

Cooking the Cat

An account of AI centric software development through Puma's practical experience

Edition dated 2 October 2026

Contents

Part 1 Finding the system worth building

- 1 Start with the outcome
- 2 Discover intent through conversation
- 3 Investigate before committing
- 4 Make the emerging design visible
- 5 Choose among meaningful alternatives

Part 2 Shape the software around its problem

- 6 Model the domain before choosing abstractions
- 7 Make ownership and lifecycles explicit
- 8 Keep the necessary complexity separate
- 9 Design boundaries that can hold
- 10 Decide when to reuse refactor or replace
- 11 Learn through real applications

Part 3 Design software that agents can use

- 12 Treat the agent as a real interface user
- 13 Build interfaces around intent
- 14 Connect standards knowledge and decisions

- 15 Separate probabilistic judgment from deterministic control
- 16 Record what happened and why

Part 4 Turn shared understanding into coordinated work

- 17 Write plans that preserve intent
- 18 Design the development path
- 19 Give agents bounded responsibilities
- 20 Coordinate parallel work while it happens
- 21 Match models to the work
- 22 Revise the plan without losing the goal

Part 5 Establish that the result deserves trust

- 23 Test the outcome as well as implementation
- 24 Use independent review effectively
- 25 Design for interruption and failure
- 26 Measure before optimizing
- 27 Finish with evidence

Part 6 Build an ecosystem without losing the products

- 28 Grow shared foundations from demonstrated needs
- 29 Balance foundational work with usable releases
- 30 Make software understandable beyond source code
- 31 Improve the development process itself
- Working vocabulary
- Sources and further reading

Preface

An account of AI centric software development

Puma wants this book to explain the thinking behind a great deal of practical work with software agents, and to expose that thinking to criticism. These are complementary aims. A method becomes easier to criticize when its assumptions and consequences are visible. It becomes more useful when readers can decide which parts fit their own work.

The subject is a way of connecting purpose, design, implementation, and evidence. AI agents make that connection unusually important. They can produce plausible code quickly, distribute work across multiple tasks, and maintain extensive records. They can also elaborate an assumption that nobody meant to make. A coherent plan, a clean architecture diagram, or a passing test suite can preserve a misunderstanding rather than expose it.

Practical engineering work provides the setting for the examples. Recurring questions concern resource lifecycles, execution, coordination, knowledge representation, and provenance. The examples make decisions, assumptions, and consequences inspectable; readers need not reproduce a particular collection of projects to examine the reasoning.

The case studies do not establish a finished, proven method. Nor do they show that every declared practice is followed consistently. A published standard says what its consumers should do. Source code shows a particular implementation. A test observes behavior under its conditions. A conversation explains an intention. These are different kinds of evidence, and the differences matter.

The account uses third-person references to Puma where describing stated views or inspected work. The explanations, exercises, and proposed alternatives are the book's synthesis. They should not be mistaken for quotations, a transcript of Puma's development sessions, or a claim that every suggested technique is already in routine use.

How to use this book

The chapters follow a change from an uncertain request to a usable result, then ask what happens when many such changes become an ecosystem. You can read them in order or start with a current problem. A developer struggling with resource ownership may begin with chapters 7 and 25. Someone coordinating several agents may start with chapters 18 through 22. A reader evaluating the approach critically may find chapters 27 and 31 useful companions to the opening.

A running example keeps the discussion concrete: an application acquires a local resource so another component can use it. Sometimes that resource is a model. The illustrative system has enough moving parts to expose identity, readiness, cancellation, ownership, and acceptance, but it is illustrative rather than a

reconstruction of any one implementation. Each example states the behavior under discussion instead of expecting familiarity with that repository.

The exercises ask you to make decisions and produce inspectable artifacts. They do not require adopting a particular agent product or reproducing Puma's technology stack. You can perform most with a small application and an ordinary version-control workflow. The work should remain proportional. If an exercise exposes no consequential uncertainty in your task, record that conclusion and use the simpler path.

Experienced practitioners appear where they illuminate an actual choice. Their accounts include enthusiasm, restraint, failures, and changes of mind. Academic and technical sources provide additional distinctions and counterweights. Neither testimony nor a benchmark validates this entire method. References are included so you can inspect the basis of a claim rather than accept it on authority.

The repository cases are pinned to revisions inspected on 2 October 2026. Local tests were not rerun for this book; where results come from project reports or hosted logs, the wording says so. Living external sources were checked during the same research period. Software and model capabilities will change. The durable questions concern ownership, feedback, evidence, and the cost of decisions, not a permanent ranking of products.

The strongest criticism will identify a specific claim, an omitted condition, or a credible alternative. Perhaps a proposed boundary creates more work than it removes. Perhaps a record is too expensive to maintain. Perhaps an ordinary implementation followed by a demonstration would discover the problem faster. Those are useful objections. The book succeeds if it makes them easier to formulate and investigate.

Part 1 Finding the system worth building

Begin with the result someone needs, then develop enough shared understanding to choose the next step. Investigation and prototypes remain part of that understanding; the aim is a defensible commitment rather than a complete prediction of the future.

1 Start with the outcome

An agent is asked to make a downloaded model available to an application. It adds a download operation, writes bytes to disk, returns success, and tests that the file exists. The application still cannot use the model.

Perhaps the model needs a companion file. Perhaps a runtime is missing. Perhaps the local catalog does not know its identity. Perhaps success was announced before finalization finished. These are different defects, but the request left one question unresolved: what does available mean?

This is an illustrative scenario. Its purpose is to show how an implementation can satisfy a literal interpretation while missing the intended result. Puma has described this problem directly: a plan may be logically consistent and still omit an invariant that would have changed what the agent built. Nothing inside the plan necessarily contradicts the omission. Executing the plan more faithfully cannot recover information that never entered it.

An outcome states what becomes possible for someone. An implementation describes how the system makes it possible. The distinction is useful without being absolute. A mechanism can itself be a requirement when compatibility, regulation, interoperability, or a deliberate product choice makes it essential. The developer needs to discover why a requested detail matters before treating it as either binding or disposable.

Find the action after success

For the download example, ask what the consumer should do immediately after receiving success. If the answer is “open the selected version through the supported runtime,” file existence is only an intermediate condition. The system must also make the selection resolvable and establish whatever readiness the runtime requires.

A first outcome statement could be:

After the operation succeeds, the intended local consumer can request the selected resource by its stable identity and begin using it through a supported runtime.

The statement still needs discussion. Does the acquisition operation install the runtime? Does it report missing support? Can it substitute another model? Those choices affect the product and should not be hidden inside an attractive method name.

Now add an unwanted result:

The interface says the operation succeeded, but the consumer receives an artifact it cannot identify or use.

This counterexample gives the team something concrete to exclude. It does not select an architecture. It establishes that “download returned success” is an inadequate acceptance observation for the intended outcome.

The same method works outside model management. If a request says “add export,” ask what the recipient must be able to do with the exported data. If it says “make search faster,” ask which delay is disrupting which activity. If it says “use a graph,” ask what relationships the person needs to see or which decision the graph should support.

Separate purpose from a plausible representation

Suppose a person wants a graph of a codebase. They may be trying to understand dependencies, locate the reason behind a decision, assess the impact of a change, or identify unfinished work. A graph of file imports

may help with the first and fail at the others.

A useful question is, “What would you want to notice or decide more quickly?” It respects the suggestion while investigating its purpose. The person might explain that spatial relationships are essential to how they work. In that case, a visual requirement is real. Or they might reveal that they mainly need a reliable answer to “Why is this here?” A graph could remain useful, but the data now needs to include reasoning and provenance rather than merely structural links.

Do not convert this into a ritual in which every request is challenged. If the outcome is clear, the requested mechanism fits, and the change is bounded, proceed. Discovery has a cost. Its value lies in resolving a decision that affects the result.

Keep the domain assumptions visible

Software operates in a world it does not completely control. A program can verify a checksum, but it relies on an expected checksum with a trustworthy origin. It can publish a local path, but it relies on filesystem behavior and permissions. It can ask a process to stop, but the request is not the same event as confirmed termination.

Requirements research distinguishes desired world outcomes, software specifications, and assumptions about the environment. Zave and Jackson's formal account makes the relationship explicit: the software's specified behavior and domain assumptions must together support the required outcome. This is a conceptual foundation, not proof that a particular conversation finds every assumption. Four Dark Corners of Requirements Engineering (<https://www.cse.msu.edu/~chengb/RE-491/Papers/dark-corners-re-zave-jackson.pdf>).

For our resource example, an assumption might be that publishing a reference can be made atomic with respect to readers. If that assumption is false on a supported platform, a correct implementation of the planned sequence may still expose a partial result. Naming the assumption lets someone investigate it before the plan treats it as settled.

Try it on a small change

Choose a request you could implement in a day. Write five short statements:

- The person or component that needs the change
- The action it should be able to perform afterward
- One superficially successful result that would disappoint it
- One environmental assumption needed for success
- One detail whose status as a requirement or preference is unclear

Explain the result back to the requester. Include a consequence they did not explicitly mention. If your export includes private metadata, your faster search returns stale results, or your download silently installs a runtime,

that consequence belongs in the discussion.

The artifact should be short enough to remain useful during implementation. It is a reference point, not a promise that no further learning will occur. When the code changes, it gives the team a way to ask whether the new behavior still serves the original purpose.

2 Discover intent through conversation

A good discovery conversation changes what the participants understand. It does not merely collect answers for headings in a plan.

Puma's concern is that consequential details often remain unstated because they seem obvious, have not yet occurred to the requester, or only become visible when a proposed solution is examined. An agent can fill those spaces with reasonable assumptions. The resulting design may look complete precisely because its assumptions are internally consistent.

The conversation should therefore expose choices that matter, while leaving room for both the person and the agent to revise the picture.

Ask questions that can change a decision

Consider the resource acquisition example. Asking “What programming language should we use?” may be premature if the work belongs in an existing component. Asking “What should the consumer see while the new version is being checked?” may reveal a product invariant.

The second question can change the design. The person might require the old version to remain usable, accept a temporary unavailable state, or want both versions addressable. Those answers imply different identity and publication behavior. A useful question has a plausible connection to a decision; it is not simply something the agent could ask.

Questions also need sequence. Begin with concrete situations the person understands. Ask how they use the current system, what goes wrong, and what they expect next. Introduce technical alternatives once the distinction they represent is clear. Otherwise the person may choose an option based on a name while misunderstanding its consequences.

Avoid leading formulations. “Wouldn't a reusable service be better?” invites agreement with an unexplained conclusion. “Two applications need this behavior; should they share one process, or is independent operation important?” identifies a real tradeoff. Even that question may need an example before the person can answer.

Keep different kinds of statement separate

The discussion may contain an intention, an observation, a constraint, a preference, a hypothesis, and a decision. They do not need six files, but they should not collapse into one undifferentiated list.

“The model must work offline” could be a firm requirement. “The existing runtime fails on this machine” is an observation with a particular environment. “I think a graph would make it easier” is a hypothesis. “I prefer one screen” is a preference whose importance may depend on the task. “We will preserve the current version until the replacement is ready” is a decision once the authorized person agrees.

When the agent presents its synthesis, it should retain those distinctions. Turning every suggestion into a mandatory requirement overconstrains the design. Treating every requirement as negotiable removes the person's control over the outcome.

The published Coding-Standards discovery procedure makes these distinctions explicit and asks what could make a technically correct implementation wrong for its intended use. It also preserves the boundary between discovery and implementation authority: research or a polished visual does not approve an unresolved design. This is a statement about the inspected policy, not evidence that every development session follows it.

Discovery and Design at the inspected revision (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning/discovery.md>).

Use examples to find the omitted rule

People often recognize a bad result before they can formulate the rule that prevents it. Ask about a normal case, an interrupted case, and an awkward boundary.

For acquisition, a normal case is a new resource becoming usable. An interrupted case is a cancelled operation. A boundary case is cancellation arriving after the bytes finish but before the resource is published. The last case exposes whether cancellation means “stop future work where possible,” “guarantee no publication,” or something else.

Do not treat the first answer as an implementation detail the person should have known. The whole point of the conversation is to make the consequence visible. The agent can explain the possible behaviors and their costs, then ask which outcome matters.

Examples should not become an exhaustive catalog. If ten examples all exercise the same distinction, another may add little. Seek a case that could change the rule. That is the difference between exploring the problem and making the conversation longer.

Explain back without claiming certainty

A useful synthesis describes what will be built, what the person will experience, and which consequences were added by the proposed solution. It also identifies the remaining question whose answer could change the direction.

For example: the current version remains usable during preparation; the new version becomes visible only after verification; cancellation before publication preserves the old version; a missing runtime is reported rather than silently installed. The person can now challenge a picture of the behavior rather than inspect a list of internal tasks.

Nuseibeh and Easterbrook describe requirements work as interpretive and iterative, with tacit knowledge limiting what interviews alone can uncover. Their roadmap supports combining conversation with observation, modeling, and revision. It does not promise complete understanding before implementation. Requirements Engineering A Roadmap (<https://www.cs.utoronto.ca/~sme/papers/2000/ICSE2000.pdf>).

When conversation is enough

A clear repair may need only one exchange. If a label shows the wrong unit, the correct unit is known, and no underlying semantic change is involved, extended discovery may delay a useful fix.

A small patch can still require substantial discovery if it changes a coupled decision. Moving one success response earlier may alter what multiple consumers assume about readiness. The size of the diff does not measure the amount of shared understanding required.

For your own task, identify the next question that could materially change the outcome, design, scope, or acceptance. If there is none, explain the direction and proceed within the available authority. If there is one, resolve it through a question, a source check, or a small experiment. The conversation should remain responsive while that investigation happens.

3 Investigate before committing

Research becomes useful engineering work when a finding can change a live decision. Without that connection, it can produce a large collection of facts while leaving the actual uncertainty untouched.

In our running example, the team is considering a new publication mechanism for acquired resources. Before designing it, someone should determine whether the existing API already preserves the old resource until a new candidate is ready. A familiar method name is not enough. Its contract, callers, state transitions, and failure behavior matter.

The investigation may reveal that the existing mechanism fits. It may expose a narrow missing operation. Or it may show that the current abstraction confuses transfer completion with consumer readiness. Those outcomes

lead to different amounts of work.

Give research a decision and a stopping condition

A useful brief has four elements: the question, the decision affected, the evidence needed, and the condition under which further investigation would stop changing the choice.

For example:

Determine whether the current resource API can publish a verified candidate while preserving existing readers of the old version. Inspect its contract and two actual callers. Exercise the failure boundary if the code does not establish it clearly. If the API already provides this behavior on supported storage, reuse it; otherwise describe the smallest missing promise.

The brief is bounded without demanding a predetermined answer. It does not ask the investigator to “research acquisition architecture” indefinitely. It names what evidence would support reuse and what would justify a change.

A stopping condition is not a timer that makes uncertainty disappear. It is an account of what the decision needs. If the missing evidence remains consequential, report the gap. If further detail would not change the selected reversible step, proceed and retain the limitation.

Investigate the system as it exists now

Puma's architectural discussions emphasize the danger of treating an old integration assumption as a current requirement. A consumer designed around an earlier API may need updating rather than forcing the provider to restore a now-unnecessary behavior. The general lesson is to inspect the current boundary before proposing a repair.

Start with the actual entry point. Follow the successful path, then the error and cancellation paths. Identify what callers rely on and which facts are merely accidental. Read the relevant tests, but distinguish their assertions from observed execution. A test file can explain the intended contract even when it has not been run in the current environment.

Sillito, Murphy, and De Volder's observational work identified programmers' questions ranging from finding a starting point to understanding relationships across code. Search results often needed integration before they answered the real question. The study is not an optimal-search recipe, but it supports describing the relationship you need to understand rather than equating investigation with locating a symbol. Questions Programmers Ask During Software Evolution Tasks (<https://www.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf>).

Treat existing implementation as a constraint with a price

Current code constrains a change because compatibility, migration, and maintenance cost are real. It is not necessarily immutable. A design that exists for historical reasons may be the very thing preventing the intended outcome.

Consider three possibilities. The desired behavior can fit the current interface directly. It can fit after a narrow refactor. Or accommodating it would perpetuate a misleading abstraction, making replacement worthwhile. Investigation should give enough evidence to compare these paths. “This is how the system works” is a description, not a complete argument for retaining it.

The same applies to an external library. Check what it actually guarantees. A cancellation method might request cancellation without waiting for resource release. A version number may correlate with a capability without defining it. A provider may expose a common format while requiring distinct runtime support. Verify the relevant promise rather than infer it from a category label.

Bryan Cantrill’s published Oxide discussion offers a useful distinction. He describes a familiar kernel task deliberately selected to favor the agent, and two subtle defects he needed to correct. That is expert-supervised delegation of a largely understood solution, not open-ended design discovery. His estimate of the time he would have needed without AI is not a control condition. The comparison reminds us to identify which uncertainty the agent is actually being asked to resolve. Engineering Rigor in the LLM Age (<https://oxide-and-friends.transistor.fm/episodes/engineering-rigor-in-the-llm-age/transcript>), especially 07:01–14:56.

Keep research from blocking the conversation

Puma wants background research to proceed while the main agent continues discussing the problem. That can work when the handoff preserves decision ownership. A research task returns evidence, alternatives, and limitations; it does not silently replace the shared direction.

The lead should integrate findings as soon as they matter. “The API already supplies the readiness boundary we need” is actionable. A long report that arrives after the plan has committed to a replacement is less useful. Conversely, an unresolved question should remain visible rather than be filled with an optimistic assumption merely to keep the conversation flowing.

Parallel research is often easier to divide than parallel implementation. One investigator can inspect API semantics while another checks platform support. Their findings can still conflict, especially if they examined different versions. Record the revision or documentation version so the disagreement can be reconciled.

Choose an experiment when reading stops helping

Sometimes the cheapest evidence is a small reversible implementation. If a library’s behavior is unclear, a minimal characterization test may be more informative than another hour of documentation. The experiment

should state what it is testing and what it does not establish.

A local process-kill test can examine cleanup after process termination. It does not prove resilience to power loss. A prototype on one filesystem can expose a flaw but cannot establish all supported platforms. An experiment becomes misleading when its result is promoted beyond the conditions exercised.

Before your next investigation, write the decision it can change. Afterward, return the finding in that same form: what was inspected, what was observed, what conclusion follows, and what remains uncertain. Research is complete enough when the next commitment has a defensible basis, not when there are no more documents to read.

4 Make the emerging design visible

A conversation unfolds one sentence at a time. A design often depends on relationships that must be considered together. Who owns a resource while another task is using it? Which state permits publication? Which unanswered question affects two competing approaches? A suitable visual can keep those relationships available while the conversation moves on.

Puma's proposal for discovery includes a local page maintained alongside the discussion. Its purpose is to show the emerging picture: findings, open questions, choices, and relevant relationships. It should help participants notice a mismatch before a formal plan makes it look settled.

The page is useful only if its representation fits the question.

Choose what the reader needs to notice

A state diagram can clarify which transitions are valid. An ownership map can show who retains a resource after a caller disappears. A sequence view can distinguish the order of events. An interface mockup can expose the words a person will see. A comparison table can put alternatives against the same criteria.

For acquisition, a progress bar alone may conceal the distinction between transfer and readiness. A better discussion aid could show:

Visible state	What is true	What the consumer may do
Transferring	Some candidate bytes are being acquired	Continue using the old published version
Checking	Transfer has finished; readiness is unresolved	Continue using the old version
Ready	The agreed checks and publication step have completed	Resolve and use the new version

Visible state	What is true	What the consumer may do
Preparation failed	This candidate cannot be published	Inspect the reason; retain the previous usable version

This is a proposed product contract, not a universal acquisition lifecycle. It makes assumptions available for discussion. Someone may object that consumers need simultaneous access to both versions, or that there is no previous version. That objection improves the model.

