

The engineering pipeline

Maxwell Connolly · August 2026

SUMMARY

This is the third of four papers. The first described a set of products with sharply different requirements, built by one operator through a single pipeline; the second described the coordinator that manages and dispatches the work. This paper describes the pipeline itself: the delivery machinery that carries an idea from ideation through design, requirements, build, testing, and deployment, with the controls applied at each step. No single stage here is new. The subject of the paper is the way the ordinary stages are wired into one enforced, self-auditing pipeline that a single operator runs. The standard is enforced by machines, and a separate machine checks daily that each control is still enforcing, because a check that has quietly stopped can still report green. Every figure below was read from the repositories and the running system in August 2026.

1 What this is

The pipeline described here is one method applied across every product, whatever the product is. The same discovery discipline shapes a consumer app's visual identity and an enterprise platform's data model, the same gates block a merge on a marketing site and on a database of minors' records, and the same go-live checklist wakes production for each. What changes between products is the content of each stage; the stages themselves, and the gates between them, stay fixed.

The pipeline makes one testable claim: that a single operator can hold a product to the standard an enterprise security review expects. It meets that claim because every control is enforced by a machine that runs on each change, and a second machine checks daily that the controls are still enforcing. The stages below run in order. The first stage, ideation, is deliberately loose. The load-bearing stage is gating and testing, where the standard is enforced and its enforcement is proven.

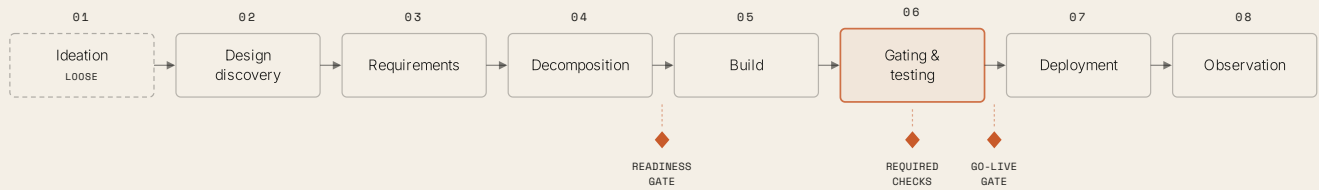


Figure 1 — The pipeline, end to end. Eight stages in order, one method applied to every product. The first stage, ideation, is deliberately loose; the seven that follow harden the idea. Three gates refuse to pass work that is not ready: a readiness gate before build, the required-checks gate at gating and testing, and the go-live gate before deployment. Gating and testing is the emphasized stage.

2 Ideation and value validation

The pipeline that follows hardens an idea into a defensible product quickly, and it will do the same for a weak idea as faithfully as a strong one, producing something well-built that no one needs. Ideation is the stage that decides whether an idea is worth hardening at all. It stays loose and ungated on purpose, to avoid constraining the creative, exploratory work before an idea has taken form.

The work of the stage is to find a direction worth building: a claim specific enough to design against. Getting there means pressure-testing the idea against a handful of areas before it firms into one.

- **The value, and who holds it.** What problem the product removes, and for whom, stated concretely enough to be wrong about.
- **The demand.** Whether the need is real and observable rather than assumed, tested against market research and direct signal from the people who would use it.
- **The differentiation.** What the product does that the alternatives do not, and whether that difference is one a user would pay for.
- **The binding constraint.** The single requirement that will govern every later decision: strict data isolation for one product, a direct-manipulation interface for another, a distinctive visual identity for a third.
- **The hardest risk.** The one technical or design unknown most likely to sink the build, named early so it can be tested with a cheap spike before the pipeline commits.

The medium is cheap and the commitments are reversible, so a wrong turn costs a conversation rather than a build. When these areas resolve into a concrete premise, the pipeline fires and every stage after it hardens that premise against research in place of intuition.



Figure 2 — The loose front. Ideation carries no gate: market research and whiteboard work converge to a shape specific enough to design against. When the shape holds, the pipeline fires, and every stage after it hardens the idea against high-quality research.

