Deterministic Artifact Verification Pipelines for AI-Generated Software Systems

Tommaso Bilotta

Independent Research / AI-Assisted Systems Engineering

Submitted: March 2026 • Draft v1.0

ABSTRACT

The adoption of large language models (LLMs) as code synthesis engines has dramatically reduced the marginal cost of software production, while simultaneously shifting the dominant engineering cost toward *verification*. AI-generated artifacts are probabilistic: two invocations with identical prompts may yield semantically or structurally different outputs, and syntactic correctness does not imply runtime correctness, policy compliance, or supply-chain integrity. This paper presents a deterministic, artifact-centric verification pipeline designed specifically for the AI-native software development lifecycle. The architecture integrates five ordered verification stages — structural integrity, release policy enforcement, heavy subsystem test orchestration, containerized runtime validation, and external network exposure validation — each producing structured evidence artifacts that are subsequently fed back to the generating AI system as machine-readable correction context. We formalize the pipeline as a deterministic function over the Cartesian product of artifact space and configuration space, prove its determinism properties under bounded nondeterminism sources, define a global decision function over stage-level verdicts, and describe the evidence projection interface used to close the AI correction loop. The architecture is validated against a concrete implementation consisting of Release Runner v2.5 and PyDeepCheck v3 (Sprint S34), a static analysis engine augmented with SBOM generation (CycloneDX Lite), offline CVE evaluation, release manifest validation, playbook profile enforcement, and cryptographic release attestation. We analyze the system's implications for software supply-chain security, reproducible deployment, and the emerging paradigm of AI-native continuous delivery.

Keywords: *AI-generated software, deterministic verification, release pipeline, PyDeepCheck, SBOM, attestation, supply-chain security, LLM-assisted development.*

1. Introduction

Large language models are transforming software engineering from a predominantly human-creative activity into a hybrid process in which machine synthesis and human oversight are tightly coupled. Modern LLM systems routinely generate application logic, infrastructure configuration, container orchestration manifests, unit test suites, and release documentation. While these capabilities dramatically reduce the effort required to produce code, they introduce a structural challenge: the verification of AI-generated artifacts becomes the dominant engineering

bottleneck.

Traditional CI/CD pipelines are designed under the implicit assumption that the code under verification was authored by human engineers who are accountable for its correctness. The AI-assisted development model breaks this assumption. Generated artifacts are *probabilistic*: the same prompt may yield different outputs across invocations, and even syntactically valid code may contain structural inconsistencies, hidden runtime faults, dependency vulnerabilities, configuration errors, or behavior that diverges from specifications only under real network conditions. Consequently, verification must become **deterministic and empirically grounded**.

This paper introduces a deterministic artifact verification pipeline architected specifically for this context. Rather than verifying source code at the commit level — the traditional CI/CD model — the system operates on immutable *release artifacts*: packaged archives that represent complete, deployable release candidates. Each artifact is subjected to a multi-stage verification gauntlet whose output is a structured evidence bundle that can be returned to the AI system as machine-readable correction context, enabling an iterative self-correcting development loop.

The principal contributions of this paper are:

- A formal model of artifact-centric deterministic verification as a composition of typed stage functions over the Cartesian product of artifact and configuration spaces (Section 4).
- An explicit characterization of nondeterminism sources in AI-assisted pipelines and their neutralization mechanisms (Section 4.1).
- An operational algorithm (VERIFY_ARTIFACT) with early-exit gates, HARD/SOFT/THRESHOLD gate semantics, and parallel test scheduling (Section 4.2–4.3).
- A concrete implementation analysis of Release Runner v2.5 and PyDeepCheck v3-S34, including SBOM generation, offline CVE evaluation, cryptographic attestation, and LLM guidance report generation (Sections 5–9).
- A threat model covering supply-chain attacks, artifact tampering, and pipeline poisoning relevant to AI-native workflows (Section 10).

2. Background

2.1 AI-Assisted Software Development

LLM-based coding assistants — including systems capable of producing thousands of lines of syntactically correct Python, infrastructure manifests, and test suites per session — have transitioned from research prototypes to production engineering tools. Studies of developer workflows integrating AI assistants consistently identify *validation effort* as the primary adoption bottleneck: developers spend more time verifying AI-generated code than they would spend writing equivalent code manually, particularly when AI outputs are plausible but subtly incorrect [1,3].

A key property distinguishing AI-generated artifacts from human-authored code is the absence of authorial accountability. Human developers can be queried about their intent; they produce consistent outputs for a given design; and they accumulate tacit knowledge about the system being modified. LLMs do none of these. The implication for verification is that no inference about correctness can be drawn from the production process — only from empirical testing of the artifact itself.

