ZTHR

ZTHR: A Pattern Language for Autonomous Software Delivery

Zero Trust – High Reliability: why software built by AI should earn its quality from the environment, and how to build that environment

Russ Little · Pattern paper · Oct 29, 2026 · v1.0 · Request for comments · © 2026 Russ Little. Licensed under CC BY 4.0. DOI: [10.5281/zenodo.23826734] (https://doi.org/10.5281/zenodo.23826734)
Pattern paper · https://zthr.ai

Abstract

Zero Trust – High Reliability (ZTHR) is a way to deliver software written by AI without making human code review the thing that holds quality together. It rests on two principles.

- **Symmetric Zero Trust.** No author, reviewer, model, operator or report gains authority by asserting success. A claim counts only on evidence the claimant does not control.
- **The Fixture Inversion.** Quality moves out of the author's skill and into the environment the author works in. Governed parts constrain what can be built, checks proven to fail on their target defects judge what was built, and protected statements of user intent define what the result must still do. Generators iterate, across as many attempts as it takes, until those fixtures are satisfied.

This paper explains why those two principles follow from the economics of cheap generation, then sets out the recurring solutions that implement them as a pattern language: thirty-three patterns in six families, from construction through runtime. Each pattern names a problem, the forces acting on it, the solution and its consequences. The paper closes with how the patterns compose, how people's work changes, how to adopt the approach in stages, how to evaluate it honestly, and what it does not claim.

> **Central proposition.** Nothing is accepted because it says so. A candidate must satisfy its acceptance conditions; the checks must have demonstrated that they can fail; the deployment must be observed; and consequential actions must stay inside authority boundaries enforced independently of the actor.

Author's note. ZTHR grew out of practice before it became a theory. Its core moves (measuring the as-built system against a datum, calibrating models to reality, and letting fixtures rather than craft hold tolerance) come from driving manufacturing robots from 3D models in the late

1990s, and from the decades since spent leading software teams and running large application portfolios.

---

1. The problem: generation outran assurance

Generative models have made a candidate implementation cheap and the decision about whether it is fit to use expensive. Generation now outpaces the attention available to inspect, integrate, deploy and maintain what it produces. A workflow that speeds up authorship while leaving every later decision in a human queue moves the bottleneck rather than removing it.

1.1 Three failure modes

When assurance stays anchored to human inspection of AI output, three failures follow.

- **The reviewer bottleneck.** Under volume, reviewers either become the constraint on delivery or approve changes they have not truly checked. Approval becomes evidence of attention, not of correctness, and attention is least reliable exactly when volume is highest.
- **The echo chamber.** Replacing the human reviewer with the model that wrote the change, or one like it, gives the appearance of review without its substance. Models that share training data and conventions share blind spots.
- **Probabilistic governance.** Instructions and guardrails in a prompt are useful, but they cannot be the final authority over access, data boundaries or business invariants. A safety property that depends on a model following its instructions is a property that holds most of the time.

1.2 Why better models do not close the gap

Each new generation of models writes better code, and the instinct is to wait for one good enough to trust. That instinct repeats the craft assumption that quality lives in the author. A better author still needs evidence that a particular change is right, and that evidence has to come from somewhere the author does not control. Better models shorten the path to an acceptable candidate. They do not supply the acceptance.

1.3 What ZTHR proposes instead

ZTHR makes the repeatable parts of engineering a property of the environment in which software is produced. Identity, authorization, data access, audit, deployment and failure reporting arrive with the foundation an application is built on. Acceptance is decided by evidence the author cannot bypass. People spend their expertise on defining intent, building and governing the checks, and judging consequential exceptions, rather than reading every diff.

Three kinds of assurance do the work, and the architecture composes them rather than asking any one to carry everything.

KIND	WHAT IT DOES	EXAMPLE	WHAT IT DOES NOT ESTABLISH
Structural	Removes an ability	A database role with no write privilege	Anything about behavior that remains possible
Deterministic	Evaluates an explicit property	Comparing the serving artifact's hash with the built one	Business correctness; a hash establishes identity only
Empirical	Measures behavior on examples	Model review, user-journey tests, adversarial cases	Behavior outside the examples; even a repeatable classifier has an error rate

A wrong value caught by a deterministic check, a prohibited implication caught by a model judge, and a write prevented by a missing privilege are three different protections. A system that has all three does not need any one of them to be perfect.

2. Principle one: Symmetric Zero Trust

Symmetric Zero Trust (SZT) says how a claim acquires authority: only through evidence and enforcement outside the claimant's control. In network security, zero trust means location confers nothing; no request is trusted because of where it came from. SZT applies the same rule to everything in a delivery system that can make a claim.

2.1 Four directions

The principle is symmetric because it runs in every direction, including toward the parts of the system that do the checking.

DIRECTION	WHO IS UNTRUSTED	WHAT THEIR ASSERTION CANNOT DO
Build time	The author, human or model	Its commit messages and explanations are not evidence, and cannot waive a required check
Run time	A model inside a running application	Its confidence cannot grant access, change a record of truth or authorize a consequential action
The checks	Every gate, scanner and test	A green result counts only if the check has been shown able to go red, and only if it actually ran
The operators and reports	Automation that acts, and documents that describe	A dispatched action is not a result until a later observation reads the outcome; a figure in a document is trusted only if derived from the system it describes

2.2 The enforceable rule

Stated as something a system can enforce, SZT becomes:

> **No participant can unilaterally redefine, suppress or substitute the evidence that authorizes its own consequential action.**

This does not require every component to be independent of every other. It requires explicit authority boundaries and a clear account of which mechanism enforces each one (Section 11).

2.3 What SZT rules out

SZT rules out the single powerful agent trusted to do everything well. A ZTHR system is an orchestrated factory instead. Generators, deterministic tools, independent reviewers and an automated operator each do one bounded job, and each is checked by something else.

SZT is the negative half of the architecture. It says what is not trusted. It does not say why software built this way is good. That is the job of the second principle.

---

3. Principle two: the Fixture Inversion

Quality stops being a property of the author, confirmed by inspecting what the author produced. It becomes a property of the environment the author works in: forced by fixtures, and met by a generator that iterates until every fixture is satisfied.

3.1 From craft quality to fixture quality

In craft production, quality lives in the craftsperson. A cabinetmaker produces a tight joint because of skill, and a master confirms it by inspection. Conventional software follows the craft model. Quality is attributed to the engineer who wrote the code, and code review is the master's inspection.

Manufacturing left the craft model two centuries ago. Interchangeable-parts production moved quality out of the worker's hands and into jigs, fixtures and go/no-go gauges. An operator of ordinary skill, working inside a good fixture and checked by a good gauge, produced parts to tolerances no craftsman could hold consistently. Skill did not disappear. It moved upstream, to the people who designed the fixtures and the gauges.

ZTHR makes the same move for software. The author, model or human, is a generator of candidates. Quality is defined, enforced and evidenced by the fixtures. A rejected candidate is a normal part of development. An unsupported claim of success is not.

	CRAFT MODEL	FIXTURE MODEL
Where quality lives	The author's skill	The fixtures
The author's role	Expert producer	Generator of candidates
Unit of work	The finished change	One generation, one attempt

	CRAFT MODEL	FIXTURE MODEL
Cost of a failed attempt	An engineer's time and attention	Compute and minutes
What review inspects	The output	The fixtures, and whether they are proven to bite
Where expertise is spent	Writing and reviewing code	Defining intent and building fixtures
What improves over time	Individual engineers	The shared fixture set

3.2 Why cheap generation makes the inversion rational

The craft model was rational when attempts were expensive. A person produces a few candidate implementations in a day, and each failure costs hours of scarce attention, so the investment goes into getting the first attempt right.

Generative models cut the cost of an attempt by orders of magnitude. When attempts are cheap and evaluation is reliable, the rational strategy is generate-and-test: produce candidates, let fixtures judge them, discard failures without regret, keep what passes. First-pass correctness stops being the goal; convergence is. A wrong attempt is a normal intermediate state, the way a part that fails a gauge goes back to the machine.

Both kinds of investment still matter, but they pay differently. A stronger generator shortens the path to acceptance. A more faithful fixture raises what acceptance means, and keeps paying on every future attempt of every application.

3.3 Anatomy of a fixture

A **fixture** is a governed mechanism that constrains a capability, evaluates a defined property, or controls progression on evidence. A fixture that forces quality, rather than merely reporting on it, has six properties.

1. **Executable.** It produces a verdict by running, not by being read.
2. **Independent of the author.** The candidate cannot supply its own verdict, and the generator does not write the fixture in the same change it is judged by.
3. **Proven to bite.** It has been shown to fail on the defect it exists to catch.
4. **Locked against the candidate.** A change cannot silently weaken the fixture that judges it.
5. **Three-valued.** Its verdict is pass, fail or unknown. Unknown withholds whatever permission a pass would grant, and stays distinct from a detected defect.
6. **Purpose-carrying.** It states what it protects, so a generator working to satisfy it is steered toward the intent and not only the letter of the check.

Fixtures differ in strength. Removing a privilege is structural, comparing hashes is deterministic, a model review is empirical. Calling a model judgment a fixture does not make it deterministic; it means the judgment has a defined task, a bounded output and a defined place in the acceptance policy.

3.4 Jigs before gauges

A jig constrains the work so it can only be made one way. A gauge measures the result. A gauge alone, applied to unconstrained work, rejects most of it and says little about why. A jig alone produces consistent parts with no evidence they are right.

The jig is what makes iteration converge. A generator searching a constrained space reaches a passing candidate in few attempts. One searching an unconstrained space may never converge, and its gauges would have to anticipate every architecture it might invent. Verification effort scales with the freedom given to the author, and every decision moved into a jig is a class of defect no gauge has to catch.

3.5 Fixture fidelity is the ceiling

A generator iterating against fixtures produces what the fixtures demand and no more. That is the inversion's power and its central risk. Quality is bounded above by **fixture fidelity**: how closely the fixture set captures what correct means.

Optimization finds slack. A model iterating to green will exploit any gap between a fixture's letter and its purpose: an assertion satisfied trivially, a check that inspects the wrong surface, a budget met by a degenerate path. ZTHR treats this as expected behavior and answers it structurally, with fixtures proven to bite, expected values derived independently of the code under test, intent the author cannot edit, held-out evaluation the generator never sees, and reviewers who judge a change without its story.

Slack, once found, becomes a fixture. Every **escape**, meaning a defect that passed every fixture, is evidence of a fidelity gap, and the learning loop (Section 12) converts it into a new check. The honest claim is that fixtured software is as correct as its fixtures are faithful, and that fidelity is visible, measurable and improvable by a defined process.