Show uncertainty without making the page unreadable

A live visual can accidentally look more authoritative than the discussion behind it. Clean typography and complete-looking boxes suggest that the design has been decided. If an option is provisional, say so near the option. If a finding depends on a particular version, keep the version available with the source.

Avoid turning uncertainty into a sea of warning badges. Distinguish the few unresolved points that affect a decision from background details. A useful page might have a current proposal, two open material questions, and a small evidence list. It need not contain every sentence of the conversation.

The page also needs a clear owner. If several agents rewrite it independently, they may merge incompatible assumptions into a polished composite. Puma's discovery design assigns a separate author to the visual while the lead owns synthesis and decisions. The benefit is coherent editing, but the arrangement still has a cost: the author must receive current information, and stale representations must be corrected promptly.

A prototype answers a particular question

Houde and Hill distinguish prototypes by what they investigate: a product's role, its look and feel, or its implementation. The distinction is useful because apparent completeness in one dimension can conceal absence in another. A convincing interface can say little about resource safety; a robust backend probe can say little about whether a person understands the result. What Do Prototypes Prototype (https://creativetech.mat.ucsb.edu/readings/Prototypes_prototype.pdf).

For our example, a mockup asks whether “Transfer complete, preparing for use” is understandable. A failure-injection test asks whether a candidate rejected before publication leaves the old version usable. A lost acknowledgment after publication is a different case: the transition may already have committed, so reconcile the uncertain result instead of assuming rollback. Chapter 25 returns to that boundary. Both are valuable. Neither replaces the other.

Sometimes making the prototype changes the intended outcome. Mitchell Hashimoto's account of a Ghostty update feature starts with a bounded UI investigation and discovers preferences through the prototype. A later implementation works but conflates concerns; he changes the model and continues. The account supports intent

and boundaries while challenging any rigid idea that intent must be completely discovered before construction. It is a first-person case with a linked change, not a controlled productivity comparison. [Vibing a Non-Trivial Ghostty Feature](https://mitchellh.com/writing/non-trivial-vibing) (<https://mitchellh.com/writing/non-trivial-vibing>).

Preserve access beyond the visual

A diagram should have readable labels and a textual account of its central claim. People may use screen readers, small screens, monochrome printouts, or slow connections. A relationship that exists only as a colored line may be difficult to inspect or cite later.

Keep the important decision in a durable record once it is made. The visual can remain an explanatory view, but it should not become a competing authority whose outdated boxes contradict the plan. The cost of a live page rises when it duplicates information that must be changed in several places.

For a current design discussion, choose one question a visual could answer. Make the smallest representation that exposes it. Ask someone to explain the proposed behavior from the view. Where their explanation differs, investigate whether the defect lies in the design, the labels, or an unstated assumption. The success criterion is improved understanding of that question, not an impressive page.

5 Choose among meaningful alternatives

An agent can often produce several architectures on request. That does not mean the alternatives expose a real decision. Three diagrams with different service names may conceal the same tradeoff, while one apparently minor difference in ownership changes the entire failure behavior.

The point of alternatives is to reveal consequences that matter to the intended outcome.

Compare the same problem

For the resource system, consider three approaches. Reuse the existing publication API if it already provides the required readiness semantics. Add a narrow operation if transfer and publication are currently conflated. Introduce a shared acquisition layer if several actual consumers need the same identity and lifecycle contract.

These options differ in scope, migration cost, and the responsibilities they centralize. Compare them against the same scenarios: a successful replacement, a failed readiness check, cancellation during finalization, an existing reader holding the old version, and a restart after an uncertain result.

Do not let each option choose its own favorable criteria. A reusable layer should answer the same immediate usability requirement as a direct implementation. A small patch should account for any repeated lifecycle logic it leaves behind. The comparison should make costs visible on both sides.

The Architecture Tradeoff Analysis Method formalizes evaluation around quality scenarios, risks, sensitivity points, and tradeoffs. A small option note can borrow that reasoning without pretending to be a complete ATAM evaluation. The full method carries a process cost that may be inappropriate for a small reversible change.

ATAM Method for Architecture Evaluation (https://www.sei.cmu.edu/documents/629/2000_005_001_13706.pdf).

Include the option of doing less

A simpler design must be a legitimate outcome. If the consumer can tolerate restarting an interrupted download, resumable transfer may not belong in the first release. If only one application needs the behavior, a local implementation may be sufficient. If the existing interface fits, the best result of discovery may be a decision not to add another abstraction.

Doing less is not automatically simpler over the whole lifecycle. Duplicating a subtle publication rule across three consumers may reduce today's diff while increasing tomorrow's risk. The relevant question is which complexity is necessary now, which is a demonstrated common need, and which is merely anticipated.

An alternative should therefore include its likely point of reconsideration. “Use the current API until a second consumer needs a different retention policy” is more informative than “keep it simple.” It tells future developers what evidence would reopen the decision.

Make consequences understandable

A requester may approve an option because it meets every stated requirement, yet be surprised by an unstated consequence. A shared service may need to remain running. A local cache may retain data longer than expected. Automatic runtime selection may use more disk space. A stronger consistency rule may reduce concurrency.

Explain these consequences in terms of behavior, not just implementation. “This keeps the old version available while preparing the new one, at the cost of temporarily storing both” is easier to judge than a storage-class diagram alone.

Sometimes the consequence belongs to someone other than the requester. A change that makes contributions easier may increase maintainer review work. An agent-friendly interface may require another component to own retries or cleanup. Include the receiving party's needs rather than optimizing only the caller's experience.

Do not manufacture numerical certainty

A weighted score can help when criteria are genuinely comparable and the weights have an agreed meaning. It can also disguise judgment. Giving maintainability eight points and complexity six points does not measure either property.

Use numbers where there is a basis: measured startup time, memory use under a stated workload, migration steps, or a known license cost. Use reasoned comparisons where there is not. State the uncertainty instead of converting it to an arbitrary decimal.

Requirements framing itself can affect the search. Mohanani, Turhan, and Ralph's two experiments found less original but more practical concepts under requirements-oriented framing than ideas-oriented framing. Their paper-design tasks and samples limit transfer to agent engineering. The useful implication is to distinguish genuine obligations from tentative ideas, not to pretend every constraint is optional. Requirements Framing Affects Design Creativity (<https://arxiv.org/abs/1902.11278>).

Record the choice at the right scale

A compact decision record can retain the context, selected approach, alternatives, and consequences. It should explain the surprising part of the choice. If the obvious existing operation was reused without controversy, a sentence in the task record may suffice. If a shared boundary is being introduced, a durable record is more valuable.

Michael Nygard's architecture-decision proposal uses short records with context, decision, status, and consequences, retaining superseded choices. It is an early practitioner account rather than measured proof of effectiveness. Its practical appeal is that future readers can understand why a choice made sense without reconstructing the entire discussion. Documenting Architecture Decisions (<https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions>).

Choose one design decision in your current work. Describe two materially different viable approaches, or explain why only one meets the constraints. Compare them through the same concrete scenario. State the selected behavior, its cost, and the observation that would justify reconsidering it. That record is enough if it allows another developer to challenge the decision intelligently.

Part 2 Shape the software around its problem

The next chapters turn intended behavior into concepts, ownership, and boundaries. The running example acquires more precision only where a distinction changes what the system must do.

6 Model the domain before choosing abstractions

A domain model earns its place when it helps explain a behavior that would otherwise remain ambiguous. It is not a list of nouns waiting to become classes.

In the acquisition example, a person asks for one resource, a transfer is attempted, candidate bytes appear locally, checks run, and a consumer eventually receives something usable. Calling all of these “the download” is convenient until one stage succeeds and another fails. Then the system needs distinctions that its vocabulary has concealed.

Start with examples. The same resource can be requested twice. One attempt can fail while another succeeds. Two artifacts may represent different versions. A candidate can exist without being ready. A published version can remain in use while a replacement is prepared. These statements reveal identity and lifecycle questions before any architecture is selected.

Distinguish things that change independently

A resource identity answers which thing is wanted. An attempt answers which effort to obtain it is being discussed. A candidate answers what is locally present but not yet accepted. A published resource answers what a consumer may currently use.

Those concepts need not become four services or four classes. A small implementation may represent them with a few records and functions. The important point is that the system can express their different states without contradictory meanings.

Consider an attempt identifier used as a permanent resource identifier. It seems adequate while every request succeeds once. When an attempt is retried, the application may create a second identity for the same resource. Alternatively, if every retry reuses one mutable record, historical events may become impossible to distinguish. The correct choice depends on what callers must be able to refer to and what records must preserve.

A vocabulary table can expose the problem:

Concept	Remains the same when	Changes when
Requested resource	A transfer is retried for the same selected identity	The selected identity or version changes
Acquisition attempt	One attempt moves through its lifecycle	A new attempt is admitted
Candidate artifact	Checks examine the same candidate bytes	The candidate is replaced or reconstructed
Published reference	Existing readers retain the agreed version	Publication deliberately changes the reference

The table states an illustrative contract. It is not a claim that all products should use these exact identities.

Name the rule before the object

An invariant is a condition the system must preserve across relevant transitions. “An unverified candidate is never advertised as ready” is one. “Cancelling one consumer does not cancel another consumer’s continuing demand” might be another if acquisition is shared.

An invariant tells you something about a boundary. If several fields must change together to preserve it, they may need one consistency owner. If unrelated changes do not affect it, they may not belong in that owner’s responsibility.

Evans’s domain-driven design reference connects shared language, identity, bounded contexts, and consistency rules. These ideas are useful for asking where a meaning is valid and who enforces a rule. They do not require a microservice per bounded context or a class for every concept. [Domain Driven Design Reference](https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf)

(https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf).

A framework can then implement the model. It should not silently supply the model’s meaning. A database row, asynchronous task, or HTTP request may be a convenient representation, but its technical lifetime need not match the domain object’s lifetime. Much later complexity comes from assuming that it does.

Test the vocabulary against disagreement

Ask two people to describe what happens when transfer completes but publication fails. If one says the resource is complete and the other says it is unavailable, they may be using “complete” for different events. Resolve the distinction before making the status field more elaborate.

The same method applies to agents. A task may be assigned, admitted, running, awaiting input, or finished at a harness level. None automatically means its requested outcome has been accepted. If a single state called done serves all those meanings, downstream code will invent its own interpretation.

Domain modeling is therefore also interface work. The words in a tool response teach the caller how to act. A misleading word can create a bug even when the underlying data is accurate.

Resist the urge to complete the ontology

There is always another distinction you could represent. The useful stopping point is the set needed for current behavior, meaningful variation, and known failure boundaries. If a proposed concept cannot explain a scenario, enforce a rule, or support an actual decision, it may not belong yet.

For an exercise, take three status names from a system you maintain. For each, write what is true, what the caller may do, and one situation it must exclude. Then identify a pair whose meanings overlap or conflict.

Repair the vocabulary before adding more states. The result may be a new concept, a simpler name, or the removal of an unnecessary distinction.

7 Make ownership and lifecycles explicit

Ownership becomes interesting when something disappears. A caller disconnects, an agent is cancelled, a process restarts, or a request times out. The system still has resources, effects, and obligations. Who owns them now?

Three meanings of ownership often become entangled. Object ownership concerns who keeps a value alive. Operational responsibility concerns who must finish or clean up work. Authority concerns who may decide that an action should happen. A reference-counted pointer can solve the first while leaving the other two unresolved.

In the resource example, a consumer may request acquisition without owning the shared transfer. A service may own the transfer without being authorized to select a substitute resource. A background effect may retain a workspace after the requesting future has been dropped. The design needs to express these differences.

Follow the effect rather than the requester

A concrete Pumas change makes the issue visible. In the inspected draft acquisition work, a blocking effect could continue after its asynchronous waiter and callback disappeared. The registered effect did not itself retain the workspace reservation. The repair captured the workspace inside that effect, so the activity still able to touch the resource retained its claim until it returned.

The associated regression uses an actual exclusive lock, holds the blocking work, cancels the waiter, and attempts a competing acquisition of the lock. The competing handle must fail while the effect remains active and succeed after it finishes. Source and test assertions were inspected, and local results were recorded by the project. This is bounded evidence for the ownership rule, not comprehensive production or cross-platform qualification. Repair (<https://github.com/MrScripty/Pumas-Library/commit/c5deb493172d3fbdab55141d129f2ac7346bb3b>); regression at the inspected head (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-core/src/acquisition/service.rs#L3128-L3225>).

The general question is simple: which activity can still produce a side effect? That activity needs the relevant resource claim, even if the person who originally asked for it has stopped waiting.

Describe transitions with obligations

A list of states is not enough. Each transition should explain its precondition, effect, authority, and outstanding obligation.

For example, moving a candidate from checking to ready may require all readiness checks to pass, publication to occur, and the consumer reference to resolve. Moving from transferring to cancel requested may only record intent to stop. The latter state must not imply that the worker has stopped touching its workspace.

A useful transition record could say:

- Before: candidate exists, no ready reference has been published
- Decision owner: acquisition service under the caller's admitted request
- Required evidence: specified checks passed for this candidate identity
- Effect: publish the ready reference
- After: consumers may use the candidate; cleanup responsibility for superseded storage is separately defined

The cleanup clause prevents a common omission. Successful publication may leave work behind. If the caller's response ends the only lifetime that retained cleanup responsibility, success becomes a resource leak.

Shared demand changes cancellation

Suppose two consumers need the same resource. Cancelling one consumer's request need not mean cancelling the shared work. The system may remove that consumer's demand while continuing for the other.

This is a different domain from one request owning one task. Reusing a cancellation token without examining its meaning can make the first departing consumer disrupt everyone else. Conversely, never cancelling shared work can waste resources after all demand disappears.

Model the demand separately when the distinction matters. Decide who counts active consumers, how stale demand is retired, and what happens when new demand arrives during shutdown. There may be races that require one serialized owner or a compare-and-update operation. The solution should follow the invariant, not a preference for a particular concurrency primitive.

Make uncertain outcomes expressible

A timeout can mean that the observer stopped waiting, not that the operation failed. If publication may already have happened, blindly retrying can duplicate effects. The next action should usually inspect authoritative state or use an operation whose retry semantics account for identity.

An explicit uncertain state can be useful, but it should have a resolution path. A system that records uncertainty forever without allowing reconciliation has described the problem without making it operable.

For your own project, choose one resource that outlives a request: a process, file lock, network operation, temporary directory, or agent task. Draw the lifetimes of the caller, effect, and cleanup owner. Terminate the

caller in the model. If an obligation has no remaining owner, the design is incomplete. If several owners can independently release the same resource, inspect that boundary too.

8 Keep the necessary complexity separate

Simple and easy describe different things. A familiar framework may be easy to use while entangling several decisions. An unfamiliar design may require learning but allow each concern to be understood independently. Neither familiarity nor brevity is a sufficient measure of simplicity.

Puma's standards use this distinction to resist arbitrary proxies such as file count, type count, or call count. The relevant question is whether independent concerns can be understood and changed independently. A coherent module can be substantial because the problem itself is substantial. Splitting it into many fragments may move complexity into coordination without reducing it. Code Design at the inspected revision

(<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/topics/code-design.md#L15-L99>).

Locate the coupling that hurts

Imagine one function that downloads bytes, chooses a runtime, updates a catalog, renders progress text, and deletes old files. It may be easy to call. Its internal decisions are entangled if changing a presentation label requires understanding publication order, or if supporting a new provider requires changing cleanup semantics.

Do not respond by extracting every step into a service. First identify the change that should be independent. Presentation might depend on a stable status model. Provider transfer details might sit behind an acquisition contract. Publication and catalog update may need to remain together because they preserve one invariant.

The design should follow the actual dependencies. Two operations occurring at different times do not necessarily belong in different owners. Two functions with similar code do not necessarily express the same rule.

Parnas's classic modularity comparison asks which design decisions should be hidden, rather than simply dividing a program by processing steps. It remains useful as a question about change: what should be able to vary without forcing unrelated consumers to vary? It is a design argument and worked comparison, not a theorem that more modules improve performance. On the Criteria To Be Used in Decomposing Systems into Modules (<https://www.cs.lafayette.edu/~gexia/cs301/resources/parnas.html>).

Count the complexity you add

A refactor can remove one coupling and add several new concepts: an interface, registry, lifecycle, configuration format, and error translation layer. Those may be justified, but they are costs.

For the running example, extracting transfer adapters may make provider variation easier. If it also introduces a global plugin service, dynamic registration, remote discovery, and version negotiation for a single fixed provider, the abstraction may exceed the demonstrated need.

A useful explanation says both what becomes easier and what becomes harder. “The readiness rule has one owner, so adding a provider no longer changes publication. The cost is that each provider must translate its progress and errors into this contract.” That is more informative than calling the result clean.

The opposite problem is under-modeling. A dozen conditionals may seem smaller than an explicit state representation, but if they collectively encode a lifecycle that nobody can explain, the design is not simple in the relevant sense. Fewer declarations can conceal more mental work.

Separate policy from mechanism when they vary for different reasons

A transfer mechanism moves bytes. A retention policy decides how long to keep them. A scheduler decides when work may run. A readiness policy decides what a consumer can rely on. These may be implemented near one another, but their reasons for changing differ.

Not every separation needs a public interface. A private function or record can make a distinction clear without creating a new compatibility obligation. Public boundaries should be introduced where actual callers need them.

The level of abstraction also matters. A generic event system can represent every lifecycle transition, but may obscure the specific rules that make acquisition correct. A domain-specific operation may be easier to reason about because it narrows the permitted combinations.

Review a design through a change scenario

Choose a plausible change: add a provider, preserve a previous version, support cancellation during verification, or change the progress display. List the decisions that must be understood to make it safely.

Then ask which dependencies are necessary. If presentation depends on the meaning of ready, that is legitimate. If presentation depends on the provider's temporary filename scheme, the boundary may be leaking incidental detail.

A useful refactor reduces the number of unrelated decisions that must be held in mind for that change. It need not reduce lines or files. Test it by explaining the before and after paths to another developer. If the explanation simply substitutes five new abstractions for one old confusion, reconsider the design.

9 Design boundaries that can hold

A boundary is a set of promises between parts of a system. Its value depends on whether those promises remain meaningful when implementations vary, fail, or evolve.

An interface can have a neat schema and still be a poor boundary. If callers must know a provider's private directory layout, retry timing, or database row ordering, the abstraction has not removed those dependencies. It has merely hidden them until something changes.

Puma's ecosystem makes boundary design a recurring concern. Model management, inference execution, workflow orchestration, and application behavior have related responsibilities, but they should not automatically own one another's internal decisions.

Begin with two actual caller situations

For acquisition, one consumer might need a verified local artifact without running it. Another might need a model ready for a particular runtime. Those are related but different outcomes. If the API returns the same undifferentiated success for both, one caller may assume a stronger promise than the provider made.

Write the contract in terms of what the caller can rely on. Which identity is returned? What does readiness include? Can the resource disappear while the caller uses it? Which failure is recoverable? What happens after the caller loses the response?

A useful boundary also says what it does not promise when that omission affects action. A model manager may identify model-specific dependencies without guaranteeing that every inference backend supports them. A runtime adapter may establish its capabilities without choosing the user's preferred model. These are domain limits, not excuses for a vague interface.

Probe the capability that matters

A hardcoded version rule may be a convenient starting point, but version identity is not the same as capability. A newer runtime may preserve the required behavior. An older build may contain a backport. A provider's Python API may be distinct from the application's serving protocol.

Before adding a compatibility check, identify the boundary being tested. Does the caller need a command, an output field, a loading behavior, or a lifecycle guarantee? Probe that property when feasible and report what was observed. A version label can remain useful context without becoming an unsupported gate.