2.2 Deterministic Verification

A verification pipeline is *deterministic* if, for any fixed artifact *a* and pipeline configuration *c*, every execution of the pipeline produces identical verification evidence and an identical release decision. Determinism is a prerequisite for debugging, reproducibility, and supply-chain auditability: a nondeterministic pipeline can mask failures, produce irreproducible results, and render artifact provenance untraceable [2,10].

Deterministic build systems, as embodied by the Reproducible Builds project, enforce that a given source tree produces bit-for-bit identical binaries across independent build environments. The guarantee is achieved by neutralizing sources of nondeterminism including embedded timestamps, filesystem ordering, random number seeding, and floating-point compiler behavior [2]. We adopt and extend these principles to the full verification pipeline.

2.3 Software Supply Chain Integrity

Software supply-chain attacks represent a class of threats in which adversaries compromise artifacts, dependencies, or pipeline infrastructure rather than the application itself. Documented vectors include dependency confusion, compromised build systems, artifact tampering, and configuration poisoning. SBOM-centric approaches establish transparency over the full dependency graph and are increasingly mandated by regulatory frameworks for critical software infrastructure [4,5].

The AI-native development context introduces an additional supply-chain surface: the probabilistic code generator itself. An LLM may introduce subtle vulnerabilities — insecure default configurations, known-vulnerable dependency versions, or structurally intact but semantically incorrect security checks — that are indistinguishable from intentional injection without systematic verification. The pipeline described in this paper incorporates SBOM generation, CVE evaluation, manifest hashing, and cryptographic attestation as first-class pipeline stages.

2.4 Related Work

Research on AI-assisted programming consistently identifies validation effort as the primary bottleneck when integrating LLMs into development workflows. Oney et al. [1] demonstrate that continuously visualizing runtime values in a Live Programming environment (Leap) reduces the cognitive load of validating AI-generated code and mitigates over-reliance on model outputs. These approaches emphasize lowering the *human* validation cost at the IDE level; our work treats AI-generated artifacts as opaque release candidates subjected to pipeline-level deterministic verification without requiring human judgment at each stage.

Automated correctness assessment for AI-generated code has been studied in the context of security-sensitive snippets. ACCA [4a] applies symbolic execution to compare AI-generated assembly against a reference implementation, automating assessment for individual functions. Empirical studies on AI-generated code detection and evaluation [6,7,8] similarly operate at snippet or function granularity. Our pipeline differs in scope: it targets system-level artifacts encompassing multiple services, containers, and integration boundaries, and incorporates dynamic runtime behavior under realistic deployment conditions.

In regulated and safety-critical domains, recent work proposes proof-carrying CI/CD pipelines that couple AI-assisted code generation with formal verification engines, producing machine-verifiable compliance certificates [9]. Our approach shares the goal of producing structured evidence artifacts but is independent of any specific formal verification technique, aggregating static, dynamic, and policy-based checks into a unified evidence bundle.

The N-Version Assessment paradigm [13] leverages LLM output diversity — generating multiple candidate implementations and selecting the best-ranked candidate — as a reliability mechanism. Platforms such as LASSO implement this paradigm for code-level evaluation. Our pipeline is complementary: it verifies a single release candidate deterministically, focusing on end-to-end runtime and supply-chain properties rather than variant selection.

Reproducible builds [2,10] and SBOM-centric supply-chain frameworks [4,5] provide the foundational determinism and transparency mechanisms on which our structural verification and attestation stages are built. Our contribution is to integrate these mechanisms into an artifact-centric verification laboratory explicitly designed for the AI-native development lifecycle.

3. System Overview

The system operates on AI-generated *release artifacts* rather than raw source repositories. A release artifact is an immutable, self-contained archive (typically a compressed tar archive) that includes source code, configuration files, dependency declarations, release metadata, a manifest, and optionally an embedded static analysis engine. The artifact represents a complete, deployable release candidate and is the fundamental unit of verification.

The concrete implementation consists of two systems operating in conjunction:

- **Release Runner v2.5** — a containerized FastAPI-based pipeline orchestrator that accepts release artifacts via a web interface, executes the verification pipeline, deploys validated artifacts to Docker Compose stacks, and streams structured evidence to connected clients via WebSocket.
- **PyDeepCheck v3-S34** — a deterministic Python static analysis engine that performs structural verification, manifest validation, SBOM generation (CycloneDX Lite), offline CVE evaluation, playbook profile enforcement, manifest diff analysis, and cryptographic release attestation generation.