3.6 What compounds

Code in the craft model is a liability that grows with every line. A fixture, built once, applies to every future attempt at every application that inherits it. An authorization check proven to bite protects every application built on the foundation that carries it. A user journey written once judges every future change to the screens it walks. Maintenance and false-positive handling remain real costs, but they are spread across the whole estate.

This is the economic engine. Engineering effort moves from producing code, which does not compound, to producing fixtures, which do. The strategic asset is the maintained relationship between intent, controls and observed outcomes.

Fixtures also make model change a non-event. Because they judge outcomes, not model responses, every build puts whatever models are in use through the full battery, whether a version change was announced or not. Variance that matters surfaces as drift against the datums and can be chased to its root. Revalidating a new model becomes a byproduct of the fixtures rather than a separate program.

The same engine removes the supposed trade between quality and scale. Each fixture replaces a person who would otherwise have to watch for that failure, so one builder or a small team can operate at a level and a standard that conventional teams reach only with far more people. Quality is not traded for reach; the fixtures are what make the reach possible.

3.7 What fixtures cannot force

The largest risk in a fixtured system is that it builds what it is told. A bad specification or a bad process definition is enforced as faithfully as a good one, so the quality of what goes in sets the ceiling on what comes out. Section 14.2 describes how the factory catches a bad specification before it is built.

- **Whether the intent is the right intent.** Fixtures verify against what was asked. If a domain expert specifies the wrong workflow, every check may faithfully enforce it.
- **Properties nobody has encoded:** a requirement no journey expresses, a failure class no gate names.
- **Qualities that resist mechanical judgment:** design coherence across many changes, clarity for the next reader, fitness for a use no one has imagined yet.
- **Novel domains** with no record from which to derive an expected value.

These edges define where people stay: defining intent, governing the fixtures, judging risk, and observing whether the software actually serves its users. The inversion does not remove human judgment. It concentrates it where it compounds.

3.8 The review question

The inversion changes the question every reviewer, auditor and engineer asks. In the craft model it is "is this code good?" In ZTHR it is:

> **Which fixture forces that, and has it been shown to bite?**

A quality claim with no fixture behind it is an opinion. A fixture with no proof behind it is a hope.

4. How to read the patterns

The two principles say why. The patterns say how. Each pattern is a recurring solution to a problem that appears whenever AI authors produce software at volume, written to be independent of any particular cloud, language, model provider or repository layout.

Each pattern is stated in four parts:

- **Problem:** the failure that appears without it.
- **Forces:** what makes the problem hard, or makes the obvious answer wrong.
- **Pattern:** the solution, stated as a rule a system can enforce.
- **Consequences:** what it buys, what it costs, and which patterns it leans on.

The patterns fall into six families, one for each class of fixture. Together they form four layers, construction, admission, an execution boundary and an observation loop, with a verification spine running through all four that preserves the link between a claim, its evidence, and the artifact or running state the evidence applies to.

FAMILY	WHAT IT FORCES	PATTERNS
Construction (jigs)	What can be built at all	1–6
Verification (gauges)	That code meets stated properties, and that the checks themselves work	7–14
Intent	That the product still does what its users came to do	15–18
Independence	That a reader without the author's story agrees	19–22
Operational	That permitted work keeps moving, and what ships is what passed	23–27
Runtime and environment	That model behavior stays within policy at the moment of use, and that integrations hold under conditions too costly or dangerous to create live	28–33

Candidates flow left to right through construction, admission, the runtime boundary and observation; what observation learns returns to construction as shared fixtures, and the spine keeps every claim tied to its evidence throughout.

No single pattern is the system. The unit of assessment is the composition: a failure traced through construction, verification, deployment and operation, asking which patterns prevent or detect it and what authority remains if detection fails (Section 13). An isolated weakness matters when it opens an uncovered path through the whole.

The patterns describe how and why, not a toolset. They carry over to most software engineering work, because each situation defines what success means and then builds the machine that makes sure the software does it.

5. Construction patterns: the jigs

Construction patterns reduce the number of decisions that must be made correctly during each build. Every decision moved into a governed part is one no generator can get wrong and no gauge has to check.

Pattern 1. Governed Foundation

Problem. A generator asked to build an application will invent its own sign-in, roles, audit trail, database access, deployment and health checks, differently every time. The verification surface has no bound.

Forces. These concerns are where the most serious defects live, they are tedious to get right, and they look easy enough that a generator will confidently write them from scratch.

Pattern. Every application starts from a shared foundation that already carries identity, authorization, audit, configuration, deployment and required checks. The foundation is assembled from the strongest existing implementation of each concern, released under a version, and delivered to each application as a change that must pass that application's own checks. Generators are instructed to read foundation files and never to regenerate or "improve" them locally, because patching a copy in place is how drift begins.

Consequences. A new application is born already taking part in the delivery process. Improvements are made once and travel everywhere. The cost is a release discipline for the foundation itself, which is control-plane work (Section 11).

Pattern 2. Parts Catalog with Purpose Notes

Problem. Without a catalog, every application re-solves solved problems, usually worse. With a catalog of bare code, a generator picks the part whose name sounds right.

Forces. Two parts often sound alike and are not. Parts depend on and conflict with each other, and some are only safe beside a companion control.

Pattern. Capabilities are packaged as parts, each with a manifest (what it adds, its dependencies, conflicts and required companion controls), its source, tests that fail when it breaks, and notes stating what it is for, which failure it was built against, and what it deliberately does not do. Search returns the notes. One shared planning function resolves dependencies and conflicts for every path that installs parts, so a preview can never promise a combination the installer would reject. Companion controls are structural: a part that can touch sensitive data cannot be installed without the part that detects and redacts it.

Consequences. Generators search before building and assemble from qualified parts. A gap that proves to be a genuinely new capability is written back as a part, so the next application meets it already built.

Pattern 3. Test the Part Where It Lives

Problem. A part's imports resolve against the application that installs it, not against the catalog. Tested inside the catalog's own suite, a part can fail when correct or pass when broken.

Forces. Building a realistic consumer for every test run costs time, and catalog-local tests are faster and easier.

Pattern. Every part is tested inside a temporary application assembled exactly as a real consumer would assemble it: same starting template, same dependencies merged in, same paths, compiler settings and environment. A ratchet holds the number of parts with no failing test at zero. A part copied from a live application records where it came from, and a check reports whether that origin has since moved; an origin that cannot be read is reported as unknown, never guessed.

Consequences. The promise that a part works is tested in the only context where it matters. One end-to-end job proves that a freshly downloaded starting template, with representative parts merged, actually builds.

Pattern 4. Single Source, Generated Mirrors

Problem. A fact stated in two hand-edited places will disagree with itself. Indexes, mirrors, agent instructions and documentation counts drift from the thing they describe.

Forces. Copies are convenient and local. Regenerating them takes tooling.

Pattern. Every fact the system states lives in exactly one hand-authored place. Everything else is generated from it, and a freshness check fails whenever a generated artifact differs from what its source would produce now. Every shared file is published with a content hash, so comparing a local copy yields three states, not two: identical, behind a known version (safe to update), or locally modified (reconcile explicitly). A version number cannot tell "behind" from "modified"; a hash can.

Consequences. Drift between source and derivative becomes a failing build, not a slow corruption. The rule extends naturally to documentation (Pattern 14).

Pattern 5. Describe, Never Write

Problem. AI sessions are the factory's primary users, and an interface that can write into their working copy is a remote code drop into whatever repository a credential reaches.

Forces. Letting the service apply changes directly is faster and feels helpful.

Pattern. The factory's interface for agents describes and never writes. Installing a part begins with a plan: the ordered parts, dependencies pulled in, any conflict, the companion-control verdict, every file with its hash, and the exact placement of each insertion. The session reads the plan, applies it in its own working copy, and submits the result through the normal

admission path. Guardrails run where the session works: hooks block forbidden writes and destructive commands and flag undeclared data sources, and a conformance check at session end blocks completion on a critical violation.

Consequences. Every change, however it was assembled, arrives as an ordinary candidate judged by ordinary fixtures. The capability to write remotely does not exist, so it cannot be misused.

Pattern 6. Authorship Tiers by Consequence

Problem. "The AI writes the code" treats a configuration change to an internal dashboard and a change to authorization logic as the same kind of act.

Forces. Restricting what a generator may produce reduces flexibility, and some requirements will not compose from existing parts.

Pattern. The higher the consequence a system carries, the less the generator is allowed to author. Authorship authority is a dial set by consequence.

TIER	WHAT THE GENERATOR MAY PRODUCE	WHERE IT FITS
Configuration only	Rows of configuration over governed data	Internal tools that are views over governed entities
Assembly only	Compositions of qualified parts; no new code	Safety-critical and other high-consequence systems
Code inside parts	New code, confined to a part qualified before release	The parts shop that feeds assembly
Code under fixtures	New application code, admitted by gates, review and journeys	Most ordinary applications

In the assembly-only tier the generator has no channel through which to emit code. It cannot hallucinate into the running system because nothing it produces becomes code. Each tier carries an explicit qualification standard for its parts, and the floor is measurable today: the known defect rate of human-built software. A standard that demonstrably exceeds it is a net gain, and the bar rises as evidence accumulates.

Consequences. Component-based engineering promised this decades ago and underdelivered because qualifying parts was expensive and composing them was slow, rigid human work. AI changes both ends at once: it can test small parts far more exhaustively than a human team could afford, and compose them faster than any person. Parts that each meet specification can still assemble into a system that does not, which is why assembly is gauged against intent (Pattern 15) and measured from datums (Pattern 12).

6. Verification patterns: the gauges

Verification patterns admit a change only on evidence the author cannot bypass, and they hold the instruments producing that evidence to a standard most pipelines never apply: each must be shown able to fail.

Pattern 7. Zero-Bypass Admission

Problem. A required check that someone can bypass is advisory for that someone, while the interface still shows it as required.

Forces. Bypass rights feel like a safety valve for emergencies, and the people who hold them are trusted.

Pattern. Every protected branch carries rules with no bypass actors, history rewrites and branch deletion are blocked, and every check that runs on a change is required, because a check that goes red and stops nothing teaches everyone to ignore red. The live rules are compared against a committed expectation on a schedule and on every change, so a check quietly removed through a settings page is caught. Admission proceeds through five logical stages: structural checks, behavioral properties, verification of the gates themselves, independent review, and release admission. Evidence at each stage must apply to the exact candidate being admitted.

Consequences. Emergencies use an audited break-glass path rather than a standing exemption. Automation that merges changes holds no administrative rights, so it can never alter the rules it must satisfy.

