Executable Architecture Fitness Functions and Conformance Testing for an Agentic Software Platform

1. Research Objective

The advent of autonomous, LLM-powered software agents introduces a fundamental tension into systems engineering: the orchestration of highly probabilistic execution pathways within the strict, deterministic boundaries required for enterprise software. The research objective is to determine how an agentic software platform can automatically and continuously prove that implementation changes preserve its non-negotiable architectural properties. In traditional development, architectural erosion occurs gradually.[1] In an agentic development platform, where agents autonomously mutate state, generate code, and orchestrate workflows, implicit architectural goals can erode instantaneously.[2]

The system must proactively detect and reject regressions at the point of origin. This includes preventing physical model names from appearing in portable agent definitions, blocking direct provider SDK imports outside of isolated provider adapters, and ensuring that direct model-to-tool execution paths are intercepted and mediated. Furthermore, the orchestration platform must reject direct protected-branch writes, deny task completion lacking criterion-level evidence, and block context items lacking cryptographic provenance or trust classification. The objective extends to preventing cross-project memory access, terminating unbounded delegation or retry loops, and strictly isolating implementation agents from self-approving their own generated outputs. The ultimate goal is an executable governance framework that renders the architecture self-testing, provider-independent, and operationally secure without imposing unmanageable burdens on human operators.

2. Primary Research Question

How can an agentic software platform automatically prove that implementation changes preserve its architectural invariants, provider neutrality, protocol contracts, security boundaries, Git integrity, project isolation, recovery guarantees, context provenance, task-completion rules, and fleet-governance policies?

This central question necessitates an architecture governance model that transforms static Architecture Decision Records (ADRs) and security policies into automated, blocking CI/CD controls.[2] The solution must bridge the gap between static code analysis (to verify structural integrity) and dynamic runtime evaluation (to contain stochastic agent behaviors), providing a mathematically sound and empirically verifiable assurance model.

3. Foundational Hypotheses and Principle Validation

The design of the architecture assurance framework rests upon foundational hypotheses regarding system behavior, failure modes, and verification mechanisms. These hypotheses dictate the implementation of fitness functions—objective integrity assessments of architectural characteristics.[1]

The primary hypothesis is that architectural principles that are not automatically tested will inevitably drift.[1] System prompt instructions are structurally insufficient for enforcing non-negotiable architecture rules; LLMs are susceptible to prompt injection, hallucination, and jailbreaking.[4] Consequently, security-critical boundaries must fail closed at the execution layer, independent of agent comprehension. Agent self-reporting is inherently invalid as conformance evidence due to the established "false-DONE" phenomenon, requiring all tests to bind to exact, immutable code and configuration versions.

Furthermore, deterministic failures (e.g., JSON schema violations) and stochastic quality variations (e.g., poor tool selection) demand distinct testing methodologies.[4] Probabilistic tests must not block every pull request unless their cost and variance are strictly controlled, whereas safety invariant failures must trigger immediate pipeline termination.

To formalize these hypotheses, the following table details the core accepted principles, mapping their required attributes for implementation:

Architectural Principle	**Supporting Evidence**	**Limitations**	**Enforcement Layer**	**Failure Mode**	**Required Telemetry**	**Release-Stage Applicability**	**Invalidation Condition**
Executable Invariants	Untested architectures decay rapidly.[1]	High initial setup cost.	Static & Runtime	Fails closed on violation.	CI pipeline logs, drift metrics.	PR blocking, Release blocking.	Deprecation of the underlying ADR.
Prompt Insufficiency	Models bypass prompts via encodings.[4]	Requires external sandboxing.	Runtime & Security	Unauthorized access denial (403).	Access violation alerts.	Commit, PR, and Production.	Provable 100% LLM determinism (currently impossible).
Provide	Lock-in	Comple	Contrac	SDK	Adapter	PR	Strategi

r Neutrality	prevents multi-model orchestration.[6]	x adapter maintenance.	t & Schema	import rejection.	capability mismatches.	blocking.	c decision to single-source providers.
Strict Separation of Determinism	LLMs exhibit standard deviation in outputs.[5]	Increases test suite complexity.	Evaluation & Production	Regression beyond baseline variance.	False-DONE rates, baseline deviance.	Nightly, Pre-release.	Emergence of strictly deterministic reasoning models.
Immutable Evidence	Mutable state leads to irreproducible tests.	Storage overhead for artifacts.	Git & Software Delivery	Base commit mismatch.	Cryptographic hash verifications.	Commit, PR.	Transition to a fully non-versioned execution model.

4. Architecture Assurance Model

The architecture assurance model establishes an unbroken chain of cryptographic and logical traceability from human-readable intent to production telemetry. The target system must implement a layered assurance model structured as follows:
Architectural Principle → Normative Requirement → ADR or Policy → Fitness-Function Obligation → Executable Rule or Test → CI or Runtime Enforcement → Evidence Record → Release Gate → Production Observation
Every fitness function executed against the agentic development platform must be traceable to a specific normative source.[2] If a repository policy mandates that all context items retain provenance, an ADR must define the cryptographic wrapping of context payloads. This ADR generates a fitness-function obligation, which is implemented as a runtime invariant test. When an agent attempts to inject unverified context, the runtime enforcement blocks the operation, generates an evidence record of the failure, and alerts production observation dashboards. When an upstream requirement or ADR changes, the platform initiates an automated

test-impact analysis. The dependency graph identifies which components consume the modified rule, triggers a fitness-function review, updates the CI execution rules, flags the relevant conformance-pack for version bumping, and forces release recertification for all dependent artifacts.

