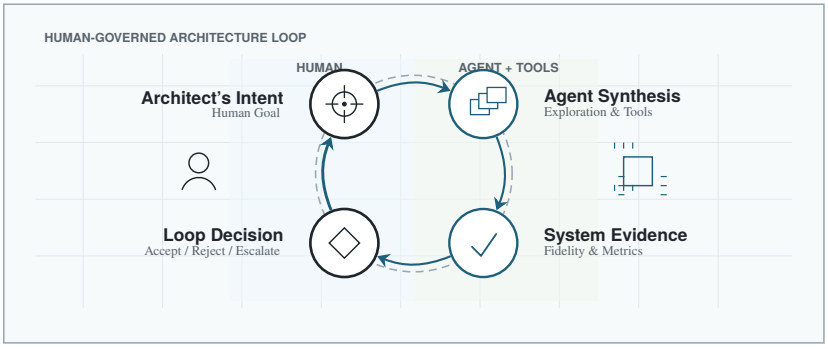
PREVIEW EDITION

Vijay Janapa Reddi

Architecture 2.0

Agentic Design Loops for Computing System Synthesis



Work in progress

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

June 25, 2026

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? This book is about that reversal and the claim that follows from it. The object an architect designs is no longer only the artifact. It is increasingly the design loop that produces, evaluates, rejects, and justifies the artifact.

I have made the broad case for this shift elsewhere. The foundations article with Amir Yazdanbakhsh argues why AI agents belong in modern computer system design and sets out the vision, history, ecosystem, capability horizons, and levels of autonomy that could follow (Janapa Reddi and Yazdanbakhsh, 2025). This book starts where that argument ends. It does not set out to convince the reader that the shift is coming. It asks a more practical and more durable question. Once AI systems can act inside architecture work, how do we design loops we can trust, and how do we tell a real result from a confident demonstration?

That question is familiar from a different field. Machine learning systems faced the same credibility problem a decade ago. Claims were everywhere and comparison was hard. The answer was not a better model. It was measurement discipline: 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 and comparable as the community learned to make AI-systems claims.

This book is written from the boundary between computer architecture, machine learning systems, benchmarking, and education. That boundary is where the central problem becomes visible. AI methods are powerful, but architecture progress depends on hardware and software interfaces, workload definitions, toolchains, evidence standards, and human judgment. My work across architecture, edge and machine learning systems, benchmarking, and machine-learning-systems education shapes the emphasis here on loops rather than on isolated models.

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: 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. The framework has five pieces: task and intent, representation and world model, tools and environment, the compound agent and method system, and feedback, evidence, and decision. Around those pieces the book builds reusable objects: a claim grammar, a design-loop card, feedback and fidelity ladders, evidence chains, rejection authority, and a boundary for nondelegable judgment. The

faster-moving record of who is doing what belongs with the community now forming around this topic. The book keeps the parts meant to last.

The reader I have in mind is plural. 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. An instructor should find artifacts that make the framework teachable: cards, checklists, and a paper-to-loop exercise. 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: name a loop, judge its evidence, and state what remains a human commitment.

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 change the unit of architectural practice from the isolated artifact to the represented, instrumented, evidence-bearing design loop, and to build loops worthy of an architect's judgment.

Vijay Janapa Reddi

Acknowledgments

This lecture grew out of work and conversations across computer architecture, machine learning systems, benchmarking, and education. 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.

Any errors that remain are my own.

Table of contents

About the Author

Vijay Janapa Reddi is a professor at Harvard University whose research spans computer architecture, runtime systems, edge computing, and machine learning systems. His work connects architecture, systems, benchmarking, and education, with an emphasis on making emerging computing platforms practical, measurable, and accessible. He has helped build teaching material and textbook resources for machine learning systems, and his perspective on Architecture 2.0 is grounded in the intersection of hardware fundamentals, software interfaces, benchmark methodology, and deployed ML systems.

Chapter 1

The Moonshot for Architecture 2.0

What this chapter gives you

After this chapter you can:

- explain why "AI for architecture" means designing the design loop, not prompt-to-chip generation;
- distinguish the familiar human-carried architecture loop from an explicit, represented Architecture 2.0 loop;
- decompose a one-line design request into the architecture state it hides;
- treat efficiency as a multidimensional, loop-level property rather than a single metric;
- separate generation, prediction, and optimization as distinct roles in a design loop.

Computer architects are used to asking what architectures are needed for new forms of computation. The recent rise of AI systems has made that question urgent: what machines, memory systems, accelerators, networks, compilers, and runtime systems should support modern AI workloads? That question remains important. This book asks the reverse question: what can AI systems do for computer architecture itself?

The answer is not that a model should simply design a chip. That framing is too small and too misleading. Computer architecture is not a single act of generation. It is the discipline of turning workload intent, technology constraints, software assumptions, physical limits, and evidence into credible hardware-software systems. 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 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). Dennard scaling made that coupling favorable for a time; dark silicon, data-movement energy, warehouse scale, and carbon accounting make it 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.

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). The artifacts changed every generation, but the loop that produced them, propose, model, measure, and commit, stayed largely the same. 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. 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. Architecture 2.0 keeps that discipline but moves it one level up: the data, feedback, evidence, and rejection processes that produce the artifact must themselves be represented and designed.

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

1.1 Ask What AI Can Do for Architecture

The phrase “AI for architecture” can be read in a shallow way: use 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?), picks a cycle-level simulator and 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.

In this book, AI does not name a single model or an autonomous designer. It names bounded method roles inside an architecture design loop: generating candidate artifacts; predicting behavior, cost, or risk before expensive evaluation; optimizing which candidates or fidelity levels to evaluate; critiquing and repairing assumptions, constraints, and artifacts; verifying constraints and evidence chains; and coordinating state, tools, feedback budgets, and human review. A role is credible only when it has an architecture object, a permitted action interface, a feedback source, a rejection condition, and a decision owner.

> **!** Central question
>
> How should architects design the loops that synthesize computing systems?

Here, *computing system synthesis* does not mean only logic synthesis or high-level synthesis. It means the broader architecture act of turning intent, constraints, representations, tools, methods, feedback, evidence, and human judgment into defensible hardware-software system designs.

> **Architecture 2.0.** Architecture 2.0 is the discipline of designing, representing, instrumenting, and governing the architecture design loop itself so that AI systems can play bounded method roles: generation, prediction, optimization, critique and repair, verification, explanation, and coordination, all under explicit evidence standards and human decision authority.

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 human 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 instructor or reviewer should be able to ask what would reject the result, not only what result was produced. These are the reusable artifacts the book is meant to provide: an ontology, a design-loop card, feedback and fidelity ledgers, method-role distinctions, negative-trace language, and a boundary for what the architect still owns.

1.2 From Architecture 1.0 to Architecture 2.0

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. The architect still owns intent, constraints, abstraction, evidence standards, rejection, and accountability. But the object being designed now includes the loop itself. 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.

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.

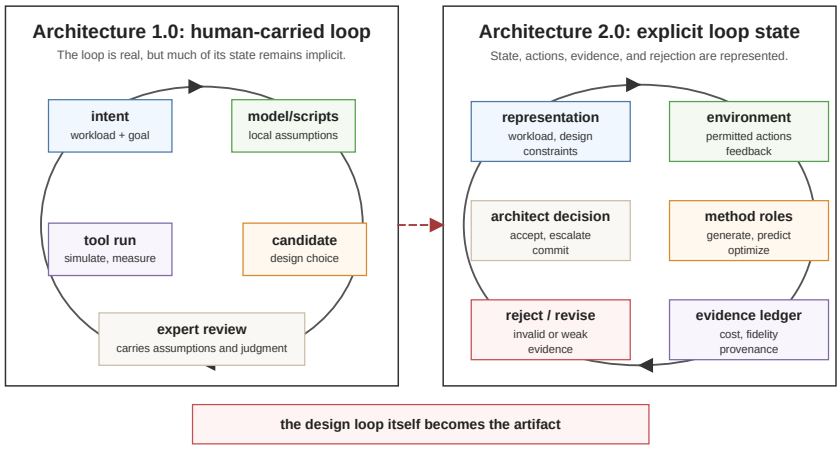


Figure 1.1: **The architecture design loop changes form:** In Architecture 1.0, intent, models, candidates, tool runs, and expert review already form a powerful human-carried circular loop. In Architecture 2.0, the loop state becomes represented, tool interfaces define permitted actions and feedback, method roles act inside the represented design space, evidence is preserved, weak outputs can be rejected, and the architect decides whether to accept, revise, escalate, or commit.

This is why the subtitle uses the phrase *agentic design loops*. The agent is not the whole system. The loop is the system. The agentic parts must be embedded in representations, environments, evidence chains, and human decision points.

1.3 The Hardware Foundation Model Moonshot

Before introducing the prompt, let us define what the word *moonshot* is doing here. 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.

The term has useful historical weight. Apollo was a literal moonshot: a national-scale program that combined a clear objective, new technology, systems engineering, software, manufacturing, operations, and risk acceptance to land people on the Moon (National Aeronautics and Space Administration, 2008). The Human Genome Project was not a single instrument or algorithm; it was a coordinated scientific infrastructure effort that produced a reference sequence and changed what biology could measure and share (National Human Genome Research Institute, 2025). DARPA's Grand Challenge did not solve autonomous driving in its first event, but it created a task, a public test, a failure surface, and a community that could iterate (Defense Advanced Research Projects Agency, 2014). AlphaFold is a later scientific AI example: it did not make biology easy, but it showed how representation, data, learning, and evidence could change the feasible boundary of protein-structure prediction (Jumper et al., 2021).

These examples have different politics, budgets, and technical domains, so the analogy should be used carefully. 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 an agent 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, teachable 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. Figure 1.2 shows that pattern in the architecture vocabulary used here.

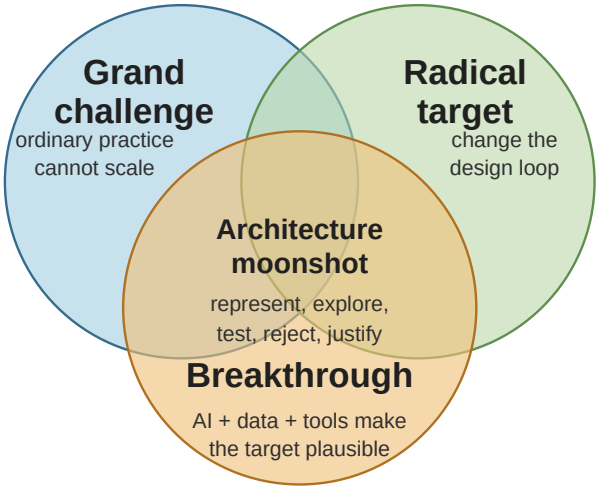


Figure 1.2: **A moonshot needs three conditions at once:** An architecture moonshot is not just a big challenge, a radical target, or a promising technology. It sits at the intersection of a grand architecture challenge, a design-loop target that changes how work is represented and evaluated, and an enabling AI/data/tool breakthrough that makes the target plausible enough to study.

> **Architecture moonshot.** An architecture moonshot is an aspirational target at the intersection of three conditions: a grand architecture challenge that ordinary practice cannot scale to meet, 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.

The architecture version 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 the loop that represents these choices, calls the right tools, preserves evidence, records failures, and gives humans rejection authority. The enabling shift is the arrival of AI methods, architecture datasets, executable environments, and tool interfaces that make pieces of that loop plausible enough to study. Some pieces are already in production, from reinforcement learning that searches the physical-design flow at commercial scale to learned circuit generators whose output has shipped in silicon (Synopsys, 2023; Roy et al., 2021). Chapter 9 returns to those examples as loop-pattern cases. The moonshot is not that these pieces exist; it is assembling them into one represented, evidence-bearing loop that a human can still govern.

Lighthouse prompt

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

This prompt is the moonshot in compact form, and it is not "type a prompt and get a chip." It is the harder target of making enough architecture state explicit that a compound system of models, tools, data, feedback, and human judgment can explore, reject, and justify design choices across the hardware/software stack. The prompt is a forcing function: it is designed to expose what a credible loop would need to know, not to claim that the field can already solve the request end to end.

Before unpacking the prompt, fix the term. 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.

This is the lighthouse prompt for the book. It is intentionally short. It looks like a request that a powerful model might eventually receive. But the sentence is not interesting because it is short. It is interesting because of the architecture state it hides. Figure 1.3 keeps the prompt visible as an object of analysis: the top panel is the request, 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.

XRBench 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. A 64-bit RISC-V contract 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. A 3 W TDP target and 3 nm-class low-power 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; 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.

One way to read the prompt 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 version is different. The left side

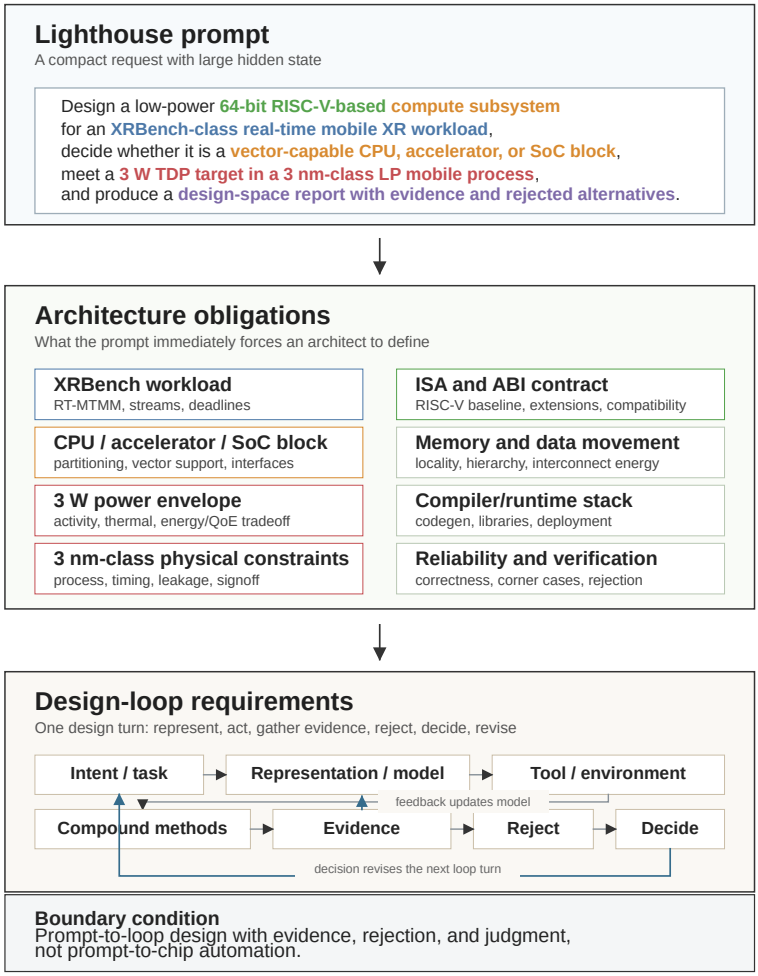


Figure 1.3: **Prompt-to-loop design:** A compact request for a 64-bit RISC-V-based compute subsystem for XRBench-class mobile XR workloads under a 3 W, 3 nm-class low-power mobile envelope spans workload definition, ISA, microarchitecture, accelerator/SoC partitioning, memory, software, tools, verification, deployment, and evidence. The important claim is not prompt-to-chip automation; it is prompt-to-loop design for representing, exploring, testing, rejecting, and justifying architecture choices.

includes workload traces, specifications, RTL or IP blocks, simulator configurations, process and library assumptions, verification logs, papers, and prior designs. The middle cannot be a language model alone. It must be a hardware architecture foundation model: 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

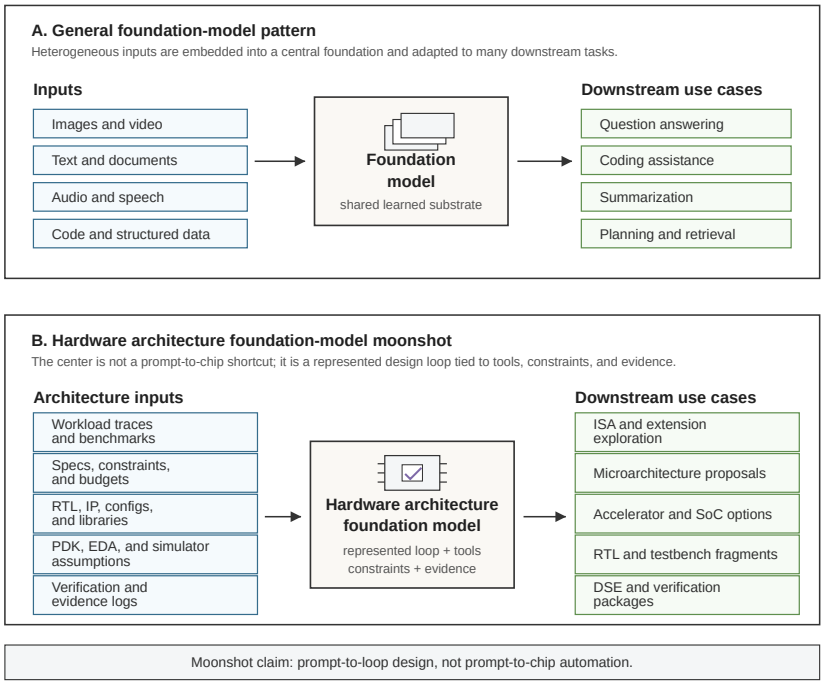


Figure 1.4: **A hardware architecture foundation model, not a prompt-to-chip shortcut:** In a general AI stack, heterogeneous inputs feed a foundation model and downstream use cases fan out. In the hardware-architecture version, the inputs are architecture artifacts and evidence, the center is a represented design loop tied to tools and constraints, and the downstream use cases include ISA exploration, microarchitecture development, RTL and testbench generation, design-space reports, verification packages, and deployment decisions.

