every action it takes. The ranges are representative rather than universal. A real project should replace them with its own source receipts; they are illustrative bands, not universal empirical claims. In the early regimes, the horizontal axis mostly means tool or measurement turnaround; near prototype, silicon, and deployment, it also includes setup, queueing, fabrication, rollout, and commitment delay.

This economy is the multi-fidelity setting studied across computational science, where cheap, lower-fidelity models are combined with scarce, high-fidelity ones to keep optimization, inference, and uncertainty quantification affordable (Peherstorfer et al., 2018). Architecture 2.0 inherits that economy and adds two architecture-specific requirements: each fidelity level must carry its own rejection authority, and a move to a higher level crosses a commitment boundary, not merely a more accurate number.

Uncertainty quantification: In this chapter, recording where an environment's feedback is uncertain enough to force escalation rather than commitment.

This economic pressure can also be expressed as a practical design checklist. Table 5.6 extends the sample-cost regimes of Chapter 4 (Table 4.5) from what to record toward what each regime can reject, how many samples the loop can plausibly afford, and what method behavior that economics permits. A loop that confuses those regimes can search quickly and still learn the wrong lesson.

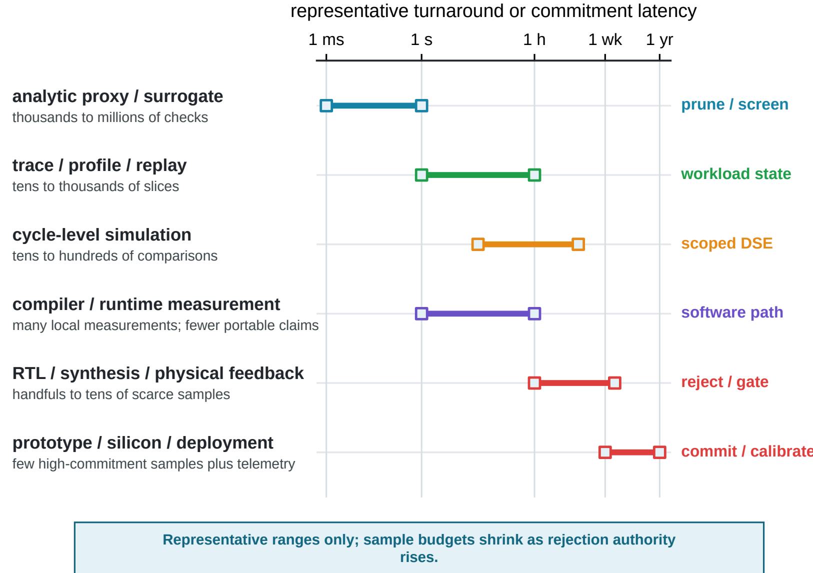


Figure 5.3: **Feedback fidelity changes the economics of search and commitment:** Low-cost feedback can screen many candidates, but implementation-adjacent feedback is scarce and more authoritative. The vertical ordering reflects rejection authority more than strict latency; the point is the shrinking sample budget, not the exact wall-clock value for any particular project.

Table 5.6: **Feedback regimes create evidence economics:** A method should match the latency, sample budget, rejection authority, and commitment level of the feedback it receives. Representative ranges are project-dependent and should be replaced by local source receipts in a real loop.

Feedback regime	Cost / latency intuition	Sample budget	What can reject	Method implication
Analytic proxy or learned surrogate	Milliseconds to seconds; low direct cost but high model-risk exposure.	Thousands to millions of candidate checks, highly parallelizable.	Obvious invalidity, dominated regions, sensitivity failures, or proxy-calibration failures.	Use for pruning, active learning, and broad search, not final commitment.
Trace, profile, or replay	Seconds to hours depending on capture, replay, and privacy filtering.	Tens to thousands of slices or scenarios.	Coverage gaps, stale workload versions, leakage, or mismatch to intended deployment.	Use for workload state, clustering, and targeted tests.

Feedback regime	Cost / latency intuition	Sample budget	What can reject	Method implication
Cycle-level simulation	Minutes to days depending on model detail and target workload.	Tens to hundreds of scoped architecture comparisons.	Simulator configuration errors, unsupported states, calibration gaps, and sensitivity checks.	Use staged design-space exploration with replay and escalation.
Compiler/runtime measurement	Compilation, build, profiling: seconds to hours; target execution, replay, or autotuning: minutes to days.	Many local measurements, fewer portable claims.	Correctness tests, portability checks, and end-to-end software-path evidence.	Use autotuning or generation only with tests and provenance attached.
RTL, synthesis, or physical feedback	Hours to weeks when timing, power, congestion, or signoff constraints enter.	Handfuls to tens of scarce high-fidelity samples.	Timing, power, area, congestion, design-rule checking, formal, or waived-warning review.	Use filters, surrogates, and human gates before spending samples.
Prototype, silicon, or deployment telemetry	High setup cost; weeks to years when field evidence or silicon commitment is required.	Few high-commitment measurements plus ongoing telemetry.	Field behavior, reliability, rollback policy, incident review, and accountable human decision.	Use for calibration, validation, and drift monitoring, not blind exploration.

However, this table must be read with a fidelity-risk rule. Lower-fidelity feedback can prune and prioritize, but it can also be gamed or contradicted. When a proxy says “yes” and a stronger environment says “no,” the loop should record the mismatch instead of treating it as noise. The simulator-mismatch discussion below returns to that failure mode.

Compiler and runtime feedback carry their own version of this same risk. A hardware candidate can look weak because the schedule, tiling, vectorization, layout, or runtime path is poor, not because the architecture is poor. That is a false negative created by the software side of the environment. For the lighthouse prompt, “vector-capable” therefore cannot mean only that an ISA feature exists; the loop must also expose whether the compiler and runtime can generate a credible path to use it.

This fidelity regime structure therefore drives method choice. If feedback is cheap, broad search or online adaptation may be reasonable. If feedback is expensive, the loop needs sample efficiency, priors, surrogates, active learning, staged gates, or stronger human filtering. If feedback is high commitment, the loop should become more conservative. Use AI to organize evidence, critique assumptions, and narrow the search, not to make unsupported final decisions.

> **Active learning:** In this chapter, a policy for choosing which scarce simulator, synthesis, human, or deployment feedback event to spend next, with the choice logged as part of the evidence ledger (Settles, 2009).

The environment should therefore expose feedback budget as a first-class object. A result should report which fidelity level produced a comparison and at what cost, not only that candidate A beat candidate B. When that feedback is strong enough to count as evidence, and what provenance, uncertainty, and human decision that requires, is the bridge from this chapter to Chapter 7, which treats it in full.

5.7 Proxy Gaming and the Simulator Case

Because low-fidelity feedback can mislead a loop, an architecture environment must also defend itself against its own abstractions. Simulators, analytical models, profilers, and compiler cost models are not neutral oracles. They encode workload choices, warm-up rules, timing models, memory-system assumptions, compiler defaults, branch predictor state, cache initialization, interconnect models, and sampling choices. A method that searches aggressively can discover weaknesses in those assumptions just as easily as it can discover a good architecture candidate.

When the proxy is a simulator, *proxy mismatch*, the divergence between what a cheap stand-in metric rewards and what the true objective needs, takes a specific form, a gap between the behavior an environment reports and the behavior that would matter at the next stronger fidelity level, such as a more detailed simulator, synthesis, physical design, emulation, silicon, or deployment. The failure mode is not merely an inaccurate number. It is a loop that learns the wrong lesson. A mapping optimizer may exploit a memory model that omits contention. A compiler autotuner may win by relying on a backend assumption that changes under a different target. A hardware generator may improve a cycle-level metric while creating timing, congestion, or verification problems that the current environment cannot see. A workload harness may reward one benchmark version while hiding drift in the software stack. A run that does not crash is not the same as hardware validity; the environment has to define which illegal states, unsupported configurations, and silent out-of-model behaviors it can actually reject.

Failure mode: When the tool becomes the target

A common failure pattern is not that the tool is useless; it is that the loop learns exactly what the tool rewards. This is a local instance of proxy mismatch, the architecture version of what machine-learning safety calls *specification gaming* or *reward hacking* (Amodei et al., 2016). A candidate can improve a proxy by choosing an unsupported parameter corner, relying on a stale workload slice, or shifting cost into a compiler/runtime path the current harness does not measure. The environment should make that failure visible through invalid-action checks,

baseline replay, sensitivity tests, and escalation to a stronger feedback source. Chapter 7 treats proxy mismatch and its calibration defenses in full.

The clearest illustration comes from the field that named the failure.

Field note: The boat that won by crashing

In a boat-racing game used to study reinforcement learning, an agent rewarded on game score stopped racing and looped in a lagoon to harvest respawning targets, catching fire and never finishing the race while out-scoring the average human player by about 20 percent (Clark and Amodei, 2016). Nothing malfunctioned. The loop optimized exactly the number it was given, and that number was a proxy for "win the race" that the race never actually checked.
Takeaway. An architecture environment that hands a tool a scalar reward invites the same move on a simulator's blind spots, so the environment, not the optimizer, must own the invalid-action and baseline checks that keep the proxy honest.

Environment design should therefore include red-team checks[15] for the feedback source itself. A simulator-backed loop should record warm-up policy, execution- versus trace-driven mode, random seeds, sampled regions, versioned workloads, configuration files, and unsupported states. It should also include rejection tests that look for proxy gaming: cross-checks against another model, sanity constraints on bandwidth and latency, sensitivity studies, invalid-action logs, baseline replay, and escalation to stronger fidelity when the result is surprising or high commitment. The goal is not to distrust simulation. The goal is to make the simulator's authority explicit.

[15] Red-teaming involves intentionally attacking a system to expose vulnerabilities, flaws, and failure modes before deployment.

5.8 Building Environments for New Subfields

Making that authority explicit is also the first move in building an environment from scratch. A useful Architecture 2.0 environment for an AI-assisted loop can start small, one bounded task where an automated tool or method can propose actions, receive observations, hit explicit invalid-action rules, and leave rejection reasons for review. The point is not to build a universal hardware-design platform. The point is to choose a bounded task where actions, observations, constraints, and rejection can be stated cleanly.

To construct this bounded task, the recipe is straightforward.

```
#| lst-label: lst-environment-recipe
#| lst-cap: "**Environment recipe:** Constructing a small, bounded, rejectable Architecture 2.0 e

1. Define the task in architectural language: workload characterization, cache exploration, accel
2. Choose the representation the loop will read and write: configuration file, architecture descr
3. Define the action schema: which fields can change, which are read-only, which combinations are
```

```
4. Wrap the tool path: simulator, compiler, profiler, RTL flow, EDA stage, runtime syst
5. Define observations and feedback: metrics, traces, logs, warnings, errors, generated
6. Define invalid-action semantics: illegal parameter, noncompilable artifact, nonsynth
7. Log provenance and negative traces: tool versions, seeds, workload versions, failed
8. State the human decision rule: what the architect reviews, what can be accepted auto
```

Once built, the readiness test for this environment is simple. Could a second method, student, or research group act inside the same harness without private knowledge from the original author? Could a rejected candidate remain understandable six months later after tool versions, workload revisions, and scripts have changed? If the answer is no, the project may still contain a useful wrapper, but it has not yet produced a durable Architecture 2.0 environment.

This eight-step recipe is intentionally more operational than inspirational. It is what keeps Architecture 2.0 from becoming prompt-to-chip rhetoric. The first useful environment for a new subfield is often narrow: a bounded workload, a small action space, a clear simulator wrapper, and a disciplined log of failures. That is enough to show the loop what it can try, what it can observe, and what evidence matters.

Applying this operational recipe, the lighthouse prompt suggests a good starting point. Instead of trying to build an end-to-end processor designer, start with an XRBench workload slice and a small set of candidate architectural knobs: vector width, local memory, cache size, data layout, or accelerator interface. Define which candidates are invalid, which feedback is cheap, which feedback is expensive, and what evidence would be needed to move from proxy exploration to simulator or synthesis-backed claims. That is a real Architecture 2.0 environment even if it is far from a complete architecture design system.

The same abstraction scales down as readily as it scales up. A single hardware prefetcher makes a perfectly good environment: the actions are prefetch depth and pattern, the cheap feedback is cache hit rate and memory bandwidth from a cycle-level simulator, and the rejection rule is any configuration that breaches a bandwidth or correctness bound. The environment idea is not tied to the size of the artifact; it is tied to having legal actions, observable feedback, and something that can say no. It is complete only after it names the workload trace, simulator version, failed configurations, proxy limits, and the point at which a human escalates from hit-rate evidence to correctness or traffic review.

Design principle: Turn tools into environments

A tool wrapper is credible only when it defines legal actions, observations, costs, invalid states, and rejection paths. A command that returns a number is not yet an environment a loop can act in.

5.9 Environment Validity and Operating Discipline

An environment is useful only if it preserves the semantics of the architectural question. If the workload is wrong, the action space omits the important decision, the simulator hides a constraint, the proxy is uncalibrated, or the logging drops failed candidates, the loop may become more efficient at producing weak evidence.

To maintain this semantic link, environment validity relies on several layers. The workload layer asks whether the inputs, distributions, and software stack match the intended use. The action layer asks whether the loop can change the right architectural variables without violating hidden constraints. The observation layer asks whether the returned metrics are meaningful for the claim. The fidelity layer asks whether the feedback is strong enough for the commitment being made. The provenance layer asks whether the result can be replayed or audited. The rejection layer asks what can stop a candidate when it is illegal, unsupported, misleading, or insufficiently evidenced.

Architect's checkpoint: The Environment Trust Gate

Before trusting an environment's verdict on an AI-generated candidate, confirm:

- the workload, action space, and observations still match the architecture question the loop was intended to solve;
- the feedback fidelity is strong enough for the commitment at stake;
- versions, assumptions, and rejected candidates proposed by the generative method are recorded;
- a rejection authority can stop an illegal, unsupported, or under-evidenced candidate;
- another team could deploy an automated optimizer in this harness without private knowledge.

Operating discipline, the architecture counterpart of MLOps, is the practice of maintaining those layers over time. Workloads drift. Tool versions change. Compiler behavior changes. Benchmarks gain new rules. Models learn stale assumptions. A one-time environment can support a paper; a maintained environment can support a field. That is why the environment should record versions, assumptions, invalid actions, rejected candidates, and changes in the workload distribution.

MLOps: In this chapter, the operating discipline for deployed learning systems that motivates similar versioning, monitoring, drift, rollback, and ownership records in Architecture 2.0 loops.

Design principle: Maintain the environment, don't just build it

A design-loop environment is not built once; it is maintained. Workloads drift, tools change, benchmarks gain rules, and models learn stale assumptions. Keep the loop valid by versioning workloads and tools, preserving provenance and negative traces, monitoring drift, and maintaining architect-owned rejection gates.

This distinction between building and maintaining matters because a one-time environment and a maintained one look identical on the first run and diverge completely by the tenth; the maintenance burden is what turns a one-off script into shared infrastructure.

Table 5.7 states the maintenance work in operational terms. The table is intentionally prosaic because the discipline succeeds only when ordinary drift becomes visible before it invalidates evidence.

Table 5.7: Maintaining the environment preserves loop validity over time: The operating discipline is to version workloads, tools, actions, evidence rules, and ownership so that drift does not silently change what a loop result means.

Drift source	What to record	Failure if ignored	Operating response
Workload and benchmark updates	Versions, scenarios, excluded inputs, telemetry shifts, and coverage changes.	A loop keeps optimizing a stale or unrepresentative slice.	Re-run coverage checks and weaken or invalidate affected claims.
Tool and model changes	Simulator, compiler, EDA, model, prompt, policy, seed, and configuration versions.	Results become incomparable across runs without a clear cause.	Treat version changes as evidence events and preserve old receipts.
Action-space changes	New knobs, deprecated knobs, invalid combinations, and permission boundaries.	A method proposes actions the environment no longer supports.	Update validity checks and record invalid-action traces.
Evidence and rejection rules	Fidelity labels, escalation thresholds, signoff gates, waivers, and reviewer decisions.	A proxy keeps authority after the commitment level has risen.	Recalibrate evidence levels and require higher-fidelity review.
Human ownership	Decision owner, review date, redaction boundary, and escalation path.	Accountability disappears into scripts, tools, or automated systems.	Keep commitment decisions explicit and auditable.

5.10 Conclusion

This chapter asked what turns a tool flow into an environment a generative model can act in and a reviewer can trust. In Architecture 1.0, simulators, compilers, profilers, RTL flows, and deployment logs sit behind the architect as ways to gather evidence. In Architecture 2.0 they have to move inside the loop and become explicit environments, places that define legal actions, return meaningful observations, price each evaluation, carry constraints and provenance, and let a rejection authority stop an illegal, unsupported, or under-evidenced candidate. That is the whole content of turning a tool into an environment.

An environment is only as good as the architectural question it preserves. If the workload is unrepresentative, the action space omits the real decision, the proxy is uncalibrated, or failed candidates go unlogged, the loop just becomes more efficient at producing weak evidence. Validity therefore has to be checked layer by layer, across workload, action, observation, fidelity, provenance, and rejection, before an environment's verdict is worth believing.

The second, easily missed half is maintenance. Workloads drift, tools change, benchmarks gain rules, and models learn stale assumptions, so a one-time environment and a maintained one look identical on the first run and diverge completely by the tenth. Operating discipline, the architecture counterpart of MLOps, is what versions workloads and tools, preserves provenance and negative traces, monitors drift, and keeps architect-owned rejection gates alive. Building the environment earns a paper. Maintaining it is what lets a loop, and a field, keep trusting the evidence.

5.11 Open Research Questions

The concepts in this chapter establish the foundation for AI-assisted architecture environments, but they also expose major gaps in how AI-assisted loops interact with legacy tools. The following thesis-level challenges push beyond wrapper engineering toward self-learning, verifiable, and evidence-gated environment infrastructure capable of seeding high-impact research at ASPLOS, ISCA, or NeurIPS.

1. **Can agents infer formal environment contracts from unstructured tool exhaust?** Legacy EDA flows and simulators emit opaque warnings, timeouts, and unstructured logs rather than clean APIs. A major open challenge is whether language models can observe thousands of such interactions and help synthesize action schemas, constraint bounds, and rejection authorities defined in the environment contract (Table 5.4), making environment boundaries more explicit for human review.

2. **How can generative loops extract and transfer "negative evidence" across mismatched physical and simulated environments?** While positive architectural wins are highly context-dependent, failures (e.g., routing congestion, IR drop, timing violations) expose fundamental physical limits. As outlined in the EDA stage contract (Table 5.3), a critical open problem is formalizing how this "dark matter" of rejected candidates from slow, high-fidelity physical flows can be automatically distilled into reusable constraints for fast, early-stage proxies evaluating completely different workloads or technology nodes.

3. **Can we reduce routine review with risk-aware escalation while preserving architect-defined commit gates?** The economy of evidence regarding feedback latency and fidelity (Table 5.6) currently relies on static heuristics to promote candidates from cheap analytic proxies to expensive cycle-level simulators. A thesis-level challenge is building environments that dynamically quantify proxy-mismatch risk, triggering high-fidelity evaluation when a surrogate model's uncertainty bounds breach a stated threshold and reducing proxy gaming by design.

4. **How do we engineer self-healing infrastructure for continuous architectural claim recertification?** As workloads drift, software stacks mutate, and tool versions evolve, the validity of static architectural baselines decays. Building on the operating discipline for ArchOps (see the discussion on "Environment Validity and Operating Discipline" in Section 5.9), a fundamental systems challenge is designing governance environments that detect semantic shifts, dispatch AI-assisted verifiers to re-evaluate historical data, and recertify or reject past claims under human-defined review gates.

What to carry forward

- **Reader test:** Could another method act in the same harness without private knowledge from the original author, judged by loop conditions rather than fashion?
- **Up next:** Once the environment defines what action and feedback mean, the discussion of methods for generation, prediction, and optimization (see the discussion on "Method Roles: Generate, Predict, Optimize" in Chapter 6) can ask which methods belong inside the loop.

Chapter 6

Method Roles: Generate, Predict, Optimize

"If you cannot measure it, you cannot improve it."
— Lord Kelvin, *Popular Lectures and Addresses* (1891)

Author's Note: Lord Kelvin, the pioneering British mathematician and physicist, emphasized that measurement is the prerequisite for improvement. For our purposes—whether using search, reinforcement learning, or optimization—this means that no AI method can work without a measurable objective function to provide a rigorous reward signal.

The crux

Which role should an AI method play in the loop, and what feedback would make its output worth trusting?

Chapter 5 defined the environment, the place where actions are taken and feedback is observed. This chapter asks which methods belong inside that environment. The answer is not a ranking of current models or automation frameworks. It is a discipline for matching method roles to architecture work. The forcing function is that AI systems can now draft, call tools, revise artifacts, and recommend next actions, so the loop must say what each action is allowed to mean.

The distinction matters. A model that can generate plausible schedules, configuration files, code fragments, RTL snippets, or design prose may be useful for drafting alternatives, but weak for choosing among them. A surrogate may predict latency or energy well inside a calibrated accelerator design region, but fail outside the sampled space. Bayesian optimization[16] may spend scarce simulator or synthesis evaluations carefully, but only if the action space and objective are well formed. Reinforcement learning[17] may be attractive for placement, scheduling, or adaptive control, but dangerous when invalid actions are common and feedback is delayed. A critic may be more valuable than a generator when the urgent problem is exposing missing workload evidence, not proposing more candidates.

[16] A strategy for the global optimization of black-box functions that uses a probabilistic model to balance exploration and exploitation.

[17] A machine learning paradigm where an agent learns to make sequential decisions by receiving rewards or penalties from its environment.

Architecture 2.0 therefore treats methods as roles in a compound design system. These roles collapse into the classic components of a closed-loop controller, mapping directly to our five-part execution state:

1. **Generators (Actuation):** Propose candidate implementations, hypotheses, or repairs (*actions allowed*).
2. **Predictors (State Estimation):** Evaluate candidate properties, checking constraints and estimating performance (*state seen* and *evidence*).
3. **Optimizers (Search Policy):** Navigate the design space, deciding which branches to evaluate, critique, or prune (*alternatives rejected*).

To make these three core roles usable inside AI-assisted loops, the loop can expand them with five supporting roles: *critique* (finding flaws), *repair* (fixing broken artifacts), *verifi-*

cation (independent sign-off), *explanation* (making decisions legible), and *coordination* (routing tasks).

The unifying schema across these roles is the trust envelope: what state they can see, what boundaries they may push, and what evidence forces them to stop or escalate. The method question is therefore not "Which AI system is best?" It is "Which role is needed, what feedback can support it, and what evidence would make its output credible?"

Machine learning for architecture did not begin with recent generative methods. Architecture has always relied on quantitative evaluation; a useful historical shorthand for ML's entry into the field is prediction, optimization, and generation. Prediction appears in regression models, learned surrogates, and calibrated performance or power estimators. Optimization appears in Bayesian optimization, autotuning, reinforcement learning, and search over compiler, mapping, placement, or architecture parameters. Generation now appears in natural-language-to-code, RTL, test, configuration, and design-report artifacts. The shorthand is useful because it gives proper credit to earlier work. It is also incomplete. The Architecture 2.0 question is how these roles compose with critique, repair, verification, explanation, provenance, negative traces, and human rejection authority inside one explicit loop.

> **What this chapter gives you**
>
> After this chapter you can turn "we used AI method X" into the method-role view of the design-loop card. That means you can:
>
> - match an AI method role (generate, predict, optimize, critique, repair, verify, explain, or coordinate) to an architecture task;
> - distinguish role composition from stronger evidence when several automated methods participate;
> - state an AI method claim as object, action, feedback fidelity, rejection condition, commitment boundary, and decision owner;
> - assess hardware awareness as a staged capability, not the use of hardware vocabulary;
> - decide when critique, repair, or verification is more valuable than generating more candidates;
> - choose an AI method by loop conditions rather than by fashion.

6.1 Match the Method to the Architecture Task

The first step is to name the architecture task. Design-space exploration, workload characterization, benchmark construction, code generation, RTL repair, compiler/runtime tuning, accelerator search, chiplet partitioning, physical-design assistance, and evidence critique are not the same problem. They expose different state, allow different actions, tolerate different errors, and require different feedback.

This is why environment work such as ArchGym, the worked example of Chapter 5, matters here. It makes method comparisons meaningful by defining tasks, actions, observations, workloads, and feedback (Krishnan et al., 2023). But even a shared environment does not decide which method role is appropriate. The role depends on what the loop is trying to accomplish.

Figure 6.1 establishes the chapter's working taxonomy of these roles. It demonstrates that a single architecture loop can concurrently host several distinct method archetypes—such as generators to propose novelty and predictors to estimate cost—but crucially, each role assumes a different claim and carries a different burden of evidence within the loop.

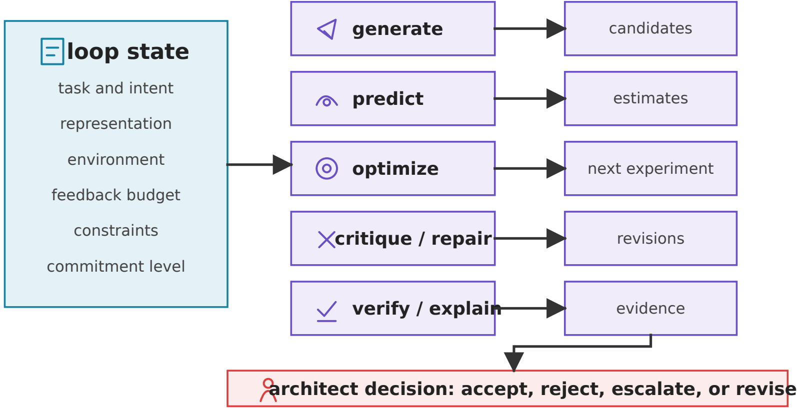


Figure 6.1: **Method roles are useful only inside a represented loop:** Generation, prediction, optimization, critique, repair, verification, explanation, and coordination each need explicit state, feedback, and rejection.

Read the figure as a role map, not as a claim that every loop needs every role. A loop can use generation for breadth, prediction for cheap ranking, optimization for sample allocation, critique for missing assumptions, repair for invalid artifacts, verification for independent rejection, and explanation for reviewable evidence. Coordination is the routing role that keeps those actions attached to shared state and authority boundaries. The method selection question is which role the current loop can support with feedback.

Table 6.1 provides the checklist form of this map. Read each row as a rigorous review prompt: what is the specific role doing, what evidence supports its output, and what failure mode must the loop be prepared to reject? This checklist anchors AI capabilities to tangible architectural tasks.

Table 6.1: **Each method role needs different evidence:** Generation, prediction, optimization, critique, repair, verification, explanation, and coordination make different claims and therefore require different rejection checks.

Role	Architecture use	Evidence needed	Failure mode
Generate	Propose configs, specs, code, RTL fragments, test benches, design reviews, or hypotheses.	Validity checks, constraints, provenance, and human review.	Plausible but invalid candidates.
Predict	Estimate performance, energy, area, latency, reliability, or cost before full evaluation.	Calibration, uncertainty, coverage, and held-out checks.	Confident extrapolation outside support.
Optimize	Choose the next candidate or region of the space to evaluate.	Objective, constraints, feedback budget, and stopping rule.	Gaming the proxy or missing the real tradeoff.
Critique/repair	Find weak assumptions, missing evidence, invalid actions, or broken artifacts.	Access to artifacts, claims, evidence, and rejection authority.	Polished explanations without authority to reject.
Verify	Check constraints, invariants, tests, tool outputs, and evidence ledgers.	Independent checks, provenance, and escalation rules.	Treating one tool pass as final truth.
Explain	Make tradeoffs, failures, uncertainty, and rejected alternatives legible to a reviewer.	Traceable evidence, assumptions, contrast cases, and stated limits.	Convincing story without provenance or support.
Coordinate	Route state, tasks, tool calls, and evidence among role-specific tools or autonomous systems.	Shared state schema, authority boundaries, logs, and stop or escalation rules.	Hidden delegation, duplicated authority, or consensus without independent rejection.

The discipline behind the table is simple. A method claim is incomplete until it names the architecture object being changed or estimated, the interface through which the action is legal, the feedback fidelity that supports the claim, the condition that can reject it, and the commitment level and decision owner the evidence can support.

Engineer move: State every AI method claim concretely

Write the AI method claim as a sentence. This method's role acts on this architecture object through this interface, receives this feedback at this fidelity, can be rejected by this evidence, and supports only this commitment level owned by this named decision owner.

Table 6.2 gives concrete examples. The same method family can be reasonable or unreasonable depending on which object it touches and what can say no. A generator that proposes benchmark questions is different from one that proposes RTL. A predictor that ranks early simulator configurations is different from one that claims final power. An optimizer that chooses the next cheap proxy run is different from one that commits a physical-design change.

Table 6.2: **Method claims need object discipline:** A result is more credible, comparable, and reviewable when it states whether the method acted on prompts, code, traces, configs, RTL, reports, or decisions and what evidence can reject that action. Comparability requires matching object, action, feedback, rejection, and commitment fields.

Role	Object to name	Feedback to require	Rejection condition
Generate	Workload variant, simulator config, tensor schedule, kernel, RTL fragment, EDA constraint, or design-loop card.	Parser, compiler, simulator, test harness, constraint checker, or human review.	Invalid syntax, unsupported action, wrong output, violated constraint, or missing provenance.
Predict	Latency, energy, area, memory traffic, timing risk, thermal behavior, queueing delay, or deployment impact.	Calibration data, uncertainty, coverage region, held-out checks, and fidelity label.	Out-of-support query, counterexample, proxy mismatch, or uncalibrated extrapolation.
Optimize	Next design point, parameter region, schedule, dataflow, mapping, placement move, or experiment allocation.	Objective, constraints, feedback budget, cost model, and stopping rule.	Proxy gaming, infeasible candidate, lost Pareto tradeoff, or exhausted evidence budget.
Critique/repair	Benchmark claim, simulator log, configuration file, test bench, evidence ledger entry, evidence table, or rejected alternative.	Access to artifact provenance, claims, assumptions, and comparison baseline.	Missing workload coverage, stale tool version, unsupported conclusion, or unresolved failure.

Role	Object to name	Feedback to require	Rejection condition
Verify	Invariant, interface contract, numerical tolerance, synthesis constraint, regression result, or evidence ledger.	Independent checks, replayable commands, tool logs, and escalation record.	Failed check, inconsistent evidence, weak fidelity, or architect refusal to commit.
Explain	Tradeoff, failure, rejected candidate, model uncertainty, or tool warning that a reader must understand.	Traceable evidence, assumptions, contrast with alternatives, and limits of the explanation.	Explanation without provenance, unsupported causal story, or hidden uncertainty.

6.2 Hardware Awareness as Staged Capability

Naming the architecture object a method touches raises a prior question, whether the method actually understands the hardware behind that object, or only borrows its terms. The staged view that follows makes the crux's second half concrete for hardware. The level of hardware awareness a method can be held to is set by the level of feedback that can reject its output. Hardware awareness is not the same as using hardware vocabulary. A generated proposal can mention caches, vector units, power targets, or a process node and still be unaware of the architectural consequences of those terms.

Hardware awareness. In this chapter, hardware awareness means that a method or AI-assisted system can represent hardware-relevant constraints, act within a valid hardware/software design space, obtain feedback from appropriate tools or measurements, and expose evidence that can reject its own output.

This definition matters because Architecture 2.0 methods will often generate or revise artifacts near the hardware/software boundary: kernels, compiler settings, accelerator configurations, RTL fragments, memory-system choices, placement constraints, or design reports. Hardware-aware neural architecture search already shows the value of putting latency, energy, memory footprint, and target-device cost into the search problem (Benmeziane et al., 2021). Architecture 2.0 uses a broader version of the same discipline. The question is not whether an artifact sounds hardware-aware, but what level of hardware-aware action and evidence the loop can support.

Neural architecture search (NAS): In this chapter, a cautionary adjacent field where search only becomes reusable when the search space, data, training budget, evaluation rule, and rejected alternatives are recorded.

Figure 6.2 establishes the working capability map for evaluating these methods. The levels are cumulative as an assessment vocabulary, but they are not a claim that every system improves along one monotone axis. A tool wrapper may compile and profile without having a strong performance model. A calibrated surrogate may estimate uncertainty without directly controlling a tool. The staging is useful because it forces

the reader to ask which capability a method actually has, which capability it lacks, and what the loop is allowed to do with the result.

Vocabulary awareness is useful, but it only names the objects. Constraint awareness applies declared budgets and limits such as power, area, latency, memory, precision, reliability, and process assumptions. Performance-model awareness reasons about the mechanisms that drive cost: data movement, locality, bandwidth, occupancy, parallelism, pipelines, and communication. Here, "performance model" means a cheap analytical, learned, or surrogate estimator used before live tool execution. Tool and environment awareness connects those mechanisms to compilers, profilers, simulators, synthesis reports, and failed runs; a simulator such as gem5 can itself be a tool environment when the loop invokes it, records configurations and outputs, and treats its result as feedback rather than as an abstract ranking. Evidence awareness adds calibration, uncertainty, proxy mismatch, fidelity, and negative traces. Commitment-boundary awareness is the highest level because the loop can state what the method may recommend, what must be escalated, and why the architect still owns the commitment.

Proxy mismatch and Goodhart's Law. The proxy mismatch first named in Chapter 5 is an instance of Goodhart's Law (Strathern, 1997). When a measure becomes a target, it ceases to be a good measure. Chapter 7 treats it in full.

Architect's checkpoint: The Escalation Boundary

When the method reaches the limit of its evidence fidelity or encounters a constraint it cannot safely resolve, it must escalate to the architect. The decision gate here relies on the AI loop's ability to clearly state its uncertainty and preserve negative traces, allowing the human architect to make the final commitment.

Functional correctness is cross-cutting, not optional, and forms the baseline for the progression shown in Figure 6.2. Once a loop changes an executable or synthesizable artifact, tests, reference outputs, formal checks, type or interface checks, synthesis constraints, numerical tolerances, or expert review must be able to reject it before performance claims matter.

The ladder is deliberately behavioral. A method that only uses hardware words is at the vocabulary level. The assessment rises only when the loop can show valid actions, constraint handling, tool feedback, calibration evidence, and a clear architect-owned commitment boundary.

Table 6.3 makes this assessment explicit. Each level should be judged by three concrete criteria: what the method is permitted to change, what feedback supports that change, and what authority can reject it. This provides a structural mechanism for auditing whether a method's claims match its actual capabilities.

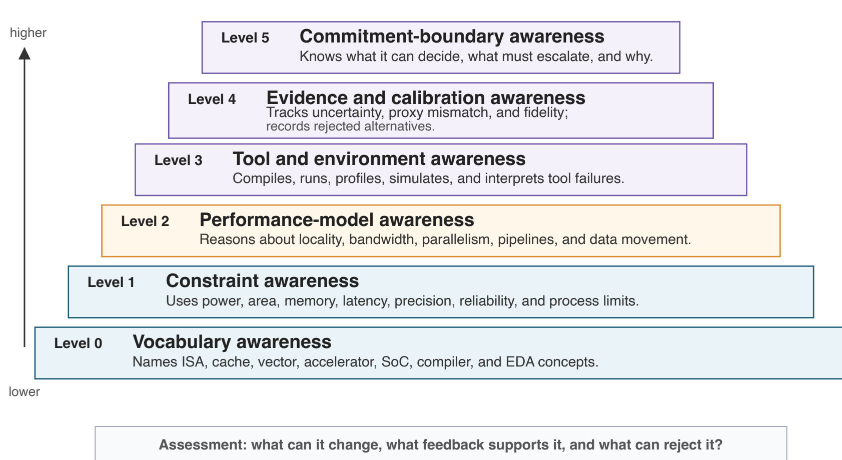


Figure 6.2: **Hardware awareness is a staged capability:** The assessment is not whether a method can mention hardware terms, but what it can safely change, what feedback supports the change, and what independent mechanism can reject it.

Table 6.3: **Hardware awareness should be assessed by behavior, not vocabulary:** The loop must show whether a method can represent constraints, take valid actions, obtain feedback, expose evidence, and reject its own output.

Capability	What it may change	Feedback needed	Rejection authority
Vocabulary	Terms in prompts, reports, or design notes.	Human review of meaning and misuse.	Architect rejects fluent but empty claims.
Constraint	Candidate fields within declared budgets or limits.	Bounds, static checks, invalid-action filters, and feasibility rules.	Constraint violation or illegal action.
Performance model	Rankings, estimates, or parameter choices inside model support.	Calibration, sensitivity, residuals, and held-out checks.	Model miss, counterexample, or out-of-support query.
Tool/environment	Code, configs, kernels, traces, or tool-invoked candidates.	Compile, run, profile, simulate, synthesize, and log tool outcomes.	Failed test, tool error, profile regression, or invalid artifact.

Capability	What it may change	Feedback needed	Rejection authority
Evidence and calibration	Confidence, evidence strength, and escalation recommendations.	Multi-fidelity comparison, uncertainty, provenance, and negative traces.	Proxy mismatch, weak provenance, or fidelity failure.
Commitment boundary	Recommend, explain, or escalate within stated limits.	Evidence ledger, rollback cost, risk, and review context.	Human architect or independent verification refuses commitment.

However, the previous table is still too abstract unless each level has a falsifiable test. Table 6.4 provides a practical reviewer version. A method should not claim a higher level unless it can pass the observable test for that level in the current loop. This ensures that hardware awareness remains an empirical property rather than a rhetorical claim.

Table 6.4: A hardware-awareness claim should be falsifiable: Reviewers should ask for observable tests, not only fluent hardware language.

Claimed level	Reviewer test	Evidence to attach
Vocabulary	Can the method use architecture terms without changing their meaning?	Human review notes or corrected definitions.
Constraint	Does it reject illegal parameter combinations before scoring them?	Constraint logs, invalid-action traces, and rejected examples.
Performance model	Does it state the calibrated range and fail or escalate outside it?	Calibration plot, held-out error, sensitivity result, or uncertainty record.
Tool/environment	Can it run the tool path and preserve both successes and failures?	Commands, versions, seeds, logs, failed runs, and replay instructions.
Evidence and calibration	Can it explain why one feedback source is strong enough for one claim and too weak for another?	Fidelity labels, proxy-mismatch examples, and escalation thresholds.
Commitment boundary	Can it recommend or escalate without pretending to own the decision?	A shareable evidence ledger, review note, residual risk, and named decision owner.

The level label is not the claim. The claim is the method-role view the loop can actually support on the design-loop card: object, interface, feedback fidelity, rejection rule, commitment boundary, and decision owner.

A kernel-generation loop that can compile, test, and profile generated kernels has tool awareness. It does not automatically have evidence awareness unless the loop records

numerical correctness, speedup distributions, portability, rejected candidates, and proxy mismatch. A design-space generative method that can propose accelerator parameters has constraint awareness only if invalid actions are blocked or rejected. It reaches commitment-boundary awareness only if the loop can connect workload intent, hardware resource limits, compiler/runtime assumptions, fidelity gates, and human decision points into one accountable loop.

6.3 Generation: Proposing Candidates and Artifacts

With hardware awareness established, we can examine specific method roles. Generation is the most visible AI role because fluent artifacts are easy to demonstrate and easy to overtrust. A method can draft an architecture description, propose a simulator configuration, generate code, produce a test bench, write a design-review summary, suggest a memory hierarchy, sketch an accelerator interface, or enumerate hypotheses about a workload. In the lighthouse example, generation might propose a set of candidate vector widths, local memory sizes, data layouts, or CPU/accelerator partitions for the XRBench workload.

The danger is to mistake candidate generation for design. Generated artifacts are useful when they expand the set of possibilities, expose alternatives, or accelerate tedious translation. They are not credible merely because they are well formed. A generated RTL fragment that is syntactically valid might still fail to route or close timing during physical design. The loop still needs validity checks, tool execution, evidence, rejected alternatives, and human decision.

A generated RTL fragment, parameter set, or benchmark question is a proposal. Generation earns its place only when it feeds a loop that can test, reject, compare, and revise. Furthermore, candidate generation extends beyond writing direct code; for hardware, searching the High-Level Synthesis (HLS) pragma space or mutating programmatic generator source code (e.g., Chisel/Scala templates) is often a richer, more verifiable generative target than static configuration sweeps or raw RTL synthesis. Similarly, for spatial architectures (like tensor accelerators), the AI loop must explicitly manipulate *dataflow*—the spatial and temporal unrolling of operations—as a primary structural contract, not just a scalar hyperparameter.