5. Fitness-Function Taxonomy

To provide holistic coverage, the orchestration platform requires a multi-dimensional taxonomy of fitness functions.[1]

5.1 Static Repository Fitness Functions

Static functions inspect code and configuration without requiring the execution of the full system, providing millisecond-level feedback.[7] Utilizing Abstract Syntax Tree (AST) analysis, dependency graph analysis, and semantic code queries, these functions enforce structural constraints.[8] The platform utilizes tools conceptually similar to pytest-archon or dependency-cruiser to define forbidden dependencies.[9] For example, a rule asserting that the core domain must not import infrastructure logic is enforced via AST parsing that scans for import statements matching physical provider names (e.g., OpenAI, Anthropic) outside of designated adapter namespaces.[2] Static checks also identify unversioned schema detections, missing generated-binding checks, exposed secrets, and prohibited branch operations prior to execution.

5.2 Build and Schema Fitness Functions

These functions verify machine-readable contracts and ensure data transition integrity between decoupled agentic components. The verification encompasses strict JSON Schema and OpenAPI/AsyncAPI validation, cross-language serialization round-trips, and schema registry consistency. A critical capability is unknown-field handling and extension namespace validation; if a new model provider returns an undocumented JSON payload field, the schema fitness function must detect the mutation and either normalize it or fail closed to prevent provider-specific semantics from leaking into the canonical state.[6]

5.3 Component Contract Tests

Component contract tests validate individual platform components—such as provider adapters, workflow ports, memory backends, and sandbox providers—as isolated black boxes.[4] These tests verify accepted and rejected inputs, canonical output formatting, error normalization, and trace propagation. For instance, testing an LLM provider adapter involves verifying that rate-limit errors (HTTP 429) from diverse external APIs are uniformly mapped to a canonical PlatformRateLimitError, ensuring the orchestration engine's retry logic remains provider-agnostic.

5.4 Runtime Invariant Tests

Runtime tests verify behaviors that only manifest during dynamic execution. These act as the

final defensive boundary against stochastic agent actions. Critical invariants include verifying that task completion is accompanied by typed cryptographic evidence rather than self-reported strings. Runtime monitors ensure that stale fencing tokens cannot mutate authoritative state, cross-project memory retrievals fail instantaneously, and external side effects strictly require a paired ToolIntent and ToolReceipt.

5.5 Git and Software-Delivery Fitness Functions

Given that agentic platforms autonomously generate and mutate code, Git integrity is paramount. These functions enforce that a single writable task maps exclusively to one isolated worktree and branch. They verify that the primary branch cannot be written directly by an agent, that the worktree base commit perfectly matches the task record, and that changed files remain strictly within an approved Change Manifest. Furthermore, generated test evidence must bind to immutable Git identifiers, ensuring that subsequent commits invalidate prior human or agent reviews.

5.6 Security Conformance Tests

Security boundaries must assume the agent is a hostile actor.[4] Fitness functions in this category execute penetration-testing paradigms against the platform's execution layer. They verify shell injection prevention, prompt-injection containment, and secret redaction. Crucially, they enforce isolation logic: sandbox-host isolation is probed via adversarial escape attempts, cross-project artefact protection is verified by simulating compromised authentication tokens, and self-approval prevention is structurally enforced to prevent an implementation agent from synthesizing its own human-approval receipts.

5.7 Reliability and Chaos Fitness Functions

Agentic systems coordinate long-running workflows across distributed state machines.[11] Chaos fitness functions inject faults to prove recovery guarantees. Targeted injections include worker crashes, provider API timeouts, malformed responses, message duplication, and corrupted checkpoints. The tests verify that task budget exhaustion gracefully halts further model calls, recursive delegation is terminated at maximum depth, and stale leases are deterministically reassigned without state corruption.

5.8 Probabilistic Conformance Tests

Separating deterministic code logic from probabilistic LLM outputs is a mandatory architectural pattern.[5] Probabilistic tests evaluate structured-output success rates, tool-selection accuracy across varied phrasing, handoff completeness, and false-DONE rates. These tests require repeated trials against versioned datasets to calculate confidence intervals and track baseline variance. Because they measure stochastic quality rather than deterministic safety, probabilistic tests are governed separately to prevent transient model variance from masquerading as a flaky infrastructure test.[4]

6. Required Testing Layers

The architecture assurance strategy dictates a layered testing pyramid to balance execution cost, determinism, and integration fidelity.

- **Layer 1 — Deterministic Code Tests:** Covers serializers, parsers, state transitions, routing, permissions, and cost accounting. These tests involve no model calls, execute in milliseconds, run on every commit, and act as hard-blocking gates.[12]
- **Layer 2 — Tool and Protocol Contract Tests:** Validates argument parsing, error handling, rate limits, and schema compatibility. Tests execute against both real transport adapters and deterministic fakes to ensure idempotency and side-effect receipt generation.[4]
- **Layer 3 — Agent-Component Tests:** Isolates specific agentic faculties such as the planner, memory retriever, handoff generator, and verifier. This layer measures singular capabilities independently of orchestration overhead.
- **Layer 4 — Workflow Integration Tests:** Tests the complex interactions between task state, context, tools, approvals, and checkpoints.[11] This layer utilizes deterministic model fakes for high-frequency CI execution and live models for nightly scheduled testing.
- **Layer 5 — End-to-End Task Tests:** Executes realistic scenarios from initial intake through final acceptance. Evaluates total system understanding, planning, context preservation, Git evidence generation, and correct stopping behaviors.
- **Layer 6 — Fleet and Orchestration Tests:** Validates multi-agent dynamics. Focuses on parallel delegation, conflict resolution, shared-state consistency, loop detection, and provider failover mechanisms.
- **Layer 7 — Production Monitoring and Online Evaluation:** Extends testing into production via sampled trace evaluation, architecture drift detection, shadow execution, and automated rollback rules. This layer detects emergent behaviors and regressions unrepresented in pre-release fixtures.

7. Architectural Rules to Formalise

The orchestration platform must codify several rule families into executable controls.

7.1 Provider-Independence Rules

The target system must aggressively defend its provider-neutral abstraction layers.[6] The CI pipeline automatically rejects physical model names appearing in portable agent or workflow definitions. Direct provider SDK imports are strictly forbidden outside approved adapter packages.[9] Provider-native response types must not traverse beyond adapter boundaries, and routing logic based solely on model nomenclature is blocked. Controlled provider-specific optimizations are legally permitted only within strictly namespaced provider extensions or capability profiles.