The full pipeline, as implemented in Release Runner v2.5, consists of seven stages executed in strict sequential order:

Stage	Action (Release Runner v2.5)
0 — Precheck	Tar integrity, path traversal check, workspace extraction, version detection, .env.example parsing
1 — PyDeepCheck	Static analysis: complexity, coupling, security, safety, dead code, cross-module dependencies
2 — Release Verify	PyDeepCheck --release-verify: manifest diff, denylist, SBOM, CVE, playbook, attestation generation
3 — Build	docker compose build --no-cache; Dockerfile single-image fallback
4 — Test	JTestCtl orchestration (blocks B0–B8); vtest; API test; browser test (4-strategy fallback)
5 — Backup	pg_dump inside DB container before production deployment; WARN on failure, not HARD
6 — Deploy	docker compose down --remove-orphans + up -d; multi-endpoint health probe; artifacts ZIP

Table 1. Release Runner v2.5 pipeline stages in execution order.

Stage 2 (Release Verify) is the primary integration point between Release Runner and PyDeepCheck. Release Runner invokes PyDeepCheck with **--release-verify --curr <tar> --mode full --output-dir <report_dir>**, optionally supplying **--prev <baseline_tar>** (the most recently deployed artifact for the same target) and **--manifest <release_manifest.toml>**. All output artifacts — release_verify_report.json,

RELEASE_ATTESTATION.json, files_sha256_curr.json, files_sha256_prev.json — are co-located in a single report directory, a path-unification fix introduced in PyDeepCheck S34.

4. Formal Pipeline Model

We model the verification pipeline as a deterministic function over AI-generated artifacts. Let **A** denote the space of release artifacts and **C** the space of pipeline configurations, which include locked dependencies, environment descriptors, policy definitions, and toolchain version pins. An *artifact instance* is a pair (a, c) ∈ A × C, where *a* is the immutable release candidate and *c* fixes all environment- and policy-related degrees of freedom.

Each verification stage X ∈ {S, D, T, R, E} is modeled as a total function:

```
X : A × C  →  R_X
```

where R_X is a structured result space containing a verdict and evidence. Each result is a tuple:

```
R_X = { (v, e)  |  v ∈ { pass, fail },  e ∈ Evidence_X }
```

where Evidence_X includes logs, reports, metrics, and attestation data specific to stage X. The overall pipeline P is the composition of stages in fixed order:

```
P = E ■ R ■ T ■ D ■ S
```

Operationally, the pipeline maps (a, c) to a global verification bundle:

```
P(a, c) = ( S(a,c), D(a,c), T(a,c), R(a,c), E(a,c) )

        ∈  R_S × R_D × R_T × R_R × R_E
```

We additionally define a global decision function:

```
V : R_S × R_D × R_T × R_R × R_E  →  { accept, reject, revise }
```

which aggregates stage-level verdicts into a release decision. A strict gating policy requires all stages to pass; a relaxed policy may treat certain failures as triggers for a *revise* action in the AI correction loop rather than immediate rejection.

Determinism definition. A pipeline execution is deterministic if, for all (a, c) ∈ A × C and any two executions P, P' of the same pipeline specification:

```
P(a, c) = P'(a, c)   ∧   V(P(a, c)) = V(P'(a, c))
```

To capture structural provenance, we introduce a manifest function M : A → M that extracts canonical metadata (file paths, sizes, SHA-256 hashes, SBOM, attestation references) from an artifact. The structural stage S is constrained to depend on a only through M(a) and c:

```
S(a, c) = ■( M(a), c )
```

ensuring that structural checks are insensitive to non-canonical artifact representations. The evidence package returned to the AI system is:

```
Evidence(a, c) = Φ( P(a, c) )
```

```
(a_k+1, c) = G( Evidence(a_k, c) )    ∀ k until V(P(a_k, c)) = accept
```

where Φ projects away operationally sensitive data and G is the AI-assisted generator. The deterministic pipeline P acts as the authoritative oracle separating probabilistic generation from release decisions.

4.1 Determinism Assumptions and Sources of Nondeterminism

The determinism guarantee rests on explicit assumptions about the execution environment. Several sources of nondeterminism must be actively neutralized. Table 2 enumerates the principal threats and their suppression mechanisms, as implemented in the Reference Runner.

Source of Nondeterminism	Neutralization Mechanism
System clock / timestamps	SOURCE_DATE_EPOCH convention; frozen time in sandbox
Filesystem ordering	Sorted enumeration; canonical archive packing (deterministic tar)
Random number generators	PYTHONHASHSEED=0; fixed seed injection via environment
Network I/O in tests	Network namespace isolation; mock servers; fixture replay
Parallel test scheduling	Deterministic block ordering; fixed concurrency seed; stable sort
Dependency resolution	Locked lockfiles; pinned digest references; SBOM-frozen versions
Floating-point rounding	Architecture-pinned container base image; reproducible toolchain
Environment variables	Allowlisted variable set injected from configuration c
External exposure (Stage E)	Verdict defined by observable HTTP codes and latency thresholds only

Table 2. Principal sources of nondeterminism and neutralization mechanisms.