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.

The prompt should be treated as a moonshot, not as a current capability claim. A present-day agent 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, QoS target, latency deadline, memory traffic, software pipeline, and drift assumptions.	Trace provenance, benchmark version, workload coverage, and rejection when results miss real-time behavior.
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, accelerator, tightly coupled unit, SoC block, memory hierarchy, interconnect, chiplet boundary if any, and integration point.	Design-space comparison and rejection of candidates that only win by moving cost, bandwidth, energy, or complexity elsewhere.

Prompt fragment	Architectural decisions	Evidence or rejection need
3 W, 3 nm-class low-power envelope	Power, voltage/frequency, thermal, area, process/library assumptions, RTL feasibility, EDA constraints, and physical signoff path.	Power-model provenance, synthesis or timing feedback, sensitivity analysis, and rejection when stronger physical evidence violates the envelope.
Design-space report with evidence and rejected alternatives	Alternatives, Pareto fronts, assumptions, uncertainty, verification plan, negative traces, and human decision points.	Evidence chain, coverage, rejected candidates, reproducible artifacts, and explicit rejection authority before higher commitment.

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

The prompt also clarifies 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.

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.

Those roles overlap, but none is sufficient alone. The center is closed-loop architectural synthesis: 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.

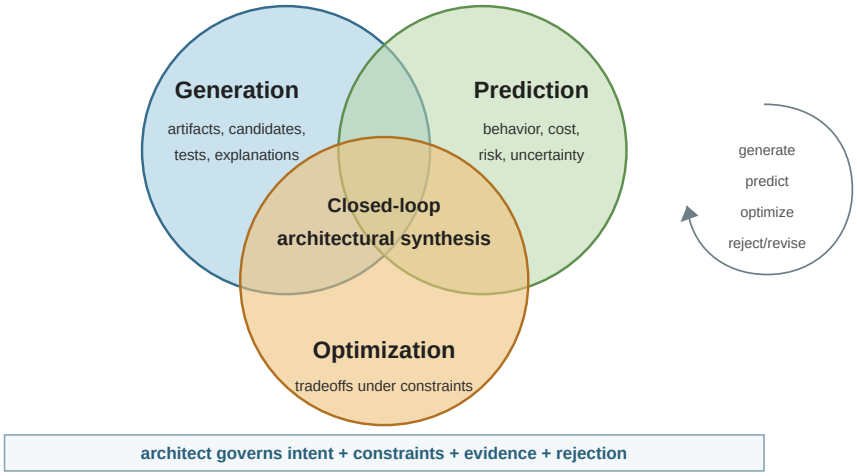


Figure 1.5: **Architecture development is broader than generation:** Generation proposes artifacts and candidates, prediction estimates behavior and risk, and optimization searches tradeoffs under constraints. Architecture 2.0 is concerned with the closed loop in the middle, governed by evidence, rejection,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

1.6 Efficiency as the North Star

Efficiency is the practical north star because architecture is the discipline of turning scarce resources into useful work through durable hardware/software interfaces. 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).

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 records, negative traces, or AI-assisted search.

Benchmarks show the same pressure. MLPerf was designed to make machine learning performance claims reproducible across systems (Mattson et al., 2020). MLPerf Inference then made deployment scenarios, latency, throughput, accuracy, and power part of the comparison problem rather than treating "fast inference" as one scalar claim (Reddi et al., 2020, 2021). This is the architectural lesson: efficiency is not one number. It is a structured claim about useful work under constraints.

> **Multidimensional efficiency.** Multidimensional efficiency is the claim that 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. 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.

Table 1.2 summarizes the dimensions that Chapter 1 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.
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.

Dimension	Efficiency question	Why it complicates the loop
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

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, agent 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 workflow 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 agentic, the architect's responsibility moves upward. The architect must frame the task, choose representations, define environments, set evidence standards, inspect negative traces, maintain rejection authority, and own the final 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 data, representations, and world models agents would need. 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. The appendices then give a bootstrap workflow and a reusable design-loop card.

Chapter 2

When Classical Architecture Design Loops Break

> **What this chapter gives you**
>
> After this chapter you can:
> - recognize design-loop pressure, the scissors gap between choices and trusted feedback, in a real architecture setting;
> - explain why the bottleneck is trusted feedback, not idea generation;
> - point to the design and verification costs that make high-fidelity feedback scarce;
> - connect each pressure source to a loop field that can no longer remain implicit;
> - identify where generic AI assumptions break at architecture boundaries.

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 low-power 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 itself become an object of design.

> **Design-loop pressure.** Design-loop pressure is the condition in which 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.

The chapter is therefore not a technology-trend survey. Specialization, chiplets, software velocity, memory movement, EDA, physical design, and verification matter here because they create a common failure mode: 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 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 itself becomes the architecture object.

2.1 Classical Loops Already Use Feedback

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 workflows (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. 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.

Consider a familiar cache-hierarchy exploration. The architect defines a workload set, chooses candidate cache sizes, associativities, replacement policies, and prefetching options, runs a simulator, studies miss rates, latency, bandwidth, energy, and area proxies, rejects poor candidates, and keeps iterating. 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.

This loop is powerful because it does not require perfect automation. It combines formal models, approximations, domain knowledge, and review. It also has an implicit contract: 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 contract holds, the loop works. When the contract breaks, the team may still generate ideas, but it cannot evaluate and reject them fast enough to make credible progress.

2.2 Cadence and Gates Manage Risk

Industrial architecture practice has long treated cadence as a design object. 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.

Other fields teach compatible lessons. EDA timing closure teaches 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, benchmark governance depends on maintained rules and comparability contracts, and control, operations, and systems engineering add the same warning in different language. A loop without observability, a decision policy, and escalation gates is not a trustworthy 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.

This is not the first time architecture has had to redesign its loop. Table 2.1 gives a few examples. 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 2.1: **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 teachable 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 implementation-cost assumptions became part of the 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.
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.

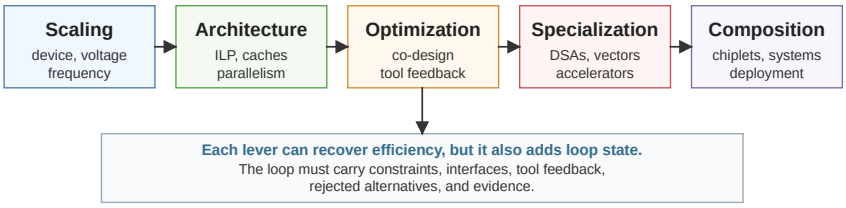


Figure 2.1: **Architecture levers add state:** Scaling, microarchitecture, optimization, specialization, and composition create new opportunities for efficiency, but each also adds constraints, interfaces, tool feedback, rejected alternatives, and evidence that the loop must carry.

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

Architecture advances 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.

The end of Dennard-style scaling and the limits of multicore scaling are not the only reasons for this pressure, but they 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. The opportunity inside the new golden age is to extend 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

Specialization is attractive because efficiency is multidimensional, the point Chapter 1 made its north star. 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 (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 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. Suppose a team is exploring only a narrow slice of the lighthouse prompt: an accelerator and memory subsystem for one XRBench-class workload family. 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.

> **Field note: the small count is already a schedule**
>
> A count such as 12,960 looks harmless on the page because multiplication is cheap. It becomes an architecture problem as soon as each surviving state needs a sample: one proxy estimate, one simulator run, one synthesis check, one review meeting, or one signoff path. The loop therefore has to decide which states are screened analytically, which are replayed in simulation, which are escalated to stronger tools, and which rejected regions are worth preserving so the team does not rediscover the same dead end.

Table 2.2 grounds that pressure in concrete architecture settings. The examples are not meant to be a single measurement scale; they show that mapping, DSE, packaging, design cost, and verification each impose a different kind of loop burden.

Table 2.2: **Design-loop pressure has source-backed scale anchors:** These examples ground the chapter's qualitative argument in benchmark scale, accelerator search, chiplet integration, design-cost, and verification-effort signals.

Loop example	Scale anchor	Loop lesson
DNN accelerator mapping	Timeloop's mapspace for a 7D CNN on a 4-tiling-level architecture includes loop permutations, factorization choices, and level-bypass alternatives; the unconstrained expression contains $(7!)^4 \times (2^4)^3$ multiplied by co-factor choices (Parashar et al., 2019).	Mapping is itself a combinatorial problem; architecture evaluation depends on the mapper and its constraints.
DNN accelerator DSE	MAESTRO reports a design-space exploration over 480M candidate designs, identifying 2.5M valid designs at an average rate of 0.17M designs per second (Kwon et al., 2019).	Validity, pruning, and cost models are part of the evidence path, not implementation details.
Chip floorplanning	Google TPU-block floorplanning experiments involved up to a few hundred macros and millions of standard cells; human designers iterated for months with EDA feedback taking up to 72 h, while a reported learned method generated comparable placements in under six hours, a claim later disputed on baselines and reproducibility (Mirhoseini et al., 2021; Cheng et al., 2023).	High-fidelity feedback can be slow and multiobjective; generation is useful only when tied to tool feedback and rejection criteria.

Loop example	Scale anchor	Loop lesson
Tensor-program tuning	AutoTVM 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.

2.5 Specialized Hardware Needs a Software Loop

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 2.1 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 2.2 pushes the point further: billion-scale operator search is already a software-side obligation of specialization. Systems such as Halide and MLIR make scheduling, lowering, and intermediate representations central parts of the performance loop (Ragan-Kelley et al., 2017; Lattner et al., 2020).

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.

2.6 Software Changes Faster than Silicon

Specialization depends on stable enough targets. But modern software stacks move quickly. AI models change. 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.

Compiler 2.0 is 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.

MLPerf is a useful example because 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.

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. XRBench provides a benchmark suite for extended-reality machine-learning workloads (Kwon et al., 2023), but 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

Architecture does not sit above physical reality. It is the layer where software intent, hardware mechanisms, and physical constraints become one design problem.

Data movement is the clearest example. 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. It is a reminder that an efficiency claim can be rejected by any term the loop failed to represent. A candidate that reduces arithmetic but increases memory movement, interconnect traffic, control overhead, or leakage has not necessarily improved the system.

Interconnect has the same character. 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, leakage, signoff, and test are not late implementation details when they can overturn an architectural choice. 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 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 2.2 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; the plot is a historical order-of-magnitude anchor, not the target-node model.

2.8 Engineering Cost Creates the Scissors Gap

The central pressure is 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.

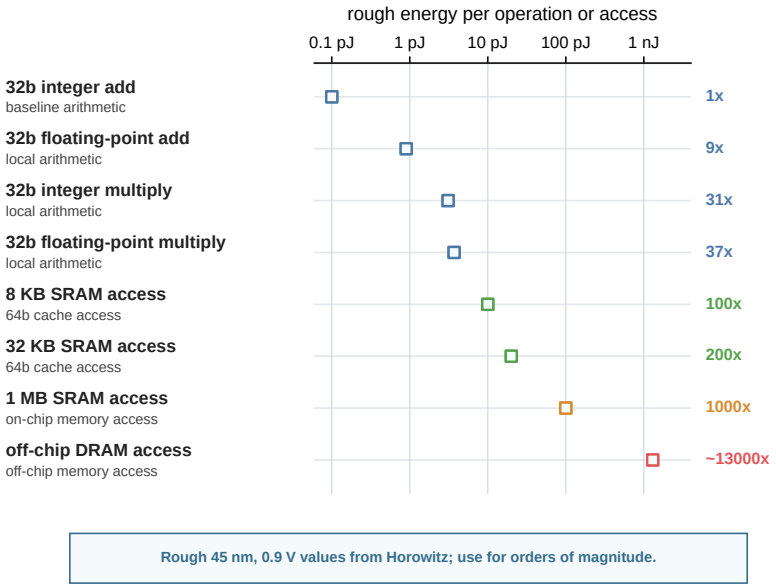


Figure 2.2: **Data movement can dominate arithmetic energy:** Rough Horowitz 45 nm energy values show why architecture loops must represent locality, buffering, precision, scheduling, and memory hierarchy rather than counting arithmetic alone. The values are order-of-magnitude teaching anchors, not current-node device estimates.

Figure 2.3 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.

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-18x 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.

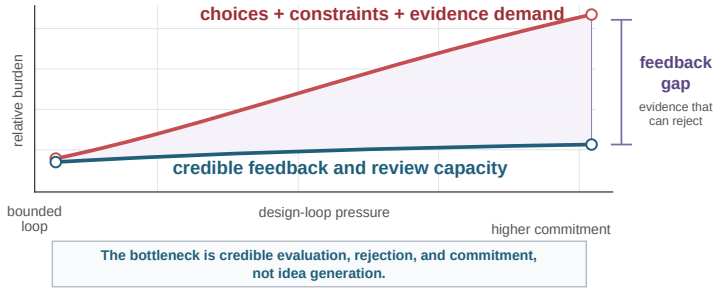


Figure 2.3: **The scissors gap is a feedback gap:** Architecture pressure rises when design choices, constraints, and evidence demand grow faster than unaided manual coordination and verification capacity. The widening region is the gap that Architecture 2.0 tries to make explicit and manageable.

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 budget, not just compute cycles.

Figure 2.4 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.

This does not mean architects are ineffective. It means the unit of work has changed. An expert can reason deeply about a few candidate designs. A team can maintain a disciplined simulation and review process. But when a design space contains thousands of plausible states, when feedback arrives at multiple fidelities, and when negative results are not recorded in a reusable form, manual coordination becomes the bottleneck.

The bottleneck is trusted feedback, not ideas. Generating one more accelerator configuration, memory policy, chiplet partition, or compiler schedule is not hard. Knowing which candidates were invalid, which proxy wins disappeared at higher fidelity, which assumptions were responsible for the result, and which decision a human should commit to is the hard part.

This is the reason the scissors gap is a feedback gap rather than a creativity gap. A loop that cannot preserve failed runs, explain proxy failures, or decide when stronger evidence

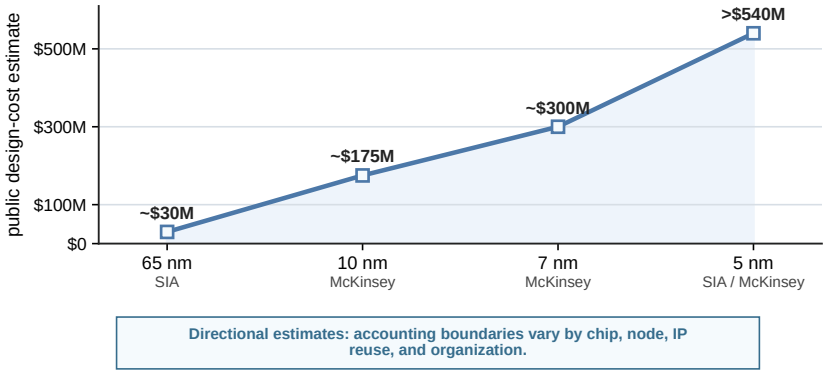


Figure 2.4: **Leading-node design-cost estimates make the scissors concrete:** Public SIA and McKinsey estimates (Semiconductor Industry Association, 2026; Bauer et al., 2020) put latest-node chip design costs from tens of millions of dollars at 65 nm to more than $500M at 5 nm. The accounting boundaries vary, but the scale makes feedback, validation, IP, tools, and human engineering effort part of the architecture loop.

is required will degrade over time: future teams will rerun old mistakes, rediscover invalid regions, and mistake more output for more architectural progress.

2.9 Feedback and Verification Become the Bottleneck

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 agent that optimizes 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 only becomes evidence when its fidelity, provenance, uncertainty, and relevance to the decision are clear.

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.