7.2 Tool Mediation Rules

To prevent arbitrary code execution and uncontrolled side effects, direct LLM-to-tool execution is prohibited. Every external tool invocation must be mediated by the platform.[4] The system automatically rejects tool execution lacking a verified ToolIntent schema. Missing policy

evaluations, unauthorized side effects, and high-risk executions lacking explicit cryptographic approval are blocked. Furthermore, tool outputs must never be treated as trusted instructions, mitigating indirect prompt injection vectors. Every executed action must yield a canonical ToolReceipt.

7.3 Git Integrity Rules

Autonomously generated code requires stringent cryptographic verification. The orchestration engine automatically rejects direct writes to protected branches. A writable task must be paired with an isolated worktree; any branch or base commit mismatch is a hard failure. The system enforces a strict ChangeManifest; modifications to files outside this approved scope are reverted. Reviews and test executions are irrevocably bound to exact commit hashes; subsequent commits instantly render prior approvals stale.

7.4 Task-Completion Rules

Given the propensity of agents to hallucinate task success, the platform mandates irrefutable evidence. Task completion is rejected if mandatory acceptance criteria lack corresponding results, or if criterion evidence is missing. Test evidence that fails to reference the exact target commit is discarded. Completions are blocked if required security reviews are absent, if blocking unresolved decisions remain, or if the completion assertion relies solely on an agent-generated summary without verifiable side-effect receipts.

7.5 Context and Memory Rules

Memory retrieval forms the operational context of the agent; polluted memory leads to catastrophic drift. Context items lacking cryptographic provenance or a defined scope are rejected. Untrusted content ingested without a formal trust classification is purged. The system fails closed on any cross-project memory retrieval attempts.[4] Unverified agent outputs cannot be promoted directly to authoritative memory without supersession support, and any context manifest that cannot be deterministically reproduced is marked invalid.

7.6 Authority and Separation Rules

Segregation of duties applies equally to human and machine identities. The platform structurally prevents self-approved implementations and self-approved security findings. Agent-generated human approval tokens are rejected via cryptographic signing requirements. Expired or reused approval tokens are blocked. High-risk tasks cannot proceed without an independent reviewer, ensuring an implementation agent never acts as its own final release approver.

7.7 Delegation and Retry Rules

To prevent resource exhaustion and infinite loops, agent autonomy is strictly bounded.[11] The orchestration platform terminates execution when delegation depth exceeds defined fleet policy. Recursive delegation loops and provider fallback loops are detected and killed. The system halts identical repeated tool calls that demonstrate no state progression. Model switching utilized to bypass capability permissions is blocked, and task attempts exceeding iteration, cost,

or time budgets are forcefully cancelled.

8. Rule Specification Format

Every architecture rule is formalized into a machine-readable Fitness Function Record (FFR). This enables policy-as-code enforcement and automated CI integration.

Field	Description	Example Value
fitness_function_id	Unique identifier for the rule.	FF-ARCH-PROV-001
title	Human-readable name.	Provider SDK Isolation
description	Detailed intent of the invariant.	Prevents provider SDK imports outside adapters.
category	Functional grouping.	Static Repository Check
normative_source_ids	Upstream requirement traceability.	ADR-012, SEC-POL-04
architectural_invariant	The formal property being proven.	Provider Neutrality
scope	Affected codebase or components.	Core Domain, Workflow Engine
applicable_components	Specific bounded contexts.	AgentPlanner, Orchestrator
enforcement_layer	CI or execution stage.	STATIC
implementation_type	Mechanism of verification.	AST Analysis
blocking_level	Severity of a failure.	COMMIT_BLOCKING
severity	Impact categorization.	CRITICAL
evidence_output	Expected artifact upon success.	DependencyGraphAudit.json

waiver_allowed	Boolean for governance bypass.	false
waiver_authority	Role required for bypass.	N/A

8.1 Enforcement Layers and Blocking Levels

Enforcement layers utilize controlled vocabulary: STATIC, BUILD, SCHEMA, CONTRACT, COMPONENT, WORKFLOW, RUNTIME, GIT, SECURITY, CHAOS, EVALUATION, PRODUCTION, and MANUAL_ASSURANCE. Blocking levels map to operational urgency: INFORMATIONAL, WARNING, COMMIT_BLOCKING, PULL_REQUEST_BLOCKING, RELEASE_BLOCKING, RUNTIME_BLOCKING, and EMERGENCY_STOP.

8.2 Evidence Requirements

Every execution of a fitness function produces a deterministic evidence record containing fitness_result_id, rule_version, subject_version, the cryptographic commit_id, a configuration_digest, and immutable evidence_refs (e.g., test logs, AST trees). This ensures that compliance audits rely on cryptographic proof rather than human attestation.

9. Conformance-Pack Architecture

A scalable agentic development platform requires independent conformance packs. These versioned test suites execute against component implementations without requiring the deployment of the entire orchestration platform. Each pack defines an interface specification, deterministic fake services, invalid input fixtures, security probes, and strict compatibility requirements.[4] Execution yields a standardized certification_result ingested by the CI pipeline.

10-18. Specific Conformance Packs

10. Provider-Adapter Conformance Pack

Verifies that custom integrations with external LLMs adhere to platform contracts. It tests capability discovery, canonical model request acceptance, and the strict rejection of unsupported capabilities.[6] It enforces the normalization of streaming events, token cost usage, and trace propagation. Crucially, it injects malformed provider responses, timeouts, and rate limits to guarantee that the adapter preserves task identity and normalizes errors without leaking provider-native exception types.

11. Coding-Runtime Adapter Conformance Pack

Validates the sandbox where agents manipulate code. The pack verifies the deterministic creation of isolated worktrees and branch bindings. It enforces RepositoryPolicy and the scope of the ChangeManifest. It audits the recording of inspected files, modified files, and captured

commands, ensuring the generation of irrefutable Git evidence. The pack also simulates protected-branch write attempts to verify fail-closed blocking.