Pattern 8. Prove the Gate Can Fail

Problem. Test suites are full of vacuous passes: an assertion mocked away, a condition weakened, a path never executed. A gate that cannot fail looks exactly like a working gate on every green run.

Forces. Proving a negative for every check is expensive, and a green board is reassuring.

Pattern. For each named safety gate, a harness holds a case that plants the exact defect the gate exists to catch. Running a case confirms the gate is green on the clean tree, plants the defect, requires the gate to go red, restores the tree, and records a proof pinned to a hash of the gate's own content. Editing a gate changes its hash, makes its proof stale, and fails the build until the proof is re-established. A floor on the harness's case count stops it from passing by shrinking.

Consequences. Every named gate is a calibrated instrument. This proves sensitivity to named defects only; unnamed defects are found by mutation (Pattern 11), review, escapes and adversarial evaluation.

Pattern 9. Unknown Is a State

Problem. A wrong path, an over-broad filter or a swallowed network error yields a successful exit and a truthful "no findings." A reviewer that never ran has approved nothing, but its absence can look like a pass.

Forces. Two-valued tooling is the default everywhere, and treating unknown as failure produces more red.

Pattern. Every verdict is pass, fail or unknown, and the governed path never converts unavailable evidence into a pass. Scanners carry floors on how much they examined, and a scan that cannot prove it ran fails. A complete run that did not reproduce a finding is kept distinct from a partial run that never looked. Every measurement carries its scope, depth and the build it applies to, so a number is never read as broader than it is.

Consequences. The system can always say what it does not know. Dashboards show not measured, not yet published and could not be read as states of their own, never as zero or empty.

Pattern 10. Ratchet

Problem. A strict new gate cannot enter a codebase that does not yet meet it, so it is either never adopted or adopted as a warning nobody reads.

Forces. Requiring every author to clean up existing debt before their own change can land is unfair and stalls delivery.

Pattern. A ratchet records a floor for a measured property and fails only when a change goes past it. Existing debt may stay; it may not grow; improvements lower the floor permanently. Every gate takes one of three shapes, and knowing the shape tells any author the fix without reading the gate:

SHAPE	HOW IT BEHAVES	HOW TO RESPOND
Correctness	Fails on a fact about the code	Fix the code
Freshness	Fails because a generated file no longer matches its inputs	Regenerate; never hand-edit a generated artifact
Ratchet	Fails only when a change goes past a recorded floor	Fix what the change added

Consequences. Strict standards can be introduced anywhere, immediately. A ratchet prevents regression on what it measures; it does not by itself prove that overall quality is rising.

Ratchets apply steady pressure, and occasionally they lock in the wrong floor. A test that was off starts flagging good behavior as breaks; the test is fixed, but the cases written against the faulty version remain and fail forever, no longer fit for purpose. Every ratchet therefore has a

reset lever. Only a person pulls it, with a stated reason, and every pull is written to the audit record as a control-plane change (Section 11).

Pattern 11. Mutation-Measured Tests

Problem. Line and branch coverage are easy to satisfy with tests that execute code without asserting anything, and generative models produce such tests readily.

Forces. Mutation testing is slower than coverage, and its aggregate score can hide a serious survivor.

Pattern. Test strength is measured by planting small defects (a flipped comparison, a changed constant, a deleted call) and requiring the suite to fail on each. Scores are held by ratchets, so weakly tested new code lowers the score and fails the change. Scores are reported with their denominator and exclusions, and designated critical defect classes must be detected regardless of the average.

Consequences. A generator cannot satisfy the test bar with tests that test nothing. Mutation runs typically move to a nightly cadence, with the ratchet enforced per change.

Pattern 12. Measure From the Datum

Problem. Most verification compares each build with the one before it. A long series of individually acceptable changes can walk a system far from where it started without any single change looking wrong.

Forces. Comparing with the last build is cheap and needs no maintained reference.

Pattern. Drafting standards distinguish chain dimensioning, where each feature is measured from the previous one and tolerances accumulate, from baseline dimensioning, where every feature is measured from a common datum and each error is bounded on its own. ZTHR measures from datums under three rules. Every measured property has a declared datum: intent first, then safety thresholds, performance, accessibility, cost and output on reference inputs. Drift is distance from the datum, with a tolerance zone; outside it the gate fails, however small the last step was. Moving a datum is a governed, attributable event with a stated reason.

Consequences. A ratchet keeps a measure from getting worse than its floor; a datum keeps the whole system from wandering. Together they let the system say at any moment how far it is from its reference and whether that distance was chosen, and by whom.

Pattern 13. Register, Don't Suppress

Problem. Suppressed findings outlive the reasons they were suppressed, and a suppression that survives its finding will hide the finding if it returns.

Forces. Not every finding can be fixed the day it is raised.

Pattern. A finding not fixed immediately goes into a register with a typed verdict, a reason and, for accepted risk, a named fix and a mandatory expiry. An open finding with no register entry fails the build, and so does a register entry whose finding no longer fires. A finding counts as resolved only after consecutive complete runs fail to reproduce it.

Consequences. Accepted risk is visible, attributable and time-limited. Nothing is silently ignored, and nothing stays ignored by accident.

Pattern 14. Documentation Derived From the System

Problem. At high velocity, hand-maintained documentation diverges from reality within weeks, and a document that reads better than the system it describes misleads every reader.

Forces. Prose resists mechanical checking.

Pattern. Documentation is checked like code. Figures in prose (counts, rates, required checks, model identifiers) are extracted and compared with the live system; a figure that disagrees fails, and so does a figure that was deleted, because removing a claim must not be a way to stop checking it. Absolute words such as "always" and "cannot" must anchor to the code that makes them true. A document describing specific source files declares them by hash, and fails when one changes until someone re-reads and re-stamps it. Diagrams and counts are generated and freshness-gated.

Consequences. Documentation stays consistent with the system, and changes to the system demand attention to the documents that describe it. Substantive judgment about prose remains a human job.

---

7. Intent patterns

Unit tests, mutation scores and structural gates establish that code does what its functions specify. They do not establish that a person can still do what they came to do. A change can keep every test green while it makes a list unscrollable, pushes a button off the screen or breaks a form's progression. Intent patterns fix what users came to do, so that no generation can quietly change it.

Pattern 15. Intent as the Primary Datum

Problem. Parts that each meet specification can assemble into a product that fails its users. Manufacturing calls this tolerance stack-up.

Forces. Component checks are cheap and precise; whole-activity checks are slower and look less rigorous.

Pattern. Manufacturing verifies parts by dimensional inspection and assemblies by functional gauging: does the part fit its mate, does the mechanism move. A functional gauge catches stack-up without knowing which part drifted, because it tests the outcome the tolerances exist to

protect. In ZTHR the functional gauge is the **journey**: a whole activity a user comes to do, declared with its goal in the user's words, its success conditions, its forbidden outcomes (things that must never appear), and an action budget for taps, scrolls and waiting. Journeys are set with users and domain experts from the use cases each role must accomplish, then extended with further journeys built by one model family and tested by another.

User intent is the system's primary datum, and it sets an order of severity. A board that is entirely green while an intent is broken is a failure. A failing component test with every intent intact is a defect to fix before it reaches anyone.

An intent or process flow that cannot be completed stops value delivery, and the system treats it that way. ZTHR does not prescribe one stop rule. It requires each team to tie what breaks to what stops explicitly: which broken intents halt a change, which halt promotion, and which halt the line, so the decision is made by declared policy rather than in the moment.

Consequences. The datum is the user's goal, not a snapshot of current behavior, so it cannot enshrine an existing bug. Layout, fields and components change freely while the intent holds; the datum moves only when what users need changes, and that move is governed (Pattern 18). For behavior that is not a number, distance from the datum is completion within the action budget with no forbidden outcome, and an activity creeping toward its budget is drift visible before it fails. Every enabled capability names the journey that evidences it, or records why it has none.

Pattern 16. Walk Like a User

Problem. Tests that locate elements by style selectors or test identifiers pass when the element exists in the page but is hidden, covered, clipped or unreachable.

Forces. Selector-based tests are easy to write and stable across visual changes, which is also why they miss visual failures.

Pattern. Journeys run in a fresh browser at real device dimensions and locate elements only by accessible role and accessible name, the interface a screen reader uses. After the first navigation, the runner must reach every later screen by interacting with the application; it cannot skip a broken path by jumping to a destination address. Every judged state is read in two representations, the document with its text and roles, and the rendered pixels, and a disagreement between what the document says and what is drawn fails the step. The runner removes its own overlays first so it never reads its own captions. The interpreter that executes journeys is generic, may contain no journey-specific special cases, and must pass its own fixture set reaching every verdict it can return before it is trusted to judge the product.

Consequences. A change that preserves what users and assistive technology perceive passes; one that removes a label, covers a button or breaks focus fails. Accessibility is enforced as a side effect of testing the way people actually use the software.

Pattern 17. Oracles From the Record

Problem. Expected values typed into a test are written by the same process that wrote the code, often the same model in the same session. When both are wrong the same way, the test passes.

Forces. Hard-coded expectations are simple and fast.

Pattern. Expected values are never typed into a test. The harness signs in as a synthetic user, reads that user's data through the live interface, and derives what the screen should show from the record. Product wording comes from its canonical source declaration. Each step records four things separately: what was checked, what the record holds, what actually appeared, and the verdict. Where an activity ends with someone else, the journey checks that side too: a request submitted by one user must appear for the person who handles it, and a reply must return to the requester.

Consequences. Each assertion becomes a statement about the world rather than about the code. Oracles from the record test faithful presentation; independently specified scenarios remain the right tool for testing the calculations upstream of the record.

Pattern 18. Locked Intent

Problem. A generator that can edit the test it is judged by will eventually pass by redefining the test.

Forces. Intent must still be able to change when user needs change.

Pattern. Journey definitions belong to the product, not to whoever is changing the code. They are protected by a hash lock and an attributable change log recording whether each update tightens or loosens expectations. A change that alters journey intent cannot, in the same change, alter the application code that intent judges. Journeys end in pass, fail or blocked, and blocked has not established success; a readiness report refuses to declare ready while any required journey is incomplete or unjudged, and never substitutes an older green result for a missing current one.

Consequences. A generator cannot make its own admission easier by moving the goalposts. Changing what users need is a deliberate, visible act with an owner.

---

8. Independence patterns

Independence patterns make sure that agreement means something. A second opinion that shares the author's story, model or vantage point adds less than it appears to.

Pattern 19. The Determinism Ladder

Problem. When large language models are the default tool, they get used for jobs a simpler instrument would do more reliably, and their variability leaks into places that need repeatability.

Forces. A general model can do almost anything passably, which makes it tempting for everything.