This approach has costs. A probe may be expensive, destructive, incomplete, or dependent on a model artifact. The design should choose a bounded check and retain uncertainty where appropriate. "Capability verified" must name the capability and conditions, not imply universal compatibility.

Errors are part of the contract

A caller needs to know what to do after failure. Compare “operation failed” with “candidate verification failed; the previous published version remains available; this attempt made no publication change.” The latter communicates state and next-step possibilities.

Do not expose every internal exception as a public error type. Preserve distinctions that affect recovery, authority, or user understanding. A transient transfer interruption and an unsupported format may need different responses. Two internal parsing exceptions with the same caller consequence may not.

An error can also be uncertain. If a response is lost after publication, the operation should offer a way to resolve the attempt's authoritative state. A retry without identity may be a new request; a retry bound to the original operation may be a continuation. The caller should not have to guess.

Avoid encoding one consumer as the domain

A shared layer can accidentally preserve the first application's schema as though it were universal. The second consumer then receives irrelevant fields or must mimic the first application's workflow to use the service.

The remedy is not speculative generality. Inspect the demonstrated common need. Which identity, operation, or lifecycle is shared? Which fields belong to one application? A storage layer can expose durable records without inheriting every application concept stored in them.

A boundary should make variation explicit where it is real and leave application-specific meaning with its owner. This is especially important when a shared engine is intended to serve agents that may otherwise infer undocumented requirements from examples.

Andreas Kling’s Ladybird account describes a deliberately bounded JavaScript frontend migration, with an existing implementation used for differential comparison. Agreement with the reference was a concrete target, even where it postponed idiomatic cleanup. It also carried a limit: matching the old implementation can preserve old defects. His later account of restricting outside pull requests addresses another boundary, the maintainers’ ability to accept responsibility for changes. Producing technically plausible code and admitting it into a maintained browser are different outcomes. These are project-specific accounts, not a universal policy for generated contributions. [Ladybird adopts Rust](https://ladybird.org/posts/adopting-rust/) (<https://ladybird.org/posts/adopting-rust/>); [Changing How We Develop Ladybird](https://ladybird.org/posts/changing-how-we-develop-ladybird/) (<https://ladybird.org/posts/changing-how-we-develop-ladybird/>).

Exercise the boundary from outside

Write a small consumer that uses only the documented contract. It should not import private modules or read hidden state. Make it handle normal success, a recoverable failure, and an uncertain result.

If the consumer needs an undocumented fact, decide whether that fact belongs in the contract or whether the provider should absorb the detail. If the contract requires a sequence whose middle states are unsafe for ordinary callers, consider an intent-oriented operation with explicit choices.

The strongest boundary is not the one with the fewest methods. It is the one whose promises callers can understand and whose owners can actually keep.

10 Decide when to reuse refactor or replace

Existing code deserves neither automatic preservation nor automatic suspicion. It represents working behavior, accumulated knowledge, and compatibility obligations. It may also encode assumptions that no longer fit the product.

The decision to reuse, refactor, or replace should begin with a present problem. What behavior is missing? What change is unusually difficult? What failure recurs? What information must a developer understand that should have remained irrelevant?

Without that problem, a redesign can become a preference disguised as necessity.

Reuse requires semantic fit

Two functions with similar names are not necessarily interchangeable. Reusing an API because it already exists can preserve the wrong readiness boundary or ownership rule. Conversely, writing a replacement because the old code is unfamiliar can discard a correct solution without evidence.

For acquisition, inspect whether the existing operation preserves the old version during preparation and what its success response means. If it already satisfies the needed behavior, reuse may be the best outcome. If it almost fits, identify the missing promise before introducing another subsystem.

Reuse also has an integration cost. Adapters, configuration, error translation, and lifecycle alignment can make a nominally free component expensive. Include those costs in the comparison rather than measuring only the implementation you avoid writing.

Refactor to improve a named property

A refactor should make a relevant change easier to understand or safer to perform while preserving required behavior. State that property before editing.

“Separate provider transfer from publication so a new provider cannot alter readiness semantics” is a testable rationale. “Clean up the architecture” is too broad to guide a review. It does not tell a reviewer which complexity should disappear or what must remain stable.

Research at Microsoft combined engineer reports with repository analysis and found varied benefits and costs around refactoring. A later study across 103 Java projects showed how mixed changes and attribution methods complicate claims about refactoring-related bugs. These studies do not make refactoring either harmless or inherently dangerous. They support evaluating the specific objective and separating observed association from cause. Refactoring Challenges and Benefits (<https://thomas-zimmermann.com/publications/files/kim-tse-2014.pdf>); Differentiated Replication (<https://arxiv.org/html/2009.11685v1>).

For a practical change, preserve characterization tests where they capture important behavior. Add a test for the actual property the refactor is meant to improve when that property is executable. Some benefits, such as comprehensibility, need a different assessment: a reviewer should be able to explain the new ownership path without reconstructing unrelated decisions.

Replacement needs a migration story

A replacement can be justified when the old model makes the required behavior persistently awkward or unsafe. It still needs a path from current use to the new design.

Who depends on the old contract? Can both versions coexist? How is stored data migrated? What makes rollback possible? Which compatibility behavior is genuinely required, and which exists only because nobody has removed it?

A clean new component with an unexplained transition is not a complete solution. Migration can dominate the work. It can also reveal that a smaller refactor would have been sufficient.

Avoid preserving compatibility indefinitely by default. Compatibility itself is code, testing, and conceptual overhead. But removing it should be an explicit decision grounded in actual consumers, rather than an assumption that no one needs it.

Defer a capability without abandoning changeability

Fowler's account of YAGNI distinguishes speculative features from practices that keep software changeable, such as refactoring and testing. His design-stamina discussion is explicitly a hypothesis, not a measured curve guaranteeing a payoff after a particular number of weeks. These distinctions help keep both extremes in view: premature infrastructure and avoidable decay. Yagni (<https://martinfowler.com/bliki/Yagni.html>); Design Stamina Hypothesis (<https://martinfowler.com/bliki/DesignStaminaHypothesis.html>).

For our example, resumable transfer may be a speculative capability. A clear publication boundary may already be necessary for today's cancellation behavior. Deferring the former does not justify entangling the latter.

Make the decision reviewable

Write a short note containing the current problem, evidence that the problem exists, the selected change, the behavior that must remain stable, migration costs, and the result that would show the work was worthwhile.

Then include one credible alternative. It may be reuse with a small adapter, a narrower refactor, or deliberate deferral. If you cannot explain why that alternative is insufficient, the replacement proposal may not yet be ready.

After the work, revisit the original claim. Did the change actually reduce the coupling, failure rate, or maintenance burden it targeted? If the answer is unknown, say so. A successful merge proves that a change entered the repository, not that its architectural promise was fulfilled.

11 Learn through real applications

A useful product can also be a place to investigate an uncertain idea. Those roles should be distinguished without reducing the product to an experiment.

A model-management application can provide a practical service while also letting engineers evaluate how different inference backends might work through a common API. A workflow or ledger component may be useful on its own while its lessons inform other work. Beginning experimentally does not make every current responsibility provisional.

This matters because product users and researchers need different kinds of confidence. A user needs to know what works and what is supported. An investigation needs to know which hypothesis a particular observation addresses.

Name the question inside the product

Suppose a model-management application adds a second inference backend. The product outcome might be the ability to run a supported model through either backend. The research question might be whether a shared request representation can cover both without hiding important differences.

Success for the product does not automatically answer the architectural question. The integration might work because of backend-specific exceptions that make the proposed common API unhelpful. Conversely, an unsuccessful prototype might still reveal a valuable boundary: generation parameters can be shared, while lifecycle or adapter dependencies need separate treatment.

Write the product commitment and the investigative question separately. This lets the team learn without quietly weakening what users were promised.

Transfer a lesson at the right level

A workflow ledger that works for one application may demonstrate durable event recording but not the right information model for every application. An early conflict detector may demonstrate one guarded edit path without establishing reliable massive parallelism. A graph view may help answer one navigation question without replacing all text-based development.

When transferring a lesson, identify what remained constant and what changed. The new system may have different concurrency, authority, latency, or retention requirements. A pattern is transferable only to the extent that its assumptions still hold.

A proposed adaptation between systems is not a delivered integration. The systems may have different responsibilities and constraints; a diagram connecting them does not establish that the connection works.

Make a prototype disposable in the right way

Disposable does not mean careless. A prototype should preserve enough evidence to explain what was learned. It need not retain production-quality infrastructure that was irrelevant to the question.

If the question is whether people understand a workflow without a node editor, a lightweight interface may be enough. If the question is whether a ledger can support deterministic reconstruction, a visual prototype alone is inadequate. Match the implementation effort to the uncertainty.

Hashimoto's earlier account of building Ghostty emphasizes testable increments, frequent demonstrations, disposable scaffolding, and using the product to discover necessities. It offers a useful pre-agent comparison: many of these habits are durable engineering practice rather than new consequences of language models. [My Approach to Building Large Technical Projects](https://mitchellh.com/writing/building-large-technical-projects) (<https://mitchellh.com/writing/building-large-technical-projects>).

Keep research from swallowing delivery

An application can become a perpetual source of interesting foundational work. Each new requirement reveals another abstraction that could serve future products. The exploration may be valuable and still postpone the original usable outcome indefinitely.

Define the next end-to-end result. What can a person do with the system after this slice that they cannot do now? Which unresolved idea blocks that result, and which can remain an experiment alongside it?

The answer need not always favor the smallest demo. Some infrastructure is necessary for a dependable product. The important point is to make the tradeoff explicit rather than let architectural interest silently choose the priority.

For a project containing research, create a two-column account: current product promises and open investigative questions. Take one proposed change and place its obligations in the appropriate column. If it belongs in both,

explain how the product remains usable while the hypothesis is tested. That separation protects both honest learning and reliable delivery.

Part 3 Design software that agents can use

An agent is a caller with its own discovery, feedback, and recovery needs. These chapters connect usable interfaces with authority and evidence, while keeping model judgment distinct from the conditions software can enforce.

12 Treat the agent as a real interface user

An agent-facing interface has users with limited attention, incomplete knowledge, and predictable ways of making mistakes. Those users are not human, but the design questions remain recognizable. Can they discover the right operation? Can they understand what it does? Does the response tell them how to recover? Can they distinguish a successful action from a merely valid call?

Puma treats these questions as central to the stack because agents are intended to consume and build the software. An API that is technically available but difficult for an agent to use is a product problem, not merely a prompting problem.

Examine the whole interaction

The interface includes more than the tool schema. It includes names, descriptions, available examples, discovery mechanisms, error messages, execution latency, and the state visible after an operation.

Suppose an agent must find a model, inspect its requirements, choose a runtime, acquire artifacts, wait for readiness, and request inference. If each step returns ambiguous identifiers or hides the next valid action, the agent must reconstruct the protocol from scattered clues. A large context window does not make that reconstruction reliable or cheap.

Observe a real interaction. Where does the agent guess? Where does it repeat a call because the response was unclear? Where does it retrieve an entire catalog to answer one question? Those observations suggest specific interface changes. “Make the tools better” does not.

SWE-agent's experiments provide bounded evidence that tools and feedback can affect issue-resolution performance with a model held fixed. They support treating the interaction as a design variable. They do not

establish one permanently optimal tool set for all models or repositories. SWE-agent
(<https://arxiv.org/html/2405.15793v3>).

Make mistakes informative

A schema error can be a useful teaching moment if it identifies the invalid field, the expected form, and the relevant constraint. A generic rejection may force the agent to try several guesses.

Do not make an error response silently grant authority or recommend an unsafe action. Its job is to explain the boundary. If the requested operation is not allowed on the current revision, say which precondition failed. If a capability is absent, distinguish absence from a temporary failure. If the outcome is uncertain, provide a way to inspect the authoritative state.

The response should match the caller's decision. An internal stack trace may help a maintainer but overwhelm an agent trying to determine whether retry is valid. Structured failure data and a concise explanation can serve both, with deeper diagnostics available when needed.

Prefer a demonstrated improvement to a new layer

An agent-first system does not automatically need a dedicated protocol for every action. Armin Ronacher's practical recommendations emphasize fast feedback, useful errors and logs, and application permission checks near the relevant operation. He often favors existing shell or database tools before adding MCP machinery. This is a first-person account, not a universal ranking, but it challenges the assumption that a new agent interface is inherently more usable. Agentic Coding Recommendations (<https://lucumr.pocoo.org/2025/6/12/agentic-coding/>).

For the resource example, a reliable command that returns stable identity and readiness information may be enough. A domain operation becomes valuable when it removes recurring choreography, centralizes a real invariant, or makes recovery understandable. The new layer should earn its cost through those effects.

Evaluate outcomes rather than presentation alone

A smaller catalog can reduce information load while making discovery harder. A compact response can omit a field needed for safe recovery. A friendly description can conceal a consequential default.

Compare interfaces on the same tasks. Record whether the agent achieved the intended result, made an invalid or unauthorized attempt, recovered from a failure, and required human intervention. Count calls, latency, and information volume as explanatory measures, not substitutes for success.

For an exercise, give an agent a task using an unfamiliar tool. Preserve the interaction. Identify one point where it lacked information needed for a decision. Change the interface or discovery path, then try a comparable task. A single improvement is not a general benchmark, but it can establish a concrete defect and a plausible repair.

The objective is not to prevent every mistake through longer instructions. It is to make the system's important distinctions discoverable, its operations understandable, and its failures recoverable.

13 Build interfaces around intent

An intent-oriented operation lets a caller request a meaningful outcome while the system owns the incidental steps required to achieve it. The idea is attractive because agents can otherwise spend many calls reconstructing a workflow that the software already understands.

The danger is that a convenient operation can hide decisions the caller should control.

“Make this model available” could mean download artifacts, choose a version, install a runtime, modify configuration, reserve memory, or start a process. Some of those may be required by the contract. Others may be surprising side effects. The operation is not well designed until those choices are explicit.

Compress choreography without hiding authority

Consider two interfaces. The first exposes every filesystem and catalog action. The caller must sequence them correctly and recover from partial failure. The second accepts a resource identity and readiness requirement, then performs the necessary acquisition and publication under a defined contract.

The second can be simpler for callers because the provider owns a domain invariant. It should still expose consequential choices: allowed substitutions, retention behavior, whether runtime installation is authorized, and what counts as ready.

An intent interface is therefore not a license for the system to infer every preference. It should reduce irrelevant representation knowledge while preserving the decisions that affect the user's outcome.

Add the missing domain transition

The standards engine provides a concrete example. Its authoring workflow could create initial policies for a new standard and revise existing policy scopes. But it lacked a supported way to register a new policy scope inside an existing standard. A legitimate domain transition was missing from the interface.

The repair added a registration edit within the existing proposal mechanism. The author supplies the owner and policy identity and the intended scope. The engine owns placement, registry handling, ordering, and validation. The design did not require a raw file-edit API, another service, or a second graph. Inspected tests check preservation of existing content, stable treatment of identities, grouped validation, and failures that do not mutate the predecessor. [Inspected registration tests](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L140-L211) (https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L140-L211) and [replay and mutation-resistance tests](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_replay.py#L140-L211) (https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_replay.py#L140-L211)

[Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L270-L305](#)) provide the direct examples. Historical reports record broader qualification; this book did not rerun it. [Registration plan](#) (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/plans/existing-policy-registration/plan.md>); [implementation](#) (https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/standards_engine/logical_authoring.py#L1026-L1085).

The lesson is about expressibility. A caller should be able to request the domain change it is authorized to make without learning incidental storage mechanics.

Define the success boundary

For acquisition, the operation might return a receipt identifying the attempt, selected resource, published artifact, and readiness claim. The receipt should not imply that all future uses will succeed. It establishes the bounded result promised by the operation.

Partial progress should remain distinguishable. The bytes may be present while checks are pending. A runtime may be installed while loading fails. If the operation returns early, its response should represent an admitted or running task rather than final success.

This matters for composability. A workflow engine may use the response to decide which node can run next. If it mistakes admission for completion, the convenience API has exported a race into every consumer.

Make retry semantics part of the operation

A caller that loses a response needs to know whether repeating the call continues the same intended action or starts another. Stable operation identity can help, but the meaning must be defined. An idempotency key that silently accepts different parameters is dangerous; a key that never expires may create its own retention and collision problems.

A useful design distinguishes same intent retried, changed intent, and a new attempt after a terminal result. It provides an authoritative lookup when the outcome is uncertain. The exact implementation can vary, but the caller should not have to infer these semantics from timing.

Retain an escape path at the correct level

High-level operations do not remove the need for diagnostics or specialized use. A caller may need to inspect why readiness failed or request a narrower artifact-only outcome. Provide those capabilities deliberately rather than exposing internal mutation as an emergency workaround.

The test for an intent interface is whether a new caller can use it correctly from its contract. Ask that caller to perform normal acquisition, handle a missing capability, and resolve a lost response. If each path requires

private knowledge, the interface has not yet absorbed the choreography it claims to own.

14 Connect standards knowledge and decisions

A standards document can contain valuable guidance and still be difficult to use at the moment it matters. An agent may receive too much irrelevant material, miss the rule that changes its decision, or apply a specialized workflow to an ordinary repair.

Puma's Coding-Standards work treats this as an interface and information-model problem. The standards are not merely a collection of agent skills. They include relationships, policy identities, routing, and qualification mechanisms intended to make relevant guidance usable by agents.

The distinction matters because publishing a paragraph and making it discoverable are different changes.

Route by the decision being made

A request with material uncertainty about the intended behavior may need discovery even if the eventual patch is small. A request to investigate a known technical behavior may need a bounded research path without reopening product design. A clear low-impact repair may need neither a long plan nor a complex coordination protocol.

Routing should distinguish these situations through facts relevant to the decision. It should not rely only on labels such as small, complex, or multiple agents. Those labels can correlate with a need without defining it.

For example, concurrent research does not necessarily create competing edits to shared authority. Several agents can investigate independently while one person integrates the findings. A stale-proposal protocol becomes relevant when independently prepared changes may be applied against a state that has changed. Participant count alone does not establish that risk.

A relationship graph is still a model

A graph can connect a topic, policy, workflow, consumer, and evidence record. Those edges make useful questions possible: what guidance applies here, which consumers could be affected by this revision, and what qualification supports it?

The graph does not discover all real relationships automatically. A declared consumer may be missing. A relationship may be stale. A route may be too broad. The implementation should make these possibilities inspectable rather than present the graph as complete because it is machine-readable.

The inspected discovery publication includes policy scopes and consumer relationships as well as prose. That supports a claim that the procedure is registered, not a claim that every agent selects it correctly or that all

downstream behavior has been verified. This distinction becomes important when evaluating any executable standards system. Registered policy scopes (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-units/workflow.planning.discovery.toml>), declared consumer relationships (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-impact/workflow.planning.discovery.toml#L1-L138>).

Provide the right amount of context

More instructions can increase cost and distraction. Less context can omit a crucial constraint. The design problem is relevance, not a universal preference for long or short prompts.

The inspected AGENTS.md study compared repository context arrangements across issue-solving tasks. Generated context increased average inference cost, while average success differences from no context were not statistically significant. Developer-written files did better than generated ones but did not show a significant average gain over no file. The tested settings do not establish the value of Puma's scoped engine; they are a useful counterweight to assuming that added instructions must help. Evaluating AGENTS.md (<https://arxiv.org/html/2602.11988v3>).

A practical standards interface should let the agent discover relevant guidance and inspect supporting detail when needed. It should also make omissions visible enough to correct. A small catalog that routes to the wrong rule is not an improvement merely because it uses fewer tokens.

Distinguish application from qualification

A rule can be available, selected, understood, applied, and effective. Those are separate claims.