12. Tool and MCP Adapter Conformance Pack

Tests the Model Context Protocol (MCP) or similar tool integration layers.[4] It verifies tool capability discovery, argument validation schema enforcement, and permission boundaries.[4] It strictly mandates that a ToolIntent is provided before execution and a ToolReceipt is produced afterward. It tests timeout handling, idempotency, secret redaction, and the containment of malicious, prompt-injecting tool outputs.

13. Sandbox-Provider Conformance Pack

Probes the execution isolation environment using adversarial techniques. It verifies absolute filesystem and project isolation, network-egress restrictions, and immutable resource limits. It tests environment reproducibility, startup/cleanup determinism, and metadata endpoint protection. The suite includes adversarial escape probes to ensure no persistence exists after teardown and that forensic evidence is preserved upon failure.

14. Project-Manifest Conformance Pack

Ensures that onboarding configurations are structurally sound. It verifies stable project/repository identity, memory and credential scopes, defined budgets, risk classifications, and release rules. It actively rejects ambiguous or missing policy fields to prevent default-allow security postures.

15. Fleet-Template Conformance Pack

Validates the configuration of multi-agent fleets. It verifies capability requirements, ensures no physical provider bindings exist within roles, and enforces mathematical limits on delegation depth, retries, and costs. It validates handoff schemas and approval rules, ensuring no role possesses self-approval authority.

16. Workflow-Definition Conformance Pack

Executes static analysis on agentic workflows. It verifies valid graph construction, reachable terminal states, and the absence of invalid cycles. It enforces bounded loops, explicit compensation paths, timeout handling, and valid state transitions, ensuring that no checkpoint boundary can be logically bypassed.

17. Memory-Backend Conformance Pack

Validates the long-term context storage. It tests scope enforcement, ensuring that provenance, versioning, and trust classifications are perfectly preserved. It probes retrieval filtering, cross-project isolation, and supersession logic.[4] It verifies that embeddings are treated strictly as derived data (capable of being recomputed from canonical records) and detects stale memory access attempts.

18. Context Compiler Target Conformance Pack

Verifies the assembly of the prompt context. It ensures the target receives required task information, active ADRs, and trust labels, while correctly excluding superseded decisions. It validates token budget adherence, records summarization truncations, and ensures the production of a mathematically reproducible Context Manifest that is free of cross-project data leakage.

19. Additional Conformance Packs

To support a growing ecosystem, independent packs must be investigated for workflow-engine ports, event producers/consumers, artefact stores, Git forge adapters, secrets brokers, evaluation graders, release controllers, and policy engines.

20. Deterministic versus Probabilistic Conformance

A cornerstone of the architecture is the rigorous segregation of deterministic constraints from probabilistic evaluations.[4]

20.1 Deterministic Tests

Deterministic tests cover rules that must yield identical results given identical inputs. Examples include forbidden AST imports, schema validity, exact state transitions, exact commit bindings, project-scope enforcements, and cost ceilings. A single execution is sufficient. A failure is absolute and immediately blocks the CI pipeline.[9]

20.2 Probabilistic Tests

Probabilistic tests evaluate semantic capabilities inherently subject to LLM variance. Examples include task semantic success, tool-selection accuracy against ambiguous phrasing, handoff quality, and adversarial prompt robustness. These require repeated statistical trials against versioned datasets to calculate confidence intervals and baseline standard deviations.[5] They do not block standard pipelines to avoid flaky builds, instead triggering regression reviews when variance exceeds acceptable limits.[4]

20.3 Hybrid Tests

Hybrid tests combine both methodologies. For example, evaluating a task completion involves a deterministic check (is the JSON result schema valid, and were required unit tests passed?) combined with a probabilistic check (did the agent semantically satisfy the user's intent without making unsupported claims?).

21. CI Execution Tiers

To balance rapid developer feedback against the high computational cost of agent evaluation, CI execution is tiered.

Tier	Name	Target Execution	Controls Run	Blocking Status
Tier 0	Local Developer Checks	< 5 seconds	AST linting, schema validation, manifest linting, unit tests.	Blocks local commit.
Tier 1	Commit Checks	< 30 seconds	Architecture linting, dependency rules, secret scanning, Git policy tests.[2]	Hard blocking on branch.
Tier 2	Pull-Request Checks	< 5 minutes	Component contracts, generated bindings, Change Manifest validation, small smoke suite, security checks.[4]	Blocks PR merge.
Tier 3	Nightly Evaluation	Hours	Broader workflow tests, repeated probabilistic component tests, multi-provider tests, chaos injection.[5]	Non-blocking (Alerts/Regression reports).
Tier 4	Release Qualification	Hours	Frozen golden datasets, full conformance	Critical failures block release.

			packs, complete statistical runs, provider comparisons.	
Tier 5	Production Sampling	Continuous	Trace sampling, drift detection, online semantic evaluation, false-DONE detection, waiver expiry checks.	Triggers rollbacks/alerts.

22. Test Selection and Affected-Scope Analysis

Running an expensive probabilistic evaluation suite for a typo correction is an anti-pattern. The CI platform implements an affected-test algorithm utilizing dependency graph analysis.[2] Inputs to the algorithm include changed files, requirement/schema links, and the Change Manifest. The algorithm traces the AST to identify directly modified components and dependent consumers. It maps these to applicable architectural rules and conformance packs. If a provider adapter is modified, only that adapter's conformance pack and related integration tests are executed. If cross-cutting core orchestration logic is altered, the system automatically escalates the execution to a full Tier 4 qualification suite.

23. Architecture-Drift Detection

Architecture drift is defined as the divergence between active ADRs and actual code dependencies, runtime communication, provider usage, and project isolation.[2]

23.1 Drift Indicators

Drift is detected via anomalous indicators: the introduction of a new forbidden dependency, provider import leakage, direct tool calls bypassing the ToolIntent wrapper, schema version bypasses, components lacking ownership tags, or expired waivers remaining active in the pipeline.