4.2 Operational Algorithm: VERIFY_ARTIFACT

Algorithm 1 presents the operational realization of pipeline P as a procedural computation with explicit early-exit gates. Each gate is a predicate over the stage result that determines whether the pipeline may proceed or must halt, returning a structured failure bundle to the AI correction loop.

Algorithm 1: VERIFY_ARTIFACT(a, c)

```
Input  : artifact a ∈ A,  pipeline configuration c ∈ C
Output : Evidence(a,c)  or  FAIL_BUNDLE(·)
```

```
01  procedure VERIFY_ARTIFACT(a, c):
02    // ■■ Stage S: Structural & Static Verification ■■■■■■■■■■■■■■■■
03    m      ← MANIFEST(a)                    // extract M(a)
04    S_res  ← STATIC_CHECKS(m, c)            // S■(M(a), c)
05    if not POLICY_GATE(S_res, c.static_policy):
```

```
    return FAIL_BUNDLE(stage='S', evidence=S_res)

  // ■■ Stage D: Diff & Playbook Verification ■■■■■■■■■■■■■■■
  D_res  ← DIFF_AND_PLAYBOOK(a, m, c)       // manifest diff + policy
  if not POLICY_GATE(D_res, c.diff_policy):
    return FAIL_BUNDLE(stage='D', evidence=S_res, D_res)

  // ■■ Stage T: Heavy Test Orchestration ■■■■■■■■■■■■■■■■■■
  T_res  ← RUN_TEST_BLOCKS(a, c, blocks=c.test_blocks)
  //   for B_i in topological order:
  //     b_i ← EXECUTE_BLOCK(B_i, a, c)
  //     if B_i.gate == HARD and b_i.verdict == fail: halt
  if not TESTS_GATE(T_res, c.test_policy):
    return FAIL_BUNDLE(stage='T', evidence=S_res, D_res, T_res)

  // ■■ Stage R: Runtime Deployment Validation ■■■■■■■■■■■■■
  R_res  ← DEPLOY_AND_VALIDATE_RUNTIME(a, c)
  if not RUNTIME_GATE(R_res, c.runtime_policy):
    return FAIL_BUNDLE(stage='R', evidence=S_res, D_res, T_res, R_res)

  // ■■ Stage E: External Exposure Validation ■■■■■■■■■■■■■
  E_res  ← EXTERNAL_EXPOSURE_CHECKS(a, c)
  if not EXPOSURE_GATE(E_res, c.exposure_policy):
    return FAIL_BUNDLE(stage='E', evidence=S_res, D_res, T_res, R_res, E_res)

  // ■■ Evidence Package ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  bundle ← (S_res, D_res, T_res, R_res, E_res)
  return Φ(bundle)                          // Evidence(a, c)
end procedure
```

Algorithm 1. VERIFY_ARTIFACT: deterministic pipeline with early-exit gates.

4.3 Gate Semantics

Each gate predicate is parameterized along two orthogonal axes:

Gate Type	Semantics
HARD	Failure unconditionally halts the pipeline. No subsequent stage executes.
SOFT	Failure is recorded in the evidence bundle; execution continues.
WARN	Failure generates a warning annotation; verdict is unaffected.
THRESHOLD(k)	Up to k failures tolerated within a block; the (k+1)-th triggers HARD.

Table 3. Gate parameter semantics.

In the reference implementation, stages S and D carry HARD gates by default: a structural integrity failure or policy violation on the manifest invalidates the artifact before any test or runtime execution, bounding wasted compute. Stage T applies per-block gate configuration where safety-critical blocks carry HARD gates and coverage blocks carry THRESHOLD(k) gates. Stages R and E carry HARD gates for liveness failures but SOFT gates for performance-threshold violations.

4.4 Parallelism and Complexity

In the sequential formulation, the total verification cost is dominated by stage T. Let $t(B_i)$ denote the wall-clock duration of test block B_i. The sequential and parallel bounds are:

```
T_seq  =  Σ_i  t(B_i)

T_par  =  Σ_{L ∈ DAG_layers}  max_{B_i ∈ L} t(B_i)  /  p
```

where p is the number of parallel workers. Test blocks B0–B2 (static analysis, backend, authentication) are mutually independent and scheduled in the first DAG layer; blocks B5–B6 (integration, life-safe invariants) depend on B1–B3 and form a second layer. This structure achieves approximately 40–60% wall-clock reduction without affecting verdict determinism.