A route test may show that a given fact pattern selects Planning. A structural check may show that a policy has a valid identity. A conversation review may show that an agent explained a consequence. An outcome observation may show that the resulting product fit the user's intent. No single check substitutes for all the others.

This is where standards work can become overbuilt. If every relationship needs elaborate proof before any ordinary change can proceed, the infrastructure may obstruct its own purpose. The system should preserve the distinctions that matter without turning all uncertainty into a blocking state.

Exercise the guidance as an interface

Take three tasks: a small clear repair, a small ambiguous request, and a larger but well-understood implementation. For each, predict which guidance should apply and why. Then observe the actual route and the agent's use of the result.

If the route is wrong, identify the missing or misleading fact. If the route is correct but the agent misuses the rule, inspect presentation and examples. If the rule is followed but the outcome is poor, revise the rule or its applicability rather than celebrating compliance.

The standards should help people and agents make better decisions. Their own design therefore needs the same outcome-based scrutiny as the software they govern.

15 Separate probabilistic judgment from deterministic control

A model can propose a design, interpret an error, or suggest a patch. A system can deterministically enforce a revision precondition, check a permission, or reject an invalid state transition. Combining these capabilities is useful, but their responsibilities should remain clear.

A deterministic check is reliable only about the predicate it actually evaluates. It does not make an unchecked proposition true.

Separate four questions

When an agent proposes a change, ask four different questions:

1. Is the proposal structurally valid?
2. Is this actor authorized to perform it?
3. Are the relevant state and revision preconditions still true?
4. Does the result serve the intended outcome?

A schema can help answer the first. An authority mechanism can answer the second within its policy. A revision guard can answer the third. The fourth usually requires a combination of domain checks, tests, observation, and judgment.

Confusing these questions creates false confidence. A valid JSON object can request the wrong action. An authorized operation can use a stale assumption. A revision-correct patch can violate an invariant the controller does not know about.

Construct a two-stage failure

In the illustrative resource system, an agent proposes changing publication behavior against revision A. Another change creates revision B before application. The controller rejects the stale proposal. That is useful: the proposal was prepared against assumptions that may no longer hold.

Now prepare a proposal against B that passes the revision check but announces ready before verification finishes. The controller accepts it if readiness semantics are not among its predicates. The first guard worked correctly; the system still needs another way to detect the second defect.

This exercise prevents a common overstatement. “Guarded application” describes a boundary. It is not synonymous with correct implementation.

CaMeL provides a related architectural comparison by separating model processing from policy-enforcing interpretation and tracking data provenance. Its guarantees depend on a defined threat model and correctly specified policies. It does not solve every misleading-text problem or remove the need to establish what the policy should be. CaMeL (<https://arxiv.org/html/2503.18813v2>).

Put the rule with its owner

A resource owner should enforce the lifecycle conditions that make its resource safe to use. A caller can validate its own inputs, but duplicating the authoritative rule across callers invites disagreement.

For publication, the component that makes a candidate visible should check the relevant readiness state at that boundary. A user interface disabling a button may help ordinary interaction, but it is not sufficient if other callers can invoke the operation.

The same principle applies to agent coordination. A proposal may describe an allowed write set, but the applying mechanism must enforce the relevant scope if correctness depends on it. A prompt instruction and an execution boundary provide different kinds of assurance.

Do not centralize every rule in one universal controller. Different owners may have distinct authority and consistency responsibilities. The design should make the handoff explicit rather than pretend a global validator can infer every domain condition.

Keep probabilistic information attributable

A model's estimate that a task is low risk may be useful input. It should remain an estimate, not silently become a verified fact. A generated explanation can help a reviewer understand a proposal, but it should not replace the actual diff or observed test result.

When uncertain judgment influences an action, record which claim was made, by whom or by which system, and which checks followed. This makes later disagreement tractable. It also prevents an interface from laundering a model's confidence into an authoritative status.

Decide what happens when evidence is missing

Some operations can proceed reversibly while uncertainty is investigated. Others require a pause because the unresolved condition affects a consequential action. That choice is part of the product's policy and risk model.

A controller that rejects every incomplete description may make ordinary work impossible. A controller that accepts every plausible proposal may leave critical authority to generation. The useful middle is explicit: which conditions are mandatory at this boundary, which are advisory, and how does a rejected operation become actionable?

For your system, list one check in each of the four categories. Then create a proposal that passes three while failing the fourth. If the team cannot explain the outcome, the boundary is probably relying on a vague idea of validation rather than a precise set of promises.

16 Record what happened and why

A record is useful when it lets someone answer a question that would otherwise remain uncertain. What was requested? Which interface was used? What response was returned? Which decision authorized a change? What state was actually observed afterward?

Puma's interest in ledgers and information graphs grows from these questions. The ambition is broader than logging errors. It includes preserving the relationship between intent, decisions, execution, and evidence so later agents and people can inspect it.

The first design task is deciding what the record can honestly claim.

Separate observation from interpretation

Suppose an agent calls an acquisition operation and receives a timeout. An observer can record the request and response. It cannot infer from that exchange alone that acquisition failed, that publication did not occur, or that the caller was wrong to make the request.

Those conclusions require other evidence. A later state lookup may show publication completed. A policy engine may report that the operation was unauthorized. A reviewer may conclude that the requested identity did not match the intended resource. These should be attributable additions, not retroactive replacements for the observed interaction.

Puma's proposed recorder design makes this distinction important: recording communication does not require the observer to decide whether an operation was malformed or semantically valid. If an interface supplies information about what should be retained, that is a separate contract. The proposal also keeps observer reporting out of the evaluated agent's input, so the act of recording does not silently become additional feedback. This is a stated design requirement, not a claim that every recorder implementation already has it.

A minimal event can be enough

For the running example, a useful record might contain an event identity, time, actor identity, operation identity, target resource, relevant revision, and a reference to the request or result. A state observation should identify what was observed and when. A decision record should identify its authority and the alternatives or consequences it preserves.

Do not add fields simply because a graph can store them. Each field creates collection, retention, interpretation, and migration obligations. A record of every byte may be expensive and may retain information nobody needs.

The W3C provenance model offers vocabulary for entities, activities, agents, derivation, and attribution. It can help distinguish relationships without prescribing one storage design. A vocabulary is not evidence that the resulting graph is complete or useful for a particular development task. [PROV Data Model](https://www.w3.org/TR/2013/REC-prov-dm-20130430/)

(<https://www.w3.org/TR/2013/REC-prov-dm-20130430/>).

Reasons are claims too

A person or model may explain why a choice was made. Preserve that explanation as an attributed account. Do not assume it captures every influence on the decision.

Research on reasoning faithfulness shows that model explanations can omit information that influenced answers in the studied settings. Those experiments do not establish a universal omission rate for coding agents. They do justify keeping a generated rationale distinct from the observable interaction and resulting state. [Reasoning Models Do Not Always Say What They Think](https://arxiv.org/abs/2505.05410v1) (<https://arxiv.org/abs/2505.05410v1>).

This distinction makes records more useful, not less. A reviewer can compare an asserted reason with the available evidence and identify a mismatch. If the record treats rationale as unquestionable truth, it prevents that comparison.

Preserve the question across changing representations

An interface version can change field names or available operations. A plan can be superseded. A source artifact can move. If the record only stores mutable labels, later readers may reconstruct the wrong context.

Stable identities and revision references help, but they should be used where they support a real reconstruction need. Not every transient observation requires a permanent globally meaningful identifier. The design should balance traceability with the cost of maintaining identity across the system.

A ledger also needs to distinguish correction from erasure. A later observation may resolve an uncertain outcome without making the original uncertainty disappear. Keeping both lets someone understand what was knowable at the time and why the next action was reasonable.

Test the record by asking a difficult question

Take one completed operation and try to reconstruct it using only the retained evidence. Which request led to it? What authority applied? Which implementation revision ran? What result was observed? What remains unknowable?

Then remove one event from the exercise. Can the record detect the gap, or does it silently imply continuity? A long log that cannot expose missing coverage may be less trustworthy than a short record that states its limits.

The purpose of provenance is not to make every decision look justified afterward. It is to preserve enough context that a later reader can understand, challenge, and improve the work.

Part 4 Turn shared understanding into coordinated work

A design must survive handoffs, parallel work, and new information. The next chapters develop compact plans, development boundaries, task ownership, and revision paths that preserve the outcome without imposing the same machinery on every change.

17 Write plans that preserve intent

A plan can be detailed enough to keep an agent busy and still be missing the reason for the work. “Add a downloader, validate the file, update the catalog” names activities. It does not say what must remain true if validation fails, which version the consumer should see during replacement, or who may decide that a different resource is an acceptable substitute. Those omissions will become implementation decisions, often without anybody noticing that a decision was made.

The handoff from discovery to implementation should preserve the information that constrains those choices. It should also discard conversational material that no longer helps. A transcript may contain abandoned suggestions, tentative interpretations, repeated explanations, and an eventual agreement. Handing all of it to an implementer can obscure which statements now govern the work.

For a substantial change, a useful plan is a small current account of the outcome, the decisions that shape it, and the evidence needed to accept it. Supporting investigations remain available by reference. The plan does not need to impersonate the entire repository's documentation.

Begin with a consequence the implementer must preserve

Our illustrative application already has a usable resource. We want to acquire a replacement, check it, and make it available through the same consumer interface. The central consequence is that a preparation failure before publication must not destroy the working version. This remains an educational example, not a description of Pumas Library's architecture.

A compact handoff might read:

Outcome: A consumer can open the requested replacement after publication. Until publication succeeds, new consumer requests continue to resolve to the previously active version.

Scope: Local acquisition, integrity checking, publication, and status reporting for one supported resource format. Automatic substitution and runtime installation are excluded from this change.

Binding decisions: Candidates are prepared outside the active location. Publication changes the active reference only after the selected checks succeed. The publication owner decides whether the transition committed; the progress display cannot make that decision.

Unresolved assumption: The supported storage environment must provide the required reference-switch behavior. Investigate this before choosing the publication implementation.

Next slice: Establish a consumer-level replacement test and implement the smallest publication path that can satisfy it. Keep acquisition behind a controlled fixture until the publication assumption is resolved.

Acceptance: Open the old version while the candidate is held before publication. Reject an invalid candidate and open the old version again. Publish a valid candidate and open the requested replacement. Report the environment and revision exercised.

This artifact connects work to behavior. It also changes the order a plausible implementation might otherwise take. Building a sophisticated downloader first would postpone the unresolved assumption that could invalidate the design. The next slice instead investigates the risky publication boundary with a deliberately uninteresting source of bytes.

The scope statement is equally consequential. An agent that discovers a missing runtime may report it and propose additional work. It cannot infer permission to install something merely because installation would help the test pass. A plan needs to distinguish an obstacle inside the agreed work from a new product commitment.

Preserve reasons selectively

Not every decision needs an essay. The reason for staging separately could be one sentence: the current version must remain usable when preparation fails. That sentence helps a later implementer recognize an apparently simpler in-place rewrite as a change in behavior, rather than a harmless refactor.

Record a rejected option when its attraction makes it likely to return, or when a new fact could make it preferable. “In-place replacement was rejected because readers could observe partial content” is useful. A catalogue of every library considered during research usually is not. If a lengthy investigation remains relevant, link its conclusion and the conditions under which it should be revisited.

Puma's inspected planning standard makes a similar proportional distinction. It permits a clear bounded repair to proceed with an inline write set and acceptance path, while requiring a written plan for material sequencing, contract, migration, coordination, rollout, risk, or acceptance complexity. The rule is an inspectable policy, not evidence that every application of it succeeds. Planning, pinned revision (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning.md#L31-L49>).

For a spelling correction in a known status label, the plan might be a sentence. For changing what that label means across a service, API, and interface, the contract must survive several handoffs. The distinction follows the decisions that need preservation, not the number of edited files.

Make stale understanding detectable

A plan becomes dangerous when it describes a world implementation has already disproved. Suppose inspection reveals that an existing reader holds the active path open throughout its work. The statement about “new consumer requests” remains useful, but the plan now needs to distinguish new readers from existing ones. The implementer should surface that distinction before quietly making old readers follow the replacement too.

One current decision owner prevents competing summaries from becoming competing authorities. A task brief can quote the binding rule and link back to it; it should not become an independently editable copy. When the rule changes, affected task owners need the change and its implications, not merely a notification that the document was updated.

There is an opposite failure: keeping the plan impeccable while useful work stalls. A discovered alternative can be documented without reopening settled choices that it does not affect. Investigation can also end with the conclusion that no change is needed. A plan is valuable when it helps choose and finish work, including deciding not to start it.

Try the handoff without the conversation

Choose a real change and give another person, or a fresh agent session, only its compact plan and the referenced repository. Ask for the intended consumer behavior, the first implementation slice, and one condition that requires a new decision. Do not grade whether the answer copies your wording. Check whether it preserves the consequential distinctions.

If the recipient can produce a plausible implementation that violates your intention while satisfying every sentence, add the missing invariant or example. If they spend their effort reconciling duplicated guidance,

remove a competing copy. The observable outcome is a handoff that permits independent progress while making the next consequential misunderstanding visible.

18 Design the development path

The order in which software is built affects what can be learned and what can be discarded. A team that integrates three weeks of acquisition work before testing publication has made the publication decision expensive to revisit. A team that creates a branch for every minor step may spend more effort maintaining branch relationships than learning about the product. Both problems begin before the code is wrong.

Designing the development path means choosing boundaries where a coherent result becomes inspectable. Sometimes that boundary is a commit. Sometimes it is a disposable experiment, a pull request, or a demonstration against the actual consumer. Git can preserve the boundary, but Git topology cannot supply its purpose.

Puma identified branch planning as something to improve. The inspected standards subsequently ask what each development or integration boundary makes visible and what evidence would reopen the design; they explicitly avoid a branch-per-milestone or merge-per-slice rule. That establishes a stated direction, without establishing a measured reduction in revision cost. Development and integration boundaries (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning.md#L161-L184>).

Put the earliest useful observation first

For the running resource example, the first useful result need not contact a real download service. A controlled candidate can establish whether the consumer sees the old version before publication and the replacement afterward. This makes a risky lifecycle assumption observable with little implementation attached to it.

The next boundary can add the real acquisition adapter. Another can expose status through the actual caller. These are vertical observations: each connects a piece of production behavior to something outside that piece. Dividing the work into “all types,” “all services,” and “all interface code” may produce tidy assignments while postponing the first meaningful result until every assignment finishes.

A proposed development map could be expressed in ordinary text:

Base A: Existing consumer behavior is characterized.

Publication slice, A to B: A controlled candidate can replace the active reference; failed preparation leaves the old version usable. One integration owner accepts B.

From B, two proposals: The acquisition adapter supplies checked candidates. The interface consumes a fixed status contract. Each proposal names B as its starting contract.

Integration C: Exercise the actual caller through acquisition, checking, publication, and failure. Accept the combination only after verifying the shared meaning of completion.

The dependency shape is small enough to see at a glance:

```
A: existing behavior
    |
B: publication contract
    /  \
  adapter interface
    \  /
C: combined acceptance
```

The letters identify observable development boundaries, not a required Git history.

This map has one deliberately serial step. The two later proposals cannot safely work independently until the status and publication contract is understandable. Starting both earlier is possible, but then their work is exploratory and may need revision. Calling the uncertainty “parallelism” does not remove it.

Give isolation a job

A worktree helps two workers edit without overwriting each other's uncommitted changes. A branch can keep an experiment out of a release candidate. Neither establishes compatibility. The interface branch may interpret completion as “bytes received,” while the acquisition branch means “consumer publication settled.” Their files can merge cleanly and their individual tests can pass.

The integration contract therefore belongs beside the branch map. In our example it says that a candidate is not available merely because checking succeeded, and that the active reference changes only at the agreed publication boundary. Both authors can test against this contract before integration.

Isolation also needs an ending. A retained branch should have a continuing purpose. A superseded experiment needs a record of what was learned and which commits, if any, remain useful. An abandoned worktree containing the only copy of unfinished work should not disappear during mechanical cleanup. Operational resources and durable explanation have different lifetimes.

There is no advantage in manufacturing these obligations for a small serial repair. If one person can make, inspect, and verify a bounded change on the allowed integration path, a separate branch may add no useful isolation. Repository rules can still require one; then the reason is institutional, not an intrinsic property of the change.

Keep a serial alternative on the table

Mitchell Hashimoto's February 2026 adoption account describes one background coding agent while he continues manual work. He distinguishes that arrangement from parallel research and triage, and avoids assigning work merely to keep agents occupied. The report is a personal workflow, not a comparative trial, but it supplies a serious alternative to a permanent implementation swarm. [My AI Adoption Journey](https://mitchellh.com/writing/my-ai-adoption-journey)

(<https://mitchellh.com/writing/my-ai-adoption-journey>).

The architectural question is whether extra concurrency shortens the path to an accepted result. Two separable tasks with stable inputs may benefit. Two tasks repeatedly changing the same contract may benefit more from a brief serial decision followed by independent implementation. A worker who finishes quickly but waits a day for integration has improved local throughput without necessarily improving delivery.

One useful counterexample is a format-version migration. The parser, writer, and compatibility behavior may be so tightly linked that dividing them between agents creates more assumptions than capacity. A single implementation owner with a separate reviewer can preserve focus while still providing another perspective. Agent count is a resource choice, not a maturity score.

Rehearse a change in direction

Draw the proposed development path for your next substantial change. At each boundary, write the behavior someone can observe and the fact that would make you reconsider. Then introduce a plausible surprise halfway through: the consumer needs an additional file, publication behaves differently on another platform, or the intended API is unavailable.

Identify what survives, what must be redone, and which unintegrated work has become stale. If every discovery forces the whole effort to restart, look for an earlier observation or a smaller commitment. If every boundary exists only because it seemed orderly, combine some. The exercise succeeds when a reviewer can trace a changed assumption to a specific reconsideration point and see the valid work that remains.

19 Give agents bounded responsibilities

“Implement acquisition” is a goal with a surprisingly large surface area. An agent might change storage, replace the networking library, alter error handling, add retries, and update the interface. Each action can sound helpful. Together they can outrun the authority and review capacity of the person who delegated the work.

A bounded responsibility joins an outcome to a permitted area of action. It gives the worker enough room to solve the assigned problem while identifying choices that belong elsewhere. The boundary must be understandable to the worker and enforceable where enforcement is warranted. A sentence asking the agent to be careful does not isolate its edits or prevent a release.

A handoff is a working contract

Consider the acquisition adapter after the publication contract has been established. Its handoff could be:

Task: Supply a checked candidate to the existing publication interface for the selected local resource.

Inputs: The selected identity, source location, expected integrity evidence, the publication contract at the agreed base, and the consumer acceptance examples.

Owned changes: The acquisition adapter and its focused tests. Shared status types are read-only until a proposed contract change is accepted by their owner.

Required behavior: Acquisition failure leaves the active version unchanged. A retry must not silently select different content. Successful transfer alone must not emit the consumer's available state.

Evidence to return: The resulting revision or patch, the focused check results, one failed-acquisition observation, and any assumptions that remain untested.

Escalation: Report an unsupported source identity, an inadequate publication contract, or a required change outside the write set. Explain the smallest decision needed and continue independent work where possible.

Integration: The integration owner reviews and applies the accepted proposal. The worker does not publish a release.

The handoff is short enough to revisit. Its restriction on shared types does not forbid noticing a design problem. It tells the worker how to make that problem visible without silently taking ownership of somebody else's decision.

An overly narrow assignment can fail too. “Change only line 42” may force a patch at the symptom when the defect belongs to a different owner. Bounded authority should permit diagnosis and a proposed scope change. It should not reward compliance with an inadequate edit location.