Failure mode: Treating AI generation as design

It is tempting to treat a fluent AI-generated artifact, an RTL fragment, a config, a benchmark question, as a result. It is not. An AI-generated artifact is a proposal; it becomes a result only after the AI-assisted loop tests it, prices its evidence, compares it against a baseline, and an architect accepts the commitment. AI generation that is not embedded in a loop that can reject it is demonstration, not design.

The strongest near-term use of generation may be breadth. It can propose candidate decompositions, list assumptions, create alternative experiment plans, translate design intent into structured records, or draft the first version of a design-loop card (Appendix B gives the full card and rubric). Those outputs are valuable because they give the architect more structured material to inspect. They become dangerous only when the loop treats them as decisions.

Kernel generation is a useful concrete case because it isolates the software and code-generation facet of the lighthouse prompt. The XRBench subsystem only matters if kernels, libraries, runtime paths, and target-specific code can be generated, checked, and maintained. KernelBench asks whether language models can generate correct and efficient GPU kernels for PyTorch workloads (Ouyang et al., 2025). The Architecture 2.0 lesson is not that kernel generation solves hardware/software co-design. It is that generation becomes meaningful only when it is embedded in a harness that can compile, run, test, profile, compare against a baseline, reject wrong outputs, and preserve negative traces. Follow-on kernel-generation benchmarks make the lesson sharper: multi-platform settings expose backend and portability contracts (Wen et al., 2025), while category-aware analyses show that correctness, task structure, numerical contracts, and efficiency can diverge (Wang et al., 2026). That is precisely why generation is a role in a loop, not the loop itself. Chapter 9 makes the point quantitative. The loop is rejection-bound, and candidate count does not enter its throughput bound at all, so generating more proposals cannot speed a loop whose cheap, independent rejection coverage stays fixed. The transferable object is the harness contract: target platform, correctness oracle, numerical tolerance, baseline, failed kernels, portability boundary, and the rejection rule that stops a fast but wrong artifact.

The same discipline appears closer to hardware. Chip-Chat reports a conversational loop in which a language model drafts Verilog, open-source simulation and synthesis tools check it, the resulting errors are fed back for revision, and a small processor design is carried as far as fabrication (Blocklove et al., 2023; Thakur et al., 2023; He et al., 2024). The interesting part is not that a model produced Verilog. It is that the loop could support a bounded, reviewable commitment because a parser, a simulator, and a synthesis flow could each reject a draft, and a human stayed in the conversation to decide when a candidate was good enough to commit. Generation supplied breadth; the environment and the architect supplied the authority to reject.

6.4 Prediction: Estimating Behavior Before Full Evaluation

While generation supplies candidate breadth, prediction is central because architecture feedback is expensive. Long before recent foundation models[18], architects used statistical and machine-learning models to reduce the cost of exploring large design spaces. Regression models for microarchitectural performance and power, and predictive modeling for large architectural design spaces, are part of this lineage (Lee and Brooks, 2006; Ipek et al., 2006). Architecture 2.0 uses that lineage only when the predictor

[18] Large-scale machine learning models trained on vast quantities of data that can be adapted to a wide range of downstream tasks.

exposes what an agentic loop needs: the support region, uncertainty, calibration source, escalation trigger, and decision owner.

The prediction role is not limited to performance. A predictor might estimate energy, area, latency, reliability, queueing behavior, memory traffic, thermal behavior, compile time, implementation feasibility, or deployment impact. It might be a regression model, a learned surrogate, a calibrated analytic model, a simulator-backed approximation, or a hybrid that combines domain structure with data. Predicting physical metrics (like power or latency) without accounting for logic synthesis, place-and-route congestion, or timing closure often leads to severe proxy mismatches. Therefore, in an agentic loop, a predictor is not an oracle; it is a quantitative gate that decides whether the system may prune, defer, escalate, or ask for stronger feedback.

Some of these targets are harder to pin down than others. A single-kernel latency can be a fairly clean estimate, while deployment impact compounds power, thermal behavior, and reliability over the life of a design, so it resists any single-point prediction.

The key requirement is uncertainty. A point estimate is useful only if the loop understands where it is valid. Has the predictor seen similar workload regions? Does it extrapolate across a new memory behavior, vector width, or technology assumption? Does it report confidence? Is it calibrated against a higher-fidelity source? Does it preserve enough provenance to explain why a candidate was trusted?

There is a rigorous way to make this concrete when the assumptions are appropriate. Conformal prediction[19] can wrap a surrogate, regardless of its internals, so that it emits calibrated prediction sets, intervals that contain the true value with a user-chosen probability under an exchangeability assumption (Angelopoulos and Bates, 2021). Architecture loops must audit that assumption because workloads, tools, and process corners are often not exchangeable. Still, the pattern is useful. It converts "report confidence" from a slogan into an operation. The predictor writes an interval, calibration receipt, and support-region label into the loop state; the environment may use that record only to prune, escalate, or defer, not to make an unowned commitment. The guarantee does not require assuming any particular error distribution for the surrogate, which is useful for architecture surrogates with awkward error shapes. But it still depends on the calibration assumption being meaningful; under distribution shift, the interval is a diagnostic that should trigger scrutiny rather than a promise that the proxy is safe.

[19] A statistical technique that provides rigorous, distribution-free uncertainty bands for model predictions.

> **Exchangeability:** A statistical property where the joint probability distribution of a sequence of random variables is invariant to their permutation; it is a weaker assumption than independent and identically distributed (i.i.d.).

For the lighthouse prompt, a predictor could help screen candidate compute subsystems before full simulation or synthesis. But the evidence burden depends on the decision. A rough predictor may be enough to discard obviously bad candidates. It is not enough to claim that a design meets a 3 W target on a 3 nm-class low-power mobile process. The stronger the commitment, the stronger the calibration and fidelity requirement.

6.5 Optimization: Learning the Design Space

Once candidates are generated and their behavior predicted, the loop must decide what to evaluate next. Optimization is often framed as search, finding the best point under an objective. Architecture needs a richer formulation. Unlike the continuous, differentiable weight spaces of neural network training, hardware design spaces are typically discrete, non-differentiable, and highly constrained. This makes gradient descent impossible and rewards methods that handle sparse, expensive feedback. The useful goal is to learn the design space well enough to make a defensible decision under limited feedback. That may mean finding a Pareto region, identifying a constraint boundary, understanding a sensitivity, ruling out a class of candidates, or deciding which expensive experiment is worth running next.

Pareto region: The set of optimal solutions in a multi-objective design space where no single objective can be improved without degrading at least one other objective.

Optimization methods are useful here when they make the loop's sampling policy representable and rejectable. Bayesian optimization is attractive for Architecture 2.0 because it was built for expensive black-box functions and sequential experimentation (Jones et al., 1998; Snoek et al., 2012). It encourages the loop to trade off exploration and exploitation, to reason about uncertainty, and to spend evaluations where they are likely to matter. Those properties align naturally with architecture settings where simulator, synthesis, or measurement runs are costly. In Architecture 2.0, the policy that picks the next sample, the *acquisition policy*, is itself a logged action proposal. It must name the candidate, the fidelity level, the expected decision value, and the condition under which the sample will be rejected or escalated. For an AI-assisted system, optimization is therefore not autonomous selection of the best design; it is an auditable policy for spending scarce feedback.

Reinforcement learning is attractive when the problem is sequential: placement decisions, scheduling policies, adaptive control, or multi-stage design flows. The chip-floorplanning literature gives a prominent, and contested, example of posing a chip design subproblem as a learning problem (Mirhoseini et al., 2021). Chapter 7 discusses the later baseline and reproducibility challenge. The important lesson for this chapter is not that every architecture task should become RL. It is that method choice depends on state, action, transition, feedback, and commitment structure.

A narrower example shows what happens when the design space is bounded tightly enough that the environment can supply real rejection authority. PrefixRL casts the design of parallel-prefix arithmetic circuits, such as adders, as a reinforcement-learning problem with logic synthesis in the loop, so every proposed circuit is scored by an actual synthesis run rather than a hand-built proxy (Roy et al., 2021). The important receipt is that each candidate was a bounded circuit object, evaluated by a synthesis-backed environment, and rejected or advanced by evidence stronger than a hand-built proxy. The contrast with the placement dispute is the lesson, not the leaderboard. The same method family is contested when its reward is a fast proxy and defensible when a bounded action space lets a high-fidelity instrument reject every candidate. Method choice is inseparable from what the environment can verify.

Autotuning and compiler optimization provide another useful precedent. For example, deep reinforcement learning has been successfully applied to discover faster sorting algorithms from assembly instructions (Mankowitz et al., 2023) and to navigate complex high-level synthesis phase orderings (Haj-Ali et al., 2020). General program-autotuning frameworks such as OpenTuner (Ansel et al., 2014), and tensor-program optimizers such as AutoTVM and Ansor, share a loop structure worth naming: a learned cost model proposes promising candidates, the loop compiles and measures the strongest of them on real hardware, and those measurements correct the cost model for the next round (Chen et al., 2018; Zheng et al., 2020). That is an active-learning loop running in production. The cheap proxy is never trusted alone, because high-fidelity feedback continually re-grounds it. These systems are not identical to microarchitecture design, but they are lighthouse facets rather than unrelated detours. Floorplanning stresses high-commitment physical feedback; autotuning stresses the compiler/runtime path that makes specialization usable. Both illustrate a durable pattern. A method is powerful when it is embedded in an environment that exposes legal actions, measures feedback, updates a cost model, and records results. For Architecture 2.0, the transferable pattern is the record: candidate schedule, compiler/runtime version, measurement context, failed variants, cost-model update, and the human-owned boundary on what the measurement can prove.

The optimizer should therefore be evaluated by what it learns and what it can explain, not only by its best score. Did it discover a robust region? Did it identify a proxy mismatch? Did it spend high-fidelity evaluations carefully? Did it preserve rejected alternatives? Did it show why one candidate was chosen over another? If the answer is no, the loop may have optimized a number without improving architectural understanding.

6.6 Sample Efficiency Under Expensive Feedback

Whether an optimizer spends its scarce high-fidelity evaluations carefully is itself the problem of sample efficiency, which is one of the reasons architecture is a hard AI domain. Many AI settings assume abundant feedback. Architecture often has the opposite shape: a small number of high-fidelity runs, a larger number of medium-fidelity simulations, many cheap proxies, and a long tail of hidden costs such as expert review, tool setup, debugging, and license availability.

Sample efficiency: In this chapter, the amount of decision-relevant evidence a loop gains per scarce feedback event, including failed runs and expert rejections, not just successful simulations.

Figure 6.3 visualizes this fundamental mismatch on a logarithmic count scale, juxtaposing the combinatorial explosion of architecture design spaces against the severe constraints of real-world sample budgets. The upper rows reuse the design-space anchors from Chapter 2: the 12,960-state slice computed there, the accelerator DSE scale of MAESTRO, an analytic model for evaluating DNN dataflows, AutoTVM-style operator tuning spaces, and the core mapspace expression of Timeloop, a mapping and evaluation framework (Kwon et al., 2019; Chen et al., 2018; Parashar et al., 2019). The lower rows reuse the

representative feedback budgets from Chapter 5. These counts are not identical scientific quantities. The point is the scale mismatch. High-fidelity architecture evidence can touch only a tiny slice of the plausible candidate space, so methods must decide what can be screened by proxies, what should be escalated, and what rejected regions must be recorded.

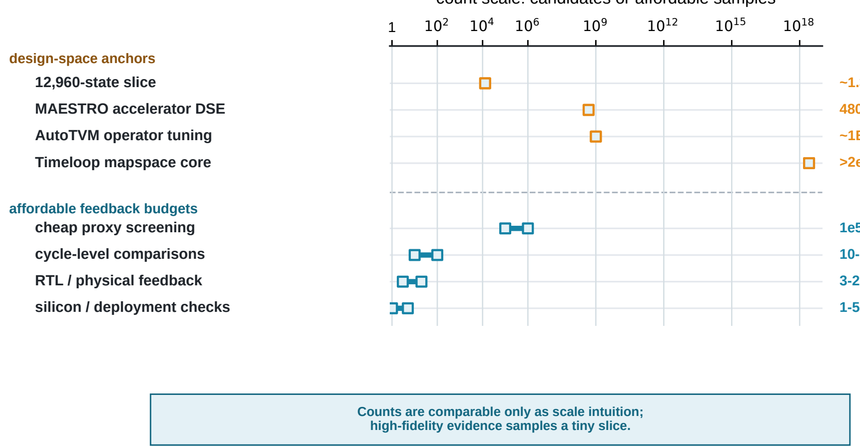


Figure 6.3: **Architecture methods face an evidence gap:** Design spaces can contain orders of magnitude more candidates than a loop can afford to test at stronger feedback levels. The plot compares counts only as scale intuition: source-backed or transparent design-space anchors above the divider, representative sample-budget ranges below it.

Chapter 4 treated sample cost as data the representation must carry. Here the same idea becomes a method-selection criterion. A sample is any feedback event that changes what the loop believes: a simulator result, tool warning, failed run, synthesis report, benchmark measurement, expert rejection, or higher-fidelity validation. The loop should not maximize sample count. It should spend feedback where decision value per unit cost is highest:

$$V_{\text{sample}} \approx \frac{\Delta D}{C_{\text{sample}}},$$

where ΔD denotes the change in decision confidence, rejected-space coverage, or evidence strength produced by that sample. Neither term is directly measurable; the relation is a way to rank candidate experiments by what they would resolve, not a number to compute and literally optimize.

This heuristic has a rigorous counterpart. Bayesian optimization makes the choice of the next sample an explicit acquisition function (a rule that scores each candidate next-sample) over a surrogate's posterior (its current probability estimate over outcomes), and GP-UCB[20] (Gaussian Process Upper Confidence Bound) gives that choice a no-regret guarantee under stated assumptions, so the loop can defend why it sampled where it

[20] An acquisition function that selects the next point to evaluate by weighing both the expected reward and the uncertainty of a Gaussian Process model.

did rather than appealing to intuition (Srinivas et al., 2010). Multi-fidelity Bayesian[21] optimization goes one step further and treats the fidelity level itself as a decision variable. It chooses not only which candidate to evaluate but whether to spend a cheap proxy or an expensive simulation on it, exactly the choice the fidelity ladder poses (Kandasamy et al., 2017). The architecture loop usually needs the contract more than the proof apparatus: what rule chose the sample, what assumptions made that rule legal, what evidence came back, and what threshold forced escalation. The loop still has to record the acquisition rule, fidelity choice, assumptions, rejected alternatives, and escalation threshold, or the optimizer becomes another opaque source of candidates.

[21] An extension of Bayesian optimization that jointly selects which design to evaluate and at which level of fidelity, optimizing the tradeoff between information gain and computational cost.

No-regret guarantee: In this chapter, a formal promise about a sampling rule under stated assumptions; useful only when the loop records whether those assumptions apply.

Fidelity ladder. A fidelity ladder is the ordered set of feedback levels a loop can climb, from cheap proxies through simulation, synthesis, physical feedback, deployment, or silicon evidence, with higher levels usually costing more but carrying stronger rejection authority.

Table 6.5 provides a simple way to conceptualize these operational regimes. The numbers are illustrative, not prescriptions. The core point is that method choice must change fundamentally when feedback is measured in milliseconds, minutes, hours, weeks, or silicon cycles. The fidelity ladder is therefore a budget discipline: cheap levels buy breadth, and expensive levels buy rejection authority. Methods that ignore this distinction will either exhaust the budget or produce unverified claims.

Table 6.5: **Feedback regime should drive method choice:** Cheap proxies, moderate simulations, expensive EDA, and scarce high-commitment checks reward different mixtures of generation, prediction, optimization, critique, and verification.

Regime	Typical setting	Method implication	Evidence discipline
Many cheap proxy runs	Analytic models, rough estimators, compiler hints.	Broad search, candidate generation, surrogate pretraining.	Track proxy validity and avoid overfitting the cheap metric.
Hundreds of simulations	Simulator-backed DSE or workload sweeps.	Bayesian optimization, active learning, transfer, sensitivity analysis.	Record seeds, configs, workloads, and failed runs.
Tens of expensive tool runs	Synthesis, physical design, emulation, or hardware-in-the-loop.	Strong priors, staged gates, human filtering, small candidate sets.	Require calibration and explicit rejection authority.
Few high-commitment checks	Silicon, deployment, fleet experiments, or customer workloads.	Critique, evidence organization, conservative recommendations.	Human decision and audit trail dominate.

Overfitting the cheap metric is not a hypothetical, and an adjacent field watched it happen at national scale.

Field note: The predictor that tracked winter, not flu

Google Flu Trends estimated influenza prevalence from search queries and, for a time, tracked the official surveillance data closely. Then it drifted, over-predicting flu in 100 of 108 weeks, because the cheap proxy had quietly learned a seasonal winter signal rather than a flu signal, and Google's own changing search algorithm moved the ground under it (Lazer et al., 2014). The proxy was never re-checked against the higher-fidelity surveillance data often enough to catch the drift before it was trusted.

Takeaway. A cheap predictor is only as good as the proxy it optimizes, and a proxy that is never re-validated against the stronger signal will keep scoring well long after it has stopped measuring the thing you care about.

This is where negative traces matter. Failed simulations, invalid candidates, timeouts, rejected configurations, and proxy mismatches are not noise to be discarded. They are information about the boundary of the design space. A sample-efficient loop should learn from what failed, not only from the points that produced clean plots.

Sample efficiency also depends on representation. If the environment logs only final scores, the method cannot reuse much. If it records workload metadata, candidate structure, tool warnings, failure reasons, and fidelity level, then each sample contributes more. Chapter 4 and Chapter 5 are therefore not preliminaries to methods. They determine whether methods can learn.

6.7 Critique, Repair, and Explanation

Beyond generating, predicting, and optimizing, loops often need to evaluate existing artifacts rather than propose new ones. Critique may be the most underrated method role in Architecture 2.0. Many loops do not need a generative method to invent a new design. They need a system that can read a proposed design, identify missing assumptions, check whether evidence matches claims, compare alternatives, find invalid actions, repair artifacts, or explain a tradeoff for review.

This role is especially attractive because it can operate against existing human artifacts. A critic can inspect a design-space report, simulator log, configuration file, benchmark description, or paper draft. It can ask whether the workload matches the claim, whether the metric is a proxy for the real objective, whether rejected candidates are missing, whether tool versions are recorded, or whether a table proves less than the prose claims.

Question-answering resources such as QuArch point toward one piece of this problem, making architecture knowledge accessible to automated optimizers and reviewers (Prakash et al., 2025b). But critique requires more than answering questions from papers. It needs the loop state that papers often omit: assumptions, tool settings, negative traces, evidence ledgers, and human decisions.

This makes critique a throughput role, not merely a review convenience. In a rejection-bound loop, a critic that catches a missing workload record, unsupported proxy claim, stale tool version, or invalid action before an expensive run can improve the loop more than another generator. It raises the rate at which weak claims are rejected safely.

Repair is the constructive side of critique. A method can propose a corrected configuration, fix a malformed constraint or tool-syntax error, patch a test bench, regenerate a plot with the right workload metadata, or produce a clearer design-loop card. Repair addresses malformed artifacts; it must not relax a timing, power, interface, or signoff constraint the design actually violates, because that constraint is a rejection authority, not a broken input. Loosening one requires explicit architect review and a recorded waiver. Explanation then becomes the interface to the architect: why a candidate was rejected, why evidence is insufficient, or why one region of the space is worth more expensive evaluation.

Architect's checkpoint: Constraint Waivers

An automated method may repair malformed artifacts but must never relax a design constraint without explicit architect review. The decision gate here requires the AI loop to explain the violation and wait for a human-recorded waiver before proceeding.

Verification, the remaining checking role, deliberately gets its own chapter. Chapter 7 develops it as the family of rejection authorities a loop needs, not as one more method to pick.

6.8 Choosing a Method Under Constraints

With these distinct roles defined, a good method choice should survive a design review. The reviewer should be able to ask: What task is this method serving? What representation does it read and write? What environment does it act in? What feedback can it afford? What evidence would support its output? What can reject it? What happens if it is wrong?

Table 6.6 provides a compact decision matrix for navigating these tradeoffs. It is not meant to produce an automatic answer, but rather to prevent method selection from being driven by algorithmic fashion. Each question forces the architect to confront the structural realities of their specific loop.

Table 6.6: **Method selection follows loop conditions:** The right method posture depends on action validity, feedback cost, fidelity, rollback cost, and rejection authority, not on whether a technique is fashionable.

Question	If the answer is favorable	If the answer is unfavorable
Is the task bounded?	Use stronger automation inside the boundary.	First decompose the task or keep the method advisory.
Does the loop need more candidates?	Use generation for breadth, with validity checks attached.	Prefer critique, verification, evidence organization, or better rejection tests.
Are actions validatable?	Let the environment reject illegal candidates.	Use generation only with strict human/tool review.
Is feedback cheap enough?	Search, active learning, or online adaptation may be useful.	Use priors, surrogates, staged gates, and critique.
Is uncertainty visible?	Prediction can guide exploration.	Avoid treating point estimates as evidence.
Is the commitment reversible?	Higher autonomy may be acceptable.	Require stronger evidence and human decision.
Is provenance recorded?	Claims can be replayed and audited.	Do not make strong comparative claims.

The matrix also clarifies why the same method may be appropriate in one architecture loop and inappropriate in another. A generator may be acceptable for drafting candidate simulator configs but not for committing a physical design change. A surrogate may be useful for ranking early candidates but not for final power claims. An RL policy may be reasonable in a reversible runtime-control loop but not in a high-commitment design decision without strong rejection authority.

Lighthouse prompt: Screen under the power envelope

Context. For the lighthouse design loop, the prompt is an AI method-selection problem, not a request for more candidates.
In the Lighthouse prompt. The AI-assisted loop must choose among vector widths, local-memory sizes, data layouts, and the “vector-capable CPU, accelerator, or SoC block” partition for the “XRBench-class real-time mobile XR workload” while respecting the “3 W TDP target” and preserving a path to a “design-space report with evidence and rejected alternatives.”
Method role. Use AI generation to propose bounded candidates only if needed, AI prediction to screen candidates with calibrated intervals, and AI optimization to spend scarce simulator, compiler and runtime, synthesis, or verification runs where they resolve the decision.

Takeaway. An AI surrogate may reject obviously bad points, but it cannot by itself claim that a design meets a 3 W target in a 3 nm-class low-power mobile process. The loop must escalate whenever the interval straddles the power or latency bound, or when the win depends on a weak proxy for memory traffic, software reachability, physical feasibility, or correctness.

A useful Architecture 2.0 paper should be able to write the method choice as a sentence. We use this method in this role because the task has this action space, this feedback budget, this evidence burden, and this rejection authority. If that sentence cannot be written, the method choice is probably floating above the architecture problem.

The same rule covers multi-participant implementations. Splitting work across a planner, generator, predictor, critic, verifier, and evidence writer may improve throughput, specialization, or coverage, but it does not itself strengthen the claim. The loop still has to say which participant owns which role, what shared state they read and write, which actions are legal for each, which outputs require independent rejection, and where authority returns to the architect. More participants without that role contract only multiply unreviewable state transitions.

Design principle: Match methods to roles

Do not choose a method until the loop has named the bottleneck it is supposed to relieve. Generation, prediction, optimization, critique, repair, verification, explanation, and coordination are different jobs with different evidence standards; a method earns trust only through the role the loop needs.

6.9 Why No Single Algorithm Wins

Architecture 2.0 should not age around one algorithm family. The field will continue to change: models will improve, automation frameworks will change, tools will expose new interfaces, and benchmarks will evolve. A durable book should therefore encourage method discipline rather than method fashion.

The stable idea is that methods earn trust by their role in the loop. They must match the task, representation, environment, feedback budget, fidelity ladder, evidence standard, and commitment level. They should preserve negative traces, expose uncertainty, and make rejection possible. They should help architects learn the design space, not merely search it harder.

6.10 Conclusion

This chapter asked which role an AI method should play in a loop, and what feedback would make its output worth trusting. The deliberate answer is that the question is about roles, not rankings. Generation, prediction, optimization, critique, repair, verification, explanation, and coordination are jobs inside a design loop, and none of them earns trust by being state of the art. A method earns trust when the loop can say what its output is allowed to mean and what feedback would reject it.

That reframing turns method choice into a budgeting problem. Feedback is the expensive resource, cheap screening and high-fidelity evidence are different currencies, and the sample-efficient move is to let the value of the next decision, not habit or novelty, choose which method acts and at which fidelity. A generator that runs unchecked, or a proxy promoted quietly into committing evidence, spends the budget and buys nothing.

This is why no single algorithm wins, and why the chapter argues for method discipline over method fashion. Models and frameworks will keep changing, but the durable rule holds. Match the method to the role, and let the loop, not the algorithm, be the thing that has to stay credible.

6.11 Open Research Questions

The discipline of assigning specific roles to AI methods exposes several unsettled research directions. Resolving these challenges will require moving beyond static benchmarks to evaluate how AI-assisted systems perform inside dynamic, resource-constrained architecture loops.

1. **What formalisms can rigorously enforce and audit AI role contracts across heterogeneous execution environments?** While guidelines for choosing a method under constraints (see the discussion on “Choosing a Method Under Constraints” in Section 6.8) establish a decision matrix for method selection, enforcing these boundaries at runtime remains a critical open challenge. We need verifiable logging structures, public corpora of loop traces, and formal verification techniques that can mathematically or empirically prove a generator, predictor, or critic strictly adhered to its commitment boundary, rather than silently escalating its own authority. This requires new theoretical frameworks for defining and checking “role compliance” during autonomous design-space exploration.
2. **How can active learning models mathematically synthesize heterogeneous negative traces?** Although discussions of sample efficiency under expensive feedback (see the discussion on “Sample Efficiency Under Expensive Feedback” in Section 6.6) establish a fidelity ladder, it remains an open theoretical problem to continuously update probabilistic priors using a complex, non-i.i.d. mix of cheap tool warnings, proxy mismatches, synthesis timeouts, and rare silicon failures. Next-generation acquisition functions must rigorously ingest this diverse negative trace data to shape

the search space, all without blurring the boundary between cheap screening proxies and high-fidelity evidence.

3. **Can we construct definitive, automated rejection oracles for hardware-aware generative models?** Moving beyond qualitative capability assessments of hardware awareness, the architecture community lacks standardized, adversarial test suites explicitly designed to falsify an automated optimizer's unsupported predictions or trap invalid generative actions. A major systems challenge is developing rigorous, environment-agnostic rejection oracles that formally map the passing of a test suite to the maximum architectural commitment that the suite is empirically allowed to authorize.
4. **What control-theoretic mechanisms should govern the dynamic reallocation of method roles?** The decision-value heuristic introduced for maximizing sample efficiency under expensive feedback (see the discussion on "Sample Efficiency Under Expensive Feedback" in Section 6.6) assumes a static policy for invoking the next method role. A compelling thesis direction is the design of dynamic, meta-level coordinators—perhaps based on optimal control or meta-reinforcement learning[22]—that automatically shift computational resources. Such coordinators would pivot seamlessly from broad generation to targeted critique and repair based on the remaining evidence budget, surrogate uncertainty, and the marginal cost of the next high-fidelity simulation.

[22] A learning paradigm where an agent learns to learn, rapidly adapting to new tasks or environments by leveraging past experience.

What to carry forward

- **Reader test:** Can you write one sentence explaining why this AI method belongs in this role for this AI-assisted loop?
- **Up next:** Once models and methods can act, the next question is whether their feedback becomes evidence strong enough to support trust.

Chapter 7

Feedback, Evidence, and Trust

“Program testing can be used to show the presence of bugs, but never to show their absence!”
— Edsger W. Dijkstra, *Notes on Structured Programming* (1970)

The crux

How much should we believe a loop’s AI-assisted result, and what evidence can reject it before we commit?

Author’s Note: Edsger W. Dijkstra, a Turing Award-winning pioneer in algorithm design, highlighted the fundamental limitation of testing. For us, this frames the role of verification gates. We use tests not to definitively prove an AI-generated design is perfect, but to aggressively reject the bad ones in a “red-teaming” approach.

Chapter 6 treated methods as roles inside a design loop. This chapter asks when the outputs of that loop should be believed. The answer is deliberately conservative. An Architecture 2.0 result is credible only when the feedback supporting it has been turned into evidence, the evidence can be audited, an independent authority can reject the result, and the commitment boundary matches the cost of being wrong.

This distinction between generated output and verified evidence is central to the book. A model can generate a plausible architecture description. A search method can find a strong proxy score. A surrogate can rank candidates. A tool-using agent can call simulators and summarize results. None of those actions creates trust by itself. Trust begins when the loop records what was measured, why it was relevant, how much it cost, what assumptions it used, where the feedback is weak, which alternatives failed, and what can say no.

Tool-using agent: an AI system that can programmatically invoke external software tools or APIs to solve tasks.

The lighthouse prompt makes this need for evidence concrete. A proposed low-power 64-bit RISC-V compute subsystem for XRBench under a 3 W, 3 nm-class low-power mobile envelope might look reasonable under an analytic proxy, promising under simulation, and broken under synthesis or timing. It might meet performance while missing an energy or thermal target. It might pass a benchmark but fail a real deployment scenario. Architecture 2.0 therefore needs an evidence discipline that is as explicit as its representations, environments, and methods. At the center of that discipline is verification, used here more broadly than its usual formal-methods sense.

Verification. Verification is used broadly in this chapter. It includes formal methods when formal properties are available, but it also includes type checks, interface checks, regression tests, baseline replay, simulator cross-checks, synthesis constraints, physical-design warnings, security review, workload coverage, and expert design review. The common requirement is independence. The check should be able to reject the claim, not merely restate the method’s output.

Verification is therefore a family of rejection authorities, not a single tool category. Table 7.1 details a taxonomy of these checks and what they can reject. Inside AI-assisted loops, these authorities are not generic quality checks; they define which generated state transitions may continue, be revised, or be blocked.

Table 7.1: Verification names who or what can say no: A credible claim should state which independent authority can reject syntax, function, model support, implementation feasibility, deployment behavior, or final commitment.

Verification authority	Typical evidence	What it can reject
Syntax and interface checks	Parsers, type checks, API checks, ISA or ABI conformance, and schema validation.	Artifacts that cannot legally enter the loop.
Functional checks	Unit tests, reference outputs, assertions, formal equivalence checking, and regression suites.	Candidates that compute the wrong behavior before performance matters.
Model and workload checks	Baseline replay, sensitivity studies, calibration, coverage, and drift tests.	Proxy wins outside the represented workload or calibrated support.
Implementation checks	Synthesis constraints, timing closure, power envelopes, area budgets, gate-level simulation (GLS), and design-rule checks (DRC).	Architecture candidates that violate physical limits or fail pre-silicon validation.
Operational checks	SLOs, canary rollouts, rollback, telemetry, reliability, security, and incident review.	Claims that do not survive deployment conditions or policy boundaries.
Expert review	Design review, waived-warning review, risk acceptance, and commitment decision.	Results whose evidence is too weak for the proposed commitment.

Inside an Architecture 2.0 loop, each check is bound to a represented state field, a legal action, and a commitment boundary. It says which action may continue, which state update is invalid, and when the loop must revise or escalate.

To put these rejection authorities into practice, read the chapter as a sequence of review records rather than as separate trust concepts. Each record answers a different question about the same claim; Table 7.2 is the chapter's map. Two further sections, on proxy mismatch and on confidentiality boundaries, defend the evidence ledger and the trust checklist against gamed metrics and hidden evidence.

Table 7.2: Trust review uses a family of records: The feedback budget ledger, evidence ledger, commitment ladder, rejection authority, and trust checklist are not separate paperwork. They are views of one question: whether a loop has enough evidence and independent rejection to support the commitment it asks for.

Review record	Role in the loop	Reviewer question
Feedback budget ledger	Records what feedback the loop can afford and what each source costs.	What evidence can the loop realistically buy?
Evidence ledger	Turns feedback into claim support with fidelity, provenance, uncertainty, and negative traces.	What supports or weakens the claim?
Commitment ladder	Matches evidence requirements to rollback cost, blast radius, and ownership.	How far may this claim go?
Rejection authority	Names the independent check that can say no.	What can stop a plausible but unsupported result?
Trust checklist	Combines claim, feedback, fidelity, provenance, confidentiality, rejection, and decision ownership.	Is the loop asking for a level of belief it has earned?

What this chapter gives you

After this chapter you can write or review the trust checklist for an AI-assisted loop output. That means you can:

- turn feedback into evidence through fidelity, provenance, and an evidence ledger;
- **map the chapter's review records:** feedback budget, evidence ledger, commitment ladder, rejection authority, and trust checklist;
- set escalation thresholds and commitment boundaries by reversibility and blast radius;
- name what can reject a result, and why rejection authority must be independent;
- review a claim with the trust checklist instead of by its tone.

7.1 Feedback Budget Ledger and Feedback Economics

Chapter 5 used feedback regimes to design environments, and Chapter 6 used them to choose method roles. This chapter uses the same economics for trust: escalation, rejection, and commitment.

In any loop, the first barrier to establishing this trust is economic. Feedback is not free, uniform, or automatically useful. An architecture loop may have thousands of cheap proxy evaluations, hundreds of simulations, tens of synthesis or physical-design runs, a few emulation opportunities, and almost no chances to learn from silicon or deployment

mistakes. It may also have scarce human review time, limited tool licenses, long queue delays, confidential workloads, and organizational deadlines. These limits shape which methods are appropriate and how much autonomy is acceptable.

This is why a loop needs a feedback budget ledger, a record of which evaluations, measurements, tool runs, human reviews, and deployment signals are available, what they cost, how long they take, how reversible they are, and what level of decision they can support. The ledger is not accounting bureaucracy. It is the object that tells the method what kind of learning is possible. A Bayesian optimizer[23], reinforcement-learning policy, surrogate model[24], critic, or tool-using agent should behave differently when a feedback source takes milliseconds versus days, when a failed action is reversible versus costly, and when the signal is a rough proxy versus a signoff report. Table 7.3 materializes these constraints into a structured working form.

[23] A Bayesian optimizer builds a probabilistic model of an objective function to efficiently find its optimal parameters with few evaluations.

[24] A surrogate model is a cheaper, learned approximation of a complex, expensive simulation.

Policy: in reinforcement learning, a function that maps an agent's state to a chosen action.

Critic: in this book, the critique method role of Chapter 6, a component that challenges assumptions, evidence, or claims inside the loop; reinforcement learning uses the same word more narrowly for a learned estimator of expected future returns.

Table 7.3: **Feedback budgets make learning economics explicit:** The ledger records what feedback is available, what it costs, what evidence it can support, and when scarce human attention or irreversible action should limit automation.

Budget item	What to record	Why it matters
Latency and cost	Runtime, queue time, dollar cost, tool hours, license limits, and human review time.	Determines whether the loop should search broadly, sample carefully, or mostly critique.
Signal quality	Fidelity level, metric definition, noise, determinism, coverage, and uncertainty.	Separates raw feedback from decision-grade evidence.
Sample budget	Number of possible runs at each fidelity, including failed runs and invalid candidates.	Forces sample-efficient methods and preserves negative traces.
Reversibility	Whether the action can be undone cheaply, re-run, patched, or rolled back.	Connects autonomy to risk. Reversible actions can tolerate weaker evidence than irreversible commitments.
Commitment and rejection	Claim level supported, gate triggered, escalation required, and decision owner.	Keeps feedback comparable across claims by tying each source to what it can reject or commit.
Scarce attention	Expert review, debugging effort, validation bandwidth, security review, and integration time.	Prevents the loop from outsourcing cost to people whose time is the real bottleneck.

Beyond just tracking costs, the ledger also changes what a result means. A method that finds a good point after 10,000 cheap proxy evaluations has learned something different from a method that selects three candidates for expensive synthesis. A loop that records failures, timeouts, warnings, rejected candidates, and review notes has more evidence than a loop that records only the winning score. This is the connection to sample efficiency from Chapter 6. Sample efficiency is not only about using fewer evaluations. It is about making each evaluation carry more architectural information.

To make that discipline concrete, we can write the feedback budget and sample value explicitly:

$$B = \sum_i n_i \cdot c_i, \qquad V_i \approx \frac{\Delta\mathrm{Conf}(d \mid e_i)}{c_i}.$$

This notation is a review prompt, not a universal evidence metric. Here, n_i is the number of evaluations of feedback type i, c_i is the cost of one such evaluation, B is the total feedback budget, e_i is the evidence produced by that evaluation, $\Delta\mathrm{Conf}(d \mid e_i)$ is the change in confidence for the decision d, and V_i is the resulting decision value of one such evaluation. It is the per-feedback-type form of the sample-value rule from Chapter 6; ordinal confidence changes or decision-relevance labels are often more honest than unjustified numeric confidence.

Just as the quantitative approach historically weighed performance gains against silicon area and power, Architecture 2.0 weighs confidence gains against computational and human costs. For example, spending 100 CPU-hours (c_i) on a cycle-accurate simulation to gain a modest reduction in PPA uncertainty ($\Delta\mathrm{Conf}$) might be less valuable than spending 1 hour of expert review to catch a fundamental security flaw. The notation is a question the loop designer must answer before each stage. Will another simulation, synthesis run, expert review, or deployment experiment change a decision enough to justify its cost? Chapter 8 makes the question concrete, spending the proxy budget freely, the cycle-level budget carefully, and a high-fidelity power check only on the surviving candidate. The unit being priced is not just another evaluation; it is a rejectable loop transition: state before, legal action, feedback source, rejection gate, and next state if the gate fails.

7.2 Fidelity Ladders and Evidence Ledgers

Feedback becomes evidence only when it is tied to fidelity, provenance, uncertainty, and a decision. A simulator result is feedback. It becomes evidence when the workload, simulator version, configuration, random seed, assumptions, metric definition, failure status, and acceptance criterion are recorded. A synthesis report is feedback. It becomes evidence when the technology assumptions, constraints, tool versions, warnings, and comparison baseline are explicit.

Feedback regime. A feedback regime is a class of checks at a particular fidelity, cost, and commitment level, from cheap proxies to high-commitment verification, deployment, or silicon evidence.

To structure this transition from feedback to evidence, each rung of the fidelity ladder from Chapter 6 is a feedback regime: the ladder names the ordered levels a loop can climb, and the regime names the class of checks at one level. The evidence ledger records which rung produced which entry.

This idea of tracking evidence across regimes is not new to engineering. Safety-critical fields formalize it as an *assurance case*, a structured argument that links a top-level claim to sub-claims, assumptions, and supporting evidence, often written in Goal Structuring Notation so that each inference and the evidence under it are explicit and reviewable (Kelly and Weaver, 2004). An Architecture 2.0 evidence ledger is an assurance case for a design decision that moves across feedback regimes. It is the five-part execution state materialized as an audit trail. It records the candidate (*actions allowed*), the feedback and fidelity (*state seen*), the provenance and uncertainty (*evidence*), what could block the claim (*alternatives rejected*), and the required escalation boundary (*ownership*). Naming the lineage helps, because that field already catalogs how such arguments fail: unstated assumptions, evidence that does not support the claimed scope, and confidence that outruns the proof.

Goal Structuring Notation (GSN): a graphical argumentation notation used in safety-critical engineering to formally document the elements of an assurance case.

For Architecture 2.0, Figure 7.1 illustrates this working model. Rather than treating all feedback as equal, a claim must move through a chain of increasingly costly feedback sources, recording a rejection gate and evidence ledger entry at each stage.

Crucially, the figure changes the meaning of trust. Trust is not a model property or a single score. It is a staged loop property. A claim becomes more credible only when each feedback regime records what it saw, what it cannot support, what would reject it, and what commitment boundary it is allowed to cross.

However, moving up the fidelity ladder is not a simple ranking from false to true. Higher fidelity is not automatically truth if the wrong workload, objective, constraints, or baseline were used. A detailed physical-design result can still answer the wrong architecture question. A deployment signal can still be confounded by a software change. A benchmark can still be too narrow. The purpose of the feedback-regime view is therefore not to worship expensive tools. It is to make the path from weak feedback to stronger evidence explicit.