3 Design discovery — from a loose idea to a tangible product

Design discovery is where an idea becomes something you can see and touch. It takes the loose, theoretical shape and makes it tangible: concrete screens, a real interaction model, a locked look. Its first discipline is that this direction is established against a real reaction to inexpensive artifacts, before the engine is built. The rule was learned the expensive way. On one product, a working user’s reaction to the running interface drove a complete redesign of its component system after the engine was already underway. Establishing that direction against cheap artifacts first would have cost a day; discovering it against a built engine cost far more.

Discovery also settles the **soul of the product**, and that judgment sets the path the rest of design takes. The soul is the felt identity a user meets first: for one product, a high-contrast hand-drawn zine with a mean streak; for another, the quiet restraint an enterprise buyer trusts. How much a product’s value rests on that identity decides which of two asset paths discovery follows. A product whose identity carries the value takes the advanced path: concept art per screen, a component audit, a bespoke cutout library, and a contact sheet that audits its own completeness. A product whose value is the workflow takes the standard path and composes from the shared component library. Both paths converge on a signed comp, and from there the pipeline continues as one.

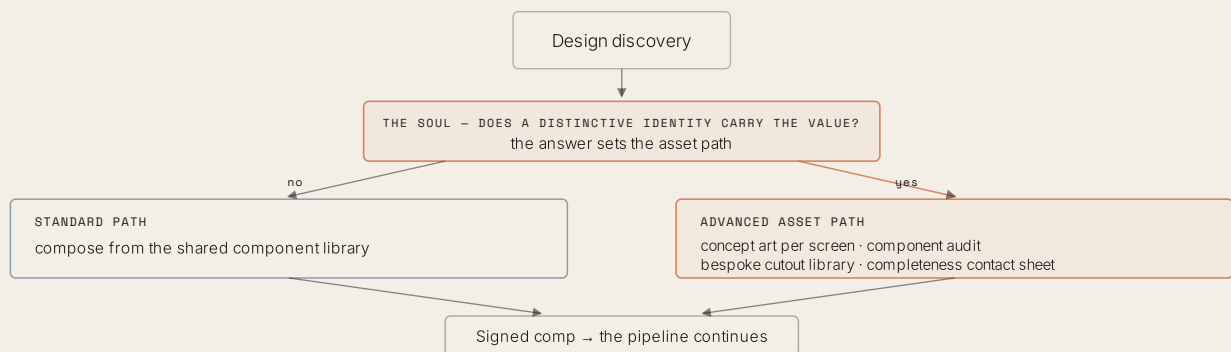


Figure 3 — The soul sets the path. Discovery judges how much a product’s value rests on a distinctive visual identity, and that judgment routes the work. The advanced path generates a bespoke identity screen by screen; the standard path composes from the shared library. Both converge on a signed comp.

Three mechanisms carry the stage.

- **Concept before comp, and a written voice specification.** Concept screens show the destination; they do not state the mechanics. Before the first production comp, the concept is distilled into a one-page specification of concrete visual rules: palette hexes, contrast level, linework, texture, frame style, type treatment, and the single success accent. That specification is signed, and the build is made to it precisely.
- **A component audit per screen.** For an advanced surface, each screen is audited into a ranked list of missing assets, so generation follows a gap list in place of intuition. On one consumer product, a distinctive identity was held across twelve screens by a single library of 64 cutouts, each generated in isolation on a locked palette, background removed and transparency verified, rendered together as one contact sheet that serves as the completeness audit.
- **A design delta, and a readiness gate.** The approved mock is treated as the front end: the build lifts its components and wires them to real data and access control in place of re-deriving the interface. A delta pass lists the behavioral gaps between the signed design and the running product. An end-to-end review of every screen is a gate that must pass before implementation begins, applied to every product without exception.