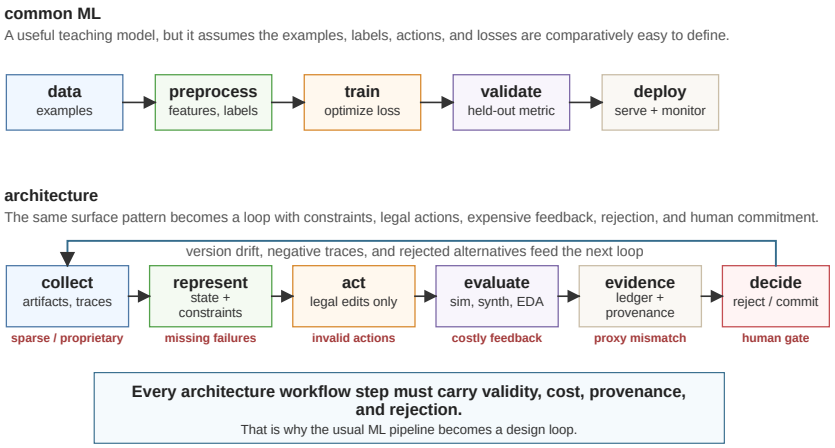


Figure 2.5: **Architecture turns the ML pipeline into a constrained loop:** A generic ML workflow can be taught as data, preprocessing, training, validation, and deployment. In architecture, the corresponding workflow must also represent constraints, legal actions, expensive tool feedback, evidence ledgers, negative traces, drift, rejection, and human commitment.

2.10 Architecture Violates Generic AI Assumptions

Many successful AI systems are built in domains with abundant data, cheap feedback, stable labels, and clear losses. Computer architecture violates that pattern at almost every boundary. Data are often proprietary, incomplete, stale, or missing the negative traces that explain why a candidate was rejected. 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 2.5 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 human commitment gate.

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 2.3 gives the checklist version of that argument.

Table 2.3: **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 negative traces.

Common AI assumption	Architecture violation	Loop implication
Abundant labeled data	Many labels are proprietary, expensive, stale, or not recorded.	The loop must build source packets, provenance, and reusable negative traces.
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.	Negative traces 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

AI becomes 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 workflow that finds a better proxy score without negative traces 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. That is the transition to the ontology in the next chapter: once the pressure is visible, the next task is to name the minimum loop state that makes AI assistance inspectable rather than merely impressive.

Chapter 3

Design Loops, Design Spaces, and Architectural Claims

> **What this chapter gives you**
>
> After this chapter you can:
> - write an architectural claim as a reviewable object: workload, baseline, design space, objectives, constraints, evidence, rejection rule, and decision owner;
> - map a paper, tool, or project onto the five-part framework;
> - distinguish an ontology from a taxonomy and say why ontology comes first;
> - judge how much autonomy a loop has actually earned.

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 field has always designed through loops of abstraction, measurement, feedback, and judgment.

What changes is the object of design. 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: that a design improves useful work, reduces energy, meets a latency target, preserves correctness, exposes a tradeoff, or changes what is possible under a workload and set of constraints. In Architecture 2.0, the architect must design the loop that produces, tests, rejects, and revises those claims. The loop itself needs a task boundary, a design space, a representation, a world model, an environment, method roles, feedback channels, evidence standards, rejection rules, and human decision points (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.

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 claim grammar and ontology: a way to name the architectural claim

being made and 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 instructor should be able to hand it to students and ask them to fill in the claim and loop for a concrete design problem. If the ontology cannot support those uses, it is only vocabulary.

3.1 The Architectural Claim Is the Unit of Review

The most common question is too coarse: can a model design hardware? The more architecture-native question is: what claim is being made, and what would 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.

> **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.

A compact way to write the review object is

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

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 rule, and H is the human or organizational decision authority. This is not a formalism for its own sake. It is a way to prevent a generated artifact from masquerading as an architectural result before the comparison, constraints, and evidence are visible.

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 rule, and decision owner 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 rule	What observation can invalidate or weaken the result?	Missed latency target, power envelope violation, invalid RTL/configuration, compiler failure, simulator mismatch, or weak coverage.
Decision owner	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.

3.2 The Design Loop Is the Unit of Analysis

Once the claim is explicit, the next question is: what design loop 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 human commitment until it produces an artifact or a revised loop.

For the lighthouse prompt, the distinction is immediate. A request for a low-power, 64-bit RISC-V-based compute subsystem for XRBench-class mobile XR 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 an agent 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

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

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 record 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.

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.

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:

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

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.

This 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 Ontology Before Taxonomy

A taxonomy groups things. It can list tasks, methods, benchmarks, tools, agent architectures, or evaluation settings. Taxonomies are useful, and this lecture 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. Agent harnesses will change. Benchmarks will change. EDA flows and simulator stacks will change. If the book is organized only around today's artifacts, it will age with them.

> **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.

The ontology asks a deeper question: what entities must exist, and how must they relate, for AI-mediated architecture work to be credible? 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. We should not first ask whether a paper uses an LLM, reinforcement learning, Bayesian optimization, a surrogate model, or a simulator wrapper. We should first ask what loop the work exposes. What is the task? What state is represented? What actions are legal? What environment returns feedback? What is the feedback budget? What evidence supports the claim? What can reject the result? What does the architect still decide? Once those questions are answered, a taxonomy of methods becomes useful. Before that, method labels can hide more than they reveal.

Adjacent fields show why this ordering matters. BERT created a general pattern for pretraining and adapting representations (Devlin et al., 2019); biomedical and clinical variants such as BioBERT, ClinicalBERT, and Med-BERT showed that domain shift forces a field to build domain-specific corpora, tasks, and representations (Lee et al., 2020; Huang et al., 2019; Rasmy et al., 2021). The architecture lesson is not simply to build a chip-specific language model. It is that a field becomes AI-addressable only when its objects of work 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, negative traces, fidelity levels, and commitment decisions. That is why this lecture starts with an ontology of the design loop rather than a catalog of current models.

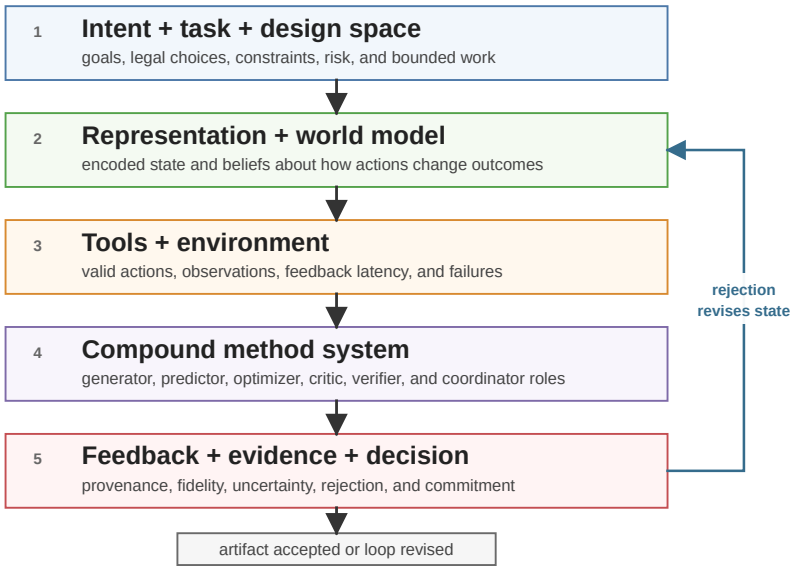


Figure 3.1: **The Architecture 2.0 ontology chain makes the loop explicit:** Intent, task, and design space define what work the loop is allowed to do; representation and world model define what the loop can know; tools and environments define valid action and feedback; compound methods act inside the loop; evidence and human decision determine whether an artifact is accepted, rejected, or used to revise the loop.

3.5 The Compact Five-Part Framework

The full ontology can be read as the loop in Figure 3.1.

For practical use, this book compresses the ontology into five pieces.

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 are *tools and environments*. 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 is the *compound agent and method system*. 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, and decision*. Feedback is any signal returned by the loop. Evidence is feedback that has been tied to provenance, fidelity, assumptions, uncertainty, coverage, 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 workflow before trusting an Architecture 2.0 claim.

Table 3.2: **The framework becomes a checklist when each loop field is explicit:** A project is easier to review when it names the task, representation, environment, method role, feedback, evidence, rejection rule, 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.
Tools and environment	What actions are legal, what feedback is returned, and what failures are observable?	Simulator, cost model, workload harness, and invalid-configuration checks.

Framework piece	Reader question	Lighthouse instance
Compound method system	Which roles generate, predict, search, critique, verify, call tools, or coordinate?	Candidate generator, surrogate or search method, verifier, evidence writer, and human reviewer.
Feedback, evidence, and decision	What supports the claim, what can reject it, and what remains a human commitment?	Pareto evidence, sensitivity checks, negative traces, rejection rules, and final architectural judgment.

This checklist is intentionally stricter than many current demonstrations. A system can be interesting without answering every row, but an architectural claim becomes stronger when the loop state, evidence, rejection authority, and human decision are visible. The point is not to make every project bureaucratic. It is to make claims comparable, reproducible, and reviewable.

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 agent 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.

Generic term	Architecture translation	Example artifacts or observations	What can reject it
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.	PPA, tail latency, power envelope, reliability, carbon, cost, evidence budget.	Proxy gaming, lost Pareto tradeoff, or missing human decision rule.
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 an existing workflow. It is about designing the workflow so that AI participation is bounded, observable, and accountable.

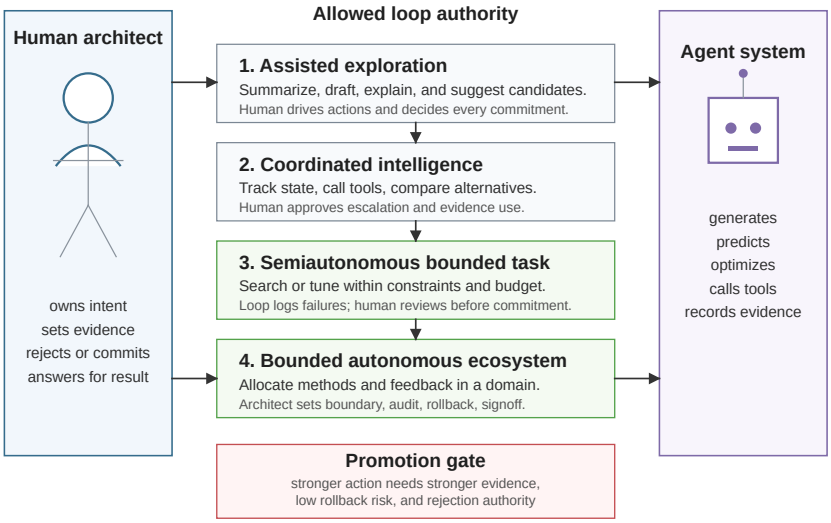


Figure 3.2: **Autonomy is earned by the loop:** Human architectural responsibility and agentic loop action meet at bounded authority. Higher autonomy is not a personality trait of a model; it is a property of a bounded loop with explicit actions, feedback, evidence, rollback paths, rejection authority, and human commitment boundaries.

3.6 Autonomy Is Earned, Not Declared

It is tempting to ask whether Architecture 2.0 systems are autonomous. That question is too coarse. 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 3.2 shows four stages of allowed loop authority. The point is not that the agent gradually replaces the human architect. The point is that each stage grants the agent a larger role only when the loop also defines the allowed action space, feedback budget, evidence standard, rollback path, and human commitment boundary. The human and agent are both visible because Architecture 2.0 is a shared loop with asymmetric responsibility: the agent may act inside the loop, but the architect owns the boundary conditions.

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 rules.

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.

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.

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.

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 agent 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.

QuArch illustrates the representation problem from one angle (Prakash et al., 2025b). A question-answering dataset can help evaluate and improve the architectural knowledge of language models, but it cannot by itself represent all of the state needed for system synthesis. It can expose what models know about concepts, but architecture loops also need tool state, workload provenance, failed candidates, constraints, and evidence trails. 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 agent 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.

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 agent may optimize a proxy that the architect cannot audit. If the environment does not log tool versions, seeds, workload revisions, and failed runs, the feedback may not become evidence.

ArchGym 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 not only that one can connect ML methods to simulators. The larger lesson is that architecture research needs environments in which tasks, action spaces, feedback, and comparisons are explicit. 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 important question is not which method is fashionable. The important question is which role the method plays in the loop, what state it consumes, what action it takes, what feedback it receives, and what evidence 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.

For example, suppose an agent proposes a vector-capable compute subsystem for a mobile XR workload and reports that it meets the 3 W target. That statement is not evidence unless the loop records what workload was used, what input distribution was assumed, what power model was used, what process assumptions were available, what software stack was compiled, what alternatives were rejected, how uncertainty was handled, and what would cause the result to be discarded. The same number can mean different things at different fidelity levels. A proxy estimate, a cycle-level simulation, a synthesis result, a post-layout estimate, and silicon measurement do not carry the same authority.

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.

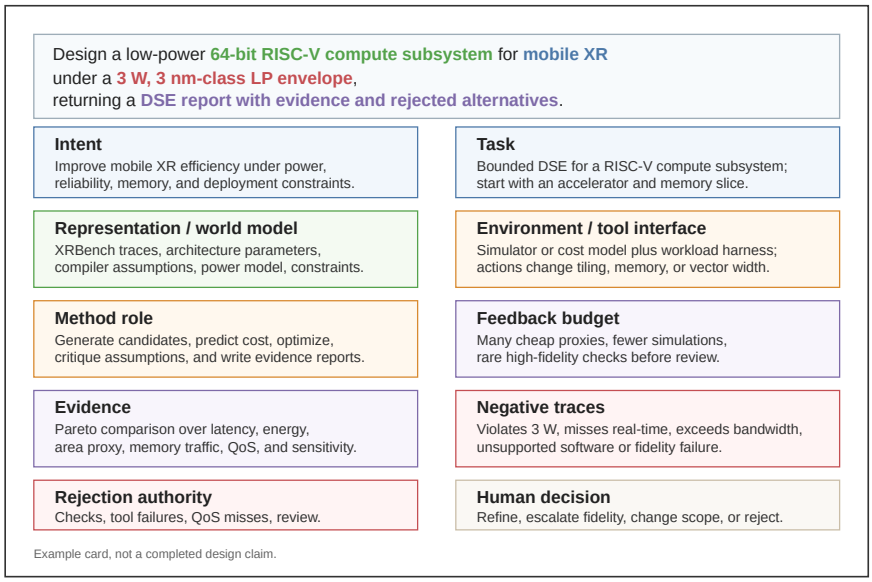


Figure 3.3: **A filled design-loop card turns a prompt into reviewable state:** The lighthouse prompt becomes explicit loop state: intent, task, design space, representation, environment, method role, feedback budget, evidence, negative traces, rejection authority, and human decision.

This distinction between feedback and evidence is one of the main safeguards against hype. Architecture 2.0 is not credible because an agent 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.

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 workflow. It asks for the loop, not only the result.

Figure 3.3 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.

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.

Appendix B gives the full 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 in order. Chapter 4 focuses on representations and world models: what architecture data must encode before AI systems can reason about it. Chapter 5 focuses on tools and environments: how simulators, compilers, EDA flows, benchmarks, and deployment systems become action settings. Chapter 6 focuses on method roles: generation, prediction, optimization, critique, verification, and coordination under feedback constraints. Chapter 7 focuses on evidence, verification, trust, 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.

The ontology is not a guarantee of correctness. It is a way to expose what must be represented, measured, checked, rejected, and decided. That is why it can outlast current models and tools. The lasting question is not which agent wins. The lasting question is how architects design loops that can synthesize systems credibly.

> **!** Architect's checkpoint
>
> Before trusting an Architecture 2.0 claim, ask:
> - Can I state it as a claim tuple: workload, baseline, design space, objectives, constraints, evidence, rejection rule, and decision owner?
> - Which loop produced it, and what could reject its result?
> - What does the architect still decide, and at what commitment level?

Chapter 4

Data, Representations, and Architecture World Models

What this chapter gives you

After this chapter 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 representation debt and the missing negative traces in a workflow;
- tell a representation apart from a world model.

Chapter 3 defined the ontology. This chapter deepens its first technical piece: representation and world model. 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 workload, constraint, tool option, rejected candidate, or uncertainty is invisible, a more capable model may simply move faster in the wrong direction.

The simple version is that architecture needs more data. That statement is true, but incomplete. Architecture does not need data in the same way a web language model needs text. It needs structured records of architecture work: what was intended, what was tried, which tool configuration was used, which workload and software stack were assumed, what failed, what evidence was accepted, and what the architect decided. The durable question is therefore not only how to collect architecture data. It is how to represent architecture work so that a loop can act on it and a human can audit it.

4.1 Why Architecture Data Is Not Web 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.

The lighthouse prompt makes the difference concrete. A request for a low-power RISC-V compute subsystem for an XRBench-class mobile XR workload cannot be answered from public text alone. XRBench gives a workload anchor (Kwon et al., 2023), but the loop still needs scenario selection, input distributions, model versions, sensor streams, latency targets, software assumptions, compiler choices, power models, memory behavior, process constraints, and deployment assumptions. The public benchmark is a start; it is not the whole design state.

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. 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.