Separate four kinds of ownership

Task ownership answers who is responsible for reaching a result. Write permission answers what that worker may mutate. Integration ownership answers who combines the work with current authority. Acceptance ownership answers who decides that the result satisfies the intended need.

These roles can belong to one person in a small change. They still answer different questions. An implementation agent may own its task while having no authority to approve a change in user-visible behavior. A reviewer may have broad read access while remaining unable to modify the candidate. A release owner may accept a verified change without having written it.

The separation becomes useful when work stops unexpectedly. If the caller disconnects, does the task disappear, pause, or continue? Which resources remain held? Where can a replacement worker learn the current status? Tying every answer to the lifetime of one conversation creates avoidable uncertainty.

At Passeur revision `71bcac5a`, task admission creates a task-owned cancellation controller rather than reusing the caller's signal. The inspected lifecycle tests distinguish a disconnected submitter or cancelled wait from cancellation of the worker. They also check a finite worker-plus-queue capacity. These are concrete source and test assertions; this book's research did not execute them. Task admission

(<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/src/core/coordinator.ts#L413-L450>), lifecycle tests

(<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/core/durable-lifecycle.test.mjs#L52-L94>).

That distinction does not imply that every task should continue unattended. The appropriate policy depends on the authority originally granted, the resources in use, and the ability to observe or stop the work. The useful lesson is to make continued execution a deliberate contract rather than a side effect of a lost connection.

Make reporting part of useful work

A worker should report a finding when it changes a decision, not only when the task is over. Suppose the adapter discovers that the requested resource consists of two files. Waiting until the final report to mention this leaves the interface and tests developing against a false single-file assumption. An early update can be brief: what was found, which assumption it affects, and which work can continue.

The opposite failure is a stream of routine narration that consumes the coordinator's attention. Reports should carry information someone can act on: a conflict, a completed usable part, a needed decision, a failed acceptance check, or a material change in scope. The coordinator remains responsible for translating those findings into a coherent account for the requester.

A final response of “done” is inadequate when the task contract asked for a patch and evidence. A long explanation without a retrievable result is inadequate too. Return the work in a form that another person can inspect, including failure and uncertainty. Reviewability is a property of the deliverable.

Test whether the boundary is real

Give a worker the sample handoff and then introduce three situations: an unrelated file needs cleanup, a shared type seems wrong, and the source returns content with an unexpected identity. Ask what the worker would change, report, or refuse to infer.

The observable outcome is not a perfect recitation of permissions. It is that the worker can continue useful authorized work, identify the decision owner, and avoid converting convenience into authority. If every surprise blocks the whole task, loosen the boundary around diagnosis. If every surprise justifies broader edits, tighten the distinction between proposing and deciding.

20 Coordinate parallel work while it happens

Two agents can work in different files and still implement incompatible systems. One changes a status producer so that `complete` means transfer has ended. Another changes a consumer so that `complete` enables immediate use. No line collides. The combined application announces readiness before publication finishes.

This is why coordination cannot be postponed to the merge command. File isolation solves one class of interference. It does not align meanings, preserve decision authority, or discover that two apparently independent tasks rely on the same assumption.

Before splitting work, identify the shared promises. For our resource example, these include resource identity, the meaning of availability, the publication boundary, and the survival of the previous version after failure. Workers can own disjoint implementations of those promises. They cannot independently redefine them and expect a clean merge to reconcile the result.

Detect the kind of conflict

A textual conflict is visible in overlapping edits. A semantic conflict appears when compatible text produces incompatible behavior. An ownership conflict occurs when two workers each believe they may decide the same change. A stale-assumption conflict arises when an otherwise valid proposal depends on a contract that has since changed.

These distinctions suggest different responses. Overlapping comments may need a trivial edit. A disagreement about readiness needs a decision and affected tests. A stale proposal may need only revalidation if the intervening change is irrelevant; automatically throwing it away wastes work. Automatically applying it risks restoring an obsolete decision.

In the running example, imagine two workers propose changes involving `ResourceStatus`. The acquisition worker wants to add `checked`; the interface worker wants to treat every state other than `acquiring` as usable. Early coordination could produce this record:

Detected dependency: Both proposals depend on the meaning of readiness.

Proposal: Add a distinct checked state. Keep usability tied to publication, not to absence of transfer.

Acquisition owner: Confirms it can emit the new state without changing publication ownership.

Interface owner: Confirms it will remain in a preparing presentation until publication settles; requests a fixture for the held boundary.

Integration owner: Accepts the contract revision, updates the shared fixture, and asks both workers to revalidate against that revision before application.

Consent here means each affected owner has actually considered the proposed obligation. A coordinator should not write “all agreed” because no worker objected before a deadline. Nor should a worker's agreement be treated as authority to override the requester on a product decision.

Guard application where stale state matters

Once work has been prepared against a shared base, integration must check whether that base still applies. A proposal can name the contract revision it assumes and the write set it intends. The integrator compares those assumptions to current state before applying it. If they differ, the next step is classification: irrelevant change, compatible adjustment, or a decision that needs reopening.

Such a mechanism is useful when several pending proposals can become stale. It may be needless machinery for two read-only researchers and one serial writer. The risk follows mutable shared authority, not participant count.

Coordination state itself can become stale. A reservation left by a stopped worker should not block the repository forever, but an old heartbeat does not prove that the owner is gone. A paused process can still own a lock. A new process can reuse a familiar identifier. Recovery is another concurrency problem, with a more dangerous opportunity to mistake uncertainty for permission.

Passeur's September 25 repair addresses this at a specific owner. Independent review found a stale-lease recovery race; the implementation introduced a persistent recovery guard and exact-generation checks. Source tests include sixteen competing reclaimers with one successful acquisition, and a paused child that continues to exclude a contender. The recorded checks are bounded Linux/process evidence, not proof of every host recovery path. Repair record (<https://github.com/MrScripty/Passeur/commit/cae37226d2d222644c85b4974e35a6c3555b0bf6>), competing reclaimers (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/core/lease-recovery.test.mjs#L119-L178>), paused owner (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/native/shared-service.test.mjs#L73-L83>).

The application-level lesson is narrower than “add locks.” Mutation authority needs evidence appropriate to the object being protected. The identifier of a task, the generation of a repository lease, and permission to alter a product contract are different kinds of evidence.

Ask whether the work should be parallel

Coordination has a cost even when it succeeds. Stable independent tasks can amortize that cost. Rapidly changing shared concepts may not. The scaling-agent study provides a useful restraint: its results vary with task structure, and its software evaluations use only twenty tasks in each of two benchmarks. It cannot establish a generally optimal number of coding agents. Towards a Science of Scaling Agent Systems, version 3 (<https://arxiv.org/html/2512.08296v3>).

A practical alternative is to keep one implementation owner and parallelize investigation, test design, or review. Another is to stabilize one small shared contract before opening several write assignments. These choices do not forgo all concurrency. They place it where workers can make progress without repeatedly renegotiating their inputs.

Exercise a clean merge that fails

Create two small changes in the illustrative application: one changes when the producer emits completion; the other changes when the consumer enables use. Keep their write sets disjoint. Verify that both focused suites can pass while a held-publication integration test fails.

Then introduce a shared readiness contract and repeat the handoff. Record when the incompatibility becomes visible and how much work must be revised. Finally, make one proposal stale by changing the contract before application. The observable outcome is early detection plus a justified integration decision, not merely a successful text merge or a larger coordination log.

21 Match models to the work

The cheapest model call is not necessarily the cheapest accepted change. A worker can generate a plausible patch quickly, then consume an hour of expert attention repairing its assumptions. A more capable model can spend unnecessary time exploring a task that a short deterministic procedure would have resolved. Choosing between them requires an account of the work, not just a ranking of models.

Cost pressure is a reason to evaluate allocation, not evidence that a particular routing policy is already effective. Model capabilities, prices, interfaces, and rate limits change. A durable method should help select the current arrangement without turning today's favorite model into an architectural dependency.

Describe the judgment the task requires

“Coding” contains several different activities. Locating the source of an error is not the same task as changing a known expression. Comparing two lifecycle designs is not the same as translating a stable interface. Reviewing a concurrency repair requires evidence and skepticism that a broad prose summary may not exercise.

For the resource application, consider three assignments:

1. Investigate whether the existing publication operation preserves the old active resource after a failed candidate
2. Implement a bounded acquisition adapter against a stable publication contract
3. Review a candidate for premature completion and resource-lifetime defects

The investigation needs relevant retrieval and calibrated uncertainty. The implementation needs reliable local editing and feedback. The review needs the ability to challenge a plausible explanation and construct a counterexample. A model that performs well on one may be poorly suited to another. The surrounding tools and supplied evidence can change the result as much as the model choice.

Some tasks should not be routed to open-ended judgment at all. A known formatting transformation or schema validation may already have a reliable tool. An agent can invoke it without being asked to reinvent its rules.

Build a small evaluation around acceptance

A useful first evaluation can be modest. Select several representative, already understood tasks from each class. Keep their desired outcomes and environments stable. Give each candidate the inputs it would receive in actual work. Save the model version, relevant settings, tool access, and supplied context so that a later result is interpretable.

The evaluation record might contain:

Task family: Bounded adapters for local resources.

Required result: The requested resource is acquired and checked; failed preparation preserves the active version; no unauthorized contract expansion occurs.

Candidate arrangements: One model with the existing tools; another with the same tools; a prescribed localization-and-edit procedure using a model only at designated steps.

Observations: Accepted or rejected result, reason for rejection, elapsed time, human intervention time, tool and inference cost, retries, and integration repairs.

Hard constraints: No unapproved dependencies, no changes outside the task's authority, and no acceptance based solely on the worker's report.

Decision: Prefer the least costly arrangement that meets the required quality and risk constraints for this task family; retain an escalation path for cases outside the tested range.

This is a proposed experiment, not a table of measured results. Its most important property is that all candidates must earn the same kind of acceptance. A cheap candidate should not receive easier tests, while an expensive one is charged for more demanding scrutiny.

The simpler arrangement deserves genuine consideration. Agentless studied a prescribed localization, repair, and validation pipeline and obtained competitive results on its historical benchmark without an open-ended agent workflow. It still used language models. Its relevance here is the existence of a credible procedural baseline, not a claim that interactive agents are unnecessary for current development. [Agentless, version 2](https://arxiv.org/html/2407.01489v2) (<https://arxiv.org/html/2407.01489v2>).

Count the expensive parts you can overlook

Inference spend is observable and easy to compare, which makes it tempting to treat it as the whole bill. Include repeated context, tool execution, test environments, unsuccessful attempts, merge repair, and human attention. Report elapsed delivery time separately: a background task can take longer on the clock while requiring less active supervision.

Avoid collapsing everything into one monetary number unless the conversions serve an actual decision. Ten minutes of scarce maintainer attention may matter more than a small cloud charge, but assigning an invented hourly value can hide that fact behind false precision. A small set of dimensions is often more honest: accepted quality, human minutes, elapsed time, and direct cost.

Sample size limits what the evaluation can settle. One success is a useful demonstration. It is weak evidence for routine reliability. One unusually difficult task can also distort a tiny comparison. Inspect failure patterns, repeat material cases when feasible, and report how much of the real workload the sample represents.

The evaluation can reveal that task design is the bottleneck. If every model misunderstands whether checking implies availability, routing among them may have little value until the contract is repaired. Likewise, if every candidate spends most of its time waiting for a build, a faster model may not shorten delivery.

Use escalation without surrendering authority

A practical routing policy can start bounded tasks with a less expensive arrangement and escalate on named signals: repeated failure at the same boundary, unresolved architectural uncertainty, or an inability to produce trustworthy evidence. Escalation should carry the current state and failed approaches, not ask a new model to reconstruct everything from a triumphant summary.

Changing models does not grant new permissions. A stronger reviewer is still a reviewer; a stronger implementer still owns the same task. Nor does provider diversity establish independent reasoning. Models can share missing context, common patterns, and the same mistaken oracle.

Test one allocation decision

Choose a task family you actually repeat. Compare two arrangements on a small set of representative cases and include at least one unsuccessful attempt in the record. Ask a reviewer who did not generate the patches to apply the same acceptance criteria.

The observable outcome is a bounded rule such as “use arrangement A for these stable adapters, escalate when publication semantics change,” with the evidence and uncertainty attached. If the sample cannot distinguish the candidates, say so. An inconclusive comparison is more useful than a permanent model hierarchy built from impression.

22 Revise the plan without losing the goal

Implementation is an encounter with facts the plan could not fully contain. The resource has an auxiliary file nobody mentioned. A filesystem operation behaves differently under the supported deployment. A consumer needs metadata before it can open otherwise valid bytes. Finding one of these conditions is not itself a planning failure. The failure is to conceal the discovery, or to treat every discovery as permission to redefine the outcome.

The plan should be revisable because the goal matters. Its current form is a means of preserving decisions while work proceeds. Once an assumption becomes false, faithful execution can become a way of building the wrong thing more efficiently.

Classify the new fact before changing the work

Suppose the acquisition adapter now obtains both the main resource and an auxiliary file. The existing progress calculation uses the known size of the main resource as its total, while received bytes include both files. There are several possible questions here, and they should not be bundled into one vague request to “fix progress.”

The known total may be a false measurement assumption. The meaning of completion may be an inadequate contract. The presentation of unknown progress may be an unresolved product choice. A proposal to substitute a different resource is a change of scope. Each belongs to a different decision, even if one test first exposes all of them.

For our illustrative application, assume the main requirement remains: a valid selected replacement becomes usable, while the old version survives a preparation failure before publication. Discovering an auxiliary file does not automatically change that goal. It changes what acquisition must include and what can truthfully be reported during preparation.

A revision record could say:

Observation: The selected resource includes a required auxiliary file whose size is unavailable before transfer.

Invalidated assumption: The known main-file size represents total acquisition size.

Preserved decisions: Selection identity remains fixed. Publication waits for all required checking. The active version survives failure before publication. An uncertain publication result requires reconciliation, not an assumed rollback.

Changed contract: Total size and estimated remaining time may be unknown. Availability continues to depend on publication, independent of any displayed fraction.

Affected work: Acquisition accounting, status serialization, interface presentation, and tests that assume a known denominator.

Decision needed: Select how the interface communicates ongoing work when the total is unknown. The measurement must remain unknown regardless of the presentation choice.

This record makes it possible to keep valid implementation. It also prevents a convenient percentage from becoming permission to invent a total.

Propagate consequences, not just text

Updating the plan is the beginning of a revision. Each affected worker needs to know which assumption changed and whether its current proposal remains valid. Tests may need a new fixture. An API example may need to represent absence. A previously reviewed diff may now require another look at the changed contract.

The integration owner should classify work rather than resetting everything. An adapter that already carries an optional total may remain usable. A presentation that divides by the total without checking absence does not. A publication test can remain valid if it never depended on progress arithmetic. This selective approach reduces rework without pretending the change has no consequences.

Keep the history of the previous decision intelligible. A future maintainer should be able to tell that a number was removed because its evidence was inadequate, not because somebody forgot to calculate it. The current plan should contain the current rule; a short dated record can preserve why it changed. Two competing active versions create uncertainty rather than traceability.

Sometimes the rule is what needs repair

It is tempting to respond to every failure by adding a stricter gate. A gate can also be based on the wrong ownership model. Passeur's October 1 record describes a Muse task whose native work produced a commit but was classified as interrupted because ordinary SDK startup lacked a protected-launcher stop proof. The repair made successful SDK close the ordinary requirement and demanded additional proof only when that launcher owned it. Recorded focused checks passed, while broader failures remained unqualified. This was a correction to an evidence contract, not permission to assume shutdown. [Operational report](#)

(<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/docs/plans/global-passeur-profile-upgrade/reports/verification.md#L97-L105>), [adapter repair](#)

(<https://github.com/MrScripty/Passeur/commit/be0185898f5013f551a008d51c826fb6ffaea86e>).

The important question is whether a requirement protects a real obligation at the correct owner. “More checks” does not answer it. A rule can reject valid work, obscure responsibility, or make the system impossible to operate without improving the outcome it was meant to protect.

Steve Yegge's September 2026 Wheelhouse account offers a larger, explicitly personal example: he describes accumulated constraints stopping progress and a substantial reduction in refusal sites. The account also discusses spending and task-sizing pressures. These are operator reports rather than controlled findings, but they challenge the assumption that adding orchestration rules is a one-way improvement. Seats and Sunsets

(<https://yegge.ai/essays/seats-and-sunsets/>).

Removing an unnecessary rule still requires understanding its purpose. Deleting a failing acceptance test solely to obtain a green run erases the question the test was asking. Revising a test because it encoded a disproved assumption can be correct, provided the intended outcome is checked by a better observation.

Decide when to involve the requester

An implementer can often correct an internal assumption within the granted scope. A change to the promised behavior, supported audience, cost, privacy, or release commitment belongs with the person authorized to decide it. The useful escalation explains the consequence and offers a bounded choice. It does not ask the requester to resolve an implementation detail that the developer already understands.

If the requester cannot decide immediately, preserve the working version and continue independent work. Do not use silence as acceptance of a broadened objective. Equally, do not freeze unrelated tasks whose inputs remain valid.

Rehearse an amendment

Take the plan from chapter 17 and add the unknown-size auxiliary file. Identify every assumption that changes, then mark each pending task as unaffected, needing adjustment, or blocked on a decision. Rewrite one acceptance example before changing implementation.

The exercise succeeds when another reader can explain both what changed and what still governs the work. They should also be able to identify who decides the unresolved presentation choice. A revised plan that merely matches the current code has not yet demonstrated that it preserves the goal.

Part 5 Establish that the result deserves trust

Completion requires a bounded account of what was observed and what remains unknown. Tests, review, failure experiments, and measurements answer different questions; the following chapters explain how they can support a useful acceptance decision.

23 Test the outcome as well as implementation

The download test passes. The expected bytes exist. The checksum matches. A user selects the resource and receives “not available.” Every preceding observation can be true. The missing link is between what the implementation checked and what the user was promised.

Testing an outcome begins with the action that should become possible. In the running example, a consumer must open the selected replacement through the supported interface after publication. Before that boundary, the old version must remain usable. A test that inspects only a staging directory cannot establish either claim.

This does not make lower-level tests unimportant. They locate defects and exercise cases that would be expensive through the whole application. The challenge is to build an evidence chain whose links answer different questions, rather than accumulate many assertions about the same intermediate event.

Name the oracle

A test oracle is the basis for deciding whether an observed result is correct. Sometimes it is a simple expected value. Sometimes it is a domain invariant, a known implementation, or a person's judgment about behavior. Whatever its form, it can be wrong.

For acquisition, “the returned status equals `complete`” is a weak oracle if the implementation itself decides when to return that status. A stronger acceptance observation asks the real consumer to open the requested identity and checks the content or behavior it receives. The status can then be checked against that independent consequence.

Independence is practical rather than absolute. The consumer might share the same faulty catalog code. A checksum might come from the same untrusted source as the bytes. A second implementation might preserve a defect in the first. State what each oracle depends on, especially where agreement could reflect a shared mistake.

Test both sides of a boundary

A useful publication test is a sequence, not just an eventual success assertion. In the illustrative application:

1. Install a known old version and prove the consumer can use it
2. Begin preparing a valid replacement, then hold execution immediately before publication
3. Confirm that a new consumer request still resolves to the old version and that status does not claim the replacement is available
4. Release publication and confirm that a new request resolves to the selected replacement
5. Repeat with an invalid candidate and confirm that the old version remains usable after the failure

The controlled hold makes a normally short interval observable. Without it, a fast machine may race past the defect so that a polling test never sees it. The barrier must be released during teardown even when an assertion fails; otherwise the test can strand the worker it was meant to inspect.