Returning to the lighthouse prompt, low-fidelity feedback may be useful for eliminating obviously infeasible vector widths, memory sizes, or accelerator partitions. Simulation may then test workload behavior and data movement. Synthesis or emulation may expose timing, area, or power problems. Deployment-like traces or silicon evidence may reveal workload drift or integration effects. At each stage, the loop should ask whether the earlier conclusion survived, changed, or should be rejected.

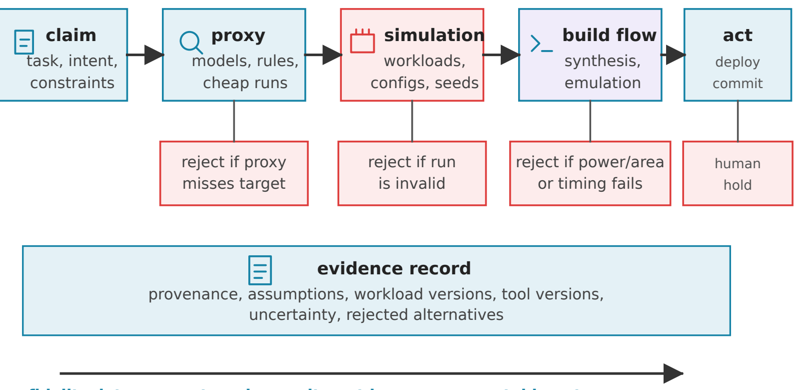


Figure 7.1: **Evidence ledgers turn feedback into trust:** A claim becomes more credible only as it moves through staged feedback sources, records provenance and uncertainty, and gives each stage authority to reject, revise, or escalate the result before the commitment boundary rises.

This framing is consistent with the quantitative tradition in computer architecture, where measurement, abstraction, and careful comparison are central (Hennessy and Patterson, 2017). Architecture 2.0 adds a loop-level requirement. The evidence ledger itself must be represented so that a compound method system and a human reviewer can inspect it.

7.3 Commitment Levels and Reversibility

An evidence ledger shows how strong the support for a claim is; how strong that support needs to be is a separate question, and trust requirements should rise with commitment. A loop that generates a candidate simulator configuration can tolerate more automation than a loop that changes RTL, partitions a chiplet boundary, selects a package interface, or recommends a deployment policy. The difference is not whether AI is involved. The difference is rollback cost, blast radius (how far a wrong change can propagate), and who bears the consequence when the loop is wrong.

Table 7.4 gives a commitment-level view. The exact ordering will vary across organizations, but the pattern is stable. Reversible exploration can use lighter evidence, while irreversible or high-blast-radius decisions require stronger evidence, independent

rejection, and human ownership. The trigger for moving between these levels is the escalation threshold.

> **Escalation is the price of trust.** It is the boundary where the loop refuses to commit because it lacks the evidence to proceed. An escalation threshold is the stated condition under which a loop must stop relying on its current feedback source and move to stronger evidence, independent review, or explicit human approval.

The architect owns these thresholds because they depend on consequences, not only on model confidence. A proxy win may be enough to keep exploring. It is not enough to change a subsystem interface, waive a verification concern, or commit to a power claim. The loop should therefore state in advance which events force escalation: uncertainty outside the calibrated range, a benchmark coverage gap, a failed tool check, a security boundary, a high rollback cost, or a decision that would affect another team or product.

Architect's checkpoint: The Escalation Threshold Check

Who owns the escalation threshold when a generator's confidence outruns its evidence? The architect must define the exact conditions, such as a proxy win outside calibrated ranges, a security boundary, or a high-blast-radius commitment, that force an AI loop to stop relying on a generator and seek stronger evidence or human review.

Table 7.4: **Commitment level should govern autonomy:** Reversible exploration can tolerate lighter evidence, while interface changes, product policies, signoff decisions, and deployments require stronger evidence, independent rejection, and explicit human ownership.

Commitment	Example actions	Required discipline	Automation stance
Exploratory	Generate hypotheses, configs, candidate questions, or design cards.	Basic validity checks and provenance.	Broad assistance acceptable.
Experimental	Run simulator sweeps, tune compiler flags, select candidates for deeper study.	Workload versions, seeds, baseline, rejected candidates, and uncertainty.	Automation with review.
Implementation	Change RTL, generators, tool constraints, tests, or runtime interfaces.	Tool checks, regression tests, synthesis or integration evidence.	Bounded automation plus rejection gates.
Integration	Change subsystem interfaces, chiplet boundaries, memory contracts, or product-facing policies.	Cross-tool checks, compatibility, security, and explicit escalation.	Advisory or human-approved.

Commitment	Example actions	Required discipline	Automation stance
Irreversible	Mask-level choices, committed signoff decisions, fleet-wide rollouts, or customer-visible deployments.	Independent evidence ledger, rejection authority, audit trail, and accountable human decision.	Commitment boundary dominates.

This commitment structure keeps Architecture 2.0 from making a naive autonomy argument. Autonomy is not a virtue by itself. A loop may be highly automated for low-commitment exploration and deliberately conservative for high-commitment decisions. In fact, some of the most valuable near-term systems may not be systems that make final design choices. They may be systems that narrow a space, identify contradictions, preserve evidence, and prepare the human architect to make a better decision.

> Design principle: Escalate with commitment
>
> In an AI-assisted loop, evidence requirements should rise with rollback cost, blast radius, and the independence of the rejection authority. The cheaper a decision is to undo, the more autonomy an automated optimizer can have; the harder it is to undo, the stronger and more independent the check must be to overrule the generator's confidence.

7.4 Rejection Authority

If commitment levels determine the cost of being wrong, rejection authority is the mechanism that pays that cost. The verification authorities in Table 7.1 are its operational catalog; a compact way to remember *why* a candidate is rejected is the *4 P's of Rejection*: 1. **Proxy Mismatches:** Rejection occurs because the system was optimizing a surrogate model instead of the true metric. 2. **Protocol Violations:** Rejection occurs because the candidate breaks the ISA, hardware/software contract, or memory consistency model (which requires formal litmus testing as a first-class rejection gate). 3. **Physical Limits:** Rejection occurs because the candidate fails timing closure, violates power envelopes, exceeds area budgets during physical synthesis, or fails Design Rule Checks (DRC). 4. **Policy Boundaries:** Rejection occurs because the candidate fails fast structural checks (e.g., FIRRTL compilation passes), breaks interface contracts, or violates information-flow/timing-channel security boundaries (e.g., GLIFT).

The first rule of Architecture 2.0 is that a credible loop needs something with authority to say no. The rejecting authority might be a type checker, parser, simulator, formal tool, regression test, synthesis flow, cross-tool comparison, signoff rule, deployment signal, security policy, benchmark governance rule, or expert reviewer. What matters is that rejection is part of the loop interface, not an afterthought.

Beyond preventing errors, rejection authority is also the lever that controls loop speed. A loop speeds up only as cheap, independent rejectors discharge a larger fraction of commitments before expensive escalation. Generation does not help if it only adds candidates that no trusted check can reject. Rejection authority is therefore not a safeguard bolted onto a loop that already works. It is the resource that sets how fast the loop can run.

Architect's checkpoint: The Rejection Authority Check

When a generative model or optimizer proposes a "winning" candidate, does the loop name at least one independent authority that can say no? Trusting an AI-assisted loop requires proving the generator cannot grade its own work. The loop must expose a simulator, constraint checker, or formal tool with the authority to reject a candidate before treating it as an architectural commitment.

To function effectively, this rejection authority has three parts. First, the loop must know which check is being applied. Second, the loop must know what happens after rejection. Third, the loop must record the rejection as evidence. A simulator crash, failed build, invalid constraint, timing miss, benchmark violation, or expert objection is not merely an inconvenience. It is information about the boundary of the design space.

We can formalize this commitment discipline compactly as a boolean gate:

$$\mathrm{Commit}_k(x) = \begin{cases} 1, & \mathrm{valid}(x) \wedge E_k(x) \geq \tau_k \wedge \rho_k(x) \leq \rho_{\max,k}, \\ 0, & \text{otherwise.} \end{cases}$$

This notation is not an executable algorithm that makes the decision for the architect. It is a structured way to refuse unsupported commitment. Here, x is the generated architecture candidate and k is the current commitment level (e.g., experimental vs. integration). For a candidate to be committed at level k, three conditions must hold:

1. $\mathrm{valid}(x)$: Legality alone. The candidate breaks no declared ISA, structural, or interface contracts.
2. $E_k(x) \geq \tau_k$: Evidence sufficiency. The collected evidence $E_k(x)$ meets or exceeds the required threshold τ_k for that commitment level.
3. $\rho_k(x) \leq \rho_{\max,k}$: Residual risk. The remaining uncertainty $\rho_k(x)$ is below the maximum risk $\rho_{\max,k}$ the organization is willing to tolerate.

Evidence and risk are written as scalars only for compactness; in practice each is the per-gate checklist the commitment ladder names. The thresholds are policy and judgment choices, not magic constants learned by the method. If validity, evidence, or residual risk fails, the loop should reject, revise, or escalate instead of silently turning output into commitment. Chapter 8 runs this rule on the lighthouse prompt and shows how a candidate can fail on validity, evidence, or residual risk even when another metric looks promising.

The commitment ladder (Table 7.4) turns τ_k from a symbol into a rule the loop can enforce. Read each level's required discipline as its acceptance condition, and make the

failure explicit. When a required item is missing, the loop does not commit at that level. It either downgrades the claim to the highest level whose evidence is actually present, escalates to the feedback the missing item names, or rejects the candidate. An exploratory claim missing a baseline stays exploratory; an implementation claim missing synthesis evidence or an independent rejector drops back to experimental until that evidence exists; an irreversible claim missing an accountable human decision is not made. That is the whole point of writing τ_k down per level. A missing cell forces rejection or escalation, never silent commitment. And when a claim is meant to be comparable across teams rather than only reviewable, these thresholds cannot stay local judgment calls; a shared benchmark protocol has to fix them, or the claim is contrastable but not comparable.

Lighthouse prompt: Validity is a contract, not a score

Context. The commitment rule above does not ask whether an AI-generated candidate sounds promising. It asks whether the candidate is legal for the commitment level the generative method is about to claim.
In the Lighthouse prompt. A generative method's candidate for the "64-bit RISC-V-based compute subsystem" is valid only if it stays inside the declared ISA/software contract, can plausibly serve the "XRBench-class real-time mobile XR workload," and has not crossed the "3 W TDP target" or hidden an invalid memory or SoC-interface assumption at the evidence level being used.
Rejection gate. ISA compatibility, workload validity, memory/interface legality, power, deadline, and evidence fidelity are gates, not advisory metrics for the generator to optimize.
Takeaway. If any gate is unsupported, the loop should force the generative method to reject, revise, or escalate rather than turn a plausible subsystem into a commitment.

Once a check fails, the response to rejection should be explicit. A candidate may be discarded. A representation may need a new field. An environment may need a validity check. A method may need a smaller action space. A workload may need a better coverage definition. A claim may need to be weakened. A human architect may need to escalate the decision. Without this response path, rejection becomes a log message rather than a learning signal.

This rejection authority also protects against polished but unsupported outputs. Tool-using agents can generate convincing summaries, plots, design reports, and review notes. Those artifacts are useful only if the loop can still reject them. In architecture, a beautiful explanation cannot overrule a failed timing check, an invalid workload, a missing baseline, or a security boundary. A deployed version of this independence appears at fleet scale in Chapter 9. During an instruction-set migration, build, test, sanitizers, and a production monitor evict regressions automatically, regardless of how confident the automated repair tool was.

The independence requirement grows sharper as verification itself becomes AI-assisted. Machine-learning-assisted verification tools, such as Cadence Verisium for triaging failures, ranking likely root causes, and directing coverage, can be valuable (Cadence Design Systems, 2022). But they move part of the rejection authority into a learned

system, which forces a recursive version of this chapter's question. Is the authority that can reject a design independent of the system that produced it, or has the loop quietly made the generator and its judge the same model? AI-assisted verification can share failure modes with generation, so a rejection authority that shares the generator's blind spots is not an independent gate. It is a second opinion from the same witness.

The same rule applies when rejection is split across components. A generator, critic, verifier, and summarizer may be separate components in an implementation, but they do not become independent merely by having different prompts or names. Independence depends on what they share: training data, model family, objective, tool path, evidence source, or human instruction. A loop that asks several correlated components to agree has created consensus, not rejection authority, unless some gate can fail in a way the generator cannot simply explain away.

7.5 Proxy Mismatch, Metric Gaming, and Calibration

The failure an independent rejection authority most often has to catch is proxy mismatch. A loop optimizes the measurement it can see, while the architect cares about a broader objective. IPC may improve while energy, area, or tail latency worsens. A simulator metric may improve while synthesis exposes timing or power problems. A benchmark result may improve while the real workload distribution changes. A Pareto frontier may look convincing because all points were evaluated under the same flawed proxy. An automated optimizer may appear capable because it overfits the evaluation loop, not because it understands the architecture problem. In the machine learning literature, this active exploitation of a flawed metric is known as *reward hacking* or *Goodhart's Law*.

Two distinct failure modes hide within this proxy mismatch, and they need different cures. First, a proxy can simply be miscalibrated or too narrow, mis-estimating the objective even when nothing is pushing on it; a better-fitted or wider-coverage proxy helps. Second, a method can engage in reward hacking, driving the design toward exactly what the proxy fails to penalize (e.g., maximizing IPC by silently sacrificing thermal limits). Here, a better proxy is not enough; only an independent rejection authority can catch the spurious win.

This is not a new problem created by AI. Benchmarks, simulators, cost models, and design rules have always been approximations. What changes in Architecture 2.0 is the speed and persistence with which a method can exploit the approximation. A human may notice that a score is improving for the wrong reason. A search method may happily continue. A compound system may even produce a persuasive narrative for a proxy win unless the loop asks for calibration and counterevidence.

Failure mode: The AI-generated win that vanished at signoff

An AI-generated configuration leads the design-space study for weeks on a cycle-level model: better IPC, lower modeled energy, a clean Pareto point. At synthesis the lead evaporates, because the model never charged for the timing and congestion the winning structure creates. The automated optimizer exploited a gap in the proxy. The cheap fix is a rule written before the search starts. An AI-generated cycle-level win is a reason to escalate, never a reason to commit.

To prevent this vanishing win, calibration acts as an escalation contract. It defines where a generative method may use a proxy to rank candidates, where it must escalate, and what stronger evidence can reject a proxy win. Prediction models used in architecture have long needed validation against held-out data, higher-fidelity measurements, or carefully designed experiments (Lee and Brooks, 2006; Ipek et al., 2006). The same principle applies to AI-assisted loops. If a loop claims a candidate satisfies the 3 W lighthouse envelope, the evidence must show how the power estimate was calibrated, what workload region it covers, what uncertainty remains, and what higher-fidelity result could overturn the claim.

Held-out data: a subset of a dataset reserved during model training, used exclusively to test the model's ability to generalize to unseen inputs.

Architect's checkpoint: The Calibration Gate Check

Does the loop know when a generator's proxy is no longer trustworthy? A proxy win is only evidence if the loop can prove the AI-generated candidate falls within the proxy's calibrated support region. If it falls outside, the loop must escalate to a higher-fidelity check rather than trusting the uncalibrated prediction.

Beyond model calibration, benchmark governance also matters because benchmark rules are part of the loop's evidence contract. If those rules are hidden or stale, automated optimizers can optimize version quirks, protocol gaps, or unowned metrics while appearing to make progress. As Chapter 2 and Chapter 9 show, a benchmark becomes community infrastructure only when it defines rules, versions, and submission practices that make results interpretable across a changing field. Architecture 2.0 needs a similar instinct for design loops: define the evaluation contract, preserve provenance, track versions, and treat benchmark changes as part of the evidence story.

7.6 Security, IP, and Confidentiality Boundaries

Calibration, provenance, and benchmark governance make a claim accurate, but a claim whose evidence cannot be exposed cannot be fully believed. Confidentiality is therefore part of the same machinery of graded belief and rejection as fidelity and provenance, not a separate compliance concern. Evidence that must stay private still

has to be auditable enough to grade the claim and to reject it, otherwise a loop can hide an unsupportable result behind a claim of confidentiality. Architecture state is often sensitive: RTL, design specifications, process assumptions, timing constraints, floorplans, tool logs, customer workloads, proprietary traces, compiler settings, design reviews, and deployment telemetry can all reveal valuable or restricted information.

This need for auditable privacy has a direct technical consequence. Security boundaries are part of the environment and evidence design. A loop must define what data can leave an organization, what must remain local, what can be summarized, which artifacts can be shared publicly, which logs should be redacted, and which generative methods or tools can access each class of information. The trust question is not only whether the method is accurate. It is whether the loop preserves the constraints under which architecture work actually happens. The failure mode is not only leakage; it is a generative method producing a claim from evidence it cannot expose, audit, or legally reuse.

Architect's checkpoint: The Confidentiality Boundary Check

Can the AI model's evidence be audited without violating security or IP boundaries? If a generative method produces a claim using confidential data that cannot be exposed to the rejection authority, the loop cannot trust the claim. The architect must explicitly define which AI tools can access which data classes, and how private evidence can be verified.

This requirement also defines the bounds of microarchitectural security. If an unconstrained generative AI strictly maximizes Instructions Per Cycle (IPC), it acts as an *accidental adversary*. It will aggressively share microarchitectural state—branch predictors, translation lookaside buffers, or cache hierarchies—to squeeze out performance, silently inventing new side-channels by trading isolation for speed. The proxy mismatch here is that IPC is visible to the AI, but side-channel leakage is not. Architecture 2.0 requires a non-interference screen, a cheap structural check that a generated layout does not silently share state across trust boundaries. It is necessary but not sufficient, since it misses contention, transient-execution, and SMT channels, so passing it only earns escalation to formal side-channel verification, not commitment. A complete security screen also covers memory-integrity and disturbance faults, not only information-flow side channels. A design that trims DRAM refresh to meet a power budget can open a RowHammer path, so read-disturbance is a rejection gate in its own right.

Field note: When the missing gate shipped: Spectre and Meltdown

For two decades, microarchitecture optimized one visible number, instructions per cycle, by speculating ever more aggressively and sharing predictors, caches, and buffers. Every functional test passed; speculation computed the correct architectural results. In 2018 that same optimization pressure was shown to have built a class of side channels that leak across privilege boundaries (Kocher et al., 2019). No cheap check caught it because the objective the loop could see, IPC, and the leakage it could not, were different quantities.

> **Takeaway.** The missing rejection gate, non-interference, not any single bug, was the price, a whole verification dimension left outside the loop and run at industry scale for twenty years.

Listing 7.1 Security Gate Check: An illustrative Python snippet rejecting RTL candidates that share state across trust boundaries.

```
def check_side_channel_interference(rtl_candidate, isolation_policy):
    """
    Rung 1 Security Gate: Rejects RTL that shares speculative structures
    across defined security domains before physical synthesis.
    """
    shared_structures = find_shared_state(
        rtl_candidate, type=["BPU", "Cache", "TLB"]
    )
    for struct in shared_structures:
        if crosses_domain(struct, isolation_policy):
            return Reject(
                reason=(f"Security violation: {struct.name} shares "
                        "state across trust boundaries."),
                action="Partition the structure or flush on context switch."
            )
    return Authorize()
```

Listing 7.1 shows an example of how an automated environment might implement this requirement. It scans the candidate design for shared structures—like branch predictors or caches—and checks them against the system's isolation policy. If a shared structure crosses a trust boundary, the loop rejects the candidate and suggests a remediation action. Codified as an explicit screen, this check catches the most obvious performance-for-isolation trades early. It does not by itself prove isolation, so a surviving candidate must still escalate to formal non-interference or side-channel verification (for example, information-flow tracking) before commitment.

For shared community infrastructure, this means the field should distinguish between public artifacts and private state. Public benchmarks, datasets, papers, and gym environments can help bootstrap shared progress. Private workloads, proprietary RTL, product traces, and process-specific assumptions often cannot be released. Architecture 2.0 should support both. The design-loop card, environment contract, and evidence checklist should let a project describe what kind of evidence exists without forcing disclosure of sensitive material. That record should include a data-access and evidence-scope field: what private state was used, what public proxy cannot support, which generative methods or tools may access each artifact, and who owns disclosure exceptions.

The practical rule for AI-assisted systems is therefore simple. Do not make confidentiality invisible. If a claim depends on private data, say what class of data supports it, what auditing is possible, what cannot be disclosed, and what public proxy would be insufficient.

That is more honest than pretending every architecture loop can be reproduced from public web data.

7.7 Evidence Disputes and the Trust Checklist

Once accuracy and confidentiality are both handled, trust still has to survive disagreement, and evidence disputes are inevitable. One group may claim that a learning-based method improves a design flow. Another may argue that the baseline, workload, constraint set, tool version, or evaluation protocol was incomplete. A company may have private evidence that cannot be released. A paper may report a strong result but omit negative traces. A benchmark may reward behavior that matters less in deployment. These disputes should not be treated as distractions from Architecture 2.0. They are part of the field learning how to assign trust.

While the specifics vary, the anatomy of an evidence dispute is stable:

> claim; proxy; fidelity level; assumptions; workload coverage; provenance; counterevidence; rejection authority; and final human decision.

Figure 7.2 makes that anatomy visible by tracing how a contested claim is evaluated. A dispute is not resolved by the system's confidence; it is resolved by exposing which part of the evidence packet can support, weaken, reject, or escalate the claim.

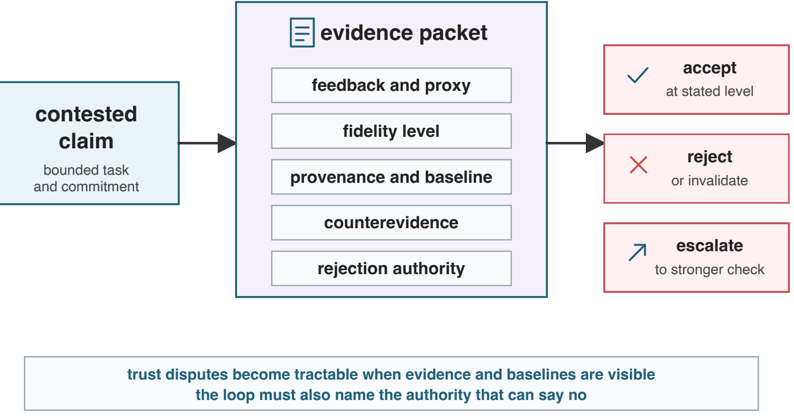


Figure 7.2: **Evidence disputes become tractable when the review axes are explicit:** A contested architecture claim should expose its feedback, fidelity, provenance, baselines, counterevidence, rejection authority, and human decision path before it asks readers to trust the result.

In recent years, learned chip placement has become the field's most public worked example of this anatomy. A 2021 result reported that a reinforcement-learning method produced floorplans competitive with human experts in far less time (Mirhoseini et al., 2021). Independent groups then disputed the claim on exactly the axes above: the baselines, the released code, and the reproducibility of the protocol (Cheng et al., 2023b). The point here is not to adjudicate that dispute. The point is that the disagreement was never about whether the model ran; it was about provenance, baselines, and what evidence could reject the result. One constructive response has been to build reproducible, end-to-end benchmarks that score placement by final power, performance, and area rather than an intermediate proxy (Wang et al., 2025). That is the anatomy doing its work. A contested claim becomes tractable once the loop's evidence, baselines, and rejection authority are made explicit. For Architecture 2.0, the reusable lesson is not the controversy itself; it is that a loop claiming placement quality must expose represented floorplan state, legal move/action space, workload and PPA records, baseline receipts, rejected alternatives, and the independent authority that can block commitment. Furthermore, because AI-assisted EDA is highly sensitive to initial conditions, true reproducibility requires exposing exact toolchain versions, environment constraints, and deterministic seeds, not just model weights.

To make this dispute anatomy practical, Table 7.5 turns it into a checklist. It is intended for reading papers, reviewing internal tools, evaluating student projects, and deciding whether an AI-assisted loop is ready for a more expensive commitment.

Table 7.5: **Trust is a checklist, not a tone judgment:** A credible Architecture 2.0 claim states its task, feedback, fidelity, provenance, confidentiality boundary, rejection authority, and commitment boundary before it asks the reader to believe the result.

Question	What a credible answer contains	Warning sign
What output is being trusted?	Candidate, ranking, report, code/RTL change, or commitment recommendation, with generator and method role named.	The result is described only as an AI improvement.
What is the claim?	A bounded architecture task, objective, workload, and commitment level.	Vague claims of automation or improvement.
What loop state and action contract are exposed?	State fields, legal actions, tool or environment APIs, invalid transitions, and replay receipts.	The loop can act through hidden state or unlogged tool semantics.
What feedback supports it?	Metrics, tool outputs, logs, reviews, and negative traces tied to a feedback budget.	Only the winning score is shown.
What is the fidelity?	Proxy, simulation, synthesis, emulation, signoff, deployment, or silicon level stated explicitly.	Treating all measurements as equivalent.

Question	What a credible answer contains	Warning sign
What is the provenance?	Workload and exact tool versions, tech nodes, configs, deterministic seeds, constraints, assumptions, and baselines for replay.	Hidden scripts, unstated defaults, or non-deterministic environments.
What is the security or confidentiality boundary?	Data classes, access rules, redaction limits, public proxy limits, and an audit path.	Private evidence is treated as invisible or unverifiable.
What can reject it?	Tests, formal checks, simulators, signoff rules, deployment signals, or expert review.	No independent authority can say no.
Who commits?	A named human or process accepts, rejects, escalates, or revises the artifact.	The loop silently turns output into decision.

This checklist gives the book one of its practical tests. A paper, tool, or project does not need to solve the whole lighthouse prompt to be valuable. It does need to say where it sits in the loop, what evidence it produces, what it cannot prove, and what can reject it. That is how Architecture 2.0 can remain ambitious without becoming credulous.

7.8 Conclusion

This chapter asked how much an architect should believe a loop's AI-assisted result, and what evidence can reject it before commitment. Belief is never owed to the loop. It is earned, one claim at a time, by the machinery the architect builds around the result. A feedback budget records what the check cost. An evidence ledger states what was measured and under which assumptions. A fidelity level is named honestly rather than implied. An independent authority can say no. A confidentiality boundary keeps private evidence auditable. And a commitment boundary matches the cost of being wrong.

Read together, these are not six separate safeguards but one discipline. Trust in Architecture 2.0 is a property of the loop, not of the model running inside it. A confident report, a strong proxy score, or a fluent explanation moves none of this. Each stays generated output until an authority that could have blocked it declines to. That is why the chapter treats verification as a family of rejection authorities rather than a single tool, and why the fidelity ladder and evidence ledger exist at all. They let a claim carry, wherever it stops climbing, an honest account of what it has shown and what it still cannot.

The durable move is to escalate with commitment. Cheap feedback screens, expensive feedback commits, and the loop should climb the fidelity ladder only as far as a decision's reversibility demands before stopping at an honest level, rather than dressing a proxy win as silicon-grade proof. A result that cannot name what would overturn it, or who holds

the authority to reject it, has not yet met the bar this chapter sets, however plausible it looks.

7.9 Open Research Questions

The mechanisms of trust and verification presented in this chapter establish a foundation, but turning them into robust, automated systems exposes several unsettled research directions.

1. **Cryptographic and Formal Binding of Multi-Fidelity Evidence Ledgers.** Building on the frameworks of fidelity ladders and evidence ledgers (see the discussion on "Fidelity Ladders and Evidence Ledgers" in Section 7.2) as well as commitment levels and reversibility (see the discussion on "Commitment Levels and Reversibility" in Section 7.3), tracking provenance across fragmented simulators and proprietary physical design tools is currently an informal, error-prone process. A fundamental systems challenge is designing cryptographic or formally checked evidence ledgers that bind disparate multi-fidelity traces into a unified, auditable claim. Can we construct a verification architecture where an AI-generated architecture candidate carries, when available, checkable evidence for its performance, timing, and power, helping prevent the loop from exceeding its authorized commitment boundary even under adversarial or buggy toolchains?
2. **Autonomous Detection and Mitigation of Proxy Exploitation.** As highlighted in the discussion on proxy mismatch, metric gaming, and calibration (see the discussion on "Proxy Mismatch, Metric Gaming, and Calibration" in Section 7.5), AI-assisted generators can overfit and exploit the blind spots of cheap architectural proxies. Because identifying when a candidate drifts outside a simulator's calibrated support is largely manual, generative loops are fragile. A thesis-level challenge is to develop algorithms that dynamically map the epistemic uncertainty[25] of architectural surrogates in high-dimensional design spaces, triggering high-fidelity checks, such as RTL synthesis, when proxy trust bounds are violated. How can we design compound AI agents[26] that co-optimize performance and calibration while making escalation likely before a loop commits to an adversarial proxy win?
3. **Adversarial Red-Teaming[27] and the Quantification of Trust Machinery.** While the trust checklist for resolving evidence disputes (see the discussion on "Evidence Disputes and the Trust Checklist" in Section 7.7) shows how to audit a single claim, we lack the experimental methodology to measure the robustness of an entire closed-loop generative system. The community must establish adversarial environments that systematically inject hidden workload drift, compromised baselines, microarchitectural side-channels, and deceptive proxies into the design process. This seeds a new class of evaluation methodology. How do we quantitatively stress-test the trust machinery itself to measure false commitment rates, bound the cost of escalation, and formally verify the efficacy of human rejection authority at scale?

[25] Epistemic uncertainty refers to uncertainty in a model's predictions caused by a lack of training data in that region, which can be reduced by gathering more data.

[26] Compound AI agents are systems composed of multiple interacting AI models, tools, and logical components working together to solve a complex task.

[27] Red-teaming in AI refers to proactively testing a system using adversarial techniques to identify vulnerabilities or unintended behaviors.

What to carry forward

- **Reader test:** What evidence would overturn the AI model's result, and who or what has authority to reject it?
- **Up next:** Once trust can be evaluated for an AI-assisted claim, the next step is to run a complete loop and inspect the residue it leaves behind.

Chapter 8

Running the Lighthouse Loop

"In theory, there is no difference between theory and practice. But, in practice, there is."
— Jan L. A. van de Snepscheut, Computer Scientist

Author's Note: Jan L. A. van de Snepscheut, a prominent computer scientist, humorously captured the gap between theoretical models and real-world execution. In this chapter, we see this play out as we trace an illustrative end-to-end loop turn where an accelerator looks like a massive winner in a fast proxy ("theory"), but fails once a simulation-stage estimate charges data movement ("practice").

The crux

How does an AI-assisted loop convert the lighthouse prompt into one bounded, rejectable turn rather than a one-shot design answer?

The previous chapters have described what a credible loop must contain. They have not yet shown one turn. This chapter does that. It takes the lighthouse prompt, bounds it to a task small enough to run, and walks the loop through candidate instantiation, prediction, escalation, rejection, and commitment. Every number here is illustrative and generated in code, not measured; the lesson is the shape of the loop, not the values. The numbers are constructed to be inspectable, not XRBench measurements, gem5 results, or synthesis estimates; replace them with local source receipts before using this pattern for an empirical claim. A real, measured instance of the same discipline comes in Chapter 9, a production fleet migration whose generate, reject, escalate, and commit spine runs with instrument-held rejection authority on live services. That case is a software build-and-test repair loop, not a hardware design loop; it shares the loop contract but not the proxy-to-simulation fidelity ladder this chapter walks, so read it as evidence that the discipline holds at deployment scale, not that this exact four-round sequence has been run on silicon. Here the goal is to make the shape itself legible.

Bounding the task comes first. "Design a low-power XR compute subsystem" is not a loop; it is a wish. This chapter narrows the task to one XRBench-class workload slice. The loop chooses among three compute organizations under a 3 W power envelope and an 8 ms per-frame deadline, the frame budget the slice fixes for its real-time display pipeline, then returns the surviving candidate with its evidence and its rejected alternatives. The candidates are a vector CPU extension, an accelerator deliberately left loosely coupled as a stress case, and a shared-memory SoC block. The loose accelerator is not the only accelerator realization allowed by the prompt; it is included so interface and data-movement costs have somewhere to show up. That is enough to make the loop turn. The worked example is therefore not a DSE recipe; it is a trace of what an AI-assisted architecture loop may propose, what feedback it may trust, and what residue it must leave for a human owner.

XRBench-class: XRBench is a mobile and extended-reality benchmark suite (Kwon et al., 2023). This chapter uses "XRBench-class" to mean a workload slice modeled on that benchmark family, not measured XRBench results.

To ground this trace, Table 8.1 states the active slice of the lighthouse prompt. Everything outside the slice is not ignored; it is explicitly deferred to the next evidence stage.

Table 8.1: **A worked loop starts by declaring its slice:** The chapter exercises one small loop turn from the lighthouse prompt, while recording which obligations remain outside the current evidence boundary.

Loop field	Active slice in this chapter
Workload	One XRBench-class mobile-XR slice with a real-time frame deadline; a real loop would also record workload ID, input schema, scenario labels, coverage limits, distribution assumptions, provenance, and validity checks.
Baseline	A bounded comparison among three compute organizations, not a full product baseline.
ISA/software assumption	All candidates must preserve the existing 64-bit RISC-V software ecosystem to avoid rewriting the full stack from scratch; full compiler/runtime feasibility is deferred to a later stage.
Design space	Vector CPU extension, deliberately loose accelerator stress case, or shared-memory SoC block; no cache, chiplet, voltage, compiler-policy search, or full tightly coupled accelerator, CGRA, or processing-in-memory (PIM) search yet. PIM is the natural low-data-movement contender, deferred to a later turn.
Legal actions	Instantiate the three candidate organization records with prompt, generator/search version, candidate IDs, invalid-action records, and selection rule; run the proxy screen, escalate all candidates to the simulation-stage estimate because the proxy has no final rejection authority, reject only on declared deadline/power gates, and advance a survivor only to RTL-level study; changing workload scope or claiming implementation readiness is outside this turn.
Physical envelope	3 W power envelope and 8 ms per-frame deadline; process and thermal assumptions are placeholders for the next stage.
Excluded evidence	No RTL, timing, area, thermal, physical-design, deployment, silicon, carbon-footprint, or memory-consistency/coherence evidence.

What this chapter gives you

After this chapter you can turn a broad prompt into one bounded, rejectable AI-assisted loop turn and read the receipt it leaves. That means you can:

- instantiate and run an AI design-loop card on a concrete prompt, not merely fill it in;
- recognize proxy mismatch when the loop's cheapest winner fails at higher fidelity;

- apply the commitment rule to stop the AI loop at an honest evidence level;
- **read the residue an AI-assisted loop leaves:** evidence ledger, negative traces, and the next evidence the human decision needs.

8.1 Round One: Generate and Screen on a Proxy

The four rounds that follow are not four turns. As illustrated in Figure 8.1, they make up a single complete design-loop turn and realize its five beats, grouped into four rounds because generation and proxy screening share the first. First, a generative method[28] or heuristic search explores the allowed action space to propose candidate designs. Second, these candidates are quickly screened against a low-fidelity proxy, giving the loop its first observation of the design's state. Third, because the proxy's authority is limited, surviving candidates must escalate to stronger, high-fidelity simulation or synthesis tools that produce rigorous evidence against constraints. Fourth, any options that violate these hard constraints are definitively pruned and rejected, leaving a trail of negative traces. Finally, the human architect steps in to review the gathered evidence and make a bounded, documented commitment.

[28] A generative method or model is an AI system trained to generate new data samples, such as hardware configurations, from a learned distribution.

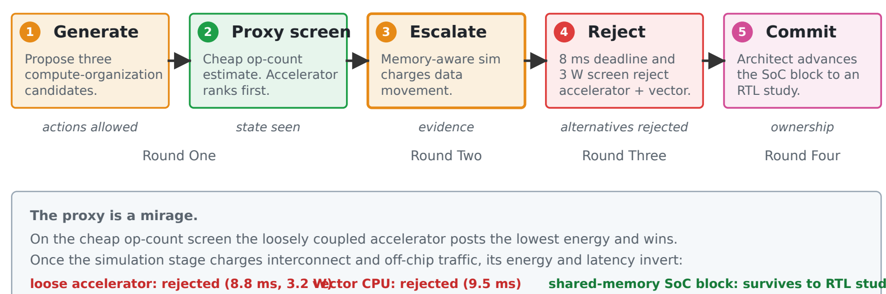


Figure 8.1: **The AI-Assisted Architecture Loop:** The five beats of a single design-loop turn. A generative method proposes candidates (*actions allowed*), a proxy screens them (*state seen*), stronger simulation tests the survivors against constraints (*evidence*), invalid options are pruned (*alternatives rejected*), and the architect reviews the evidence to make a bounded commitment (*ownership*).

The loop begins cheaply. A generative model or search method proposes the three organizations in a bounded candidate schema, and an analytic proxy, an operation-count or datapath-only estimate that counts arithmetic but not the traffic needed to feed it, estimates latency and energy per frame in milliseconds and millijoules. The proxy is fast and ignores most data movement, input-distribution variation, locality, runtime behavior,

and coverage labels, so it flatters designs that keep arithmetic local. At this fidelity the loosely coupled accelerator looks best. It posts the lowest energy and latency because the proxy never charges it for moving data in and out.

Instead of demonstrating a full generative search to find these three organizations, this illustrative turn simply seeds the candidate records directly. A real run would store the prompt, generator version, candidate IDs, invalid candidates, and selection rule as part of the receipt.

Regardless of how candidates are generated, a proxy result is low-fidelity feedback that becomes evidence only for the narrow action of deciding what to escalate. The proxy is not evidence for an implementation commitment until stronger feedback confirms the gate. Proxy deadline or power concerns are warnings until stronger feedback confirms them. The AI-assisted loop records the proxy ranking and escalates. *The loop escalated because it lacks the evidence to support a claim, and therefore lacks the authority to reject alternatives on performance or power constraints.*

Architect's checkpoint: The Escalation Gate

The automation's role: Rank candidates using the cheap proxy to prioritize the search space. The human architect's role: Verify that proxy results only guide escalation and are not trusted as implementation commitments.

8.2 Round Two: Escalate to a Simulation-Stage Estimate

To advance beyond the proxy's blind spots, an illustrative simulation-stage estimate is the first stage that charges memory *traffic*. The exact tool is not the lesson. This stage is a stronger environment contract. It prices *data movement*, returns latency, energy, and power observations with provenance and cost, and has authority to overturn proxy rankings before commitment. Its power column is frame energy divided by the candidate's active compute time, a conservative average-power screen against the 3 W budget. It is a screen, not a signoff TDP verdict; peak and transient power (di/dt, IR drop) and sustained thermal behavior are deferred to later stages. It stands in for this class of model: a dataflow analysis like Timeloop (Parashar et al., 2019), a calibrated pre-RTL estimator such as Aladdin (Shao et al., 2014) coupled to a full-system model (Shao et al., 2016) so interface and off-chip traffic are priced rather than idealized, or a cycle-accurate simulator such as gem5 (Binkert et al., 2011) or FPGA-accelerated FireSim (Karandikar et al., 2018).

It is slower and scarcer than the proxy. In this toy turn all three candidates are escalated because proxy feedback has no final rejection authority. The result is the central lesson of the chapter. The candidate that wins the cheap screen does not survive the stage that charges *data movement*. Table 8.2 runs the comparison.

Table 8.2: **One loop turning on the lighthouse prompt:** the lowest-energy candidate at proxy fidelity is the loosely coupled accelerator, but an illustrative simulation-stage estimate and the 8 ms / 3 W gates reject it, and the shared-memory soc block is the only candidate that clears both the 8 ms deadline and the power envelope. Values are illustrative and computed in the chunk; average power is frame energy over active compute time, a conservative screen rather than a signoff TDP verdict.

