The Case for Compiled Execution

Opus Experiment O-02: Execution Properties in High-Stakes AI Systems

Nesean Crofford

Abstract

As large language models move from conversational interfaces into operational infrastructure, a structural question emerges: can runtime inference serve as the execution substrate for systems that must be correct, deterministic, temporally faithful, and governable? This paper examines that question through a controlled comparison of 3,900 executions across five system architectures, three operational domains, and four execution properties.

The central finding is not simply that one system performs better on average. It is that the relevant execution properties do not emerge reliably under runtime inference. Compiled execution (Opus) achieves 100% decision correctness, 0% temporal misbinding, perfect determinism, and zero runtime-induced failures across 780 cells. No inference-based system, including a structured-output stack with validation and retries, multi-role agent orchestration, and agent-generated deterministic artifacts, matches these properties on any axis.

The contribution of this paper is therefore architectural as much as empirical. The observed failures are not best understood as bugs awaiting stronger models or more careful prompting. They arise from using probabilistic inference in a role that demands deterministic execution. The results suggest that as AI systems move deeper into governed operational settings, they increasingly benefit from an execution layer with different properties from the reasoning layer above it.

1 Introduction

Every computational abstraction eventually encounters a boundary where its native strengths no longer align with the requirements placed upon it. Interpreted systems are flexible, expressive, and easy to adapt, but they yield ground when predictability, reproducibility, and structural control become central. In those settings, the relevant question is not what can be expressed, but what can be relied upon.

AI systems are approaching an analogous boundary. Large language models are highly capable reasoning engines. They summarize, synthesize, translate, and generate with increasing sophistication. But when these models are placed inside operational workflows, processing claims, adjudicating credit, enforcing policy, or producing auditable artifacts, they are asked to do something more specific than reason. They are asked to execute.

Execution systems have requirements that reasoning quality alone does not satisfy. This paper studies four of them:

1. **Correctness**: outputs must exactly match the intended operational decision.
2. **Determinism**: identical inputs must produce identical outputs.

3. **Temporal fidelity**: systems must bind to the correct historical signal or rule version, not infer it opportunistically.
4. **Governance**: systems must support replay, diff, attribution, and reporting in a verifiable manner.

These are not aspirational properties. They are baseline expectations for software systems whose outputs feed payments, compliance records, audit trails, and downstream automation. The central question, then, is whether runtime inference is the right execution mechanism for environments where these properties are non-negotiable.

The results reported here suggest that it is not.

1.1 What is being argued

The argument of this paper is narrower than a blanket critique of language models, but broader than a benchmark comparison. It is not that one system wins a leaderboard. It is that the observed gap between inference-based execution and operational requirements appears structural rather than incidental.

The analogy to compiled programming languages is useful because it highlights a difference in runtime properties rather than a difference in intelligence. Compilation did not matter because it made source programs "smarter." It mattered because it introduced a representation that was predictable, optimized, and structurally analyzable. In the same way, compiled execution matters here because it executes with different guarantees.

The experiment does not prove that compiled execution solves every operational problem. It does show that several failure modes of inference-based execution are not reduced to zero by stronger scaffolding, lower temperature, or more elaborate orchestration, while a compiled alternative removes them from the runtime path.

A useful way to read the results is as a distinction between two layers that are often conflated in practice: a *reasoning layer*, where inference is highly effective for interpretation, synthesis, and authoring, and an *execution layer*, where stronger guarantees may be required. The question explored here is whether runtime inference can satisfy both roles at once in governed operational settings.

1.2 From O-01 to O-02

Experiment O-01 established that language models are unreliable execution systems. Across 1,260 runs on 12 deterministic policy scenarios, no LLM exceeded a 25% exact-match pass rate, while a deterministic reference system achieved 100%. The dominant failure mode was not incorrect reasoning but execution non-conformance: models that often understood the task still could not reliably produce exact operational artifacts.

Experiment O-02 extends that work in three directions:

1. **Beyond correctness**: O-02 introduces temporal binding and governance as first-class measurement axes, moving from "can the system get the right answer?" to "can the system get the right answer, under the right rule version, with the ability to explain why?"

2. **Broader system comparison**: O-02 evaluates five system architectures spanning the full spectrum from raw inference to compiled execution, including the production patterns practitioners actually deploy.

3. **Architectural framing**: O-02 is designed not merely to show that inference fails in particular ways, but to examine what kind of execution representation best satisfies those requirements.

1.3 Contributions

1. An execution-oriented evaluation framework measuring four operational properties, correctness, determinism, temporal fidelity, and governance, rather than conversational quality or benchmark accuracy.

