

An Architecture Philosophy for Agentic Systems

Designing accountable systems in an agentic world requires information architecture and system architecture to be treated as one discipline, so the concepts people see correspond to the truths the system enforces.

Written by **Wes Bailey**, Founder and CEO of fndry.sh

Contents

- Architecture Starts before the Diagram
- What Changes in an Agentic World
- The Principles
- Where This Differs from Traditional Architecture
- How fndry.sh Approaches an Architectural Problem
- What This Architecture Makes Possible

Architecture Starts before the Diagram

Architecture usually starts with a diagram. At fndry.sh, we think that is already too late. By the time we are debating where an application should run, which database it should use, or how its services should communicate, somebody has already made decisions about what the system believes, who can change it, and what everyone else is expected to trust.

We start with those decisions. What is this system supposed to make possible? Which distinctions matter? Who is allowed to establish the truth, and what will the next person, agent, or system be entitled to assume? A clean diagram cannot rescue unclear answers. It can only make the uncertainty look organized.

This is why fndry.sh treats information architecture and system architecture as one discipline. The concepts people see, the choices they are offered, and the states they rely on should correspond to truths the system actually preserves. If the interface says work is complete when an agent merely stopped running, the problem is not the label. The architecture has collapsed two different states. If a dashboard can quietly rewrite the source it summarizes, the presentation layer has acquired authority it should not have. If the system turns missing evidence into a successful result, it has traded trust for convenience.

Architecture is the design of truthful relationships among meaning, authority, responsibility, state, evidence, change, and recovery. Technology matters because it has to enforce those relationships. It does not get to define them.

What Changes in an Agentic World

Most software organizations were designed around scarce implementation capacity. Work became a production line: define a ticket, assign it to a person, create a branch, review the code, merge it, and move the work into the next queue. We are now dropping agents into that production line and congratulating ourselves when they perform the same steps faster.

Speeding up the old process is useful, but it is not an agent-native architecture. Once an agent can perform substantial work, the interesting question is no longer how quickly it completed the steps. What outcome did it accept responsibility for? What was it allowed to read, change, or publish? Which rules did the system enforce? What evidence survived the work? Which judgment still belongs to a person? Most importantly, what can the next responsible actor safely assume?

That changes the unit of progress. A collection of implementation events is not an outcome. A finished agent run is not acceptance, and a passing test is not proof of every product claim. Progress is a coherent result that meets its specification, carries enough evidence to be understood, and can become a trustworthy input to whatever happens next.

The human role changes with it. People should set direction, bring taste, accept responsibility, and make consequential judgments. They should not become the workflow engine that advances every state or approves every intermediate action. If the system needs a person to supervise routine work that it could govern itself, we have kept the bottleneck and merely hired faster participants.

So how do we respond to that world? At fndry.sh, we start by refusing to treat agents as faster hands inside an architecture built around human bottlenecks. We design around outcomes, meaning, authority, evidence, adoption, and trustworthy handoffs. The principles that follow are the framework we use to build agentic systems without turning that framework into another process.

The Principles

Begin with Meaning and Authority

The first question is not, "Which service owns this function?" It is, "What authority does this function require?" One component may own intent, another execution, another evidence, and another operation. They can cooperate without pretending those responsibilities are interchangeable.

A technical event is always trying to impersonate a product decision. A build completes, so the work gets marked complete. A deployment succeeds, so the product gets called healthy. An agent stops, so the outcome gets called accepted. These shortcuts make a system look simpler by making its meaning unreliable.

Every consequential state needs an authority. Every transition between authorities needs to preserve what happened, why it happened, and which responsibility moved with it. Otherwise the system can tell us that something changed without being able to tell us what the change means.

Preserve Distinctions that Change the Conclusion

Selected is not started. Started is not produced. Produced is not verified. Verified is not accepted. Accepted is not deployed. Deployed is not healthy, and healthy is not promoted. This is not pedantry because each state changes what someone is entitled to conclude or do next.

The person using the system does not need to see every internal state all the time. The architecture does need to preserve the distinction until it matters. Good information architecture reveals it at the right moment, and good system architecture makes sure it was never lost underneath.

Unknown deserves the same respect. Unknown is not false, zero, safe, complete, or unavailable forever. It is a legitimate state that protects the value of future evidence. A system that guesses may look decisive, but it is no longer trustworthy.

Preserve the Context that Made the Record True

Many systems have one place that defines an entity and another that records what happens to it. The first may know the current name of a client, provider, or patient. The second knows which work was performed, which documents were produced, and what happened over time. It is tempting to store only the relationship and look up the descriptive information whenever it is needed. That works until the description changes.

If a client changes its name, three years of work did not occur under the new name. The stable identifier still tells us that this is the same client, but it does not tell us what was true when the work happened. The operational record needs both. It should retain the stable relationship to the defining system and

copy the specific context required to understand the operation when that operation begins. A later change can govern new work without silently rewriting old work.