Pattern. Use the most deterministic approach that can solve the problem. Plain code comes first. Small, specialized or non-language models and classification engines come next, where patterns alone are not enough. Generative models are used where open generation or open judgment is the task. Deterministic layers can be reasoned about, measured exhaustively on a labeled corpus, and kept running through a model outage. A classifier is still an empirical instrument even when its inference is repeatable, so every deterministic or classifier layer is measured and backed by later layers.

Consequences. Each layer carries the weight it can bear. A classifier that is wrong is wrong consistently, which hides its error; several model families, each given different context, reviewing the same outputs expose those systematic errors through their disagreements. Retraining or replacing a classifier is a fixture change and travels the governed path (Section 11).

Pattern 20. Roles Defined by What They Are Denied

Problem. A capable agent given broad access will use it, and an instruction not to is a probabilistic control.

Forces. Narrow roles mean more components and more orchestration.

Pattern. Each role is filled by an engine suited to it, and each is defined as much by what it is denied as by what it receives. Every role produces candidates or evidence; permission to act comes only from governed admission and execution mechanisms.

ROLE	RECEIVES	DENIED
Author	Requirements, the parts catalog, its repository, a scratch environment, local tests	Any write path outside its own working copy
Context-aided checker	The change, its intended behavior, integration context	The author's reasoning
Unaided reviewer	The diff and one short context page	The author's prompts, conversation, rationale and repository
Triage	Failure output, change state, check history	Any write except through the operator's single write point
Runtime responder	The current user's own data	Anything outside that data; any write to it
Runtime judge	The candidate output and an enumerated policy	The responder's configuration
Boundary screening	Inbound and outbound text	Generation of any kind

Consequences. Safety does not depend on any role choosing to stay in its lane. The denial is structural.

Pattern 21. Multi-Family Review

Problem. A reviewer who can be told why a change is fine will be persuaded by the explanation. A reviewer who knows nothing about a change cannot judge whether it fits its surroundings.

Forces. Context helps judgment and also contaminates it, and any single model family carries blind spots its own members share.

Pattern. Every change is reviewed by at least three models from different families before it is committed, each carrying a different part of the acceptance obligation. A **context-aided checker** in a fresh context, which did not write the code, is given the intended behavior and integration context and tasked with breaking the claim that the change meets its goal. An **unaided reviewer** receives the diff and a short context page and nothing else, so persuasion has no path in. An **independent code checker** examines the code on its own terms, without the author's framing or the other reviewers' findings. Verdicts are recorded against a fingerprint of the exact code state. The recommended configuration adds a fourth family after merge: a reviewer that re-reads the change in place, as integrated, as a safeguard before the release is fully promoted. Each surviving finding becomes one tracked issue. Families are chosen for difference, and a low-cost open model is a legitimate member of the set.

The reviewers, the judges and the documentation are themselves tested for drift as the system runs. A solution can drift toward a different intent while every test stays green, and that is a failure for the user, however clean the board looks. Drift in a reviewer is caught the same way as drift in a product: against declared datums and the intents they protect (Patterns 12, 15).

Model review follows rules. Questions and outputs are bounded: a finding names a location or behavior, the violated expectation and enough support to investigate. A run that could not review fails. Findings are fixed, or answered in a register entry that quotes what it cites. The reviewer is pinned to the base branch, so a change cannot ship the reviewer that approves it, and runs under an identity that can reach model inference and nothing else. A model can raise a blocking concern; no model can waive an unrelated control. Repetition is not independence: four identical passes are not four assurances.

Consequences. Persuasion is removed from one path, fit is covered by another, and the code is read a third way by a family with no stake in either. Each reviewer's contribution can be measured by the unique, validated findings it produces, and the set can be tuned by evidence rather than habit.

Pattern 22. Diversity of Representation

Problem. Authors and reviewers drawn from different model providers still share some tendencies, because all current large language models share some. Provider diversity alone is weak evidence of independent error.

Forces. Adding more model reviewers is easy and feels like more assurance.

Pattern. The strongest controls read different representations of the system. A reviewer reads source. A privilege check reads the database. A bundle scan reads the built artifact. A journey reads the rendered screen. The operator queries the running service. Several readers of the same source can share a blind spot; readers of different representations cannot all share the same one.

Consequences. This, more than the number of models involved, is the architecture's main defense against correlated failure. When designing a new control, the first question is which representation it reads, and whether anything else already reads it.

9. Operational patterns

An accepted change is useful only when it is actually serving, and only if the work around it keeps moving. When software is written by many concurrent, finite sessions, the pipeline is where work is lost. Sessions end; the pipeline does not.

Pattern 23. Done Means Serving

Problem. Merged is a state on the source host. Shipped is a running service whose artifact matches what was approved. The most expensive failure in a delivery system is believing something shipped when it did not, or that what is running is what was tested.

Forces. Deployment tooling reports its own success, and checks on a proposed change are a verdict on a different commit from the one that lands.

Pattern. Deployment sits inside the assurance architecture, not after it. Each link in the release chain refuses one specific thing.

LINK	REFUSES
Tests and security run on the merged commit	Shipping code whose integrated state was never tested
Deploy identity is federated, usable only from the main branch	A leaked key, because none is stored; a proposed change reaching deploy credentials
Deploy by the content hash read from the build	Deploying an image this commit did not build; a movable tag is never consulted
Rollback target captured before traffic moves	A rollback with nowhere to go

LINK	REFUSES
Assert serving equals built	A hand-run deploy, an unrecorded rollback, a tag moved underneath
Health check, including that unauthorized callers are refused	A broken or over-exposed deploy staying up
Inspect the served bundle	A forbidden capability reintroduced by a dependency or build step
Open every screen	A release whose pages compile and then fail
Release record, filed on failure as well as success	Having no answer to "prove the code you showed me is the code that was running"

The comparison that decides whether a deploy matched is a pure function with its own tests, and every uncertain case resolves to "does not match." A false match puts a false claim in an auditor's hands; a false mismatch only fails a pipeline. Applications declare their recovery mode (traffic rollback, forward repair, restoration or compensation), because reverting an image is not enough after a schema change or an external side effect.

Consequences. "Did it ship?" always has an evidenced answer. A fuller release manifest binds source revision, artifact, configuration, policy, model configuration, dependency lock and migration state, so verification can ask whether every element of what is serving matches the approved release.

Pattern 24. The Persistent Operator

Problem. Changes collide. Checks never start because an event was dropped. A check fails for a reason unrelated to the change. A green change waits for a merge nobody presses. A merged change's deploy never completes. None is hard to fix, but each needs someone watching all the time, and asking people to watch turns automation back into a queue.

Forces. An automated actor with write access across many repositories is dangerous if its limits depend on its judgment.

Pattern. Unfinished work gets a persistent operating owner: an automated operator that runs on a short, fixed cycle, observes every change and service in its scope, classifies each into a closed set of states, proposes a bounded remedy, acts where authorized, and verifies the outcome on a later cycle. It decides by one written rule:

> **Act, unless the action is destructive, has an outside audience, or carries a wide blast radius.**

Destructive means no single step brings back what was removed: unique commits, data, a secret, history. *Outside audience* means people beyond the team are using the surface now; on those surfaces, and only there, an action that changes what they see waits for a person. Asking

permission for something that cannot hurt anyone is not caution; it is a queue. Every hold must name the loss a person's click prevents, and a hold that cannot name its loss is removed. When the operator cannot act because something is unprovisioned, that is an engineering gap, never a decision handed to a person.

Blast radius means the action reaches beyond its own target: a shared part, the foundation, or a rule every application inherits. Those actions wait for a person even when each individual effect would be reversible.

Some states exist to refuse. Re-runs follow evidence: a check may be re-run only if its history shows it has both failed and passed on unchanged code, or its runner was reclaimed mid-job; otherwise the operator refuses and reports what the record shows, because re-running until green hides real race conditions. Green is never asserted on a check list that could not be read. An operator that always does something is less trustworthy than one that knows when not to.

Consequences. People see a short, ranked list of decisions that genuinely need them, instead of a backlog automation could clear. A dead-man's switch alerts if no cycle completes, and the operator reviews its own stalls on a slower cadence and names each class once, with the remedy owed.

Pattern 25. A Dispatch Is Not a Result

Problem. Automation that records "done" when it sends a command reports success for work that may have failed, and automation that retries an interrupted write can duplicate an action.

Forces. Verifying outcomes later means holding state across cycles.

Pattern. Work is recorded as accepted when dispatched, and a later cycle reads the real outcome. An interrupted action is marked outcome unknown and never retried blind. Each proposal is claimed once, by a single conditional update, so overlapping cycles cannot both act. Each cycle runs fixed phases in a fixed order and records the result of each, so a cycle that did nothing can say why.

Consequences. The operator's own reports are held to Symmetric Zero Trust. Ambiguity stays visible until resolved.

Pattern 26. Leases

Problem. An automated operator and a working session acting on the same branch will overwrite each other, and sessions that end silently leave branches in limbo.

Forces. Locks that never expire block work; locks that expire too easily allow collisions.

Pattern. Before work starts, a session takes a heartbeat-backed lease on a repository and branch. A second claim on a live lease is refused and names the holder. A silent lease becomes stealable after a set interval, and an expired lease over uncommitted work becomes a finding.

While a lease is live, the operator does nothing on that branch. A hook claims a lease for a session that forgot to. Writes carry the lease's generation and the expected branch head, so an owner whose lease has passed to someone else cannot complete a stale write.

Consequences. Many finite sessions and one persistent operator share a codebase without stepping on each other, and abandoned work surfaces instead of rotting.

Pattern 27. Bounded by Structure

Problem. An operator with broad write access is acceptable only if its limits do not depend on its judgment being right.

Forces. Every structural limit is another piece of plumbing to maintain.

Pattern. Each limit is enforced by something other than the operator.

IT CANNOT	BECAUSE
Act on a repository it was not assigned	Every write checks a declared allow-list before a credential is obtained, and each entry carries a written reason
Merge a change that fails its checks	Its identity lacks administrative rights, so it cannot alter the rules it must satisfy
Rewrite history	Its update path pushes only fast-forward changes
Touch a branch a session is working on	A live lease stands it down
Act twice at once	Proposals are claimed once by a conditional update
Hide what it did	Every action, skip and reason goes to an append-only ledger
Delete anything	No remedy deletes; closing remedies keep the branch, the text and the reason

Before any remedy runs, a chain of gates resolves it to observe, propose or act, and names the gate that bound it: a kill switch effective on the next cycle, the remedy's own setting, the target's exposure posture, the credential, and the change class. Each remedy has an attempt cap under a global ceiling, and a change that exhausts its attempts moves to a quarantine lane with a sentence saying what is owed.