2. Empirical evidence from 3,900 runs that the gap between inference-based execution and operational requirements is structural rather than marginal.

3. A combined failure surface analysis showing that compiled execution removes multiple runtime-induced failure categories from the execution path.

4. Evidence that determinism alone is insufficient: agent-coded Python recovers correctness but not governance, demonstrating the importance of formal execution representations.

5. A concrete basis for thinking about execution and reasoning as distinct layers in AI-native operational systems.

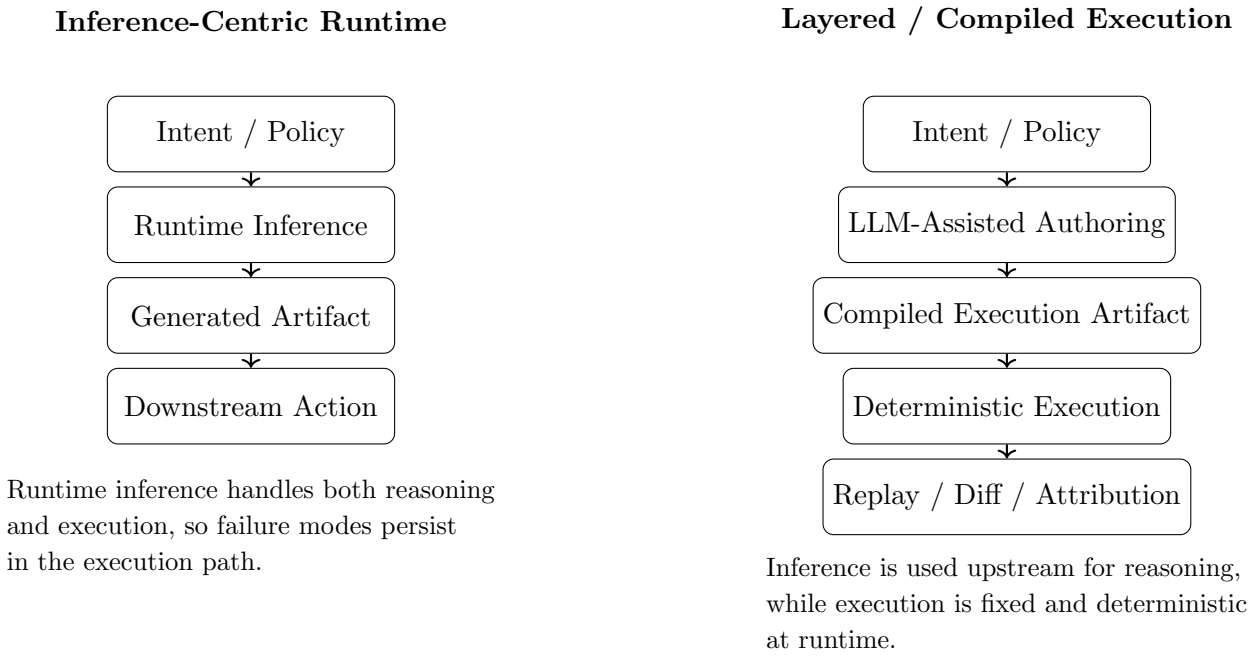

Figure 1: Two ways to position language models in an operational system. On the left, inference remains in the runtime execution path. On the right, inference is used upstream, while execution is performed through a compiled representation with fixed runtime behavior.

2 Related Work

2.1 Capability Benchmarks

Language model evaluation has historically focused on capability: MMLU (Hendrycks et al., 2021), BIG-Bench (Srivastava et al., 2022), and subsequent reasoning benchmarks measure whether models can answer questions correctly. These benchmarks are valuable, but they do not evaluate whether models can function as execution systems. A model that scores highly on a reasoning benchmark may still fail as an execution substrate, because execution imposes requirements, determinism, conformance, temporal binding, and replayability, that capability benchmarks do not measure.

2.2 Agent Benchmarks

Agent-oriented evaluations such as AgentBench (Liu et al., 2023) and related tool-use benchmarks measure whether models can complete multi-step tasks. These are closer to execution evaluation, but they still focus primarily on task completion rather than artifact reliability. A system that completes a task with a "close enough" output can perform well on an agent benchmark while still failing in operational settings where exact outputs are required and downstream systems are intolerant to approximation.

2.3 Structured Output and Validation

JSON modes, function calling, and structured output frameworks improve format compliance. Production stacks that combine these tools with validation and retries are common patterns for deploying language models in structured settings. This paper evaluates whether those patterns are sufficient for high-stakes execution. The results suggest that they improve output regularity without recovering the deeper properties required of an execution substrate.