> **Architecture representation.** An architecture representation is the record of workload state, design state, tool state, constraints, assumptions, evidence, uncertainty, negative traces, and decisions that a loop can read, compare, change, replay, and audit.

The token scale makes the contrast sharper. A broad corpus built from roughly fifty years of public computer-architecture, systems, and EDA literature might only be on the order of 10^9 tokens. A rough receipt is simple:

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

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.

Figure 4.1 puts that estimate on the same log-scale axis as three public AI-data anchors: Chinchilla's 1.4-trillion-token training run, Meta's report that Llama 3.1 405B was

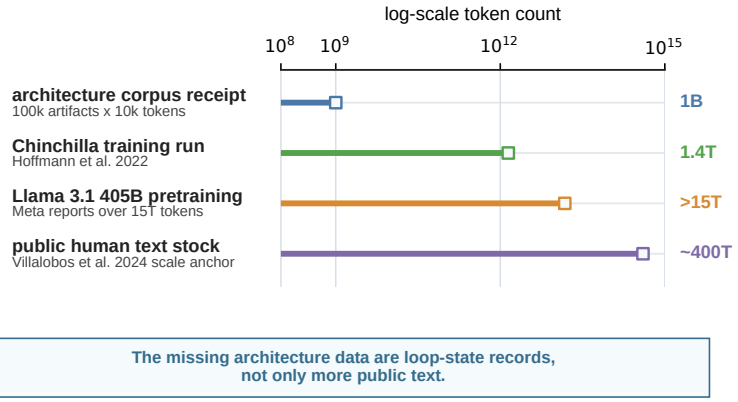


Figure 4.1: **Architecture text is small by web-scale standards:** A rough public architecture, systems, and EDA corpus receipt on the order of 10^9 tokens is far below trillion-token language-model training runs and hundreds-of-trillions-token public-text stock estimates. The important missing data are loop-state records: traces, tool logs, negative results, assumptions, and rejected candidates.

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 4.1 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.

Table 4.1: **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.

Receipt	Current value	What it supports	Caveat
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.
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 and representation design at the same time: more artifacts, but also better records of why those artifacts are credible.

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 pays off, and it illustrates the limit this chapter presses: it maps text to text, so it addresses the data-ingestion problem, not the 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.

Benchmark lineages such as SQuAD and GLUE teach a related lesson (Rajpurkar et al., 2016; Wang et al., 2018). They helped turn language understanding into repeatable evaluation objects: common inputs, common metrics, and comparable results. Architecture needs shared evaluation objects too, but 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.

4.2 Sample Cost Is Architecture Data

In many benchmark settings, a sample is treated 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.

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

$$C_{\text{sample}} = \langle C_{\text{setup}}, C_{\text{tool}}, C_{\text{license}}, C_{\text{compute}}, C_{\text{human}}, C_{\text{fidelity}}, C_{\text{opportunity}}, C_{\text{risk}} \rangle.$$

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, 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 4.2 is therefore not a tool taxonomy. It is a representation checklist: if the row changes, the loop must record different state.

Table 4.2: **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 teaches	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.
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.

Feedback source	Latency / cost intuition	What it teaches	Record for reuse
RTL, gate, or EDA feedback	Hours to days when synthesis, timing, power, or physical 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.

The timing side 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

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

Table 4.3 gives the intuition for a 1 GHz target. 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 4.3: **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 MHz	100 MHz
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 $5 \times 4 \times 6 = 120$ candidates 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., 2023), which only sharpens the point. The architecture lesson is broader than placement. When samples are expensive, the dataset must record what each sample cost, what region of the space it informs, and what it rules out.

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

An architecture description is a boundary object. 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 agents or methods to modify without breaking the design contract.

At minimum, an architecture description should say what is being described, what hierarchy it assumes, what timing model or abstraction level is being used, what memory semantics matter, what workload scope is in bounds, what constraints must hold, and which tools can consume the description. 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.

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 Representation Debt and Structured Design Data

Architecture teams accumulate representation debt. The debt 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 the debt 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 debt 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.

Field note: the result no one could rerun

A team reports a strong design-space result, then is asked to reproduce it six months later. The winning configuration survives in a plot, but the simulator version has moved, a default flag has changed, the workload trace has been re-cut, and the script that generated the run lived on a laptop that has since been reimaged. The number was real; the loop that produced it was not recorded. The fix is not heroics at reproduction time. It is recording the tool version, seed, flags, workload revision, and exact command as the run happens, so the result stays auditable after the people and machines move on.

Table 4.4 gives common sources of representation debt. 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 4.4: **Architecture artifacts become useful data only as 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

QuArch is a useful stress test for this chapter because it starts from a real need: evaluating architecture knowledge in AI agents. A question-answering dataset can turn part of the literature into a structured evaluation object (Prakash et al., 2025b). 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 agents if the agents lack basic architectural knowledge.

But QuArch also exposes the limit of paper-derived data. Papers preserve accepted claims 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

> **Architecture world model.** An architecture world model is the loop's belief about how actions change architecture state and outcomes, including dynamics, costs, constraints, invalid-action rules, uncertainty, and decision policies.

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 4.2 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.

There are several kinds of architecture world models. 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 a human review policy for high-commitment decisions.

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.

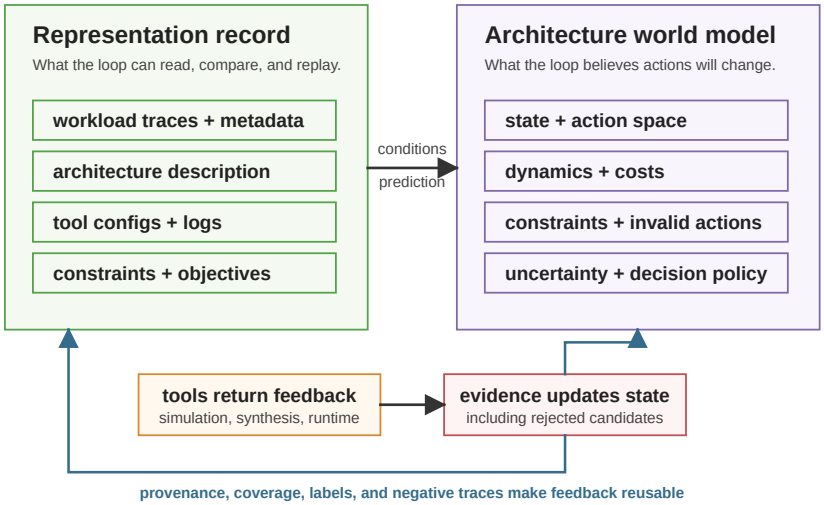


Figure 4.2: **A world model connects artifacts to valid architectural action:** A representation record captures what the loop can read, compare, and replay; a world model captures what the loop believes actions will change. Feedback becomes reusable only when provenance, coverage, labels, and negative traces are recorded.

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.

4.7 Provenance, Coverage, Labels, and Negative Traces

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 teaches the loop where the boundary lies.

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 4.5 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 4.5: **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
Invalid configuration	Candidate violates constraints, tool schema, or interface contract.	Shrink 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. A failed result without context is not very useful. The loop needs to know which workload, tool version, seed, parameters, constraints, and fidelity level produced the failure. It 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

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 teach the loop a biased view of the space.

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 rules. 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?

Chapter 5 takes the next step. Once state is represented, tools can become environments. They define what actions are legal, what feedback is returned, how expensive evaluation is, and what evidence a loop can produce.

Chapter 5

Architecture Environments and Tool Interfaces

What this chapter gives you

After this chapter you can:

- turn a tool into an environment with an explicit action, observation, cost, and rejection contract;
- read a result by naming the environment that produced it;
- reason about feedback latency and fidelity as an economy of evidence;
- detect simulator mismatch and proxy gaming before they mislead the loop.

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. They define what actions are legal, what observations are returned, how expensive feedback is, which assumptions are baked in, what state is logged, and what can reject a candidate before it wastes more effort.

> **Architecture environment.** An architecture environment is the explicit action boundary around tools: it specifies legal actions, observations, feedback costs, assumptions, logged state, provenance, and rejection rules for an architecture design loop.

That shift 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 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.

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.

The usual way to describe this is 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 emphasizes measurement, abstraction, and careful comparison (Hennessy and Patterson, 2017). Architecture 2.0 keeps that tradition, but it asks for one more layer of explicitness: the tool interface itself must be part of the design object.

This matters because agentic 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 rules. Once those are visible, method claims become easier to compare.

5.2 Interfaces Are Action Boundaries

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, 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.

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 agent 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.

Interface	What it makes actionable	Evidence it makes interpretable	Failure if hidden
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, leakage rules, and submission constraints.	Coverage, reproducibility, benchmark validity, and claim scope.	The loop overfits a stale or leaky benchmark slice.
Telemetry and deployment schema	Live workload mix, 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 lighthouse prompt crosses nearly every row of the table. The phrase "64-bit RISC-V-based" invokes an ISA contract. "Vector-capable" invokes compiler and runtime obligations. "XRBench real-time mobile XR" invokes a benchmark harness, workload distribution, and quality-of-service target. "Low power" invokes power modeling, physical constraints, and eventually EDA or silicon evidence. The environment for this prompt is therefore not one API. It is a bundle of interfaces that must remain coherent as the loop proposes, checks, rejects, and revises candidates.

Concrete tools instantiate that bundle 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.

Even the rows in this table hide internal ladders. 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.

This arrangement is not hypothetical. Production electronic-design-automation systems already treat the synthesis and place-and-route flow as a search environment: the action space is a set of tool directives and floorplan parameters, the transition is a full tool run, and the reward is the resulting power, performance, and area. Synopsys reported that one such reinforcement- learning system reached its first hundred commercial tapeouts by 2023 (Synopsys, 2023), and Cadence describes a comparable loop driving the flow from RTL to layout (Cadence Design Systems, 2021). The durable point is not which vendor leads, since the products will be renamed and replaced. It is that the EDA flow becomes an environment in exactly the sense this chapter requires once its directives are an action schema, its reports are observations, and its high-latency, high-cost runs set the feedback budget. The architect's work then moves from running the flow by hand to specifying the environment the loop searches and the rejection gates that bound it.

5.3 The Architecture Environment Abstraction

The environment is executable, not merely descriptive. 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. Figure 5.1 shows the basic shape.

The abstraction can be described as a contract. The contract does not require one software framework or one programming language. It requires that the loop make a small set of obligations explicit. Table 5.3 lists the main fields.

Table 5.3: **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?

Field	What it defines	Question it answers
Constraints and validity	Type checks, feasibility rules, physical limits, software compatibility, and invalid-action handling.	What makes a candidate illegal before performance is considered?
Feedback budget	Cost, latency, fidelity, determinism, 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 rule	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.

For 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 is useful because it makes the environment idea concrete in the architecture domain. It frames architecture design as a gymnasium for machine-learning-assisted design, with agents interacting with architecture environments through defined interfaces (Krishnan et al., 2023). 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).

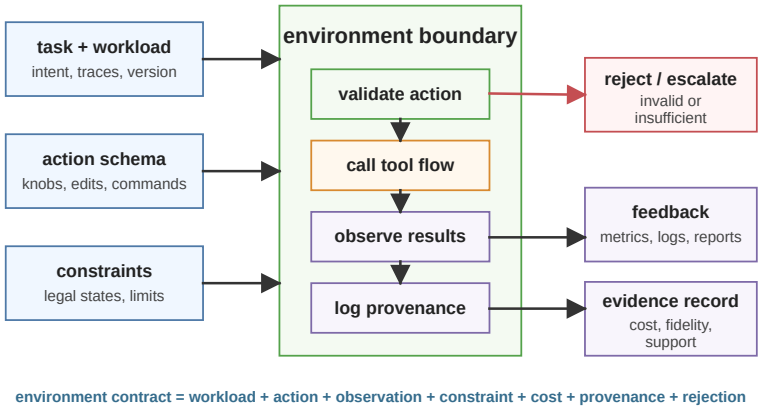


Figure 5.1: **A tool becomes an environment when its contract is explicit:** A credible environment does more than call a simulator or tool. It defines workload state, legal actions, observations, constraints, cost, provenance, feedback, invalid-action semantics, and rejection paths.

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 agents or methods were compared under the same conditions. That makes method claims less anecdotal.

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.

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

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. Benchmarks such as MLPerf did not become useful only because they named workloads. They also created rules, versions, metrics, submissions, and comparison conventions that helped a community interpret results (Mattson et al., 2020). Architecture 2.0 needs a similar instinct at the loop level. 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.

It is useful to reserve the word *harness* for this larger object. 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. An agentic 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 environment record 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.

Composability also changes how we interpret agentic 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. 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

The hardest environment 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 creates an economy of evidence. Cheap feedback buys breadth and pruning; expensive feedback buys stronger rejection authority. Figure 5.2 shows the basic pressure: as feedback moves toward implementation and deployment, the loop usually spends more time per sample, gains less freedom to explore, and needs 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. 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 is a human commitment, not merely a more accurate number.

The same pressure can be expressed as a design checklist. Table 5.4 asks 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.

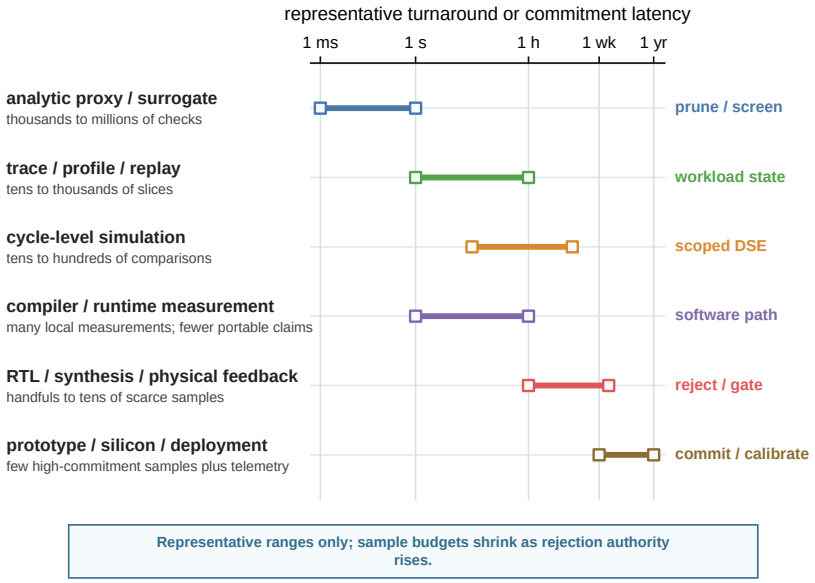


Figure 5.2: **Feedback fidelity changes the economics of search and commitment:** Low-cost feedback can screen many candidates, but implementation-adjacent feedback is scarce and more authoritative. The vertical ordering reflects rejection authority more than strict latency; the point is the shrinking sample budget, not the exact wall-clock value for any particular project.

Table 5.4: **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.	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.
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.

Feedback regime	Cost / latency intuition	Sample budget	What can reject	Method implication
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, DRC, 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 measure-ments plus ongoing telemetry.	Field behavior, reliability, rollback policy, incident review, and accountable human decision.	Use for calibration, validation, and drift monitoring, not blind exploration.

The table should also 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 have their own version of the 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 regime structure 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.

The environment should therefore expose feedback budget as a first-class object. A result should not say only that candidate A beats candidate B. It should say which fidelity level produced that comparison, how many evaluations were spent, what failed, which assumptions were held constant, and what higher-fidelity check would be needed before commitment. That is the bridge from this chapter to Chapter 7.

5.7 Simulator Mismatch and Proxy Gaming

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.

> **Simulator mismatch.** Simulator mismatch is the 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. No simulator 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.

> Field note: 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. 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.

Environment design should therefore include red-team checks 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.

5.8 Building Environments for New Subfields

A useful Architecture 2.0 environment can start small. 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.

The recipe is straightforward.

1. Define the task in architectural language: workload characterization, cache exploration, accelerator parameter search, compiler/runtime tuning, chiplet partitioning, reliability analysis, or design-review critique.
2. Choose the representation the loop will read and write: configuration file, architecture description, graph, trace record, report, RTL fragment, test bench, or design-loop card.
3. Define the action schema: which fields can change, which are read-only, which combinations are legal, and which changes require human approval.
4. Wrap the tool path: simulator, compiler, profiler, RTL flow, EDA stage, runtime system, benchmark harness, or telemetry pipeline.
5. Define observations and feedback: metrics, traces, logs, warnings, errors, generated artifacts, cost, and fidelity level.
6. Define invalid-action semantics: illegal parameter, noncompilable artifact, nonsynthesizable design, violated constraint, timeout, simulator crash, or unsupported workload.
7. Log provenance and negative traces: tool versions, seeds, workload versions, failed candidates, rejected alternatives, and reasons for rejection.
8. State the human decision rule: what the architect reviews, what can be accepted automatically, and what must escalate.

The readiness test 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 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 teach the loop what it can try, what it can observe, and what evidence matters.