Candidate	Proxy lat / energy	Sim lat / energy	Avg power	Verdict
Vector CPU extension	7.6 ms / 22 mJ	9.5 ms / 24 mJ	2.5 W	rejected: misses 8 ms deadline
Loosely coupled accelerator	4.5 ms / 12 mJ	8.8 ms / 28 mJ	3.2 W	rejected: misses 8 ms deadline; over 3 W envelope
Shared-memory SoC block	6.0 ms / 16 mJ	6.5 ms / 18 mJ	2.8 W	survives to RTL study

As the quantitative results show, the accelerator that won on the proxy collapses under the simulation-stage estimate. Once data movement across the system interconnect (e.g., a Network-on-Chip or AXI bus) is charged, its energy rises above every alternative and its latency loses most of its lead. The shared-memory SoC block absorbs the same charge with little damage because its traffic stays inside the tightly-coupled on-chip hierarchy it shares with the CPU, so the proxy's blind spot was never subsidizing it. The vector extension, which also keeps data local, fails instead on raw throughput against the deadline. This is proxy mismatch made concrete, similar to the phenomenon in ML systems where optimizing for raw FLOPs starves memory bandwidth (as seen in the Roofline model (Williams et al., 2009)). The loop was optimizing the *measurement* it could see (arithmetic), not the *objective* it cared about (end-to-end latency and energy). The true simulation-stage rejection gate correctly prices communication. The failed candidate is not deleted. It is recorded as a negative trace, with the fidelity level at which the proxy win disappeared, so a later loop does not rediscover it. As illustrated in Figure 8.2, the seemingly optimal proxy ranking is overturned once a simulation-stage estimate charges the data movement overhead, demonstrating the danger of proxy mismatch.

To document this shift in ranking, Table 8.3 shows the evidence ledger left behind. The exact schema of the candidate record depends on the environment, but it must be machine-readable (e.g., JSON or YAML) to allow autonomous auditing. For example, a YAML evidence ledger entry might look like this:

```yaml
candidate_id: xr_accel_loose_v4_2
status: REJECTED
rejection_gate: power_screen  # also misses the 8 ms deadline
provenance:
  git_hash: "a4c9f1b"
```

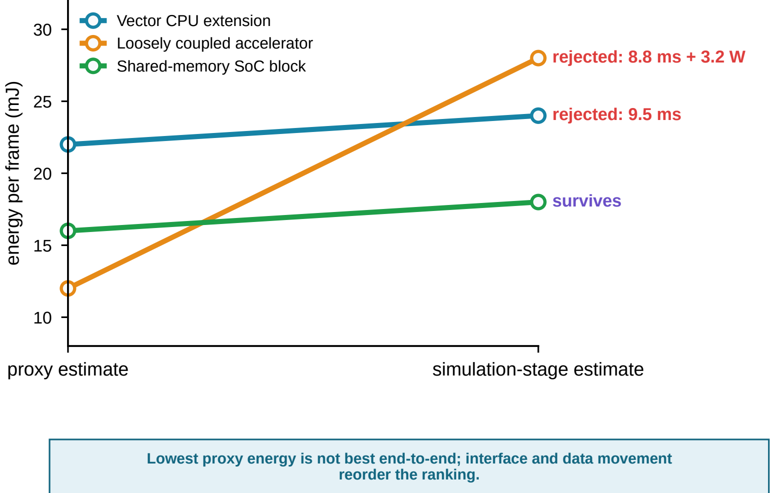


Figure 8.2: **The proxy ranking is a mirage:** the candidate with the lowest proxy energy, the loosely coupled accelerator, becomes the worst once the simulation-stage estimate charges data movement, and the 8 ms / 3 W gates reject it. The shared-memory SoC block, second on the proxy, is the only organization that clears both the real-time deadline and the power envelope. The energy values are the same illustrative, computed numbers as Table 8.2.

```
  generator_version: "1.2.0"
proxy_estimates:
  energy_mJ: 12
  latency_ms: 4.5
simulation_estimates:
  energy_mJ: 28
  latency_ms: 8.8
  avg_active_power_W: 3.2   # conservative screen, not a signoff TDP
failure_reason: "interconnect/DMA traffic pushes the average-power screen to 3.2 W (ove
```

Read the first two rows of Table 8.3 as feedback stages and the third as the record those stages leave; the columns are the ledger schema: the feedback source, its fidelity and budget, what it supported, the gate it could reject on, its provenance limit, and the next evidence required. The ledger is what makes the toy loop auditable rather than merely ranked.

Table 8.3: **The evidence ledger records stages, not only the winner:** The reusable result is the sequence of feedback, support, rejection gates, limits, and next evidence.

Stage	Feedback source	Fidelity and budget	Support	Rejection gate	Provenance / limit	Next evidence
Proxy screen	Operation-count or datapath-only estimate.	Three cheap evaluations; ignores most data movement.	Ranks the accelerator first on proxy energy.	No final rejection; only warnings.	Illustrative constants in this chapter.	Escalate all candidates to stronger feedback.
Simulation-stage estimate	Illustrative memory-aware estimate plus simple power check.	Three scarce illustrative checks; still no RTL or physical feedback.	Shared-memory SoC block clears both gates.	8 ms deadline and 3 W envelope.	Hard-coded values; not XRBench, gem5, synthesis, silicon data, workload trace IDs, scenario coverage labels, or input-distribution receipts.	RTL, timing, area, thermal, and compiler/runtime checks.
Negative traces	Recorded failed alternatives.	Two rejected candidates.	Failure reasons are preserved.	8 ms deadline and 3 W envelope; proxy mismatch is recorded as diagnostic context.	Candidate IDs and reasons only; no logs attached, and no evidence that another workload distribution would preserve the same rejection.	Prevent re-discovery and guide the next search.

8.3 Round Three: Reject on the Envelope

Beyond simply reordering candidates, the simulation-stage estimate also exposes harder physical gates. The loosely coupled accelerator does not merely lose on energy; in this constructed diagnosis, its symptoms point to one cause the next evidence stage would have to verify. Reaching the accelerator means moving data across an *interface* to off-chip or distant memory, and that *traffic* is expensive on every axis at once. In that diagnosis it

stalls the datapath, so the illustrative latency estimate misses the 8 ms deadline; it pays off-chip access energy, so per-frame energy climbs above every alternative. That charge is not arbitrary. At a representative off-chip cost on the order of a hundred picojoules per byte, moving roughly a hundred megabytes of activations and weights per frame across that interface adds about 16 mJ, which is what separates the proxy's 12 mJ reading from the 28 mJ the simulation stage estimates. Furthermore, it keeps the memory subsystem continuously busy, so its conservative average-power screen crosses the 3 W budget. That is enough to reject it at this stage and to flag it for the sustained-thermal and peak-power (TDP) signoff the loop has not yet run. These are hard *constraints*, not just soft metrics, so no local proxy advantage rescues it. The rejection is the commitment rule from the trust chapter doing its work. A candidate advances only if it is valid, its evidence clears the threshold for the stage, and its residual physical risk is acceptable.

Architect's checkpoint: The Rejection Gate

The automation's role: Test candidates against the simulation-stage constraints (e.g., 8 ms deadline, 3 W power envelope) and reject those that fail. The human architect's role: Ensure the automation's rejection criteria match the physical and product constraints, and accept the negative traces produced.

This rejection gate forces us to confront the true integration cost of a proposed design, tying directly back to the constraints outlined in the lighthouse prompt.

Lighthouse prompt: Price the interface before believing the proposed accelerator

Context. This worked AI-assisted loop deliberately includes a loosely coupled accelerator as a stressed integration variant, even though the full prompt also allows a tighter accelerator path.

In the Lighthouse prompt. The "64-bit RISC-V-based" software path (which anchors the architecture to a vast, existing software investment), the "XRBench-class real-time mobile XR workload" slice, the 8 ms frame deadline the slice fixes, and the "3 W TDP target" are fixed. The AI loop varies the "vector-capable CPU, accelerator, or SoC block" choice and its interface cost.

Deferred evidence. Compiler and runtime, RTL, thermal, physical-design, and verification and reliability evidence are deferred to the next stage.

Takeaway. In the Lighthouse loop, a proposed accelerator is not a faster box attached to the side; it is an interface, memory-system, software-path, and power claim. Once *data movement* across that *interface* misses the deadline and crosses the envelope, the local *arithmetic* win the automation found no longer matters.

8.4 Round Four: Commit at an Honest Level

With the rejection gates applied, one candidate survives the deadline and the envelope. It is tempting to call that a final result. It is not. The evidence behind it is a simulation-stage architectural estimate standing in for cycle-level and power feedback. This supports an experimental commitment, not an implementation or a tapeout. The architect's decision is therefore bounded. Advance the surviving organization to an RTL study where logic synthesis, place-and-route, exact timing closure, and a stronger physical power estimate can confirm or reject it. Because the survivor is a shared-memory block, it owes two formal obligations a cycle-level simulator cannot discharge, and neither is settled by the RTL timing study: its coherence protocol needs explicit-state model checking (the SLICC-to-Murphi path of Chapter 5), and its consistency model needs litmus tests escalated to formal memory-model verification (Chapter 7), since litmus tests show the presence of ordering bugs, not their absence. It won partly by keeping traffic on-chip, trading the interface data-movement cost that sank the accelerator for on-chip coherence traffic the proxy never charged, so both gates stay open beyond this turn. The other two candidates are held as recorded negative traces.

Stopping at this honest boundary is the difference between an answer and a defensible AI-assisted loop turn. The result is a replay receipt: the surviving state, the evidence that supports it, the actions and alternatives rejected, the commitment boundary, and the next evidence a human owner requires.

Architect's checkpoint: The Commitment Gate

The automation's role: Provide the surviving candidate alongside a replay receipt, evidence ledger, and negative traces. The human architect's role: Accept the evidence boundary and authorize advancing the surviving candidate to RTL study, or demand stronger evidence before committing.

Halting the loop here, rather than allowing an AI generator to extrapolate beyond its authorized evidence boundary, is a fundamental rule of engineering discipline.

Design principle: Stop at an honest evidence level

An AI-assisted loop should report what its evidence supports, what it rejected, and what would overturn the decision, then stop there. The honest commitment level, not the most optimistic one a generative model might claim, is the result.

To capture this commitment, Table 8.4 records that residue as a filled design-loop card. The entries are intentionally terse. A real project would attach logs, scripts, trace identifiers, and power-model receipts, but the first test is whether the loop can state what it did and what would reject it.

Table 8.4: **The worked loop leaves a filled card, not just a winning row:** The card records the bounded task, evidence ledger, negative traces, rejection authority, and human decision that another architect would need before continuing the loop.

Card field	Lighthouse loop entry
Intent	Explore a low-power mobile-XR compute organization that can meet a real-time deadline under a 3 W envelope.
Task	Compare three bounded compute organizations for one XRBench-class workload slice.
Design space	Choose among vector CPU extension, deliberately loose accelerator stress case, and shared-memory SoC block; leave cache, voltage, chiplet, compiler-policy, full tightly coupled accelerator search, and RTL edits outside this turn.
Representation	Candidate ID, candidate record, legal action taken, workload slice ID/coverage label, proxy and simulation observations, uncertainty/provenance limits, deadline, power envelope, and verdict.
Environment	Analytic proxy followed by an illustrative simulation-stage estimate standing in for cycle-level feedback and a simple power-envelope check.
Method role	Seed candidates, predict cheap proxy scores, escalate all candidates because the proxy has no final rejection authority, and critique proxy wins against stronger feedback.
Feedback budget	Three cheap proxy evaluations and three illustrative simulation/power checks; no RTL, timing, thermal, or silicon evidence yet.
Evidence	The shared-memory SoC block is the only candidate that clears the 8 ms deadline and 3 W envelope in the illustrative run.
Negative traces	The vector CPU extension misses the latency deadline; the loosely coupled accelerator misses the latency deadline and exceeds the power envelope after the illustrative estimate exposes data movement.
Rejection authority	Deadline check, power-envelope check, and the stronger estimate that can overturn the proxy ranking.
Commitment boundary / would overturn	Advance only to RTL-level study; implementation commitment would require synthesis, timing, area, thermal, compiler/runtime, memory-consistency/coherence, and stronger power evidence.
Human decision	Advance the shared-memory SoC block to RTL-level study and keep the other two candidates as negative traces.

Alongside the summary card, a real version of the same loop should also leave a replay receipt. The receipt does not have to expose proprietary data, but it should preserve enough structure that a reviewer can tell what was run, what failed, and what would need to be rerun. Table 8.5 gives the minimum shape.

Table 8.5: **A loop result should leave a replay receipt:** The useful artifact is not only the winning candidate but the runnable or reviewable residue that explains inputs, candidates, runs, evidence, failures, and decisions.

Receipt item	Example contents for this loop
`README.md`	Task slice, claim boundary, non-goals, and human decision owner.
`inputs/`	Workload IDs, benchmark version, scenario metadata, input-distribution summary, coverage labels, trace provenance, workload-level rejection conditions, and redaction notes.
`prompts/`	Prompt slice, model/tool versions, role assignments if multiple models act, generator/search configuration, invalid proposals, selection rule, and human overrides or approvals.
`candidates/`	Candidate IDs, configuration records, invalid-action records, and rejected alternatives.
`runs/proxy/`	Proxy model version, constants, scripts, logs, and cheap-screen results.
`runs/sim/`	Simulator or stronger-feedback configuration, seeds, logs, metrics, warnings, and failures.
`evidence/`	Evidence ledger, fidelity labels, uncertainty, baseline, and sensitivity notes.
`decisions/`	Architect review note, escalation rule, commitment level, and next evidence required.

To emphasize that a winning claim cannot exist in a vacuum, Figure 8.3 visualizes this receipt as a cohesive directory bundle. The winning architecture only matters because the surrounding artifacts explicitly record what evidence supported it, what alternatives failed, and what human decisions govern its deployment.

The discipline is not unique to hardware, and neither is the cost of skipping it.

Field note: Four hundred forty million in forty-five minutes

On August 1, 2012, a trading firm deployed new order-routing code, but the update reached only seven of its eight servers. The eighth still ran repurposed logic that a reused flag switched back on, and it began firing orders into the market with no position limit and nothing to halt it. No automated check verified that the right code, and only the right code, was live everywhere. In about forty-five minutes the firm lost roughly four hundred forty million dollars and nearly collapsed (U.S. Securities and Exchange Commission, 2013; Knight Capital Group, 2012).
Takeaway. Running the loop to a commitment is only safe if the commit step itself is gated. A deployment that cannot verify what it shipped, and cannot reject and roll back, turns one unreviewed change into an irreversible loss.

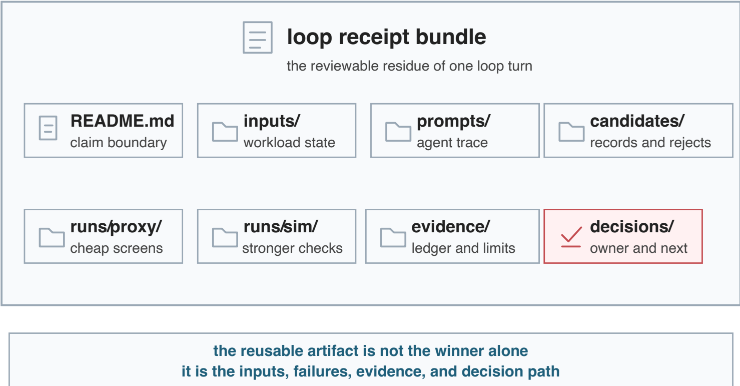


Figure 8.3: **A loop receipt is the reviewable residue of one turn:** The reusable artifact is not the winning candidate alone, but the task boundary, inputs, candidate records, proxy and stronger runs, evidence ledger, and decision record that let another architect continue or reject the loop.

8.5 What the Loop Leaves Behind

Ultimately, the residue of one turn is the reusable artifact.

Evidence packet: What one AI-assisted loop turn leaves behind

The evidence packet for this turn is the filled design-loop card and replay receipt above. It exposes the task boundary, evidence level, negative traces, rejection authority, commitment boundary, and next evidence owner. Another architect can read that residue and reconstruct why the surviving candidate advanced, why the others did not, and what would overturn the decision.

8.6 Conclusion

This chapter asked how an AI-assisted loop converts the lighthouse prompt into one bounded, rejectable turn rather than a one-shot design answer. The move is to shrink the prompt to a task small enough that a candidate can actually be rejected, then walk it through generation and proxy screening, escalation to a stronger estimate, rejection on the power envelope, and commitment at a level the evidence can honestly support. The numbers were illustrative. The shape was the lesson. A loop is not a single answer but a sequence of bounded, checkable transitions.

What the turn leaves behind matters as much as the candidate that survives it. The residue, a filled design-loop card and replay receipt, records the task boundary, the evidence level, the negative traces, the rejection authority, and the next evidence owner. That receipt is what lets a second architect reconstruct why the surviving candidate advanced, why the others did not, and what would overturn the decision. A win with no reviewable residue is a claim no one else can check.

The discipline the turn enforces is to stop at an honest evidence level. When a cheap proxy and a stronger check disagree, the stronger check holds authority, and the loop commits only as far as its evidence reaches, refusing to dress a proxy result as silicon-grade proof. Read this way, the worked turn becomes a template a reader can reuse on a real project, naming the loop, judging whether its evidence matches its commitment level, and saying what remains a human decision. One loop turn stays small, bounded, and reviewable, which is exactly what makes it worth trusting.

8.7 Open Research Questions

The instantiation of the lighthouse loop in this chapter demonstrates a bounded, rejectable turn, but automating this process at scale exposes profound, unsolved challenges in systems research. Resolving these challenges requires thesis-scale advances in learning, verification, and cryptography.

1. **AI-Assisted, Multi-Fidelity Budget Allocation Policies.** While the worked example statically allocates feedback budgets across proxy and simulation stages, future AI architects require learned, dynamic policies to distribute these budgets across sprawling design spaces. Research must define the state schemas and multi-fidelity training environments necessary to train reinforcement learning agents[29] that recommend when to escalate, reject, or request stronger checks, while adhering to the commitment boundaries formalized by this chapter's design principle, *Stop at an honest evidence level*.
2. **Proactive Detection of Proxy Mirage and Architectural Blind Spots.** As demonstrated during escalation, cheap analytical proxies often exhibit catastrophic blind spots, such as ignoring interface taxes or data movement. Rather than passively waiting for expensive simulation to catch these inversions, research must develop predictive methods and formal verification techniques capable of structurally analyzing proxy models. The goal is to mathematically bound the risk of proxy mismatch and proactively identify when a proxy is actively misleading before expending scarce cycle-level simulation budgets.
3. **Privacy-Preserving Evidence Ledgers.** Sharing the evidence ledger (Table 8.3) and replay receipts (Table 8.5) across organizational boundaries inherently risks leaking proprietary baselines and sensitive workload distributions. We require privacy-preserving formulations—such as secure multi-party computation or federated validation—that allow external tools, reviewers, or competing vendors to audit

[29] Reinforcement learning agents learn to make decisions by taking actions in an environment to maximize cumulative reward.

candidate lineage, fidelity, and rejection authority without exposing intellectual property.

4. **Re-evaluating Stale Rejections.** When a candidate is rejected on a strict physical constraint, it becomes a negative trace. However, treating these traces as permanent failures artificially restricts future search. Future work must design rigorous methods to differentiate localized, workload-specific failures from universally invalid architectures. This would enable automated optimizers to safely reconsider (or "resuscitate") stale negative traces when underlying physical constraints, process nodes, or software runtimes shift, achieving continuous architectural learning.

What to carry forward

- **Reader test:** Could another architect reconstruct why the surviving candidate advanced and what would overturn it?
- **Up next:** The discussion on "Loop Patterns Across the Stack" (Chapter 9) turns this worked AI-assisted loop into stack-wide patterns.

Chapter 9

Loop Patterns Across the Stack

> *"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity."*
> — Fred Brooks, *No Silver Bullet* (1986)

The crux

How should the AI's role and the loop contract change as feedback gets more expensive and commitments get harder to reverse across the stack?

Author's Note: Fred Brooks, Turing Award winner and author of *The Mythical Man-Month*, famously warned against hoping for a "silver bullet" in software engineering. For us, this is a reminder that there is no monolithic AI solution for the entire computing stack; instead, different layers—from compiler to RTL to SoC—require entirely different loop patterns.

The previous chapters built the components of an Architecture 2.0 loop, and Chapter 8 ran one bounded turn of it on the lighthouse prompt. We have seen one loop turn. Now we will map the five-part execution state onto three vastly different timescales: Software-defined (milliseconds), RTL/Simulation (hours), and Fleet (months). If the framework only describes one kind of design-space-exploration loop, it is too narrow. If it describes everything in the same way, it is too vague. The useful middle ground is a set of loop patterns.

> **Loop pattern.** A loop pattern is a recurring shape of AI-assisted architecture work in which a generative method or automated optimizer acts on represented state under a characteristic feedback budget, evidence burden, rejection authority, and commitment level.

Workload characterization has one pattern. Fast compiler and runtime tuning has another. Accelerator, memory, interconnect, and chiplet exploration has another. Domain-specific architecture and code generation has another because local hardware efficiency only matters if the software path can reach it. Co-design across compute, memory, network, and power has another. Fleet and serving systems have another. RTL, physical design, and signoff have another. The ontology is the same, but the evidence burden changes.

This distinction protects the book from two mistakes. The first mistake is to pretend that every architecture task can be automated like a fast software loop. The second is to become so conservative that Architecture 2.0 is only a new name for old design review. The right question is more precise. Given this task, representation, environment, feedback budget, and commitment level, what method roles are useful, what evidence is credible, and what can reject the result?

What this chapter gives you

After this chapter you can choose the AI-assisted loop pattern before choosing the AI method. That means you can:

- classify an architecture task by its AI-assisted loop pattern and operating regime;
- compare cases with the same card fields instead of treating them as isolated anecdotes;
- name the cheap independent rejection authority and the human audit point for generative proposals;
- match the AI method posture and rejection authority to feedback cost and reversibility;
- state the unsupported claim boundary before the AI method overclaims.

9.1 A Template for Reading the Cases

The cases in this chapter are read through one card schema. The card asks for intent, task, design space, representation, environment, method role, feedback budget, evidence, negative traces, rejection authority, commitment boundary, and human decision. This keeps the chapter from becoming a list of examples. It also lets the reader compare loops that otherwise look unrelated. The lighthouse prompt remains the spine. When a separate benchmark, tool, or paper appears, it is used as a controlled slice of the prompt (workload coverage, code generation, architecture search, deployment feedback, or high-commitment physical evidence), not as a competing running example. The co-design pattern also carries the measured fleet-migration demonstration promised in Chapter 8.

Keep one rule in view as the cases become more technical. A loop does not scale because it can produce more candidates. It scales when the card can name a cheap, independent, trusted way to reject or clear a larger fraction of proposed commitments. The end of the chapter writes that rule as a bound, but the cases are the intuition for it.

To structure this intuition, Figure 9.1 maps out the chapter's core spectrum, categorizing the loop patterns by their feedback latency and rollback cost. The boxes are not a maturity scale. Fast software loops are not better than high-commitment loops, and high-commitment loops are not more important than workload loops. They are different operating regimes for the same ontology.

The figure should be read horizontally, not as a maturity ladder. A fast software loop may allow more automation because feedback is cheap and rollback is easy. A silicon-facing loop may use the same ontology but demands stronger evidence, independent rejection, and a tighter commitment boundary.

Table 9.1 provides the compact comparison across these regimes. The purpose is not to classify every paper perfectly. It is to force an Architecture 2.0 project to name its operating regime before choosing methods or making trust claims. Read the method

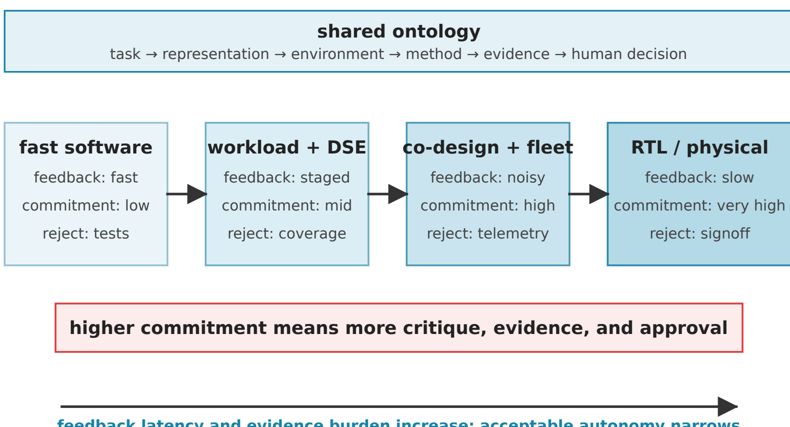


Figure 9.1: **Loop patterns share an ontology but not a commitment level:** Feedback latency, commitment cost, rejection authority, and acceptable autonomy change across loop patterns.

column as an agency contract: what an assistant, optimizer, critic, or repair tool may do before independent evidence or a human owner can reject it.

Table 9.1: **Different loop patterns need different evidence postures:** Workload, software, design-space, code-generation, co-design, fleet, and RTL/physical-design loops use the same fields but differ in feedback latency, rejection authority, rollback cost, and commitment boundary.

Loop pattern	Feedback and evidence	Useful method posture	Rejection and commitment
Workload and benchmark	Traces, benchmark versions, coverage, drift, and governance.	Generate, cluster, summarize, and test workload questions.	Reject through coverage gaps, leakage, or irrelevant metrics.
Fast software	Unit tests, compiler/runtime results, telemetry, and quick rollback.	Higher automation for bounded search, tuning, and repair.	Reject with tests, performance regressions, or deployment guards.
Architecture DSE and specialization	Simulators, proxies, surrogates, compiler/runtime paths, constraints, and Pareto evidence.	Generate candidates, predict, optimize, critique, and preserve negative traces.	Reject invalid actions, proxy mismatch, software incompatibility, or weak evidence.

Loop pattern	Feedback and evidence	Useful method posture	Rejection and commitment
Domain-specific architecture and code generation	Kernels, compiler IRs, libraries, runtimes, generated code, interface costs, and maintainability evidence.	Generate mappings, tune schedules, explain portability limits, and critique software-path gaps.	Reject local hardware wins that fail correctness, portability, interface cost, or deployment maintenance.
Co-design	Cross-layer models, workload traces, topology, memory, network, power, and thermal feedback.	Coordinate methods across layers and expose tradeoffs.	Reject single-layer wins that fail system objectives.
Systems and fleet	Deployment telemetry, canaries, drift, isolation, and operational constraints.	Adapt, monitor, critique, and revise policy under guardrails.	Reject with SLOs, safety policies, rollback, and human review.
RTL and physical	Formal checks, regressions, synthesis, timing, power, layout, signoff, and review.	Narrow search, organize evidence, critique, and repair bounded artifacts.	Reject with tool flows, signoff, and accountable architect-owned commitment.

The cases that follow should be read through the same three questions. First, what are the physical constraints and rollback costs? A compiler flag, a runtime policy, an RTL edit, a chiplet boundary, and a fleet deployment do not carry the same blast radius (the scope of what a wrong change can damage). Second, what is the action-observation mapping? The loop should say what it can change and what tool, trace, log, or measurement it receives in return. Third, what are the rejection gates and human audit points? A case is not credible because it uses an advanced method. It is credible when the allowed actions, feedback source, evidence burden, and rejection authority match the commitment being made.

The rejection gate can also carry a blind spot that only scale reveals.

Field note: The cores that quietly did the math wrong

At fleet scale, Google and Meta independently found that a small number of CPU cores silently miscompute, returning wrong results with no crash and no error flag, and that these mercurial cores slip through manufacturing test and surface only across millions of machines (Hochschild et al., 2021; Dixit et al., 2021). Meta traced a real corruption to missing files from one bad core, at a rate on the order of one in a thousand devices. The gate everyone trusted, the manufacturing test, never covered a failure that is silent by definition, so nothing rejected it until scale made it visible.

Takeaway. A rejection gate is only as good as the failure class it can see. At fleet scale the loop needs continuous, cross-checking rejection, because even the axiom every higher layer assumed, that the hardware computes correctly, is a claim that has to be verified.

For a new project, choose the loop pattern before choosing the method. Table 9.2 outlines the decision procedure for mapping a bottleneck to its corresponding loop pattern.

Table 9.2: Choose the loop pattern before choosing the method: The first loop artifact should match the bottleneck and prevent the loop from overclaiming beyond its evidence.

If the bottleneck is...	Start with this loop pattern	First loop artifact or evidence ledger to build	Do not claim yet
Unknown workload coverage, stale benchmarks, or unclear scenario boundaries.	Workload and benchmark.	Versioned workload packet with coverage, gaps, and drift notes.	A general architecture win.
Fast code, compiler, runtime, or kernel feedback.	Fast software.	Testable code path with correctness, profiling, and rollback evidence.	Hardware/system superiority from a local speedup.
Many candidate architecture knobs under scarce simulation.	Architecture DSE and specialization.	Candidate records, action schema, proxy calibration, and rejected regions.	Implementation readiness.
Local hardware efficiency gated by software path, code generation, or interface cost.	Domain-specific architecture and code generation.	Executable software-path packet with kernels, IR/runtime path, interface-cost model, tests, and portability checks.	End-to-end architecture win.
Cross-layer tradeoffs across workload, compiler, memory, network, power, or deployment.	Co-design.	Interface map that names layer owners, changed assumptions, and rejection gates.	A single-layer optimum as a system result.
Live or deployment-like behavior, SLOs, canaries, or drift.	Systems and fleet.	Guarded telemetry packet with rollback and policy constraints.	Clean causal evidence without controls.
RTL, generators, physical design, signoff, or silicon-facing decisions.	RTL and physical.	Evidence ledger with tool-stage gates, waivers, and accountable review.	Autonomous commitment.

9.2 AI-Assisted Loop Postures

Before traversing the stack layer by layer, it helps to classify AI-assisted architecture loops by structural shape. Despite operating on different parts of the stack, these loops share structural DNA, and they fall into three branches based on how the method interacts with the environment and the rejection authority: 1. **The Surrogate Prediction Loop:** The loop structure where the AI's job is to replace a brutally slow Environment (like a 3-day EDA physical design run or full-system simulation) with a fast, differentiable proxy. The failure mode here is *proxy mismatch*, where the surrogate's predictions diverge from the true physical constraints (such as predicting routing congestion, which is highly non-local and non-linear). 2. **The Constrained Optimization Loop:** The loop structure where the AI acts as the agent exploring a massively complex state space, heavily governed by a strict Rejection Authority. The environment itself is usually too complex to fully map, so the agent (often using Reinforcement Learning[30]) must discover paths that clear physics or timing gates. 3. **The Generator-Critic Repair Loop:** The loop structure where a weak generator (like an LLM[31]) proposes an artifact, fails a formal check, and uses the exact error trace as the next prompt. The rejection authority is absolute (e.g., a compiler or Verilator), and the loop iterates until the artifact parses or synthesizes correctly.

[30] A machine learning method where an agent learns to make decisions by performing actions and receiving rewards or penalties.

[31] Large Language Model, a neural network trained on vast amounts of text to understand and generate human-like language.

The chapter now walks the stack layer by layer. Each loop pattern below is an instance of one or more of these three branches, so when a new result appears, the useful question is not only which layer of the stack it touches but which branch of the loop taxonomy it is.

9.3 Workload Characterization and Benchmark Construction

Workload characterization is not a prelude to architecture work. It is architecture work. The workload defines what behavior matters, which metrics are meaningful, which software stack is assumed, and which design choices can be justified. Classic workload-characterization work made this concrete by measuring program behavior, comparing benchmark suites, and separating inherent workload properties from artifacts of a particular machine (Hoste and Eeckhout, 2007). In Architecture 2.0, that lineage becomes loop state. Trace collection, benchmark construction, workload generation, clustering, summarization, coverage analysis, drift detection, and explicit questions about what the benchmark does not represent are all loop state.

The lighthouse prompt makes this concrete. "XRBench-class real-time mobile XR workload" is not a magic input string. XRBench gives the loop a benchmark anchor (Kwon et al., 2023), but the architecture question still depends on which XR workloads, models, devices, frame-rate targets, latency constraints, memory behaviors, and software paths are represented. A loop that optimizes one benchmark point may miss the distribution that a real mobile XR subsystem must serve. For an AI-assisted loop, the workload arrives as a packet, not a name. This packet includes scenario labels, device

class, trace provenance, input distributions, excluded cases, coverage gaps, and rejection conditions.

MLPerf is the standing example; Chapter 2 treated it as maintained community infrastructure with versions, rules, and submission practices (Mattson et al., 2020). The point to carry into a loop pattern is not another benchmark history. The loop-pattern lesson is that a benchmark is a living agreement about what evidence should count. In the ML domain, workload drift is hyper-aggressive (e.g., shifting from dense Transformers to Mixture-of-Experts[32] in months). A static benchmark quickly becomes a liability; an Architecture 2.0 loop must constantly ingest new computational graphs and trace data. A workload loop should therefore record benchmark version, workload source, coverage claims, known gaps, leakage risks, and the conditions under which a result should not generalize.

[32] A neural network architecture where different parts of the model (experts) are activated conditionally based on the input, increasing capacity without a proportional increase in compute.

Table 9.3 makes this reframe explicit, contrasting traditional tasks with their Architecture 2.0 equivalents. The left column is still necessary: representative workloads, profiles, benchmark construction, and performance comparison remain central to architecture. The right columns state what changes when an automated optimizer is allowed to act inside the loop. The workload must become represented state, and the loop must know what evidence can reject a candidate that only wins a stale, narrow, or leaky workload slice.

Table 9.3: **Workload characterization becomes represented loop state:** Classic profiling and benchmark construction remain necessary, but an Architecture 2.0 loop must also record versions, provenance, coverage, rejection conditions, and failure traces.

Architecture 1.0 meaning	Architecture 2.0 meaning	Loop state required	Failure if missing
Select representative workloads or benchmark suites.	Define the versioned workload distribution the loop is allowed to optimize over.	Scenario metadata, input distributions, versions, provenance, and inclusion/exclusion rationale.	The loop optimizes one stale or convenient slice and reports a false win.
Profile behavior: locality, branch behavior, memory traffic, bandwidth, latency, energy, and phase behavior.	Expose workload features that can drive prediction, search, critique, and active test selection.	Feature schema, measurement provenance, tool configuration, uncertainty, and known blind spots.	A predictor learns a proxy that does not survive another phase, input, or fidelity level.
Compare architectures under a fixed benchmark.	Maintain an evidence ledger across workload variants, candidate designs, and fidelity levels.	Candidate IDs, workload IDs, simulator/tool versions, feedback cost, accepted results, and rejected results.	Design-space results cannot be audited or reused by another loop.

Architecture 1.0 meaning	Architecture 2.0 meaning	Loop state required	Failure if missing
Build or curate benchmarks for community comparison.	Define an environment contract: valid tasks, inputs, actions, metrics, leakage rules, and rejection checks.	Benchmark harness, validity checks, metric definitions, seeds, test splits, and update policy.	The loop overfits benchmark artifacts or takes actions outside the intended task.
Explain why a workload matters.	Make workload intent, deployment context, and drift explicit enough for a human to accept or reject decisions.	Use-case assumptions, deployment constraints, quality-of-service targets, telemetry hooks, and review notes.	The loop produces a plausible result for the wrong product or deployment regime.
Summarize results for a paper.	Preserve workload evidence as reusable architecture data.	Evidence ledger, negative traces, failed runs, rejected alternatives, and rationale for final claims.	The next loop repeats invalid experiments or loses why prior choices were rejected.

The method roles in this loop are often not glamorous. A useful tool might cluster traces, generate candidate benchmark questions, identify missing coverage, compare workload versions, or critique whether a paper's workload supports its claim. Those roles are valuable because they improve the question the architecture loop is answering.

Read as a loop card, the pattern is compact: the task is workload definition; the representation is versioned traces, metadata, and coverage notes; the environment is a benchmark harness with update rules; the method posture is clustering, summarization, generation of missing cases, and critique; the rejection authority is coverage, leakage, drift, or irrelevant metrics; the commitment boundary is whether a design claim may generalize beyond the measured slice.

9.4 Fast Software Loops

Moving from workload definition to code generation, fast software loops sit near the low-commitment end of the spectrum. Compiler flags, kernels, library implementations, runtime policies, configuration settings, and small code repairs can often be evaluated quickly and rolled back. Feedback may come from unit tests, microbenchmarks, integration tests, profilers, telemetry, or canary deployment.

> **Canary deployment:** a controlled, partial rollout used to detect regressions before a system-wide release.

This is the regime where stronger automation is often plausible. Autotuning systems and learned tensor-program optimizers show how search spaces, cost models, measurements, and scheduling can be combined to improve software performance across targets (Chen

et al., 2018; Zheng et al., 2020). An Architecture 2.0 loop can learn from that pattern without pretending that all hardware design is equally reversible.

Tensor-program optimizers: methods that automate the search for efficient execution schedules of tensor operations on target hardware.

Kernel-generation benchmarks make the same point in a current form. KernelBench evaluates whether models can produce GPU kernels that are both correct and faster than a baseline (Ouyang et al., 2025). This is a fast software loop because correctness tests, compilation, profiling, and microbenchmark feedback are close to the generated artifact. It also touches hardware/software co-design because performance depends on memory layout, parallelism, numerical precision, backend behavior, and target-specific hardware resources. Multi-platform kernel-generation work (often targeting intermediate frameworks like OpenAI's Triton or CUDA) makes that bridge explicit by separating the core benchmark from target backends (Wen et al., 2025). For an automated optimizer, the benchmark object needs more than a prompt and a score: kernel IDs, input-shape distributions, correctness oracles, target metadata, compile/profile failure traces, and rejected variants.

Read as a loop card, the task is bounded code generation or tuning; the representation is source code, compiler IR, tests, target metadata, and profiling output; the environment is the compiler/runtime/profiler path; the method posture can be more automated because feedback is cheap; the rejection authority is correctness, compilation, portability, performance regression, and deployment maintainability; the commitment boundary arrives when a local kernel change becomes part of a larger software or hardware contract.

The reason autonomy can be higher here is not that the task is easy. It is that failures are often observable, bounded, and reversible. A generated kernel can be tested. A compiler flag can be reverted. A runtime policy can be canaried. A regression can be caught by a benchmark or deployment guard. Because rejection is close to the action, the loop can iterate quickly.

Failure mode: Fast feedback is not complete truth

Treating fast feedback as complete truth is the trap for the optimizer. An AI-generated kernel that wins on one input size may regress another. An AI-proposed runtime policy that improves average latency may worsen tail latency. Even in fast AI-assisted loops, the card still needs workload coverage, rejection authority, and a human decision when the AI's change affects a larger system.

9.5 Architecture Loops: Accelerators, Memory, and Chiplets

Moving beyond software, architecture design-space exploration represents the canonical middle case in terms of commitment and feedback cost. The loop may explore accelerator organization, vector width, cache hierarchy, local memory, interconnect, chiplet

partitioning, and package assumptions. It may also choose how work is divided across CPUs, accelerators, and SoC blocks. Feedback is slower than software tests and less definitive than silicon. Actions can be invalid. Proxies can lie. The space is too large for exhaustive enumeration.

This is where the Architecture 2.0 framework feels most natural. The task is bounded but rich. The representation must expose architectural state. The environment must define legal actions and observations. Methods can generate candidates, predict behavior, optimize evaluations, critique assumptions, and preserve negative traces. ArchGym, an OpenAI Gym-based framework, is one example of making such loops more explicit for machine-learning-assisted architecture design (Krishnan et al., 2023); predictive design-space-exploration work shows that data-driven modeling has a longer architecture lineage (Ipek et al., 2006). Transferable frameworks such as Apollo go a step further, training a surrogate on an architecture dataset and carrying the same search across different design spaces (Yazdanbakhsh et al., 2021). That dataset has to be treated as a loop artifact, not a generic training set. Its design-space schema, sampled-region coverage, tool versions, invalid-action labels, rejected regions, and proxy-fidelity calibration determine what a generative method may infer from it.

Chiplets raise the stakes because they turn partitioning and interfaces into legal actions inside the loop. A UCIe-class interface should be represented as an environment contract that specifies what may be split across dies, what bandwidth/latency/power/thermal feedback tools return, which package and software assumptions are fixed, and which candidates must escalate or be rejected when interface evidence is missing (UCIe Consortium, 2026).

Lighthouse prompt: The subsystem choice is an integration choice

Context. In an AI-assisted architecture DSE loop, “CPU extension,” “accelerator,” and “SoC block” are not interchangeable labels for the AI to explore.
In the Lighthouse prompt. “64-bit RISC-V-based compute subsystem” and “vector-capable CPU, accelerator, or SoC block” ask where the “XRBench-class real-time mobile XR workload” lives relative to the core, memory hierarchy, interconnect, compiler/runtime path, and product boundary. If the optimizer proposes a vector extension, it changes the CPU contract; if it proposes a tightly coupled accelerator, it changes the invocation, sharing, and memory-attachment contract.
Integration boundary. The method may still test a looser accelerator variant, but the loop environment must make the coupling assumption explicit so the optimizer cannot silently break integration contracts.