23.2 Drift Scorecard and Trend Analysis

Relying on a single aggregate "health score" is dangerous.[7] The platform produces a multidimensional scorecard reporting separately on Provider-Neutrality Drift, Security-Boundary

Drift, Git-Integrity Drift, and Waiver Debt. Trend analysis tracks the velocity of new violations, the age of unresolved drift, rule false-positive rates, and the specific components exhibiting repeated architectural decay.

24. Waiver and Exception Governance

A fitness function failure cannot be bypassed via an unstructured CI override or a # pragma: no cover comment. Operational realities require exceptions, but they must be strictly governed.[2]

24.1 Waiver Record and Rules

A formal Waiver Record requires a waiver_id, the targeted fitness_function_id, the justification (reason), identified risk, mandatory compensating_controls, an approving authority, and an immutable expires_at timestamp. Waivers automatically expire, immediately reinstating the blocked CI status. Dashboards provide high visibility into accumulating waiver debt.

24.2 Non-Waivable Rules

Certain architectural and security boundaries represent existential risks and are structurally non-waivable. These include cross-project data leakage, direct protected-branch writes, sandbox-host filesystem access, secret exposure, unsigned high-risk approvals, and stale-worker state mutations.[4] Attempting to issue a waiver for these categories results in an automated EMERGENCY_STOP.

25. Flaky and Stochastic Test Governance

The platform explicitly distinguishes between infrastructure flakes (e.g., transient network loss), provider API failures, test data defects, and genuine probabilistic regressions (variance in model reasoning).[4] Statistical tests are not marked as "flaky" merely because LLM quality varies; that variance is the measured property.[5] Probabilistic regression policy requires repeated runs, baseline comparisons, and defined confidence intervals. Tests failing due to deterministic infrastructure issues are quarantined with a maximum duration and automatic issue creation, forbidding permanent ignore lists.

26. Golden Datasets and Architectural Scenarios

Architectural guarantees are proven against version-controlled, immutable golden datasets. A Scenario Record defines the exact initial state, project state, available agents, available tools, provider configuration, expected state transitions, required actions, and forbidden actions. Scenarios cover standard coding tasks, but crucially focus on edge cases: provider failures, worker crashes, cross-project memory attacks, prompt injections, stale approvals, and self-approval attempts. These datasets act as the baseline against which all probabilistic drift is measured.

27. Trace Replay

Production incidents and historical observations are systematically converted into regression

tests. The platform captures comprehensive traces including the context manifest, model requests/responses, tool intents/receipts, state transitions, and Git state. This enables exact replay (mocking historical responses to verify the platform now handles the error correctly) and counterfactual replay (running the historical prompt against a new provider version). A new architectural guardrail is only validated when a historical trace replay proves the incident can no longer reproduce.

28. Failure Taxonomy

Every failure must map to a conformance-oriented taxonomy to accelerate remediation. Categories include:

- *Architecture rule failure* (e.g., forbidden dependency [9]).
- *Provider/Tool contract failure* (e.g., malformed argument, un-normalized error [6]).
- *State-transition / Git-integrity failure* (e.g., branch mismatch).
- *Context-provenance / Memory-isolation failure*.
- *Authority / Handoff / Coordination failure*.
- *Security / Evaluation / False-completion failure*.[4] Every failure telemetry event identifies the likely layer, owning component, relevant fitness function, diagnostic evidence, and standard remediation path.

29. Proof Obligations by Architectural Decision

The platform translates static Architecture Decision Records (ADRs) into executable proof obligations.[2] For example, an ADR mandating **Git Isolation** requires the following executable obligations:

1. *Rule*: One task per writable worktree (Enforced at Runtime).
2. *Rule*: Branch base is cryptographically recorded (Enforced at Git hook / CI).
3. *Rule*: Change Manifest scope is strictly enforced (Enforced at CI Pull Request).
4. *Rule*: Protected branch is not modified (Enforced at Static / Git layer). Each consequence maps to a specific fitness_function_id, ensuring the ADR is permanently enforced.

30. Conformance Certification

Component implementations (e.g., a newly written Anthropic adapter) undergo automated certification. Statuses include NOT_TESTED, PARTIAL, CONFORMANT, CONFORMANT_WITH_WAIVER, NON_CONFORMANT, EXPIRED, and REVOKED. A certification record binds the implementation version to specific test run IDs and expiration dates. Schema changes, core provider updates, or time-based expiry triggers automatically revoke the certificate, forcing the component through the recertification pipeline prior to the next release.

31. Required Experiments

The research programme mandates executable proof via specific experiments:

1. **Ten Executable Invariants:** Implement ten core controls (e.g., protected-branch blocks,

exact-commit bindings, context provenance). Intentionally violate each to verify hard failure.
2. **Provider Adapter Fake:** Build a fake adapter. Run the conformance pack injecting malformed output, timeouts, and native-type leakage to verify normalization and rejection.
3. **Coding Runtime:** Create a task to verify isolated worktree creation, Change Manifest enforcement, and deterministic cleanup upon cancellation.
4. **Tool Adapter Comparison:** Route a canonical ToolIntent through both an MCP adapter and a direct internal adapter; assert the resulting ToolReceipts are structurally identical.[6]
5. **Sandbox Attack:** Execute adversarial probes attempting host filesystem access, forbidden egress, and cross-project access; verify fail-closed containment.[4]
6. **Cross-Provider Continuation:** Start a task with Provider A, checkpoint the state, and resume with Provider B. Verify state preservation and successful semantic continuation without duplicated actions.
7. **Context Truncation:** Force an artificially small token budget. Verify that critical architectural constraints survive truncation, excluded content is logged, and the Context Manifest remains reproducible.
8. **Cross-Project Retrieval Attack:** Attempt memory access utilizing a stale cache, wrong project IDs, and crafted queries. Verify absolute fail-closed isolation.[4]
9. **False Completion:** Attempt to trigger task completion using missing criteria, stale tests, and unresolved blockers. Verify immediate runtime rejection.
10. **Delegation Loop:** Initialize recursive agents that continuously delegate to one another. Verify deterministic depth enforcement, cost ceiling enforcement, and controlled escalation.[11]
11. **Schema-Breaking Change:** Introduce an incompatible schema modification. Verify the CI algorithm identifies downstream consumers and blocks the release.
12. **Waiver Expiry:** Issue a temporary waiver. Verify it bypasses CI for its allowed scope, appears in release evidence, expires automatically at the timestamp, and causes subsequent CI runs to fail.
13. **Trace Replay:** Replay a historical failed task trace. Verify that a newly implemented fitness function successfully prevents recurrence.
14. **Probabilistic Evaluation:** Run repeated trials evaluating tool selection and memory retrieval to establish variance and baseline confidence intervals.[5]
15. **Project Onboarding:** Submit a deliberately invalid project manifest (missing isolation policy, invalid budget). Verify onboarding is instantly rejected.