Pumas provides a bounded real counterpart. At draft head `42a198bf`, a public HF status test holds the importer at a metadata-write barrier. The input files already exist, but the test requires nonterminal status and no completion receipt. After release, it checks completion alongside the receipt, model/index presence, adopted acquisition state, and queue release. The repository reports a focused pass; the test was not rerun for this book. It exercises a controlled Linux public-core/DTO path, not JSON-RPC, Electron, or every supported platform.

Public-boundary test (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-core/src/api/hf.rs#L1985-L2378>), **recorded execution and limits** (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/docs/plans/artifact-acquisition/execution-ledger.md#L707-L720>).

The transferable technique is to make the consequential interval testable and inspect what consumers can observe there. The particular internal state names belong to that repository.

Include the actual entry point

A function-level test can establish a sound core while a wrapper mishandles units, errors, or completion. At least one acceptance path should therefore use the interface the relevant consumer actually calls. For a command-line tool, invoke the command. For an API, exercise the protocol boundary. For a desktop interaction, inspect the interface and its behavior under the intended conditions.

Simon Willison's account of agentic manual testing advocates this complementary practice: start the system, invoke its interfaces, inspect observed output, and convert useful discoveries into regressions. His Showboat approach records commands and output rather than relying only on an agent's narrative. Such a demonstration covers the behavior exercised; it is not a proof of unvisited paths. [Agentic manual testing](https://simonwillison.net/guides/agentic-engineering-patterns/agentic-manual-testing/) (<https://simonwillison.net/guides/agentic-engineering-patterns/agentic-manual-testing/>).

Exploration and repeatable tests serve different purposes. Exploration can reveal a confusing status transition nobody thought to specify. A regression can make that discovered requirement durable. Automating every exploratory observation immediately may be wasteful; leaving a consequential repeated failure dependent on memory is wasteful in another way.

Make a passing suite vulnerable to a meaningful fault

Ask what deliberate defect should cause the suite to fail. Move the availability signal before publication. Replace the requested identity with a different valid resource. Delete the old active version before checking the candidate. If a suite still passes, it does not establish the corresponding claim, however large its test count.

This is a useful local mutation exercise, not a demand to construct an exhaustive mutation-testing framework. Choose a fault tied to a material promise. The result often exposes that several tests copied the implementation's

assumptions rather than checking an external consequence.

Acceptance can also require human observation. A technically accurate progress display might be misleading because the user reasonably interprets a finished bar as permission to close the application. Describe that interpretation and inspect the actual flow. Human judgment should address the unresolved experience, not replace checks that the machine can perform precisely.

Build one complete evidence chain

Take a change whose tests already pass. Write the promised consumer action, the observation that proves it became possible, and one plausible defect the current tests could miss. Add or perform the smallest check that distinguishes the defect from the desired behavior.

Record the exercised revision and environment, then state one important limit. The observable outcome is an acceptance claim supported by a relevant observation, including a failure case. A larger passing count is optional; a clearer connection to the intended result is the point.

24 Use independent review effectively

A second agent reads a patch and agrees with the first. This may be useful, but the number two does not establish independence. Both agents may have received the same mistaken requirement, trusted the same passing test, and followed the same persuasive explanation. Two approvals can therefore be two presentations of one blind spot.

Effective review creates an opportunity to challenge the work from a different position. The reviewer needs enough context to understand the promise, enough access to inspect the result, and enough freedom to disagree with the implementer's account. The goal is a justified acceptance decision, not a ritual exchange of approval.

Give the reviewer a question worth answering

“Review this for bugs” can be a useful broad request, but it should not be the only instruction for a consequential boundary. For the acquisition change, ask whether failed replacement preserves the usable old version, whether completion corresponds to consumer availability, and whether cancellation can release resources still in use.

Supply the exact candidate revision, the intended behavior, the diff, relevant contracts, and reproducible evidence. Include known failures and unrun checks. Let the reviewer inspect the implementer's rationale, but do not make agreement with it the assignment. A reviewer who cannot verify a claim should be able to report that limitation without inventing a defect.

A bounded review brief could read:

Review candidate R against base B for the replacement lifecycle. Focus on publication ordering, old-version survival, and cancellation while blocking work remains active. Inspect production paths and the tests that claim to cover them. Do not modify the candidate. Return actionable findings, material uncertainty, and the scope you actually inspected. A clean report applies only to R and that scope.

The read-only role makes it easier to separate findings from repairs. If the reviewer silently fixes the code, the author may never understand the defect, and the reviewed revision becomes ambiguous. A reviewer can propose a repair or be given a subsequent implementation task; the transition should be explicit.

Require a finding to carry its consequence

Consider an illustrative finding:

Behavior: Cancelling the public waiter releases the candidate directory while a blocking checker can still read it.

Location: The boundary that starts the checker retains the task handle but does not retain the directory's ownership capability.

Consequence: A replacement attempt can acquire the same directory before the first checker finishes.

Counterexample: Hold the checker, cancel the waiter, and try the competing acquisition. It must remain excluded until the checker returns.

Uncertainty: This follows from the inspected lifetime path; the competing acquisition has not yet been exercised on the target platform.

This report is useful even before a patch exists. It names the violated promise, the mechanism, a distinguishing observation, and its present evidential limit. “Race condition, high severity” without those details can consume more attention than it saves.

Severity should follow consequence and reachability, not dramatic terminology. A race in an unreachable fixture differs from a race at a public cancellation boundary. A style disagreement can be valuable without being a release blocker. The reviewer should distinguish a demonstrated defect, a plausible risk requiring investigation, and a preferred alternative.

Allocate human understanding deliberately

Practitioners disagree about how much source a human must read. That disagreement deserves more than a compromise slogan. Simon Willison argues that unreviewed generated patches can transfer the real work to collaborators; his proposed contribution standard includes a small understood change and evidence of testing.

This is a responsibility norm rather than a measured defect-detection rate. Anti-patterns

(<https://simonwillison.net/guides/agent-engineering-patterns/anti-patterns/>).

Salvatore Sanfilippo's July 2026 essay places more emphasis on controlling design ideas, comparative checks, performance, and QA than on routinely reading every generated line. He still describes reading Redis code for its maintainers and users, and leaves some skill-development questions open. His position challenges universal line-review requirements without removing the need to understand the system. Control the ideas, not the code

(<https://antirez.com/news/169>).

Daniel Stenberg describes AI review after human review while retaining human architectural and merge responsibility for curl. That is a different allocation suited to a long-lived shared system, not proof that a particular sequence always finds more defects. A human in control (<https://daniel.haxx.se/blog/2026/06/10/a-human-in-control/>).

The decision should reflect what must be understood and who will maintain the result. In a disposable personal experiment, behavior-focused QA may be sufficient. In a shared parser that handles hostile input, accepting a large generated change without a maintainer able to reason about its structure may be unacceptable. These settings do not differ merely in how enthusiastic their authors are about AI.

Design review, source review, runtime testing, and differential comparison reveal different failure classes. One cannot be substituted for another solely because it is easier to automate. Conversely, requiring every method on every tiny change can make review too costly to perform well.

Close findings against the repaired revision

Once the author changes the candidate, verify that the repair addresses the counterexample and preserves neighboring behavior. A passing new test is useful, but inspect whether the test actually reaches the former defect. Review the incremental repair instead of repeatedly regenerating the whole implementation and forcing everyone to start again.

The final acceptance record should name which findings were fixed, rejected with reasons, or deferred under an explicit decision. A later unrelated commit does not inherit an unlimited clean bill of health. Nor must a bounded review expand forever into every architectural concern a reviewer can imagine.

For an exercise, give a reviewer the chapter 23 acceptance claim and a candidate with premature completion deliberately introduced. Withhold your proposed diagnosis. Ask for a counterexample and a scope-limited conclusion. Repair the defect, then ask the reviewer to check the new revision. The observable result is a traceable finding-to-repair-to-verification sequence, including any claim the reviewer could not establish.

25 Design for interruption and failure

An operation is easiest to explain when every step returns. The interesting questions begin when a process disappears after changing state but before reporting the result. Did publication fail, or did the acknowledgment

fail? Is the worker gone, or has only its caller stopped waiting? Can cleanup proceed, or is some effect still using the files?

These are not exceptional questions appended to a completed design. They determine ownership and observable behavior. If the application promises that an unsuccessful replacement preserves the working version, that promise must survive the interruption points the application claims to handle.

The running example has three broad phases: acquire a candidate, check it, and publish it. That description is enough to start identifying failures, but it is not enough to decide recovery. We also need to know which state is authoritative, which effects can continue independently, and how a later observer distinguishes a completed action from an unattempted one.

Stop treating cancellation as one event

Cancellation can mean that a caller no longer wants a result, that a worker has been asked to stop, that the worker has actually stopped, or that cleanup has finished. Those events can occur at different times. Collapsing them into one boolean invites a resource to be reused while an old effect can still access it.

For example, a blocking checker may outlive the async task waiting for it. Cancelling the waiter does not necessarily terminate the checker. The ownership capability must remain with work that can still use the candidate, and shutdown must account for that work. A timeout provides an observation about waiting; it does not prove cessation.

Passeur's inspected transport distinguishes parent-process exit from stream closure because a descendant can retain the output stream. A controlled native fixture leaves such a descendant alive and expects close to remain unconfirmed. This establishes a specific process boundary in source and test assertions, not a guarantee for every provider's process tree. Transport observation

(<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/src/agents/codex/transport.ts#L124-L144>), descendant fixture (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/native/codex-transport.test.mjs#L80-L92>).

If stop is uncertain, the system needs an explicit unresolved state and a recovery path. Starting a replacement worker immediately may duplicate effects. Waiting forever without exposing the obligation is also a failure of operability. The appropriate response can be to block conflicting mutation, retain evidence, and request the observation or decision that resolves ownership.

Put failure points beside the intended guarantees

A small fault plan for the illustrative acquisition can be written as four cases:

During acquisition: Interrupt before the candidate is complete. The active reference must still open the old version. Partial candidate bytes must remain either owned for a defined recovery path or eligible for safe cleanup.

During checking: Hold a checker, request cancellation, and attempt conflicting reuse. Reuse must remain excluded while the checker can access the candidate.

At publication: Interrupt after the authoritative state might have changed but before the caller receives its result. Recovery must inspect that state and identify the exact candidate; a missing acknowledgment alone cannot justify another publication.

After publication: Interrupt before cleanup or status notification. The published resource must remain usable, and recovery must not remove it while treating the incomplete notification as a failed candidate.

The third case exposes an important qualification to “failure preserves the old version.” A reported error does not prove the publication itself failed. If publication committed, recovery must describe that outcome truthfully. Keeping the previous version available for deliberate rollback can be useful, but silently rolling back on every communication error introduces another state change that also needs authority and evidence.

Retries need the same care. Retrying the same intended acquisition should retain its identity. Starting a new acquisition that happens to use the same URL may select different content if the source is mutable. A retry policy cannot repair an unspecified identity contract.

Match the claim to the fault actually exercised

Pumas's draft acquisition work includes a Linux subprocess test that kills a child after native destination rename and directory synchronization, before metadata publication. The test then checks exact receipt-based recovery without new requests to its controlled source. This is useful evidence about one SIGKILL boundary. It does not reproduce sudden power loss, establish other filesystems' durability, or qualify Windows and macOS. The book's research inspected the test and repository reports rather than rerunning it. [Crash-boundary regression \(https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-app-manager/src/version_manager/mod.rs#L2982-L3259\)](https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-app-manager/src/version_manager/mod.rs#L2982-L3259).

That limitation is part of the engineering result. Different faults invalidate different assumptions. Throwing an exception preserves the process and much of its memory. Killing a process removes that memory but leaves the operating system alive. Losing power challenges persistence below the process. A network partition creates uncertainty at more than one observer. A claim should name the fault class, environment, and boundary tested.

Failure injection should also be safe for the test environment. Use disposable roots, controlled sources, bounded cleanup, and a way to release held workers when assertions fail. A test that corrupts unrelated data or strands a process has introduced a new lifecycle problem.

Give long-running development its own recovery design

The same reasoning applies to the agent doing the work. Save a useful candidate, current assumptions, failed checks, and the next bounded action where another session can inspect them. A long conversation is not a

durable result by itself.

Armin Ronacher's September 2026 account describes an intentionally oversized, largely unattended CPython experiment that ran for about 35 hours and produced extensive code and commits without an outcome he considered useful. He acknowledges the setup's limitations. The episode is a counterexample to equating continued activity with progress, not evidence that all long-running agent work fails. [Astra for Coding](https://lucumr.pocoo.org/2026/9/7/astra-why/) (<https://lucumr.pocoo.org/2026/9/7/astra-why/>).

A stop condition should concern the ability to continue usefully: repeated attempts without new evidence, a missing decision, exhausted authorized resources, or an acceptance requirement that the environment cannot exercise. Stopping should preserve the result and explain the blocker. It should not pretend an unfinished task is complete merely because a turn ended.

Interrupt one boundary deliberately

Choose one phase transition in your application and state what must remain true on both sides. Add a controlled interruption, attempt the forbidden competing action, then recover through the public owner. Record the exact environment and fault class.

The observable outcome is a state you can explain after interruption, with resource ownership intact and no invented success. If you cannot tell whether publication occurred, preserve that uncertainty and improve the recovery observation before adding automatic retry.

26 Measure before optimizing

An agent notices that the test suite repeatedly constructs a temporary repository. It proposes sharing a fixture. That may be an excellent optimization. It may also allow one test's mutations to affect another, or save little time because compilation dominates the run. The first task is to discover which situation applies.

Optimization begins with a desired consequence. “Fewer subprocesses” is a mechanism. “A developer gets reliable feedback soon enough to stay in the task” is an outcome. Intermediate measures can explain the outcome, but they should not quietly replace it.

The same applies to agent workflows. Fewer tokens can mean less irrelevant context or less necessary information. More parallel workers can reduce waiting or increase integration repair. A faster first patch can still take longer to accept. Measurement should follow the result the team actually values.

Establish the unit of comparison

Consider an illustrative investigation of a slow standards-engine test suite. The hypothesis is that repeated fixture construction, rather than the assertions themselves, dominates execution. No performance result is

asserted here; this is a worked measurement design.

An investigation record could be:

Outcome: Reduce the time to reliable suite feedback without changing what the suite checks or introducing order dependence.

Baseline: Record the exact revision, test selection, command, machine, process settings, and cache conditions. Separate setup time, test-body time, and teardown where feasible.

Hypothesis: Repeated construction of an equivalent immutable corpus accounts for enough of the run to justify reuse.

Candidate change: Reuse only the immutable source fixture. Give each mutating test its own derived state.

Comparison: Run the same workload before and after under comparable conditions. Include repeated runs, changed test order where relevant, and checks that a mutation cannot leak between tests.

Decision: Retain the optimization only if the observed improvement matters and the isolation contract remains credible.

The record prevents an easy mistake: changing both the fixture and the test workload, then attributing the whole difference to reuse. It also keeps correctness in view. A fast suite that occasionally inherits a previous test's state is worse feedback, even if its median runtime improves.

Measure enough to answer the decision. A small isolated optimization may need simple timings and a clear distribution of observed variation. A noisy shared runner may require more repetitions or a more controlled comparison. Sophisticated statistics cannot rescue a workload that does not represent the actual use.

Interpret intermediate measures carefully

Suppose instrumentation confirms that the candidate launches fewer processes. That supports a mechanism claim. It does not yet establish shorter end-to-end execution, lower memory use, or faster development. Those are separate observations.

Likewise, a smaller agent-tool catalog can reduce payload bytes while increasing discovery calls. A narrower task can reduce context while shifting specification work to the coordinator. These tradeoffs may be worthwhile. They need to be counted where they occur.

For the running resource application, total acquisition latency can be decomposed into locating the source, transferring bytes, checking, publishing, and making the result visible to the consumer. If checking dominates a particular workload, improving connection setup may not change the experience materially. If publication blocks all readers, average transfer speed may be the wrong priority altogether.

Use representative sizes and failure cases. A tiny cached fixture may hide the path that matters. An enormous synthetic input may exaggerate a bottleneck absent from normal use. Report what the workload represents and what it excludes.

Measure development without confusing its clocks

Active human time, elapsed delivery time, and machine cost are different quantities. A background agent can require little attention while taking hours. Several simultaneous tasks can overlap elapsed time while competing for the same review capacity. Summing their wall-clock durations does not yield the person's working time.

METR's early-2025 randomized study found a slowdown for experienced contributors using the evaluated tools on familiar mature repositories. That is a result in a particular population and period, not a permanent verdict on coding assistance. Its 2026 continuation warns that selection effects and measuring concurrent-agent work make newer estimates unreliable. The useful response is careful local measurement, not choosing whichever headline confirms an existing preference. Early-2025 trial (<https://arxiv.org/abs/2507.09089v2>), 2026 update (<https://metr.org/blog/2026-02-24-uplift-update/>).

A local comparison should include specification, supervision, unsuccessful attempts, review, repair, and integration. Keep quality criteria stable. If one workflow produces more ambitious results, record that change in scope rather than calling the longer duration a simple slowdown or the extra output a simple speedup.

Subjective experience matters too. A developer may find a workflow less tiring or more enjoyable without finishing faster. That can be valuable, but it is a different claim. Record it in its own terms rather than forcing it into a productivity number.

Look for the receiving bottleneck

Daniel Stenberg's April 2026 curl account describes a shift from low-quality report overload toward useful findings arriving faster than the project could comfortably remediate and distribute fixes. The changing report mix and operating conditions prevent a simple causal attribution to AI. The maintenance lesson is nevertheless concrete: better discovery can move the limiting work downstream. High-Quality Chaos (<https://daniel.haxx.se/blog/2026/04/22/high-quality-chaos/>).

For agent-assisted development, the receiver may be the reviewer, integrator, release engineer, or user who must evaluate changed behavior. Improving generation without considering that receiver can enlarge a queue of plausible unfinished work. A useful optimization may therefore reduce batch size or improve evidence rather than increase patch output.

Once the relevant checks pass and the decision is supported, stop measuring for this change. More instrumentation has its own maintenance cost. Keep enough evidence to understand the result and revisit it when the workload or environment materially changes.

Test one suspected bottleneck

Choose a delay you routinely notice. State its consequence, measure the baseline, and identify one change that could affect it. Hold the workload and acceptance criteria steady while comparing the candidate. Include an unsuccessful or noisy observation rather than retaining only the best run.

The exercise succeeds when you can say what improved, by how much under the observed conditions, and what did not follow from the measurement. Discovering that the suspected bottleneck is unimportant is a useful result: it prevents a confident optimization of the wrong thing.

27 Finish with evidence

A change can be implemented, tested locally, reviewed, integrated, and still not accepted for its intended use. These are distinct states because they answer distinct questions. Treating them all as “done” forces the next person to reconstruct what actually happened.

Completion should be a bounded claim about a particular result. It identifies what was requested, what revision carries the result, which observations support it, and which obligations remain. This is neither a victory speech nor an archive of every command. It is the account someone needs to decide what they can safely rely on next.

The resource example gives us a concrete target. The change is complete for its agreed scope when the selected replacement can be used after publication, failed preparation preserves the working version, and the required evidence supports that behavior in the named environment. A future desire for another source provider does not automatically keep this change open. A required publication check that never ran does.

Bind the claim to the candidate

A passing result belongs to the revision and environment exercised. Subsequent source changes may invalidate it. A documentation-only successor can sometimes preserve the relevant implementation evidence, but that relationship should be justified rather than assumed. A report copied from an earlier branch is historical evidence, not a fresh run.

