The Unknowable Code: A Framework for Measuring and Mitigating Software Liability

Joshua Hickson, Oleksandr Voievodin, Deepti Sawhney, Samuel Lee Cong, Kuan Zhou, Alaa Elobaid

February 2026

Operational Context:

- **Status:** Validated Post-Mortem Architecture.
- **Result:** LogoMesh secured 1st Place in the Software Testing track of the Berkeley RDI AgentBeats Competition (Phase 1).
- **Open Research:** The core evaluation framework has been open-sourced for Phase 2. Tryout issues for researchers and engineers joining the Lambda Track (Adversarial Prompt Injection) and Purple Agent development are available at: `https://github.com/LogoMesh/LogoMesh/labels/phase-2-tryout`

1 Executive Summary: Operationalizing Contextual Debt

Software organizations have spent two decades managing **Technical Debt**—a compounding cost from suboptimal implementation choices. A more measurable liability is now accumulating at scale: **Contextual Debt**, defined here as the systematic erosion of the verifiable *intent* and *rationale* linking code to its requirements.

This metric is increasingly relevant due to two concurrent trends: the widespread adoption of AI coding assistants that generate locally correct but globally incoherent code, and the proliferation of microservice architectures that fragment domain knowledge. The result is systems where functional correctness is maintained, but the traceability of design decisions is lost.

This paper formalizes Contextual Debt as a measurable proxy construct within controlled evaluation environments and introduces the **LogoMesh** architecture as an empirical framework for bounding it. LogoMesh implements a **Contextual Integrity Score (CIS)**, a **Star-Topology Evaluation Architecture**, and a **Decision Bill of Materials (DBOM)**. The system was validated by securing 1st Place in Phase 1 of the Berkeley RDI AgentBeats Competition, providing empirical evidence that contextual alignment between code and intent can be quantified and adversarially stress-tested.

2 Defining Contextual Debt: A Proxy for Lost Intent

Contextual Debt represents a shift in how software liability can be conceptualized. Where technical debt describes a system that is poorly constructed, Contextual Debt describes a system where the link between artifact and purpose has degraded.

2.1 The Anatomy of Contextual Debt

For the purposes of this study, Contextual Debt is defined as the accumulated liability arising from uncaptured information across the software development lifecycle. This liability rests on three measurable pillars:

1. **Loss of Intent Traceability:** The inability to map a feature back to the business goals that drove its creation.
2. **Loss of Architectural Rationale:** The absence of documentation explaining significant design choices.
3. **Loss of Domain Knowledge:** The evaporation of nuanced business logic from the codebase itself.

2.2 Technical Debt vs. Contextual Debt: A Comparative Framework

The difference between technical and Contextual Debt lies in the axis of failure: technical debt is a failure of implementation efficiency; Contextual Debt is a failure of intent preservation.

Dimension	**Technical Debt**	**Contextual Debt**
Nature of Failure	Implementation inefficiency	Intent preservation failure
Core Metaphor	Financial loan requiring repayment	Loss of traceability
Typical Manifestation	High complexity, outdated libraries	Ambiguous logic, undocumented choices, lost rules
"Interest" Payment	Increased maintenance time	Reduced ability to safely modify system behavior

3 The Accelerants: Modern Catalysts for Contextual Debt

The accumulation of Contextual Debt is not new, but its rate has been supercharged by generative AI coding assistants and the architectural paradigm of microservices.

3.1 The AI Co-Pilot's Blind Spot: Comprehension Debt

Generative AI tools operate at the scope of code snippets and functions. They are productive precisely because they do not require—and cannot be given—a full understanding of system-wide architectural constraints, long-term business goals, or undocumented invariants. The code they generate may be locally correct but globally incoherent.

A concrete illustration: consider an AI assistant asked to implement a rate limiter for a web API. The generated code may pass all unit tests against a single thread—correctly enforcing 10 requests per minute—but silently omit the mutex or atomic counter required for thread-safe access under concurrent load. The tests pass. The code ships. The production failure arrives only under real traffic. No test caught this because the *rationale*—"this service is stateless and will run across multiple threads"—was never communicated to the AI and does not appear anywhere in the generated artifact. This is Comprehension Debt: valid syntax encoding invalid assumptions.