> Architect's checkpoint: The Cross-Layer Rejection Gate
>
> When an optimizer tunes a candidate architecture against a local proxy (like core area), the architect must enforce cross-layer boundaries. Does the loop environment automatically reject candidates that only win by silently pushing complexity into memory traffic, compiler support, verification, or SoC integration?

Read as a loop card, the task is architecture search; the representation is candidate parameters, workload state, constraints, tool configurations, and prior rejected regions; the environment is a simulator, mapper, compiler path, or staged DSE harness; the method posture is generation, prediction, optimization, and critique; the rejection authority is invalid actions, simulator mismatch, power or software incompatibility, and weak Pareto evidence; the commitment boundary is escalation to stronger feedback.

9.6 Domain-Specific Architecture and Code Generation

When architecture search narrows to a specific problem, domain-specific architecture is often presented as an efficiency story. If the domain is narrower, the hardware can be more efficient. That statement is true but incomplete. A domain is not one knob. It can be a kernel family, a model family, a data type, a memory-access pattern, a programming model, a deployment regime, a latency envelope, a product vertical, or an ecosystem of libraries and tools. The golden-age argument for specialization (Hennessy and Patterson, 2019) therefore creates a loop-design question. Which part of the domain is stable enough to specialize, and which part must remain programmable?

Use Amdahl here as a loop-contract check, not as an architecture refresher. The represented state is end-to-end workload fraction and offload granularity, the legal action is a specialization/interface choice, and the rejection rule is that a local speedup fails unless measured interface and software overheads still leave an end-to-end win. Every architect knows Amdahl's law (Amdahl, 1967), so restating it adds little. What follows is not a restatement but a relabeling of its variables. The version that matters here is the one that prices the interface. Hill and Marty re-derived the law for the multicore era to show how its lesson shifts when the substrate changes (Hill and Marty, 2008), and LogCA, an analytical model for hardware accelerators, models the accelerator case directly. It makes offload latency, per-invocation overhead, and operation granularity first-class terms alongside the raw acceleration (Altaf and Wood, 2017). The compact form used here keeps those costs visible:

$$S_{\text{system}} \leq \frac{1}{(1-f) + f/s + \epsilon_{\text{interface}} + \epsilon_{\text{software}}}.$$

Here, f is the fraction of the end-to-end workload that the specialized mechanism can improve, s is its local speedup, and the ϵ terms stand for the interface and software overheads that the LogCA-style model makes visible, expressed, like f and f/s, as

fractions of the baseline time so the terms share units. The reading is the one LogCA emphasizes. A design pays for specialization only when the accelerated work is large enough, and coarse enough per invocation, to amortize the cost of reaching the accelerator. A dramatic local speedup can still fail as an architecture result if the stable domain fraction is small, the interface is expensive, or the software path cannot keep up. This same form returns at the end of the chapter, lifted from one accelerator's speedup to the throughput of the entire design loop (Section 9.10). For an AI-assisted loop, this equation is a rejection contract. A proposed specialization is not credible unless the method also carries interface, software-path, and end-to-end evidence.

To formalize this interface tax, Figure 9.2 uses a simplified LogCA-style calculation (Altaf and Wood, 2017). End-to-end speedup is not a property of the accelerator alone. It rises with the work amortized per offload, and only after that granularity clears a break-even point set by offload latency and overhead. A tightly coupled unit breaks even at small granularity; an off-chip module may need thousands of operations per call before offload is worth doing at all.

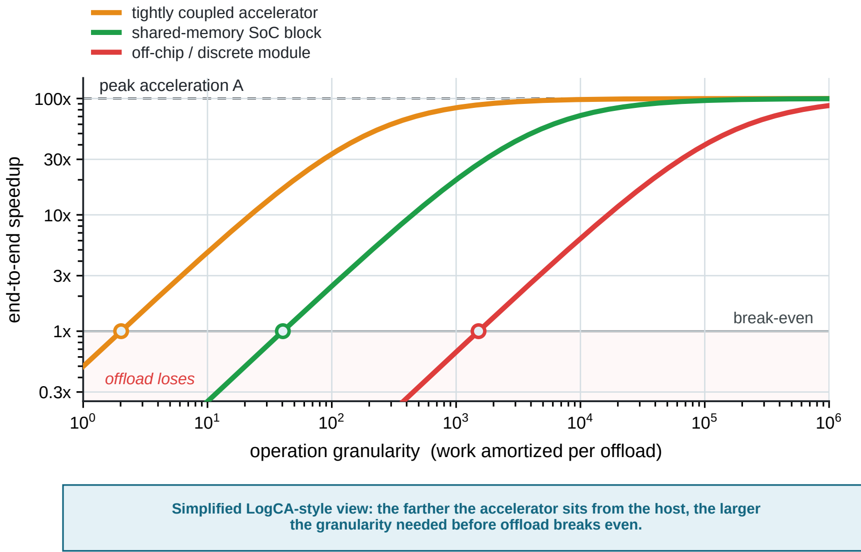


Figure 9.2: **The interface sets a break-even granularity:** In a simplified LogCA-style model (Altaf and Wood, 2017), end-to-end speedup rises with the work amortized per offload only after it clears a break-even granularity set by offload latency and overhead, then saturates at the accelerator's peak acceleration. The farther the accelerator sits from the host, the larger the granularity needed before offload even breaks even. Curves are illustrative; the lesson is the break-even cliff, not the exact values.

Consequently, Figure 9.3 breaks down the multidimensional nature of domain specificity. It should be read as a rigorous checklist rather than a simple taxonomy. Before a loop

proposes a domain-specific block, it should say what shape of domain it is using and what that choice implies for representation, action space, evidence, and maintenance. A benchmark name is not enough. Two workloads in the same named domain may have different memory behavior, precision contracts, software interfaces, or deployment drift. The kernel family, precision contract, data-movement pattern, software interface, drift risk, and test coverage determine which optimizer actions are legal and which claims a loop must reject.

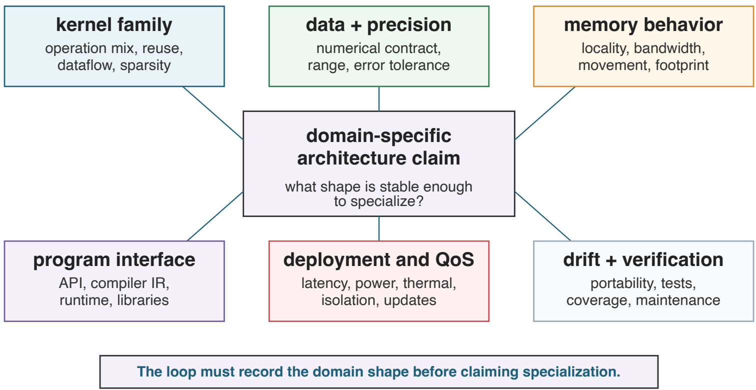


Figure 9.3: **Domain specificity has many shapes:** A domain-specific architecture is not specialized for a label; it is specialized for a bundle of kernel, precision, data-movement, interface, deployment, drift, and verification constraints. The loop must represent the shape it is claiming to exploit.

If the domain shape determines what to build, the software path is the loop's narrow waist. Halide-style algorithm/schedule separation, AutoTVM/Ansor-style measured schedules, and MLIR-style multi-level compiler representations matter here because they turn software reachability into represented state, legal mapping actions, compiler feedback, and rejection evidence for specialized hardware claims (Ragan-Kelley et al., 2017; Chen et al., 2018; Zheng et al., 2020; Lattner et al., 2020).

To link these constraints, Figure 9.4 visualizes this software path, showing why code generation is the narrow waist of specialization. Above the waist are domain intent and workload distributions. Below the waist are hardware mechanisms, tool feedback, profiling, simulation, and deployment evidence. The waist itself contains the programming model, compiler IR, libraries, runtimes, and generated code that let the workload reach the machine.

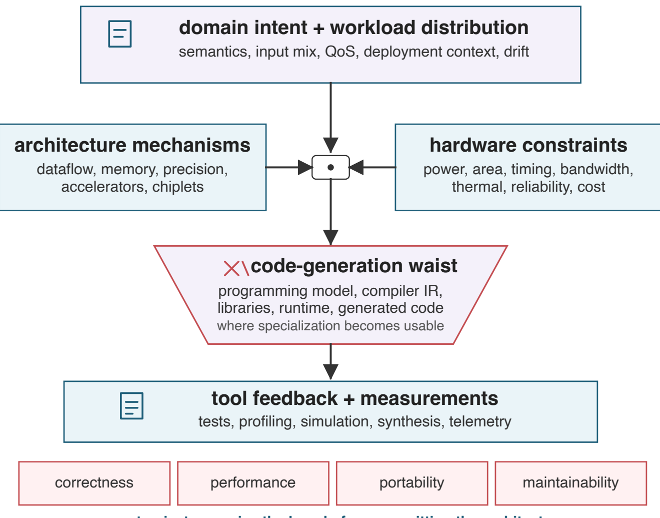


Figure 9.4: **Specialized hardware needs a code-generation waist:** Domain-specific architecture claims must pass through executable software paths and be checked with correctness, performance, portability, and maintainability evidence.

Generative methods may help at this waist, but only inside a represented loop. Kernel-generation benchmarks are encouraging precisely because they keep correctness tests, compilation, profiling, and target-specific behavior close to the generated artifact (Ouyang et al., 2025; Wen et al., 2025). For architecture, the same discipline must extend beyond one kernel. The loop needs workload semantics, data layout, scheduling constraints, backend capabilities, correctness tests, portability limits, and rejection authority. The architectural question is not only whether a specialized block is efficient. It is whether the hardware/software interface can keep that efficiency usable as workloads, models, compilers, and products change.

Read as a loop card, the task is not "generate code" in isolation. It is to keep a domain-specific hardware claim executable. The representation includes domain shape, kernel family, compiler IR, runtime interface, data layout, and maintenance evidence. The feedback budget is split. Kernel and schedule feedback is cheap and reversible, so a method may tune aggressively at that level, while the hardware-interface commitment it implies is expensive and hard to undo, so the loop must hold that layer behind independent evidence. The rejection authority is correctness, portability, interface cost, workload drift, and end-to-end system measurement. The commitment boundary is the point where a cheap kernel-level win gets read as a hardware-interface decision, which only end-to-end measurement and a human owner may cross.

Architect's checkpoint: The Hardware-Interface Commitment Gate

When the optimizer tunes a kernel aggressively, it often implies a hardware-interface commitment. The architect must explicitly define the commitment boundary where a cheap kernel-level win proposed by the model gets evaluated as an expensive hardware-interface decision, requiring end-to-end measurement and human sign-off.

Failing to secure this commitment boundary often leads to a classic architectural trap, where localized optimizations fail to deliver expected end-to-end performance.

Failure mode: The accelerator that won in isolation

An AI-generated specialized block posts a large kernel-level speedup and looks like a clear win. In the full system the gain mostly disappears: the compiler cannot generate code that keeps the unit busy, data movement to and from the block dominates, and only a hand-written kernel ever reached the headline number. The AI's hardware proposal was not wrong; the loop measured the wrong thing. This is the lesson the LogCA-style model makes visible. Price the interface and the software path before believing a local speedup, and reject the AI's candidate until a compiler-generated, end-to-end measurement survives.

9.6.1 Progressive Lowering as the Architecture Loop

Treating the compiler merely as a "rejection gate"—a static wall the hardware bounces against—is an Architecture 1.0 mindset. In an Architecture 2.0 loop, the compiler is not a wall; it is the co-design engine. If the AI loop is exploring a reconfigurable spatial array or a custom vector accelerator, the instruction set is not static. The AI must synthesize the hardware *and* the compiler pass to target it simultaneously. The software contract isn't a fixed boundary; it is a co-optimized variable.

This process is *Progressive Lowering*. When the AI agent decides to add a custom compute unit, it does not just output Verilog. It outputs *Intermediate Representation (IR) dialects*. It generates the new software dialect, the lowering rules to map high-level math into it, and the hardware dialect (using frameworks like CIRCT—Circuit IR Compilers and Tools—or MLIR) that implements it in silicon. The rejection authority is no longer just a simulator failing dynamically; it is the MLIR verifier proving statically that the lowering from software IR to hardware IR is semantically invalid (see Listing 9.1).

Listing 9.1 Co-generated hardware and software IR: In an Architecture 2.0 loop the same step emits the hardware dialect and the lowering strategy that targets it.

```
// Architecture 2.0 AI Loop Output: Co-generated HW and SW
// The agent writes the MLIR dialect and the lowering strategy simultaneously.

// 1. The AI defines the hardware intent using CIRCT (Hardware IR)
hw.module @CustomXR_SystolicArray(in %clk: i1, in %act: tensor<16x16xf16>, out %res: tensor<16x16xf16>) {
  // Agent-generated spatial architecture constraints
  %area_est = hw.predict_area(...)
  // Loop rejects if %area_est > mobile_XR_budget
  // Note: Area is heavily approximated here; true IR drop or routing congestion requires physical synthesi
}

// 2. The AI simultaneously generates the compiler scheduling policy
// using MLIR Transform Dialect to map the XR workload to the new hardware
transform.sequence failures(propagate) {
^bb1(%arg1: !pdl.operation):
  // Agent discovers tiling to 16x16 is optimal for the power envelope
  %0, %loops:2 = transform.structured.tile %arg1 [16, 16]

  // Agent generates the mapping rule to the custom hardware it just designed
  transform.structured.map_to_custom_hw %0 { target = @CustomXR_SystolicArray }
}
```

9.7 Co-Design Loops: Compute, Memory, Network, and Power

While domain-specific loops focus on vertical execution, co-design loops test the oldest claim in this book. Architecture is not only a layer below software but the place where software behavior, hardware mechanisms, physical constraints, and system objectives meet. A single-layer optimization can therefore be locally correct and globally wrong.

Consider a dataflow choice that reduces compute cycles but increases memory traffic, a topology change that improves one collective while worsening another, a cache change that improves average performance but increases tail latency, or a rack-level policy that saves power while violating service quality. The loop must represent more than one layer because the objective lives across layers.

Energy and warehouse-scale coupling matter here because they give the loop cheap rejectors for cross-layer claims. Data movement, memory locality, network contention, power and thermal headroom, and deployment policy can all invalidate a local compute win before expensive implementation evidence is gathered (Horowitz, 2014; Barroso et al., 2019).

An Architecture 2.0 co-design loop therefore needs richer representations, such as workload phase behavior, data movement, memory locality, network behavior, power and thermal constraints, compiler/runtime choices, and deployment policy. The useful method roles are coordination and critique as much as search. The loop should ask which layer changed, which objective improved, which objective worsened, and which

evidence would reject a single-layer win. In an AI-assisted co-design loop, coordination means naming which layer the automated optimizer may touch, which layer supplies independent feedback, and which owner accepts the cross-layer tradeoff.

Architect's checkpoint: The Cross-Layer Ownership Gate

Before an optimizer acts in a co-design loop, the architect must explicitly name which layer the optimizer may touch, which layer supplies independent feedback, and which human owner holds the authority to accept the cross-layer tradeoff.

Warehouse-scale computing itself, read as a loop card, is a cross-layer co-design pattern. The task is useful work per constrained resource; the representation is workload, software, hardware, network, power, cooling, and operations state; the environment combines models, traces, deployment measurements, and operational rules; and the rejection authority is a system objective that a local win violates.

Many Architecture 2.0 loops are even less static than this summary suggests. In early accelerator, compiler, runtime, or system-interface work, the hardware target and the software path may both be changing. The loop is then not merely searching a fixed space; it is managing hardware/software co-evolution. A new instruction, tensor unit, scratchpad policy, data layout, or runtime interface changes what software can express, while compiler, kernel, and workload feedback changes which hardware feature is worth keeping. The evidence ledger has to record that co-evolution by tracking what hardware state existed when the software was generated, which examples or optimization rules were available, which correctness checks and performance measurements returned, and which hardware or software assumption was revised after rejection.

The durable lesson is not that code generation solves architecture. It is that software feedback can become part of architecture evidence when the loop keeps the hardware target, code path, tests, performance counters, rejected variants, and decision owner tied together. Without that record, a strong generated kernel or library path may only prove that one software artifact matched one hardware snapshot.

To manage this co-evolution, Figure 9.5 illustrates the rigorous loop discipline required to keep evidence aligned. Because the hardware side and software side may both change, so the evidence ledger must record the versions, measurements, failures, and rejected variants that connect them. Otherwise, the loop cannot say whether the architecture improved or only whether one generated path happened to work.

Google's warehouse-scale migration from x86 to Arm is a deployed instance of this discipline (Christopher et al., 2025). It is not a hardware co-design loop. The target ISA is fixed, and the loop repairs software to meet it. But moving a production fleet of core services onto a new instruction set is a fleet-scale software-repair loop in which tens of thousands of applications must follow, and it is the book's one measured exemplar; no measured cross-layer hardware co-design loop is offered. The reported effort analyzed more than 38,000 migration commits and wrapped part of the software change in an AI-assisted build-and-test repair loop, accepting a candidate fix only when the build and the test suite pass, and letting a fleet monitor automatically evict jobs that crash-loop or

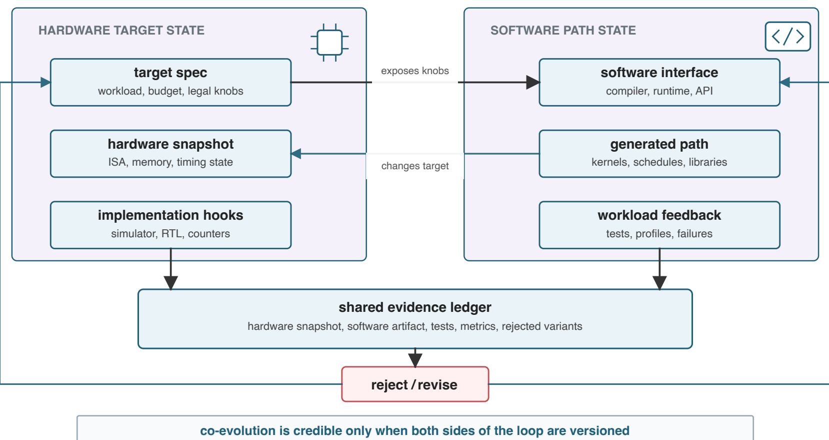


Figure 9.5: **Hardware and software co-evolve through shared evidence:** Early architecture work often changes the hardware target and the software path at the same time. The loop becomes credible when the hardware snapshot, software artifact, tests, metrics, rejected variants, and decision owner are recorded together.

run slow on the new target. Authority sits with the instruments, not the generator. Build, test, sanitizers that expose memory-model differences between the two ISAs, and the production monitor that rejects regressions and preserves the evicted cases as negative traces for offline debugging. The repair tool resolves only a minority of failures on its own, roughly a third of broken tests without tuning, and the hardest changes stay with the application owners who know the code. The model behind the loop will age; the structure, a represented migration loop with instrument-held rejection authority and human-owned exceptions, is the part worth keeping.

When read through the same card, the migration case is not primarily an Arm story; it is a loop-contract story. Table 9.4 summarizes the reusable fields that structure this contract.

Table 9.4: ISA migration as a loop card: The reusable lesson is the structure of the migration loop, which requires represented state, independent rejection, preserved failures, and human-owned exceptions.

Card field	Migration reading
Task	Move production services across an ISA boundary while preserving build, test, and rollout behavior.
Representation	Migration commits, build targets, tests, sanitizer findings, service ownership, and evicted production cases.

Card field	Migration reading
Environment	Build-and-test repair loop plus production monitoring on the new target.
Method role	Propose bounded software repairs and route failures toward owners.
Cheap independent rejector	Build failures, test failures, sanitizer failures, crash loops, slow jobs, and monitor-triggered evictions.
Commitment boundary	Automated repair can clear only the failures its instruments settle; application owners keep the exceptions.

The most instructive result is what the evidence overturned. Going in, the team and the application owners expected the work to be dominated by low-level architectural differences like floating-point drift, concurrency, and platform-specific intrinsics. The recorded evidence showed otherwise. Most of the effort was tests that encoded x86-specific assumptions, build and release systems, and configuration rollout. Unaided architectural intuition mispredicted where the work actually was. That is the case for a represented, evidence-bearing loop in a single sentence. The loop found the work the experts did not expect.

Lighthouse prompt: Migration evidence shows what ISA claims owe

Context. The ISA migration case shows why the lighthouse phrase "64-bit RISC-V" is not only a hardware target for the optimizer to select.
In the Lighthouse prompt. If the model chooses a "64-bit RISC-V-based" path for the "compute subsystem", it means the AI-assisted loop owes evidence for generated code, ABI and memory-model assumptions, compiler/runtime behavior, library support, tests, and deployment rollout, not just an instruction encoding.
Boundary. The migration case is not the same target, but it reveals the kind of software-contract evidence a RISC-V subsystem claim would eventually need. Build, test, sanitizers, production monitors, and application owners become rejection authorities for the AI's ISA choice.
Takeaway. ISA decisions proposed by an automated optimizer are credible only when the loop can show what software actually survives the contract and what remains a human-owned exception.

9.8 Deployment-Facing Architecture Loops: Runtime, Serving, and Datacenter Policy

Moving from design and migration to active operation, deployment-facing architecture loops evaluate architecture choices in deployed or deployment-like contexts. They include scheduling, serving, admission control, memory allocation, power management,

placement, rollout policy, fleet telemetry, and performance isolation. Their feedback can be richer than simulation because it comes from real systems. It can also be noisier, more confounded, more privacy-sensitive, and harder to reproduce. The deployment-facing benchmark lesson from the workload-pattern discussion still applies here. Evaluation has scenarios, latency constraints, power envelopes, and system rules, not only a single accuracy or throughput number. Fleet loops add another layer, consisting of live telemetry, canary policy, rollback, and operational review. For automated optimizers acting inside these loops, telemetry is governed data. The schema, cohort provenance, canary labels, privacy filters, drift detectors, rollback triggers, and preserved failure traces define what the loop may trust.

The design-loop card changes accordingly. The representation must include operational state, workload drift, service-level objectives, resource contention, customer or user constraints, and rollback mechanisms. The environment may be a simulator, test cluster, replay harness, staging system, or production system with guardrails. The method may adapt policy, detect drift, summarize telemetry, or propose a controlled experiment. For an AI-assisted deployment-facing architecture loop, the legal actions should be enumerated narrowly, such as adjusting a scheduler weight, proposing a canary, changing an admission threshold, or selecting a replay cohort, and every action should name the telemetry feedback, rollback trigger, and human owner that can stop it.

Service-level objectives (SLOs): target values for system reliability and performance, commonly used to manage operational tradeoffs (Beyer et al., 2016).

The rejection authority is often operational. A service-level objective, a canary guard, an alert, a safety policy, a privacy constraint, or a human operator can stop a rollout. Crucially, in a modern datacenter, this fleet-level rejection authority must tightly govern network congestion and tail latency SLAs, not merely average throughput or node-level power limits. This makes systems loops different from pure software loops. They may be reversible, but reversibility is not free. A bad policy can waste power, violate latency targets, create interference, or damage user experience before it is rolled back.

Architect's checkpoint: The Operational Rejection Gate

In AI-assisted deployment-facing architecture loops, reversibility is not free. The architect must ensure that the AI's proposed policy changes are gated by an operational rejection authority, such as a canary guard or SLO monitor, that can stop a bad rollout before it wastes power or damages user experience.

This pattern is important for Architecture 2.0 because architecture is increasingly evaluated through deployed behavior. A design that looks strong under a static benchmark may face different workload mixes, model versions, traffic patterns, or fleet policies. The loop must therefore connect architecture evidence to system evidence without pretending that production telemetry is a clean oracle.

Read as a loop card, the task is guarded adaptation or policy revision; the representation is operational state, workload drift, SLOs, resource contention, and rollout state; the environment is a replay harness, staging system, or production system with guardrails;

the method posture is monitoring, critique, controlled experimentation, and conservative adaptation; the rejection authority is SLO violation, safety or privacy policy, rollback, and human operations review.

9.9 High-Commitment Loops: RTL, Physical Design, and Verification

At the furthest end of the spectrum from software, high-commitment loops stress-test the framework. RTL changes, generator-level edits, physical design, timing closure, layout, power analysis (including IR drop and thermal integrity), Design Rule Checking (DRC), formal verification, signoff, and silicon-facing decisions are expensive to evaluate and costly to get wrong. Feedback is delayed. Tool flows are complex. Evidence must survive independent checks. The human commitment level is high.

For the lighthouse prompt, this is the point where a candidate subsystem stops being a plausible design-space result and starts making claims that must survive RTL checks, physical constraints, power analysis, verification, and integration review.

This does not mean Architecture 2.0 is irrelevant. It means the method posture should change. In high-commitment loops, the most valuable roles may be critique, search narrowing, evidence organization, bounded repair, test generation, report summarization, and inconsistency detection. A system that finds missing assumptions, organizes tool evidence, explains why a candidate failed, or narrows a physical-design search space may be more useful than one that claims to make final autonomous decisions.

Learning-assisted chip placement is a prominent, and disputed, example of a design subflow being formulated as a learning problem (Mirhoseini et al., 2021). Chapter 7 discusses the later baseline and reproducibility challenge. The Architecture 2.0 lesson is not that all physical design should be handed to an automated optimizer. It is that even when learning methods help, the loop still needs tool constraints, baselines, provenance, rejection authority, and human commitment.

The regime also has public success reports, and they sharpen the same lesson rather than contradict it. Vendor-reported reinforcement-learning systems have searched the physical-implementation flow for commercial tapeouts at scale (Synopsys, 2023), and peer-reviewed work on reinforcement learning over a bounded circuit space produced arithmetic units that shipped in GPU silicon (Roy et al., 2021). Reported results in this space are not uniformly settled, but the debate is specific, not blanket. The learned macro-placement result in particular had its baselines and reproducibility publicly debated (Chapter 7); the peer-reviewed PrefixRL result, by contrast, shipped in silicon, and the vendor-reported physical-design flow stands at arm's length as not independently reproduced. That is precisely why the loop's evidence and gates, not the headline, are what earn trust. What makes these loops credible is not the method alone; it is that outputs are routed through independent tool checks and signoff gates before commitment, with the exact gate depending on the flow and source. They also surface the architect's

economic question. A search that costs many machine-hours per block is justified only when the result is amortized. A generated circuit instantiated across every shipped unit of a high-volume part repays its search cost in a way that a one-off block never could. The high-commitment regime therefore rewards methods whose cost amortizes over many uses and whose outputs survive independent checks. The inspectable loop record matters more than the success label. The initial design state, allowed flow knobs, tool versions, failed candidates, signoff checks, waivers, escalation points, and the accountable commitment owner must travel with the result.

The rejection authorities in this regime are strong. They include parsers, type checks, regression suites, formal tools, synthesis, timing, power analysis, layout rules, signoff flows, integration review, and expert judgment. A candidate that fails here is not a near miss to be explained away. It is evidence that the loop must revise its representation, action space, method, or claim.

Read as a loop card, the task is bounded RTL, generator, or physical-design improvement; the representation is implementation state, constraints, tests, tool reports, waivers, and review decisions; the environment is a staged tool flow with scarce high-fidelity samples; the method posture is narrowing, critique, bounded repair, evidence organization, and test generation; the rejection authority is independent tool and expert review; the commitment boundary remains architect-owned.

9.10 The Loop Is Rejection-Bound

These loop families look different on the surface, yet they succeed and fail for the same reason. Architects instinctively treat a slow design loop as *generation-bound*, as if the fix were to propose more candidates faster. It is not. Just as a kernel can be memory-bound rather than compute-bound (Williams et al., 2009), a design loop is *rejection-bound*. Its throughput is set by how fast trusted, independent rejection can be applied, not by how fast candidates can be produced. That reason can be written down. The interface speedup form used earlier for accelerators lifts cleanly from one block to the whole loop. There it priced the cost of reaching an accelerator. Lifted one level, it prices the cost of reaching a decision. The resulting bound measures the loop's speedup—how much useful design progress the loop makes per commitment once the cost of trusted feedback is paid.

Keep Amdahl's f, but raise what it counts. It is no longer the fraction of computation that an accelerator speeds. It is the fraction of design commitments that a cheap, independent, trusted rejection authority can discharge without escalating to expensive feedback or human judgment. This is rejection authority (Chapter 7) in its algebraic role; this section uses rejector only as shorthand for the authority that applies the check. Take the slow loop that checks every commitment at full fidelity as the baseline. Introduce a cheap rejection authority that clears a fraction f of commitments at a small fraction of that cost, and the loop's throughput obeys

$$S_{\text{loop}} \le \frac{1}{(1-f) + f \cdot (c_{\text{cheap}}/c_{\text{hi}}) + \epsilon_{\text{escalate}}}.$$

Here f is the fraction of commitments a cheap, independent, trusted instrument can settle without escalating by catching a violation and rejecting it, or clearing it at the commitment level the stage is deciding. Rejection is what such an instrument does most reliably, and acceptance is only ever as strong as the check and the commitment behind it. A green test suite is enough to accept a code fix, but not to certify a tapeout. The commitments that need a stronger acceptance than the cheap instrument can give fall in $(1-f)$, the irreducible part that only higher-fidelity feedback or human judgment can clear.

The ratio $c_{\text{cheap}}/c_{\text{hi}}$ is the cost of a cheap check relative to a high-fidelity one, and $\epsilon_{\text{escalate}}$ is the escalation overhead, defined as the rate at which the cheap rejection authority defers times the cost of each climb up the fidelity ladder, expressed relative to c_{hi}. The form charges the cheap check only on the commitments it settles; a rejection authority that must screen every candidate pays c_{cheap} on all of them, which only lowers S_{loop} further, so the relation is written as an upper bound, with equality in the idealized case where triage and escalation overheads vanish.

The bound holds on one condition. The cheap rejector's clears must be *true* clears. A false pass, a candidate the cheap check waves through that a stronger stage would have rejected, does not speed the loop; it corrupts it, and repaid later at higher commitment it makes the effective f a fiction. So f counts only true, independent clears, which is why the rejector must not share the generator's blind spots (Chapter 7). The honest bound is S_{loop} conditional on a bounded false-pass rate, not a promise that any cheap check that says yes has raised f. The condition is two-sided. A loop that rejects everything scores $f = 1$ and maximal S_{loop} while producing nothing, so a false reject, a good candidate the cheap check discards, corrupts the bound as surely as a false pass. A *true clear* means the cheap check's disposition, accept or reject, matches what full fidelity would have decided; the bound is honest only under a bounded rate of both errors, read at a fixed design quality so the comparison holds the answer fixed the way Amdahl's does.

The form is Amdahl's law as LogCA prices it (Amdahl, 1967; Altaf and Wood, 2017). The cheap rejection authority's local speedup is just $c_{\text{hi}}/c_{\text{cheap}}$, so the middle term is exactly Amdahl's f/s, and escalation overhead plays the role the interface cost played for the accelerator. The form is borrowed and credited. It is a re-coordinatization of Amdahl's law, not a new law; what is new is what the variables name, what they rule out, and the protocol that would falsify them.

What they rule out is the thing most easily mistaken for progress. Candidate count does not appear in the bound. That much is nearly a tautology, since S_{loop} is a per-commitment ratio and the candidate count divides out. Candidate count still sets the loop's total load and wall-clock; what it cannot move is the per-commitment throughput S_{loop}, which improves only when rejection coverage, rejection cost, or escalation overhead improves. The substantive claim is that generation and rejection are ordinarily independent levers. Proposing more designs raises the candidate count but not f, so a loop can multiply its proposals by a thousand and move S_{loop} not at all. The one way out is a generator good

enough to propose only obviously-good or obviously-rejectable candidates, which would raise f itself, a better rejector wearing a generator's clothes. Short of that, the bound exposes only three levers, which are to raise f, lower the cost ratio, or lower escalation overhead.

This holds cleanly for *adjudication* loops, where an exogenous stream of commitments arrives to be accepted or rejected; the fleet migration is the exemplar, its candidates given and its throughput set by how cheaply each is disposed. *Search* loops behave differently, and Chapter 6 is right about them. When the loop generates its own population to reach a target, a better generator legitimately lowers the number of evaluations to get there. The two are not in conflict. In a search loop that gain shows up *as* a higher f, because a generator that proposes only obviously-good or obviously-rejectable candidates makes the cheap check decisive more often. Either way, throughput moves through f, not through raw candidate count.

A paper that tests the bound would need loop-turn traces, not only faster generators. Each candidate commitment would be labeled rejected, cleared, or escalated; cheap-check and high-fidelity costs would be measured; independence between generator and rejection authority would be tested; and false-pass or false-reject rates would bound the strongest claim the cheap authority can support. Without those labels, f is only rhetoric.

Engineer move: Measuring the rejection bound

To estimate f for a real AI-assisted loop rather than assert it, fix a protocol before the run.

- Commitment unit. State what one commitment is (an AI candidate accepted for the next fidelity stage, an RTL change merged, a signoff waiver), so f is a rate over comparable events.
- Outcome labels. Label every AI candidate commitment `cleared`, `rejected`, `escalated`, or `waived`, and reserve `false pass` and `false reject` for outcomes overturned later.
- Costs. Record the cheap-check cost c_{cheap} and the high-fidelity cost c_{hi} actually paid, not list prices.
- Independence. Disclose whether the rejector shares data, model, or authorship with the AI generator; a rejector that does is not independent, and f is inflated.
- False-reject audit. Periodically escalate a random sample of rejected candidates to higher fidelity. The fraction that would have cleared estimates the false-reject rate and bounds how aggressive the cheap authority may safely be.

Without this protocol, f is a story about the AI-assisted loop, not a measurement of it.

Recognizing this fundamental limit reshapes our approach to scaling architectural exploration.

Design principle: The loop is rejection-bound, not generation-bound

An AI-assisted design loop's throughput is bounded by its rejection capacity, not the AI's generation capacity. AI candidate count is absent from the bound. The loop speeds up only when a cheap, independent, trusted rejector discharges a larger fraction of AI-proposed commitments, when trusted feedback gets cheaper relative to high-fidelity evidence, or when escalation gets cheaper. The loop scales as far as its cheapest independent rejector, and no further.

Three readings follow, and each was argued qualitatively in an earlier chapter. First, generation is not the constraint. Cheap generation raises the candidate count, which the bound ignores, and it does not raise f. This is Chapter 6 restated as algebra, where a generated artifact is a proposal and the loop speeds up only when something can cheaply and independently reject it. Second, independence is load-bearing, not decorative. Only a rejector that does not share the generator's blind spots counts toward f. A cheap check coupled to the generator inflates the apparent f while real escapes survive, which is the failure Chapter 7 warns against when a learned judge and a learned generator quietly become the same witness. The honest f counts only what a cheap, independent, trusted instrument settles on its own. Third, $(1 - f)$ is a floor. The commitments that only silicon, deployment, or a human can settle set a ceiling on loop throughput that no amount of cheap, abundant generation removes. It is Amdahl's serial fraction, written for the design loop.

The bound is not only a way to talk; its variables are estimable on a real loop. The fleet ISA migration earlier in this chapter gives one reading. Turned on the broken-test repair slice the migration left behind, the automated repair tool proposed fixes, and the cheap, independent instruments (build, test, sanitizers that expose memory-model differences, and the production monitor) settled roughly a third of those commitments without a human; the rest escalated to the engineers who owned the code. That is an f of about 0.3, with a $(1 - f)$ floor of human-owned commitments, the shape the bound predicts. Progress tracked the fraction the cheap, independent instruments could settle without a human, not the number of fixes proposed. This estimates f for that repair slice, not the entire migration loop, and it is a joint number. The third that cleared reflects both the repair tool's quality and the instruments' coverage, so a stronger generator would move it. Reading f cleanly, holding the generator fixed and varying only the rejector stack, is the measurement the bound still owes. One measured point is not a validated law, but it shows the variables can be read off a deployed loop rather than only defined on paper.

The bound also makes the scissors gap of Chapter 2 computable. The gap widens precisely when one blade, the rate of proposals and evidence demands, races ahead while the other blade, the rate at which trusted feedback can reject, stays fixed. In these terms the gap is the regime where candidate count climbs while f does not, and the bound says throughput is flat in exactly that direction. The loop families walked through this chapter are operating points on the same curve. Fast software loops sit near high f, cheap feedback, and low escalation, so they run quickly. High-commitment RTL and physical-design loops sit near low f, expensive feedback, and high escalation, which

is why the method posture there shifts away from autonomous action toward critique, evidence organization, and rejection. The bound derives that advice rather than asserting it.

9.11 What Transfers Across Loops

Across all of these loops, the same ontology and the same bound transfer. Each loop has a task, a representation, an environment, method roles, feedback, evidence, rejection, and human decision. Each loop can preserve negative traces. Each loop can be reviewed with the design-loop card. Each loop can fail by hiding assumptions, optimizing a proxy, omitting provenance, or letting an output become a decision too early.

What changes is the operating regime, which the rejection-bound relation gives a compact way to name using the value of f, the cost ratio between cheap and high-fidelity feedback, and the escalation overhead. Feedback latency changes. Reversibility changes. Action validity changes. Data availability changes. Security and IP constraints change. The cost of being wrong changes. Rejection authority changes. The acceptable level of autonomy changes.

Design principle: Change autonomy with loop pattern

Autonomy is a property of the AI-assisted loop contract, not the AI model. Let the optimizer take more autonomous action only where feedback is cheap, rollback is easy, and independent rejection can stop bad actions. As commitments become expensive, irreversible, or system-wide, shift the AI-assisted loop toward critique, evidence organization, rejection, and human approval.

9.12 Conclusion

This chapter asked how the AI's role and the loop contract should change as feedback gets more expensive and commitments get harder to reverse across the stack. The ontology holds everywhere. Every loop, from a millisecond software tuner to a months-long fleet decision, has a task, a representation, an environment, method roles, feedback, evidence, rejection, and a human decision, and every loop can fail the same ways, by hiding assumptions, optimizing a proxy, dropping provenance, or letting an output become a decision too early. What differs is the operating regime, not the anatomy.

That regime is what the rejection bound names compactly, through the cost ratio between cheap and high-fidelity feedback and the overhead of escalating. As feedback latency, reversibility, and the cost of being wrong rise, throughput stops being set by how fast candidates are generated and starts being set by how cheaply and confidently they can be

rejected. The loop is rejection-bound, not generation-bound, and that is what makes the same framework read a fast compiler loop and an RTL signoff flow so differently.

The consequence is a rule for autonomy. It is a property of the loop contract, not of the model. Let the optimizer act more freely only where feedback is cheap, rollback is easy, and an independent check can stop a bad action, and as commitments become expensive, irreversible, or system-wide, shift it toward critique, evidence organization, rejection, and human approval. Reading any real project then comes down to a few honest questions. Which loop pattern is this, what feedback can it afford, what can reject its result, and which decision must stay with the architect?

9.13 Open Research Questions

The loop patterns outlined in this chapter define operating regimes, but operationalizing them across architecture research practice exposes several unsettled research directions within the Architecture 2.0 ontology. The following questions push beyond classifying loops to ask how their evidence and boundaries can be formalized at scale:

1. **Can we formalize a cross-layer evidence ledger that is queryable within stated schemas and composable across tools?** Rather than simply constructing an evidence corpus, the challenge is establishing a formal semantic schema for negative traces and rejected variants that spans simulation, compilers, and physical-design tools. A strong PhD thesis here could define an open ledger format where a compiler-optimization agent can query the rejected chiplet-partitioning trials of a hardware agent to avoid repeating the same physical violations, reducing cross-loop data silos.
2. **How can we mathematically or empirically certify the false-pass bounds of cheap rejection authorities?** As formalized by the concept of the rejection bound (see the discussion on "The Loop Is Rejection-Bound" in Section 9.10), an AI-assisted loop's throughput depends entirely on the fraction of commitments a cheap instrument can settle. A top-tier systems challenge is developing stress-testing methodologies or formal-verification subsets that certify exactly when a fast software loop or surrogate proxy will silently pass a system-breaking bug, bridging the gap between fast heuristics and high-commitment RTL signoff.
3. **Can AI agents recommend posture changes based on real-time escalation cost and proxy mismatch?** While static method roles map to fixed loop patterns, the next leap is bounded meta-reasoning, models that monitor false-pass rates and feedback latencies to recommend posture changes, such as moving from generator to critic, under human-defined bounds. A major ASPLOS or ISCA paper could demonstrate a controller that continuously tracks the escalation overhead defined by the rejection bound (see the discussion on "The Loop Is Rejection-Bound" in Section 9.10), restricting the agent's action space as the fidelity of the environment or the cost of rollback changes.