32-33. Deliverables and Repository Structure

The architecture governance codebase must enforce a separation of concerns, reflected in the repository structure. Deliverables include a comprehensive fitness-function catalogue (fitness-functions.yaml), conformance suite architectures, CI execution policies, waiver governance rules, drift scorecards, and executable PoC tests.
The repository structure isolates rules by enforcement layer: conformance/ ├── catalogue/ (fitness-functions.yaml, packs.yaml) ├── rules/ │ ├── static/ # AST and linting rules [8] │ ├──

schemas/ # OpenAPI/JSON Schema contracts │ ├── git/ # Worktree and commit checks │ ├── security/ # Boundary and injection tests [4] │ ├── runtime/ # State-machine invariants │ └── production/ # Telemetry and drift monitoring ├── packs/ # Isolated conformance suites (MCP, Sandbox, Provider, etc.) [6] ├── scenarios/ # Golden dataset definitions ├── fixtures/ # Mock data and expected outputs ├── fakes/ # Deterministic service fakes ├── chaos/ # Fault injection scripts [11] ├── evaluations/ # Probabilistic LLM test harnesses [5] ├── waivers/ # Active and expired waivers └── schemas/ # FFR and certification JSON schemas

34. Tooling Comparison

Deploying a single monolithic platform to solve all testing layers is a recognized anti-pattern.[10] Optimal architecture demands specialized tools for distinct enforcement layers.

Tool / Pattern	Target Layer	Problem Solved	Limitations	CI Integration	Provider Independence
pytest-archon	Static	Python AST dependency rule enforcement (e.g., blocking forbidden imports).[8]	Python ecosystem only.	Native pytest hook; blocks on PR.	High (checks raw source).
dependency-cruiser	Static	JS/TS dependency boundary mapping.[10]	Node/TS ecosystem only.	Direct CI blocking.	High.
ArchUnit	Static	Java/Kotlin modular monolith boundary tests.[10]	JVM only; requires bytecode compile.	Native JUnit execution.	High.
MCP Inspector	Contract	Validates MCP tool handshake, capability discovery,	Does not simulate high concurrency .[4]	Scriptable CLI integration.	High (protocol-level).

		and schema conformance.[4]			
DeepEval	Probabilistic	LLM evaluation, stochastic metric tracking, and baseline comparisons.[5]	High execution cost; subject to LLM latency.	Tier 3/4 execution.	Medium (relies on evaluator LLMs).
OWASP ZAP / Snyk	Security	Automated boundary, dependency risk, and payload vulnerability scans.[4]	Prone to false positives; requires tuning.	Blocking in Tier 2/4.	High.

35. Mutation and Negative Testing

A fitness function is inherently untrustworthy if its failure mode has not been empirically demonstrated. The platform enforces negative testing—testing the tests—against intentional violations. Every critical fitness function must be validated via schema mutation, policy mutation, state-transition reversal, Git metadata spoofing, and approval-token tampering. The CI pipeline must prove that the fitness function fails closed when subjected to these mutations.

36. Production Evaluation Integration

Pre-release fitness testing relies on simulated approximations of reality. Production integration connects real-world anomalies back to the CI pipeline.[1] The platform utilizes continuous invariant sampling to monitor false-DONE rates and contextual drift in real-time. Every severe production incident must automatically generate a regression scenario, prompt the authoring of a new fitness function, update the affected conformance-pack, and introduce a mandatory release-gate review.

37. Dashboards and Operator Experience

Aggregating all conformance data into a single "health score" dangerously masks systemic rot.[7] Operator views must display segmented, multidimensional data. Dashboards separate

deterministic safety failures (requiring immediate hotfixes) from statistical probabilistic regressions (requiring prompt tuning or model evaluation). Views highlight blocked pull requests, active versus expiring waivers, drift by component, and historical incident regression tracking, ensuring the operational process remains manageable for a single human operator.

38. Anti-patterns to Reject

The research programme explicitly evaluates and rejects the following anti-patterns:

- **Architecture documented but not tested:** Confluence documents do not prevent drift; ADRs must be codified as executable rules.[2]
- **Enforcing critical rules only in prompts:** System prompts fail unpredictably and cannot enforce deterministic boundaries.[4] *Safer replacement:* Runtime state-machine invariants.
- **Single aggregate architecture score:** Collapses distinct risk vectors (e.g., security vs. code style) into an unusable metric. *Safer replacement:* Multidimensional scorecards.
- **Agent self-report as pass evidence:** Trusting an agent to declare task success bypasses empirical verification. *Safer replacement:* Exact-commit test evidence and typed receipts.
- **Permanent test skips / waivers:** Waivers must have forced expiration to prevent the crystallization of technical debt.[2]
- **Using LLM review as the sole blocking control:** LLM architectural review is supportive but fundamentally insufficient as a deterministic blocking control.

