Boundary as an Execution-Time Primitive for AI-Assisted Software Development Governance

> Why AI-Assisted Development Fails Before Governance Begins

Author: Spark Tsai
ORCID: https://orcid.org/0009-0006-8847-4703
Email: spark.tsai@gmail.com
Date: February 2026

Abstract

Loss of control, behavioral drift, and non-auditability in AI-assisted software development are commonly attributed to model misalignment, hallucination, or insufficient guardrails. This paper argues that such diagnoses overlook a fundamental category distinction.

We distinguish between **model alignment boundaries**, established at training time through data distributions, RLHF, and safety fine-tuning, and **task execution boundaries**, which must be explicitly constructed at execution time for a specific engineering task. While the former provides general, statistical safety tendencies, it does not—and cannot—automatically inherit the concrete, task-specific constraints required for engineering governance.

We show that many widely reported failures, including insecure yet functional code generation, arise not from deficient model alignment but from the absence of a **decidable task execution boundary** at runtime. When such a boundary is missing, drift and violation become epistemically undecidable, and model preferences fill the resulting vacuum.

We formalize task execution boundaries as the resolution of visible scope and explicit prohibitive constraints, introduce boundary evidence as the minimal auditable unit, and demonstrate through engineering scenarios that governance mechanisms operating without this primitive rest on interpretive rather than decidable foundations.

Keywords

AI Governance, AI Accountability, Software Engineering Governance, AI-Assisted Software Development, LLM Agents, Task Execution Boundary, Model Alignment, Boundary Evidence, Decidable Governance, Prohibitive Constraints, Task-Specific Constraints, Execution-Time Constraints, Category Error, Behavioral Drift,
Decision Provenance, Insecure Code Generation, Auditability, Technical Accountability, Prompt Engineering,
AI Safety, Code Generation, Runtime Governance

1. Introduction

AI-assisted software development has rapidly progressed from code completion to autonomous agents capable of modifying production systems. In response, governance efforts have largely focused on improving **model alignment**: safer training data, stronger RLHF, and increasingly sophisticated guardrails.

These efforts resemble attempts to build ***safer engines***. Yet many engineering failures persist—not because the engine is unsafe, but because the road has no speed limits.

This paper argues that a critical distinction has been overlooked between two orthogonal categories of boundaries:

- **Model Alignment Boundaries (The "Safer Engine"):** These are statistical, soft constraints learned during training. They ensure the "car" doesn't explode or intentionally crash. However, a safe engine alone doesn't know the difference between a highway and a sidewalk—it is inherently probabilistic and context-agnostic.
- **Task Execution Boundaries (The "Road Constraints"):** These are explicit, decidable, hard constraints constructed per task. Just as a driver needs to know the specific lane, direction, and speed limit of a particular street, an AI agent needs a defined action space (e.g., *"MUST NOT modify files outside of* `src/auth` *"*).

We demonstrate that **model alignment boundaries do not—and cannot—automatically inherit task-specific constraints.** Even the most "aligned" AI, like a high-performance car with the best brakes, can still drive the wrong way down a one-way street if the **Task Execution Boundary** is missing.

2. From Narrative to Technical Governance

Without an explicitly constructed boundary, governance collapses into interpretation. When an AI agent drifts, the post-mortem becomes an argument over "intent" rather than a verification of "fact." Accountability becomes **narrative** (explaining what went wrong) rather than **technical** (proving what was violated).

This paper proposes a framework to bridge this gap, transforming general model safety into a decidable, evidence-based engineering governance model.

3. The Formalism of Task Execution Boundaries

To transition from narrative governance to technical accountability, we must define the primitives that constitute a boundary. We argue that a boundary is not a static property of a model, but a dynamic resolution of scope and constraints at execution time.

3.1 Visible Scope

Visible scope refers to the set of artifacts accessible to the model at execution time, such as specifications, tests, source code, explicit goals, and context.
Scope defines **what the model can observe**, not what it is permitted to do.
Scope alone does not bound behavior; it merely sets the possible action space.

3.2 Constraints as Prohibitive Primitives

A critical distinction exists between prescriptive goals (MUST) and prohibitive constraints (MUST NOT).

- **MUST** (Prescriptive): Expresses semantic completeness (e.g., "The code must be efficient"). Verifying "completeness" is computationally expensive and often involves subjective interpretation.
- **MUST NOT** (Prohibitive): Expresses safety invariants (e.g., "The code must not hardcode secrets"). Prohibitions are **decidable**; they provide a clear, binary pass/fail condition that can be checked at runtime without semantic ambiguity.