Consequences. Trust in the operator rests on structure, not on its reasoning. Human-triggered deploys run under a different identity, so the identity that decides which changes land can never change the rules those changes must pass.

10. Runtime and environment patterns

The same trust model applies when an application invokes a model while it runs. The application supplies the permitted context and capabilities, evaluates the candidate output, and decides what may be shown or executed. The model generates; fixtures decide what reaches a user. These patterns matter most where a model speaks directly to people about their own information, and where a wrong answer has consequences. The last pattern extends the same discipline to the environment the software runs in: the external services it depends on but does not own.

Pattern 28. The Deterministic Sandwich

Problem. A model asked to be safe is safe most of the time. A second model asked to check the first is also right most of the time, and the two can fail together.

Forces. Pattern checks are reliable but shallow; model judges are deep but probabilistic. Neither alone is enough.

Pattern. Deterministic layers surround the models, so no model is ever the floor.

1. **Inbound screen**, deterministic: reads every incoming message for conditions that must never wait on a model, such as an emergency or a request that must go to a person, and routes them directly.
2. **Identifier scrub**, deterministic: removes identifying details from free text before it reaches a model or storage.
3. **Responder model:** answers only from the current user's own data, in a structured output.
4. **Claim guard**, deterministic: fixed patterns for statements the application must never make, held identical everywhere they run.
5. **Field verifier**, deterministic: extracts every factual value in the candidate output and checks each against the source record.
6. **Judge model:** a different model, configured independently, given an enumerated list of prohibited categories and what is explicitly allowed, never an open question.

A block at any layer, or a layer that cannot return a verdict, withholds the output.

Consequences. Each layer has a distinct detection role. The deterministic floor survives a model outage, and the judge recovers what patterns cannot reach. Each layer is measured standalone on a labeled corpus and the composed path is measured end to end, because a strong local score does not show the whole path works and a layer's miss does not show the system failed.

Pattern 29. Refuse to Start Misconfigured

Problem. Most safety properties are configuration, and configuration can be dropped, defaulted or silently replaced by a fallback.

Forces. Services are usually built to start whatever happens and log a warning, and warnings are not read.

Pattern. When a service is told it is handling real data, a startup guard refuses every request unless every safety-relevant condition holds at once: identity verification, the approved model boundary, a durable tamper-evident audit sink, durable stores for anything a refusal produces, every expected migration applied, an explicit origin allowlist. The guard checks resolved reality, not stated intent; a store that silently fell back to memory cannot pass as configured.

Consequences. A misconfiguration becomes a service that does not come up, which the deploy's health check then rolls back. A safety property cannot be lost by omission.

Pattern 30. Authorization by Absence

Problem. Every permission check is code that can be wrong, and a model that produces persuasive text can sometimes talk its way past logic that depends on interpretation.

Forces. Separate deployments and fine-grained grants add operational overhead.

Pattern. Prevent actions by not providing the capability at all, rather than by checking permission. Routes that should not be reachable from a surface are not mounted on it, so they answer "not found" even to an authenticated caller. Privileges that should not be used are not granted: records of truth and audit trails carry no update or delete grant for any application role, and that absence is verified against the live system. Services are split so that the public surface holds no sensitive data at rest, the data service is private and callable only by the surface's identity, and administrative functions run separately behind named-user access. Every read is scoped to the verified user, and isolation is enforced in the database itself.

Consequences. There is no permission check to get wrong, and no model output can obtain a privilege that does not exist.

Pattern 31. Fail Closed, Keep Serving the User

Problem. Failing closed protects users from bad output but can abandon them, and a system tuned only to block will block ordinary requests too.

Forces. The cost of a harmful answer and the cost of an unhelpful one are both real, and they are not symmetric.

Pattern. Anything the layers will not clear goes to a durable human review queue, and the user immediately receives an honest deferral that tells them where help comes from. A judge that errors, times out or returns something unparseable counts as a block. The bias is deliberate and stated: an over-escalation costs one unnecessary human contact, while the opposite error has no comparable price. Usefulness is measured alongside safety: queue age, capacity, handoff completion, repeated deferrals and ordinary questions answered are operating measures, not

afterthoughts. Where wording must be tightly controlled, verified fields populate approved templates. For applications that take actions rather than converse, actions are typed, carry source references, and are authorized deterministically immediately before execution.

Consequences. The user is never left with a silent failure or a fabricated answer. The system shows how often it errs toward caution, so that bias can be tuned rather than guessed at.

In regulated, safety-critical and security-sensitive applications, over-deferral is the default position. Caution comes first with AI in those settings. As the system validates, the standard moves: set accuracy expectations comparable to the human level of support already accepted in that domain, and innovate from there. The question is never whether the AI is perfect. It never will be, and it does not need to be, because the fixtures and the measurements are what make it safe. The question is whether it does better than the way the work is done today.

Pattern 32. Held Out Means Held Out

Problem. An evaluation set read during tuning stops measuring anything unseen, and generators iterating against a visible test learn the test.

Forces. Every look at a failing case is useful for fixing it, and every look contaminates the set.

Pattern. Evaluation sets meant to measure unseen behavior are sealed in storage readable only by a dedicated measurement identity, pinned by hash, reported only in aggregate, and checked for contamination. A case is retired the moment a rule is written to match it. Any set that was read during tuning is reclassified rather than quietly kept. Access records, evaluation budgets and the lineage of replacement cases are part of the design, because aggregate reporting reduces exposure but does not make repeated tuning against an evaluation free.

Consequences. Measured safety stays a measurement of generalization rather than of memorization. Dashboards count held-out cases but never show them.

Pattern 33. Twin What You Don't Own

Problem. Software depends on services it does not control: identity providers, payment processors, messaging, document stores, partner APIs. Testing against the live services is rate-limited, costly and unsafe for the failure modes that matter most. Mocks are usually written by the same process that wrote the code, so they agree with it and miss how the real service behaves.

Forces. A faithful clone of a substantial external service was always possible and almost never economical, so teams settled for thin mocks. A clone can also drift from the service it imitates, and a twin that lies is worse than none.

Pattern. Build a behavioral twin of each external dependency, and calibrate it to the service as built, not as documented. Documentation describes the nominal service; the live service, with its quirks, undocumented edge cases, latencies and failure behavior, is the as-built one, and a

twin built from documentation alone carries the false confidence of a simulation of a machine that does not exist. Calibration runs in a loop:

1. **Probe the real service.** Record real requests and responses, including errors, limits and edge cases, as measured points.
2. **Fit the twin to the measurements.** Correct the twin wherever its behavior departs from what was observed, until it reproduces the recorded behavior within tolerance.
3. **Build and test against the calibrated twin.** Journeys, adversarial runs and nightly break-it campaigns execute against it at volumes and rates no live service would allow, including outages, throttling, malformed responses, revoked credentials and partial failures too dangerous to trigger for real.
4. **Ship, and keep probing.** Real behavior is re-measured on a schedule; drift beyond tolerance fails the twin until it is recalibrated.

Twins are fixtures and follow the fixture rules. They are owned and locked apart from the code they judge, their recorded measurements are their datum, and they live in the parts catalog, so every application that integrates with a service inherits its calibrated twin.

Consequences. Integrations become as testable as internal code, and intent journeys can cover the whole activity even where it crosses into systems you do not own. Cheap generation is what makes this practical: building a high-fidelity twin is now an ordinary task for the factory rather than a project nobody would approve. Twin fidelity becomes another ceiling, and it is measured and raised the same way as fixture fidelity.

11. Authority and the control plane

The inversion depends on fixtures the work cannot weaken. An autonomous system therefore needs a clear account of who may propose, evaluate, authorize and execute, and a record that shows each did only that.

11.1 Separation of authority

PARTICIPANT	NORMAL AUTHORITY	CHANGE THAT NEEDS A DISTINCT PATH
Application author	Propose code, tests, configuration and evidence	Weakening the acceptance policy that governs its own candidate
Review and verification instruments	Produce findings and bounded verdicts	Changing their own required scope or exemptions
Admission mechanism	Admit a candidate that satisfies applicable policy	Replacing the policy or accepting an exception
Deployment executor	Deploy an authorized release inside its environment	Enlarging credentials, exposure or permitted side effects

PARTICIPANT	NORMAL AUTHORITY	CHANGE THAT NEEDS A DISTINCT PATH
Pipeline operator	Apply declared remedies within scope and caps	Granting itself new permissions or bypassing required checks
Accountable human owner	Define goals, operating limits and accepted risks	None: these decisions remain attributable and recorded

This is a separation of authority, not a requirement for a person to approve every routine change. An automated policy transition can itself be governed by existing policy. The essential condition is that **a candidate cannot quietly make its own admission easier and then use the weakened verdict to proceed.**

The rule for what proceeds on its own is simple:

- **Proceeds automatically:** a change that is deterministic, fully audited, and can be reverted in a single step.
- **Needs sign-off:** a change that takes multiple steps to reverse, or is destructive.
- **Needs sign-off:** any change to the AI itself, and any request by an agent or tool for escalated permissions, wider scope or a new ability.

The last line is the one that protects the factory. If an agent decided to stop running tests, that breaks the protocol, and the only way around the protocol is to ask a person. A change to the factory gets human eyes so that someone confirms it is the right change, not just a permitted one.

11.2 The control plane

The gates, thresholds, rulesets, journeys, datums, rubrics and exemption policies together form the **control plane**. A change that lowers a threshold, loosens a journey's budget, widens an exemption, moves a datum, resets a ratchet or shifts an architectural boundary travels its own attributable path. It is evaluated under the prior policy, challenged with known defects, replayed against recorded past failures, and activated as an identifiable policy release.

The control plane changes at a small fraction of the rate of application code, and it is where human ownership matters most. Specific separations make it hold: a change to intent cannot also change the code that intent judges; reviewers are pinned to the base branch; a gate's proof of sensitivity expires when the gate changes; the identity that merges cannot administer the rules; and live rules are compared against their committed expectation.

11.3 Exceptions and new gates

An exception names the finding, the scope it applies to, the reason, the issuer, the duration and the required follow-up. Recording an exception makes it inspectable; authorization decides whether it is permitted. Neither a hash nor a model's persuasive explanation grants that authority.

The assurance system is software and must evolve. A new gate can first run in observation mode against historical and current candidates, exposing its false positives and cost, and become required once its behavior is understood.