2.4 Deterministic and Rule-Based Systems

Rule engines, decision tables, and policy DSLs have long provided deterministic execution for structured decisions. What is distinctive in the present framing is not the existence of deterministic logic itself, but the role it appears to play in AI-native systems. The relevant distinction is no longer between "AI" and "rules," but between systems that infer their way through execution at runtime and systems that transform human-readable intent into stable, inspectable execution artifacts before runtime.

2.5 Alternative LLM-Runtime Architectures

Recent practice has begun to explore more disciplined uses of language models at runtime, including constrained decoding over typed schemas, retrieval-bound context for temporal grounding, and state-machine wrappers that limit where inference is applied. These approaches reduce certain classes of failure relative to unconstrained generation.

However, they retain a common property: execution artifacts are still produced through runtime inference. As a result, they may reduce certain risks materially relative to unconstrained generation

while still preserving non-zero instability, potential temporal ambiguity, and governance limitations tied to generated rather than structural representations. The distinction explored in this paper is therefore not between naive prompting and better prompting, but between inference as the execution mechanism and compilation into a deterministic execution representation.

3 Experiment Design

3.1 Systems Under Test

Five system architectures spanning the full spectrum from compiled execution to raw inference:

System	Architecture	LLM at Runtime	Key Property
Opus	Compiled execution modules	No	Deterministic by construction
Raw LLM	Direct prompt	Yes	No scaffolding
Structured + Validation Stack	Structured output + retries	Yes	Format-enforced
Agent System	Multi-role orchestration	Yes	Planner / executor / validator
Agent-Coded Python	Frozen generated code	No	Deterministic artifact

Table 1: Five system architectures evaluated in O-02. All LLM-based systems use Claude Sonnet 4 at temperature=0.

These five systems were chosen to span the design space practitioners actually consider:

- **Opus** compiles policy logic into deterministic execution artifacts. At runtime, execution is a pure function over a frozen execution artifact, a versioned state snapshot, and the input payload. No LLM is involved at execution time.
- **Raw LLM** represents the simplest possible deployment: a single prompt with no structural enforcement.
- **Structured + Validation Stack** represents a common production pattern: structured JSON output, schema validation, and up to 3 retry attempts on structural failure.
- **Agent System** represents the frontier of LLM orchestration: planner, executor, assembler, and validator roles with up to 10 orchestration steps.
- **Agent-Coded Python** represents a hybrid approach: LLM-generated code frozen at build time and executed deterministically at runtime, removing inference from the execution path but retaining opaque execution representation.

3.2 Domains

Three operational policy domains, each with a v1 and v2 policy version representing a temporal signal change:

Domain	Policy Change (v1 → v2)	Output Schema
Refund	30-day → 14-day return window	{decision, refund_amount, reason_code}
Insurance	Waiver threshold above \$10K → above \$15K	{decision, payout_amount, reason_code}
Credit	Score threshold $\geq 680 \rightarrow \geq 700$	{decision, credit_amount, reason_code}

Table 2: Three operational domains with versioned policy changes for temporal binding measurement.

The version changes are designed to test temporal binding: can a system correctly apply the *right version* of a policy to the *right input*? This requirement matters in any regulated environment where policies change over time and historical decisions must be reproducible under the rules that were in effect when they were made.

3.3 Measurement Tiers

52 frozen scenarios across three tiers:

Tier	What It Measures	Scenarios	Key Metric
Tier 1	Decision correctness	19	Exact-match pass rate
Tier 2	Temporal binding	30 (11 signal-dependent)	Misbinding rate
Tier 3	Governance capability	3	Authority, completeness, precision

Table 3: Three measurement tiers isolating distinct execution properties.

3.4 Run Matrix and Controls

Dimension	Value
Systems	5
Scenarios	52 (19 T1 + 30 T2 + 3 T3)
Domains	3 (refund, insurance, credit)
Runs per cell	15
Total runs	3,900
Temperature	0.0 (fixed, all LLM systems)
Manifest body hash	`2757fbbd...` (identical in pre-O-02 calibration runs and formal execution)
Lock validation checks	7 (all passed pre- and post-execution)
Environment drift	None detected

Table 4: Experiment parameters. The experiment surface is hash-locked and validated before and after execution.

Every scenario payload is content-addressed. Manifest integrity is verified through a 7-gate validation process covering scenario freeze, adapter freeze, dataset freeze, scoring freeze, manifest hashes, locked registry, and mutation detection. All LLM runs are live API calls with evidentiary enforcement. No stubs and no simulation layers were used.