This failure mode is categorically different from a bug. It is not that the AI wrote incorrect code for a known requirement—it is that the AI wrote correct code for a *misunderstood* requirement. Existing static analysis tools and linters are not equipped to detect this class of error, because the error is semantic rather than syntactic.

3.2 Microservices and the Fragmentation of Domain Knowledge

Microservice architectures distribute ownership of domain logic across team boundaries and independent deployment cycles. Each service may be internally consistent while the system-wide invariants—rate limits, consistency guarantees, ordering constraints—are held only in the collective memory of the engineers who designed the original decomposition. As teams turn over and services proliferate, this distributed institutional memory degrades, leaving organizations with systems they can operate but no longer fully understand.

4 Related Work and Differentiation

Prior approaches to automated code quality assessment occupy two broad categories that are individually insufficient.

Static analysis and linting tools (e.g., SonarQube, Semgrep, Bandit) evaluate syntactic and structural properties of code against known anti-patterns. They are deterministic and fast, but they cannot evaluate whether code satisfies the *intent* behind a requirement, nor can they assess whether tests are semantically meaningful or merely pass trivially. They measure the "how" with no access to the "why."

LLM-as-judge frameworks (e.g., MT-Bench, G-Eval, and related work) use language models to evaluate code or text quality holistically. These approaches offer semantic awareness but suffer from well-documented reproducibility problems: the same LLM evaluating the same code across two independent runs can produce scores that diverge by 0.10–0.20 on a normalized scale, depending on temperature, prompt sensitivity, and model version drift.

CIS is distinguished from both categories by three properties: (1) *ground-truth anchoring*—the Testing Integrity score is derived directly from sandbox execution results, not LLM opinion; (2) *adversarial stress-testing*—an embedded attacker actively probes for vulnerabilities using Monte Carlo Tree Search rather than relying on pattern matching; and (3) *bounded LLM contribution*—the language model component can adjust scores by at most ± 0.10 from the ground-truth anchor, containing the primary source of non-determinism in existing LLM-judge approaches. The result is a hybrid evaluator that combines the reproducibility of ground-truth verification with the semantic awareness that pure static analysis lacks.

5 Formalization: The Contextual Integrity Score (CIS)

We define **Contextual Integrity (CI)** as a computable probability that a given software artifact preserves its original intent across four orthogonal dimensions: rationale alignment, architectural constraint satisfaction, empirical test validity, and logic correctness.

$$CIS_{raw} = 0.25 \cdot R(\Delta) + 0.25 \cdot A(\Delta) + 0.25 \cdot T(\Delta) + 0.25 \cdot L(\Delta)$$

$$CIS_{final} = CIS_{raw} \cdot (1 - \text{red\_penalty\_applied}) \cdot \text{intent\_penalty}$$

The four components are weighted equally at 0.25 each. This reflects an initial calibration choice subject to future empirical revision based on longitudinal failure analysis. In the absence of domain-specific weighting priors, equal weighting serves as a baseline assumption to avoid introducing implicit biases toward any single dimension of code quality. The penalty multipliers are applied post-aggregation to operate independently of the component weights.

5.1 Rationale Integrity (R): The Semantic Alignment Vector

Rationale Integrity measures whether the generated code semantically fulfills the stated requirement. Let $\vec{v}_{code}$ be the sentence embedding of the submitted source artifact and $\vec{v}_{intent}$ be the embedding of the associated task description. R is computed as:

$$R(\Delta) = \cos(\vec{v}_{code}, \vec{v}_{intent}) = \frac{\vec{v}_{code} \cdot \vec{v}_{intent}}{\|\vec{v}_{code}\| \cdot \|\vec{v}_{intent}\|}$$

Embeddings are produced via the `all-MiniLM-L6-v2` model. We acknowledge the limitations of using this specific model and cosine similarity for this task, particularly the risk of embedding collapse over long code artifacts and sensitivity to adversarial prompt padding. However, as a baseline metric for semantic alignment, it provides a necessary signal for intent-code mismatch.