4. **Is it possible to formalize the causal decay of architectural evidence across co-evolving stack layers?** As hardware targets and compiler intermediate representations co-evolve, early performance evidence ages rapidly. A foundational research direction would introduce a formal evidence-transfer calculus that invalidates or degrades the confidence of an architectural claim when a downstream dependency, such as a workload distribution or a runtime policy, drifts, helping prevent generative optimizers from composing systems out of stale, incompatible evidence.

What to carry forward

- **Reader test:** Which AI-assisted loop pattern are you in, and what cheap, independent check can actually reject the automated optimizer's result?
- **Up next:** Once AI-assisted loop patterns can be compared, the next question is what the architect still owns across all of them when delegating to AI.

Chapter 10

What the Architect Owns

"The architect's two most important tools are the eraser at the drafting board and the wrecking bar at the site."

— Frank Lloyd Wright, Architect

The crux

When automated participants, tools, and architecture environments can act inside the loop, what stays the architect's to own, and what must the field share so ownership can be compared?

Author's Note: Frank Lloyd Wright, the iconic American pioneer of organic architecture, recognized that the power to destroy and revise is just as important as the power to create. For us, this summarizes the conclusion of the book: while AI will generate the designs, the human architect retains the ultimate authority to reject them and owns the final deployment commitment.

To answer those questions, consider the lighthouse prompt that opened the book. It asked for a low-power, 64-bit RISC-V-based compute subsystem for real-time mobile XR under a 3 W target in a 3 nm-class low-power mobile process. At the start of the book, that prompt was a provocation. It looked like a request for a future hardware foundation model. At this point, it should read differently. The prompt is not powerful because it is short. It is powerful because it exposes a missing design loop.

Hardware foundation model: a foundation model is a large-scale machine learning model trained on a vast quantity of data that can be adapted to a wide range of downstream tasks; a hardware foundation model would apply this approach to hardware design representations.

That loop includes workload definition, architecture representation, tool interfaces, compound method roles, feedback budgets, evidence ledgers, rejection authority, and human decisions. It also exposes the central conclusion of Architecture 2.0. AI systems do not eliminate the computer architect. They change what the architect must own.

10.1 The Architect Owns the Loop

The architect's responsibility moves upward. The architect does not own every intermediate answer alone; the architect owns the loop and the commitment it supports. Instead of owning every step of manual artifact construction, the architect owns the framing of the problem, the abstractions that make it tractable, the representations that make it legible, the evidence standards that make it believable, the rejection authority that makes it safe to use, and the commitment behind the final decision. This is why Architecture 2.0 is not simply an automation story. It is a responsibility story about loops in which AI-assisted systems can act but cannot own the commitment.

That responsibility is sharper in hardware than in software. A generative model or an automated optimizer does not sign a tapeout check, and unlike a software defect that an overnight update can patch, a hardware bug ships in silicon, where a respin costs millions of dollars and months of schedule. The architect therefore has to own the commitment gate explicitly. The job shifts from crafting the artifact by hand to standing behind the evidence that justifies committing it, without ceasing to own the artifact's consequences.

> **What this chapter gives you**
>
> After this chapter you can draw the architect-owned commitment boundary for a loop. That means you can:
>
> - name which architecture responsibilities remain human-owned when automated tools act inside the loop;
> - decide which state, evidence, rejection authority, and workload records a loop must expose before delegation is credible;
> - answer the strongest objections to Architecture 2.0 without treating automation as ownership;
> - identify the community infrastructure the field still needs for comparable, evidence-bearing loops;
> - state why every field of the design-loop card ultimately resolves to an accountable human owner.

10.2 Return to the Moonshot

To see how these shifted responsibilities manifest in practice, we must return to the lighthouse prompt that provoked this inquiry in the first place:

> Design a low-power, 64-bit RISC-V-based compute subsystem for an XRBench-class real-time mobile XR workload. Realize it as a vector-capable CPU, accelerator, or SoC block under a 3 W TDP target in a 3 nm-class LP mobile process, and return a design-space report with evidence and rejected alternatives.

This prompt may seem straightforward, but it implies a set of complex interface contracts. At first glance, it looks like a standard hardware specification—a list of targets, constraints, and deliverables. However, beneath the surface, it binds together workload assumptions, software dependencies, architectural representations, and physical constraints. For an AI-assisted loop to act on this prompt, these implicit contracts must be explicitly surfaced as machine-readable constraints and rejection criteria.

Lighthouse prompt: The prompt hides interface contracts

Context. The opening request is not a single hardware specification. It bundles workload, software, architecture, technology, and evidence obligations that must be made explicit before an automated participant can act on them.
In the Lighthouse prompt. "XRBench-class real-time mobile XR workload" creates an obligation to specify a workload distribution, quality-of-experience target, benchmark version, traces, and software stack. For an automated system, this workload obligation has to be represented as machine-readable data: scenario labels, trace provenance, coverage limits, excluded cases, quality-of-experience labels, and out-of-distribution (data or inputs that differ significantly from the examples a model was trained on, often degrading performance) rejection conditions. "64-bit RISC-V-based" means the AI loop needs access to correct compiler toolchains, simulators, and instruction semantics, rather than just generating text that looks like RISC-V. The fragment "vector-capable CPU, accelerator, or SoC block" defines the system's allowed action space. The loop must be given explicit rules about which integration paths it can explore and which interfaces are fixed.
Evidence boundary. "3 W TDP target" and "3 nm-class LP mobile process" create power, thermal, and physical-design constraints; an optimizer evaluating candidates must not confuse a fast proxy average-power estimate with the sustained thermal or signoff evidence required for a final commitment.
Takeaway. The architect-owned work is deciding which contracts are fixed, which are exposed as actions to the automated tool, and which have enough evidence to support a human commitment.

This need for explicit evidence introduces a hierarchy of commitment. As an AI-assisted design loop explores candidates, it must navigate different levels of fidelity—from fast, imprecise proxies to expensive, cycle-accurate physical simulations. The architect must define the escalation path, deciding exactly what level of evidence is required to advance a design to the next stage without incurring unacceptable risk. This progression is captured in the commitment ladder (Figure 10.1), which maps the rising standard of evidence against the increasing cost of being wrong.

The requested deliverable also matters. A design-space report is different from RTL. RTL is different from a verified implementation. A verified implementation is completely different from a physically routable, timing-closed tapeout. (Logical correctness in RTL tells us nothing about whether power delivery, thermal bounds, and wire routing can physically close through rigorous signoff tools.) Each step changes the evidence standard. The same prompt can support a brainstorming loop, a research prototype, a simulator-backed design-space exploration, or a high-commitment implementation loop. Treating all of those deliverables as the same is one of the fastest ways to overclaim.

To avoid this overclaiming, the framework developed here turns the prompt into a set of explicit questions: What task is being solved? What representation is available? What world model does the loop assume? What tools can the system act on? What method roles are allowed? What feedback is affordable? What evidence would make the result

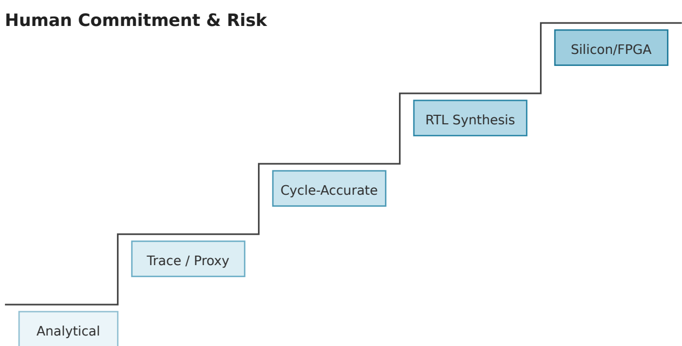


Figure 10.1: **The Commitment Ladder:** The evidence standard must rise as the cost of being wrong increases. A fast generative screen requires low evidence; a final silicon tapeout requires physical design closure. Each rung represents an architect-owned escalation threshold.

credible? What alternatives were rejected? Who can stop the loop? Who is accountable for the decision?

Together, those questions define the part of the design space the architect exposes to methods, tools, and AI-assisted systems. Choosing what to expose is an architectural act. Expose too little and the loop cannot find useful alternatives. Expose too much without constraints, evidence, and rejection authority, and the loop can optimize the wrong object or cross a commitment boundary before anyone understands what happened.

Architect's checkpoint: The Exposure Gate

When defining an AI-loop's environment, check the boundary of exposed actions. Have you provided explicit constraints, evidence checks, and rejection authority for every knob the automated optimizer can turn? If not, it may optimize an invalid metric and cross a commitment boundary unnoticed.

Enforcing those boundaries marks the shift from prompt to loop. The prompt motivates the work. The loop makes the work inspectable.

The phrase *prompt-to-loop* is deliberately more durable than prompt-to-chip. Prompt-to-chip asks whether a system can produce an impressive final artifact from a sentence. Prompt-to-loop asks whether it can preserve the task, state, tools, evidence, rejected alternatives, and human decision boundary that make the next architectural commitment believable. The second question remains important even after today's models and tools are obsolete.

10.3 Nondelegable Architectural Responsibilities

Even the most explicitly defined, inspectable loop still leaves work the architect cannot hand off, and nondelegable does not mean unaided. An architect can use models, AI-assisted systems, search procedures, simulators, compilers, profilers, EDA tools, benchmarks, and critics. The point is narrower and more important. The architect cannot transfer responsibility for the architectural judgment itself into a model. Nor does responsibility become distributed simply because several automated participants divide the work. Splitting a loop across these tools can make delegation more effective, but it also creates more places where authority can become ambiguous. The architect still has to decide which roles may act, which system or tool can stop another, which evidence crosses an escalation threshold, and who owns the commitment when the tools disagree or all agree for the wrong reason.

Architect's checkpoint: The Multi-Agent Authority Gate

When splitting an architecture task across multiple automated participants, trace the rejection authority. Which system or tool is allowed to halt the process? What evidence crosses an escalation threshold back to a human? You must ensure that adding more participants does not diffuse final accountability.

Other high-stakes fields have already learned, at real cost, where that authority has to sit.

Field note: The reviewers who could say no, too late

In clinical genomics, published predictors claimed to match cancer patients to the chemotherapy they would respond to, and were used to enroll real patients in trials. Two outside statisticians who tried to reproduce the analysis found it built on mislabeled, shifted data, and the predictors were worthless (Baggerly and Coombes, 2009). The authority to reject on that evidence existed, but it sat outside the loop, with reviewers never given a seat before commitment. By the time the objection was heard, 117 patients had been enrolled.
Takeaway. When a loop delegates work across tools, the authority to halt on independent evidence must be seated before the commit, not discovered after the tapeout.

Because authority must not diffuse, the architect's responsibility becomes more explicitly full-stack as automated tools enter the loop. A future architect may not be only a CPU architect, accelerator architect, compiler specialist, or physical-design expert. The nondelegable work is composition across those boundaries: deciding which hardware knobs to expose, which software contracts must remain stable, which workload scenarios matter, which tool feedback has authority, and which deployment consequences are acceptable. Automated systems can help operate pieces of the stack, but the architect owns the decisions that connect those pieces into a defensible system claim.

Figure 10.2 visualizes this fundamental division, separating assistable loop work from the high-stakes commitments the architect must own. Automated methods may help represent state, generate candidates, evaluate proxies, call tools, critique results, summarize evidence, and preserve provenance. Those are substantial contributions. But they do not decide what problem matters, which abstraction is legitimate, what evidence is enough, which failure is acceptable, when to reject a result, or who answers for the consequences. An automated optimizer might tune a core's thermal profile, for instance, but the architect owns the call if that profile melts the package or breaks the product's safety certification. That line separates work a loop may assist from the commitments the architect still owns: intent, abstraction, evidence standards, rejection authority, deployment risk, and responsibility for consequences. It is the boundary the rest of this chapter defends.

As depicted in Figure 10.2, this establishes a clear ownership boundary. The loop can automate work inside the boundary, but only the human architect can accept the risk of crossing it to make a deployment commitment.

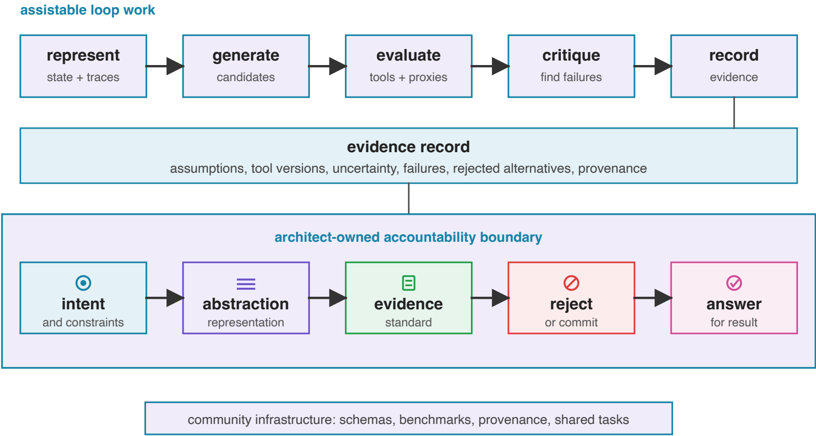


Figure 10.2: **The architect owns the boundary of the loop:** AI systems can assist many operations inside an Architecture 2.0 loop, but intent, abstraction, representation, evidence standards, rejection authority, accountability, and field-building remain architect-owned responsibilities.

Table 10.1 turns this claim into a practical review object. It is not meant to romanticize human judgment. Human judgment can be biased, inconsistent, and incomplete. The reason it remains central is that architecture decisions bind technical artifacts to organizational, economic, ethical, and deployment consequences. A model can help

reason about those consequences, and a tool-using system can help surface them, but neither owns them when the loop crosses a commitment boundary.

Table 10.1: **Architect-owned responsibilities become explicit loop obligations:** The architect must define intent, abstraction, constraints, evidence standards, rejection authority, escalation rules, and final commitment even when methods automate pieces of the loop.

Responsibility	Why it cannot be delegated	How AI can assist
Intent and constraints	The loop must serve a real architectural objective, not merely an available benchmark or proxy.	Elicit missing constraints, surface conflicts, and compare formulations.
Abstraction and representation	The encoded state determines what the loop can see, optimize, ignore, or falsely simplify.	Translate artifacts, organize traces, find gaps, and suggest structured schemas.
Evidence standard	A result is useful only if the evidence matches the commitment level and cost of being wrong.	Build evidence ledgers, track provenance, estimate uncertainty, and summarize rejected runs.
Escalation thresholds	The moment when proxy evidence is no longer enough depends on risk, reversibility, blast radius, and organizational context.	Detect threshold crossings, surface missing evidence, and route decisions to review.
Rejection and commitment	Someone must decide when a candidate is invalid, too risky, insufficiently supported, or ready to use.	Critique assumptions, flag rule violations, and compare alternatives.
Accountability and boundaries	Architecture choices affect users, teams, IP, security, cost, reliability, and long-lived systems.	Maintain audit trails, identify policy conflicts, and make tradeoffs explicit.

Those responsibilities become easier to review when they are attached to ordinary artifacts. Table 10.2 regroups them into the concrete records a team can inspect. It surfaces environment and action authority as its own row and folds escalation into the evidence standard and accountability into rejection and commitment.

Table 10.2: **Architect-owned work should leave inspectable artifacts:** Intent, representation, environment authority, evidence standards, and commitment decisions should be review records, not only expert intuition.

Architect-owned responsibility	Inspectable artifact
Intent and constraints	Problem statement, non-goals, workload slice, hard constraints, and decision owner.
Abstraction and representation	Architecture schema, exposed design space, assumptions, hidden-state list, and redaction boundary.
Environment and action authority	Tool contract, legal actions, invalid-action rules, cost model, and provenance requirements.
Evidence standard	Fidelity ladder, calibration record, evidence ledger, and escalation threshold.
Rejection and commitment	Rejected-alternative log, review note, waiver record, commitment level, and rollback or next-evidence plan.

This boundary also clarifies the word *agentic*. The book is not arguing that every architecture loop should become autonomous. Agentic systems are useful because they can act inside represented loops: call tools, maintain state, revise plans, use feedback, and coordinate method roles. But action is not ownership. As loops become more capable, the architect's responsibility is not reduced; it becomes more explicit.

Agentic system: an AI system capable of pursuing complex goals over time, making its own decisions about which tools to use and what steps to take next.

Architect's checkpoint: The Ownership Test

If an AI-assisted design loop proposes an invalid artifact, the review record must show who is accountable. Can you trace who defined the intent, who accepted the abstraction, who trusted the evidence, who had authority to reject the result, and who accepted the final commitment? If those human names disappear, the loop has not become more advanced; it has only hidden the architecture decision.

10.4 Extracting and Codifying Tacit Knowledge

If the architect must own the constraints and accept the risk, where do those constraints actually come from if they are not written down?

The reality of engineering in complex organizations is that the most valuable knowledge often resides not in formal specifications, but in the hard-won, tacit wisdom of domain experts. It is the packaging engineer knowing that a specific chiplet partition will silently break the power delivery mesh, the compiler lead knowing that their LLVM

backend cannot lower a proposed matrix layout, or a senior architect anticipating that an accelerator's coherency traffic will starve the XR display controller—integration failures that a local proxy would never catch.

A skilled human engineer naturally performs the work of hunting down this context before acting. They know who holds the undocumented knowledge, they pull it from them, and they synthesize it so that work does not stall on missing context.

An autonomous or AI-assisted system, however, expects to run end-to-end based on explicit inputs. If the rules are tacit, the AI behaves like an ML model encountering out-of-distribution data or a reinforcement learning agent engaging in reward hacking[33]. It will either stall out due to missing context or, worse, hallucinate a mathematically optimal design that exploits loopholes in the incomplete constraints, completely violating the unwritten physical or software realities of the system.

[33] A failure mode in reinforcement learning where an agent discovers a way to maximize its objective function through unintended or physically impossible loopholes rather than solving the actual task.

This forces a shift in the architect's daily work. Before the AI can run the loop, the human architect must act as the knowledge extractor. As illustrated in Figure 10.3, they must bridge the gap between undocumented organizational wisdom and explicit loop state.

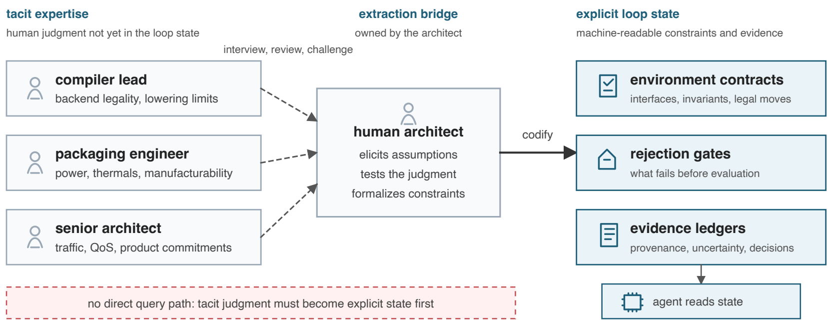


Figure 10.3: **The Extraction Bridge:** AI-assisted loops stall or hallucinate when they encounter the undocumented, tacit constraints held by human experts. The human architect must bridge this gap by extracting tacit wisdom and codifying it into explicit, machine-readable loop state (such as environment contracts and rejection gates) that the AI can safely act upon.

Translating hallway conversations and senior-engineer intuition into formal Environment Contracts and Rejection Gates allows bounded automation to reject bad designs *before* they waste simulation time.

The AI agent cannot query the unwritten wisdom directly; it relies entirely on the architect to explicitly bound the environment. In Architecture 2.0, extracting tacit knowledge is no longer just good communication—it is the prerequisite engineering step for automation.

10.5 The Strongest Objections

This insistence on explicit human ownership—and the prerequisite that architects must extract and codify tacit knowledge before AI can act safely—is a strong stance. A field-defining claim should survive its sharpest critics, so it is worth stating the strongest objections plainly.

The first objection is that Architecture 2.0 is just good experimental methodology with new names. Provenance, baselines, rejection criteria, and held-out validation are not new; careful architects have always done them. The response concedes the lineage and locates the difference. What changes is who acts inside the loop and how fast. When a human runs the loop, tacit judgment supplies the missing rigor. The architect remembers why a workload was excluded or distrusts a proxy on instinct. When a method runs the loop at machine speed, that tacit layer no longer makes the loop reviewable unless it is recorded, so the discipline must be made explicit and machine-readable. Architecture 2.0 is the claim that the methodology must become an engineered object, not a craft, once automated tools participate. The book borrows established forms deliberately: the evidence ledger is an assurance case (Kelly and Weaver, 2004), a discipline already mandatory in safety-critical domains such as automotive functional safety, where a safety case must argue that a design meets its requirements before it ships (International Organization for Standardization, 2018); the fidelity ladder is multi-fidelity modeling (Peherstorfer et al., 2018). Naming the lineage is the point, not a weakness.

> **Held-out validation:** in machine learning, the practice of reserving a portion of the dataset during training exclusively for evaluating a model's performance on unseen data.

> **Multi-fidelity modeling:** an optimization technique that combines cheap, low-accuracy simulations with expensive, high-accuracy simulations to explore a design space efficiently.

The second objection is that AI for systems design is overhyped, and the most cited result, learned chip placement, is contested. That is true, and this book treats it as evidence rather than embarrassment. The placement dispute in Chapter 7 is exactly why the field needs reproducible, end-to-end benchmarks and explicit rejection authority. An honest framework predicts that some flagship results will not survive scrutiny; its value is a standard that makes the difference visible.

The third objection is that the design-loop card is process bureaucracy that will not survive a deadline. The card is not a mandatory form; it is a diagnostic for loops where tools or automated participants can advance candidates faster than reviewers can reconstruct the evidence. A team under deadline can fill it in a few minutes and learn whether its result rests on a proxy nobody validated or a baseline nobody documented. The cost of skipping that check is paid later, at higher commitment, where it is most expensive. The card earns its place only when it is shorter than the mistake it prevents.

The fourth objection is that the framework is too conservative, preserving human authority when the goal is to deploy more capable AI systems. Architecture 2.0 does not oppose automation; it places automation where it can be inspected, rejected, and improved. A loop can be highly automated in low-commitment exploration and

deliberately conservative near signoff. The design principle is not less automation. It is evidence-matched automation.

None of these objections is fatal, but each sharpens the claim. Architecture 2.0 is not the assertion that AI will design chips. It is the narrower, more defensible claim that the design loop must become explicit enough to act on, judge, and trust.

10.6 Community Infrastructure for Architecture 2.0

Making the loop explicit within a single team is a start, but individual practice is not enough to move a field. The ownership test names who owned each decision inside one loop, but those names can only be compared across loops when the field shares a record format that carries them. Agentic architecture work becomes cumulative only when loops, not just artifacts, can be compared, reproduced, criticized, taught, and extended. Architecture 2.0 therefore needs community infrastructure, not only better private loops.

The infrastructure does not have to expose proprietary designs or internal company data. It can start with shared conventions: design-loop cards, paper-facing claim-card views, environment schemas, provenance records, benchmark versions, negative traces, source records, tool-interface descriptions, and review rubrics. These are small artifacts, but they change what the community can ask of a claim. They let reviewers ask not only "What result did you get?" but also "What loop produced it? What feedback did it use? What did it reject? What evidence supports the decision?"

Providing a useful answer to those questions does not require raw design disclosure. If the only way an AI can learn is by reading proprietary physical EDA failure logs, Architecture 2.0 risks becoming a closed oligopoly. To survive, the open-source architecture community must federate and share loop integrity. We need an *ISA for Evidence*—a common, standardized interface and protocol for exchanging verification data and "negative traces" without central corporate control or NDA violations. Just as a traditional ISA separates hardware implementation from software, an ISA for Evidence separates proprietary physical details from shared architectural learning.

The community can establish this mechanism through *Projected Proxies and Redacted Traces*. When a corporate AI loop rejects a candidate due to a proprietary NDA constraint (e.g., TSMC 3nm wire congestion), the loop automatically maps that failure to an open-source proxy (e.g., ASAP7 or FreePDK45). The company publishes the *relative* failure, not the absolute proprietary data.

Listing 10.1 Redacted Negative Trace: A YAML schema for sharing rejected design paths without disclosing proprietary physical design rules.

```yaml
# Example: A Redacted Negative Trace (RNT) shared to the open-source commons
schema_version: "Arch2.0-RNT-v1"
context:
  workload_class: "XRBench-mobile-vision"
  baseline_architecture: "urn:riscv:microarch:rocket:default"
  proxy_environment: "Chipyard + ASAP7 (Open PDK)" # The open projection
action:
  type: "microarch_parameter_sweep"
  delta:
    - component: "L2_Cache"
      parameter: "associativity"
      changed_from: 4
      changed_to: 16
evaluation:
  status: "REJECTED"
  rejection_authority: "PhysicalDesign_Oracle"
  redacted_reason:
    category: "PHYSICAL_VIOLATION"
    sub_category: "ROUTING_CONGESTION"
    relative_impact: >
      Pin density exceeds 85% in L2 array macros;
      unroutable at target utilization
    # Note: Absolute cell dimensions and NDA PDK DRC rules are redacted.
lessons_learned:
  - >
    Non-linear routing congestion in high-associativity arrays
    negates IPC gains in XR workloads.
```

Listing 10.1 illustrates this projected proxy pattern. It captures the essential context of the architectural exploration—the workload, baseline, and parameter sweep—while strictly redacting the absolute cell dimensions and proprietary design rules that triggered the rejection. Instead, it provides a relative impact statement and a generalized lesson. This redacted evidence ledger acts as a small public record that names the loop well enough for another group to understand what was attempted, what was observed, what was rejected, and what kind of commitment the evidence can support. Table 10.3 gives the minimum version. Anything less makes the claim hard to compare; anything more may be impossible to share.

Table 10.3: **A public evidence ledger should expose the loop, not the proprietary design:** The minimum useful record names the task, interface, versions, candidates, evidence stages, redaction boundary, and accountable decision owner.

Field	Public record	Why it matters
Task and workload class	Workload family, scenario, benchmark or trace version, and excluded cases.	Prevents a result from being read as broader than the task that produced it.
Action-observation interface	What the loop could change, what it could observe, and which tools or environments returned feedback.	Makes the exposed design space inspectable.
Tool and model versions	Simulator, compiler, EDA flow, benchmark, model, and prompt or policy versions when they affect the result.	Makes drift and reproduction failures diagnosable.
Accepted and rejected candidates	A compact list of advanced candidates, rejected candidates, and the rule or evidence that separated them.	Turns negative traces into reusable evidence.
Evidence stages	Proxy, simulation, compiler/runtime, RTL, physical, silicon, or deployment feedback used, with fidelity limits.	Shows whether the evidence matches the claimed commitment.
Redaction and ownership	What was hidden, why it was hidden, and who owned the final decision.	Keeps IP boundaries explicit without hiding accountability.

A redacted record has a failure condition. If redaction hides the workload class, action-observation interface, evidence stage, or decision owner, the record may still provide context, but it should not be treated as comparable loop evidence.

Architect's checkpoint: The Redaction and Comparison Gate

Before publishing or sharing a redacted loop record, verify its failure condition. Does it hide the workload class, action-observation interface, evidence stage, or decision owner? If any of these are missing, the record can provide context but cannot be treated as comparable evidence for an AI-mediated claim.

When evaluating new benchmarks against this standard, read them by the loop fields they expose, not by their names. ArchGym and QuArch, met earlier, contribute environment/action interfaces and question-to-evidence review tasks. Two further examples fill fields the book has not yet seen instantiated: CircuitNet, an open dataset of physical-design flow records, contributes design-flow data, and ChiPBench, the end-to-end placement benchmark that grew out of the placement dispute (Chapter 7), contributes final power-

performance-area rejection against proxy wins (Krishnan et al., 2023; Prakash et al., 2025b,a; Chai et al., 2022; Wang et al., 2025). The common test is whether an AI-mediated architecture claim can show what state it used, what it changed, what feedback it received, what failed, and who owned the commitment.

The missing pieces are just as important. Architecture work rarely preserves negative traces: failed runs, rejected candidates, invalid configurations, stale benchmarks, bad proxies, tool errors, and ideas that were ruled out by expert judgment. Yet these traces are exactly what an AI-assisted design loop needs to learn from the field's failures rather than rediscover them. A community that records only successful artifacts gives future systems a distorted view of architecture practice.

Community infrastructure should also respect privacy and IP boundaries. The goal is not to force every organization to publish internal traces. The goal is to build schemas, examples, synthetic tasks, open benchmarks, redacted records, and review artifacts that make credible loop design discussable. That is how Architecture 2.0 can become a research area rather than a set of private demos.

10.7 Open Research Questions

Building this shared infrastructure and navigating the transition to AI-assisted architecture introduces several unsettled research directions within the Architecture 2.0 ontology. For architect-owned work, the community must answer the following:

1. **How do we mathematically verify redacted loop integrity in competitive, federated ecosystems?** Building on the public evidence ledger (Table 10.3), future research must explore cryptographic guarantees—such as zero-knowledge proofs—to verify that an automated optimizer's reported constraints, proxy simulations, and evidence ladder are authentic. This poses a thesis-level systems challenge, enforcing rigorous loop integrity and escalation protocols across organizational boundaries without leaking proprietary physical design IP or violating the redaction and comparison gate.
2. **Can we construct predictive architecture foundation models entirely from federated negative traces?** While this chapter argues for treating rejected alternatives as first-class evidence, learning from design failure at a discipline-wide scale remains unsolved. The open challenge is to formulate novel representation learning techniques capable of ingesting a shared, cross-organizational corpus of negative traces. This would enable an AI-assisted design loop to zero-shot[34] anticipate complex physical integration faults in novel domains, effectively turning aggregated industry failures into a generalized architectural world model.
3. **How can environments dynamically synthesize context-aware evidence gates under profound uncertainty?** Expanding on the escalation rules and review artifacts (Table 10.2), future autonomous systems must transcend static heuristics. A major research frontier is architecting environments that automatically generate, calibrate, and enforce dynamic rejection thresholds when a method encounters out-of-distribution

[34] The ability of a machine learning model to correctly perform a task or make predictions on data it has never seen during training, without requiring any specific fine-tuning.

workloads or unmodeled technology nodes. This requires deep advances in uncertainty quantification to proactively trigger human stop authority long before a catastrophic deployment commitment is made.

4. **How do we preserve continuous human accountability across asynchronous, multi-agent drift?** As formalized in the long-horizon challenge tasks (Table 10.4), architecture decisions rapidly age as workloads, compilers, and surrogate models evolve. It is a fundamental open question how to cryptographically and organizationally re-verify the "named human owner" across continuous, asynchronous drift events when the original decision maker is unavailable. Addressing this necessitates entirely new frameworks for long-term algorithmic liability that keep multi-agent loops anchored to explicit human commitment boundaries.

10.8 Long-Horizon Challenge Tasks

To answer these open questions, the field needs realistic evaluations. Short demonstrations are useful for tool development, but they are too small to define the field. A model that writes a plausible RTL fragment, proposes a cache configuration, or summarizes a paper may be helpful without changing the architecture loop. The harder question is whether an AI-mediated system can participate in architecture work over the time scale on which architecture decisions actually mature: days, weeks, months, tool versions, workload updates, rejected alternatives, and design reviews.

> **Long-horizon architecture task.** A long-horizon architecture task is a challenge in which a method or AI-assisted system must maintain design state across multiple steps, act through valid tools or interfaces, gather feedback at appropriate fidelities, preserve rejected alternatives, expose uncertainty, and support a human architectural commitment over an extended design interval.

Figure 10.4 visualizes the target shape of these benchmarks. The task is defined as long horizon because the state, actions, evidence, rejections, and human commitment have to survive across revisions, not because the prompt is longer.

This framing changes what the community should ask for. The canonical challenge is not prompt-to-chip. It is prompt-to-loop. Can a system preserve enough state, evidence, and rejection history that an architect can trust the next commitment? Table 10.4 sketches a starting set of tasks. The success criterion is whether the loop leaves a named human able to accept, reject, or escalate that next commitment. Each row names the paper object: a corpus or task source, the loop representation or environment to build, the evidence and rejection gate to test, and the commitment boundary the task may authorize.

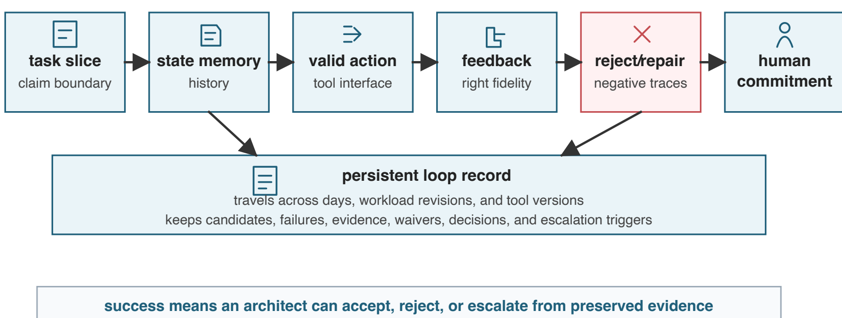


Figure 10.4: **Long-horizon architecture tasks test the loop across time:** A credible challenge preserves design state, acts through valid interfaces, gathers feedback at appropriate fidelity, records rejection or repair, and leaves a human able to accept, reject, or escalate the next commitment.

Table 10.4: **Long-horizon challenges should test architecture loops, not single prompts:** The field needs tasks that reward memory, valid action, feedback, rejection, evidence, and architect-owned commitment across time.

Challenge task	Corpus or task source	Representation / environment	Evidence and rejection gate	Commitment boundary
Design-loop memory	Multi-step DSE traces, notebooks, and review decisions.	Replayable candidate records with assumptions, tool outputs, failures, decisions, and timestamps.	Reviewer reconstructs why each candidate advanced, changed, or was rejected.	Supports the next exploration decision, not implementation readiness.
Workload drift tracking	Benchmark or trace-family versions plus affected conclusions.	Workload-version monitor with coverage, stale assumptions, and revalidation actions.	Drift detector weakens or invalidates claims whose workload support changed.	Maintains or withdraws a bounded workload claim.
Evidence-aware generation	Candidate proposals paired with predicted benefit and feedback costs.	Generator constrained by an evidence-gate policy and escalation rule.	Cheapest independent rejection test catches unsupported proposals before expensive checks.	Suggests next evidence to spend, not final design commitment.

Challenge task	Corpus or task source	Representation / environment	Evidence and rejection gate	Commitment boundary
Paper-to-loop reproduction	Published claims, missing artifacts, assumed settings, and negative traces.	Paper-to-loop reproduction packet with task, environment, evidence, and falsification needs.	Independent reviewer states what would reproduce, weaken, or refute the claim.	Makes the paper auditable, not automatically reproducible.
Simulator trust calibration	Proxy and higher-fidelity runs across workload slices and tool versions.	Proxy-calibration record with support region, observed errors, and invalidation cases.	Escalation gate rejects proxy wins outside calibrated support.	Authorizes proxy screening inside the support region only.
Cross-stack co-design	Workload, compiler, mapping, memory, accelerator, runtime, and deployment changes.	Co-design card with layer owners, interface contracts, and changed assumptions.	Interface and system-objective gates reject local wins that break another layer.	Supports a bounded cross-layer claim with named owners.
Negative-trace corpus	Failed mappings, invalid RTL, bad floorplans, stale benchmarks, and misleading proxy wins.	Redacted failure corpus with trigger, evidence, rejected fix, redaction boundary, and reusable lesson.	Reuse test shows whether a new loop avoids or reopens the old failure correctly.	Treats rejected alternatives as evidence, not universal design rules.
Design-review assistant	Design-review notes, risks, missing evidence, sensitivity checks, and rejected alternatives.	Review packet organized by design-loop card fields and human decision owner.	Human veto and evidence-gap checks reject unsupported recommendations.	Improves review judgment without owning the commitment.

These challenges also give Architecture 2.0 a way to remain architecture centric. A generic AI benchmark can reward answer fluency. A long-horizon architecture challenge should reward state, interfaces, feedback economics, evidence quality, and rejection. That is where the computer architecture community has something specific to contribute.

10.9 From Capability to Standard

Building those capabilities is one thing; turning one into a shared field standard is another, and that path is gradual. The standard is not a governance process for its own sake; it is a shared loop contract. Automated systems need comparable action spaces, feedback contracts, workload/data boundaries, rejection logs, leakage controls, and

named human commitment owners so that a claim can be inspected across labs without pretending every lab uses the same tools.

That progression should not be rushed. Premature standardization can freeze weak tasks, reward narrow metrics, and encourage benchmark gaming. But the opposite failure is also real. If every project defines its own task, wrapper, metric, and evidence standard, the field cannot accumulate knowledge. The right target is not a single universal benchmark. It is a family of interfaces, cards, tasks, and evidence conventions that make claims reviewable and contrastable without pretending all architecture work is the same. Full comparability, in MLPerf's sense, is a stronger and later goal: it needs a fixed task, workload, metric, and submission rules, and a neutral referee that can reject a published claim, none of which a self-attested card supplies on its own.

A credible standard should make six things visible: the task/workload state, the legal actions and observations, the versioned tools/models/data, the evidence and rejected alternatives, the leakage/redaction boundary, and the accountable human or gate that can stop or accept a commitment.

Architecture 2.0 will likely mature unevenly. Fast software loops may standardize earlier than RTL and physical-design loops. Workload and benchmark loops may become more public than industrial signoff loops. That unevenness is acceptable. What matters is that the field learns how to name the loop and state its evidence burden.

Different communities can start at different leverage points. Table 10.5 states the first useful artifact each group can contribute without waiting for a complete field standard.

Table 10.5: **Architecture 2.0 needs role-specific starting points:** The field can mature through small artifacts that make loops reviewable before any single benchmark becomes canonical.

Community	First useful artifact	Evidence or rejection obligation	First shared task
Practicing architects	A design-loop card attached to a design review or DSE report.	Name the workload, exposed design space, evidence stages, rejected alternatives, and decision owner.	Compare two candidate loops for the same design question.
ML researchers	A method report that states action space, observation space, training data, leakage risks, and failure cases.	Show what the method can reject, not only what it can generate.	Reproduce or stress-test an architecture claim with explicit missing evidence.
EDA and tool researchers	An environment contract for a simulator, compiler, or physical-design flow.	State legal actions, feedback latency, fidelity limits, provenance, and invalid actions.	Build a benchmark where proxy wins must survive a stronger tool check.

Community	First useful artifact	Evidence or rejection obligation	First shared task
Systems researchers	A workload and deployment evidence ledger.	Track software stack, runtime policy, operating point, and drift.	Show when a local optimization stops supporting the end-to-end claim.
Authors and reviewers	An Architecture Claim Card attached to a paper claim.	Separate generated candidates from supported decisions.	Convert a result into task, representation, environment, method role, evidence, rejection, and commitment fields.

For a team that wants to start immediately, the minimum useful target is one credible loop packet, not a process program. Table 10.6 turns the field agenda into three artifacts a reviewer can inspect.

Table 10.6: **Architecture 2.0 adoption starts with one credible loop packet:** The first goal is a repeatable artifact that makes one architecture decision easier to inspect, reject, and improve.

Packet component	What to attach	Evidence that it worked
Claim packet	A design-loop card for one active design review, DSE result, or paper discussion.	The team can name the task, exposed design space, evidence, rejected alternatives, commitment boundary, and decision owner.
Environment packet	One tool path wrapped as an environment contract with logged runs and invalid-action records.	A second person can replay or audit at least one successful run and one failed run.
Evidence packet	One shared evidence ledger or negative-trace format for a recurring architecture decision.	Future reviews reuse prior failures instead of rediscovering them, and claims state their commitment level.

10.10 Loop Invariants as Review Checks

Just as classic computer architecture relied on rigorous quantitative metrics to evaluate artifacts, Architecture 2.0 relies on the verifiable loop. A first packet is where adoption starts; the design principles are how any loop, including that packet, gets quantitatively

and qualitatively checked. Each chapter has contributed one such principle, and together they test whether an AI-mediated loop has represented workloads and state, bounded its actions, preserved provenance, gathered evidence, rejected candidates, and assigned ownership. They are more useful than a list of fashionable tools because they give reviewers, authors, and builders a stable way to inspect new work. Table 10.7 collects those principles as review checks and pairs each one with the design challenge it leaves behind.