4 Results

4.1 Decision Correctness (Tier 1)

System	Passed	Total	Pass Rate
Opus	**285**	**285**	**100.0%**
Agent-Coded Python	**285**	**285**	**100.0%**
Structured + Validation Stack	246	285	86.3%
Agent System	35	285	12.3%
Raw LLM	30	285	10.5%

Table 5: Tier 1 exact-match decision correctness. Systems that remove LLM inference from the execution path (Opus, Agent-Coded) achieve 100%. Systems that retain inference at runtime do not.

The divide is immediate: systems that execute deterministic artifacts (Opus, agent-coded Python) achieve perfect correctness, while systems that rely on runtime inference do not, regardless of scaffolding sophistication.

The structured + validation stack achieves 86.3%, the highest among inference-based systems, through structured output enforcement and retries. But 39 failures remain: 15 wrong decisions and 24 wrong reason codes. The retries close the format gap; they do not close the reasoning gap. The critical distinction is that structural regularity and operational correctness diverge sharply once inference remains in the runtime path.

The agent system performs far worse. Despite multi-role orchestration and a 10-step budget, it achieves only 12.3% exact-match correctness. This result matters because it shows that more elaborate inference at runtime does not necessarily improve execution behavior. In this setting, it often compounds the problem.

4.2 Temporal Binding (Tier 2)

Temporal binding measures whether a system applies the correct policy version to each input. Only signal-dependent scenarios, where the v1 to v2 policy change affects the expected output, are scored.

System	Correct	Scoreable	Correctness	Misbinding
Opus	**165**	**165**	**100.0%**	**0.0%**
Agent-Coded Python	**165**	**165**	**100.0%**	**0.0%**
Structured + Validation Stack	76	79	96.2%	3.8%
Agent System	21	72	29.2%	70.8%
Raw LLM	0	58	0.0%	100.0%

Table 6: Tier 2 temporal binding on signal-dependent scenarios. Misbinding rate is the fraction of scoreable runs where the system used the wrong policy version. Scoreable counts vary by system because runs that fail before producing a usable artifact cannot be evaluated for temporal binding.

The temporal binding gradient is monotonic and exposes a deeper property of the execution mechanism:

- **Structural systems** (Opus, agent-coded): 0% misbinding. The temporal signal is embedded in the execution artifact, the compiled artifact or frozen code, not resolved inferentially at runtime.
- **Structured + validation stack**: 3.8% misbinding. Structured output and retries can improve artifact regularity, but they do not guarantee that the system has resolved the correct historical state before producing the output.
- **Agent system**: 70.8% misbinding. Multi-role orchestration does not repair temporal confusion; in this setting it substantially worsens it.
- **Raw LLM**: 100% misbinding. Every scoreable run uses the wrong policy version.

This gradient suggests that temporal fidelity is not best understood as a prompt-quality issue or a general reasoning issue. It is a property of whether the temporal signal is *bound structurally* into execution. When it is, misbinding is zero. When it is not, misbinding remains non-zero regardless of how much scaffolding surrounds the inference call.

Figure 1: Temporal misbinding gradient

Temporal Misbinding Rate (Signal-Dependent Subset)

Opus	0.0%	← structural binding
Agent-Coded Python	0.0%	← frozen artifact
Structured + Validation Stack	3.8%	
Agent System	70.8%	
Raw LLM	100.0%	← pure inference

The gradient is monotonic from structural to inferential. No amount of output scaffolding reduces misbinding to zero.

Figure 2: Temporal misbinding increases as systems move from structural binding toward pure inference.

4.3 Output Determinism

System	Mean Stability	Min	Inconsistent	Assessment
Opus	**1.000**	**1.000**	**0/52**	Deterministic by construction
Agent-Coded Python	0.942	0.600	3/52	Near-deterministic
Structured + Validation Stack	0.864	0.133	25/52	Moderate instability
Raw LLM	0.663	0.133	39/52	Significant instability
Agent System	0.499	0.133	49/52	Near-random

Table 7: Output hash stability across 15-run cohorts. Inconsistent = scenarios where stability $<$ 1.0.

Opus produces identical output on every invocation across all 52 scenarios (780 runs). This follows directly from the execution model: $f(\text{artifact}, \text{snapshot}, \text{input}) = \text{output}$. Same function, same inputs, same output.

The agent system is the most unstable: 49 of 52 scenarios produce different outputs across 15 identical runs, with mean stability of 0.499. A deployment of this system would produce a different decision roughly half the time on repeated identical inputs.