The inspected Coding-Standards history provides an uncomfortable and useful example. At main revision `dcc56f26`, the exact-head hosted Engine job reported 613 tests with one failure. The failing proposal-analysis test received `AUTHORING.INVALID_SEMANTIC_REVISION`; the later complete verification command was not reached because the shell stopped on failure. Passing earlier groups did not make the whole gate pass. Observed hosted job (<https://github.com/MrScripty/Coding-Standards/actions/runs/36947175564/job/110651832967>), workflow sequence (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/.github/workflows/purpose-separated-engine.yml#L1-L78>).

Source inspection showed a fixture hardcoding accepted revision one for a policy whose current declaration was revision two. That strongly supports a stale-fixtue diagnosis. It does not show that a repair was applied or that

the intended projection-count assertion passed afterward. The book's research made no repair and performed no rerun. Fixture (https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_analysis.py#L2191-L2211), current declaration (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-units/planning.toml#L45-L49>).

The distinction is practical. “The failure appears to be an obsolete fixture” tells a maintainer what to investigate. “Verification passed” would tell them something the evidence does not establish. A persuasive diagnosis cannot stand in for the missing observation.

Write an account someone can use

For the illustrative application, a release account awaiting a required hosted check might read:

Result: Candidate R1 implements replacement through the supported local consumer. Preparation failures preserve the old active version. No additional source provider or runtime installation is included.

Local evidence: The recorded checks cover opening the old version before publication, rejecting an invalid candidate, opening the requested replacement after publication, and cancellation while checking is held. The record names the commands, environment, and candidate R1.

Review: The lifecycle reviewer inspected R0 and found premature completion. The repair produced R1; the reviewer rechecked the finding on R1, and the local checks above exercised R1. Any later source change requires assessment of which evidence remains applicable.

Integration: R1 is on the agreed integration branch. The required hosted verification of R1 has failed, so release acceptance remains blocked.

Next obligation: Investigate the failing hosted check, repair within the agreed scope, and obtain the missing result. The release owner makes the acceptance decision after that evidence is available.

This is a specimen of wording, not a report of tests performed on a real application. It demonstrates that substantial useful work can be complete while a specific release obligation remains open. The account gives the next worker a concrete continuation rather than either an exaggerated success or a vague “not done.”

When all required checks pass, the account can become shorter. State the delivered behavior, revision, verification, material limits, and destination. Do not make the user navigate a long ledger to find the answer to whether they can use the result.

Keep unknowns distinct from defects

An untested platform is not automatically a known failure. A failed required check is not merely an untested platform. A known defect accepted for a bounded release is different from either, and should name who accepted the consequence and under what conditions.

This distinction prevents both overclaiming and needless paralysis. If the agreed release supports one local platform, absence of qualification elsewhere can be an explicit limit. If cross-platform behavior is part of the promise, the same absence is a blocker. Acceptance follows the intended scope rather than a universal checklist.

Do not hide unresolved incidents behind nearby successful repairs. A source defect may have been corrected without reproducing the original report. Preserve the distinction until the evidence connects them. Likewise, an operator's decision to retire an uncertain task does not rewrite its historical outcome into confirmed success.

Finish the operational edges

Delivery includes the artifacts someone needs to use or continue the work: the actual patch or release, the relevant documentation, and retrievable evidence. Temporary branches and environments should receive a justified disposition. Cleanup must preserve uniquely held work and any evidence required for recovery or maintenance.

Remaining tasks should have a purpose and an owner. “Improve everything later” is not a useful obligation. “Qualify the same publication behavior on the second supported platform before enabling it there” is. If a required external process is still running, completion reporting should reflect that and the responsible person should continue following it to a result.

There is also a stopping rule. Once the requested outcome is accepted and its operational obligations are resolved, unrelated architectural ambitions belong to another decision. A rigorous process should make completion easier to recognize, not place it beyond reach.

Audit the word done

Take a recently completed change and try to reconstruct its claim from the final report alone. Can you identify the exact candidate, the behavior delivered, the checks that ran, the checks that failed or did not run, the review scope, and the acceptance decision?

Rewrite the account so another person can distinguish evidence from diagnosis and current status from historical success. The observable outcome is a reader who knows what they may rely on and what, if anything, still needs action. That is a useful end to a change and a trustworthy beginning for the next one.

Part 6 Build an ecosystem without losing the products

The final part examines the broader consequences of the approach: shared foundations, usable releases, representations beyond source text, and the process's own need to remain open to criticism and revision.

28 Grow shared foundations from demonstrated needs

An ecosystem begins to make sense when several products need the same responsibility handled well. It becomes difficult to direct when every possible future use is treated as a present requirement.

Several systems may encounter recurring needs: model management, execution, agent composition, coordination, knowledge representation, and provenance. Those connections justify investigating shared infrastructure. They do not establish that the systems should become one monolith or that proposed integrations already work.

A shared foundation should have a reason that can be expressed through actual consumers.

Draw responsibilities before dependencies

A repository diagram often begins with arrows. An arrow is ambiguous. It may mean a runtime dependency, a shared concept, a planned integration, a source of research, or merely an application that might benefit later.

First describe the responsibility: managing resources, executing work, composing agents, preserving records, supplying guidance, or coordinating participants. An application may need several of these responsibilities without sharing the same purpose or information model as another application.

Then label the connection. Is there an inspected call path? A shared data contract? A proposal to adapt an idea? A dependency needed for an end-to-end release? The label prevents architectural intention from becoming a false implementation claim.

A responsibility map describes a proposed architecture. Concrete implementation claims need evidence from observed call paths, contracts, and behavior; a connection on a diagram is not that evidence.

Extract the common rule rather than the first application's schema

Suppose two applications need durable records. They may share transaction handling, identity, ordering, and retention mechanisms. They may not share the meaning of a workflow result, a research claim, or a model-loading attempt.

If the common layer encodes the first application's tables as its public domain, the second application must imitate that application's assumptions. The extraction has preserved code while missing the common responsibility.

A better investigation asks which invariants genuinely match. Can both applications use the same record identity and append semantics? Do they need the same consistency boundary? Which fields belong to the application and should remain opaque to the storage mechanism?

The result might be a small shared component rather than a broad platform. That is a legitimate foundation if it removes a demonstrated duplication without centralizing unrelated meaning.

Give a shared layer its own product contract

Infrastructure is used by people and agents. It needs discoverable capabilities, understandable failures, versioning, and a definition of supported behavior. Treating it as an internal convenience can leave callers dependent on incidental structure.

A shared layer also creates an organizational obligation. Who accepts a change that benefits one consumer but burdens another? How are incompatible requirements represented? Who owns migration? These questions matter even when one person currently maintains all the repositories, because agents can act as separate consumers with different local contexts.

An abstraction should make disagreement explicit. If one consumer needs durable retention and another needs ephemeral cleanup, do not bury the difference in a boolean whose meaning changes by caller. Establish the policy boundary and show what each choice costs.

Keep independent projects independent where it helps

A project can contribute ideas without becoming a runtime dependency. A ledger or workflow experiment may inform another system while remaining useful on its own. Knowledge-graph work may reveal needs relevant to research or memory tools without imposing one application schema everywhere.

This independence can improve learning: a concept encounters a different problem and reveals which assumptions were local. It can also produce duplicated effort. The decision should be based on the value of those different contexts, not a reflex to consolidate everything.

For an exercise, draw a map of three projects using five connection labels: implemented dependency, shared contract, planned integration, research influence, and product prerequisite. Select one shared foundation and identify two actual consumer scenarios. If their invariants do not match, narrow the proposed common layer. The map is useful when it prevents an unnecessary dependency or clarifies a real one.

29 Balance foundational work with usable releases

A foundation can be necessary and still become a reason that no product reaches a user. Each layer exposes another problem worth solving. Model handling leads to runtime management. Runtime management leads to acquisition and resource ownership. Execution leads to scheduling. Scheduling leads to distributed coordination. The work can remain technically meaningful while the original application stays out of reach.

Define the smallest valuable end-to-end result

A horizontal milestone completes a layer: the registry, the scheduler, the ledger. A vertical milestone lets a person do something useful through the necessary layers.

For our running example, a vertical result could be selecting one supported resource, acquiring it, verifying readiness, and using it from a real consumer. It may support only a narrow model and runtime. That limitation is acceptable if it is explicit and the result establishes the path the product needs.

The narrow slice should not fake the difficult boundary. If a demo skips readiness checks that the real consumer depends on, it has demonstrated presentation rather than the intended outcome. The aim is a small honest path, not a broad illusion.

Write the result in terms a user can verify. “The acquisition subsystem is implemented” is weaker than “this consumer can use this selected resource after a normal acquisition and after the defined retry path.” The latter also reveals what remains outside the milestone.

Distinguish blockers from ambitions

A blocker prevents the selected result. An ambition improves or broadens a future result. They can both be important, but they should not occupy the same status.

A missing runtime capability may block the chosen model. Support for every future provider probably does not. A resource-lifecycle defect may make the slice unsafe to release. A distributed scheduler may be unnecessary for a local-only first use.

The distinction should be revisited when evidence changes. A supposedly optional feature may turn out to be required for the target user. A presumed dependency may disappear when an existing interface is understood. Keep the outcome visible so these revisions are possible.

Price temporary choices honestly

A narrow release often includes temporary decisions. They are not automatically technical debt, and they are not free. Record the condition under which they become costly.

Supporting one runtime may be a sound product boundary. Hardcoding that runtime's private layout into every consumer may make the next integration unnecessarily expensive. A local implementation can be appropriate while still preserving a clear boundary where later variation is likely.

The right question is not whether the design is future-proof. No design is. Ask whether it preserves the important current invariants and leaves a credible path for demonstrated next needs.

Match qualification to the intended use

Different projects may justify different balances. A personal tool can be useful with a narrow interface. A shared library needs clearer compatibility and failure contracts. A public service may require operational capabilities that a local prototype can defer. The development path should reflect those consequences.

A receiving institution can also set conditions that the producer's workflow cannot decide for it. Drew DeVault's SourceHut account reports resource-use concerns while explicitly grounding its contribution restrictions primarily in ethical and political choices. It also acknowledges useful private AI review. The account should not be converted into a technical claim that the tools never work, or a controlled estimate that generated projects always cost more. It illustrates a practical boundary: a contributor's preferred method does not determine another project's admission rules. [SourceHut policy account \(https://sourcehut.org/blog/2026-08-27-tos-changes-and-llms/\)](https://sourcehut.org/blog/2026-08-27-tos-changes-and-llms/).

Review the next user action

For the work in hand, write the next useful action a person should be able to perform. Identify the one or two missing capabilities that actually prevent it. Then ask whether current foundational work removes those blockers.

If it does, the connection should be easy to explain. If it does not, decide deliberately whether the research value justifies the delay. This is not a demand to abandon long-term work. It is a way to preserve agency over how much of the present is being spent on the future.

The practical risk is not ambition itself. It is losing the line from today's work to a result someone can eventually use and evaluate.

30 Make software understandable beyond source code

Source code describes executable structure, but it rarely contains the whole reason that structure exists. A developer can follow a call graph and still not know which requirement created a boundary, which alternative was rejected, or what evidence supports a claim of readiness.

Connecting code representations with intent, decisions, applied guidance, and other information can help readers understand why software has its present structure. A visual representation should be evaluated through the questions it helps answer, rather than the fact that the result is a graph.

Begin with a question that code alone answers poorly

Consider: “If we change when a candidate becomes ready, which acceptance claims and consumers need review?” A call graph may identify callers. It may not identify a decision record explaining why the current boundary was chosen, or an acceptance observation made through a particular application.

A useful representation could link the readiness decision to its contract, implementation owner, tests, consumers, and evidence. Each edge needs a meaning. “Related to” is too vague if readers use the graph to decide what must be checked.

Another question is historical: “Why did we keep the old version available during preparation?” A source comment may explain it, or the reason may live in a discussion, plan, or failure report. The representation should help retrieve that reason without pretending every historical influence was recorded.

Keep structural and semantic relationships distinct

A code reference is an observed relationship in the implementation. A decision depends on a requirement is a semantic claim. A test supports an acceptance claim is an interpretation of what the test establishes. These edges have different evidence and maintenance needs.

Automatically extracting references can populate one part of the graph. It cannot infer all intent relationships with equal authority. A generated edge may be a useful suggestion, but should remain attributable and revisable.

RepoGraph reports benefits from code-relationship context for evaluated issue-resolution baselines. That is evidence about code retrieval in its setting. It does not validate a broader graph of intent, authority, and decisions. The broader design needs its own evaluation. [RepoGraph](https://arxiv.org/html/2410.14684v2) (<https://arxiv.org/html/2410.14684v2>).

Design for reading rather than displaying everything

A graph containing every node and edge can be less understandable than a short explanation. Readers need entry points, meaningful grouping, filtering, and progressive detail.

For the readiness question, begin with the decision and immediate dependencies. Let the reader expand a consumer or inspect the supporting evidence. Do not force them to navigate the entire repository to answer a bounded question.

Visual position and color should not carry unsupported meaning. A large node may look important even if its size merely reflects code references. A central position may suggest authority when it is only a layout artifact.

Explain the encoding and provide a textual path through the same information.

The representation also needs to show missing or stale evidence. An edge tied to an older revision may still be historically useful but should not silently certify the current implementation.

Make the maintenance cost visible

Every semantic relationship must come from somewhere. A person may record it, an agent may propose it, or a tool may derive it under explicit rules. Changes can invalidate it.

If maintaining the graph takes more effort than the questions it answers justify, the system needs a narrower scope. Begin with a small set of valuable relationships rather than an ontology of all development knowledge.

The graph may be a view over existing records rather than another source of truth. That can reduce duplication, provided the underlying records have stable meanings and identities. It also makes it possible to offer alternative views for different readers.

Test comprehension directly

Give a reader a concrete question and observe whether the representation helps them answer it correctly. Measure time if useful, but also inspect what they misunderstood. A fast wrong answer is not improved understanding.

Compare the graph with a simpler baseline: a linked decision record, a searchable index, or a short narrative. The graph should earn its place through the relationship task, not through novelty.

For an exercise, build a tiny map connecting one outcome, one design decision, one implementation boundary, and two pieces of evidence. Mark the provenance of every edge. Ask another developer what would need reconsideration if the outcome changed. Where the map suggests certainty it cannot support, revise the representation.

The aim is software that can be understood and challenged beyond its source text. A visual model is one possible means to that end.

31 Improve the development process itself

A development method is also a system. It has users, interfaces, state, costs, and failure modes. It can accumulate compatibility layers, stale assumptions, and rituals that no longer serve their original purpose.

If the method advocates proportionality, it must apply that principle to itself.

A process can be too rigorous in the wrong way

The inspected Planning Proportionality Recovery records describe a correction to an overgeneralized planning architecture. Generic transition and digest machinery had been applied too broadly. The repair restricted stale-state controls to work where independently prepared proposals could become outdated before integration. Multiple participants alone no longer implied that risk.

A recorded read-only pilot prepared a temporary 132-line Pantograph plan and checked its structure and links while avoiding changes to a checkout with unrelated dirty documents. It did not implement the proposed product change. The case shows an accepted policy correction and bounded pilot, not a measured productivity improvement. Recovery plan ([https://github.com/MrScripty/Coding-](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/plan.md)

[Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/plan.md](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/plan.md)); pilot report (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/reports/downstream-ordinary-plan-pilot.md>).

This is more instructive than a rule to always use a plan. The method recognized that a specialized safeguard was imposing obligations on tasks without the relevant hazard. It preserved scope, ownership, and acceptance while reducing unnecessary machinery.

Identify the cost before adding a rule

A new rule often follows a real failure. An agent used stale context, omitted a test, or changed a shared file without coordination. The natural response is to require another check everywhere.

Before doing that, identify the conditions that made the failure possible. Does the risk apply to every task, or only to a particular concurrency pattern? Can the owner enforce it once instead of every caller documenting it? Is the new requirement likely to prevent the failure, or merely record that someone considered it?

A safeguard can be valuable and still need a narrower trigger. It can also create new failure modes: blocked work, confusing authority, duplicated records, or false declarations of uncertainty. The Passeur proof-boundary case discussed in the coordination chapters illustrates why demanding more evidence is not automatically the correct contract.

Distinguish compliance from effectiveness

A workflow can be followed perfectly and produce a poor result. A developer can omit a prescribed artifact and still make a sound decision. Neither observation alone settles whether the workflow is useful.

Compliance tells you whether the process occurred. Effectiveness asks whether it improved an outcome relative to a relevant alternative. The latter requires a comparison and careful attention to task differences.

Choose a bounded improvement to investigate. Perhaps targeted guidance could replace a broad instruction dump. Perhaps a small serial task can omit a coordination record. Perhaps a runtime demonstration catches errors missed by the current tests. Define the outcome and the cost you will observe before changing the process.

Boehm's risk-driven account treats both insufficient exploration and excessive exploration as risks, and scales effort and artifact detail to what a decision needs. It provides a useful conceptual basis for this work, without a universal formula for the optimal stopping point. Spiral Development Experience Principles and Refinements (https://resources.sei.cmu.edu/asset_files/specialreport/2000_003_001_13655.pdf).

Make criticism actionable

Puma wants criticism to be one purpose of this book. The following questions are proposed prompts for that criticism, not invented statements about what Puma must believe:

- Where does the approach mistake a well-kept record for actual understanding?
- Which improvements may depend more on domain familiarity or model capability than on the method?
- Where would a direct implementation followed by user feedback be cheaper and equally dependable?
- Which important work is missing because it is difficult to represent or measure?
- Do the cases include enough rejected alternatives and adverse outcomes to judge the decisions?
- What can another developer reproduce, and what depends on individual judgment?

A useful response identifies the claim being challenged and a condition or example that changes it. “This is too much process” becomes more actionable when it names the task class, the unnecessary obligation, and the simpler alternative. “This needs more rigor” becomes more useful when it identifies the failure the current evidence cannot exclude.

Keep the method open without making every task a referendum

Not every implementation should reopen the entire engineering philosophy. Stable guidance reduces repeated decision cost. The method needs a way to distinguish a local exception from evidence that a rule is wrong or mis-scoped.

Record recurring friction. Look for patterns across tasks rather than treating one inconvenience as a universal defect. At the same time, do not require repeated harm before correcting a clear contradiction or dangerous boundary.

The approach developed in this book is a set of relationships to preserve: purpose to behavior, concepts to invariants, authority to action, and claims to evidence. Its artifacts are means of preserving those relationships. Their form can change when a simpler or more effective method becomes available.

That is the position readers are invited to examine. The work is not complete because the process has a name, the graph is connected, or the tests are green. It becomes useful when another person can understand what was intended, inspect what happened, judge what the evidence supports, and decide what to do next.

Working vocabulary

These short definitions summarize how the book uses its terms. The linked chapters provide the examples and qualifications. They are aids to discussion, not a universal terminology standard.

Acceptance. A decision that the stated outcome has been supported by the agreed evidence within a defined scope. It is distinct from finishing code, passing a test, or merging a branch. See [chapter 27](#).

Agent interface. The operations, discovery paths, descriptions, feedback, and observable environment through which an agent uses a system. It includes errors and recovery, not only schemas. See [chapter 12](#).

Authority. The right to make a decision or perform an action within a specified scope. Possessing information or a technical capability does not automatically grant it. See [chapter 15](#).

Boundary. A set of promises separating responsibilities. A useful boundary exposes the distinctions callers need and contains incidental implementation choices. See [chapter 9](#).

Candidate. In the running example, locally present material that has not yet met the conditions for publication as ready. See [chapter 6](#).

Capability. A behavior a component can provide under stated conditions. A version label may be evidence about capability but is not identical to it. See [chapter 9](#).

Characterization test. An experiment or test used to establish current behavior, often before changing unfamiliar code. It may reveal what exists without deciding whether that behavior is desirable. See [chapter 3](#).