11.4 The tamper-evident record

Consequential actions are written to an audit trail in a separate store, through a role that can only read and insert. Each entry is chained to the previous one by a keyed hash, a scheduled job running under its own read-only identity verifies the chain, and an external high-water mark closes the one gap a chain cannot close alone: truncation of the tail. Audit entries record identifiers, counts and categories, never sensitive free text, because an audit trail that quotes the content it protects recreates the exposure it exists to detect. Users are deactivated, not deleted, and retention is enforced by withheld delete grants rather than by policy.

Evidence lives in purpose-specific stores. Producers write create-only, versioned, build-stamped records, and the console that displays them reads and never writes, so it cannot manufacture its own results. A status is a word, not only a color. A score carries its depth. Not published is not zero, and unreadable is not empty. Rounding never erases a failing case.

11.5 Governing an estate

A factory that produces many applications must answer a question no single application faces: what exactly is the estate, and what must each member carry? The answer is a declared register, every repository in scope or out of scope with a written reason, because automation whose credentials could reach everything is exactly why the list must be declared rather than inferred. Every member carries a defined set of pipeline artifacts, and a sensor grades every member against that set and names what is missing along with how to provision it. Each application records its exposure tier and declares every external system it reads or writes, and a session using an undeclared source is flagged as it works.

---

12. The learning loops

Two loops run through a ZTHR system. The generation loop turns intent into an accepted candidate. The improvement loop turns every escape, stall and recurring cost into a better fixture, so the next generation starts from a stronger environment.

12.1 The bounded generation loop

The generator begins from a governed foundation and an explicit, protected goal. It produces a candidate, fixtures evaluate it, and diagnostics say what remains unsatisfied. The next attempt may repair the candidate or start again from a clean base. Progression follows evidence that applies to the current candidate and current policy. An exhausted budget produces an explicit unresolved state, never indefinite retries.

Four properties make it work.

- **Diagnostics are fuel.** A useful diagnostic names the violated expectation, the observed behavior, the relevant artifact and the next evidence needed. A fixture that prints only "failed" leaves the loop blind, so designing a fixture includes designing its failure message for a model reader.
- **Generations, not patches.** Restarting is cheap, so a later attempt may start fresh rather than layering fixes onto a failing one. A restart keeps the acceptance contract and what earlier attempts learned.
- **Non-convergence is diagnosed, not assumed.** A coherent task with valid fixtures may still exceed a model's ability or its budget. The loop distinguishes product defects, unavailable dependencies, ambiguous requirements, fixture defects and resource limits before choosing a remedy: a new implementation, clearer intent, a different part, a corrected fixture or a more capable generator.
- **Several timescales.** The loop runs within a working session, per proposed change, per operator cycle, and on a slower nightly cadence. Each catches what the faster ones cannot.

The loop records attempts, compute, elapsed time, repeated failure classes and the age of unresolved work, so convergence is a measurement rather than an impression.

12.2 The improvement loop

A lesson becomes a shared improvement only through the same discipline as any other change.

1. **Observe.** An escape, a stall or a recurring cost is recorded with the affected requirement, the evidence, and the control that was absent, ineffective or too expensive.
2. **Propose.** The fix may change an application, a fixture, a foundation part, an operator remedy or the requirement itself. These are different changes, and each follows the authority appropriate to it (Section 11).
3. **Validate.** A reusable control must be shown to catch the intended defect and to let representative valid behavior pass, tested in a consumer context.
4. **Release.** A governed release delivers it to each application through that application's own checks.
5. **Adopt.** Adoption is recorded, not inferred from publication.
6. **Measure.** The loop closes only when observation shows the intended effect: the defect stops recurring, the stall clears, false blocks stay acceptable, and consumers actually use the corrected part.

Publication is not adoption, and adoption is not yet demonstrated improvement. An improvement is a hypothesis until observation supports it.

12.3 Finding what nobody considered

Named gates catch named defects. Three mechanisms go after the rest, and together they make the system ratchet its own fixtures and measurements upward.

- **Mutation testing and ratchets** plant defects nobody wrote a test for and hold the line once the suite catches them (Patterns 10, 11).
- **Nightly attempts to break things:** injected bad SQL, malformed and hostile API calls, broken inputs and adversarial users, run against the composed system and the twins of everything it depends on (Pattern 33).
- **Adversarial review with newer and newer models.** Each new model generation re-reviews the platform, and its findings act as a recurring test of the whole factory. The platform is re-examined by a reader smarter than the one that built it, again and again.

Every finding from these runs is converted into a fixture, so the next build faces a harder exam than the last.

12.4 Machinery that becomes a part

Quality machinery built inside one demanding application, such as the journey runner, the adversarial harness and the evidence readers, is extracted into catalog parts. What was specific to that application is lifted into configuration each adopter supplies: its journeys, its oracle, its sign-in recipes, its declared navigation. The factory reproduces assurance along with functionality.

12.5 Improving as often as is viable

The operator's inner loop senses, proposes, acts and verifies, and periodically names its own stall classes with the remedy owed. An outer loop can let it draft changes to its own machinery, but drafting confers no authority. A drafted change must be proven by replay against recorded cases, pass its fixtures and a fresh review, be activated through a controlled release, and then show that the targeted stall class actually declines. The model writes a proposal, never a commit, and never one that enlarges its own permissions. People decide which classes of policy change may be automated at all.

Self-improvement should run as often as is viable. When the loop produces the intended outcomes and passes its fixtures, there is no reason to slow it down. This is why ZTHR favors small units of change over large lifts: each unit can be understood completely, validated against its fixtures, and reverted in one step, so frequent improvement stays safe.

---

13. Following a failure through the whole

The patterns are easiest to evaluate by following a failure through them. For each candidate failure the question is: what would have to go wrong, at how many independent layers, for it to become an uncontained outcome? The examples illustrate the design; they are not a record of incidents.

CANDIDATE FAILURE	PATTERNS THAT PREVENT OR DETECT IT	BOUNDED OUTCOME
A new endpoint reads another user's data	Governed Foundation (1); context-aided review (21); a negative test using another user's identity; per-user scoping and database isolation (30)	Rejected wherever those controls apply; journeys test the composed boundary
A reviewer approves a wrong business rule	Domain contract and protected journey goal (15, 18); oracle from the record (17); multi-family review (21)	A later gate rejects it; an escape becomes a new fixture
A scanner runs against an empty path	Scan-depth floor; three-valued verdicts (9)	Missing coverage is an explicit failure, not a clean report
A gate is edited so it no longer catches its defect	Proof pinned to the gate's hash; harness case floor (8)	The build fails until the gate is shown to go red again
A dependency reintroduces a forbidden capability	Source checks; served-bundle inspection (23)	The release fails when the capability is observed
A merged change never reaches service	Release record; serving-equals-built; reconciliation every operator cycle (23, 24)	The gap stays visible and eligible for a bounded remedy
A refactor hides a button behind another element	Walking by accessible name; two-representation screen check; action budget (16)	The journey fails with the step, the expected state and the observed screen
A runtime answer contains an unsupported claim	The deterministic sandwich (28); end-to-end adversarial cases	An honest deferral and a durable handoff; the case enters the evaluation set
A slow series of changes degrades a key activity	Datums with tolerance zones (12); intent as primary datum (15)	The gate fails at the tolerance boundary, however small the last step
The operator is interrupted mid-write	Claim-once proposals; outcome-unknown state; later verification (25)	No blind retry; the ambiguous action stays visible until resolved

Two observations follow from tracing failures this way.

First, the strongest protection comes from controls that read different representations of the system (Pattern 22). A failure that slips past every reader of source code can still be caught by a reader of the database, the built artifact, the rendered screen or the running service.

Second, some failures stay correlated because the shared requirement is wrong. A domain expert may specify the wrong workflow, and every implementation check will faithfully enforce it. Domain acceptance, contrasting examples, actual use and outcome measures

address that class, and each has a route into durable contracts. The practical review question is therefore whether any important failure path remains uncovered or inadequately bounded, not whether any single component guarantees the whole.

---

14. What changes for people

ZTHR does not remove people. It moves their expertise from reading diffs to the places where judgment compounds: defining intent, building and governing fixtures, owning the control plane, and judging consequential risk.

14.1 From domain knowledge to working software

The intended builder is a person who understands the work an application should support. That person works with an AI development tool to express the workflow, inspect behavior and refine the result, starting from the governed foundation and parts catalog. The first conversation establishes the users, the information involved, the consequential actions and the observable conditions of success, and those decisions become configuration, contracts and journeys.

- "A contractor may edit only their own sites" becomes an authorization boundary and a negative test that uses another contractor's identity.
- "Publishing changes the public site" identifies a consequential action and the evidence required to release it.
- "A customer requesting a call-back must see it acknowledged, and staff must see the request" becomes a journey that checks both sides.

This is how domain judgment becomes persistent engineering knowledge. Once encoded, it travels with every later change. The builder does not restate it to every agent, and the next session does not infer it from a long conversation.

14.2 Catching a bad specification before it is built

Because the factory builds what it is told, the specification gets its own review before any code is generated.

1. **Map the specification to the commons.** The generator reviews how each requirement maps to existing parts and solutions in the library, and picks from those assets rather than coding from scratch. Gaps are named explicitly.
2. **Review it from first principles.** A separate pass asks whether the specification actually achieves the stated outcome, independent of how the existing parts happen to work.
3. **Let the parts fill in what the specification forgot.** Each part already knows its integrations, its security requirements, its interface conventions, its documentation, its training material, its API surface, its tests and its mutation floor. A specification assembled from

parts cannot quietly omit those basics, because they arrive with the parts and are gated with them.

The result is a complete, consistent package: the application, its documentation, training, API, tests and mutation tests, all integrated, all gated, and maintained by the same loops that maintain the code.

14.3 What remains for engineers

The goal is minimal repeated hardening, with engineering depth matched to the application. New data integrations, changes in exposure, privileged actions, unusual performance needs and unfamiliar failure modes still require engineering judgment, and the factory should flag those departures explicitly. Work that recurs across applications moves into the foundation, a reusable part or an executable acceptance rule. The engineer's leverage shifts from the code they write to the fixtures they build, because a fixture serves every application that inherits it.

14.4 Review as a contribution to be measured

Human review can add domain insight and judgment, but its presence does not establish those outcomes on any given change. Model review likewise needs a defined task, appropriate evidence and an observed contribution. The relevant comparison is between complete delivery systems: escaped defects, delay, recovery, cost and human effort. ZTHR's position does not depend on ranking people against models. It requires every recurring approval step to justify its contribution, and useful judgment to be converted into reusable controls wherever possible.

14.5 The division of labor

People govern purpose and consequential boundaries. Agents propose and execute bounded work. Mechanisms enforce the rules. Observations decide whether the result actually happened. As more repeated engineering is captured in the shared foundation and catalog, the system becomes progressively easier to build with.