39. Source Hierarchy and Recommendations

To maintain analytical rigor, architectural decisions draw strictly from a defined hierarchy of truth: Formal software architecture standards [1] > Official protocol specifications (e.g., MCP) [4] > Peer-reviewed research > Incident reports [4] > Reproducible practitioner material.[8]

Example Recommendation Execution:

- *Architectural Principle:* Tool Mediation & Provider Neutrality.
- *Normative Source:* ADR-042 (Model Context Protocol Adoption).[6]
- *Enforcement Layer:* Contract / Component.
- *Implementation Mechanism:* MCP Conformance Pack runner executing against the adapter URL.[4]
- *Positive Test:* Adapter successfully negotiates protocol version and returns tool list.[4]
- *Negative Test:* Adapter rejects malformed JSON-RPC invocation with -32600.[4]
- *CI Tier:* Tier 2 (Pull Request).
- *Blocking Level:* PULL_REQUEST_BLOCKING.
- *Evidence Produced:* MCPConformanceReport.json.
- *Waiver Policy:* Non-waivable for production releases.
- *Operating Cost:* Low (executes in milliseconds against local binaries).
- *Invalidation Trigger:* Deprecation of MCP protocol standards.

40. Research Quality Controls

The platform design utilizes distinct roles—architecture governance, security conformance, agent evaluation—acting in concert but maintaining independent review authority. Deterministic scripts actively scan the repository to ensure fitness-function schema validation, detect duplicate rules, identify missing normative source links, flag expired waivers, and highlight untested critical invariants. Executive acceptance requires architecture, test-design, and security reviews against an executable prototype.

41. Phased Implementation Roadmap

Implementation proceeds sequentially to manage operator burden and establish immediate ROI:

- **Phase 0 (Invariant Catalogue):** Identify non-negotiable principles, link to ADRs, define the FFR JSON schema.
- **Phase 1 (Static and Schema Controls):** Roll out AST rules rejecting physical model names and direct SDK imports (Tier 1 CI).[2]
- **Phase 2 (Git and Task Controls):** Enforce isolated worktrees, immutable commits, Change Manifest limits, and exact-commit evidence.
- **Phase 3 (Tool and Runtime Conformance):** Deploy MCP conformance packs [4]; implement ToolIntent and ToolReceipt invariants.
- **Phase 4 (Security and Isolation):** Enforce cross-project boundaries, sandbox limits, and approval tokens; introduce active mutation testing.[4]
- **Phase 5 (Workflow and Fleet Testing):** Validate retry bounds, loop detection, checkpoints, and provider failover.[11]
- **Phase 6 (Probabilistic Evaluation):** Implement statistical tracking for tool selection and memory accuracy (Tier 3 CI).[5]
- **Phase 7 (Certification and Drift):** Launch multidimensional drift dashboards, automated trace replay, and complete release-gate integration.

42. Completion Criteria

The architectural assurance model is structurally complete when: every non-negotiable ADR executes as a CI-blocking fitness function traceable to a normative source; portable definitions are structurally immune to physical provider lock-in; tool mediation strictly requires ToolIntent and ToolReceipt execution; task completion accepts only typed, irrefutable Git/test evidence; cross-project memory bounds and sandbox escapes fail closed under adversarial load; waivers expire deterministically; and probabilistic tests inform regressions without generating false-positive pipeline blocks. The framework must remain provider- and project-independent.

43. Final Synthesis Questions

- **Which architectural principles can be enforced statically?** Boundary enforcement, provider-independence (no SDK/model name leakage), dependency direction rules, permitted schema versions, and file-location rules are statically enforceable via AST and linting.[2]

- **Which require runtime tests?** Authorization logic, timeout/retry boundaries, token expiration, state-transition validation, memory provenance, and ToolReceipt generation require dynamic runtime invariant tests.
- **Which require chaos tests?** Distributed systemic guarantees—such as provider failover, database restart recovery, idempotent check-pointing, message duplication handling, and recursive delegation mitigation—require active fault injection.[11]
- **Which require probabilistic evaluation?** Semantic tasks inherently subject to variance, including tool-selection accuracy, handoff relevance, memory retrieval quality, unsupported claim detection, and false-DONE identification require statistical evaluation.[5]
- **Which require manual assurance?** High-risk, ambiguous semantic completions, the granting of exceptional governance waivers, and evaluations of nuanced cultural bias require human, auditable sign-off.[4]
- **Which rules should block commits?** Static AST violations, basic schema faults, forbidden dependency injections, exposed secrets, and Git branch integrity checks.
- **Which should block pull requests?** Component contract test failures, generated binding mismatches, sandbox isolation violations, and Change Manifest scope breaches.
- **Which should block releases?** Full conformance suite failures, chaotic recovery failures, missing exact-commit evidence, and statistically significant probabilistic baseline regressions.
- **Which should trigger an emergency stop?** Production detection of cross-project data leakage, unapproved physical provider routing, unprotected side-effect execution, or active sandbox escapes.
- **How should requirements and ADRs generate fitness functions?** Every ADR must conclude with explicit, machine-readable proof obligations linking directly to a newly authored or updated fitness_function_id executed in CI.[2]
- **How should fitness functions be versioned?** Versioned directly alongside the source code in the repository, treating architecture tests identically to domain feature code.
- **How should tests bind to versions?** Through strict cryptographic commit hashes, configuration digests, and pinned schema tags, ensuring tests cannot falsely validate subsequent mutations.
- **What constitutes provider-adapter conformance?** Absolute adherence to canonical platform capabilities (handshake, normalizations, schema validations, timeouts) using independent fakes, completely abstracting the underlying LLM API.[4]
- **What constitutes coding-runtime conformance?** Deterministic management of isolated worktrees, flawless generation of Git evidence, Change Manifest enforcement, and deterministic cleanup.
- **What constitutes tool-adapter conformance?** Successful MCP (or equivalent) capability discovery, strict argument validation, permission enforcement, and the non-negotiable production of ToolReceipts.[4]
- **What constitutes sandbox conformance?** Provable filesystem/network isolation, process containment, and survival against adversarial escape probes.
- **What constitutes project-manifest conformance?** Presence of stable identity, defined

memory/credential scopes, strict risk classifications, and explicit budget allocations.