Temperature is fixed at 0.0 for all LLM-based systems, the fairest possible setting. Despite this, output stability ranges from 0.499 to 0.864. Temperature=0 selects the highest-probability token at each position, but the inference process is not a pure function. For execution systems, "usually the same" is not determinism.

The agent-coded Python result is also instructive. Its 0.942 stability reflects minor output variation in governance scenarios rather than variation in the underlying decision itself. That distinction matters later: deterministic artifacts can recover correctness and temporal fidelity, but not necessarily the governance properties associated with a formal execution representation.

4.4 Combined Failure Surface

System	Taxonomy	Wrong Dec.	Mis-binding	Incon-sistent	Observed Surface	Label
Opus	**0**	**0**	**0**	**0/52**	**0**	**Clean**
Agent-Coded	0	0	0	3/52	3	Contained
Structured + Validation Stack	39	15	3	25/52	82	Moderate
Raw LLM	255	Not separable	58	39/52	352+	Severe
Agent System	250	211	51	49/52	561	Severe

Table 8: Combined failure surface merging taxonomy failures, post-scoring observations, and determinism. Opus has a clean surface; every other system has non-zero failures.

The failure surface is the experiment's primary evidence artifact. It merges three views:

1. **Taxonomy failures**: malformed output, schema violations, enum errors, decision errors (recorded

per-run during execution).

2. **Post-scoring observations**: wrong decisions (from T1 scoring), temporal misbinding (from T2 scoring).

3. **Determinism failures**: scenarios where output varies across 15 identical runs.

Opus is the only system with a clean failure surface: zero failures across all categories, all 780 cells. This is not simply strong performance. It is a qualitatively different runtime profile.

The structured + validation stack still exhibits a moderate failure surface despite eliminating schema failures. This distinction is important. The system looks operationally improved if one watches only format-level metrics, but the broader surface shows that the execution problem remains unresolved.

4.5 Failure Elimination

Category	Opus	Present In	Material (>5%)?
enum_value_invalid	0	raw_llm (255)	No
wrong_decision	0	structured + validation stack (15), agent (211)	Yes (>5%)
wrong_reason_code	0	structured + validation stack (24)	No
wrong_amount	0	agent_system (38)	No
malformed_output	0	agent_system (1)	No

Table 9: Failure categories removed from the runtime execution surface under compiled execution.

The absence of these failures in Opus is not only empirical (0/780 observed), but reflects how the execution model is structured. Runtime inference is not used to produce execution artifacts, and therefore the associated stochastic failure modes do not appear in the execution path.

It is important to clarify what this means. Failures are not eliminated from the system as a whole. Rather, they are relocated. Instead of appearing during runtime execution as stochastic artifacts (e.g., invalid enums, incorrect decisions, malformed outputs), they move to earlier stages such as specification, compilation, or snapshot definition. These stages are static, inspectable, and testable in ways that runtime inference is not.

The distinction is therefore not between "no failures" and "failures," but between:

- failures emerging dynamically at runtime through probabilistic generation, and
- failures emerging in pre-runtime artifacts that can be validated, versioned, and governed.

This relocation of failure modes is a key difference in execution profile.

5 Execution Requirements and Architectural Implications

5.1 Why "Better Models" Is Not the Complete Answer

The failure patterns observed in this experiment do not behave like ordinary capability gaps. They align more closely with properties of the execution mechanism itself:

- **Non-determinism** persists at temperature=0 because inference is not a pure function. Stronger models may reduce variance in practice, but do not eliminate it.
- **Temporal misbinding** occurs because the system infers context rather than binding to versioned data. Improvements in reasoning do not, by themselves, introduce structural binding.
- **Governance instability** arises because explanations are generated rather than derived from formal representations. This limits their verifiability.

These observations suggest that some execution properties may not reliably emerge from runtime inference alone, even as model capability improves. This does not imply that inference is ineffective, but that it may be better suited to upstream reasoning and authoring roles than to final execution in settings where strict guarantees are required.

5.2 The Compilation Analogy

The history of computing offers a useful analogy. Early software was often interpreted: source representations were read and executed at runtime, with all the flexibility and unpredictability that implies. Compiled languages emerged when the reliability demands of systems software exceeded what interpretation could provide. The compiler did not make the source program smarter. It transformed it into a representation with different runtime properties: deterministic, optimized, and structurally analyzable.

Compiled execution for AI-native systems follows the same logic:

Property	**Interpreted / Inference**	**Compiled / Execution**
Output	Probabilistically generated	Deterministically computed
Temporal context	Inferred from prompt	Bound to versioned snapshot
Governance	Generated narrative	Structural diff + attribution
Reproducibility	Approximate (temp=0)	Exact (pure function)
Failure surface	Non-zero, multi-dimensional	Runtime-induced failures removed from path

Table 10: The interpreted/compiled analogy applied to AI execution.

