

Handling Non-Determinism in AI Systems: A Distributed Systems Perspective

Poonam Dhavale

poonamdhavale.aisystems@gmail.com

Preprint

Version: v0.2 | Date: April 2026

AI Use Disclosure

The author is a Principal Software Engineer with over 25 years of experience building production distributed databases and clustered storage systems, with multiple patents in distributed systems, along with several years of applied AI R&D experience.

The central thesis and overall framework of this paper are based on the author's professional experience and engineering judgment. The concepts, analogies, and design recommendations are informed by the author's direct hands-on experience with the distributed systems principles discussed.

AI tools were used as a research and writing assistant under the author's direction. They assisted with drafting, editing, and refining the presentation of ideas. The author defined the intellectual framework, guided all analytical decisions, reviewed and revised all content, and is fully responsible for the accuracy and integrity of the work.

Abstract

Large Language Models (LLMs) and the pipelines built around them are increasingly deployed in production systems where reliability, consistency, and auditability are critical. These pipelines are often treated as deterministic software systems, but in practice they behave as probabilistic distributed systems. This mismatch is a root cause of recurring production failures, including inconsistent outputs, untraceable errors, and silent behavioral drift.

This paper presents a position and systems-design framework grounded in distributed systems theory. We introduce a taxonomy of six sources of non-determinism in AI systems and formalize AI pipelines as Probabilistic Compute Graphs (PCGs) composed of Probabilistic Compute Nodes (PCNs) and deterministic components. Building on this model, we derive five architectural principles—Version Everything, Causal Semantic Tracing, Deterministic Replay, Quorum-Based Validation, and Guardrails as Consistency Constraints—and instantiate them in a reference architecture that separates the Intelligence Plane from the Reliability Plane.

Our focus is on runtime reliability of inference pipelines rather than training-time reproducibility. Empirical validation across production deployments is left to future work.

1. Introduction

1.1 The Rise of AI Systems

Imagine a financial services company that deploys an AI assistant to help analysts query earnings reports. The system retrieves relevant documents from a vector database, passes them to a large language model, and generates a structured summary with a recommended action. On Monday morning, an analyst asks: "What is the revenue trend for Acme Corp over the last three quarters?" The system returns a confident summary: revenue is declining, and it recommends caution. The analyst flags the report for review. Forty minutes later, a colleague runs the identical query. This time, the system reports that revenue is stable and trending upward. No data changed. No model was updated. The two analysts now hold contradictory AI-generated assessments of the same company, derived from the same vector database, within the same hour.

This is not a bug in the traditional sense. No code path failed. No exception was thrown. The system behaved exactly as designed, and that is precisely the problem.

Over the past several years, large language models (LLMs) have moved rapidly from research demonstrations into the critical path of real software systems. Modern AI applications are no longer simple question-answering interfaces. They are complex, multi-stage pipelines. In a typical Retrieval Augmented Generation (RAG) system, a user query triggers a retrieval step that fetches documents from a vector store, a reranker filters and orders those documents by relevance, an LLM synthesizes a response, and one or more tool-calling steps execute downstream actions (Lewis et al., 2020). In more advanced deployments, multiple AI agents run in parallel, coordinate results, and route work across conditional branches. Emerging protocols including the Model Context Protocol (MCP) and agent-to-agent (A2A) coordination frameworks further extend these pipelines across process and network boundaries (Anthropic, 2024; Google, 2025). The framework presented here applies to all of these AI system patterns.

To our knowledge, no prior work provides a unified distributed-systems framework for modeling and managing non-determinism in AI inference pipelines.

1.2 Determinism: The Assumption Behind All Traditional Software

For decades, software engineering has been built around a working assumption: given the same input and the same system state, a program will produce the same output. In practice, real systems involve concurrency, timing effects, and environmental variability. However, they are engineered to approximate determinism closely enough that this assumption holds for most purposes. As a result, determinism is so deeply embedded in how we build, test, and debug software that it is rarely stated explicitly. It is simply taken for granted.