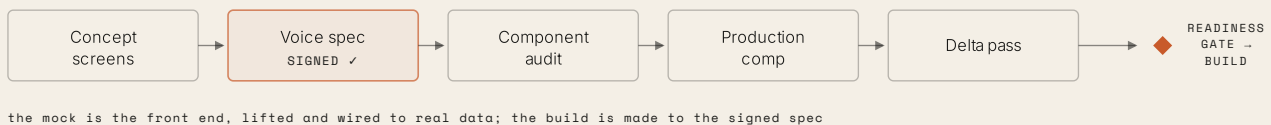


Figure 4 — Discovery, made mechanical. Concept establishes the destination; a signed one-page voice specification fixes the rules before the first production comp. A per-screen component audit drives asset generation from a gap list, and a delta pass names the behavioral gaps before a readiness gate clears the design for build.

4 Requirements — the PRD as the spec that bounds scope

A signed product requirements document is the specification, and it does one thing the rest of the pipeline depends on: it bounds scope. Work is built to the document and stops at its edge; an idea that is not carded is not in scope until it is. The document is the single source of truth for what is being built and why, and it is the running log of intent, so any session, or the operator on another machine, resumes from it.

The document is also the **anchor every later stage dereferences**. Decomposition reads it for the cards, the build reads it for what to build, review reads it for what is in scope, and the go-live checklist reads it for what the product was meant to be. Every stage orchestrates from one source instead of re-deciding locally, and that is what keeps a solo-

run build coherent as it grows: consistency in the finished product traces to a single anchor rather than to a hundred small choices made in the moment.

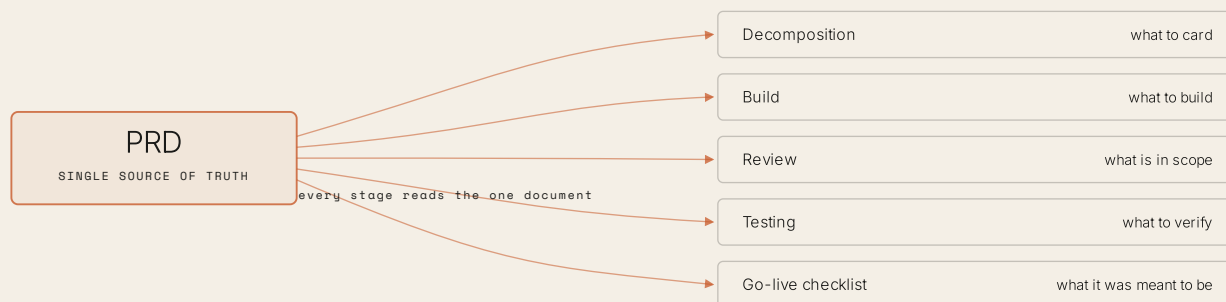


Figure 5 — One anchor, dereferenced. Each downstream stage reads the PRD rather than deciding for itself, so a solo-run build stays coherent as it grows. The cohesion of the finished product traces to a single source instead of a hundred choices made in the moment.

Two decisions are made at this stage rather than deferred, because deferring them is how a product outgrows the assessment it was built to.

- **Data classification and security level are one decision.** Every product is placed on a four-level data scheme (public, internal, confidential, restricted) at requirements time, and its application-security level follows from what it holds instead of being chosen separately. A product holding minors' data or payment instruments is restricted, and the higher assurance level is required.
- **Every third party is entered before data flows.** A vendor that will receive user data is recorded in a subprocessor register, with the data level it receives and the state of its data-processing agreement, as a requirements-time and setup-time obligation. The initiation and decomposition steps carry the gate, so a feature that routes data somewhere new produces its own register entry as a build card.