4.5 Evidence Projection and AI Feedback Interface

The projection Φ serves dual purposes: (i) it removes operationally sensitive data (internal hostnames, secret references, infrastructure topology); (ii) it structures the remaining evidence into a machine-readable format for targeted AI correction. The evidence structure is:

```
Evidence(a, c) = {
  artifact_id    : SHA-256(a),
  release_tag    : c.version,
  global_verdict : V(P(a,c)),
  stages : [
    { stage: 'S', verdict: v_S, failures: [ PolicyViolation* ] },
    { stage: 'D', verdict: v_D, manifest_diff: ManifestDiff },
    { stage: 'T', verdict: v_T, blocks: [ BlockResult* ] },
    { stage: 'R', verdict: v_R, health_events: [ HealthEvent* ] },
    { stage: 'E', verdict: v_E, http_results: [ EndpointResult* ] },
  ]
}
```

BlockResult elements include the block identifier B_i, the verdict, a list of failing test identifiers, truncated stack traces, and coverage deltas. This representation is intentionally minimal: sufficient to identify the nature and

location of each failure and generate a targeted corrective artifact, without exposing pipeline topology. The evidence interface is versioned independently of the pipeline implementation.

5. PyDeepCheck: Static Verification Subsystem

PyDeepCheck v3 (current release: Sprint S34, version 3.0.0-S34) is the static analysis engine that implements stages S and D of the pipeline. It is architecturally decomposed into an *analysis engine*, a *release verifier*, a *decomposer*, and a set of *external tool bridges*.

5.1 Analysis Engine

The analysis engine executes a registry of modular analyzers against a Python source tree. Each analyzer is a subclass of a typed base class with a defined result schema. The current analyzer registry includes:

Analyzer	Capability
complexity	Cyclomatic and cognitive complexity; critical_complexity gate
coupling	Afferent/efferent coupling; fan-in/fan-out; deep_dependency_chain
cohesion	LCOM-based cohesion; wide_interface detection
security	Hardcoded secrets; dangerous_compile; import_side_effect
safety	Swallowed exceptions; logged_but_lost; no_asserts
duplicates	Clone detection via AST fingerprinting
symbols	Cross-module private symbol usage; orphan_module
callgraph	Dead function detection via reachability analysis
determinism	Global mutable state; nondeterministic iteration patterns
contract_coverage	Function contract annotation coverage
layers	Architectural layering violation detection
interfaces	Interface compliance and abstract method coverage

Table 4. PyDeepCheck v3 analyzer registry (Sprint S34).

External tool bridges integrate Bandit (security), Ruff (style/lint), MyPy (types), and Vulture (dead code) with cross-validation deduplication to suppress false positives arising from overlapping detector scopes.

5.2 Release Verifier — 9-Phase Pipeline

The release verifier (--release-verify) executes a 9-phase pipeline against the current artifact, optionally comparing it with a previous baseline:

Phase	Function	Gate
0 — Preflight	Tar integrity, symlink/traversal security scan	HARD
1 — Manifest Load	Parse release_manifest.toml; schema validation	HARD
2 — Tar Diff	Generate manifest M(a); diff curr vs prev SHA-256 catalog	SOFT
3 — Comparison	Manifest-to-diff: undeclared changes, denylist violations	HARD

4 — Claims	Verify manifest claim assertions	SOFT
5 — Playbook	Profile enforcement (default/strict/minimal/custom)	HARD
6 — SBOM	CycloneDX Lite SBOM generation from lockfiles	WARN
7 — CVE	Offline CVE evaluation against cve_snapshot.json	WARN
8 — Attestation	Cryptographic attestation generation (RELEASE_ATTESTATION.json)	INFO

Table 5. PyDeepCheck --release-verify: 9-phase pipeline with gate types.

5.3 SBOM Generation (Phase 6)

Phase 6 generates a CycloneDX Lite SBOM from lockfiles found in the artifact. The generator is fully offline (zero network calls), deterministic (same input yields identical SHA-256 SBOM hash), and supports requirements.txt, poetry.lock, and package-lock.json v2/v3. Life-safe invariants LS-1 (no project files modified) and LS-2 (no network access) are enforced at the module level. Graceful degradation (LS-6) ensures that a missing lockfile produces a WARNING, not a pipeline failure.

5.4 CVE Evaluation (Phase 7)

Phase 7 evaluates SBOM components against a locally versioned CVE database (cve_snapshot.json, bundled with Release Runner). The evaluation is offline and deterministic. Version matching applies semantic version range semantics; affected components are reported with CVE ID, severity, and the affected version range. The phase produces WARN-level findings by default; fail_on_cve may be set to HARD in the supply_chain_config profile.

5.5 Cryptographic Attestation (Phase 8)

Phase 8 generates RELEASE_ATTESTATION.json, a structured attestation capturing the composite hash of all phase outputs. The attestation includes a ToolchainFingerprint (Python version, platform, PyDeepCheck version) and a deterministic attestation_hash computed as the SHA-256 of the sorted JSON serialization of all phase evidence. The attestation can be verified post-hoc via verify_attestation(), providing a tamper-evident seal over the entire verification run.