This assumption holds across a wide range of systems when appropriate constraints are applied. A SQL query executed against a consistent snapshot of a database returns the same result. A compiler, given the same source code and build environment, produces the same binary. Distributed systems use consensus algorithms such as Raft to maintain a consistent view of system state, ensuring that clients observe coherent results even when requests are served by different nodes.

Determinism or more precisely, the practical illusion of determinism, is not just a convenience. It is the foundation on which debugging, regression testing, auditability, and production reliability are built.

1.3 AI Systems Break This Assumption

Large language models do not operate under the same assumptions as traditional software. Instead of computing a single deterministic output, an LLM generates text by sampling from a probability distribution over possible next tokens. At non-zero temperature settings, this sampling process introduces randomness by design. Even at temperature zero, outputs may still vary due to infrastructure and numerical effects.

Model-level stochasticity, however, is only one source of variability. At the hardware level, modern LLMs run on GPUs that execute floating-point operations in parallel; because floating-point arithmetic is not associative, small numerical differences can arise from variations in accumulation order across runs. At the retrieval layer, vector databases rely on approximate nearest neighbor algorithms that trade exactness for speed, meaning the same query can return slightly different results. At the workflow level, agents that call external tools receive responses that reflect the state of the world at the moment of the call — a state that changes continuously over time. These sources of variability compound across the pipeline.

In multi-agent systems, this effect is amplified further. Parallel agent execution introduces race conditions in context assembly, where different agents may observe and contribute context from different points in time. Router nodes introduce path-dependent variability, where small differences in intermediate outputs lead to entirely different execution paths. Fan-in joins must reconcile outputs from multiple non-deterministic sub-agents, and the order in which those outputs are merged can affect downstream behavior. The result is a system whose behavior cannot be reliably reproduced, predicted, or audited without purpose-built infrastructure.

1.4 Why This Matters

A common failure mode when deploying probabilistic systems in contexts that assume deterministic behavior produces serious practical consequences:

- **Debugging becomes unreliable without proper instrumentation.** Without a record of what the system retrieved, what seed was used, and which model version ran, failures cannot be consistently reproduced or diagnosed.
- **Regression testing becomes probabilistic rather than deterministic.** The same input may pass a test on one run and fail it on the next, making traditional pass/fail testing insufficient.
- **User trust erodes.** Inconsistent behavior in high-stakes domains such as medicine, law, and finance is a liability, not a minor annoyance.
- **Evaluation results drift over time.** A system that scored well on a benchmark last month may score differently today, even when no intentional model change occurred, due to updates in retrieval indexes or changes in external data sources.

1.5 Scope

This paper focuses on the runtime reliability of AI inference pipelines in production systems. It does not address training-time reproducibility or formal verification of model behavior or internals. Instead, this paper provides a principled engineering framework, grounded in distributed systems theory, for managing the consequences of non-determinism in production inference pipelines.

1.6 Contributions

We argue that treating AI pipelines as deterministic software when they are, in practice, probabilistic distributed systems is a category error. While non-determinism cannot be fully eliminated at the system level, it can be systematically managed using techniques developed in the distributed systems literature. This paper makes the following contributions:

- A conceptual framing of AI pipelines as Probabilistic Compute Graphs (PCGs) composed of Probabilistic Compute Nodes (PCNs) and Deterministic Compute Nodes (DCNs).
- A taxonomy of non-determinism spanning six major sources: model stochasticity, infrastructure variability, context variability, workflow variability, semantic drift, and human-in-the-loop variability.
- A mapping between common AI failure modes and classical distributed systems failure modes.
- A Reliability Plane architecture for managing non-determinism in production systems, along with five design principles grounded in established distributed systems precedents.
- An analysis of tradeoffs between latency, accuracy, determinism, and total cost of ownership, with applicability across RAG, multi-agent, and emerging agent coordination patterns.