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 chip-design system.

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.

Environment validity has 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.

Operating discipline 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.

> **ArchOps.** This book proposes *ArchOps*, by deliberate analogy to MLOps, for the operational discipline of keeping an architecture design loop valid over time: versioning workloads and tools, recording provenance and negative traces, monitoring drift, and maintaining the rejection gates that let the loop stay trustworthy across many runs. It is the architecture counterpart of MLOps, but its artifacts are simulators, RTL, constraints, and evidence ledgers rather than models and datasets.

Naming the discipline matters because a one-time environment and a maintained one look identical on the first run and diverge completely by the tenth.

This chapter's main claim is simple: tools become architecture environments when they expose actions, observations, costs, constraints, provenance, feedback, and rejection. Once that happens, Chapter 6 can ask which methods belong inside the loop. Without it, method choice floats free of the architecture problem. With it, generation, prediction, optimization, critique, and verification become roles inside a credible design system.

Chapter 6

Methods for Generation, Prediction, and Optimization

What this chapter gives you

After this chapter you can:

- match a method role (generate, predict, optimize, critique, verify) to an architecture task;
- state a method claim as object, action, feedback, and rejection condition;
- assess hardware awareness as a staged capability, not the use of hardware vocabulary;
- choose a method by loop conditions rather than by fashion.

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 agent frameworks. It is a discipline for matching method roles to architecture work.

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 may spend scarce simulator or synthesis evaluations carefully, but only if the action space and objective are well formed. Reinforcement learning 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.

Architecture 2.0 therefore treats methods as roles in a compound design system. A credible loop might include a generator that proposes candidates, a predictor that estimates behavior, an optimizer that selects the next experiment, a critic that challenges assumptions, a verifier that checks constraints, and a human architect who decides whether the evidence is strong enough to proceed. The method question is 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 agents. A useful historical shorthand 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 workflows. 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, provenance, negative traces, and human rejection authority inside one explicit loop.

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 is important: 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 shows the chapter's working map. The same architecture loop can contain several method roles, but each role has a different claim and a different evidence requirement.

Table 6.1 gives the checklist form. The table is deliberately phrased as questions because method choice should be defended in architecture terms, not only in machine-learning terms.

Table 6.1: **Each method role needs different evidence:** Generation, prediction, optimization, critique, repair, verification, and explanation 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.

Role	Architecture use	Evidence needed	Failure mode
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 rules.	Polished explanations without authority to reject.
Verify	Check constraints, invariants, tests, tool outputs, and evidence chains.	Independent checks, provenance, and escalation rules.	Treating one tool pass as final truth.

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 that supports the claim, and the condition that can reject it.

Object-action-evidence rule. State every method claim as: this role acts on this architecture object through this interface, receives this feedback, and can be rejected by this evidence.

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.

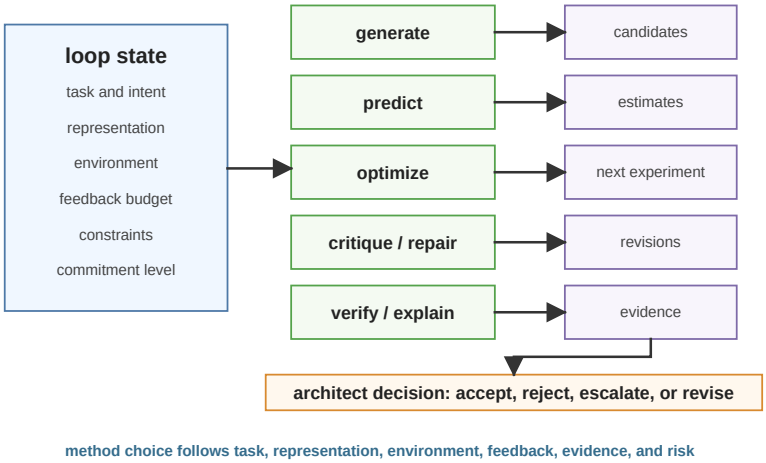


Figure 6.1: **Method roles are useful only inside a represented loop:** Generation proposes candidates, prediction estimates behavior, optimization chooses evaluations, critique and repair improve artifacts, and verification turns feedback into rejectable evidence. The roles can be implemented by different models, scripts, tools, or humans.

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

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.

Role	Object to name	Feedback to require	Rejection condition
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, source packet, 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.
Verify/explain	Invariant, interface contract, numerical tolerance, synthesis constraint, regression result, or evidence chain.	Independent checks, replayable commands, tool logs, and escalation record.	Failed check, inconsistent evidence, weak fidelity, or architect refusal to commit.

6.2 Hardware Awareness as Staged Capability

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 lecture, hardware awareness means that a method or agent 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.

Figure 6.2 gives the working capability map. The levels are cumulative as an assessment vocabulary, but they are not a claim that every system improves along one monotone

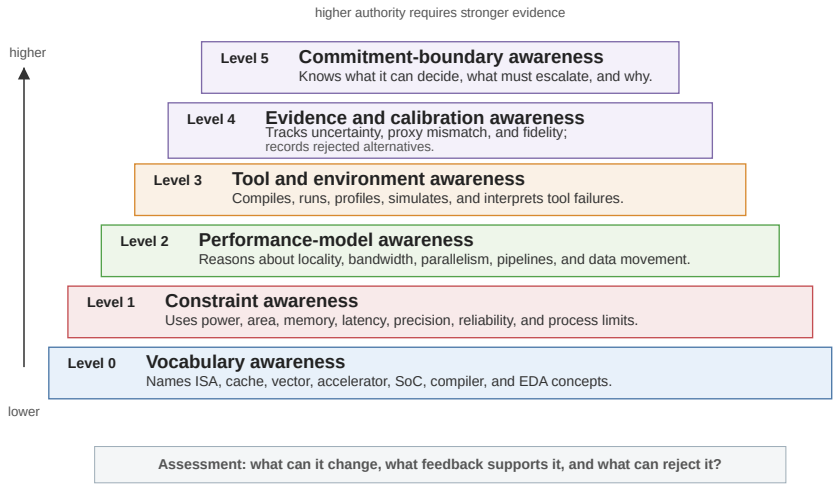


Figure 6.2: **Hardware awareness is a staged capability:** The assessment is not whether a method can mention hardware terms, but what it can safely change, what feedback supports the change, and what independent mechanism can reject it.

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. Tool and environment awareness connects those mechanisms to compilers, profilers, simulators, synthesis reports, and failed runs. Evidence awareness adds calibration, uncertainty, proxy mismatch, fidelity, and negative traces. Commitment-boundary awareness is the highest level because the method can state what it is allowed to decide, what must be escalated, and why.

Functional correctness is cross-cutting, not optional. 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.

Table 6.3 makes the assessment explicit. Each level should be judged by what the method may change, what feedback supports that change, and what 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.
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	Accept, revise, reject, or escalate recommendation.	Evidence ledger, rollback cost, risk, and review context.	Human architect or independent verification refuses commitment.

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 agent that can propose accelerator parameters has constraint awareness only if invalid actions are blocked or rejected. It has commitment-boundary awareness only if it can connect workload intent, hardware resource limits, compiler/runtime assumptions, fidelity gates, and human decision points into one accountable loop. Here, "performance model" means a cheap analytical, learned, or surrogate estimator used before live tool execution. 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.

6.3 Generation: Proposing Candidates and Artifacts

Generation is the most visible role because it is easy to demonstrate. 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. The loop still needs validity checks, tool execution, evidence, rejected alternatives, and human decision.

This distinction keeps Architecture 2.0 from becoming prompt-to-chip rhetoric. A generated RTL fragment is not an architecture result. A generated parameter set is not a design-space conclusion. A generated benchmark question is not a validated workload. Generation earns its place when it feeds a loop that can test, reject, compare, and revise.

> ⚠ Fallacy: generation is design
>
> It is tempting to treat a fluent generated artifact, an RTL fragment, a config, a benchmark question, as a result. It is not. A generated artifact is a proposal; it becomes a result only after the loop tests it, prices its evidence, compares it against a baseline, and an architect accepts the commitment. 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. 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.

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). The interesting part is not that a model produced Verilog. It is that the loop became trustworthy only 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

Prediction is central because architecture feedback is expensive. Long before recent foundation models, 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).

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.

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. Conformal prediction turns any surrogate, regardless of its internals, into one that emits calibrated prediction sets: intervals that contain the true value with a user-chosen probability under only an exchangeability assumption (Angelopoulos and Bates, 2021). For an architecture loop this converts "report confidence" from a slogan into an operation. The predictor returns not a point latency or energy but an interval, and the loop escalates to higher fidelity whenever that interval straddles a constraint, such as the 3 W envelope, or grows wider than the decision can tolerate. The guarantee is distribution-free, so it survives the non-Gaussian residuals architecture surrogates actually produce, and it degrades visibly under distribution shift, which is exactly when a proxy should not be trusted.

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

Optimization is often framed as search: find the best point under an objective. Architecture needs a richer formulation. 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.

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.

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); independent baselines later challenged the result (Chapter 7) (Cheng et al., 2023). The important lesson for this lecture is not that every architecture task should become RL. It is that method choice depends on state, action, transition, feedback, and commitment structure.

A less contested example narrows the design space until 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 reported circuits reached the area-delay Pareto frontier of a production design flow, and the resulting arithmetic units were adopted in shipping GPU silicon. 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. 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.

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

Sample efficiency 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.

Figure 6.3 turns that mismatch into a count-scale picture. The upper rows reuse the design-space anchors from Chapter 2: the small product example from this book, the MAESTRO accelerator DSE scale, AutoTVM-style operator tuning spaces, and the core Timeloop mapspace expression (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.

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 maximize decision value per unit cost:

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

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 quantity the loop should try to maximize.

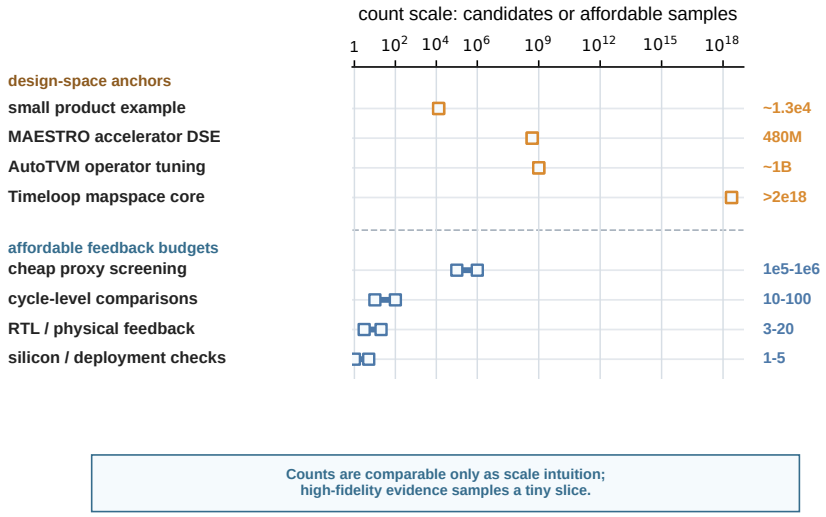


Figure 6.3: **Architecture methods face an evidence gap:** Design spaces can contain orders of magnitude more candidates than a loop can afford to test at stronger feedback levels. The plot compares counts only as scale intuition: source-backed or transparent design-space anchors above the divider, representative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

This heuristic has a rigorous counterpart. Bayesian optimization makes the choice of the next sample an explicit acquisition function over a surrogate's posterior, and GP-UCB gives that choice a no-regret guarantee under stated assumptions, so the loop can defend why it sampled where it did rather than appealing to intuition (Srinivas et al., 2010). Multi-fidelity Bayesian 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 rarely needs the full apparatus, but naming it matters: sample efficiency is a solved problem in form, and a loop that ignores it is leaving evidence on the table.

Table 6.4 gives a simple way to think about the regimes. The numbers are illustrative, not prescriptions. The point is that method choice changes when feedback is measured in milliseconds, minutes, hours, weeks, or silicon cycles.

Table 6.4: **Feedback regime should drive method choice:** Cheap proxies, moderate simulations, expensive EDA, scarce human review, and deployment signals 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 rules.
Few high-commitment checks	Silicon, deployment, fleet experiments, or customer workloads.	Critique, evidence organization, conservative recommendations.	Human decision and audit trail dominate.

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 teaches more. Chapter 4 and Chapter 5 are therefore not preliminaries to methods. They determine whether methods can learn.

6.7 Critique, Repair, and Explanation

Critique may be the most underrated method role in Architecture 2.0. Many loops do not need an agent 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 agents 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 chains, and human decisions.

Repair is the constructive side of critique. A method can propose a corrected configuration, rewrite an invalid constraint, patch a test bench, regenerate a plot with the right workload metadata, or produce a clearer design-loop card. 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.

6.8 Choosing a Method under Constraints

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 6.5 is a compact decision matrix. It is not meant to produce an automatic answer. It is meant to prevent method selection from being driven by fashion.

Table 6.5: **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.
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.

Question	If the answer is favorable	If the answer is unfavorable
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.

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 rule. If that sentence cannot be written, the method choice is probably floating above the architecture problem.

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, agent frameworks will change, tools will expose new interfaces, and benchmarks will evolve. A durable lecture should therefore teach 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, 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.

This chapter completes the core loop components: representation, environment, and method. The next question is credibility. Once a loop can act and choose methods, how do we know whether its feedback is evidence? Chapter 7 answers that question by making fidelity, verification, rejection, and trust explicit.

Chapter 7

Feedback, Verification, and Trust

What this chapter gives you

After this chapter you can:

- turn feedback into evidence through fidelity, provenance, and an evidence chain;
- set escalation thresholds and commitment levels 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.

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 level of human commitment matches the cost of being wrong.

This distinction is central to the lecture. 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.

The lighthouse prompt exposes the issue. 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.

7.1 Feedback Budget Ledger and Feedback Economics

The first trust problem 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. The ledger is not accounting bureaucracy. It is the object that tells the method what kind of learning is possible. A Bayesian optimizer, reinforcement-learning policy, surrogate model, 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 7.1 gives the working form.

Feedback budget. A feedback budget records 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.

Table 7.1: **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.

Budget item	What to record	Why it matters
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.

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.

One way to make that discipline concrete is to write the feedback budget and sample value explicitly:

$$B = \sum_i n_i c_i, \qquad V_i \approx \frac{\Delta\mathrm{Conf}(d \mid e_i)}{c_i}.$$

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, and $\Delta\mathrm{Conf}(d \mid e_i)$ is the change in confidence for the decision d. The equation is not a universal currency for evidence; it 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.

7.2 Fidelity Ladders and Evidence Chains

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 regimes and evidence chain. Feedback regimes organize sources from cheap proxies to high-commitment checks. An evidence chain records how a claim moves across those regimes, preserving provenance, uncertainty, negative traces, rejection gates, and human decisions.

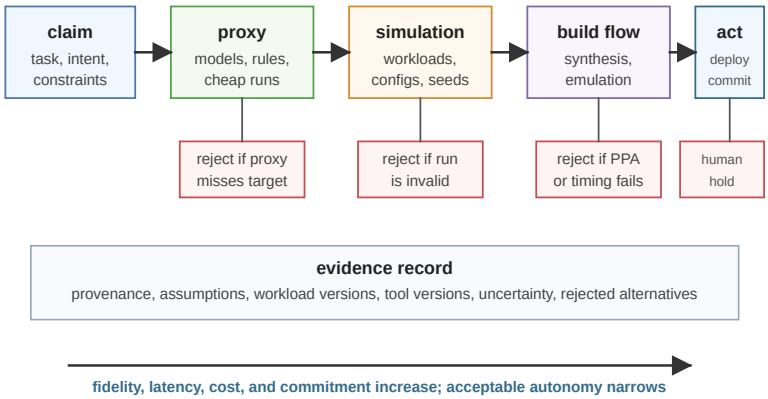


Figure 7.1: **Evidence chains turn feedback into trust:** A claim becomes more credible only as it moves through staged feedback sources, records provenance and uncertainty, and gives each stage authority to reject, revise, or escalate the result before human commitment increases.

This idea 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 chain is an assurance case for a design decision: it states what is claimed, at what fidelity, on what evidence, and what could reject it. 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.

Figure 7.1 shows the working model. A claim should move through a chain of increasingly costly feedback sources, with a rejection gate and evidence record at each stage.

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.

The regime view 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.

For 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.

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 chain itself must be represented so that a compound method system and a human reviewer can inspect it.

7.3 Commitment Levels and Reversibility

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, and who bears the consequence when the loop is wrong.

Table 7.2 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.

> **Escalation threshold.** 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.

Table 7.2: **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.
Irreversible	Mask-level choices, committed signoff decisions, fleet-wide rollouts, or customer-visible deployments.	Independent evidence chain, rejection authority, audit trail, and accountable human decision.	Human commitment 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.

7.4 Rejection Authority

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.