The claim is not that compiled execution is novel in all of software, but that it appears to be a missing layer in AI-native operational systems. Inference is highly valuable upstream: for understanding intent, authoring policy, synthesizing rules, and generating specifications. The question is where it belongs in the stack. The evidence here suggests it does not belong at the final execution boundary when correctness, determinism, temporal fidelity, and governance are required simultaneously.

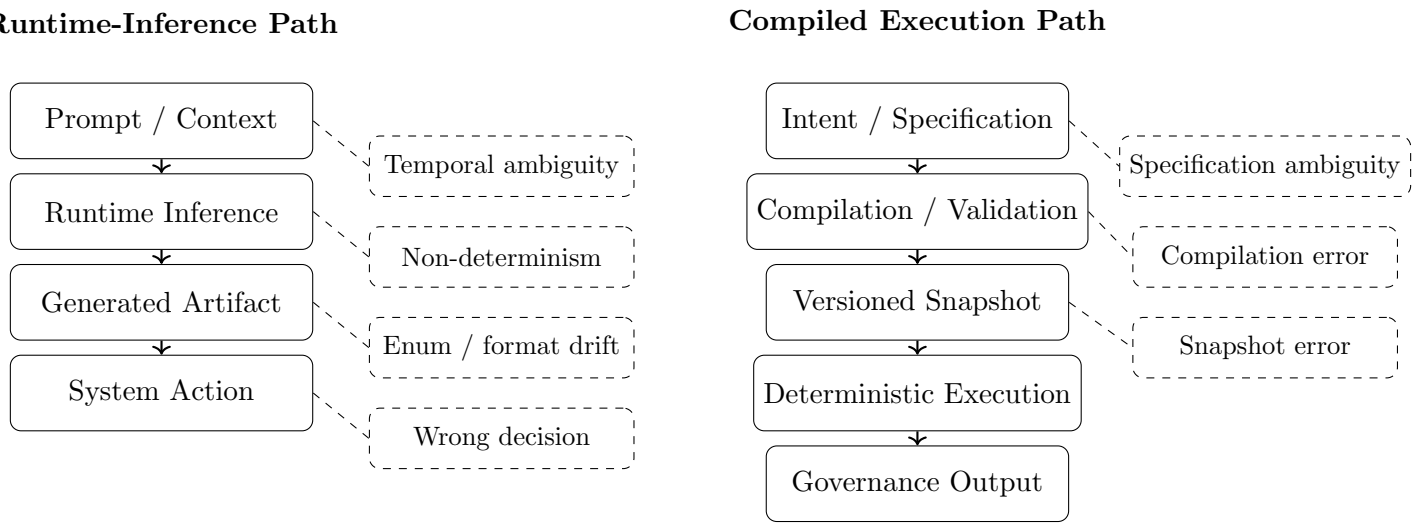

Figure 3: Location of failure modes under different execution models. In inference-based systems, failures emerge during runtime execution. In compiled execution, failures are shifted to pre-runtime artifacts where they can be validated and governed.

5.3 Relation to Rule Engines and Deterministic Systems

Deterministic execution itself is not new. Rule engines, decision tables, and policy DSLs have long provided reliable execution for structured decisions. A natural question is whether the results observed here simply restate the value of those systems.

The distinction explored in this paper is not determinism alone, but the representation and lifecycle of execution in AI-native systems. In particular:

- Execution artifacts are derived from human-readable intent through a transformation step, rather than authored directly as low-level rules.
- Temporal binding is treated as a first-class concern through explicit versioned snapshots rather than implicit context.
- Governance operations such as replay, diff, and attribution operate over a structured execution representation rather than logs or generated explanations.

In this sense, the comparison is not between "rules" and "AI," but between:

- systems where execution is produced at runtime through inference, and
- systems where execution is defined prior to runtime in a form that can be inspected, versioned, and analyzed.

The latter has precedents in traditional systems. What is new in the present setting is the role such a representation appears to play when combined with LLM-based reasoning layers.

5.4 The Determinism Trap

Experiment O-02 includes an important control: agent-coded Python. This system achieves 100% T1 correctness and 0% temporal misbinding through frozen deterministic artifacts. It shows that removing inference from the execution path recovers correctness and temporal binding.

But agent-coded Python does not achieve governance authority. Its execution representation is opaque Python code, not a formal intermediate representation. It can produce syntactic governance outputs, for example code diffs, but it cannot support structural attribution or invertible replay in the same way a formal execution representation can.