At fndry.sh, we call this preserving operational context. The definition system remains authoritative for who the entity is now. The operational system becomes authoritative for the context under which its own history occurred. That is not confused or duplicated ownership because the systems answer different questions. One answers, "What is true now?" The other answers, "What was true when this happened?"

The copy should be deliberate. Copying an entire profile creates unnecessary storage, privacy, and retention obligations. Copying too little makes the operational history dependent on mutable information it does not control. The architecture has to identify the smallest set of contextual facts that allows the record to remain truthful, searchable, and useful after the source changes or the old operation moves into an archive.

Define Outcomes, Then Let Capable Actors Act

fndry.sh prefers frameworks to processes. A useful framework defines the outcome, the boundaries, the available capabilities, the important evidence, and the actions that are not allowed. It gives a capable person or agent room to decide how to accomplish the goal.

A process assumes that the sequence is the source of safety. Sometimes sequence matters. Often it is accumulated ceremony that nobody is willing to remove. Every time a step is added to a process, a kitten goes to die. We all love kittens, so let's not add a step to a process.

This distinction becomes more important when agents are doing the work. If every agent action creates another human approval, we have rebuilt the old bottleneck with faster participants. Governance should live in permissions, contracts, valid state transitions, evidence requirements, isolation, and failure behavior. Human attention should be reserved for decisions that actually require human authority or judgment.

Make the Simple Path Safe

Powerful systems should not require people to understand all of their machinery before they can accomplish something useful. A person should be able to express the outcome while validation, isolation, evidence capture, recovery, and routine operational work happen underneath. The architecture should carry that complexity so the person does not have to carry it in their head.

That does not mean hiding consequence. The goal is to hide ceremony, not substance. A simple action should still make it clear what will change, which target is affected, whether the action is reversible, and what proof will exist afterward. Simplicity is not the absence of rigor; it is what rigor looks like after the system has done its job.

Treat Constraints as Design Material

Some of the most useful architecture begins when the obvious solution is not available. A platform rule blocks the preferred distribution model. A legal requirement prohibits the expected integration. Cost, policy, time, access, or organizational reality changes what can be built now. Those constraints are part of the architecture whether we like them or not.

The answer is not to ignore the constraint or abandon the outcome. Restate what must be accomplished, accept the constraint as part of the problem, and look for a different system shape. A local bridge, a read-only adapter, or a smaller independent capability may look like a workaround. It may also expose the right boundary for a much larger platform.

The mechanism is not the principle. "Local first" is not automatically a philosophy, just as "cloud first" is not one. The principle is to find the smallest constructive solution that respects reality, delivers value, and leaves behind a boundary that can support what comes next.

Design Adoption with the System

An architecture nobody can enter is not finished. If the first useful experience requires someone to understand the final platform, configure every future capability, or accept a giant operating model all at once, the platform will remain an idea admired by its designers.

A platform should start with something small enough to understand and complete enough to matter. The first layer should solve a real problem on its own. It should also establish a stable identity, contract, or boundary that allows the next layer to attach without invalidating the first.

This is how platforms should emerge. The product experience can remain coherent even when the implementation spans multiple repositories, runtimes, languages, or release cycles. One product does not require one codebase. It requires one understandable direction and clear responsibility across its parts.

Adoption is not a marketing problem to address after the architecture is finished. The order in which capability becomes understandable, trustworthy, and useful is part of the architecture itself. A platform has to give people a way in before it can give them somewhere larger to go.

Design the Handoff, Not Merely the Artifact

Every meaningful outcome should be able to become a trustworthy input. When work crosses a boundary, the next person, agent, or operating system should not have to reconstruct what the previous actor intended or decide for itself whether the result was accepted. The handoff has to carry the meaning of the work, not merely the files it produced.

What that requires depends on the consequence. It may include the intended outcome, governing constraints, decisions, resulting state, evidence, unresolved questions, recovery information, and the

authority being transferred. The point is not to attach a larger report to every action. The point is to preserve whatever the next actor needs to proceed without guessing.

This is especially important in agentic work because implementation can move faster than understanding. If the next agent has to reverse-engineer why the previous agent made a decision, or a person has to replay the entire execution history to decide whether the result is fit for use, the architecture did not complete the handoff.

Make Change Inspectable, Attributable, and Recoverable

Systems change. The architectural question is whether a change produces understanding or merely produces a new state. Important changes should reveal what will happen before they happen, record what actually happened, identify who or what was responsible, and provide a practical recovery path.

History should remain available without forcing everyone to live inside it. Canonical state should remain distinct from projections, caches, and generated views. Missing evidence should remain visible instead of being rewritten into reassurance. These are not reasons to move slowly; they are how a system earns the ability to move quickly without turning a failed change into a permanent mystery.

The Bailey Postulate says: "If nothing changes, it never gets better." That does not mean every change is good. It means improvement requires the ability to change, observe the result, learn from it, and change again without losing the truth of what happened.