> **Rejection authority.** A credible Architecture 2.0 loop should name what can say no: at least one independent authority that can reject a candidate before the loop treats it as an architectural commitment.

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.

A compact way to write the commitment rule is

$$\mathrm{Commit}_k(x) = \begin{cases} 1, & \mathrm{valid}(x) \wedge E_k(x) \geq \tau_k \wedge \rho_k(x) \leq \rho_{\max,k}, \\ 0, & \text{otherwise.} \end{cases}$$

Here, x is the candidate, k is the commitment level, $E_k(x)$ is the evidence available at that level, τ_k is the evidence threshold, $\rho_k(x)$ is the residual risk, and $\rho_{\max,k}$ is the risk the architect or organization is willing to tolerate. 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: the accelerator fails validity at the 3 W envelope, the vector CPU fails it at the latency deadline, and only the SoC block clears every term at the experimental commitment level.

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.

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.

The independence requirement grows sharper as verification itself becomes AI-assisted. Production verification platforms now use machine learning to triage failures, rank likely root causes, and direct coverage (Cadence Design Systems, 2022). Such tools are valuable, 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? A rejection authority that shares the generator's blind spots is not an independent gate. It is a second opinion from the same witness.

7.5 Proxy Mismatch, Metric Gaming, and Calibration

The central failure mode 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 agent may appear capable because it overfits the evaluation loop, not because it understands the architecture problem.

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 agent may even produce a persuasive narrative for a proxy win unless the loop asks for calibration and counterevidence.

> Field note: the win that vanished at signoff
>
> A 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 team did not pick a bad candidate; it trusted a proxy past the point the proxy could support. The cheap fix is a rule written before the search starts: a cycle-level win is a reason to escalate, never a reason to commit.

Calibration is therefore a trust requirement. 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 agentic 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.

Benchmark governance also matters. MLPerf is useful not just because it names benchmarks, but because it defines rules, versions, and submission practices that make results more interpretable across a changing field (Mattson et al., 2020). 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

Industry readers will not trust an Architecture 2.0 workflow that ignores security, intellectual property, and 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 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 agents or tools can access each class of information. The trust question is not only whether the method is accurate. It is whether the workflow preserves the constraints under which architecture work actually happens.

For 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.

The practical rule is 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

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.

The anatomy of an evidence dispute is stable:

> claim; proxy; fidelity level; assumptions; workload coverage; provenance; counterevidence; rejection rule; and final human decision.

Learned chip placement is 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., 2023). The point here is not to adjudicate that dispute, which remains unsettled. 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. The 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 rules are made explicit.

Table 7.3 turns that anatomy into a checklist. It is intended for reading papers, reviewing internal tools, evaluating student projects, and deciding whether an agentic workflow is ready for a more expensive commitment.

Table 7.3: **Trust is a checklist, not a tone judgment:** A credible Architecture 2.0 claim states its task, feedback, fidelity, provenance, rejection authority, and human commitment before it asks the reader to believe the result.

Question	What a credible answer contains	Warning sign
What is the claim?	A bounded architecture task, objective, workload, and commitment level.	Vague claims of automation or improvement.
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.
What is the provenance?	Workload versions, tool versions, configs, seeds, constraints, assumptions, and baselines.	Hidden scripts or unstated defaults.
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.

The next chapter puts this discipline to work. It runs one loop end to end on the lighthouse prompt: generating candidates, escalating from proxy to simulation, rejecting on the power envelope, and stopping at an honest commitment level. The chapter after that generalizes the single loop into the patterns that recur across the stack, where the same ontology applies but the task, representation, environment, method role, feedback budget, evidence burden, and commitment level all change.

! Architect's checkpoint

Before believing a result, ask:

- What fidelity produced this, and does the evidence match the commitment level?
- What independent authority can reject it, and what would force escalation?
- Who is accountable for the commitment if the result turns out wrong?

Chapter 8

Running the Loop: The Lighthouse Prompt, End to End

What this chapter gives you

After this chapter you can:

- instantiate and run a design-loop card on a concrete prompt, not merely fill it in;
- recognize proxy mismatch when the cheapest winner fails at higher fidelity;
- apply the commitment rule to stop at an honest evidence level;
- read the residue a loop leaves: evidence chain, negative traces, and the next evidence the decision needs.

The lecture has described what a credible loop must contain. It has 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 generation, 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.

Bounding the task comes first. “Design a low-power XR compute subsystem” is not a loop; it is a wish. The bounded task for this chapter is narrower: for one XRBench-class workload slice, choose among three compute organizations under a 3 W power envelope and an 8 ms per-frame real-time deadline, and return the surviving candidate with its evidence and its rejected alternatives. The candidates are a vector CPU extension, a tightly coupled accelerator, and a shared-memory SoC block. That is enough to make the loop turn.

8.1 Round One: Generate and Screen on a Proxy

The loop begins cheaply. A generator proposes the three organizations, and an analytic proxy, a pre-RTL estimator in the spirit of Aladdin or a dataflow model like Timeloop (Shao et al., 2014; Parashar et al., 2019), estimates latency and energy per frame in milliseconds and millijoules. The proxy is fast and ignores most data movement, so it flatters designs that keep arithmetic local. At this fidelity the tightly coupled accelerator

looks best: it posts the lowest energy and latency because the proxy never charges it for moving data in and out.

A proxy result is feedback, not evidence. It is enough to keep all three candidates alive and to rank them for the next, more expensive stage. It is not enough to choose. The loop records the proxy ranking and escalates.

8.2 Round Two: Escalate to Simulation

Cycle-level simulation, a gem5-class model (Binkert et al., 2011), is the first stage that models memory traffic. It is slower and scarcer than the proxy, so the loop spends it only on the candidates that survived screening. The result is the central lesson of the chapter:

Table 8.1 runs the comparison.

Table 8.1: **One loop turning on the lighthouse prompt:** the lowest-energy candidate at proxy fidelity is the tightly coupled accelerator, but cycle-level simulation and the 3 W power model 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.

Candidate	Proxy lat / energy	Sim lat / energy	Power	Verdict
Vector CPU extension	9.0 ms / 22 mJ	9.5 ms / 24 mJ	2.1 W	rejected: misses 8 ms deadline
Tightly coupled accelerator	4.5 ms / 12 mJ	8.8 ms / 28 mJ	3.6 W	rejected: 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

The accelerator that won on the proxy collapses under simulation. Once data movement is charged, its energy rises above every alternative and its latency loses most of its lead. This is proxy mismatch made concrete: the loop was optimizing the measurement it could see, not the objective it cared about. 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. Figure 8.1 shows the reorder.

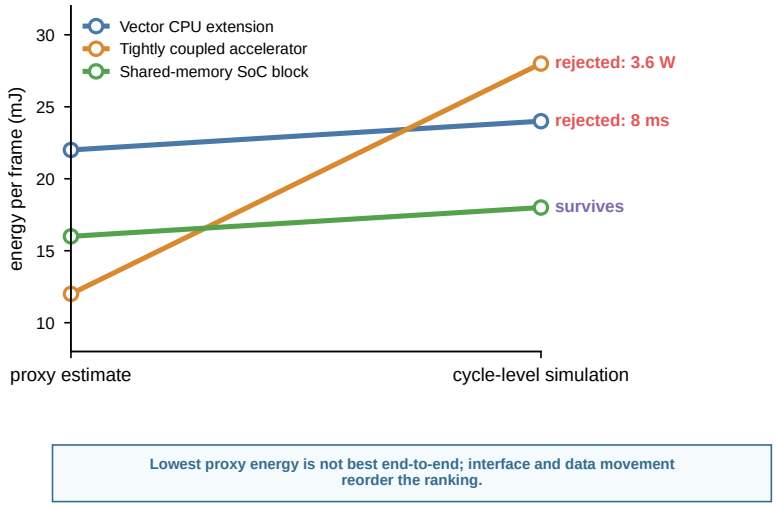


Figure 8.1: **The proxy ranking is a mirage:** the candidate with the lowest proxy energy, the tightly coupled accelerator, becomes the worst once cycle-level simulation charges data movement, and the 3 W power model rejects it. The shared-memory SoC block, second on the proxy, is the only organization that clears both the real-time deadline and the power envelope. The energy values are the same illustrative, computed numbers as Table 8.1.

8.3 Round Three: Reject on the Envelope

Simulation also exposes a harder gate. The tightly coupled accelerator does not merely lose on energy; its power model puts it over the 3 W envelope. That is a constraint, not a metric, so no amount of latency 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 risk is acceptable.

The accelerator's failure has a familiar shape. Its attraction was a large local speedup, but the interface and data-movement cost to reach it dominated the end-to-end result. The next chapter makes this precise with an accelerator performance model; the loop-level lesson is simpler: price the interface before believing the local win.

8.4 Round Four: Commit at an Honest Level

One candidate survives the deadline and the envelope. It is tempting to call that a result. It is not, yet. The evidence behind it is a cycle-level simulation and a power model, which support 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 synthesis, timing, and a stronger power estimate can confirm or reject it, and hold the other two as recorded negative traces.

This is the difference between an answer and a defensible answer. The loop did not produce a chip. It produced a surviving candidate, the evidence that supports it, the alternatives that failed and why, and the next evidence the decision needs. That is what the lighthouse prompt was always asking for: a design-space report with evidence and rejected alternatives, not a one-shot generation.

8.5 What the Loop Leaves Behind

The residue of one turn is the reusable artifact. The loop leaves a filled design-loop card, an evidence chain that records the fidelity at each stage, two negative traces with the reasons they failed, and an explicit open question for the next stage. Another architect can read that residue and reconstruct why the surviving candidate advanced, why the others did not, and what would overturn the decision.

That is the test the rest of the lecture has been building toward. The reader should now be able to take a new project and do three things: name the loop, judge whether its evidence matches its commitment level, and state what remains a human decision. The next chapter generalizes this single worked loop into the patterns that recur across the stack, from fast software loops to high-commitment silicon-facing work.

> **!** Architect's checkpoint
>
> Running your own loop, ask:
> - Is the task bounded enough that a candidate can actually be rejected?
> - When a cheap proxy and a stronger check disagree, which do I trust, and why?
> - At what commitment level does my evidence actually let me stop?

Chapter 9

Loop Patterns across the Stack

What this chapter gives you

After this chapter you can:

- classify an architecture task by its loop pattern and operating regime;
- match method posture and rejection authority to feedback cost and reversibility;
- explain why the same ontology carries a different evidence burden across the stack.

The previous chapters built the components of an Architecture 2.0 loop: representations, environments, method roles, feedback budgets, evidence chains, and human decision points. This chapter asks whether the framework travels. If it only describes one kind of design-space-exploration workflow, 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 architecture work with a characteristic task, representation, environment, 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. 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?

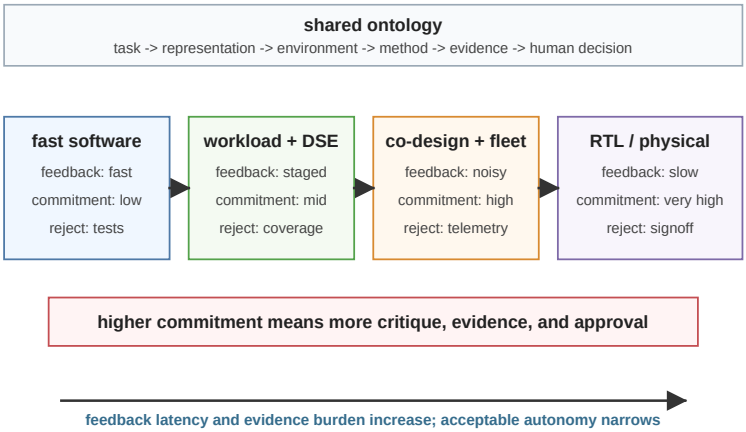


Figure 9.1: **Loop patterns share an ontology but not a commitment level:** The same ontology applies across software, workload, design-space exploration, co-design, systems, and silicon-facing flows. What changes is feedback latency, commitment cost, rejection authority, and acceptable autonomy.

9.1 A Template for Reading the Cases

The cases in this chapter use one card. The card asks for the task, representation, environment, method role, feedback budget, evidence, rejection authority, human decision, failure mode, and commitment level. 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.

Figure 9.1 gives the chapter's spectrum. 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.

Table 9.1 is the compact comparison. 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.

Table 9.1: **Different loop patterns need different evidence postures:** Workload, software, design-space, co-design, fleet, RTL, and physical-design loops use the same fields but differ in feedback latency, rejection authority, rollback cost, and human commitment.

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.
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 human 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. 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.

9.2 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.

The lighthouse prompt makes this concrete. "XRBench real-time mobile XR" 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.

MLPerf provides a useful analogy because it treats benchmarks as maintained community infrastructure, with versions, rules, and submission practices (Mattson et al., 2020). MLPerf Inference made the deployment-facing scale concrete: more than 100 organizations were building ML inference chips, systems spanned at least three orders of magnitude in power and five orders in performance, and the first submission round produced more than 600 reproducible measurements from 14 organizations (Reddi et al., 2020). The Architecture 2.0 lesson is broader: a benchmark is a living agreement about what evidence should count. 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.

Table 9.2 makes the reframe explicit. 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 a method or agent 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 9.2: **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 chain 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.
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, QoS targets, telemetry hooks, and review notes.	The loop produces a plausible result for the wrong product or deployment regime.

Architecture 1.0 meaning	Architecture 2.0 meaning	Loop state required	Failure if missing
Summarize results for a paper.	Preserve workload evidence as reusable architecture data.	Source packet, 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 agent 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.

9.3 Fast Software Loops

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.

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.

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 makes that bridge explicit by separating the core benchmark from target backends (Wen et al., 2025).

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.

The failure mode is treating fast feedback as complete truth. A kernel that wins on one input size may regress another. A runtime policy that improves average latency

may worsen tail latency. A compiler change that improves one benchmark may harm portability or maintainability. Even in fast loops, the card still needs workload coverage, rejection authority, and human decision when the change affects a larger system.

9.4 Architecture Loops: Accelerators, Memory, and Chiplets

Architecture design-space exploration is the canonical middle case. 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 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).

Chiplets raise the stakes because they turn partitioning and interfaces into architectural decisions. Standards such as UCIe make chiplet integration more concrete (UCIe Consortium, 2026), but they do not remove the architecture problem. The loop still has to reason about bandwidth, latency, power, thermal behavior, packaging, yield, verification, software contracts, and business constraints.

For the lighthouse prompt, this loop asks whether the XRBench subsystem should be a CPU extension, an accelerator, a shared-memory SoC block, or a more specialized partition. It asks which design regions are obviously infeasible, which are worth simulation, which survive power modeling, and which should be rejected before expensive evaluation. This is where candidate generation is useful, but only because the loop also records evidence and rejected alternatives.

9.5 Domain-Specific Architecture and Code Generation

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?

Every architect knows Amdahl's law (Amdahl, 1967), so restating it adds little. 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 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 LogCA makes precise. 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.

Figure 9.2 makes the interface cost precise with the LogCA model (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.

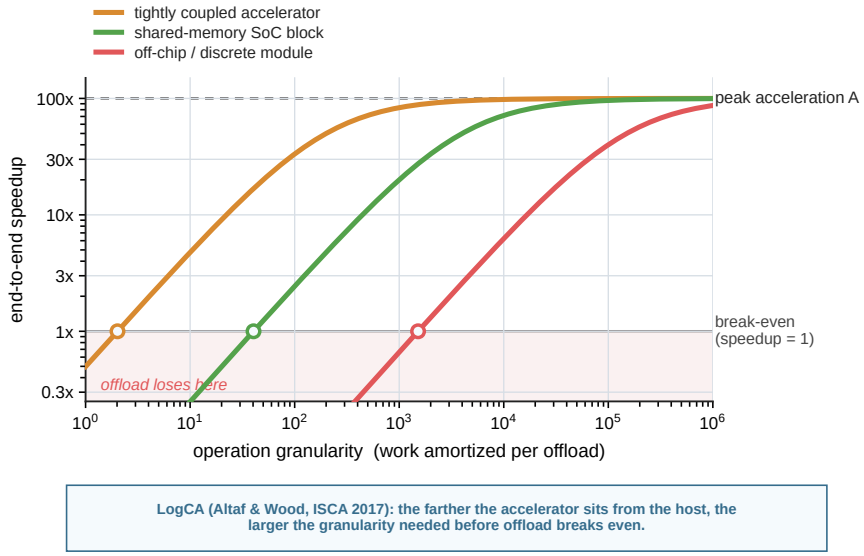


Figure 9.2: **The interface sets a break-even granularity (LogCA):** Using the LogCA model (Altaf and Wood, 2017), end-to-end speedup rises with the work amortized per offload only after it clears a break-even granularity set by offload latency and overhead, then saturates at the accelerator's peak acceleration. 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.