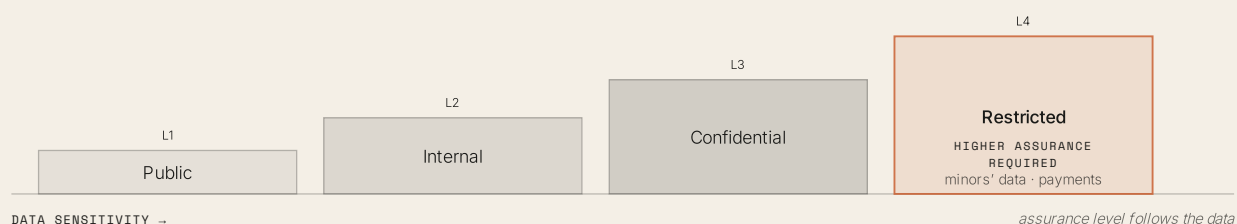


Figure 6 — Classification sets assurance. Every product is placed on a four-level scheme at requirements time, and its security level follows from what it holds. A product on the restricted level, covering minors' data and payment instruments, takes the higher assurance level as a requirement.

The product's shape also selects a profile at this stage (application, library, or internal service), and the profile sets the branch model, the CI gates, and the deployment wiring the later stages apply.

5 Decomposition — a signed design becomes dispatchable work

Between a locked specification and a running team sits a mechanical step: the specification is decomposed into build cards, one per surface, each carrying a plain-language statement of what and why, concrete acceptance criteria a closeout can verify, the security-sensitive flags (tenancy, personal data, authorization) so review is scoped up front, and sequencing notes. The cards are placed in the board's ready column before any work starts.

The card is the unit the rest of the pipeline runs on. It is the requirement and the running log at once: it moves across the board as work proceeds, it carries a comment at each decision and a hand-off note when work pauses, and it is read in full (card, thread, repository state) before it is picked up. A surface that cannot be named as a card ships uncarded, which is the failure this discipline exists to prevent.

A readiness gate stands between the cards and dispatch. It confirms the design is signed, the deltas are run, and each card carries acceptance criteria, and it refuses to release work that is not ready. Decomposition creates the cards; the gate clears them for a team.

6 Scaffolding the environment — infrastructure in phases

Infrastructure is provisioned in phases, matched to how much is known. The repository is stood up first, early enough that design has a real home: a preview environment is wired to it so that navigable mockups deploy from the first commit, while the production stack stays deliberately unprovisioned. Committing to a full stack before a product's form is settled is how a project accretes infrastructure it later has to unwind.

A new repository is born with the standard applied. The project template carries the gate workflows, the fuzzing scaffold, the dependency and secret configuration, and the standards the later stages check; a repository scaffolded from it arrives with the security scanners running on its first commit and the dependency watcher already active. This makes the standard uniform across products instead of proportional to the age of the repository.

The runtime stack is provisioned once the design is locked and the build is about to begin. Every product draws from the same set, so the pipeline reasons about one stack rather than many: a hosting platform for the web tier, a Postgres database with row-level security and object storage, a managed identity provider, a jobs-and-cron runner, transactional email, error and product analytics, a model gateway, and a single secrets manager. Secrets are read from that manager at runtime and are never committed; the repository never holds a credential.

Production is the deliberate exception. The production environment is created but left unwired (no separate production database, no live domain, no migrate-on-deploy) through the entire build. A product serves only its development environment until go-live, when production is woken on purpose. That final act is the subject of the deployment stage.

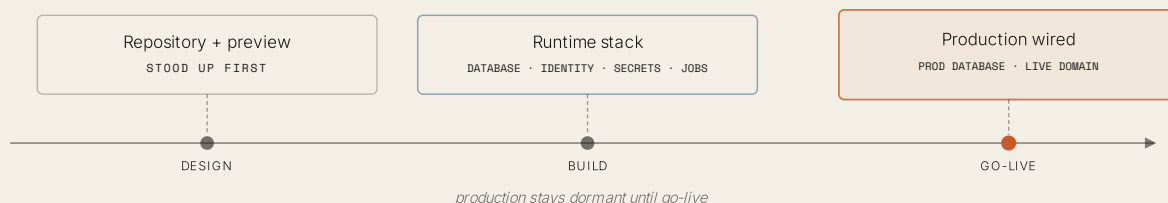


Figure 7 — Infrastructure, provisioned in phases. The repository and a preview environment come first, so design has a home. The runtime stack is stood up when the build begins. Production, its database and live domain, is wired only at go-live, and stays dormant until then.