In our framework, **decidability in governance emerges primarily from prohibitive constraints.**

3.3 Boundary as Execution-Time Resolution

The task execution boundary is the set of actions that the model is both capable of generating (given what it can see) and explicitly permitted to take (given what it is forbidden from doing) in a specific task instance. In other words, it is the effective, permissible action space at execution time—neither the full generative possibility nor a vague safety guideline, but a concrete, decidable subset shaped by the interplay of visibility and hard prohibitions.

We formalize the Task Execution Boundary as:

$$B(t) = \{a \mid a \in Possible(S_t) \land \forall c \in C, \neg c(a)\}$$

- t — Execution time (a specific task instance).
- S_t — Visible Scope at time t: the set of artifacts accessible to the model during this execution (e.g., specific code files, API specifications, test suites, explicit goals, or context). Scope determines what the model *can observe*, but not what it is *permitted to do*.
- C — Prohibitive Constraints: the set of explicit MUST NOT predicates that define the invariants and "no-go zones" of the engineering task.
- a — An individual action or output generated by the model.
- $Possible(S_t)$ — The set of all possible actions that the model could generate given the visible scope S_t (the unrestricted generative space).
- $B(t)$ or B_t — The Task Execution Boundary: the resolved, permissible action space at time t.

A critical corollary follows directly from the definition:

$$\text{If } S_t = \emptyset \text{ or } C = \emptyset, \text{ then } B(t) = \bot$$

When $B(t) = \bot$ (boundary non-existence), the AI agent operates in a **boundary vacuum**. In this state:

- No task-specific governance boundary constrains the generative action space.
- Model-level statistical preferences (from training-time alignment) become the dominant determinant of output.
- Drift, insecure code, or unintended behavior is not a deviation from a boundary—it is the default outcome of boundary absence.
- Governance claims become epistemically undecidable: one cannot objectively determine whether an action is a violation, because no explicit permissible space was ever defined.

This explains why even highly "aligned" models frequently produce insecure yet functional code (e.g., hardcoded secrets or weak cryptography): they follow general probabilities because no task-specific boundary was constructed to override them.

3.4 Boundary Evidence

Boundary evidence records the resolved boundary state at execution time. It represents the minimal auditable unit required for reproducibility and governance.

```yaml
execution_record:
  id: "exec_unique_id"
  timestamp: "2026-02-06T14:30:00Z"
  scope:
    goals: ["Replace auth logic with JWT"]
    specifications: ["docs/auth_spec.md"]
    tests: ["tests/auth_test.py"]
    source_code: ["src/auth_manager.py"]
  constraints:
    - "MUST NOT modify files outside /src"
    - "MUST NOT hardcode cryptographic secrets"
    - "MUST NOT introduce new external dependencies"
  output_hash: "sha256:..."
```

This representation is illustrative and does not prescribe a specific storage format or governance system.

4. Model Boundary vs. Task Boundary

4.1 Model Alignment Boundary (Training Time)

- Established during training
- Statistical, soft constraints
- Cross-task generalization
- Example: “Avoid generating malware”

4.2 Task Execution Boundary (Execution Time)

- Constructed per execution
- Explicit, hard constraints
- Task-specific
- Example: “Do not modify tests/”

Model boundaries reduce the likelihood of harmful outputs. They do not define permissible engineering actions.

5. Engineering Scenarios

We examine three scenarios involving modification of an existing login system to use JWT authentication.

5.1 Scenario A: Prompt Only

- Model Alignment Boundary: Present
- Task Execution Boundary: $\perp$

Outcome: Functional but insecure JWT implementation (e.g., hardcoded secrets, weak algorithms). These outputs violate implicit engineering expectations but not model alignment boundaries.

5.2 Scenario B: Prompt + SDD

- Model Alignment Boundary: Present
- Task Execution Boundary: Partially formed

Outcome: Improved structural coherence but persistent insecure patterns. Functional specifications do not constrain security decisions without explicit prohibitions.

5.3 Scenario C: Prompt + SDD + Explicit Constraints

- Model Alignment Boundary: Present
- Task Execution Boundary: Resolved

Explicit constraints such as “MUST NOT hardcode secrets” and “MUST NOT accept ‘none’ algorithm” confine generation to an auditable subspace. Violations become decidable.

6. Discussion

6.1 Drift as an Epistemic Illusion

Behavioral drift is commonly treated as a stochastic property of model behavior. We argue that in the absence of a task execution boundary, drift is **epistemically undefined**. Without a formal boundary, there is no reference baseline against which deviation can be measured. In such cases, apparent drift reflects subjective human reinterpretation of results rather than an objective system violation.