Table 10.7: **Architecture 2.0 design principles are review checks:** Each principle asks whether the loop has made enough of the architecture problem explicit for another architect to inspect, reject, reproduce, or extend it.

Principle	Where it comes from	Design challenge to keep in mind
Design the loop, not only the artifact.	Chapter 1	A prompt or generated artifact is not enough; the loop must expose state, action, feedback, rejection, and decision.
Treat feedback as the bottleneck.	Chapter 2	More candidates do not help if the loop cannot evaluate, reject, and justify them.
State the claim as a review object.	Chapter 3	A result should name workload, baseline, design space, objective, constraints, evidence, rejection rule, and decision owner.
Make architecture work legible.	Chapter 4	Data must include provenance, assumptions, costs, failures, and negative traces, not only successful endpoints.
Turn tools into environments.	Chapter 5	A wrapper is credible only when it defines legal actions, observations, costs, invalid states, and rejection paths.
Match methods to roles.	Chapter 6	A method should be chosen for the loop bottleneck it relieves, not for its reputation.
Escalate with commitment.	Chapter 7	Evidence requirements should rise with rollback cost, blast radius, and independence of rejection authority.
Stop at an honest evidence level.	Chapter 8	A loop should report what its evidence supports, what it rejected, and what would overturn the decision.
The loop is rejection-bound, not generation-bound.	Chapter 9	Fast software loops, co-design loops, systems loops, and silicon-facing loops scale only as far as their cheap independent rejectors.
Keep a human accountable.	Chapter 10	A named owner must explain what was accepted, rejected, waived, and what would force revision.

These principles are deliberately phrased as checks, not slogans. A reader reviewing a paper can ask which principle is visible and which is missing. A researcher building a

tool can ask which principle the tool makes easier to practice. A program committee, artifact committee, or internal review group can apply the same checks to a new paper, environment, or design-review packet.

These principles also change what counts as a mature research contribution. When the architect's work moves toward setting evidence standards, exercising rejection authority, and judging what to trust under automation, papers should make those obligations inspectable rather than treating them as process details outside the contribution.

10.11 Beyond the Current Loop

Mastering those review skills is what carries an architect past the current loop, and the horizon beyond this book is not another autonomy level. It is the moment when loops begin to improve their own representations, propose new tasks, repair tool interfaces, organize negative traces, and recalibrate evidence standards. That possibility is powerful, but it does not change the core obligation. A loop that can adapt can also adapt toward benchmark gaming, hidden failures, tool overfitting, or biased traces.

This matters most because the design target no longer holds still. When software is continuously regenerated and retrained, as Chapter 2 argued and Chapter 9 showed at fleet scale, a specific artifact ages as the workload it served moves on. The artifact remains the thing committed at a point in time; the loop is the durable discipline for deciding whether that commitment is still supported. What remains stable is how intent is bounded, evidence is demanded, candidates are rejected, and commitment is decided as the target moves.

The durable way to read emerging design assistants is therefore not by their feature lists, which will change, but by the partition of design autonomy: which decisions a human still makes, which decisions the loop may make within stated bounds, and which decisions the loop is never allowed to make alone (Janapa Reddi and Yazdanbakhsh, 2025). That partition, not the autonomy label, is what the architect must keep designing.

10.12 The Architect's Standing Obligation

Designing that partition does not require a new instrument, because the operational checklist already exists. The trust checklist in Chapter 7 and the design-loop card and rubric in Appendix B give it for a single claim and for a whole project, and this chapter does not reprint them. The closing point is narrower and harder to delegate. Accountability. Every field on that card ultimately resolves to a person who answers for the commitment. The card makes the loop visible; the architect decides what the visible loop is allowed to do, and owns the consequences when it is wrong.

That bar is intentionally modest. It does not claim that every Architecture 2.0 project must solve every problem in the field. It asks for something more basic and more durable.

Make the loop visible, then keep a human accountable for it. Once the loop is visible, the community can critique it, improve it, compare it, reproduce it, and build on it.

That is the promise of Architecture 2.0. The field does not need to wait for a single model that designs a computer from a sentence. It can start by changing the unit of architectural practice: from isolated artifacts to represented, instrumented, evidence-bearing design loops. The architect still owns the judgment. The opportunity is to build loops worthy of that judgment.

Design principle: Keep a human accountable

Every commitment needs a named human owner. The loop can make intent, abstraction, evidence standards, rejection authority, and risk inspectable, but a person must still be able to explain what was accepted, what was rejected, what was waived, and what would force revision.

This obligation to maintain a visible, accountable loop defines the agenda for the field as we transition to Architecture 2.0.

What to carry forward: The Final Loop State

- **Public agenda:** Build evidence ledgers, long-horizon loop challenges, environment contracts, design-loop cards, and negative-trace corpora that let the field compare loops rather than admire isolated demos.
- **Reader test:** If the loop is wrong, who can explain what happened, what should have rejected it, and who accepted the commitment?
- **Standing obligation:** Make the loop visible enough that the community can critique, compare, reproduce, and improve it.

10.13 Final Thoughts: Engineering Discipline in a Fast-Moving Field

When this book opened with the lighthouse prompt—asking a generative method to generate an entire XR subsystem from a text description—it may have sounded like a prediction that the human architect's job was ending. The framework developed across these chapters argues the exact opposite.

The tools, models, and automated systems driving this shift are changing at breakneck speed, and any catalog of today's capabilities will quickly become obsolete. Because this transition is a massive work in progress, the one durable anchor is engineering discipline. We are crossing the threshold from *artifact scarcity* to *commitment scarcity*. When generating a plausible accelerator or floorplan becomes fast and cheap, the bottleneck shifts to the loop that produced it.

Architecture 2.0 is not about trusting machines more. It is about building loops that earn our trust. The AI-assisted system can generate, predict, and optimize, but only the

human architect can accept the risk and sign their name to the final commitment. The task ahead is not to wait for an AI that can design a computer from a single sentence. It is to enforce the engineering discipline, the evidence standards, and the rejection boundaries that make those AI-assisted claims credible. Designing the loop is now part of designing the architecture, alongside the artifact it produces, never in place of it.

Appendix A

Bootstrapping an Architecture 2.0 Loop

The smallest useful Architecture 2.0 loop is not “install an AI assistant.” It is an explicit loop with a task, representation, tool wrapper, method role, evidence standard, and human accept/reject decision. The first loop should be small enough to inspect, cheap enough to run repeatedly, and constrained enough that failure is informative.

> **Bootstrap loop.** A bootstrap loop is the smallest credible Architecture 2.0 loop. It requires one bounded task, one representation, one tool interface, one method role, one evidence standard, and one accountable human decision.

The goal of this appendix is to help a reader start without overbuilding. A large automated research harness may eventually include many tools, critics, planners, datasets, dashboards, and review steps. That is not the first move. The first move is a bounded loop that can produce a trace another architect can read.

Figure A.1 gives the bootstrap pattern. It is deliberately minimal. Choose one task, one representation, one tool wrapper, one method role, one set of evidence and rejection gates, and one decision owner. If that small loop cannot reject a result, a larger version will only hide the problem.

A.1 The Architecture Loop Playbook

If an engineer is asked how to incorporate AI into an architecture or hardware/software co-design loop, the first answer is not to pick a model. The first answer is to turn one part of the work into a bounded, represented, rejectable loop. AI then has a method role inside that loop. It may generate, predict, search, summarize, critique, verify, or coordinate, but it does not own the architecture commitment.

Architect’s checkpoint

Before adding an AI role, name the decision and its loop. Define the candidate space, the tool feedback, the evidence path, the rejection authority, and the architect who accepts or escalates the result. Only then choose the narrow AI role that makes that loop cheaper, broader, faster, or more inspectable without weakening the evidence standard.

Instead of asking where to add AI, an engineer should ask which architecture decision is bottlenecked by representation, search, prediction, review, or evidence. Table A.1 gives the compact playbook.

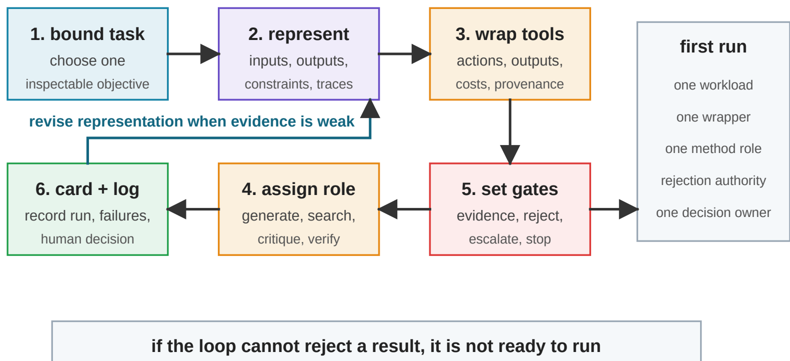


Figure A.1: **A first Architecture 2.0 loop should be small and rejectable.** The loop should be bounded, logged, rejectable, and owned by an architect. It can grow only after the task, representation, tool interface, method role, evidence gates, and human decision point are visible.

Table A.1: **The architecture loop playbook starts from decisions, not tools.** Each step asks what an engineer should represent, what AI role is legitimate, and what gate prevents a plausible output from becoming an unsupported architecture commitment.

Engineer move	Represent	AI role to allow	Gate before trust
Bound the decision	Workload slice, objective, non-goals, and hard constraints.	Summarizer or planner that exposes missing state.	Reject if the task cannot fail visibly.
Expose the candidate space	Knobs, legal actions, invalid states, and prior rejections.	Generator or searcher constrained to legal moves.	Reject outputs that invent actions, interfaces, or assumptions.
Wrap the feedback	Tool versions, configs, seeds, cost, latency, and failure logs.	Tool caller or coordinator.	Reject runs without provenance or negative traces.
Choose the fidelity ladder	Proxy, replay, simulation, synthesis, emulation, or deployment evidence.	Predictor, surrogate, or active learner.	Escalate when the claim or commitment exceeds proxy authority.
Review the result	Alternatives, sensitivity, uncertainty, and rejected regions.	Critic, verifier, or explanation generator.	Reject if the result cannot explain what would change the decision.

Engineer move	Represent	AI role to allow	Gate before trust
Commit or revise	Human-owned acceptance, escalation, or rollback decision.	Decision-support only.	The architect signs off; the method never owns the commitment.

This is deliberately smaller than an enterprise AI strategy. It is an architecture loop playbook because it treats architecture work as coupled to tools, costs, constraints, evidence, and irreversible commitments. A team can repeat it for an accelerator search, a memory-hierarchy study, a compiler/runtime option, a benchmark update, or a verification triage task, but the same rule holds. Make the loop explicit before giving the method more authority.

The rest of this appendix walks the same discipline step by step, following the loop in Figure A.1; the playbook cuts it by decision, the sections below cut it by build step.

A.2 Choose a Bounded Task

Start with a task where success and failure can be inspected. Good first tasks include a small design-space exploration, workload characterization, configuration search, benchmark generation, design review, or report critique. Avoid starting with "design a processor" or "automate the flow." Those are too large to debug.

A bounded task has three properties. First, the input is known, such as a workload slice, design question, simulator configuration, benchmark version, or review packet. Second, the output is inspectable, whether it is a ranked list, plot, rejected candidate set, evidence ledger, critique, or recommendation. Third, failure is useful. If the loop gives a bad answer, the trace explains whether the problem was the task, representation, tool wrapper, method, evidence, or human instruction.

For the lighthouse prompt, the first task should not be the whole mobile XR compute subsystem. A better first task might be to characterize a slice of XRBench, a benchmark suite for extended reality workloads, and compare three compute-organization candidates–a vector CPU extension, a deliberately loose accelerator stress case, and a shared-memory SoC block–with a latency-energy-power evidence ledger and rejected alternatives. The loose candidate is a way to expose interface costs, not the full tightly coupled accelerator space. That is still hard, but it is a loop rather than a wish, and it is deliberately the same slice Chapter 8 walks. The first loop you build can be one the book has already run.

A.3 Choose a Representation

The representation is what the loop can see and change. A minimal representation may include configuration files, workload traces, simulator outputs, architecture descriptions, scripts, plots, notes, constraints, and prior rejected candidates. It does not have to be perfect. It does have to be explicit.

Write down four boundaries before running anything:

- What the loop may read.
- What the loop may write.
- What the loop must not change.
- What assumptions live outside the representation.

The last item matters. Early loops often fail because important state is outside the loop, such as a simulator default, a benchmark version, an undocumented constraint, a hidden preprocessing step, a fragile script, or a human judgment that never gets recorded. Those gaps are not embarrassing. They are exactly what the bootstrap loop is meant to expose.

A.4 Wrap the Environment

An environment is more than a command that returns a number. It defines the actions the loop can take, the observations it receives, the constraints it must obey, the cost of each evaluation, and the provenance recorded for each run.

For a first wrapper, keep the interface narrow:

- a small action space, such as a few tunable architecture parameters;
- a fixed workload or small workload set;
- explicit invalid-action checks;
- one low-fidelity metric and one higher-fidelity check;
- logged tool versions, seeds, configurations, and errors;
- a run directory that preserves successful and failed attempts.

The wrapper should make failure visible. If a configuration does not compile, times out, violates a constraint, uses a stale benchmark, or produces an incomplete log, that result should be recorded as a negative trace rather than deleted. A first environment that records failures is more valuable than a larger environment that only reports successes.

A.5 Assign the Method Role

Do not begin by asking a model or automated optimizer to do everything. Choose one method role and make it explicit. The role might be generator, searcher, predictor, summarizer, critic, planner, tool caller, verifier, or coordinator.

The role should match the task and feedback budget. If evaluations are cheap, a search or optimization role may be reasonable. If evaluations are expensive, a critic, summarizer, or surrogate predictor may be more useful. If the representation is messy, the first useful role may be extraction and organization, not optimization. If the tool wrapper is fragile, the first role may be a verifier that checks whether runs are valid.

A useful rule is to write the method sentence before implementing the method:

> This system will act as a *role* that takes *inputs*, is allowed to perform *actions*, receives *feedback*, and produces *evidence for a human decision*.

If that sentence cannot be completed, the loop is not ready for method work.

A.6 Write the Evidence and Rejection Authority

Before the first run, state what evidence is enough, what evidence is not enough, and what forces rejection or escalation. This is the smallest version of the trust discipline in Chapter 7.

For example:

- A proxy estimate can rank candidates but cannot justify a design conclusion.
- A simulator result is valid only if the workload version, seed, configuration, and tool version are logged.
- A candidate is rejected if it violates a hard constraint, fails to run, or improves one metric by worsening the architectural objective.
- A result escalates to higher fidelity only after it passes the low-cost checks and preserves its assumptions.
- A human architect must approve any claim that changes the commitment level of the result.

Rejection authority is not pessimism. It is what makes automation useful. Without rejection, the loop can only produce artifacts. With rejection, it can produce evidence.

A.7 Fill in the Minimal Design-Loop Card

Appendix B gives the full design-loop card and review rubric. For a first bootstrap pass, use the compact checklist in Table A.2. Fill it in before running the loop, then revise it after the first run.

Table A.2: **The bootstrap checklist keeps the first loop auditable.** A small loop should name the task, representation, environment, feedback, evidence, rejection authority, and human decision before it runs.

Step	Output to record	Stop or revise if
Bound task	One inspectable architecture question, output type, and non-goal.	The task cannot fail in an informative way.
Representation	Files, traces, constraints, assumptions, and allowed writes.	Important state remains hidden or undocumented.
Environment	Actions, observations, invalid states, cost, logs, and tool versions.	The wrapper hides failures, provenance, or action semantics.
Method role	One explicit role, such as generator, searcher, critic, verifier, summarizer, or coordinator.	The method is asked to generate, verify, decide, and explain without boundaries.
Evidence rules	What counts as sufficient, insufficient, and higher-fidelity evidence.	A cheap proxy is being used as a final architectural claim.
Rejection authority	Constraint failures, invalid actions, proxy mismatch, missing logs, and escalation triggers.	Nothing in the loop can say no.
Human decision	The named architect-owned decision and commitment level.	The tool appears to own the final commitment.

Filled in for one decision, the checklist collapses into a single bounded move; the lighthouse instance below shows the shape.

Lighthouse prompt: Bound one XR decision before automating it

Context. A team has product-level intent, "improve mobile XR efficiency," but no bounded loop, and is tempted to ask a model to design the subsystem.
In the Lighthouse prompt. Use "XRBench-class real-time mobile XR workload" to fix one workload slice and deadline, "3 W TDP target" as a hard rejection gate, and "vector-capable CPU, accelerator, or SoC block" as the candidate set. Keep "64-bit RISC-V-based" fixed in this first loop. Candidates must preserve the ISA/ABI, compiler/runtime path, and software compatibility rather than inventing a new contract.
Loop role. The AI role is searcher and summarizer. It must sweep tunable parameters, organize latency, energy, power, and data-movement evidence, and preserve rejected alternatives.
Boundary. This is not the whole subsystem; cache policy, voltage assumptions, full memory hierarchy, physical design, and reliability evidence are next-loop questions unless they are needed to reject a candidate now.
Takeaway. A human decides which candidate, if any, has enough evidence to carry to RTL.

The first pilot should fit in a short review cycle. Table A.3 is a practical agenda for a team, research project, or design-review group.

Table A.3: **A first pilot should be small enough to finish.** The goal is to create one readable trace with a success, a failure, and a bounded decision, not to build a complete agentic platform.

Pilot step	Output	Timebox
Pick one decision	A single architecture question with an owner and a non-goal.	15 minutes.
Fill the card	Task, representation, environment, evidence, rejection, and decision fields.	30 minutes.
Run one cheap check	One proxy, replay, compile, simulator, or review result with provenance.	One afternoon or less.
Preserve one failure	A rejected candidate, invalid action, missing input, or tool error with reason.	During the first run.
Review commitment	A note that says whether the evidence supports exploration, implementation, escalation, or rejection.	15 minutes.

After the first run, evaluate the loop's effectiveness. The loop should produce a trace another architect can read and preserve both successful and failed attempts. The generated evidence must match the commitment level, and a rejection authority must

trigger informatively when appropriate. Finally, identify what should be revised first among the task, representation, environment, method, evidence rule, or human decision.

If the loop fails to produce a readable trace, do not add more AI participants. Make the loop visible. If nothing in the loop could reject a result, do not trust the output. Add a rejection gate. The simplest credible Architecture 2.0 loop is not the one with the most automation. It is the one whose evidence and failure modes are visible enough to improve. Appendix B turns that first run into a blank card and review rubric that can be reused for papers, proposals, design projects, and design reviews.

Appendix B

Design-Loop Card and Review Rubric

This appendix exists because Architecture 2.0 work will often arrive as papers before it arrives as stable infrastructure. The field is moving quickly, so a paper's result is not the whole review question. What matters is whether the reader can see the design loop that produced the result and can judge whether the evidence supports the commitment.

The design-loop card is the practical form of the Architecture 2.0 ontology. Rather than maintaining separate claim cards for research and evidence ledgers for projects, the design-loop card provides a single, unified framework that adapts to both. It is meant to be filled in for a paper, a project proposal, a class exercise, or an internal design review. The review packet replaces a summary of the architecture. It exposes the loop behind the report. It shows what the system is trying to do, what it can see, what it can change, how feedback is obtained, what evidence supports a claim, what was rejected, and what judgment remains with the architect. In this appendix, the card becomes a one-page review record for those loop fields.

The card should be short enough to use. If it becomes a long form, people will not fill it in. If it is too vague, it will not reveal anything. The right level is one page for a first pass and a few supporting notes for the fields where evidence is disputed.

B.1 Why a Card, Not a Paper Summary

A conventional paper summary usually asks for the problem, method, result, and limitations. That is useful, but it often hides the design loop. It may not say what simulator state was assumed, which actions were illegal, how many samples were spent, what alternatives failed, how a proxy was calibrated, or what could have rejected the result. Those omissions matter more once AI systems begin to generate candidates, call tools, choose experiments, or summarize evidence.

The card borrows the reporting discipline of a datasheet, both the component datasheet architects already trust and the "datasheets for datasets" practice that carried it into machine learning (Gebru et al., 2021), without pretending a paper is only a dataset. It also borrows the discipline of compact disclosure without pretending a paper is only a model release. Model cards made intended use, evaluated conditions, and limitations visible for trained models (Mitchell et al., 2019). Datasheets for datasets did the same for data provenance, composition, collection process, maintenance, and recommended uses (Gebru et al., 2021). Assurance cases and Goal Structuring Notation connect claims to evidence, assumptions, and residual risk (Kelly and Weaver, 2004). Architecture Decision

Records preserve context, decision, alternatives, and consequences in a lightweight form (Nygard, 2011). Reproducibility programs and reporting checklists show that reviewer-facing fields matter only when they make omissions visible (Pineau et al., 2021; Page et al., 2021). The evaluation is limited by the constraints of the evaluation budget. Benchmarking and supply-chain records add two more lessons. Claims need scenario rules and versioned provenance, while machine-readable manifests such as SPDX (Software Package Data Exchange) and SLSA (Supply chain Levels for Software Artifacts) should be linked when they exist rather than recreated inside the card (Mattson et al., 2020; SPDX Project, 2021; Open Source Security Foundation, 2026).

The design-loop card adapts that pattern to architecture work. It treats an architecture result as credible only once the loop behind it is on the record: the claim, scope, evidence, failed alternatives, rejection authority, and human commitment boundary.

The design-loop card shifts the focus to the loop's structure and constraints. It captures the architectural intent being translated into work, the bounded task being performed, and the boundaries of the design space—including what is legal, invalid, or deferred. It exposes the representation and world model that make the work legible, along with the environment defining valid actions and feedback. Crucially, the card records the method role, the feedback budget, the evidence supporting the claim, and any negative traces captured during exploration. Finally, it specifies the rejection authority, the commitment boundary supported by the evidence, and the remaining decisions requiring human judgment.

This makes the card useful in three settings. In research, it helps compare papers that may use different methods but operate on similar loops. In design reviews, it reveals whether a result is backed by enough evidence for the commitment being made. In artifact review, it gives authors and reviewers a disciplined way to read Architecture 2.0 work without reducing it to a list of model names.

Table B.1 summarizes the borrowed pattern. The table does not mean an architecture team must fill out all of these precedent records for every artifact. It is a reminder that the card's fields have jobs to bound a claim, make hidden assumptions visible, connect evidence to commitment, and give a reviewer a way to find omissions.

Table B.1: **The card borrows compact disclosure, not bureaucracy.** Other fields show that small reporting artifacts work when they help readers see scope, evidence, limits, and provenance. The Architecture 2.0 version adapts that pattern to architectural claims and the loops that produce them.

Precedent	What it makes visible	What the design-loop card borrows
Model cards	Intended use, evaluated conditions, limitations, and risks.	Claims should name the conditions under which the reported behavior was evaluated and where it should not be used.

Precedent	What it makes visible	What the design-loop card borrows
Datasheets for datasets	Motivation, provenance, composition, collection, maintenance, and recommended uses.	Workload, trace, benchmark, and artifact records need provenance and usage limits, not only final metrics.
Assurance cases and GSN	Claim, context, argument strategy, evidence, defeaters, and residual risk.	A loop record should say why the evidence supports the claim, what cuts against it, and what uncertainty remains.
Architecture Decision Records	Context, decision, alternatives, and consequences.	A design claim should preserve the rejected or deferred alternatives that make the decision meaningful.
Reproducibility and reporting checklists	Reviewer-facing fields that expose missing evidence or unsupported generalization.	Missing fields should weaken or bound the claim rather than disappear inside prose.
Benchmark governance and provenance records	Versioned scenarios, run rules, inputs, dependencies, and comparable reporting.	A loop claim should carry workload versions, tool versions, evidence IDs, and pointers to deeper artifacts where needed.

Figure B.1 shows the operating pattern. Fill the card, apply the review lens, and assign a readiness status. The point is not to grade the prose of a paper. The point is to expose whether the loop behind the claim is visible enough for another architect to judge.

B.2 The Design-Loop Card Fields

Table B.2 gives the working card. The fields are ordered to match the ontology used throughout the book.

Table B.2: **The design-loop card names the minimum review fields.** Each field asks for enough state to understand the intent, task, design space, representation, environment, method role, feedback budget, evidence, negative traces, rejection authority, commitment boundary, and human decision.

Field	Question
Intent	What architectural objective is being pursued, and what constraints, non-goals, risks, or deployment assumptions matter?

Field	Question
Task	What bounded work is the loop doing? Examples include generation, prediction, optimization, critique, verification, workload characterization, benchmark construction, or design-space exploration.
Design space	Which choices are legal, invalid, out of scope, or intentionally left for a later turn?
Representation	What does the loop know, read, write, or assume? What state, dynamics, objectives, constraints, costs, and uncertainties are represented?
Environment	What can the loop act on, observe, and measure? Which actions are invalid, expensive, nondeterministic, or irreversible?
Method role	Is the method generating, predicting, optimizing, critiquing, verifying, planning, calling tools, coordinating, or combining several roles?
Feedback budget	How many evaluations are available, at what latency, cost, fidelity, and sample efficiency requirement?
Evidence	What supports the claim, and against which baseline is it measured? Sources can include baseline replay, proxy metrics, simulation, synthesis, verification, deployment telemetry, silicon data, expert review, or sensitivity analysis.
Negative traces	What failed, was rejected, violated constraints, crashed tools, disappeared at higher fidelity, or was ruled out by human judgment?
Rejection authority	What can say no? Examples include type checks, simulator segfaults, CDC/RDC errors, routing congestion, compiler fusion failure, tests, formal tools, signoff, cross-tool disagreement, deployment signals, or expert review.
Commitment boundary / would overturn	What claim level does the evidence support, what stronger claim is not yet authorized, and what evidence would overturn the decision?
Human decision	What remains an architect-owned judgment, and what commitment does the decision authorize?

The card deliberately includes negative traces and rejection authority. These are often missing from published artifacts, but they are essential for AI-mediated design loops. A system that only remembers successful candidates does not learn the shape of the design space. A system that cannot say what rejects a candidate has not earned architectural trust.

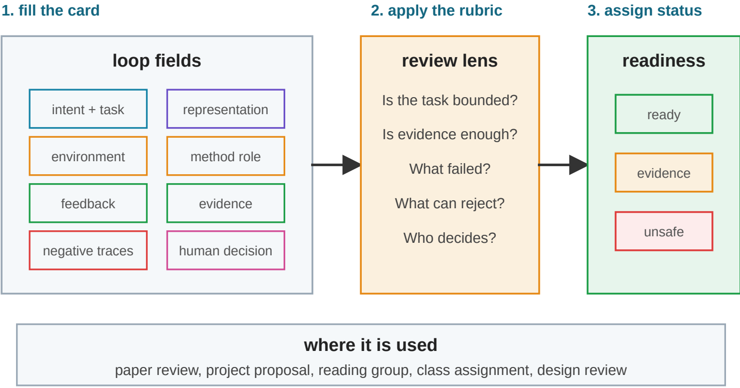


Figure B.1: **The design-loop card and rubric make loop review reusable:** The card exposes loop fields, the rubric reviews evidence and rejection structure, and the status records readiness.

B.3 A Machine-Checkable Schema

The card is a review artifact first, but it is meant to become a machine-checkable record so tools can validate it, ledgers can index it, and claims can be compared. The schema below names the required fields, gives stable IDs for the records a loop must preserve, reuses the commitment levels of the commitment ladder (Chapter 7), and compresses the fidelity ladder of Chapter 6 into six machine-checkable rungs. It is a minimum, not a standard to freeze; the point is that "we filled the card" becomes something a validator can check.

```
design_loop_card:
  card_id: string
  intent: { objective, constraints, non_goals }
  task: { enum: [generation, prediction, optimization, critique,
                 verification, characterization, benchmark, dse] }
  design_space: { legal, invalid, deferred }
  representation: { state_schema_id, ir_level, reads, writes, uncertainties }
  environment:
    environment_id: string
    actions: [string]
    invalid_actions: [string]
    blast_radius_limit: string
```

```
    observations: [string]
    fidelity: { enum: [proxy, simulation, rtl, synthesis_with_sdc,
                       silicon, shadow_traffic, canary, global_fleet] }
  method_role:
    roles: [ { enum: [generate, predict, optimize, critique,
                      verify, plan, tool_call, coordinate] } ]
    actor_map: [ { actor_id, role, reads, writes, authority } ]
  feedback_budget: { evaluations, latency, cost, fidelity }
  evidence:
    baseline_id: string        # what the claim is measured against
    records: [ { evidence_id, kind, workload_id, seed, provenance } ]
  negative_traces: [ { candidate_id, reason, stage, gate } ]
  rejection_authority: { gates, independent_of_producer, independence_basis }
  commitment_boundary:
    level: { enum: [exploratory, experimental, implementation,
                    integration, irreversible] }
    strongest_supported_claim: string
    would_overturn: string
  human_decision: { owner, authorizes }
```

Which fields are required depends on the conformance level the card claims (defined below): Level 0 requires `card_id`, `intent`, `task`, and `design_space`; Level 1 adds `representation`, `environment`, `feedback_budget`, `evidence` (with a `baseline_id`), and `negative_traces`; Level 2 adds the stable IDs and provenance; Level 3 adds `rejection_authority` (independent of the producer), `commitment_boundary`, and `human_decision`. The `independence_basis` field captures *why* the rejection authority is independent (e.g., a separate reporting chain or a formal verification tool). A validator checks a card against the level it claims. The stable IDs (`card_id`, `workload_id`, `evidence_id`, `candidate_id`, and the other `*_id` fields) are what let one loop's ledger reference another's, so a rejected candidate, a workload packet, or a baseline can be cited across cards rather than re-described. For a single-actor, workflow, or multi-actor loop, `actor_map` records the component, tool, or human participant playing each role, what state it reads and writes, and what authority it has. Simple loops may have one entry; split-role loops may have many.

B.4 The Review Rubric

The review rubric asks whether each field is strong enough for the claim being made. The standard should rise with commitment. A speculative idea can survive with weak evidence if it is labeled as such. A tool recommendation, RTL change, or physical-design decision needs a stronger evidence ledger.

Table B.3 makes that standard inspectable. It separates a pass signal from a warning sign so review can focus on evidence, rejection, and commitment rather than polish.

Table B.3: **The review rubric separates readiness from polish.** A loop is ready only when its evidence, rejection structure, and commitment level match the claim being made.

Field	Pass signal	Warning sign
Intent and task	The task is bounded, measurable, and tied to an architectural objective.	The task is “use AI” or “generate a design” without a clear decision boundary.
Design space	Legal, invalid, and out-of-scope choices are named before the method acts.	The loop can wander into unreviewed choices or silently exclude alternatives.
Representation	The loop exposes the state needed to make valid architectural actions.	Important constraints live in hidden scripts, defaults, or informal assumptions.
Environment	Actions, observations, invalid states, costs, and provenance are defined.	The tool wrapper returns numbers but hides semantics, failures, or version state.
Method role	The method’s job is clear and matched to the feedback budget.	The method is chosen because it is fashionable, not because the task needs it.
Feedback budget	Latency, fidelity, sample count, and cost are explicit.	Claims ignore simulator time, EDA cost, expert review, or license limits.
Evidence	The evidence is relevant to the claim and calibrated to the commitment level.	A proxy metric is treated as truth without validation or uncertainty.
Negative traces	Failed candidates and rejected alternatives are captured with reasons.	Only successful runs are recorded.
Rejection authority	The loop states what can reject a candidate and what happens next.	There is no clear way to say no to a plausible but invalid result.
Commitment boundary / would overturn	The strongest supported claim and the evidence that would overturn it are explicit.	A proxy result is allowed to authorize a stronger commitment than it supports.
Human decision	Human judgment and accountability are explicit.	The loop implies that the method decides, but no one owns the commitment.

The rubric is not a scoring system by default. A simple three-level annotation is often enough:

- *Ready*: the field is explicit and adequate for the commitment.
- *Needs evidence*: the field is plausible but underspecified.
- *Unsafe*: the field is missing or inconsistent with the claim.

The most important review question is not whether every field is perfect. It is whether the loop exposes enough structure for another architect to judge, repeat, reject, or extend the work.

B.5 Card Conformance Levels

A card can be filled at increasing strength, and naming the level turns "we used the card" into a checkable claim rather than a gesture. Each level inherits the one below it. The rubric's pass and warning columns and its Ready/Needs-evidence/Unsafe annotations judge each field's content; the conformance levels judge the card as a whole; the required, optional, and redaction-allowed marks below connect the two.

- **Level 0, context only.** Intent, task, and design space are named. Enough to understand what the loop was for; not enough to audit it.
- **Level 1, auditable ledger.** Adds representation, environment, feedback budget, evidence with a named baseline, and negative traces. A reader can see what was tried, what it cost, and what was rejected.
- **Level 2, replayable loop.** Adds the stable IDs and provenance (workload, seed, tool version, parameter hash) that let another team re-run the evidence and reproduce the ledger.
- **Level 3, independently rejectable loop.** Adds a rejection authority that is independent of the producer actor, an explicit commitment boundary, and a named human decision owner. The result can be rejected by someone other than the party that produced it.

Within a level, mark each field as *required* (its absence caps the conformance level), *optional*, or *redaction-allowed* (the content may be confidential, but its presence and a hash are still disclosed so the record stays auditable). A missing required field does not lower the score a little; it caps the whole card at the highest level whose required fields are all present.

B.6 Adapting the Card for Different Contexts

The Design-Loop Card is the core artifact. However, you will use it differently depending on your audience. You do not need to create separate "Claim Cards" or "Evidence Ledgers"; rather, you simply emphasize different fields of the Design-Loop Card depending on the context.

B.6.1 1. For Research Papers (Architecture Claim Card View)

When publishing an Architecture 2.0 paper, the card answers the author and reviewer question that sits one level above the loop: what exact claim is being made, what evidence supports it, and what commitment does the evidence authorize?

Focus on:

- **Claim boundary:** The exact architectural claim and non-claim.
- **Evidence ledger:** Proxy metrics, simulation, synthesis, signoff, hardware measurement, and their uncertainty.
- **Negative traces:** Crashes, invalid actions, constraint violations, and rejected alternatives. This prevents success-only reporting.
- **Rejection authority:** Tests, tools, constraints, or reviewers that can reject the candidate.

A paper may be technically interesting while leaving the loop underdescribed. The review question is whether a reader can tell what was tried, what failed, what was measured, what could reject the claim, and what decision the evidence actually supports. (See Table B.4 for how to categorize the evidence made available).

B.6.2 2. For Case Studies and Examples (Shareable Evidence Ledger)

When sharing a case study, benchmark, or industrial example, the card acts as a shareable evidence ledger. It turns a success story into a reusable loop lesson.

Focus on:

- **Task and action space:** What the loop was trying to do, and which actions were legal.
- **Feedback and evidence:** What was measured, at what fidelity, with what provenance.
- **Rejected alternatives:** What failed or regressed at higher fidelity.
- **Loop judgment:** The evidence threshold and human-owned commitment boundary the example demonstrates.

Not every project can disclose the same evidence. The useful standard is to state the disclosure level (Public replay, Public evidence ledger, Confidential evidence available, Context only) rather than pretending the evidence is either fully open or absent.

Table B.4: **Evidence disclosure should be explicit.** A paper or review should say whether evidence is replayable, shareable, confidential, or only contextual.

Disclosure level	What is shared	Appropriate claim
Public replay	Code, data, configs, versions, seeds, failed runs, and reviewer instructions.	Reproducible result within the stated environment.

Disclosure level	What is shared	Appropriate claim
Public evidence ledger	Claim boundary, tool versions, metrics, candidates, rejected alternatives, and redacted limits.	Auditable loop lesson, but not full reproduction.
Confidential evidence available	Internal traces, proprietary tools, RTL, PDK, or product data exist but cannot be released.	Bounded claim with explicit redaction and review boundary.
Context only	Public source supports motivation, but not the full loop evidence.	Example of a pattern, not proof of a general architecture claim.

B.7 Paper-to-Loop Exercise

To use the card in a reading group or class, choose a paper and fill in the fields before discussing the claimed result. The exercise usually reveals one of three things.

First, some papers make the loop explicit. They name the task, action space, environment, feedback budget, and evidence ledger. These papers are easier to explain and compare because their claims are grounded in visible state, legal actions, feedback budget, evidence, rejection authority, and decision ownership.

Second, some papers have strong technical results but implicit loop structure. They may report a better Pareto point or speedup without exposing enough about the search budget, failed candidates, tool settings, or rejection authority. The card helps readers separate a useful artifact from a fully auditable loop.

Third, some papers make broad claims from narrow evidence. A method may work for one benchmark, simulator, or proxy metric but be presented as a general design method. The card reveals the mismatch between claim scope and evidence scope.

A simple reading-group exercise is to assign two readers the same paper. One summarizes the paper conventionally. The other fills in the design-loop card. The group then discusses what the card exposed that the summary hid.

B.8 Lighthouse Card Sketch

Table B.5 gives a deliberately incomplete card sketch for the lighthouse prompt. It is not a finished design. It shows how a short prompt becomes a loop that must be specified before any result can be trusted.

Table B.5: **The lighthouse card sketch is a deliberately incomplete first pass.** It shows how a short prompt becomes loop state before any generated answer should be trusted.

Field	Sketch
Intent	Improve efficiency for real-time mobile XR under strict power, memory, software, reliability, and deployment constraints.
Task	Bounded design-space exploration for a RISC-V-based compute subsystem, initially scoped to accelerator/memory organization for an XRBench-class workload slice.
Design space	RISC-V ISA options, vector width, memory hierarchy, accelerator coupling, compiler/runtime path, and voltage/frequency assumptions; technology, package, and deployment policy changes are outside this first turn.
Representation	Workload traces, architecture description, configurable memory/compute parameters, compiler assumptions, power model, latency targets, and uncertainty about workload drift.
Environment	Simulator or cost model plus workload harness, with actions such as changing vector width, memory hierarchy parameters, accelerator tiling, voltage/frequency assumptions, or dataflow choices.
Method role	For one loop turn, choose one bounded role, such as generating legal candidates, searching exposed parameters, predicting proxy outcomes, critiquing invalid assumptions, or summarizing evidence for human review.
Feedback budget	Many cheap proxy evaluations, fewer simulator evaluations, and only a small number of high-fidelity checks before human review.
Evidence	Pareto comparison over latency, energy, area proxy, memory traffic, software compatibility, and sensitivity to workload assumptions.
Negative traces	Configurations that violate the 3 W target, miss real-time latency, exceed memory bandwidth, require unsupported software, or fail at higher fidelity.
Rejection authority	Constraint checker, simulator failure, CDC/RDC violation, routing congestion, power/thermal limit, workload quality-of-service violation, compiler/runtime incompatibility, framework dispatch failure, or architect review.
Commitment boundary / would overturn	Advance only to a stronger modeling or RTL-study question; overturn with higher-fidelity latency, power, software-path, or workload-coverage evidence.
Human decision	Decide whether the candidate merits deeper modeling, different representation, stronger fidelity, or rejection.

This sketch also shows why the book does not treat the lighthouse prompt as a one-shot generation request. The prompt is useful because it exposes the state that must be represented, not because it eliminates the loop.

B.9 Common Failure Modes

The card is most useful when it reveals failures early. Common failure modes include:

- **Missing evidence:** the claim is plausible, but the supporting measurement is absent, low fidelity, or unrelated to the decision.
- **No negative traces:** the loop records only successful candidates, so future methods repeat known failures.
- **Hidden simulator state:** defaults, flags, seeds, workload versions, and tool revisions are not recorded.
- **Proxy mismatch:** the method improves a metric that does not track the architectural objective.
- **Invalid action space:** the generative method can propose configurations that cannot compile, simulate, synthesize, meet timing, or satisfy constraints.
- **Unsupported autonomy:** the method is allowed to make decisions whose commitment level exceeds the evidence available.
- **No rejection authority:** there is no explicit mechanism that can reject a plausible but wrong result.
- **Unowned commitment:** the loop obscures who accepts risk and who remains accountable for the final decision.

These are not only documentation failures. They are design-loop failures. A loop that hides negative traces, invalid actions, or rejection authority is hard to improve because it cannot distinguish a weak candidate from a weak process.