7 Build — dispatched execution on a standard foundation

Work is dispatched as durable orders through a coordinator that turns a request into a queue row executed by a daemon, so a task survives the session that created it, and the row is the registry of what is running. A team of agents runs against a card, each agent in an isolated working copy so parallel work cannot collide, and the operator is interrupted only for a judgment, with a recommendation attached.

The branch model is uniform. An integration branch is the trunk every change lands on, and it deploys to a development environment on a development database; production is a protected branch, advanced only by a deliberate integration-to-production request, serving the live domain on a separate production database. A release is a single reviewed act.



Figure 8 — One trunk, one reviewed release. Every change lands on *develop*, which continuously serves a development environment on its own database. Production is the protected *main* branch on a separate database and domain, advanced only by a single reviewed integration-to-production request. A release is a deliberate act.

8 Gating and testing — where the standard is enforced, and proven

This is the stage the pipeline is built around. A change reaches production only through a pull request, and a pull request merges only when a defined set of checks pass. The checks are required in branch protection, so a red check blocks the merge; a workflow that runs without blocking is treated as absent.

CHECK	WHAT IT BLOCKS
Static analysis (Semgrep)	Dangerous code patterns, such as injection or unsafe handling, fail the build on findings.
Secret scanning (gitleaks)	A committed credential fails the build; it cannot reach the default branch.
Dependency scanning (osv-scanner)	A known-vulnerable dependency cannot merge; a release-age cooldown holds back freshly published versions.
Typecheck and build	Code that does not type or does not build is refused.
Coverage floors	A drop below the configured floor fails the run instead of printing a report.
Cross-tenant isolation	On the multi-tenant products, a change that lets one tenant reach another's data fails against a policy-capable database.
Browser end-to-end	The behavioral flows (invite, authentication, the builder) are exercised in a real browser in CI.
Continuous fuzzing	Parsing and boundary code is fuzzed on every change in the code it touched; a crashing input fails the job.

Two disciplines make the checks trustworthy rather than decorative.

- **Enabled is not proven.** A guard is not accepted until it has been observed failing: the build is run with the value absent to confirm the check goes red, and a hostile value is injected to confirm the check catches it. The red observation is the acceptance evidence, and the reviewer produces it. Continuous fuzzing earned this standing on adoption, finding a real cross-tenant path traversal in one product and an input-handling crash in the pipeline's own tooling on its first runs.
- **A separate machine audits enforcement.** A conformance sweep reads every repository daily and reports, per control, whether it is compliant, present but not enforcing, or absent and relevant. It exists because a check that reports success while never running is the failure a passing dashboard hides. The sweep reads seventeen controls across the estate, and it reads the project template itself, so a standard silently dropped from what new repositories inherit is caught by machine. Its own failure mode, reading clean on a partial read, is guarded: an incomplete sweep fails loud instead of reporting green.