- **What constitutes fleet-template conformance?** Structurally defined capability limits, absence of physical provider bindings, and mathematically bounded delegation and retry limits.
- **What constitutes workflow-definition conformance?** Reachable terminal states, bounded loops, defined compensation paths, and valid state transitions via static graph analysis.
- **What constitutes memory-backend conformance?** Perfect provenance preservation, scope enforcement, cross-project isolation, and supersession logic.[4]
- **What constitutes Context Compiler target conformance?** Deterministic reproducibility of the Context Manifest, strict token budget adherence, and accurate inclusion/exclusion of ADRs and trust labels.
- **How should probabilistic variance be handled?** Cordone probabilistic suites into Tier 3 (Nightly) or Tier 4 (Release) executions. Track baseline standard deviations, alerting only on statistically significant regressions, ensuring they do not block standard pipelines.[5]
- **How should flaky tests be distinguished from real regression?** Statistical variance in LLM output is not a "flake"—it is the measured property. True flakes (network timeouts, infrastructure failures) are quarantined with mandatory time limits and automated issue tracking.[4]
- **Which rules may be waived?** Stylistic guidelines, non-critical dependency paths, temporary cost-ceiling breaches during incidents, or experimental feature bounds.
- **Which rules are non-waivable?** Cross-project isolation, secret redaction, direct protected-branch writes, unsigned high-risk approvals, and sandbox host protection.[4]
- **How should architecture drift be measured?** Via multidimensional scorecards tracking new dependency violations, waiver debt accumulation, expired waivers, and unmapped provider imports over time.[2]
- **How should historical incidents become regression tests?** Incident telemetry is captured into a trace manifest. The manifest is replayed in CI via exact and counterfactual replays to ensure the newly introduced fitness function reliably blocks the anomalous state transition.
- **How should production observations affect certification?** If production sampling detects an architecture drift or invariant violation, the component's certification is immediately flagged as REVOKED, forcing recertification.
- **What is the minimum viable fitness-function suite?** A combination of static AST limits preventing provider leakage [8], Git worktree isolation, runtime ToolIntent/Receipt enforcement, and project memory isolation.
- **Which conformance packs are required before the first trustworthy release?** Provider-adapter, Tool/MCP adapter [4], Coding-runtime, and Sandbox-provider conformance packs.
- **Which can be deferred until multi-project operation?** Fleet-template, advanced Memory-backend supersession, and complex Workflow-definition static analysis can be iteratively added.

- **What evidence would invalidate the final recommendation?** The emergence of mathematically provable, 100% deterministic LLM reasoning engines that eliminate the need for stochastic guardrails, or a shift to fully non-versioned execution environments rendering Git-based verification obsolete.

44. Final Execution Instruction

This research establishes an architecture-assurance, CI enforcement, distributed-systems testing, security-conformance, and agent-evaluation framework. It optimizes for executable guarantees, demanding deterministic enforcement where possible, absolute provider and project independence, and exact cryptographic evidence binding.[2]
By stratifying deterministic safety limits from bounded probabilistic evaluations, the platform ensures rapid developer feedback while guaranteeing release safety.[5] The multidimensional tracking of architecture drift and the strict governance of expiring waivers guarantee a low operator burden.[7] Where a principle cannot be enforced automatically today (e.g., assessing nuanced cultural bias in LLM output), the framework enforces an explicit, auditable manual assurance procedure, assigned ownership, and mandatory review frequencies, rather than silently leaving the principle unenforced.[4] This executable architecture transforms passive engineering guidelines into an active, self-defending software platform.

Works cited

1. Building Evolutionary Architectures – Chapter 2: Fitness Functions - Rafael Bernard Araujo, accessed July 26, 2026, https://rafael.bernard-araujo.com/building-evolutionary-architectures-chapter-2-fitness-functions.php
2. AI Architecture Drift: Detection & Fix - TechDebt.works, accessed July 26, 2026, https://techdebt.works/ai-architecture-drift/
3. Fitness Functions for Your Architecture - InfoQ, accessed July 26, 2026, https://www.infoq.com/articles/fitness-functions-architecture/
4. MCP (Model Context Protocol) Testing Guide - QAPractices, accessed July 26, 2026, https://qapractices.com/documentation/mcp-model-context-protocol-testing-guide/
5. 8 LLM Eval Tools Ranked: No Product to Sell | TECHSY, accessed July 26, 2026, https://techsy.io/en/blog/best-llm-evaluation-tools
6. Model Context Protocol (MCP) | Mendix Documentation, accessed July 26, 2026, https://docs.mendix.com/agents/mcp/
7. The Up-and-Running Guide to Architectural Fitness Functions, accessed July 26, 2026, https://mikaelvesavuori.se/blog/2023-08-20_The-Up-and-Running-Guide-to-Architectural-Fitness-Function
8. Inspecting un-imported modules using ast - devork, accessed July 26, 2026, https://devork.be/blog/2012/06/inspecting-un-imported-modules-using/
9. GitHub - jwbargsten/pytest-archon: Rule your architecture like a real developer,

accessed July 26, 2026, https://github.com/jwbargsten/pytest-archon
10. The Modular Monolith 2026 Complete Guide — Spring Modulith, ArchUnit Fitness Functions, and Lessons from Shopify's 30TB/min Architecture - DEV Community, accessed July 26, 2026, https://dev.to/x4nent/the-modular-monolith-2026-complete-guide-spring-modulith-archunit-fitness-functions-and-lessons-878
11. Temporal: Why Developers Are Ditching Cron Jobs for ... - AI Prompts, accessed July 26, 2026, https://prompts.brightcoding.dev/blog/temporal-why-developers-are-ditching-cron-jobs-for-durable-execution
12. pytest-analyzer - PyPI, accessed July 26, 2026, https://pypi.org/project/pytest-analyzer/