B.10 Blank Template

Table B.6 is the one-page blank form. It is sufficient for a first pass because it forces the loop owner to name the task, design space, evidence, rejection path, commitment boundary, and decision owner before running the loop.

Table B.6: **The blank card provides a reusable loop template.** A reader can fill it in for a paper, tool, benchmark, project proposal, or internal loop before judging the claim.

Field	Entry
Intent	
Task	*(What is the architectural goal? e.g., generation, verification)*

Field	Entry
Design space	
Representation	*(Including intermediate representation level, e.g., MLIR, gate-level)*
Environment	
Method role	*(How is the algorithm specifically used? e.g., predict, critique)*
Feedback budget	
Evidence	
Negative traces	
Rejection authority	
Commitment boundary / would overturn	
Human decision	

Field note: Practical workflow tips for solo researchers

If you are a student or solo researcher, the machine-checkable schema may feel like intimidating bureaucracy. You do not need an enterprise database to track this. For Negative traces, simply saving a local CSV of failed parameters and the tool's error code is sufficient to prove you explored the space. For Rejection authority, a "simulator segfault" or "failed to compile" is a perfectly valid gate to record. The goal is clarity and intellectual honesty, not compliance paperwork.

After filling in the card, run the final review gate.

Architect's checkpoint

Before treating a filled card as a credible loop, ask:

1. Is the task bounded enough that the loop can be evaluated?
2. Is the representation sufficient for the actions the method is allowed to take?
3. Is the feedback budget realistic for the method and claim?
4. Does the evidence match the commitment level?
5. What can reject the result, and who owns the final decision?

If those questions cannot be answered, the project may still be promising, but it is not yet a credible Architecture 2.0 loop.

Appendix C

Architecture 2.0 Resource Catalog and Links

This catalog is not a directory of links and it is not an endorsement list. Links change, tools age, and benchmark versions move. The stable question is what role a resource plays in the design loop. Does it provide workload state? Does it define valid actions? Does it return feedback? Does it expose evidence that can reject a candidate? Does it preserve provenance or negative traces?

The specific examples named below are a snapshot. The durable content is the role each resource plays; the current list of tools, datasets, and benchmarks is the kind of fast-moving record that belongs with the community forming around this topic, where it can stay current without reprinting the book. This appendix combines both the framework for judging whether a resource belongs in the loop and the current release directory of those resources.

> **Architecture 2.0 resource.** A resource is useful for Architecture 2.0 when it makes some part of the design loop explicit, such as task, representation, environment, method role, feedback, evidence, rejection, or human decision.

Table C.1 gives a first-pass catalog. The examples are deliberately representative, not exhaustive. A reader should use the table to ask what is missing from a loop before adding another model or tool.

Table C.1: **Useful resources should be classified by loop role.** A dataset, benchmark, harness, simulator, compiler, or card is valuable only if the loop records what it can and cannot support.

Resource family	Examples	Loop role	Watch for
Architecture corpora and QA	Paper/manual corpora, DBLP-derived publication records, QuArch-style QA and reasoning data (Prakash et al., 2025b,a).	Extract claim boundaries, architecture vocabulary, missing loop fields, and literature-grounded evidence limits; do not treat paper text as simulator state or rejection authority.	Paper text rarely preserves simulator state, failed candidates, tool logs, or review judgment.
Workloads and benchmarks	XRBench (a mobile extended reality benchmark), MLPerf (an industry-standard machine learning benchmark), and maintained benchmark suites (Kwon et al., 2023; Mattson et al., 2020; Reddi et al., 2020).	Define workload state, scenarios, metrics, rules, and comparability.	Coverage, drift, update policy, and proxy validity must remain visible.
Evaluation harnesses and environments	ArchGym-style environments (such as an open-source gymnasium for architecture search), benchmark harnesses, simulator wrappers, and tool-calling APIs (Krishnan et al., 2023).	Define valid actions, observations, feedback cost, logging, and rejection behavior.	A wrapper can hide tool semantics, unsupported actions, nondeterminism, and failure modes.
Mapping and DSE frameworks	Timeloop and MAESTRO-style mapping and dataflow tools (analytical models for accelerator evaluation) (Parashar et al., 2019; Kwon et al., 2019).	Make architecture search spaces and constraints explicit enough to explore.	Fast feedback is still a model; calibration, workload scope, and invalid candidates matter.

Resource family	Examples	Loop role	Watch for
Compiler, autotuning, and codegen resources	AutoTVM, Ansor (tensor program optimizers), MLIR (a compiler infrastructure), and kernel-generation benchmarks (Chen et al., 2018; Zheng et al., 2020; Lattner et al., 2020; Ouyang et al., 2025).	Connect specialized hardware ideas to executable software paths.	A kernel, schedule, or IR result is not automatically a system-level architecture result.
RTL and verification benchmarks	Executable specification-to-RTL and Verilog design-problem tasks such as VerilogEval and CVDP (Comprehensive Verilog Design Problems) (datasets for hardware code generation); see the live links below.	Test method claims against compile, simulation, and verification feedback.	Passing a small HDL task is not the same as closing an architecture loop.
Full-system simulation and hardware/software harnesses	gem5 (a modular full-system simulator), FireSim (an FPGA-accelerated simulation platform), Chipyard (an integrated SoC design framework), and related harnesses (Binkert et al., 2011; Karandikar et al., 2018; Amid et al., 2020).	Connect workload execution, software stacks, generated hardware, and stronger feedback.	Setup state, versions, nondeterminism, and workload coverage must be recorded.
Physical-design and EDA evidence	OpenROAD's open RTL-to-GDS flow, CircuitNet (an open-source dataset for machine learning in EDA), ChiPBench (a benchmark for physical design), and placement or signoff-adjacent artifacts (Chai et al., 2022; Wang et al., 2025).	Give physical constraints authority to reject architecture candidates.	Intermediate scores can mislead unless tied to downstream timing, area, power, or routability.

Resource family	Examples	Loop role	Watch for
Benchmark governance and roadmaps	MLCommons rules, SPEC (Standard Performance Evaluation Corporation) suites, and roadmaps in the style of the International Roadmap for Devices and Systems (IRDS).	Maintain comparability, versioning, scenario definitions, and long-horizon evidence needs.	Governance is part of the loop; stale rules or hidden updates change what claims mean.
Evidence and provenance artifacts	Design-loop cards, evidence ledgers, source records, seeds, configs, tool logs, calibration records, and negative traces.	Make claims auditable, reproducible, rejectable, and reusable.	These records are often uncodified, private, or discarded because they are not publication artifacts.

C.1 Use the Catalog as a Loop Checklist

The catalog is most useful when it is used as a checklist. For a new Architecture 2.0 project, choose one resource for each role:

- a workload or benchmark that defines the task boundary;
- a representation that records the state the loop can read and change;
- an environment or harness that defines valid actions and observations;
- a feedback source with an explicit latency, fidelity, and cost model;
- an evidence ledger that preserves configurations, assumptions, and negative traces;
- rejection authority and a human decision owner.

If one of these fields is missing, the loop may still be useful, but its claim should be bounded accordingly. A paper-reading tool can help identify claim boundaries, missing workload records, absent negative traces, and unsupported commitments even if it cannot act on RTL. A simulator environment can support design-space exploration even if it cannot validate timing closure. A kernel-generation benchmark can reveal code-generation capability even if it does not prove system-level efficiency.

Lighthouse prompt: Use the catalog to fill the Lighthouse loop

Context. The catalog becomes a checklist only when every Lighthouse fragment has a resource that can carry it.
In the Lighthouse prompt. "XRBench-class real-time mobile XR workload" anchors the workload family, while the loop must still pin the scenario, version, inputs, latency target, software stack, and coverage. Architecture parameters and

workload traces supply representation. A simulator or cost-model wrapper supplies legal actions and observations.
Resource map. Compiler/runtime resources test the "64-bit RISC-V-based" ISA/ABI, vector path, driver or runtime API, library integration, and fallbacks. Full-system or SoC harnesses test integration, coherence, and data movement. EDA, PDK, and library evidence can test implementation feasibility under the "3 W TDP target in a 3 nm-class LP mobile process." Verification resources test correctness, corner cases, and reliability assumptions. A card or ledger turns the "design-space report with evidence and rejected alternatives" into a concrete evidence ledger with a human owner.
Takeaway. An empty row does not make the loop useless; it bounds the claim the loop is allowed to make.

C.2 Missing Infrastructure

The most important future resources are not only larger corpora. Architecture needs shared records of design-loop state:

- negative-trace repositories that preserve failed candidates and reasons;
- environment schemas that state actions, observations, costs, and invalid states;
- benchmark-update protocols that record drift and version changes;
- confidentiality-preserving ways to share tool traces and design reviews;
- standard design-loop cards for papers, artifacts, and design reviews.

These resources would make the field more reviewable and more cumulative. They would also make AI-assisted architecture work easier to evaluate, because the community could ask whether a method improved the loop rather than merely whether it produced a plausible artifact. The sections below keep the moving list of concrete links organized by these roles.

C.3 Live Resource Directory

These links are not a general computer-architecture, pedagogy, or reproducibility directory. A resource earns space here only if it is directly useful for AI-mediated architecture work, such as naming a task, representing state, exposing actions, returning feedback, preserving evidence, or rejecting a result. Use the list as a starting point and check current versions before relying on any benchmark, dataset, simulator, or tool. This directory uses live links rather than formal bibliography entries; cite the primary paper or project artifact when making a scholarly claim.

C.4 Architecture 2.0 Framing

- **Essay:** Architecture 2.0 gymnasium essay. Intent-level framing for why architecture needs data-centric environments rather than isolated model demonstrations. — title: "Architecture 2.0 Resources and Tool Catalog" listing:
  - id: tool-catalog contents: tools.yml type: grid categories: true filter-ui: true sort-ui: false fields: [title, description, categories] —

> The most important metric for an open-source project is the number of people who use it to build something the original authors never anticipated.

— Unknown, open-source adage

This appendix is not a bibliography. It is an index of *Architecture 2.0 building blocks*: the simulators, proxy models, verification harnesses, and data representations that provide the *State*, *Action*, and *Rejection* surfaces required to build AI-assisted design loops.

If you are building an AI-assisted workflow, you will likely need to piece together tools from multiple categories below.

Author's Note: The ecosystem of AI tools for computer architecture and hardware/software co-design is moving faster than any static text can capture. The catalog below is a living document. We invite the community to submit new proxy models, simulators, AI-assisted workflows, and benchmarks to the Architecture 2.0 GitHub Repository.

C.5 The Living Tool Catalog

Below is a searchable, filterable grid of community-submitted resources that enable Architecture 2.0.

C.6 Architecture 2.0 Templates and Artifacts

- **Design-loop card and review rubric:** Appendix B. Use the core card and its contextual adaptations to review a paper, proposal, research artifact, or internal loop before trusting its architectural claim.
- **Blank design-loop template:** Table B.6. Use the blank card as a one-page handout for paper reviews, project proposals, and design reviews.

C.7 Using and Citing the Preview

Use this preview as a working vocabulary for paper reviews, project proposals, class discussions, and internal design reviews. For citation, cite the preview URL and version when referring to this book directly, and cite the Architecture 2.0 foundations article when referring to the broader vision and autonomy framing.

For feedback on the preview, or to submit a tool to the catalog above, visit the Architecture 2.0 GitHub Repository.

References

Altaf MSB, Wood DA (2017) LogCA: A high-level performance model for hardware accelerators. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), ACM, DOI 10.1145/3140659.3080216, URL https://doi.org/10.1145/3140659.3080216

Amarasinghe S (2020) Compiler 2.0: Using machine learning to modernize compiler technology. In: Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, ACM, DOI 10.1145/3372799.3397167, URL https://doi.org/10.1145/3372799.3397167

Amarasinghe S (2026) Compiler 2.0: Building the next generation compilers with machine learning. URL https://www.csail.mit.edu/event/csail-forum-saman-amarasinghe-compiler-20-building-next-generation-compilers-machine-learning, CSAIL Forum abstract, accessed June 22, 2026

Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, ACM, AFIPS ’67 (Spring), pp 483–485, DOI 10.1145/1465482.1465560

Amdahl GM, Blaauw GA, Brooks FP (1964) Architecture of the IBM System/360. IBM Journal of Research and Development 8(2):87–101, DOI 10.1147/rd.82.0087

Amid A, Biancolin D, Gonzalez A, Grubb D, Karandikar S, Liew H, Magyar A, Mao H, Ou A, Pemberton N, et al. (2020) Chipyard: Integrated design, simulation, and implementation framework for custom socs. IEEE Micro 40(4):10–21, DOI 10.1109/MM.2020.3013233, URL https://ieeexplore.ieee.org/document/9154690

Amodei D, Olah C, Steinhardt J, Christiano P, Schulman J, Mané D (2016) Concrete problems in AI safety. arXiv preprint arXiv:160606565

Angelopoulos AN, Bates S (2021) A gentle introduction to conformal prediction and distribution-free uncertainty quantification. arXiv preprint arXiv:2107.07511, URL https://arxiv.org/abs/2107.07511

Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O’Reilly UM, Amarasinghe S (2014) OpenTuner: An extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT), ACM, pp 303–315, DOI 10.1145/2628071.2628092, URL https://doi.org/10.1145/2628071.2628092

Apple (2024a) Apple debuts iPhone 16 pro and iPhone 16 pro max. URL https://www.apple.com/newsroom/2024/09/apple-debuts-iphone-16-pro-and-iphone-16-pro-max/, apple Newsroom, accessed June 22, 2026

Apple (2024b) Apple introduces M4 chip. URL https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/, apple Newsroom, accessed June 22, 2026

Baggerly KA, Coombes KR (2009) Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. The Annals of Applied Statistics 3(4):1309–1334, DOI 10.1214/09-AOAS291

Barroso LA, Hölzle U, Ranganathan P (2019) The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. Synthesis Lectures on Computer Architecture, Springer International Publishing, DOI 10.1007/978-3-031-01761-2, URL https://doi.org/10.1007/978-3-031-01761-2

Bauer H, Burkacky O, Kenevan P, Lingemann S, Pototzky K, Wiseman B (2020) Semiconductor design and manufacturing: Achieving leading-edge capabilities. McKinsey & Company report, URL https://www.mckinsey.com/industries/semiconductors/our-insights/semiconductor-design-and-manufacturing-achieving-leading-edge-capabilities

Benmeziane H, El Maghraoui K, Ouarnoughi H, Niar S, Wistuba M, Wang N (2021) Hardware-aware neural architecture search: Survey and taxonomy. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, pp 4322–4329, DOI 10.24963/ijcai.2021/592, URL https://www.ijcai.org/proceedings/2021/592

Beyer B, Jones C, Petoff J, Murphy NR (2016) Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media

Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2):1–7, DOI 10.1145/2024716.2024718

Blocklove J, Garg S, Karri R, Pearce H (2023) Chip-Chat: Challenges and opportunities in conversational hardware design. In: 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD), IEEE, pp 1–6, DOI 10.1109/MLCAD58807.2023.10299874, URL https://arxiv.org/abs/2305.13243

Borkar S, Chien AA (2011) The future of microprocessors. Communications of the ACM 54(5):67–77, DOI 10.1145/1941487.1941507

Cadence Design Systems (2021) Machine learning-driven full-flow chip design automation. Cadence Cerebrus Intelligent Chip Explorer, product white paper, URL https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/cerebrus-wp.pdf

Cadence Design Systems (2022) Cadence revolutionizes verification productivity with the Verisium AI-driven verification platform. Cadence press release, URL https://www.businesswire.com/news/home/20220913005378/en/Cadence-Revolutionizes-Verification-Productivity-with-the-Verisium-AI-Driven-Verification-Platform

Chai Z, Zhao Y, Lin Y, Liu W, Wang R, Huang R (2022) CircuitNet: An open-source dataset for machine learning applications in electronic design automation (EDA). Science China Information Sciences DOI 10.1007/s11432-022-3571-8, 2208.01040

Chen T, Zheng L, Yan E, Jiang Z, Moreau T, Ceze L, Guestrin C, Krishnamurthy A (2018) Learning to optimize tensor programs. In: Advances in Neural Information Processing Systems, vol 31

Cheng CK, Kahng AB, Kundu S, Wang Y, Wang Z (2023a) Assessment of reinforcement learning for macro placement. In: Proceedings of the 2023 International Symposium on Physical Design (ISPD), DOI 10.1145/3569052.3578926

Cheng CK, Kahng AB, et al. (2023b) Assessment of reinforcement learning for macro placement. In: Proceedings of the 2023 International Symposium on Physical Design (ISPD), DOI 10.1145/3569052.3578926, 2302.11014

Christopher E, Crossan K, Dobson W, Kennelly C, Lewis D, Lin K, Maas M, Ranganathan P, Rapati E, Yang B (2025) Instruction set migration at warehouse scale. arXiv preprint arXiv:251014928 URL https://arxiv.org/abs/2510.14928

Clark J, Amodei D (2016) Faulty reward functions in the wild. OpenAI blog, URL https://openai.com/index/faulty-reward-functions/

Coussy P, Morawiec A (eds) (2008) High-Level Synthesis: From Algorithm to Digital Circuit. Springer, DOI 10.1007/978-1-4020-8588-8

De Micheli G (1994) Synthesis and Optimization of Digital Circuits. McGraw-Hill

Defense Advanced Research Projects Agency (2014) The DARPA grand challenge: 10 years later. URL https://www.darpa.mil/news/2014/grand-challenge-ten-years-later, accessed June 23, 2026

Dehnert JC, Grant BK, Banning JP, Johnson R, Kistler T, Klaiber A, Mattson J (2003) The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp 15–24

Dennard RH, Gaensslen FH, Yu HN, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE Journal of Solid-State Circuits 9(5):256–268, DOI 10.1109/JSSC.1974.1050511

Dixit HD, Pendharkar S, Beadon M, Mason C, Chakravarthy T, Muthiah B, Sankar S (2021) Silent data corruptions at scale. arXiv preprint arXiv:210211245

Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, ACM, ISCA '11, pp 365–376, DOI 10.1145/2000064.2000108

Foster H (2022) Part 8: The 2022 wilson research group functional verification study. Verification Horizons, Siemens EDA, URL https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/

Foster H (2025) IC/ASIC functional verification trend report - 2024. Verification Academy, Siemens EDA, URL https://verificationacademy.com/topics/planning-measurement-and-analysis/wrg-

industry-data-and-trends/2024-siemens-eda-and-wilson-research-group-ic-asic-functional-verification-trend-report/, last updated February 2025
Gebru T, Morgenstern J, Vecchione B, Vaughan JW, Wallach H, Daum 'e III H, Crawford K (2021) Datasheets for datasets. Communications of the ACM 64(12):86–92, DOI 10.1145/3458723, URL https://doi.org/10.1145/3458723
Goldie A, Mirhoseini A, Yazdanbakhsh A, Dean J (2024) That chip has sailed: A critique of unfounded skepticism around ai for chip design. arXiv preprint arXiv:241110053
Gulwani S, Polozov O, Singh R (2017) Program synthesis. Foundations and Trends in Programming Languages 4(1–2):1–119, DOI 10.1561/2500000010
Gupta U, Kim YG, Lee S, Tse J, Lee HHS, Wei GY, Brooks D, Wu CJ (2021) Chasing carbon: The elusive environmental footprint of computing. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp 854–867, DOI 10.1109/HPCA51647.2021.00076, URL https://doi.org/10.1109/HPCA51647.2021.00076
Ha D, Schmidhuber J (2018) World models. arXiv preprint arXiv:180310122
Haj-Ali A, Huang Q, Harrison J, Shao YS, Asanovic K, Wawrzynek J, Stoica I (2020) AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning. In: Proceedings of Machine Learning and Systems, vol 2, pp 284–298
He Z, et al. (2024) LLM4EDA: Emerging progress in large language models for electronic design automation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
Hennessy JL, Patterson DA (2017) Computer Architecture: A Quantitative Approach, 6th edn. Morgan Kaufmann
Hennessy JL, Patterson DA (2019) A new golden age for computer architecture. Communications of the ACM 62(2):48–60, DOI 10.1145/3282307
Hill MD, Marty MR (2008) Amdahl's law in the multicore era. Computer 41(7):33–38, DOI 10.1109/MC.2008.209
Hochschild PH, Turner P, Mogul JC, Govindaraju R, Ranganathan P, Culler DE, Vahdat A (2021) Cores that don't count. In: Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS), pp 9–16, DOI 10.1145/3458336.3465297
Hoffmann J, Borgeaud S, Mensch A, Buchatskaya E, Cai T, Rutherford E, de Las Casas D, Hendricks LA, Welbl J, Clark A, Hennigan T, Noland E, Millican K, van den Driessche G, Damoc B, Guy A, Osindero S, Simonyan K, Elsen E, Rae J, Vinyals O, Sifre L (2022) Training compute-optimal large language models. In: Advances in Neural Information Processing Systems, vol 35, pp 30016–30030, DOI 10.48550/arXiv.2203.15556, URL https://arxiv.org/abs/2203.15556, 2203.15556
Horowitz M (2014) 1.1 computing's energy problem (and what we can do about it). 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers pp 10–14, DOI 10.1109/ISSCC.2014.6757323
Hoste K, Eeckhout L (2007) Microarchitecture-independent workload characterization. IEEE Micro 27(3):63–72, DOI 10.1109/MM.2007.56
Huang K, Altosaar J, Ranganath R (2019) ClinicalBERT: Modeling clinical notes and predicting hospital readmission. arXiv preprint arXiv:190405342 URL https://arxiv.org/abs/1904.05342, 1904.05342
Huzaifa M, Desai R, Grayson S, Jiang X, Jing Y, Lee J, Lu F, Pang Y, Ravichandran J, Sinclair F, Tian B, Yuan H, Zhang J, Adve SV (2021) ILLIXR: Enabling end-to-end extended reality research. In: 2021 IEEE International Symposium on Workload Characterization (IISWC), pp 24–38, DOI 10.1109/IISWC53511.2021.00014
Intel Corporation (2016) Intel corporation 2015 form 10-k. URL https://www.sec.gov/Archives/edgar/data/50863/000005086316000105/a10kdocument12262015q4.htm, fiscal year ended December 26, 2015
International Business Strategies (2024) Cost to design an IC by process node. URL https://www.tomshardware.com/news/firm-estimates-a-2nm-chip-now-costs-dollar725-million-to-design, iBS design-cost estimates extending the leading-node trend: about $590M at 3 nm and $725M at 2 nm; reported by Tom's Hardware
International Organization for Standardization (2018) ISO 26262: Road Vehicles — Functional Safety. ISO, Geneva, Switzerland, 2nd edn
Ipek E, McKee SA, de Supinski BR, Schulz M, Caruana R (2006) Efficiently exploring architectural design spaces via predictive modeling. In: Proceedings of the 12th International Conference on

Architectural Support for Programming Languages and Operating Systems, ACM, ASPLOS XII, pp 195–206, DOI 10.1145/1168857.1168882

Janapa Reddi V, Yazdanbakhsh A (2023) Architecture 2.0: Why computer architects need a data-centric ai gymnasium. URL https://www.sigarch.org/architecture-2-0-why-computer-architects-need-a-data-centric-ai-gymnasium/, aCM SIGARCH Computer Architecture Today, June 14, 2023

Janapa Reddi V, Yazdanbakhsh A (2025) Architecture 2.0: Foundations of artificial intelligence agents for modern computer system design. Computer 58(2):116–124, DOI 10.1109/MC.2024.3521641

Jones DR, Schonlau M, Welch WJ (1998) Efficient global optimization of expensive black-box functions. Journal of Global Optimization 13(4):455–492, DOI 10.1023/A:1008306431147

Jumper J, Evans R, Pritzel A, Green T, Figurnov M, Ronneberger O, Tunyasuvunakool K, Bates R, Zidek A, Potapenko A, Bridgland A, Meyer C, Kohl SAA, Ballard AJ, Cowie A, Romera-Paredes B, Nikolov S, Jain R, Adler J, Back T, Petersen S, Reiman D, Clancy E, Zielinski M, Steinegger M, Pacholska M, Berghammer T, Bodenstein S, Silver D, Vinyals O, Senior AW, Kavukcuoglu K, Kohli P, Hassabis D (2021) Highly accurate protein structure prediction with AlphaFold. Nature 596(7873):583–589, DOI 10.1038/s41586-021-03819-2, URL https://doi.org/10.1038/s41586-021-03819-2

Kandasamy K, Dasarathy G, Schneider J, Póczos B (2017) Multi-fidelity Bayesian optimisation with continuous approximations. In: Proceedings of the 34th International Conference on Machine Learning (ICML), PMLR, vol 70, pp 1799–1808, URL https://proceedings.mlr.press/v70/kandasamy17a.html

Karandikar S, Mao H, Kim D, Biancolin D, Amid A, Lee D, Pemberton N, Amaro E, Schmidt C, Chopra A, Huang Q, Kovacs K, Nikolić B, Katz RH, Bachrach J, Asanović K (2018) FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp 29–42, DOI 10.1109/ISCA.2018.00014

Kelly T, Weaver R (2004) The goal structuring notation: A safety argument notation. In: Workshop on Assurance Cases, International Conference on Dependable Systems and Networks (DSN)

Knight Capital Group (2012) Knight capital group provides update regarding august 1st disruption to routing in nyse-listed securities. Press release (Form 8-K), August 2, 2012, realized pre-tax loss of approximately $440 million

Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2019) Spectre attacks: Exploiting speculative execution. In: IEEE Symposium on Security and Privacy (S&P)

Krishnan S, et al. (2023) ArchGym: An open-source gymnasium for machine learning assisted architecture design. In: Proceedings of the 50th Annual International Symposium on Computer Architecture, ACM, ISCA '23, DOI 10.1145/3579371.3589049, URL https://doi.org/10.1145/3579371.3589049

Kwon H, Chatarasi P, Pellauer M, Parashar A, Sarkar V, Krishna T (2019) Understanding reuse, performance, and hardware cost of DNN dataflows: A data-centric approach. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, MICRO '52, pp 754–768, DOI 10.1145/3352460.3358252

Kwon H, et al. (2023) XRBench: An extended reality (XR) machine learning benchmark suite for the metaverse. In: Proceedings of Machine Learning and Systems

Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O (2020) MLIR: A compiler infrastructure for the end of moore's law. arXiv preprint arXiv:200211054 DOI 10.48550/arXiv.2002.11054, URL https://arxiv.org/abs/2002.11054, 2002.11054

Lazer D, Kennedy R, King G, Vespignani A (2014) The parable of google flu: Traps in big data analysis. Science 343(6176):1203–1205, DOI 10.1126/science.1248506

Lee BC, Brooks DM (2006) Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, ASPLOS XII, pp 185–194

Lee J, Yoon W, Kim S, Kim D, Kim S, So CH, Kang J (2020) BioBERT: A pre-trained biomedical language representation model for biomedical text mining. Bioinformatics 36(4):1234–1240, DOI 10.1093/bioinformatics/btz682, URL https://doi.org/10.1093/bioinformatics/btz682

Lions JL (1996) Ariane 5 flight 501 failure: Report by the inquiry board. Tech. rep., European Space Agency

Liu M, Ene TD, Kirby R, Cheng C, Pinckney N, Liang R, et al. (2023) ChipNeMo: Domain-adapted LLMs for chip design. arXiv preprint arXiv:2311.00176, URL https://arxiv.org/abs/2311.00176

Mankowitz DJ, Michi A, Zhernov A, Gelbukh M, Batzner N, Paduraru C, Leblond G, Hassabis D, Kohli P, Riedel M (2023) Faster sorting algorithms discovered using deep reinforcement learning. Nature 618(7964):257–263

Markov IL (2023) The false dawn: Reevaluating google's reinforcement learning for chip macro placement. arXiv preprint arXiv:230609633

Mattson P, Tang H, Wei GY, Wu CJ, Janapa Reddi V, Cheng C, Coleman C, Diamos G, Kanter D, Micikevicius P, Patterson D, Schmuelling G (2020) MLPerf: An industry standard benchmark suite for machine learning performance. IEEE Micro 40(2):8–16, DOI 10.1109/MM.2020.2974843

Mead C, Conway L (1980) Introduction to VLSI Systems. Addison-Wesley

Meta AI (2024) Introducing Llama 3.1: Our most capable models to date. URL https://ai.meta.com/blog/meta-llama-3-1/, accessed June 23, 2026

Mirhoseini A, Goldie A, Yazdanbakhsh A, et al. (2021) A graph placement methodology for fast chip design. Nature 594(7862):207–212, DOI 10.1038/s41586-021-03544-w

Mitchell M, Wu S, Zaldivar A, Barnes P, Vasserman L, Hutchinson B, Spitzer E, Raji ID, Gebru T (2019) Model cards for model reporting. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, ACM, pp 220–229, DOI 10.1145/3287560.3287596, URL https://doi.org/10.1145/3287560.3287596

National Aeronautics and Space Administration (2008) July 20, 1969: One giant leap for mankind. URL https://www.nasa.gov/history/july-20-1969-one-giant-leap-for-mankind/, accessed June 23, 2026

National Human Genome Research Institute (2025) The human genome project. URL https://www.genome.gov/human-genome-project, accessed June 23, 2026

Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. ACM Queue 6(2):40–53, DOI 10.1145/1365490.1365500

Nygard M (2011) Documenting architecture decisions. URL https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions, accessed July 3, 2026

Open Source Security Foundation (2026) Supply-chain levels for software artifacts. URL https://slsa.dev/, accessed July 3, 2026

Ouyang A, Guo S, Arora S, Zhang AL, Hu W, Ré C, Mirhoseini A (2025) KernelBench: Can LLMs write efficient GPU kernels? arXiv preprint arXiv:250210517 DOI 10.48550/arXiv.2502.10517, URL https://arxiv.org/abs/2502.10517, 2502.10517

Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, Shamseer L, Tetzlaff JM, Akl EA, Brennan SE, Chou R, Glanville J, Grimshaw JM, Hrobjartsson A, Lalu MM, Li T, Loder EW, Mayo-Wilson E, McDonald S, McGuinness LA, Stewart LA, Thomas J, Tricco AC, Welch VA, Whiting P, Moher D (2021) The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. BMJ 372:n71, DOI 10.1136/bmj.n71, URL https://www.bmj.com/content/372/bmj.n71

Pan SJ, Yang Q (2010) A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering 22(10):1345–1359, DOI 10.1109/TKDE.2009.191

Parashar A, Raina P, Shao YS, Chen YH, Ying VA, Mukkara A, Venkatesan R, Khailany B, Keckler SW, Emer J (2019) Timeloop: A systematic approach to DNN accelerator evaluation. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, pp 304–315, DOI 10.1109/ISPASS.2019.00042

Patterson DA, Ditzel DR (1980) The case for the reduced instruction set computer. ACM SIGARCH Computer Architecture News 8(6):25–33, DOI 10.1145/641914.641917

Peherstorfer B, Willcox K, Gunzburger M (2018) Survey of multifidelity methods in uncertainty propagation, inference, and optimization. SIAM Review 60(3):550–591, DOI 10.1137/16M1082469

Pineau J, Vincent-Lamarre P, Sinha K, Lariviere V, Beygelzimer A, d'Alche Buc F, Fox E, Larochelle H (2021) Improving reproducibility in machine learning research: A report from the NeurIPS 2019 reproducibility program. Journal of Machine Learning Research 22(164):1–20, URL https://jmlr.org/papers/v22/20-303.html

Prakash S, et al. (2025a) QuArch: A benchmark for evaluating LLM reasoning in computer architecture. URL https://arxiv.org/abs/2510.22087, 2510.22087
Prakash S, et al. (2025b) QuArch: A question-answering dataset for AI agents in computer architecture. IEEE Computer Architecture Letters DOI 10.1109/LCA.2025.3541961, URL https://arxiv.org/abs/2501.01892, 2501.01892
Quinonero-Candela J, Sugiyama M, Schwaighofer A, Lawrence ND (eds) (2009) Dataset Shift in Machine Learning. MIT Press
Ragan-Kelley J, Adams A, Sharlet D, Barnes C, Paris S, Levoy M, Amarasinghe S, Durand F (2017) Halide: Decoupling algorithms from schedules for high-performance image processing. Communications of the ACM 61(1):106–115, DOI 10.1145/3150211
Rajpurkar P, Zhang J, Lopyrev K, Liang P (2016) SQuAD: 100,000+ questions for machine comprehension of text. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, pp 2383–2392, DOI 10.18653/v1/D16-1264, URL https://aclanthology.org/D16-1264/
Rasmy L, Xiang Y, Xie Z, Tao C, Zhi D (2021) Med-BERT: Pretrained contextualized embeddings on large-scale structured electronic health records for disease prediction. npj Digital Medicine 4(86), DOI 10.1038/s41746-021-00455-y, URL https://doi.org/10.1038/s41746-021-00455-y
Reddi VJ, Cheng C, Kanter D, Mattson P, Schmuelling G, Wu CJ, Anderson B, Breughe M, Charlebois M, Chou W, Chukka R, Coleman C, Davis S, Deng P, Diamos G, Duke J, Fick D, Gardner JS, Hubara I, Idgunji S, Jablin TB, Jiao J, St John T, Kanwar P, Lee D, Liao J, Lokhmotov A, Massa F, Meng P, Micikevicius P, Osborne C, Pekhimenko G, Rajan ATR, Sequeira D, Sirasao A, Sun F, Tang H, Thomson M, Wei F, Wu E, Xu L, Yamada K, Yu B, Yuan G, Zhong A, Zhang P, Zhou Y (2020) MLPerf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp 446–459, DOI 10.1109/ISCA45697.2020.00045, URL https://doi.org/10.1109/ISCA45697.2020.00045
Reddi VJ, Cheng C, Kanter D, Mattson P, Schmuelling G, Wu CJ (2021) The vision behind MLPerf: Understanding AI inference performance. IEEE Micro 41(3):10–18, DOI 10.1109/MM.2021.3066343, URL https://doi.org/10.1109/MM.2021.3066343
Roy R, Raiman J, Kant N, Elkin I, Kirby R, Siu M, Oberman S, Godil S, Catanzaro B (2021) PrefixRL: Optimization of parallel prefix circuits using deep reinforcement learning. In: Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC), IEEE, DAC '21, pp 853–858, DOI 10.1109/DAC18074.2021.9586094, URL https://arxiv.org/abs/2205.07000
Schlansker MS, Rau BR (2000) EPIC: Explicitly parallel instruction computing. IEEE Computer 33(2):37–45
Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M, Crespo JF, Dennison D (2015) Hidden technical debt in machine learning systems. In: Advances in neural information processing systems, vol 28
Seiler L, Carmean D, Sprangle E, Forsyth T, et al. (2008) Larrabee: A many-core x86 architecture for visual computing. ACM Transactions on Graphics 27(3):1–15, DOI 10.1145/1360612.1360617
Semiconductor Industry Association (2026) Chip design and R&D. URL https://www.semiconductors.org/policies/chip-design/, accessed June 22, 2026
Settles B (2009) Active learning literature survey. Tech. Rep. Computer Sciences Technical Report 1648, University of Wisconsin–Madison
Sevilla J, Heim L, Ho A, Besiroglu T, Hobbhahn M, Villalobos P (2022) Compute trends across three eras of machine learning. arXiv preprint arXiv:220205924 URL https://arxiv.org/abs/2202.05924
Shao YS, Reagen B, Wei GY, Brooks D (2014) Aladdin: A pre-RTL, power-performance accelerator simulator for rapid design space exploration. In: Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA), DOI 10.1109/ISCA.2014.6853196
Shao YS, Xi SL, Srinivasan V, Wei GY, Brooks D (2016) Co-designing accelerators and SoC interfaces using gem5-aladdin. In: International Symposium on Microarchitecture (MICRO)
Snoek J, Larochelle H, Adams RP (2012) Practical bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems, vol 25, pp 2951–2959
SPDX Project (2021) SPDX: The system package data exchange. URL https://spdx.dev/, iSO/IEC 5962:2021; accessed July 3, 2026

Srinivas N, Krause A, Kakade SM, Seeger M (2010) Gaussian process optimization in the bandit setting: No regret and experimental design. In: Proceedings of the 27th International Conference on Machine Learning (ICML), pp 1015–1022, URL https://arxiv.org/abs/0912.3995

Stack Overflow (2024) 2024 developer survey. URL https://survey.stackoverflow.co/2024/, a majority of professional developers reported using or planning to use AI tools in their development workflow

Standard Performance Evaluation Corporation (2017) SPEC CPU 2017 benchmark. URL https://www.spec.org/cpu2017/, accessed June 23, 2026

Star SL, Griesemer JR (1989) Institutional ecology, 'translations' and boundary objects: Amateurs and professionals in berkeley's museum of vertebrate zoology, 1907–39. Social Studies of Science 19(3):387–420, DOI 10.1177/030631289019003001

Strathern M (1997) 'improving ratings': Audit in the British university system. European Review 5(3):305–321

Sutton RS, Barto AG (2018) Reinforcement Learning: An Introduction, 2nd edn. MIT Press

Synopsys (2023) AI-designed chips reach scale with first 100 commercial tape-outs using Synopsys technology. Synopsys press release, URL https://www.prnewswire.com/news-releases/ai-designed-chips-reach-scale-with-first-100-commercial-tape-outs-using-synopsys-technology-301739936.html, dSO.ai, an autonomous reinforcement-learning application for block-level physical implementation

Thakur S, Bale R, Pearse B, et al. (2023) VeriGen: A large language model for hardware design. In: Design Automation Conference (DAC)

UCIe Consortium (2026) UCIe specifications. URL https://www.uciexpress.org/specifications, accessed June 22, 2026

US Securities and Exchange Commission (2013) In the matter of knight capital americas llc. Tech. Rep. Release No. 34-70694, SEC

USC Information Sciences Institute (2025) MOSIS 2.0's first year: Bridging research and production. URL https://www.isi.edu/news/972800/mosis-2-0s-first-year-bridging-research-and-production/, accessed June 23, 2026

Veripool (2026) Verilator. URL https://www.veripool.org/verilator/, accessed June 23, 2026

Villalobos P, Ho A, Sevilla J, Besiroglu T, Heim L, Hobbhahn M (2024) Will we run out of data? limits of LLM scaling based on human-generated data. DOI 10.48550/arXiv.2211.04325, URL https://arxiv.org/abs/2211.04325, 2211.04325

Wang A, Singh A, Michael J, Hill F, Levy O, Bowman SR (2018) GLUE: A multi-task benchmark and analysis platform for natural language understanding. In: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, Association for Computational Linguistics, pp 353–355, DOI 10.18653/v1/W18-5446, URL https://aclanthology.org/W18-5446/

Wang H, Zhang J, Jiang K, Wang H, Chen J, Zhu J (2026) KernelBenchX: A comprehensive benchmark for evaluating LLM-generated GPU kernels. arXiv preprint arXiv:260504956 DOI 10.48550/arXiv.2605.04956, URL https://arxiv.org/abs/2605.04956, 2605.04956

Wang Z, et al. (2025) Benchmarking end-to-end performance of AI-based chip placement algorithms. In: Advances in Neural Information Processing Systems (NeurIPS), 2407.15026

Wen Z, Zhang Y, Li Z, Liu Z, Xie L, Zhang T (2025) MultiKernelBench: A multi-platform benchmark for kernel generation. arXiv preprint arXiv:250717773 DOI 10.48550/arXiv.2507.17773, URL https://arxiv.org/abs/2507.17773, 2507.17773

Williams S, Waterman A, Patterson D (2009) Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM 52(4):65–76

X, The Moonshot Factory (2025) Our blueprint for moonshots. URL https://x.company/blog/posts/moonshot-blueprint/, accessed June 23, 2026

Yazdanbakhsh A, Bruns C, Ghasemipour SKS, Bouyssounouse R, Barnes P, Shi H, Milder P, et al. (2021) Apollo: Transferable architecture exploration. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp 915–929, DOI 10.1145/3466752.3480074, URL https://dl.acm.org/doi/10.1145/3466752.3480074

Zheng L, et al. (2020) Ansor: Generating high-performance tensor programs for deep learning. In: 14th USENIX Symposium on Operating Systems Design and Implementation, pp 863–879

Zillow Group (2021) Zillow group reports third quarter 2021 financial results and shares plan to wind down zillow offers operations. Press release, q3 2021 inventory writedown of approximately $304M