---

15. Adopting ZTHR in stages

An organization can adopt ZTHR incrementally. The first requirement is a governed path that actually produces and operates one useful application. The strongest next additions are the ones that remove recurring work or close a failure path already visible in that application.

1. **Establish the construction path.** A foundation with identity, access, audit, deployment and required checks suited to the application class (Pattern 1). A catalog and a planning interface for agents (Patterns 2, 5). A starting template verified in a real consumer context (Pattern 3). Make the governed path easy enough that builders take it naturally, and enforce the boundaries that cannot be optional (Pattern 7).
2. **Make acceptance executable and observable.** Express domain expectations as contracts, journeys and forbidden outcomes (Patterns 15–18). Review every change with at least three

model families, adding a fourth after merge where it can be afforded (Pattern 21). Prove the important gates can fail (Pattern 8). Twin the external services the application depends on (Pattern 33). Deploy by identity and inspect what serves (Pattern 23). Keep unknown and incomplete outcomes visible (Pattern 9).

3. **Operate continuously.** Add reconciliation for stalled changes, missed events, abandoned sessions and serving drift (Patterns 24–26). Constrain the operator by structure (Pattern 27). Give people a short decision surface instead of a backlog automation could clear.
4. **Generalize what is learned.** Promote reusable controls into the foundation or catalog, deliver them through each application's own checks, and measure their effect (Section 12). Introduce datums for the properties that matter most (Pattern 12), and keep a path for the assurance system to improve without authorizing itself (Section 11).

For applications in which a model speaks to people or takes actions, the runtime patterns (28–33) arrive with the first such feature, not after it.

Formal methods earn their cost where a property and its state space can be stated precisely: claiming work exactly once, lease ownership, allowed state transitions, role exclusivity. A proof obligation must state its assumptions and bounds, and a timeout or an unknown solver result does not satisfy a required proof. Formal sophistication is valuable when aimed at a specific property; it is not a prerequisite for a first deployment.

16. How to evaluate it

The headline measure is **intent completion**: how often the software does what it is supposed to do for the people using it. That moves evaluation from counting what went wrong to measuring what goes right, which is a more direct measure of success. Volume shows the method runs; intent completion shows it works.

Measure the incumbent first. Compare the AI-built process with the current human process on equal terms: revision rate per change, rework, time to usable software and intent completion, normalized for volume. AI can generate more rework simply by generating more work; the fair question is whether its rate is better, not whether its count is larger.

QUESTION	MEASURE	WHAT IT TESTS
Does the software do what it is supposed to do?	Intent completion rate per journey, per release, against its datum	The outcome users came for; the primary measure
Is it better than how the work is done today?	The same measures taken on the incumbent process, normalized for volume	Whether the change is an improvement, not whether it is perfect
Does generation converge?	Median and tail attempts to acceptance; compute; quarantine rate	The generation loop, and where it fails to converge

QUESTION	MEASURE	WHAT IT TESTS
How fast are problems found and fixed?	Time to detect; time to restore; exposure before the fix	The real cost of a defect now that fixes are cheap
Does the system learn?	Recurrence of known defect classes; findings converted into fixtures; adoption of shared fixes	Whether fixture fidelity is rising
How much engineering remains per application?	Human hours on setup, hardening, exceptions and recovery	Whether the factory removes repeated work
Which reviewers earn their place?	Unique validated findings per reviewer and family; false-block rate	Contribution, without assuming independence
Is automation creating another queue?	Age of stalled work and handoffs; intervention rate; retry exhaustion	Hidden dependence on human attention
How far has the system drifted?	Distance from each declared datum; how many datum moves and ratchet resets were deliberate	Whether change is chosen or accumulated

Escapes still matter for one class of defect: those whose harm lands before any fix can, such as exposed data or an irreversible action. For everything else, fixes are fast and cheap, and time to detect and exposure before the fix say more than the escape count does.

Measurements need denominators and stable scope. A defect count without users, releases or exposure misleads, and a reviewer that reports the same issue four times has contributed one finding. The best evidence is prospective: for each new application, record the requested capabilities, what was inherited from the platform, elapsed time, human effort, the engineering changes that remained, and operating results.

The same measures should drive policy. A control with high cost and little unique benefit is improved or retired through the governed path. A recurring escape produces a concrete acceptance or containment improvement. Evaluation can stay privacy-preserving: outside reviewers can assess anonymized failure classes, aggregate outcomes, state diagrams and evidence formats without access to source code, user data or credentials. The target is useful assurance at delivery speed, not an ever-growing checklist.

16.1 Economics

ZTHR spends compute to save attention. Generation, review and verification all cost money on every change, and the case for the approach is that the total is far below the fully loaded cost of the human process it replaces, at equal or better intent completion.

The unit of cost is the accepted change, and above it the delivered application: compute and model usage, plus every human hour spent on setup, exceptions, sign-offs and recovery. Compare it with the incumbent's cost for the same unit, including review time, rework and the cost of defects that reach users.

WHERE COST GOES	HOW IT BEHAVES	HOW TO KEEP IT PROPORTIONATE
Generation	Cheap per attempt, many attempts per change	Strong jigs and diagnostics shorten the path to acceptance
Multi-family review	Three or four model calls per change	Match model cost to role; a low-cost family earns its seat when its findings are unique
Per-change verification	Gates, journeys, falsification proofs	Run the deterministic rungs first; stop early on a red result
Scheduled verification	Mutation runs, nightly break-it campaigns, re-review by newer models	Set cadence by risk; run the expensive work off the critical path
Twins and fixtures	Built once, maintained continuously	Amortized across every application that inherits them
Persistent operator	Small, continuous	Bounded remedies and caps keep it from becoming a cost of its own

Where the savings come from. The review queue and the attention it consumes disappear. Repeated hardening per application moves into the foundation. Rework falls because defects are caught before a person ever sees them. Above all, fixtures compound: each one is paid for once and protects every later change, which is why one person or a small team can operate an estate that would otherwise need many more people.

The levers are the patterns themselves. The determinism ladder (Pattern 19) puts the cheapest reliable instrument first. Small units make every failed attempt cheap. Authorship tiers (Pattern 6) reduce how much must be generated and verified at all. Every control is measured for its unique contribution, and one whose cost exceeds its value is improved or retired through the governed path.

16.2 Operating evidence

ZTHR is in production use for the development of greenfield systems in a regulated industry, where it has proven highly effective. To compare implementations on equal terms without exposing proprietary operating baselines, adopters should benchmark and report across six standard dimensions: intent completion rate; revision rate per change against the incumbent process; time from agreed workflow to usable application; time to detect and restore; human hours per application; and total compute-plus-human cost per accepted change.

17. Lineage

ZTHR is new in its composition and in its subject, an AI author. Its parts continue established traditions, and naming them gives skeptical readers a frame they already respect.

Interchangeable parts and gauging. The Fixture Inversion is the software counterpart of the American system of manufactures, in which jigs, fixtures and gauges let operators of ordinary skill produce interchangeable parts [1]. ZTHR's jigs are the foundation, templates and parts catalog. Its gauges are the verification spine and the journeys.

Datums and functional gauging. Geometric dimensioning and tolerancing measures features from declared datums rather than from each other, and verifies assemblies with functional gauges [2]. ZTHR applies both: drift is measured from datums (Pattern 12), and user intent is the functional gauge on the assembled product (Pattern 15).

Quality built into the process. Deming argued that organizations should stop depending on inspection to achieve quality and build it into the process instead [3]. Human code review is inspection. ZTHR moves the quality mechanism into the process that produces the change, and its measures of convergence, escape and recurrence apply statistical thinking to the process itself.

High-reliability organizations. Weick and Sutcliffe identified five principles that let organizations operate reliably under hazard [4]. ZTHR implements each in mechanism rather than culture.

HRO PRINCIPLE	ZTHR MECHANISM
Preoccupation with failure	No gate trusted until shown to fail; every escape becomes a fixture
Reluctance to simplify	Three-valued verdicts; unknown as a first-class state; standalone measurement per safety layer
Sensitivity to operations	A persistent operator on a short cycle; serving compared with merged on every cycle
Commitment to resilience	Fail-closed products; deploy by identity with rollback; services that refuse to start misconfigured
Deference to expertise	People own intent, the control plane and risk; domain experts review withheld outputs

Zero trust. Symmetric Zero Trust borrows its premise from zero-trust network architecture, which removes implicit trust based on network location [5], and applies it to models, checks, operators and documents.

Mutation testing and falsification. Measuring a test suite by whether it detects deliberately planted faults dates to DeMillo, Lipton and Sayward [6]. ZTHR extends the planted-fault idea upward to the gates themselves. The broader stance, that a claim is worth something only if it could be shown false, is Popper's criterion of falsifiability [7] applied to engineering controls.

Continuous delivery and site reliability engineering. From continuous delivery ZTHR inherits small changes, automated pipelines, immutable artifacts and rollback; from site reliability engineering, the treatment of operations as software. It departs from both by removing the required human approval most continuous-delivery practice retains, and by treating every check as an instrument that must itself be calibrated.

Digital twins and calibration. Offline robot programming has long depended on simulations calibrated to the real cell as built, not as drawn. ZTHR applies the same loop to the external services software depends on (Pattern 33), and holds the twins to the same fixture discipline as everything else.

Origins. The framework grew out of the author's practice across several fields: using 3D models to drive manufacturing robots from the late 1990s, programming robot cells and verifying parts to tolerance in precision aerospace manufacturing, building self-correcting systems in games and software-defined manufacturing, and decades of leading software teams and running large application portfolios. Jigs, gauges and datums are not metaphors borrowed for software here; they are the working tools the approach was built from.

References

1. Hounshell, D. A. *From the American System to Mass Production, 1800–1932.* Johns Hopkins University Press, 1984.
2. ASME Y14.5. *Dimensioning and Tolerancing.* American Society of Mechanical Engineers.
3. Deming, W. E. *Out of the Crisis.* MIT Press, 1986.
4. Weick, K. E., and Sutcliffe, K. M. *Managing the Unexpected.* Jossey-Bass, 2001.
5. Rose, S., Borchert, O., Mitchell, S., and Connelly, S. *Zero Trust Architecture.* NIST Special Publication 800-207, 2020.
6. DeMillo, R. A., Lipton, R. J., and Sayward, F. G. "Hints on Test Data Selection: Help for the Practicing Programmer." *IEEE Computer* 11(4), 1978.
7. Popper, K. *The Logic of Scientific Discovery.* Hutchinson, 1959.

---

18. What ZTHR does not claim

A framework built to refuse unearned claims should state its own limits plainly.