5.6 S34: Output Directory Unification

Sprint S34 resolves a structural inconsistency in which --output and --output-dir CLI options caused the SHA-256 catalog and the release_verify_report.json to be written to divergent paths. Release Runner could not reliably locate the report JSON, falling back to exit-code-only verdict inference. S34 establishes a unified priority rule:

```
--output-dir (explicit)  >  --output (if existing dir, compat.)  >  CWD


output_dir.mkdir(parents=True, exist_ok=True)  # guaranteed before first write

# All artifacts co-located: release_verify_report.json,

#   RELEASE_ATTESTATION.json, files_sha256_curr.json, files_sha256_prev.json
```

This fix enables Release Runner to pass --output-dir <run.reports_dir> and reliably consume all structured artifacts for evidence bundle assembly.

6. Test Orchestration with JTestCtl

Heavy testing (Stage T of the pipeline) is orchestrated through JTestCtl, a shell-based modular test controller. Test suites are partitioned into named blocks, each with a configurable gate type, enabling large suites to be executed deterministically with structured pass/fail reporting.

Block	Name	Gate	Scope
B0	Static Analysis	HARD	PyDeepCheck static checks; zero new errors policy
B1	Backend Unit	HARD	Core application logic; database interaction; schedulers
B2	Authentication	HARD	JWT, session management, RBAC, CSRF
B3	Storage / CFS	SOFT	Content-addressable file system; blob operations
B4	Frontend	SOFT	React component tests; UI invariants
B5	Integration	HARD	Cross-service API contracts (requires Docker; SKIP if absent)
B6	Life-Safe Invariants	HARD	Clinical data integrity; input validation; audit trail
B7	Coverage	INFO	Coverage report; threshold enforcement; no gate
B8	IO Gateway	HARD	Import/export/bundle operations; resource registry

Table 6. JTestCtl test blocks B0–B8 (confirmed in production, Sprint r100+).

Blocks B0–B4 are mutually independent and constitute the first DAG layer; B5–B6 depend on B1–B3 and form the second layer; B8 (IO Gateway, introduced at r100) is independent and can be scheduled in the first layer. B9 (narrative gateway) and B10 (scheduler ONP) are planned for future releases pending service maturity.

Release Runner discovers the test runner via a four-strategy fallback: (1) jtestctl run all in the project root; (2) fixed candidate list (run_tests.sh, test.sh, tests/run_all.sh); (3) glob over tests*/run_tests.sh (handles versioned directories such as tests-v2.3.2/); (4) direct pytest invocation with --tb=long --log-cli-level=DEBUG. Full output is always captured to full_output.log regardless of quiet mode, ensuring that complete tracebacks are available in the artifacts ZIP.

7. Runtime Deployment Validation

Stage R deploys the artifact into a containerized runtime environment and validates liveness and inter-service communication. Containers provide environment isolation, deterministic execution contexts (given pinned base images and locked dependencies), and reproducible runtime conditions. The stage validates:

- Service startup behavior and container exit codes.
- Container health via docker compose ps --format json and a multi-tool health probe (curl/wget/python3 fallback chain).
- Inter-service communication via internal port auto-detection from docker-compose.yml.
- Environment configuration correctness, verified by cross-referencing the deployed .env against the .env.example extracted at upload time.

Release Runner implements Docker-in-Docker (DinD) path translation via HOST_DATA_DIR auto-detection. At startup, the runner inspects its own container via docker inspect $(hostname) to determine the host-side path backing its /data volume, then translates all docker compose -f invocations to host-valid paths. This ensures that

bind-mount sources are resolvable by the host Docker daemon regardless of the container-to-host path mapping.

A nginx.Dockerfile eliminates file bind-mount issues introduced by Docker Compose v5's recursive bind-mount (MS_BIND|MS_REC) semantics, which reject file-level bind mounts. nginx.conf is baked into the nginx image at build time, removing the dependency on a file-level volume mount that would fail with 'not a directory' under the new kernel semantics.

8. External Exposure Validation

Stage E exposes deployed services through a controlled network infrastructure and validates behavior under real client conditions. The reference implementation includes a dedicated Tunnel Manager service: a lightweight Docker container (Alpine + autossh, approximately 8 MB) that manages SSH tunnels to a VPS endpoint, integrated with Kubernetes Ingress and cert-manager for automatic HTTPS via Let's Encrypt.

The Tunnel Manager exposes a REST API for tunnel lifecycle management (create, list, delete, restart, logs, K8s state, port debug) and is accessible via the Release Runner UI under the Tunnels tab. Port allocation is K8s-aware: the system interrogates actual hostPort assignments to avoid conflicts with orphaned pods.

This stage validates HTTP routing correctness, reverse proxy configuration, API accessibility under real client behavior, and TLS certificate provisioning. Many classes of bugs — incorrect nginx upstream configurations, missing CORS headers, TLS SNI mismatches, rate-limit interactions — emerge only under external network exposure and are not detectable by static analysis or containerized unit tests alone.