2. Sources of Non-Determinism in AI Systems

Non-determinism in AI pipelines does not have a single cause. It has six major sources, each at a different layer of the stack, each requiring different mitigation. Treating them as one problem leads to solutions that fix one layer while leaving the others untouched.

2.1 Model Stochasticity

Stochasticity means that a process produces different outcomes each time it runs, even when the starting conditions are identical. In LLMs, this arises because the model does not pick a single correct word at each step; it randomly samples from a distribution of possible words, so each run can take a different path. LLMs generate responses token by token. At each step, the model computes a probability distribution over its entire vocabulary and samples from that distribution. The temperature parameter controls how sharp or flat this distribution is. At temperature zero, the model always picks the single highest-probability token (greedy decoding). Many engineers believe this makes an LLM deterministic. It does not, as Section 2.2 explains. Other sampling strategies such as nucleus sampling (top-p), top-k each introduce their own variability. For any non-trivial prompt, an LLM output is one sample from a distribution of possible responses (Holtzman et al., 2020). Because LLMs are auto-regressive (the next token depends on all previous tokens), non-determinism is subject to compounding amplification: a single divergent token choice early in a sequence, caused by even a marginal shift in the probability distribution, alters the entire attention context for all subsequent tokens, potentially leading to radically different semantic outcomes from nearly identical probability distributions.

2.2 Infrastructure Variability

Modern LLMs run on GPUs, which achieve their performance by executing thousands of operations in parallel. Under the IEEE 754 standard, floating-point addition is not associative: $(a + b) + c$ does not always equal $a + (b + c)$ in finite-precision binary arithmetic. When large numbers of such operations are executed in parallel, partial results may be accumulated in different orders across runs, producing small but non-zero numerical differences.

Across the many layers and billions of operations in a large model, these small differences can propagate and occasionally amplify through the computation. In some cases, a difference that is negligible at the numerical level can lead to a different token being selected at the output (PyTorch, 2025).

Additional sources of infrastructure variability include inter-GPU communication ordering in multi-GPU deployments, quantization-induced rounding differences (Dettmers et al., 2022), and batch composition and ordering effects in inference servers. This variability is not a bug; it is an inherent property of large-scale parallel floating-point computation.

2.3 Context Variability

In any system that feeds external context into the LLM—retrieved documents, memory stores, tool outputs, and API responses—the context itself is a source of non-determinism independent of the model. Vector databases use Approximate Nearest Neighbor (ANN) algorithms that trade exactness for speed and may return different top-k results for the same query due to variations in search paths or approximations (Malkov & Yashunin, 2018).

Context also changes over time: indexes are updated, embedding models are versioned—altering what gets retrieved even without changes to application logic—and LLMs exhibit positional sensitivity, responding differently to the same documents when presented in a different order (Liu et al., 2023).

In stateful or multi-agent systems, additional variability arises from context assembly over time. Memory buffers may include or exclude prior interactions depending on timing, truncation policies, or token limits, and context window management strategies can lead to different subsets or orderings of information being presented to the model across runs.

2.4 Workflow Variability

In multi-step systems, non-determinism compounds across agent and tool boundaries. When an LLM must choose between multiple tools, that choice is often a stochastic decision. External tools return results that reflect the state of the world at the moment they are called, introducing time-dependent variability. In agents that plan and re-plan based on intermediate results, small differences in early steps can lead to entirely different execution paths.

In parallel multi-agent systems, the order in which sub-agent results are received and merged is not fixed, introducing timing-dependent variability in the context assembly step. More generally, systems that rely on external services or distributed agents may observe different responses over time as underlying data changes. As workflows span multiple agents or services, variance propagates and can amplify across these boundaries.

Variance, in this context, refers to the degree to which a system produces different outputs across repeated runs on the same input. High variance indicates unpredictable and inconsistent behavior, while low variance corresponds to more stable and reliable outputs.