Figure 9.3 should be read as a checklist rather than a 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 Achilles heel is the software path. Specialized hardware becomes useful only when computations can be expressed, mapped, compiled, tested, profiled, maintained, and revised for the target. Halide made one version of this lesson visible by separating algorithms from schedules (Ragan-Kelley et al., 2017). AutoTVM and Ansor show the same pressure in tensor-program optimization, where schedules and measurements are part of the performance story (Chen et al., 2018; Zheng et al., 2020). MLIR pushes on the infrastructure problem by making multi-level compiler representations extensible across domains and targets (Lattner et al., 2020).

Figure 9.4 makes the implication architectural: 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

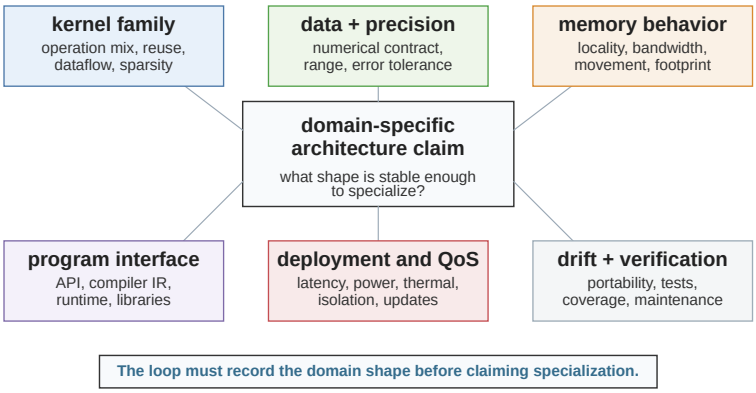


Figure 9.3: **Domain specificity has many shapes:** A domain-specific architecture is not specialized for a label; it is specialized for a bundle of kernel, precision, data-movement, interface, deployment, drift, and verification constraints. The loop must represent the shape it is claiming to exploit.

evidence. The waist itself contains the programming model, compiler IR, libraries, runtimes, and generated code that let the workload reach the machine.

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 rules. 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.

Field note: the accelerator that won in isolation

A 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 hardware was not wrong; the loop measured the wrong thing. This is the lesson LogCA makes precise: price the interface and the software path before believing a local speedup,

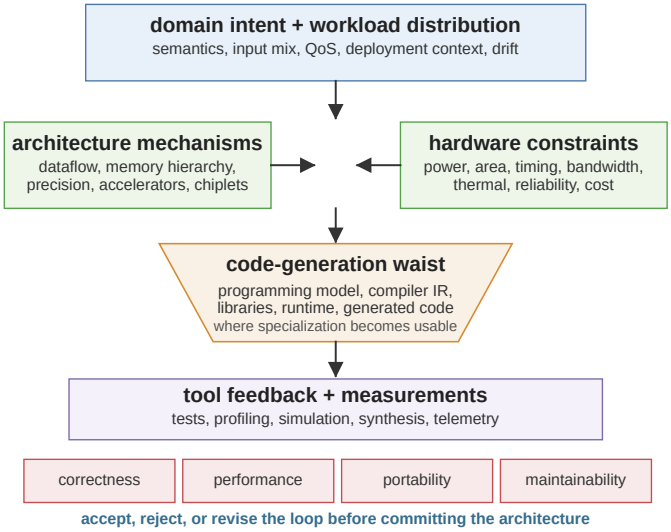


Figure 9.4: **Specialized hardware needs a code-generation waist:** Efficiency claims for a domain-specific architecture must pass through programming models, compiler representations, libraries, runtimes, and generated code, then be rejected or accepted using correctness, performance, portability, and maintainability evidence.

> and reject the candidate until a compiler-generated, end-to-end measurement survives.

9.6 Co-Design Loops: Compute, Memory, Network, and Power

Co-design loops are where the fusion-point argument becomes concrete. Architecture is not only a layer below software. It is 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 is a useful reminder. Moving data, accessing memory, and operating large systems can dominate the cost of useful work; Horowitz's energy analysis is often cited because it makes that imbalance concrete (Horowitz, 2014). At datacenter scale, the system is explicitly a warehouse-scale computer, with hardware, software, power, cooling, networking, and operations coupled together (Barroso et al., 2019).

An Architecture 2.0 co-design loop therefore needs richer representations: 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.

9.7 Systems Loops: Runtime, Serving, and Datacenter Policy

Systems 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 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.

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. 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.

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.

9.8 High-Commitment Loops: RTL, Physical Design, and Verification

High-commitment loops stress-test the framework. RTL changes, generator-level edits, physical design, timing closure, layout, power analysis, 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); its baselines and reproducibility were later challenged (Chapter 7) (Cheng et al., 2023). The Architecture 2.0 lesson is not that all physical design should be handed to an agent. 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 documented production successes, and they sharpen the same lesson rather than contradict it. Reinforcement-learning systems that search the physical-implementation flow have reached commercial tapeouts at scale (Synopsys, 2023), and reinforcement learning over a bounded circuit space produced arithmetic units that shipped in GPU silicon (Roy et al., 2021). What makes these loops credible is not the method; it is that every candidate had to pass the strongest available instrument, synthesis, timing, and signoff, before it could advance. 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 every output survives an independent gate, not methods that merely automate a single decision.

The rejection authorities in this regime are strong: 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.

9.9 What Transfers across Loops

Across all of these loops, the ontology transfers. 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. 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.

> **Same ontology, different evidence burden.** The more expensive, irreversible, or system-wide the commitment, the more the loop should shift from autonomous action toward critique, evidence organization, rejection, and human approval.

This is the practical reason to keep this chapter in the lecture. It prevents Architecture 2.0 from becoming either an abstract ontology or a paper survey. The reader should be able to take a new project and ask: Which loop pattern is this? What feedback can it afford? What representation does it need? What method roles are safe and useful? What can reject the result? What decision must remain with the architect?

The final chapter turns that question back toward professional judgment. Once the loop can be described, instrumented, acted on, and checked, what does the architect still own?

! Architect's checkpoint

Facing a new project, ask:

- Which loop pattern is this, and what feedback can it afford?
- Does the method posture match the reversibility and blast radius of the decision?
- What can reject the result, and what decision must stay with the architect?

Chapter 10

What the Architect Owns

What this chapter gives you

After this chapter you can:

- name the architectural responsibilities that cannot be delegated to a method;
- answer the strongest objections to Architecture 2.0;
- identify the community infrastructure the field still needs;
- use the design-loop card as a teaching and review artifact.

The opening prompt 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 lecture, that prompt was a provocation. It looked like a request for a future hardware foundation model. By 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.

That loop includes workload definition, architecture representation, tool interfaces, compound method roles, feedback budgets, evidence chains, rejection gates, 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.

The architect's responsibility moves upward. 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 rules that make it safe to use, and the accountability boundary around the final decision. This is why Architecture 2.0 is not simply an automation story. It is a responsibility story.

10.1 Return to the Moonshot

Return to the lighthouse prompt:

> Design a low-power, 64-bit RISC-V-based compute subsystem for an XRBench real-time mobile XR workload. Realize it as a vector-capable CPU, tightly coupled accelerator, or SoC block

under a 3 W TDP target in a 3 nm-class low-power mobile process, and return a design-space report with evidence and rejected alternatives.

The prompt now has visible structure. The workload is not just "XR." It is a workload distribution, a quality-of-experience target, a benchmark version, a set of traces, and a software stack. The contract is not just "64-bit RISC-V." It includes ISA assumptions, vector behavior, memory ordering, compiler support, operating-system interfaces, and software compatibility. The architecture object is not just a processor. It might be a CPU, accelerator, SoC block, or compute subsystem with specific interfaces and integration constraints. The technology envelope is not just "3 W" or "3 nm-class." It includes power, thermal behavior, process assumptions, physical design feasibility, verification burden, and deployment risk.

The requested deliverable also matters. A design-space report is different from RTL. RTL is different from a verified implementation. A verified implementation is different from a tapeout-ready design. 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 workflow. Treating all of those deliverables as the same is one of the fastest ways to overclaim.

The lecture's framework 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 credible? What alternatives were rejected? Who can stop the loop? Who is accountable for the decision?

That is the shift from prompt to loop. The prompt motivates the work. The loop makes the work inspectable.

10.2 Nondelegable Architectural Responsibilities

Nondelegable does not mean unaided. An architect can use models, agents, 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.

Figure 10.1 separates assistable loop work from the accountability boundary. Agents 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.

Human accountability boundary. The human accountability boundary is the line between work a loop may assist and commitments the architect still owns: intent, abstraction, evidence standards, rejection authority, deployment risk, and responsibility for consequences.

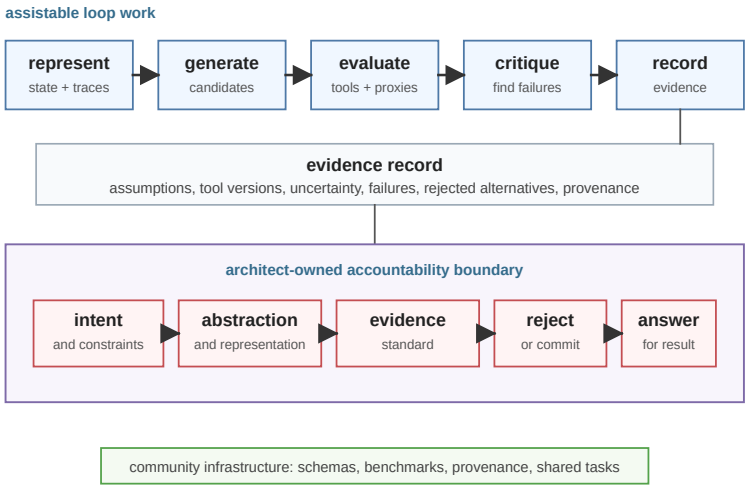


Figure 10.1: **The architect owns the boundary of the loop:** AI systems can assist many operations inside an Architecture 2.0 loop, but intent, abstraction, representation, evidence standards, rejection authority, accountability, and field-building remain architect-owned responsibilities.

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, but it does not own them.

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.

Responsibility	Why it cannot be delegated	How AI can assist
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 packets, 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.

This boundary also clarifies the word *agentic*. The book is not arguing that every architecture workflow 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.

10.3 The Strongest Objections

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 is gone, 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 agents participate. The book borrows established forms deliberately: the evidence chain 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.

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. 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.

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.

10.4 Community Infrastructure for Architecture 2.0

Individual practice is not enough to move a field. A field moves when work can be compared, reproduced, criticized, taught, and extended. Architecture 2.0 therefore needs community infrastructure, not only better private workflows.

The infrastructure does not have to expose proprietary designs or internal company data. It can start with shared conventions: design-loop cards, environment schemas, provenance records, benchmark versions, negative traces, source packets, 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?”

> **i** Research question
>
> What is the smallest public evidence record that would let two Architecture 2.0 loops be compared without exposing proprietary workloads, RTL, process assumptions, or internal design reviews?

Benchmarks show why this matters. MLPerf is useful not only as a list of machine-learning performance tasks, but as an evolving benchmark ecosystem with rules, versions, submissions, and governance (Mattson et al., 2020). Architecture 2.0 needs a similar instinct for loops. ArchGym shows one way to make architecture design-space exploration more environment-like by defining interfaces between agents and architecture tools (Krishnan et al., 2023). QuArch shows one way to start from the paper corpus and ask whether models can answer and reason about architecture questions (Prakash et al., 2025b,a). None of these is the whole field. Each is a partial infrastructure move.

Shared data and honest evaluation are starting to appear alongside these environments. CircuitNet assembles an open dataset of design-flow samples for machine learning in electronic design automation, directly attacking the field's shortage of reproducible training and benchmarking data (Chai et al., 2022). ChiPBench attacks the evaluation problem: it scores AI placement methods by their effect on final power, performance, and area, and reports that strong intermediate proxy metrics often fail to translate into better final designs (Wang et al., 2025). That finding is the community's version of this book's proxy-mismatch warning, and it shows why measuring progress in Architecture 2.0 means scoring the loop's end-to-end evidence rather than a surrogate.

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 agentic design loop needs to learn from the field's failures rather than rediscover them. A community that records only successful artifacts teaches 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 teaching 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.

10.5 Long-Horizon Challenge Tasks

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 agent 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.

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 10.2 sketches a starting set of tasks. They are deliberately stated as loop challenges rather than as single benchmark scores.

Table 10.2: **Long-horizon challenges should test architecture loops, not single prompts:** The field needs tasks that reward memory, valid action, feedback, rejection, evidence, and human commitment across time.

Challenge task	What makes it architectural	Success signal
Design-loop memory	The loop must preserve candidates, assumptions, tool outputs, failures, and decisions across a multi-step project.	A reviewer can reconstruct why a candidate advanced, changed, or was rejected.
Workload drift tracking	The workload is a moving object: benchmark version, model, software stack, trace mix, and deployment scenario can all change.	The loop detects when prior conclusions no longer support the current architectural claim.
Evidence-aware generation	Candidate generation is useful only when paired with the cheapest evidence path that can reject or advance the candidate.	The loop proposes both designs and the next evidence needed to trust or reject them.
Paper-to-loop reproduction	Architecture papers often omit scripts, traces, tool settings, negative results, or precise assumptions.	The system reconstructs the experiment loop, identifies missing artifacts, and states what would be needed to reproduce or falsify the claim.
Simulator trust calibration	Fast proxies are necessary but can mislead when the workload, model, or design point moves outside the calibrated region.	The loop knows when to trust a proxy, when to escalate fidelity, and when to invalidate a result.
Cross-stack co-design	Real wins often require coordinated changes across workload, compiler, mapping, memory, accelerator, runtime, and deployment policy.	The loop changes multiple layers while preserving valid interfaces and exposing which layer owns each assumption.

Challenge task	What makes it architectural	Success signal
Negative-trace corpus	Failed mappings, invalid RTL, bad floorplans, stale benchmarks, and misleading proxy wins are architecture data.	Rejected alternatives become reusable evidence rather than disappearing from the record.
Design-review assistant	The highest-value output may be a review packet, not a design artifact.	The system prepares assumptions, risks, missing evidence, sensitivity checks, rejected alternatives, and escalation questions for human judgment.

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.

10.6 From Capability to Standard

The path from a local capability to a field standard is gradual. A research group may first build a tool wrapper for its own simulator. A lab may then create an internal leaderboard. A workshop may define a shared task. A community may agree on benchmark versions, submission rules, provenance requirements, and reporting templates. Eventually, the field may decide that certain claims are not credible unless they expose the loop that produced them.

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 comparable without pretending all architecture work is the same.

A credible standard should make five things visible. First, the task should be replayable: another group can understand what was attempted. Second, the result should be comparable: the metrics, workloads, and constraints have enough common structure to support interpretation. Third, the artifact should be versioned: benchmark, tool, model, and dataset versions are part of the claim. Fourth, the evidence should be auditable: assumptions, provenance, and rejected alternatives can be inspected. Fifth, the evaluation

should resist leakage: a method should not succeed merely because it saw the benchmark, the answer, or the hidden rule during training.

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.

10.7 Education for Loop Designers

If Architecture 2.0 changes practice, it should also change education. Future architects still need the classical foundations: ISA, microarchitecture, memory systems, interconnects, compilers, operating systems, parallelism, power, reliability, and quantitative evaluation (Hennessy and Patterson, 2017). They also need a new form of literacy: how to design, inspect, and govern architecture design loops.

This is not a replacement curriculum. It is an added layer. Students should learn how to describe a design-space-exploration problem as a loop. They should learn to distinguish feedback from evidence, a proxy from an objective, a benchmark score from a deployment claim, and a generated candidate from a defensible decision. They should learn to ask what an agent can act on, what it cannot see, what it might optimize incorrectly, what was rejected, and what human decision remains.

The design-loop card is the simplest classroom artifact. A paper discussion can ask students to fill out the card. A project proposal can require the card before implementation. A review exercise can compare two papers not only by results, but by loop quality: task clarity, representation coverage, environment validity, feedback cost, evidence chain, negative traces, rejection authority, and human decision. This makes Architecture 2.0 teachable without turning a course into a tool tutorial.

The broader educational goal is taste under automation. Students should learn when to trust a tool, when to instrument it, when to reject its output, when to escalate to higher fidelity, and when to step back because the task itself is wrong. Those are architectural skills.

10.8 The Architecture 3.0 Horizon

Architecture 3.0 is not the subject of this lecture, but it is useful to name the horizon carefully. If Architecture 2.0 is about designing the design loop, then Architecture 3.0 would begin when the loop itself becomes adaptive at community scale. Agents would not only generate candidates inside a fixed environment. They would help discover better representations, propose new tasks, improve tool interfaces, organize negative traces, calibrate evidence standards, and refine the community's shared infrastructure.

The early signs are visible in industry, where electronic-design-automation vendors have begun packaging design assistants that chain tool steps with less human intervention between them. The durable way to read such systems is not by their feature lists, which will change, but as a shift in the partition of design autonomy: which decisions a human still makes, which the loop may make within stated bounds, and which the loop is never allowed to make alone. That partition, not the autonomy level a system claims to reach, is what the architect must keep designing (Janapa Reddi and Yazdanbakhsh, 2025).