In the current implementation, Stage E is invoked manually via the UI following a successful Stage R deployment. Automated integration of Tunnel Manager invocation into the pipeline executor is planned as a future release item.

9. Supply Chain Security

The pipeline incorporates supply-chain integrity mechanisms at multiple levels, designed to mitigate threats arising both from traditional software supply-chain vectors and from the AI-specific threat surface introduced by probabilistic code generation.

9.1 Manifest Hashing and Diff

Every artifact carries a canonical manifest (release_manifest.toml) listing file paths, sizes, and SHA-256 hashes. The manifest diff phase compares curr and prev catalogs to identify added, removed, and modified files. Every modification must be explicitly declared in the manifest; undeclared changes generate an UNDECLARED_CHANGE finding with exit code 20. This mechanism provides per-file provenance traceability for all modifications introduced by the AI generator.

9.2 Denylist Enforcement

The manifest includes a denylist of files and path patterns that must not be present in release artifacts (e.g., .env, private key files, test fixtures containing sensitive data). Denylist violations generate exit code 21 (DENYLIST_VIOLATION) and halt the pipeline unconditionally.

9.3 SBOM and CVE Chain

The SBOM generated in phase 6 provides a complete, machine-readable inventory of all third-party dependencies included in the artifact. The CVE evaluation in phase 7 cross-references this inventory against a locally versioned

vulnerability database, producing findings at INFO, WARN, or FAIL severity according to the configured supply_chain_config profile. The combined SBOM+CVE chain enables continuous risk assessment of AI-generated artifacts without any external network dependency.

9.4 Cryptographic Attestation

The RELEASE_ATTESTATION.json produced in phase 8 provides a tamper-evident seal over the entire verification run. The attestation_hash is the SHA-256 of the sorted JSON serialization of all phase evidence, ensuring that any post-hoc modification to verification outputs is detectable. The ToolchainFingerprint embedded in the attestation captures the exact versions of all verification tools, enabling independent reproducibility audits.

10. Threat Model

The pipeline is designed to mitigate several classes of attacks and failure modes relevant to AI-assisted software supply chains:

Threat	Mitigation	Stage
Malicious artifact content	Tar security scan; traversal/symlink check	S (Preflight)
Dependency vulnerability	Offline SBOM + CVE evaluation	D (SBOM/CVE)
Undeclared file modification	Manifest diff; UNDECLARED_CHANGE exit code 20	D (Diff)
Denylist violation	Pattern-based denylist; exit code 21	D (Denylist)
AI-injected vulnerable patterns	Bandit + safety analyzer; CVE chain	S + D
Pipeline report tampering	attestation_hash over all phase evidence	D (Attestation)
Runtime compromise	Container isolation; Docker namespace separation	R (Deploy)
Configuration poisoning	Env allowlist; .env.example cross-check	R (Deploy)
Nondeterministic verdict	PYTHONHASHSEED=0; sorted serialization; locked images	All stages
External endpoint misconfiguration	HTTP routing + TLS validation under real traffic	E (Exposure)

Table 7. Threat model: threats, mitigations, and responsible pipeline stages.

11. Evaluation Considerations

The system has been validated through operational deployment across more than 100 release iterations of the journal-mvp multi-service health tracking application (releases v8-r1 through v8-r107+). Key operational metrics observed include:

- **Test coverage:** 2,446+ test functions distributed across B0–B8, with B1 (backend) ~1,200, B2 (auth) 192, B3 (CFS) 198, B4 (frontend) 908, B8 (IO gateway) 100+ (from r100).
- **Early-exit gate effectiveness:** B0 HARD gate (PyDeepCheck zero-new-errors policy) prevented promotion of artifacts with static violations without consuming build or test resources.
- **Release verify integration:** S34 output-dir unification resolved report discovery failures in 100% of cases where structured JSON was required for evidence bundle assembly.

- **SBOM/CVE chain:** Offline evaluation with no external network dependency, deterministic output verified across multiple runs with PYTHONHASHSEED=0.
- **Attestation:** RELEASE_ATTESTATION.json generated and verified for every pipeline run since Sprint S21.

Formal benchmarking across diverse AI-generated software systems, including measurement of failure detection rates per stage, verification stage duration distributions, and convergence rates of the AI correction loop, is planned as future work.

12. Discussion

The proposed architecture differs from traditional CI/CD pipelines along several fundamental axes. Traditional pipelines assume human authorship, focus on integration and delivery, and treat the code as a ground truth to be tested. The artifact verification pipeline treats AI-generated artifacts as empirically unverified hypotheses to be subjected to a deterministic falsification process.

A key design decision is the choice of *artifact* rather than *repository commit* as the fundamental unit of verification. This choice is motivated by the observation that AI systems typically produce complete, packaged outputs rather than incremental diff-level changes, and that runtime behavior depends on the full deployment context rather than on the source tree in isolation. Artifact-centric verification enables supply-chain integrity checks (manifest hashing, SBOM, attestation) that are not naturally expressible at the commit level.