5.2 Architectural Integrity (A): Constraint Validation

Architectural Integrity evaluates the Abstract Syntax Tree (AST) of the generated code against pre-defined constraint sets for the task. A begins at a baseline of 0.80 and is penalized for constraint violations: banned imports (e.g., `eval()`, network calls where prohibited), missing required patterns (e.g., recursion where mandated), and structural boundary violations. This produces a deterministic, reproducible sub-score independent of any LLM call.

5.3 Testing Integrity (T): Empirical Verification

Testing Integrity is derived from the ground-truth pass rate of code executed within an aggressively constrained, ephemeral Docker sandbox (128 MB RAM, 50% CPU quota, 50 PID limit, disabled networking). The mapping from pass rate to T score is fixed and deterministic: 100% pass rate yields $T = 0.85$; partial credit scales linearly; 0% pass rate yields $T = 0.20$. This is the single most reproducible signal in the pipeline, as it is derived entirely from execution facts rather than model inference.

5.4 Logic Integrity (L): The Senior Review

Logic Integrity is computed via an LLM-based "Senior Code Review" that evaluates the submission for edge-case handling, algorithmic complexity, and adherence to task-specific rules. To bound non-determinism, the LLM judge operates at temperature 0 with a fixed seed, and its output is constrained by a hard floor that caps the maximum downward adjustment to -0.10 (protecting the ground-truth anchor), while any upward adjustment relies on prompt-level constraints rather than a mathematical ceiling.

5.5 Penalty Multipliers

Two penalty multipliers are applied to CIS_{raw}:

1. **Red Agent Penalty:** Derived from the severity of vulnerabilities discovered by the adversarial Red Agent. Critical vulnerabilities apply a $\times$0.60 multiplier; High $\times$0.75; Medium $\times$0.85. Each vulnerability level penalizes exactly once, avoiding double-counting.

2. **Intent Penalty:** Applied when $R(\Delta)$ falls below a mismatch threshold, indicating that the submitted code addresses a fundamentally different problem than specified. This can reduce CIS_{final} by up to $\times$0.30.

6 Empirical Execution & Telemetry

The LogoMesh architecture tracks these dimensions empirically at runtime. Each evaluation produces a rigid JSON schema aggregating the four component metrics and the Red Agent's adversarial findings:

```
{
  "rationale_score": 0.85,
  "architecture_score": 0.90,
  "testing_score": 0.80,
  "logic_score": 0.75,
  "cis_score": 0.825,
  "red_penalty_applied": 0.0,
  "red_analysis": {
    "attack_successful": false,
    "vulnerability_count": 0,
    "max_severity": "low"
  }
}
```

Unlike frameworks that discard evaluation telemetry, LogoMesh ensures rigorous persistence. Evaluations are aggregated via SQLite into `battles.db`. The raw JSON payload is preserved entirely in the `raw_result` column and subsequently hashed to generate the cryptographic Decision Bill of Materials (DBOM).

7 The Protocol: Star-Topology Evaluation Architecture

To enforce the Contextual Integrity Score without introducing single-model bias, we adopted a **Star-Topology Evaluation Architecture** that structurally segregates code generation, orchestration, and adversarial auditing.

- **The Green Agent (Orchestrator/Judge):** Sits at the center of the topology. It retrieves the task, requests code from the target agent via JSON-RPC, executes the Docker sandbox, commissions the adversarial attack, and aggregates the final CIS.
- **The Purple Agent (Target/Generator):** The external service under evaluation. It receives the task specification and returns source code, unit tests, and a natural-language rationale. During infrastructure testing, this role is fulfilled by a FastAPI mock server returning structured static responses keyed on `battle_id`.
- **The Red Agent (Attacker/Auditor):** A Monte Carlo Tree Search (`MCTSPlanner`) engine embedded as a library within the Green Agent. The MCTS expands attack nodes by proposing strategic vulnerability pathways, evaluated via a modified UCB1 bandit algorithm. The Red Agent operates solely on the code artifact handed off by Green; it never communicates directly with Purple, preserving the integrity of the evaluation boundary. The MCTS engine operates with the following default hyperparameters:
  - **Max Rollout Depth:** 10 steps (variable: `max_steps`)
  - **Computational Budget:** 60.0 seconds (variable: `max_time_seconds`)
  - **Branching Factor:** 3 branches (variable: `mcts_branches`)
  - **Exploration Weight:** 1.414 ($\approx \sqrt{2}$) (variable: `exploration_weight`)