That horizon is plausible, but it should be treated with restraint. A loop that adapts itself can also adapt in the wrong direction. It can chase benchmarks, hide failures, overfit to available tools, or encode the biases of the traces it sees. The more adaptive the loop becomes, the more important the architect-owned boundary becomes.

The durable question is therefore not whether Architecture 3.0 will make architects unnecessary. The question is what form of judgment, accountability, and community governance is needed when the loop itself can change. That is why the final object in this lecture is not a prediction. It is a question of ownership.

10.9 The Architect's Standing Obligation

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, teach it, compare 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.

Appendix A

Bootstrapping an Architecture 2.0 Workflow

The smallest useful Architecture 2.0 workflow is not "install an agent." 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 workflow: 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 agentic 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 Silicon Playbook

If an engineer is asked how to incorporate AI into a silicon or architecture workflow, the first answer is not to pick a model. The first answer is to turn one part of the workflow 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.

> Engineer move
>
> Start with a decision the team already has to make. Name the candidate space, the tool feedback, the evidence path, the rejection rule, and the architect who accepts or escalates the result. Then choose the narrow AI role that makes that

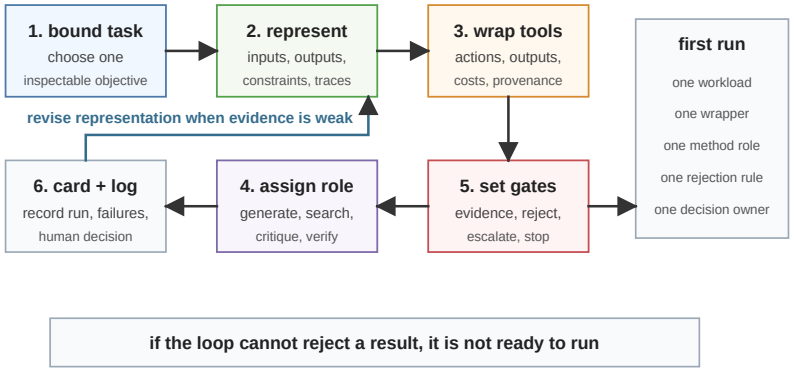


Figure A.1: **A first Architecture 2.0 workflow should be small and rejectable:** The workflow should be bounded, logged, rejectable, and owned by an architect. The loop can grow only after the task, representation, tool interface, method role, evidence gates, and human decision point are visible.

> loop cheaper, broader, faster, or more inspectable without weakening the evidence standard.

The practical question is therefore not "Where can we add AI?" It is "Which architecture decision is bottlenecked by representation, search, prediction, review, or evidence?" Table A.1 gives the compact playbook.

Table A.1: **The silicon 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.

Engineer move	Represent	AI role to allow	Gate before trust
Choose the evidence ladder	Proxy, replay, simulation, synthesis, emulation, or deployment evidence.	Predictor, surrogate, or active learner.	Escalate when proxy confidence exceeds its 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.
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 a silicon 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.

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: a workload slice, design question, simulator configuration, benchmark version, or review packet. Second, the output is inspectable: a ranked list, plot, rejected candidate set, evidence packet, 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: characterize an XRBench workload slice and produce three candidate accelerator/memory configurations with a latency-energy evidence packet and rejected alternatives. That is still hard, but it is a loop rather than a wish.

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 workflows often fail because important state is outside the loop: 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 agent 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 Rules

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 chapter.

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.

The rejection rules are not pessimism. They are what make 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 workflow should name the task, representation, environment, feedback, evidence, rejection rule, 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: 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 rules	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.

After the first run, ask five questions:

1. Did the loop produce a trace another architect can read?
2. Did it preserve both successful and failed attempts?
3. Did the evidence match the commitment level?
4. Did any rejection rule fire, and was that failure informative?
5. What should be revised first: task, representation, environment, method, evidence rule, or human decision?

If the answer to the first question is no, do not add more agents. Make the loop visible. If the answer to the fourth question is no because nothing could reject a result, do not trust the output. Add a rejection gate. The simplest credible Architecture 2.0 workflow is not the one with the most automation. It is the one whose evidence and failure modes are visible enough to improve.

Appendix B

Design-Loop Card and Review Rubric

The design-loop card is the practical form of the Architecture 2.0 ontology. It is meant to be filled in for a paper, a project proposal, a class exercise, or an internal design review. The card does not replace a technical report. It exposes the loop behind the report: 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.

> **Design-loop card.** A design-loop card is a one-page record of the task, representation, environment, method role, feedback budget, evidence, negative traces, rejection authority, and human decision behind an architecture claim.

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 design-loop card asks different questions:

- What architectural intent is being translated into work?
- What bounded task is the loop performing?
- What representation and world model make the work legible?
- What environment defines valid actions and feedback?
- What method role is being played?
- What is the feedback budget?
- What evidence supports the claim?
- What negative traces were captured?

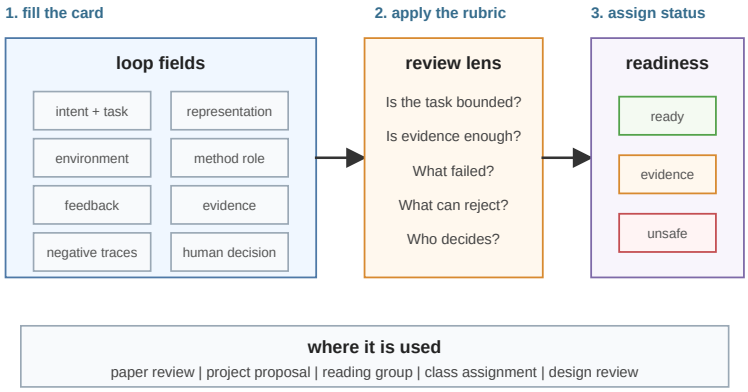


Figure B.1: **The design-loop card and rubric make loop review reusable:** The card exposes loop fields, the rubric reviews evidence and rejection structure, and the status records whether the claim is ready, needs evidence, or is unsafe at its current commitment level.

- What can say no?
- What does the human architect still decide?

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 teaching, it gives students a disciplined way to read Architecture 2.0 work without reducing it to a list of model names.

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.1 gives the working card. The fields are ordered to match the ontology used throughout the lecture.

Table B.1: **The design-loop card names the minimum review fields:** Each field asks for enough state to understand the task, representation, environment, feedback, evidence, rejection rule, and human decision.

Field	Question
Intent	What architectural objective is being pursued, and what constraints, non-goals, risks, or deployment assumptions matter?
Task	What bounded work is the loop doing: generation, prediction, optimization, critique, verification, workload characterization, benchmark construction, or design-space exploration?
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: 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: type checks, simulator failures, tests, formal tools, signoff, cross-tool disagreement, deployment signals, or expert review?
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.

B.3 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 chain.

Table B.2 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.2: **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.
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.
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.4 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 chain. These papers are easier to teach and compare because their claims are grounded in a visible process.

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 rules. 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 classroom exercise is to assign two students the same paper. One summarizes the paper conventionally. The other fills in the design-loop card. The class then asks what the card exposed that the summary hid.

B.5 Teaching Uses across the Lecture

The card is the central teaching artifact, but each chapter gives instructors a different classroom move. Table B.3 maps the lecture into reusable exercises. The goal is not to turn every chapter into a problem set. It is to make the framework active: students should decompose a prompt, inspect a loop, identify missing state, judge evidence, and state what the architect still owns.

Table B.3: **Each chapter should produce a reusable teaching artifact:** The artifacts help students decompose prompts, inspect loops, identify missing state, judge evidence, and state what the architect owns.

Unit	Artifact	Classroom use
Ch. 1: Moonshot	Lighthouse prompt decomposition.	Ask students to turn one prompt phrase into architecture decisions, evidence needs, and rejection conditions.
Ch. 2: Loop pressure	Design-loop pressure ledger.	Have students identify which costs are candidate count, feedback latency, verification effort, software drift, or human review.
Ch. 3: Framework	Ontology and design-loop card preview.	Map a paper or project onto task, representation, environment, method role, feedback, evidence, and decision.
Ch. 4: Data and world models	Architecture data-state checklist.	Compare what a paper preserves with what a loop would need: traces, configs, provenance, failed runs, and tacit assumptions.
Ch. 5: Environments and tools	Environment contract.	Specify actions, observations, invalid states, costs, and rejection paths for a simulator, compiler, or benchmark harness.
Ch. 6: Method roles	Method-role matrix.	Classify a method as generation, prediction, optimization, critique, verification, or coordination, then ask what evidence can reject it.
Ch. 7: Feedback and trust	Trust checklist.	Decide whether feedback is strong enough for the claimed commitment level and where escalation is required.

Unit	Artifact	Classroom use
Ch. 8: Running loop	Lighthouse loop walkthrough.	Run the opening prompt through one end-to-end loop turn, from intent and representation to evidence, rejection, and revision.
Ch. 9: Loop patterns	Loop-pattern comparison.	Compare a fast software loop with a high-commitment architecture loop using the same card fields.
Ch. 10: Ownership	Architect-owned boundary.	Ask what can be assisted, what must be owned by the architect, and who is accountable for the final commitment.
Appendix A	Bootstrap recipe.	Design a minimal loop for a new subfield with one task, one representation, one environment, and one rejection rule.
Appendix B	Full card and rubric.	Use the blank card as a paper review, project proposal, or final design-review handout.

B.6 Lighthouse Mini-Card

Table B.4 gives a deliberately incomplete mini-card 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.4: **The lighthouse mini-card 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.

Field	Sketch
Task	Bounded design-space exploration for a RISC-V-based compute subsystem, initially scoped to accelerator/memory organization for an XRBench-class workload slice.
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	Candidate generator, search/optimization method, surrogate predictor, critic for invalid assumptions, and report generator.
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, power/thermal limit, workload QoS violation, compiler/runtime incompatibility, or architect review.
Human decision	Decide whether the candidate merits deeper modeling, different representation, stronger fidelity, or rejection.

This mini-card 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.7 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 agent 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 workflow 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.8 Blank Template

Table B.5 is the one-page blank form. It is sufficient for a first pass because it forces the loop owner to name the task, evidence, rejection path, and decision boundary before running the workflow.

Table B.5: **The blank card provides a reusable loop template:** A reader can fill it in for a paper, tool, benchmark, classroom project, or internal workflow before judging the claim.

Field	Entry
Intent	
Task	
Representation	
Environment	
Method role	
Feedback budget	
Evidence	
Negative traces	

Field	Entry
Rejection authority	
Human decision	

After filling in the card, ask five final questions:

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

Resource Catalog for Architecture 2.0 Loops

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 a companion edition can keep it current without reprinting the book.

> **Architecture 2.0 resource.** A resource is useful for Architecture 2.0 when it makes some part of the design loop explicit: 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 spines, QuArch-style QA and reasoning data (Prakash et al., 2025b,a).	Bootstrap architecture vocabulary, concepts, and literature-grounded reasoning.	Paper text rarely preserves simulator state, failed candidates, tool logs, or review judgment.

Resource family	Examples	Loop role	Watch for
Workloads and benchmarks	XRBench, MLPerf, 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, 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/dataflow tools (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.
Compiler, autotuning, and codegen resources	AutoTVM, Ansor, MLIR, 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.
Evidence and provenance artifacts	Design-loop cards, source packets, seeds, configs, tool logs, calibration records, and negative traces.	Make claims auditable, reproducible, rejectable, and teachable.	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 record that preserves configurations, assumptions, and negative traces;
- a rejection rule and 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 agent can help with literature triage 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.

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 class projects.

These resources would make the field more teachable 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.

Appendix D

Architecture 2.0 Resource Directory

These links are not a general computer-architecture directory. A resource earns space here only if it helps an Architecture 2.0 loop name a task, represent state, expose actions, return feedback, preserve evidence, or reject a result. Use the list as a starting point and check current versions before relying on any benchmark, dataset, simulator, or tool.

D.1 Architecture 2.0 Framing

- **Course:** CS249r: Architecture 2.0. Course material on agentic AI for computer systems design.
- **Essay:** Architecture 2.0 gymnasium essay. Framing for the data-centric gymnasium argument.
- **Foundations article:** IEEE Computer article. Foundations article on AI agents for modern computer system design.
- **Workshop:** ISCA 2026 Architecture 2.0 workshop. Preview-edition workshop page and community entry point.
- **Portal:** Architecture 2.0 resource site. Living collection of Architecture 2.0 links and materials.

D.2 Architecture Reasoning and Design-Problem Benchmarks

- **QuArch:** quarch.ai. Architecture question-answering and reasoning benchmark. Use it to test whether a model can reason over architecture concepts, but not as evidence that a loop can act through tools or reject candidates.
- **CVDP benchmark:** NVlabs/cvdp_benchmark and Hugging Face dataset. Comprehensive Verilog design problems for RTL design and verification. Use it when the loop claim involves HDL generation, test harnesses, simulation feedback, or verification failures.

- **VerilogEval:** NVlabs/verilog-eval. Specification-to-RTL and Verilog code-generation benchmark with executable checks. Use it for method-role claims about RTL generation and compile/test feedback, not for system-level architecture claims.
- **KernelBench:** ScalingIntelligence/KernelBench. GPU-kernel generation benchmark with correctness and performance evaluation. Use it to study software-loop feedback, codegen, and performance evidence before claiming architecture-level benefit.

D.3 Architecture Environments and Design-Space Exploration

- **ArchGym:** Architecture Gym. Environment interface for ML-assisted architecture design. Use it as a concrete example of actions, observations, costs, and feedback in a bounded architecture loop.
- **Timeloop:** NVlabs/timeloop. Mapping, modeling, and code-generation tool for tensor workloads on accelerator architectures. Use it for dataflow, mapping, memory hierarchy, and accelerator design-space loops.
- **Accelergy:** Accelergy. Energy-estimation infrastructure for accelerators. Use it when a loop needs an explicit energy feedback source and calibration boundary.
- **MAESTRO:** maestro-project/maestro. Analytical cost model for DNN dataflows and tiling. Use it as a fast-feedback model whose limits must be recorded before higher commitment.

D.4 Full-System Simulation and Hardware/Software Harnesses

- **gem5:** gem5 simulator. Modular computer-system simulator. Use it for architecture feedback that needs workload execution, timing behavior, and reproducible simulator state.
- **FireSim:** FireSim. FPGA-accelerated full-system simulation. Use it when the loop needs stronger hardware/software feedback than a software-only proxy can provide.
- **Chipyard:** Chipyard docs. Integrated framework for generating and evaluating hardware systems. Use it when the loop must connect generators, RTL, simulation, and implementation artifacts.

D.5 Physical-Design and EDA Evidence

- **OpenROAD:** OpenROAD project. Open-source RTL-to-GDS flow. Use it for loops that need physical-design feedback, timing/area/power evidence, or signoff-adjacent rejection.

- **CircuitNet:** CircuitNet. VLSI CAD dataset for machine-learning applications in EDA. Use it for cross-stage prediction and physical-design learning claims, while preserving tool provenance and task scope.
- **ChiPBench:** AI chip-placement benchmark. Benchmark focused on end-to-end physical-design impact for AI chip placement. Use it when placement evidence must be tied to downstream physical metrics, not only intermediate scores.

D.6 Workload and Benchmark Governance

- **XRBench:** XRBench paper. Extended-reality machine-learning benchmark suite. Use it as a workload anchor for mobile-XR architecture loops, including scenario definition and workload coverage questions.
- **MLCommons benchmarks:** MLCommons benchmarks. Benchmark governance and reporting infrastructure. Use it as a model for workload versions, run rules, comparability, and community-maintained evidence rather than as a generic performance leaderboard.

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

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

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

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, et al. (2023) 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

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

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

Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, pp 4171–4186, URL https://aclanthology.org/N19-1423/

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

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

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

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

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 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)

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

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

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

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, Yazgan M, Jiang JW, Songhori E, Wang S, Lee YJ, Johnson E, Pathak O, Nazi A, Pak J, Tong A, Srinivasa K, Hang W, Tuncer E, Le QV, Laudon J, Ho R, Carpenter R, Dean J (2021) A graph placement methodology for fast chip design. Nature 594(7862):207–212, DOI 10.1038/s41586-021-03544-w

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

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

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

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

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

Semiconductor Industry Association (2026) Chip design and R&D. URL https://www.semiconductors.org/policies/chip-design/, accessed June 22, 2026

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

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

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

Standard Performance Evaluation Corporation (2017) SPEC CPU 2017 benchmark. URL https://www.spec.org/cpu2017/, accessed June 23, 2026

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

UCIe Consortium (2026) UCIe specifications. URL https://www.uciexpress.org/specifications, accessed June 22, 2026

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

X, The Moonshot Factory (2025) Our blueprint for moonshots. URL https://x.company/blog/posts/moonshot-blueprint/, accessed June 23, 2026

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