This result matters because it shows that determinism is *necessary but insufficient*. Freezing execution into artifacts recovers part of the required profile. Recovering the full profile requires not just deterministic execution, but an execution representation that supports governance structurally rather than heuristically.

5.5 What the Evidence Points Toward

A distinct execution layer becomes plausible when three conditions hold:

1. **The existing abstraction exhibits persistent limitations**, not only performance gaps, but runtime properties that remain non-zero across implementations.
2. **The requirements are stable and consequential**, driven by regulation, liability, or operational dependence.
3. **An alternative execution model demonstrates different properties** under controlled conditions.

The results presented here are consistent with these conditions:

1. Inference-based systems exhibit non-zero instability, temporal misbinding, and governance limitations across multiple architectures.
2. The requirements, correctness, determinism, temporal fidelity, and governance, are standard in high-stakes operational domains.
3. Compiled execution (Opus) demonstrates a different runtime profile, with these failure modes removed from the execution path across 780 cells.

These results do not, by themselves, establish a single inevitable architecture. They do suggest that separating reasoning from execution, and representing execution in a deterministic, inspectable form, is a productive direction for AI-native systems operating under strict constraints.

6 Discussion

6.1 The Role of LLMs in Compiled Execution

The argument of this paper is not anti-LLM. It is pro-architecture. Language models are powerful upstream tools:

- **Intent authoring**: LLMs can help humans express policy intent in natural language.
- **Specification generation**: LLMs can translate intent into formal specifications.
- **Code generation**: LLMs can generate implementation artifacts, as demonstrated by agent-coded Python.
- **Validation**: LLMs can review and critique execution outputs.

The argument is about *where in the stack* LLMs operate most effectively. Upstream, as reasoning and authoring tools: yes. At the execution boundary, as runtime decision engines: the evidence here suggests no. This is not a rejection of LLMs. It is a recognition that reasoning and execution are different computational operations and place different demands on the systems that perform them.

6.2 Temperature, Scaffolding, and the Limits of Mitigation

Practitioners often respond to inference failures with engineering mitigations: lower temperature, better prompts, schema validation, retry logic, and orchestration. Experiment O-02 evaluates the most common of these through the structured + validation stack and finds that they close the format gap without recovering the deeper execution properties.

This matters because it separates *solvable engineering problems* from *architectural mismatches*. Schema validation is a solvable engineering problem: add a validator, retry on failure, and format compliance improves. Temporal binding is not the same kind of problem. No amount of output-layer validation can ensure that the model interpreted the input using the correct policy version. The mitigation acts on the wrong surface.

6.3 Governance as a Structural Property

Governance is often treated as an observability concern: log what the system did, then generate an explanation after the fact. In regulated operational systems, this is insufficient. Governance must be *structural*: the system must be able to demonstrate, through its own execution representation, why a decision was made, under which rule version, and what would change under a policy revision.

Compiled execution supports structural governance because the execution representation is inspectable. Structural diff identifies every rule change between policy versions. Counterfactual attribution identifies which changes caused which decision changes. Replay produces identical outputs because execution is deterministic.

Inference-based systems cannot provide the same guarantees, because their execution representation is a token sequence. The model's "reasoning" is a generated narrative, not a formal derivation. Even when that narrative is plausible, it is not structurally verifiable in the same way.

6.4 From O-02 to O-03

This paper establishes the need for distinguishing execution from runtime inference in AI-native operational systems. It shows that the distinction is not rhetorical. It is measurable in correctness, determinism, temporal fidelity, and governance.

The next step is to shift focus from the failure surface of inference to the capability surface of compiled execution. Where O-02 asks "why do inference systems fail under execution constraints?", O-03 can ask "what does a compiled execution layer enable once those constraints are satisfied?" That includes more complex workflows, richer governance scenarios, multi-domain composition, and larger execution surfaces.

7 Limitations

Workflow complexity. Three domains with 3-field output schemas are sufficient to expose the observed failure patterns, but more complex workflows may reveal additional distinctions. This paper establishes the execution boundary under controlled conditions; it does not map the full capability surface of compiled execution.

Single LLM provider. All inference-based systems use Claude Sonnet 4. Different models may produce different rates. However, O-01 showed similar patterns across five models from three providers, and the structural argument here does not depend on any one model's exact rate.

Prompt optimization. LLM prompts were not exhaustively optimized. However, prompt sensitivity is itself relevant evidence in an execution study. A system whose reliability depends materially on phrasing is not well-aligned with execution requirements.