7.1 The Decision Bill of Materials (DBOM)

For every evaluation, LogoMesh generates a separate JSON artifact recording a cryptographic tuple:

$$DBOM_i = \langle H(\Delta_i),\ \vec{v}_{intent},\ \text{Score}_{CIS},\ \sigma_{judge} \rangle$$

where $H(\Delta_i)$ is the SHA-256 hash of the evaluation result, $\vec{v}_{intent}$ is the semantic intent vector, and σ_{judge} is a simulated signature. This artifact is stored on the filesystem and linked to the SQLite record in `battles.db`.

The system persists the full evaluation payload in the `raw_result` column along with its hash. This structure provides a basis for *output-layer* auditability: it establishes a record that can be checked for consistency between the reported score and the underlying data, contingent on consistent JSON serialization.

It is important to be precise about the trust boundary this establishes. The DBOM documents that a specific score was generated for a specific result payload at a specific time. It does not, by itself, attest to the integrity of the inputs to the evaluation pipeline—the task specification, the Purple Agent's response, or the sandbox execution environment. The DBOM functions as a rigorous audit trail for compliance purposes, providing defensible documentation of the evaluation process.

7.2 Phase 2 Roadmap: Cryptographic Assurance

To move from passive auditability to active tamper-evidence, the Phase 2 research roadmap prioritizes the following cryptographic enhancements:

1. **Automated Integrity Verification:** Development of a verification harness that actively re-computes hashes for stored records, enforcing strict JSON serialization standards to eliminate false positives caused by key reordering.

2. **Input Entanglement:** Extension of the DBOM to include hashes of the input task specification and the target agent's initial response, preventing "input poisoning" attacks where the evaluation context is altered prior to execution.

3. **Merkle Chaining:** Implementation of a Merkle tree structure linking sequential evaluations. This would ensure that any deletion or modification of a historical record invalidates the cryptographic chain of all subsequent evaluations, providing a guarantee of log immutability.

8 Conclusion & Open Research Invitation

The accumulation of Contextual Debt is a measurable source of software liability in an era where AI coding assistants can produce syntactically valid, semantically incoherent code. As legal standards for software liability evolve, organizations that cannot account for the *why* behind their code will face increasing challenges. The DBOM, as an evaluator-generated artifact cryptographically bound to each code assessment, is a concrete step toward a defensible evidentiary record.

The LogoMesh architecture, validated by its Phase 1 victory in the Berkeley RDI AgentBeats Competition, demonstrates that Contextual Debt can be quantified, bounded, and adversarially stress-tested using a multi-agent pipeline with reproducible, ground-truth-anchored scoring. This result serves as external stress validation of the architecture under competitive conditions. The benchmark's variance of less than 0.05 across identical runs—achieved by anchoring to execution facts rather than model opinion—establishes a baseline for reproducibility in LLM-assisted code evaluation.

Phase 2 Research Handoff: The LogoMesh evaluation framework has been transitioned to an open-source research initiative. For the Phase 2 AgentBeats sprints, we are coordinating a distributed strike team across two tracks: the **Lambda Track**, which retargets the MCTS engine from AST mutation to adversarial NLP prompt injection; and **Purple Agent development**, focused on zero-shot domain adaptation to novel task categories. We formally invite the academic and open-source security communities to collaborate. Tryout issues, architectural discussions, and the core repository are available at:

https://github.com/LogoMesh/LogoMesh/labels/phase-2-tryout