6.2 Limitations and Future Work

This paper focuses on the static resolution of boundaries at execution time. We deliberately exclude dynamic boundary evolution, enforcement mechanisms (e.g., runtime interception), and versioned lifecycle governance. These concerns, while critical, presuppose the existence of a decidable task execution boundary as defined here.

6.3 Scope of Definition

The primitives defined in this work—visible scope and explicit prohibitive constraints—represent the minimal set of conditions required to construct a decidable boundary. While additional primitives may emerge as AI agents evolve, we establish a necessary foundation: without at least these elements, task-level governance remains undecidable by design.

7. Conclusion

This paper argues that many failures in AI-assisted software development stem from a fundamental misdiagnosis. The problem is not merely insufficient model alignment, but the **absence of a decidable task execution boundary at runtime**.

Model alignment boundaries and task execution boundaries are orthogonal; conflating them constitutes a **category error** that undermines the validity of AI governance. By formalizing these boundaries and introducing **boundary evidence** as the minimal auditable unit, this work reframes AI governance as an execution-time engineering problem rather than a training-time alignment deficiency.

8. Related Work

8.1 Behavioral Drift and Context Degradation

Recent studies document various forms of behavioral degradation in LLM-based agents, including semantic drift, goal drift, coordination drift, and context degradation over extended interactions [1–5]. These works provide valuable empirical observations and detection techniques.

However, drift is typically treated as a detectable violation of an assumed boundary. Few works examine the prior engineering condition: whether a decidable boundary existed at execution time in the first place.

8.2 Guardrails and Policy Enforcement

Guardrail frameworks and policy-based enforcement mechanisms aim to constrain undesirable outputs via filtering, rejection, or runtime checks [6–9]. In AI-assisted coding, multiple studies report the generation of insecure yet functional code, including hardcoded secrets and weak cryptographic practices [10].

These mechanisms operate post-generation and implicitly assume that a permissible action space has already been defined.

8.3 Agent Frameworks and Context Management

Surveys of LLM-based agents for software engineering emphasize tool orchestration, memory isolation, and context delivery [11–13]. The Model Context Protocol (MCP) standardizes context transport and tool access [14,15].

While these frameworks improve infrastructure-level governance, they do not define the decidable boundaries of the generative action space itself.

8.4 Model Alignment Boundary vs. Task Execution Boundary

A substantial body of work addresses model alignment boundaries through training-time techniques such as RLHF and safety fine-tuning. These establish statistical, soft constraints applicable across tasks.

This literature often assumes that strong model alignment subsumes task-specific execution boundaries. We argue that this is a category error.

Model alignment boundaries and task execution boundaries differ in origin, scope, and nature. They may have little or no intersection in practice. Training-time safety does not imply execution-time governance.

References

- [1] Abhishek Rath et al. "Agent Drift: Quantifying Behavioral Degradation in Multi-Agent LLM Systems Over Extended Interactions." arXiv:2601.04170, 2026.
- [2] Lin Chen et al. "AI Agent Behavioral Science." arXiv:2506.06366v2, 2025.
- [3] Technical Report: "Evaluating Goal Drift in Language Model Agents." arXiv:2505.02709, 2025.
- [4] Adnan Masood. "Agent Drift: the reliability blind spot in multi-agent LLM systems." Medium, 2026.
- [5] Various authors. Studies on context degradation and user behavior/data drift in LLMs (e.g., Deepchecks reports, 2024–2025).
- [6] Y. Dong et al. "Safeguarding Large Language Models: A Survey." arXiv:2406.02622, 2024. (Also published in Artificial Intelligence Review, 2025).
- [7] "Building Guardrails for Large Language Models." arXiv:2402.01822, 2024.
- [8] "SoK: Evaluating Jailbreak Guardrails for Large Language Models." arXiv:2506.10597, 2025.
  [9] "A Flexible Large Language Models Guardrail Development Methodology Applied to Off-Topic Prompt Detection." arXiv:2411.12946, 2024 (v2 2025).
- [10] "When Developer Aid Becomes Security Debt: A Systematic Analysis of Insecure Behaviors in LLM Coding Agents." arXiv:2507.09329, 2025.
- [11] Junwei Liu et al. "Large Language Model-Based Agents for Software Engineering: A Survey." arXiv:2409.02977, 2024. (Accepted by TOSEM).
- [12] "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future." arXiv:2408.02479v2.
- [13] "A Survey on Code Generation with LLM-based Agents." arXiv:2508.00083, 2025.
- [14] Anthropic. "Introducing the Model Context Protocol (MCP)." Anthropic Announcements, November 2024. https://www.anthropic.com/news/model-context-protocol
- [15] Model Context Protocol official documentation and subsequent engineering blogs (Anthropic, 2024–2025). https://modelcontextprotocol.io/