The largest risk is stated first: **the system builds what it is told.** A bad specification or a bad process definition will be built and enforced faithfully. Section 14.2 is the answer to that risk, and specification review deserves the same care as any fixture.

- **Not that fixtured software is correct.** It is as correct as its fixtures are faithful. The claim is that fidelity is visible, measurable and improvable by a defined process.
- **Not that model diversity removes shared blind spots.** It reduces correlated error. Diversity of representation (Pattern 22) carries more of that weight.

- **Not that falsification proves completeness.** It proves each named gate catches its named defect. Unnamed failure classes are found by escapes, mutation, adversarial evaluation and post-merge review.
- **Not that deterministic layers are complete.** A pattern screen is a fast first line with later layers behind it, not a full answer on its own.
- **Not that fixtures force taste, design coherence or the right intent** (Section 3.7).
- **Not that people are removed.** They move from reading diffs to owning intent, fixtures, the control plane and risk.
- **Not that it is a compliance certification.** Mapping controls to a regulatory framework is useful evidence; it is not an audit.
- **Not that the factory's own growth is free.** Larger catalogs, more agents and more gates carry their own costs, and those costs are measured and governed like any other.
- **Not that volume is a quality measure.** Delivery volume shows the method runs. Section 16 names the measures that would show it works.

One residual risk is stated rather than explained away. **Enforced independent human review of each code change is not a control ZTHR relies on.** What bounds an unreviewed change instead is stated: required checks with no bypass, gates proven to fail, multi-family model review, protected intent, deploy by identity with rollback, serving verification, and fail-closed products. None of those is a second pair of human eyes. Whether that trade is right for a given domain is a judgment each adopter must make deliberately (Section 19).

---

19. Objections and open questions

19.1 Anticipated objections

These are the objections engineers and peers raise first, with where the patterns answer them.

OBJECTION	ANSWER
Drift tests only protect whatever bugs already exist	The primary datum is user intent in the user's words, owned by the product, not a snapshot of current behavior (Pattern 15)
In a fast-moving product, re-baselining becomes routine and meaningless	Implementation changes freely beneath an intent that holds. The datum moves only when user needs change, and that move is governed (Patterns 12, 18)
A classifier is wrong consistently, so its errors hide	Several model families with different context review the same outputs, and disagreement exposes systematic error. Retraining is a fixture change (Pattern 19)
"Tested exhaustively" is not a standard	Each authorship tier sets an explicit qualification standard, with today's human-built defect rate as the floor (Pattern 6)

OBJECTION	ANSWER
Real requirements do not all compose from parts	Generators fill gaps following the platform's conventions; gap code faces the same fixtures, and a new capability is written back as a part (Patterns 2, 6)
Parts in specification can assemble into a system that is not	Journeys are functional gauges on the assembly, and drift is measured from datums (Patterns 12, 15)
Users depend on behavior nobody wrote down	Intents are set with users and domain experts from each role's use cases, extended across model families, and every escape becomes a new intent (Pattern 15, Section 12)
If models help write the intents, the second opinion shares the first's blind spots	Intent is built by one model family and tested by another, and it is the thing the people building an application review in every round (Patterns 15, 21)
A model optimizing against an ambiguous fixture converges confidently on the wrong thing	Fixtures carry their purpose, are proven to bite, and are read by controls over several representations; non-convergence and suspicious convergence are both diagnosed (Section 3.5, Patterns 8, 22)
People will route around the factory when it is slow	The source host accepts changes only in the governed shape; there is no path that skips admission (Pattern 7)

19.2 Open questions

These are the questions on which outside review would most sharpen the approach.

1. **Correlated failure.** Which plausible failure could pass through every representation at once: source, database, built artifact, rendered screen and running service?
2. **Iteration pressure.** When should a generation loop restart from a clean base rather than keep iterating on a failing candidate?
3. **Specification quality.** Beyond mapping to the commons and first-principles review, what catches a wrong requirement earliest?
4. **Setting the bar.** How should a field set accuracy standards for AI-delivered work and support, measured against the human level already accepted there?
5. **Held-out evidence.** Sealed evaluation sets are in use; which long-run observations would best show that measured safety reflects generalization rather than tuning?

20. Conclusion

When generating code becomes nearly free, grafting AI authors onto a workflow paced by human review produces either a bottleneck or a rubber stamp. ZTHR takes a different path, built on two ideas.

Symmetric Zero Trust says how claims acquire authority: only through evidence and enforcement beyond the claimant's control, whether the claimant is a model, a check, an operator or a document. The Fixture Inversion says where engineering effort goes: into the constraints and instruments that shape every later candidate. The construction patterns are the jigs. The verification and intent patterns are the gauges, with user intent as the primary datum. The operational patterns make sure what passed is what serves, and keep the work moving. The runtime patterns apply the same discipline at the moment a model speaks to a person. The learning loops carry every lesson into the shared foundation, so each application starts stronger than the last.

The generator is allowed to be wrong during development. The environment must detect the failures it claims to detect, refuse unsupported progression, and keep a useful path to correction. The result is software as correct as its fixtures are faithful, in a system where that fidelity is visible, measurable, and rising with every escape it converts.

> ZTHR does not make AI-written software correct by trusting better models. It makes correctness a property of the environment: fixtures proven to bite, intent the author cannot rewrite, and a loop that turns every escape into a new fixture. The models iterate; the fixtures hold; the humans decide what the fixtures must demand.

Appendix A. Pattern index

#	PATTERN	FAMILY	IN ONE LINE
1	Governed Foundation	Construction	Every application starts with identity, audit, deploy and checks already built in
2	Parts Catalog with Purpose Notes	Construction	Assemble from qualified parts that say what they are for and what they are not
3	Test the Part Where It Lives	Construction	Test each part inside a real consumer, not in the catalog
4	Single Source, Generated Mirrors	Construction	One hand-authored place per fact; everything else generated and freshness-gated
5	Describe, Never Write	Construction	The factory plans changes; only the session's own admitted change writes them
6	Authorship Tiers by Consequence	Construction	The higher the stakes, the less the generator may author
7	Zero-Bypass Admission	Verification	Every check required, nobody exempt, live rules compared with expected
8	Prove the Gate Can Fail	Verification	Plant the defect; the gate must go red; the proof expires when the gate changes

#	PATTERN	FAMILY	IN ONE LINE
9	Unknown Is a State	Verification	Pass, fail or unknown, and unknown never becomes pass
10	Ratchet	Verification	Debt may stay; it may not grow; only a person resets a floor, with a logged reason
11	Mutation-Measured Tests	Verification	Tests must fail on planted defects, not merely execute code
12	Measure From the Datum	Verification	Drift is distance from a fixed reference, not from the last build
13	Register, Don't Suppress	Verification	Every unfixed finding has a reason and an expiry; stale entries fail
14	Documentation Derived From the System	Verification	Figures and claims in prose are checked against reality
15	Intent as the Primary Datum	Intent	Whole user activities are the functional gauge on the assembly
16	Walk Like a User	Intent	Accessible names, real navigation, two readings of every screen
17	Oracles From the Record	Intent	Expected values come from the data, never typed into the test
18	Locked Intent	Intent	A change cannot edit the goal it is judged by
19	The Determinism Ladder	Independence	Use the most deterministic tool that can solve the problem
20	Roles Defined by What They Are Denied	Independence	Each role's limits are structural, not instructed
21	Multi-Family Review	Independence	Three model families before commit, a fourth in place before promotion
22	Diversity of Representation	Independence	Read the system as source, database, artifact, screen and service
23	Done Means Serving	Operational	Deploy by identity and prove what is running is what passed
24	The Persistent Operator	Operational	Act unless destructive, user-facing or wide in blast radius; every hold names its loss
25	A Dispatch Is Not a Result	Operational	Verify outcomes later; never retry an ambiguous write blind
26	Leases	Operational	Heartbeat-backed claims keep sessions and operator from colliding

#	PATTERN	FAMILY	IN ONE LINE
27	Bounded by Structure	Operational	The operator's limits never depend on its judgment
28	The Deterministic Sandwich	Runtime and environment	Deterministic layers around the models; no model is the floor
29	Refuse to Start Misconfigured	Runtime and environment	A missing safety condition is a service that does not come up
30	Authorization by Absence	Runtime and environment	Do not mount it, do not grant it
31	Fail Closed, Keep Serving the User	Runtime and environment	Default to caution where stakes are high; judge against how the work is done today
32	Held Out Means Held Out	Runtime and environment	Sealed evaluation sets, retired the moment a rule matches them
33	Twin What You Don't Own	Runtime and environment	Behavioral clones of external services, held to fixture discipline

Appendix B. Glossary

TERM	MEANING
Authorization by absence	Preventing an action by not providing the capability at all, rather than by checking permission
Blast radius	How far an action reaches beyond its own target; actions affecting a shared part, the foundation or a rule every application inherits wait for a person
Calibration	Correcting a model or twin from measurements of the real, as-built system so it reproduces observed behavior within tolerance
Context-aided checker	A model that did not write a change, given its intent and context, tasked with breaking the claim that the change meets its goal
Control plane	The gates, thresholds, rules, journeys, datums and policies that govern what may merge and ship
Datum	A declared reference from which a property is measured, with a tolerance zone
Digital twin	A behavioral clone of an external service, calibrated to its real behavior and held to fixture discipline
Escape	A defect that passed every applicable fixture
Falsification	Planting the exact defect a gate exists to catch and requiring the gate to fail
Fixture	A governed mechanism that constrains a capability, evaluates a defined property, or controls progression on evidence

TERM	MEANING
Fixture fidelity	How closely the fixture set captures what correct means; the ceiling on quality
Freshness gate	A gate that fails when a generated artifact no longer matches what its source would produce
Gauge	A fixture that evaluates a result: tests, structural gates, journeys
Generation	One attempt by a generator to produce a candidate that satisfies the fixtures
Intent completion	How often the software lets its users complete the activities they came to do; the headline measure of success
Jig	A fixture that constrains what can be built: foundation, templates, parts
Journey	A product-owned, locked user activity with a goal, forbidden outcomes and an action budget
Lease	A heartbeat-backed claim on a repository and branch that stands the operator down while live
Multi-family review	Review of every change by at least three model families before commit, and a fourth in place before promotion
Outside audience	People beyond the team currently using a surface; actions that change what they see wait for a person
Ratchet	A recorded floor that may improve and must not worsen; only a person resets it, with a logged reason
Register	A record in which every unfixed finding carries a typed verdict, a reason and, for accepted risk, an expiry
Symmetric Zero Trust	No participant gains authority by asserting success; claims count only on evidence the claimant does not control
Unaided reviewer	A reviewer from a different model family given only the diff and a short context page