Pre-O-02 calibration to formal metric shifts. Some LLM metrics shifted between pre-O-02 calibration runs and formal execution (e.g., raw LLM T1: 94.7% in calibration $\rightarrow$ 10.5% in formal). This variability is not treated here as a nuisance to be averaged away. It is part of the behavior under study: the same frozen prompt, same model, and same temperature can yield materially different correctness rates across execution windows.

T3 scorer sensitivity. The governance authority composite metric does not fully capture Opus's structural governance properties. Individual governance scores (completeness=1.0, precision=1.0, structural invertibility) are therefore the more appropriate evidence for governance claims in this paper.

8 Conclusion

Across 3,900 controlled runs on frozen operational workflows, this study identifies a consistent gap between runtime inference and the requirements of high-stakes execution. The gap is not best characterized as a simple accuracy deficit. It appears in properties such as determinism, temporal fidelity, and governance, where non-zero failure rates persist even under structured output, validation, and orchestration.

Compiled execution offers one way of addressing this gap by relocating key failure modes out of the runtime path and into static, inspectable artifacts. In doing so, it produces a different

execution profile: stable outputs, explicit temporal binding, and governance capabilities grounded in representation rather than generation.

The broader implication is architectural rather than benchmark-oriented. As AI systems move deeper into operational roles, it becomes increasingly useful to distinguish between:

- reasoning layers, where inference is highly effective, and
- execution layers, where stronger guarantees may be required.

The results presented here suggest that this distinction is not only conceptual, but measurable in system behavior. Further work, including O-03, will explore how such execution layers behave under greater complexity and how they integrate with LLM-based reasoning systems in practice.

9 References

Hendrycks, D., Burns, C., Basart, S., et al. (2021).
Measuring Massive Multitask Language Understanding.
arXiv:2009.03300.

Srivastava, A., Rastogi, A., Rao, A., et al. (2022).
Beyond the Imitation Game: Quantifying and Extrapolating the Capabilities of Language Models.
arXiv:2206.04615.

Liu, X., Yu, Y., Zhang, J., et al. (2023).
AgentBench: Evaluating LLMs as Agents.
arXiv:2308.03688.

Leucker, M., & Schallhart, C. (2009).
A Brief Account of Runtime Verification.
Journal of Logic and Algebraic Programming.

A Supplementary Result Tables

Appendix Table A1: Per-system failure taxonomy

Category	Opus	Raw LLM	Structured + Validation	Agent System	Agent-Coded
enum_value_invalid	0	255	0	0	0
wrong_decision	0	0	15	211	0
wrong_reason_code	0	0	24	0	0
wrong_amount	0	0	0	38	0
malformed_output	0	0	0	1	0
Total	**0**	**255**	**39**	**250**	**0**

Table 11: Failure taxonomy by system. Only categories with non-zero incidence are shown.

Appendix Table A2: Temporal misbinding type distribution

Type	Opus	Raw LLM	Structured + Validation	Agent System	Agent-Coded
none (correct)	165	0	76	21	165
decision_flip	0	58	3	22	0
value_distortion	0	0	0	29	0
dominated_by_error	0	107	86	93	0
signal_irrelevant	285	285	285	285	285

Table 12: Temporal misbinding type distribution across all T2 runs. Signal-irrelevant scenarios are not scored for temporal binding.

Appendix Table A3: Determinism detail

System	Mean Stability	Min	Inconsistent	Cohorts
Opus	1.000	1.000	0	52
Agent-Coded Python	0.942	0.600	3	52
Structured + Validation Stack	0.864	0.133	25	52
Raw LLM	0.663	0.133	39	52
Agent System	0.499	0.133	49	52

Table 13: Output hash stability across 15-run cohorts (52 per system, 260 total).

B Experimental Controls and Integrity

- **Manifest integrity**: Body hash `2757fbbd...` identical across pre-O-02 calibration runs and formal execution. A 7-check lock validation process was run before and after execution.
- **Evidentiary enforcement**: No stubs, live execution, hash identity, runtime bounds, environment fingerprint on all 3,900 records.
- **Environment continuity**: `model_signature_hash` uniform across all records. No pre-O-02 calibration drift detected.
- **Artifact validation**: All 3,900 run artifacts validated for schema conformance, JSON integrity, and required field completeness (0 validation errors).
- **Retry transparency**: 785 runs with `attempt_count` $>$ `1` logged. Agent system's multi-role orchestration accounts for 780 (by design). 5 structured + validation stack retries on structural failures (within bounds).
- **Invariant sanity checks**: Structural invariants verified (Opus determinism=1.000, temporal correctness=100%). Expected instability patterns confirmed (agent system stability=0.499, structured + validation stack misbinding=3.8%).