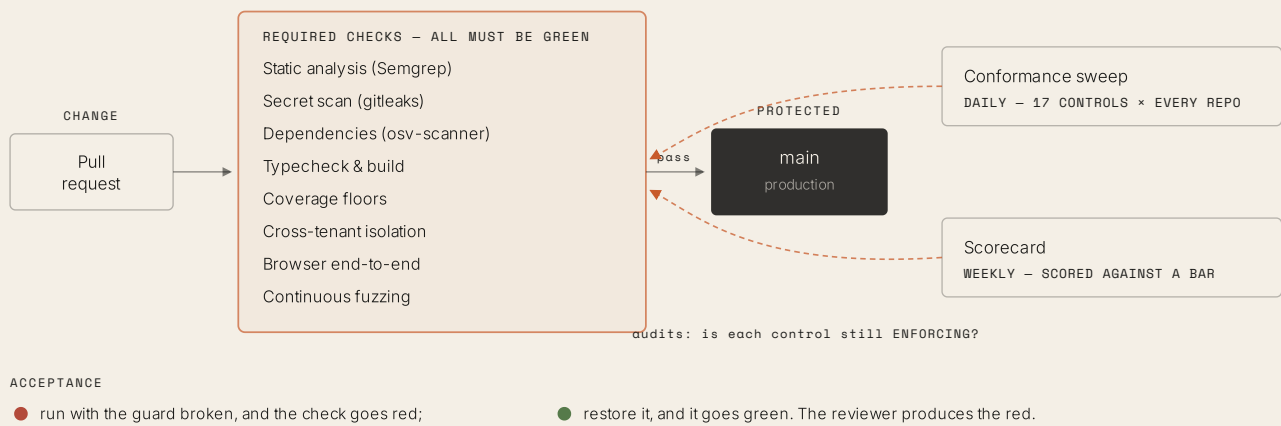


Figure 9 — The machine that watches the machine. A change enters production only by passing every required check. A separate conformance sweep reads each repository daily, and a scorecard weekly, auditing whether the checks are still *enforcing* and not merely present. A guard is accepted only after it has been observed failing.

Alongside the sweep, an automated scorecard re-runs on a schedule and scores every repository against a published bar, and a document-control job holds the canon documents to their own review and retention triggers. The pipeline measures itself on a fixed cadence instead of at a person's discretion.

9 The standard underneath

The gates described under gating and testing are the enforcement surface of a written standard, and the standard is measured. Four external references define it, each at a named version: an application-security verification standard for whether the product is sound, a secure-software-development framework for whether the process is sound, a supply-chain-integrity level for whether a build can be traced to its source, and an automated scorecard as a directional proxy. Every requirement the pipeline claims to meet names the clause it satisfies and the date it was read.

The standard propagates forward, through the template a new repository is born from, and it is verified outward, by the sweep that reads every existing repository against it. Branch protection is itself held to a written rule set and checked daily, because a protection setting that quietly reverts would otherwise pass unnoticed. Where a control cannot be reached from a single seat, an independent code review for instance, the exception is declared in the standard with the compensating controls named and the condition under which it retires.

10 Deployment — waking production deliberately

A product is born with production dormant: only the development environment serves, and the production environment of the same hosting project is unwired, which is correct until go-live. Deployment is the deliberate act of wiring it: pointing the production environment at a separate production database, attaching the live domain, and turning on the production migrate-on-deploy arm. After that, a release is a single integration-to-production request.

A go-live checklist gates the cutover, and its first section is not infrastructure. Before production carries real data, it re-asks the requirements-stage questions against the product as it actually shipped: what data classification does it hold now, does its assurance level still match, is every vendor in its chain in the register with an accepted agreement, and is the identity provider on a production instance. A product can outgrow the assessment it was built to during its own build, and the checklist is where that is caught, before real users arrive.

11 Observation

Three products built through this pipeline required very different things: a security posture that survives enterprise review, an interaction model a document engine cannot provide, and a visual identity that survives translation to components. They passed through one pipeline: one discovery discipline, one requirements gate, one build foundation, one set of enforced checks, one deployment method, and one standard measured the same way across all of them.

What makes this hold from a single seat is that the controls are mechanical and a second control audits them daily, so nothing depends on being remembered. A product built this way reaches the standard an enterprise review expects, where a fast solo build stops at something that runs but cannot be defended.

RELATED DOCUMENTS

Paper 01 — Three products, one pipeline. The products this machinery was built to produce, and the divergence in their requirements that makes a common method worth examining.

Paper 02 — Helm: an experiment in AI continuity. The coordinator that manages and dispatches the work this pipeline runs, and the instrument that certifies its memory.

Paper 04 — The measured standard. The security standard this pipeline enforces, measured from the inside: the adjusted scorecard, the watchers that audit

enforcement, and the safety nets.