Choose Technology for the System

Architecture has its own version of not-invented-here syndrome: "I know this tool, therefore I use it." Familiarity has value because it can reduce implementation risk, improve estimates, and make a system easier for the current team to operate. It is still only one factor, and it does not get to define the problem in advance.

The choice has to fit the outcome and constraints. Capability, reliability, security, compliance, cost, operability, ecosystem, adoption, available skills, lock-in, migration, and eventual replacement may all matter. Their importance changes with the problem, which is why a defensible decision begins with the problem rather than the architect's existing toolbox.

The moment the choice is made, it is already legacy. Somebody inherits it. They will have to understand it, operate it, secure it, upgrade it, defend it, and eventually replace it. Thinking about a brand-new decision as legacy forces those obligations into the design while there is still time to do something about them.

A defensible decision does not have to remain correct forever. It has to make sense given the outcome, alternatives, constraints, evidence, risks, and tradeoffs known at the time. Future people should be able to reconstruct that reasoning without depending on the architect's memory. It's not done until it's documented.

Where This Differs from Traditional Architecture

Traditional architecture has given us essential disciplines: cohesion, coupling, interfaces, reliability, security, performance, scalability, maintainability, and cost. None of those are wrong, and none of them are optional. The problem is where traditional architecture often stops.

It tends to assume the problem has already been framed, the concepts have already been named, and a human organization will provide whatever interpretation, coordination, approval, and recovery the technical design leaves outside its boundaries. Information architecture becomes a product concern. Adoption becomes a marketing concern. Human judgment becomes a process concern, and operation becomes somebody else's concern.

fndry.sh does not accept those separations. A component is not well bounded merely because its API is clean; its authority must also be clear. A system is not scalable merely because it can handle more traffic; it must support more decisions and more autonomous work without making people the coordination layer. A platform is not complete merely because its services can be deployed; people must be able to understand where to begin, trust what its states mean, and adopt more capability without swallowing the entire platform at once.

The important change is not that agents can write code. It is that implementation capacity is becoming less scarce, which changes what architecture has to organize. Supervising activity matters less than assigning responsibility. Human ceremony has to give way to system governance. Implementation events matter less than outcomes, evidence, and handoffs. The boundary between what a system does and how a person understands what it did can no longer be dismissed as somebody else's discipline.

That is the maverick part of this philosophy. fndry.sh is not trying to make traditional software architecture slightly more efficient with AI. We treat architecture as the structure of agency: who or what can act, under which authority, toward which outcome, with what evidence, and how the resulting truth moves safely to whatever happens next.

How fndry.sh Approaches an Architectural Problem

We start with the outcome. What should a person, agent, or system be able to accomplish when this works? Then we look for the meaning that has to survive: the important concepts, distinctions, identities, states, and historical context. We assign authority before mechanism. Who may decide? Who may act? Which part establishes current truth, and which part has to preserve the truth under which an operation occurred?

Then we take the constraints seriously. Technical feasibility is only part of the problem. Legal obligations, platform rules, cost, policy, time, distribution, adoption, and organizational reality all shape the available system. We look for the smallest constructive solution that can deliver value now and establish a boundary worth building on.

Only then do we choose mechanisms. Each choice has to fit the responsibility it is being given. We assume the choice is already legacy, so we consider how it will fail, how it will be operated, how it can change, and how somebody will replace it. We make the simple path safe, put routine governance into the system, and preserve the evidence needed to understand what happened.

Finally, we design the handoff. What does the next responsible actor receive? What may they assume? What remains unknown? Which judgment still belongs to a person? If those answers cannot survive the transition, the architecture is not finished.

This is a framework for making decisions, not a process for manufacturing steps. The questions remain stable because they expose the responsibilities every system has to resolve. The path through them should change with the problem.

What This Architecture Makes Possible

The point of these principles is not to make architecture more elaborate. It is to build systems that can carry more responsibility without transferring that complexity to the people who use them. When meaning, authority, evidence, and recovery are designed together, people and agents can act with greater independence because the system makes clear what they may do and what everyone else may safely conclude.

That matters because implementation is no longer the scarce resource it once was. As agents make more software possible, the architectural challenge shifts from organizing production to preserving trust across decisions, changes, and handoffs. The systems that endure will not be the ones that generate the most activity. They will be the ones that turn intent into outcomes without losing the reasoning, evidence, or responsibility that made those outcomes trustworthy.

This is what fndry.sh is building toward: systems that begin small enough to adopt, grow through stable boundaries, and remain understandable as their capabilities expand. Every choice will eventually become legacy, and every system will eventually pass into somebody else's hands. The architecture is finished only when that person can understand what was built, why it was built, what it can be trusted to do, and how it can change without losing the truth of what came before.

Acknowledgment: This blog post was developed by Wes Bailey with editorial assistance from ChatGPT. The principles, examples, judgments, and final decisions are Wes's and guide the work of fndry.sh. ChatGPT helped synthesize prior conversations, organize the argument, challenge interpretations, and refine the language through an iterative editorial process.