Contract. The obligations and permitted expectations between a component and its callers, including success, failure, identity, lifetime, and recovery where relevant. See [chapter 9](#).

Decision record. A compact account of a consequential choice, its context, alternatives, status, and consequences. Its purpose is to preserve useful rationale rather than reproduce the entire discussion. See [chapter 5](#).

Deterministic control. Enforcement whose result follows defined rules for the relevant input and state. It guarantees only the properties those rules actually check. See [chapter 15](#).

Discovery. The conversational and investigative work of establishing what should be built and why, including uncertainty that may require a prototype or later revision. See [chapter 2](#).

Domain model. A representation of concepts, identity, relationships, and rules needed to explain the problem's behavior. It need not map one-to-one to classes or services. See [chapter 6](#).

Evidence boundary. The limit on what an observation supports: a revision, environment, path, population, or fault class, for example. See [chapter 27](#).

Higher-order conflict. An incompatibility that can survive a clean text merge, such as disjoint changes violating the same semantic assumption or causing an integration failure. See [chapter 20](#).

Hypothesis. A proposition being considered or tested, rather than an established fact or binding requirement. See [chapter 2](#).

Identity. What allows a system to refer to the same thing across relevant changes. Resource identity, attempt identity, and artifact identity can have different lifetimes. See [chapter 6](#).

Independent review. A procedure that gives a reviewer a separate opportunity to challenge a change. Separate roles do not guarantee statistically independent mistakes. See [chapter 24](#).

Intent. The purpose and desired consequences motivating a request. It may require interpretation and can develop through investigation and use. See [chapter 1](#).

Intent-oriented operation. An interface operation that requests a domain outcome while its owner handles incidental choreography, preserving consequential caller choices. See [chapter 13](#).

Invariant. A condition that must remain true across the relevant states and transitions of a system. Its scope and owner must be clear. See [chapter 7](#).

Lifecycle. The meaningful states and transitions of an activity or resource, including the obligations that remain after a caller stops waiting. See [chapter 7](#).

Material uncertainty. An unresolved question that can change intended behavior, design, scope, or acceptance. It is not measured by patch size alone. See [chapter 2](#).

Operational responsibility. The obligation to complete, clean up, or reconcile work. It can outlive the caller and differs from permission to initiate the work. See [chapter 7](#).

Outcome. The behavior or capability that should become possible for the intended person or consumer. An implementation step is not necessarily an outcome. See [chapter 1](#).

Proportionality. Matching investigation, abstraction, coordination, and evidence effort to the decisions and consequences at stake. It supports both adding and removing process. See [chapter 31](#).

Provenance. Attributable relationships connecting artifacts, activities, actors, and sources. It helps reconstruction but does not by itself prove correctness, completeness, or causal explanation. See [chapter 16](#).

Publication. In the running example, the transition that makes a candidate available through the consumer's agreed ready reference. It is distinct from completing transfer. See [chapter 13](#).

Readiness. The bounded promise that the resource satisfies the conditions required for the intended next use. Its meaning must be defined by the contract. See [chapter 7](#).

Refactoring. A change intended to improve internal structure while preserving required behavior. In practice, mixed changes require care when attributing benefits or defects. See [chapter 10](#).

Revision guard. A check that a proposal applies to the state or version against which it was prepared. Passing it does not prove the proposal serves the intended outcome. See [chapter 15](#).

Running example. The explicitly illustrative resource-acquisition system developed through this book. It is not a reconstruction of any one implementation. See [chapter 1](#).

Shared foundation. A layer whose responsibility serves demonstrated common needs across consumers, while preserving application-specific meaning where appropriate. See [chapter 28](#).

Test oracle. The basis used to judge whether an observed result is correct. An oracle can encode the same mistaken assumption as the implementation. See [chapter 23](#).

Uncertain outcome. A situation in which the observer cannot yet establish whether an operation's side effect occurred. It requires reconciliation or defined retry semantics, not an invented success or failure. See [chapter 25](#).

Vertical slice. An end-to-end increment that lets a person or consumer achieve a bounded useful result through the necessary layers. See [chapter 29](#).

Sources and further reading

The references below identify the sources used in the chapters. Links beside particular claims remain the best starting point. Repository links preserve the inspected revisions where available. Living pages were accessed on 2 October 2026. Practitioner accounts, research studies, specifications, and project records serve different purposes; their limits are discussed where they are used. No practitioner is claimed to endorse this entire approach. Interview sources refer to inspected published text or transcript sections; no complete source videos or commercial books are represented as having been consumed for this research.

Reference 1

Pamela Zave and Michael Jackson, “Four Dark Corners of Requirements Engineering,” ACM TOSEM 6(1) (1997), pp.1–30, DOI 10.1145/237432.237434, university-hosted paper

Open source (<https://www.cse.msu.edu/~chengb/RE-491/Papers/dark-corners-re-zave-jackson.pdf>). Used in chapter 1.

Reference 2

Puma, Coding-Standards. Discovery and Design at the inspected revision.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning/discovery.md>). Used in chapter 2.

Reference 3

Bashar Nuseibeh and Steve Easterbrook, “Requirements Engineering: A Roadmap” (2000), author-hosted PDF

Open source (<https://www.cs.utoronto.ca/~sme/papers/2000/ICSE2000.pdf>). Used in chapter 2.

Reference 4

Jonathan Sillito, Gail C. Murphy, Kris De Volder, “Questions Programmers Ask During Software Evolution Tasks,” FSE-14 (2006), pp.23–34, DOI 10.1145/1181775.1181779, author-hosted PDF

Open source (<https://www.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf>). Used in chapter 3.

Reference 5

Oxide and Friends. Engineering Rigor in the LLM Age. Published transcript, 15 January 2026; cited passage featuring Bryan Cantrill.

Open source (<https://oxide-and-friends.transistor.fm/episodes/engineering-rigor-in-the-llm-age/transcript>). Used in chapter 3.

Reference 6

Stephanie Houde and Charles Hill, “What do Prototypes Prototype?” in Handbook of Human-Computer Interaction, 2nd ed. (1997), pp.367–381; author-format university-hosted PDF

Open source (https://creativetech.mat.ucsb.edu/readings/Prototypes_prototype.pdf). Used in chapter 4.

Reference 7

Mitchell Hashimoto, 11 October 2025. Vibing a Non-Trivial Ghostty Feature

Open source (<https://mitchellh.com/writing/non-trivial-vibing>). Used in chapter 4.

Reference 8

Rick Kazman, Mark Klein, Paul Clements, “ATAM: Method for Architecture Evaluation,” CMU/SEI-2000-TR-004 (2000), official PDF

Open source (https://www.sei.cmu.edu/documents/629/2000_005_001_13706.pdf). Used in chapter 5.

Reference 9

Rahul Mohanani, Burak Turhan, Paul Ralph, “Requirements Framing Affects Design Creativity,” IEEE TSE 47(5) (2021), pp.936–947; author manuscript 2019; DOI 10.1109/TSE.2019.2909033, arXiv record and PDF

Open source (<https://arxiv.org/abs/1902.11278>). Used in chapter 5.

Reference 10

Michael Nygard, “Documenting Architecture Decisions,” 15 November 2011, original post

Open source (<https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions>). Used in chapter 5.

Reference 11

Eric Evans, “Domain-Driven Design Reference: Definitions and Pattern Summaries,” March 2015, author's PDF

Open source (https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf). Used in chapter 6.

Reference 12

Puma, Pumas-Library. Repair.

Open source (<https://github.com/MrScripty/Pumas-Library/commit/c5deb493172d3fbddab55141d129f2ac7346bb3b>). Used in chapter 7.

Reference 13

Puma, Pumas-Library. regression at the inspected head.

Open source (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-core/src/acquisition/service.rs#L3128-L3225>). Used in chapter 7.

Reference 14

Puma, Coding-Standards. Simplicity as reduced entanglement; domain ownership and criteria for abstraction/deletion.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/topics/code-design.md#L15-L99>). Used in chapter 8.

Reference 15

David L. Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules,” CACM 15(12) (1972), pp.1053–1058, DOI 10.1145/361598.361623, university-hosted HTML

Open source (<https://www.cs.lafayette.edu/~gexia/cs301/resources/parnas.html>). Used in chapter 8.

Reference 16

Andreas Kling, 23 February 2026. Ladybird adopts Rust, with help from AI

Open source (<https://ladybird.org/posts/adopting-rust/>). Used in chapter 9.

Reference 17

Andreas Kling, 5 June 2026. Changing How We Develop Ladybird

Open source (<https://ladybird.org/posts/changing-how-we-develop-ladybird/>). Used in chapter 9.

Reference 18

Miryung Kim, Thomas Zimmermann, Nachiappan Nagappan, “An Empirical Study of Refactoring Challenges and Benefits at Microsoft,” IEEE TSE 40(7) (2014), pp.633–649, DOI 10.1109/TSE.2014.2318734, author manuscript

Open source (<https://thomas-zimmermann.com/publications/files/kim-tse-2014.pdf>). Used in chapter 10.

Reference 19

Massimiliano Di Penta, Gabriele Bavota, Fiorella Zampetti, “On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication” (2020), author HTML manuscript

Open source (<https://arxiv.org/html/2009.11685v1>). Used in chapter 10.

Reference 20

Martin Fowler, “Yagni,” 26 May 2015, author's essay

Open source (<https://martinfowler.com/bliki/Yagni.html>). Used in chapter 10.

Reference 21

Martin Fowler, “Design Stamina Hypothesis,” 20 June 2007, author's essay

Open source (<https://martinfowler.com/bliki/DesignStaminaHypothesis.html>). Used in chapter 10.

Reference 22

Mitchell Hashimoto, 1 June 2023. My Approach to Building Large Technical Projects

Open source (<https://mitchellh.com/writing/building-large-technical-projects>). Used in chapter 11.

Reference 23

John Yang et al. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. arXiv:2405.15793v3, 11 November 2024. Paper

Open source (<https://arxiv.org/html/2405.15793v3>). Used in chapter 12.

Reference 24

Armin Ronacher, 12 June 2025. Agentic Coding Recommendations

Open source (<https://lucumr.pocoo.org/2025/6/12/agentic-coding/>). Used in chapter 12.

Reference 25

Puma, Coding-Standards. Registration does not rewrite prose or certify coverage; same-candidate ordering and tombstones.

Open source ([https://github.com/MrScripty/Coding-](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L140-L211)

[Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L140-L211](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L140-L211)). Used in chapter 13.

Reference 26

Puma, Coding-Standards. Cold/full versus incremental replay, failed suffix isolation, caller mutation resistance.

Open source ([https://github.com/MrScripty/Coding-](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L270-L305)

[Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L270-L305](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_policy_registration.py#L270-L305)). Used in chapter 13.

Reference 27

Puma, Coding-Standards. Registration plan.

Open source ([https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/plans/existing-policy-](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/plans/existing-policy-registration/plan.md)
[registration/plan.md](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/plans/existing-policy-registration/plan.md)). Used in chapter 13.

Reference 28

Puma, Coding-Standards. Immutable projection table; staged registration before final compile.

Open source ([https://github.com/MrScripty/Coding-](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/standards_engine/logical_authoring.py#L1026-L1085)

[Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/standards_engine/logical_authoring.py#L1026-L1085](https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/standards_engine/logical_authoring.py#L1026-L1085)). Used in chapter 13.

Reference 29

Puma, Coding-Standards. Registered policy scopes.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-units/workflow.planning.discovery.toml>). Used in chapter 14.

Reference 30

Puma, Coding-Standards. Scoped discovery consumer declarations; presence is not an exhaustive actual-consumer audit.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-impact/workflow.planning.discovery.toml#L1-L138>). Used in chapter 14.

Reference 31

Thibaud Gloaguen et al. *Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?* arXiv:2602.11988v3, 29 September 2026. Paper

Open source (<https://arxiv.org/html/2602.11988v3>). Used in chapter 14.

Reference 32

Edoardo Debenedetti et al. *Defeating Prompt Injections by Design*. arXiv:2503.18813v2, 24 June 2025. Paper; official code

Open source (<https://arxiv.org/html/2503.18813v2>). Used in chapter 15.

Reference 33

Luc Moreau and Paolo Missier, editors. *PROV-DM: The PROV Data Model*. W3C Recommendation, 30 April 2013. Versioned recommendation

Open source (<https://www.w3.org/TR/2013/REC-prov-dm-20130430/>). Used in chapter 16.

Reference 34

Yanda Chen et al. *Reasoning Models Don't Always Say What They Think*. arXiv:2505.05410v1, 8 May 2025.
Paper record; author-hosted paper

Open source (<https://arxiv.org/abs/2505.05410v1>). Used in chapter 16.

Reference 35

Puma, Coding-Standards. Planning, pinned revision.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning.md#L31-L49>). Used in chapter 17.

Reference 36

Puma, Coding-Standards. Purposeful development/integration boundaries; no branch/merge-per-slice rule.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/workflows/planning.md#L161-L184>). Used in chapter 18.

Reference 37

Mitchell Hashimoto, 5 February 2026. My AI Adoption Journey

Open source (<https://mitchellh.com/writing/my-ai-adoption-journey>). Used in chapter 18.

Reference 38

Puma, Passeur. Task admission.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/src/core/coordinator.ts#L413-L450>). Used in chapter 19.

Reference 39

Puma, Passeur. lifecycle tests.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/core/durable-lifecycle.test.mjs#L52-L94>). Used in chapter 19.

Reference 40

Puma, Passeur. Repair record.

Open source (<https://github.com/MrScripty/Passeur/commit/cae37226d2d22644c85b4974e35a6c3555b0bf6>). Used in chapter 20.

Reference 41

Puma, Passeur. competing reclaimers.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/core/lease-recovery.test.mjs#L119-L178>). Used in chapter 20.

Reference 42

Puma, Passeur. paused owner.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/native/shared-service.test.mjs#L73-L83>). Used in chapter 20.

Reference 43

Yubin Kim et al. *Towards a Science of Scaling Agent Systems*. arXiv:2512.08296v3, 8 April 2026. Paper

Open source (<https://arxiv.org/html/2512.08296v3>). Used in chapter 20.

Reference 44

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, Lingming Zhang. *Agentless: Demystifying LLM-based Software Engineering Agents*. arXiv:2407.01489v2, 29 October 2024. Paper

Open source (<https://arxiv.org/html/2407.01489v2>). Used in chapter 21.

Reference 45

Puma, Passeur. Operational report.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/docs/plans/global-passeur-profile-upgrade/reports/verification.md#L97-L105>). Used in chapter 22.

Reference 46

Puma, Passeur. adapter repair.

Open source (<https://github.com/MrScripty/Passeur/commit/be0185898f5013f551a008d51c826fb6ffaea86e>). Used in chapter 22.

Reference 47

Steve Yegge, 15 September 2026. Seats and Sunsets

Open source (<https://yegge.ai/essays/seats-and-sunsets/>). Used in chapter 22.

Reference 48

Puma, Pumas-Library. Public-boundary test.

Open source (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-core/src/api/hf.rs#L1985-L2378>). Used in chapter 23.

Reference 49

Puma, Pumas-Library. recorded execution and limits.

Open source (<https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/docs/plans/artifact-acquisition/execution-ledger.md#L707-L720>). Used in chapter 23.

Reference 50

Simon Willison. Agentic Manual Testing. Agentic Engineering Patterns, living guide, accessed 2 October 2026.

Open source (<https://simonwillison.net/guides/agentic-engineering-patterns/agentic-manual-testing/>). Used in chapter 23.

Reference 51

Simon Willison. Anti-patterns Things to Avoid. Agentic Engineering Patterns, living guide, accessed 2 October 2026.

Open source (<https://simonwillison.net/guides/agentic-engineering-patterns/anti-patterns/>). Used in chapter 24.

Reference 52

Salvatore Sanfilippo, 13 July 2026. Control the ideas, not the code

Open source (<https://antirez.com/news/169>). Used in chapter 24.

Reference 53

Daniel Stenberg, 10 June 2026. A human in control

Open source (<https://daniel.haxx.se/blog/2026/06/10/a-human-in-control/>). Used in chapter 24.

Reference 54

Puma, Passeur. Transport observation.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/src/agents/codex/transport.ts#L124-L144>). Used in chapter 25.

Reference 55

Puma, Passeur. descendant fixture.

Open source (<https://github.com/MrScripty/Passeur/blob/71bcac5a89519c9c03ada01f5f9834f0f6f2a227/tests/native/codex-transport.test.mjs#L80-L92>). Used in chapter 25.

Reference 56

Puma, Pumas-Library. Crash-boundary regression.

Open source (https://github.com/MrScripty/Pumas-Library/blob/42a198bfc73e9f32e5107bc57fe5c25ed7e3c2c9/rust/crates/pumas-app-manager/src/version_manager/mod.rs#L2982-L3259). Used in chapter 25.

Reference 57

Armin Ronacher, 7 September 2026. Astra for Coding: Why Are We Doing This Again?

Open source (<https://lucumr.pocoo.org/2026/9/7/astra-why/>). Used in chapter 25.

Reference 58

Joel Becker, Nate Rush, Elizabeth Barnes, David Rein. *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*. arXiv:2507.09089v2, 25 July 2025. Paper record; researcher report

Open source (<https://arxiv.org/abs/2507.09089v2>). Used in chapter 26.

Reference 59

Joel Becker et al., METR. *We are Changing our Developer Productivity Experiment Design*. 24 February 2026. Researcher update

Open source (<https://metr.org/blog/2026-02-24-uplift-update/>). Used in chapter 26.

Reference 60

Daniel Stenberg, 22 April 2026. High-Quality Chaos

Open source (<https://daniel.haxx.se/blog/2026/04/22/high-quality-chaos/>). Used in chapter 26.

Reference 61

Puma, Coding-Standards. Exact-head run completed failure; 613 Engine tests, one failure INVALID_SEMANTIC_REVISION. Complete verification not reached.

Open source (<https://github.com/MrScripty/Coding-Standards/actions/runs/36947175564/job/110651832967>). Used in chapter 27.

Reference 62

Puma, Coding-Standards. Fail-fast sequential verification; complete verifier after Engine suite.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/.github/workflows/purpose-separated-engine.yml#L1-L78>). Used in chapter 27.

Reference 63

Puma, Coding-Standards. Proposal projection test hard-codes accepted revision 1/proposed 2.

Open source (https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/tools/standards_engine/tests/test_analysis.py#L2191-L2211). Used in chapter 27.

Reference 64

Puma, Coding-Standards. Written-plan-applicability revision is 2.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/evaluation/standards-effectiveness/policy-units/planning.toml#L45-L49>). Used in chapter 27.

Reference 65

Drew DeVault/SourceHut, 27 August 2026. Changes to SourceHut's terms of service regarding LLMs

Open source (<https://sourcehut.org/blog/2026-08-27-tos-changes-and-llms/>). Used in chapter 29.

Reference 66

Siru Ouyang et al. *RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph*. arXiv:2410.14684v2, 18 March 2025; ICLR 2025. Paper

Open source (<https://arxiv.org/html/2410.14684v2>). Used in chapter 30.

Reference 67

Puma, Coding-Standards. Recovery plan.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/plan.md>). Used in chapter 31.

Reference 68

Puma, Coding-Standards. pilot report.

Open source (<https://github.com/MrScripty/Coding-Standards/blob/dcc56f26e884ade260770beceba2501d3746200d/docs/archive/plans/planning-proportionality-recovery/reports/downstream-ordinary-plan-pilot.md>). Used in chapter 31.

Reference 69

Barry Boehm, edited by Wilfred J. Hansen, “Spiral Development: Experience, Principles, and Refinements,” CMU/SEI-2000-SR-008 (2000), official PDF

Open source (https://resources.sei.cmu.edu/asset_files/specialreport/2000_003_001_13655.pdf). Used in chapter 31.