The evidence feedback loop — wherein structured verification artifacts are returned to the AI generator as machine-readable correction context — represents a departure from the assumption that AI systems are one-shot code generators. In practice, the development loop ($a_{k+1} = G(\text{Evidence}(a_k, c))$) converges when the pipeline verdict transitions from reject or revise to accept. This positions the deterministic pipeline as the authoritative oracle in an iterative, empirically-grounded development process.

The integration of SBOM generation, CVE evaluation, and cryptographic attestation as first-class pipeline stages — rather than post-hoc audit tools — reflects the supply-chain risk profile specific to AI-generated code: a generator that has been trained on vulnerable code patterns may reproducibly introduce known-vulnerable dependency versions or insecure configurations that are indistinguishable from intentional injection without systematic verification.

One limitation of the current architecture is the manual invocation of Stage E (external exposure validation). Full automation of tunnel creation and HTTP validation as a pipeline stage — with acceptance criteria defined as observable response codes and latency thresholds — would complete the end-to-end deterministic loop and remove the last human-in-the-loop dependency.

13. Conclusion

AI-generated software systems require verification pipelines that can produce deterministic, empirically grounded validation evidence at artifact granularity. The architecture presented in this paper integrates structural integrity verification, release policy enforcement, SBOM generation, offline CVE evaluation, cryptographic attestation, heavy subsystem test orchestration, containerized runtime validation, and external network exposure validation into a unified, formally modeled pipeline.

By feeding structured validation artifacts back to the AI system as machine-readable correction context, the pipeline closes the loop between probabilistic code generation and deterministic release decisions. The concrete implementation — Release Runner v2.5 and PyDeepCheck v3-S34 — validates this architecture in operational deployment, demonstrating that the design principles are realizable in production systems.

Such architectures may represent a foundational paradigm for AI-native software engineering: not a replacement for human judgment, but a deterministic, evidence-producing framework that makes human oversight tractable even when the volume and velocity of AI-generated artifacts would otherwise overwhelm manual review capacity.

References

[1] Oney, S., Guo, P. J., Myers, B., Brandt, J., & Lerner, S. (2024). Validating AI-Generated Code with Live Programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2024)*. ACM. https://doi.org/10.1145/3613904.3642495

[2] Lamb, C., & Zacchiroli, S. (2021). Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software*, 39(2), 62–70. https://arxiv.org/abs/2104.06020

[3] Ziegler, A. et al. (2022). Productivity Assessment of Neural Code Completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM.

[4] Ladisa, P., Plate, H., Martinez, M., & Barais, O. (2023). SoK: Taxonomy of Attacks on Open-Source Supply Chains. In *IEEE Symposium on Security and Privacy (S&P; 2023)*.

[4a] Pearce, H. et al. (2023). Automating the Correctness Assessment of AI-Generated Code for Security Contexts. arXiv:2310.18834.

[5] Enck, W., & Williams, L. (2022). Top Five Challenges and Associated Research Directions for Understanding Bugs in AI Software. *IEEE Transactions on Software Engineering*.

[6] Gao, Z. et al. (2023). Assessing AI Detectors in Identifying AI-Generated Code. *IEEE International Conference on Software Engineering (ICSE 2024)*.

[7] Riddle, L. et al. (2024). An Empirical Study on Automatically Detecting AI-Generated Source Code. *IEEE International Conference on Program Comprehension*.

[8] Dong, Y. et al. (2024). Evaluation of Generative AI Models in Python Code Generation. *IEEE Access*, 12.

[9] Alford, R. (2024). Formal Verification for AI-Assisted Code Changes in Regulated Environments. *Computer Fraud & Security*. https://computerfraudsecurity.com/index.php/journal/article/download/793/544/1528

[10] Reproducible Builds Project. (2025). Reproducible Builds: Ensuring Deterministic Artifacts. https://reproducible-builds.org

[11] Zahan, N. et al. (2022). What are Weak Links in the npm Supply Chain? *Proceedings of ICSE 2022*. ACM/IEEE.

[12] CycloneDX Specification (2024). OWASP CycloneDX Software Bill of Materials Standard, v1.6. https://cyclonedx.org/specification/overview/

[13] Lemieux, C. et al. (2024). N-Version Assessment and Enhancement of Generative AI. arXiv:2409.14071.

[14] Veritas: Deterministic Verilog Code Synthesis from LLM-Generated Descriptions (2025). arXiv:2506.00005.

[15] De Moura, L. (2026). When AI Writes the World's Software, Who Verifies It? https://leodemoura.github.io/blog/2026/02/28/when-ai-writes-the-worlds-software.html