2.5 Semantic Drift

In production systems, the four sources above can compound into a fifth: semantic drift. Here, semantic drift refers to the gradual change in an AI system's observable behavior over time. This occurs not because of a single deliberate change to the application, but because multiple small changes accumulate: a model update by the provider, an index refresh, an embedding model revision, or a tool server schema change.

Over time, these changes can cause the system at month six to produce materially different outputs for identical queries than it did at month one. This drift is often invisible without longitudinal monitoring and is a common phenomenon in production systems.

2.6 Human in the Loop

When a human is incorporated as an evaluator or decision-maker within an AI pipeline, they introduce an additional source of non-determinism that is qualitatively different from the technical sources described above. Two different human reviewers assessing the same AI-generated output may reach opposite conclusions, and the same reviewer may evaluate the same output differently over time depending on cognitive state, context, and accumulated experience. Unlike GPU floating-point variability, human variability is not bounded by numerical precision; it reflects variability in human judgment, bias, and expertise.

This variability matters particularly in workflows where human-in-the-loop approval gates are used as quality controls or safety checks. If these gates are not themselves consistent, they cannot serve as a strictly reliable consistency constraint on the overall system. Mitigations include structured evaluation rubrics that narrow the space of valid human judgments; multi-reviewer quorum patterns (analogous to Principle 4) requiring agreement between independent reviewers before an action proceeds; and calibration sessions that align reviewer standards over time. Human-in-the-loop steps can be modeled as probabilistic nodes in the PCG framework, with the same logging and tracing requirements as any other PCN.

None of these sources of non-determinism can be fully eliminated, only managed. This motivates the distributed systems framing in Section 3: building reliable systems from components that are inherently variable requires principled engineering approaches.

3. AI Systems as Probabilistic Distributed Systems

3.1 The Probabilistic Compute Node (PCN)

In classical software, a compute node is a deterministic function: $y = f(x)$ as shown in Figure 1.

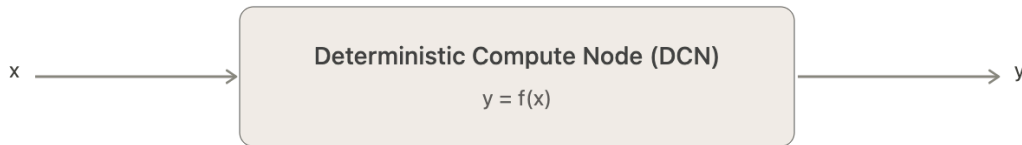


Figure 1 — Deterministic Compute Node (DCN)

AI components break this model. An LLM does not compute one output from an input; it samples one output from a distribution of possible outputs. We formalize this with the Probabilistic Compute Node (PCN).

For a PCN, the output y is sampled from a conditional distribution as shown in Figure 2:

$$y \sim P(y \mid x, \theta, r)$$

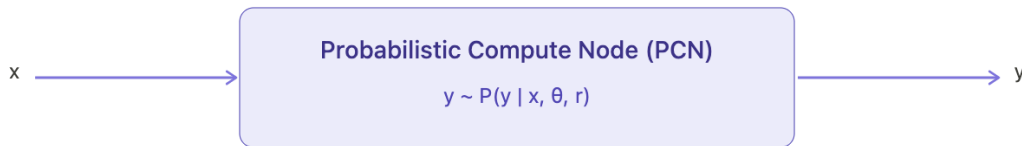


Figure 2 — Probabilistic Compute Node (PCN)

Here is the breakdown of what each term indicates:

- **y (The Output):** This is the specific token, response, or decision generated by the node. Unlike a deterministic function where y is the only possible result for a given input, here y is just one sample drawn from a wider distribution.
- **$P(y \mid \dots)$ (The Probability Distribution):** This indicates that the output is probabilistic. The node doesn't just calculate a value; it computes a probability distribution over all possible outputs and then "samples" one.
- **x (The Input):** This represents the input to the node at execution time, including the user query, conversation history, and any injected context.
- **θ (The Model/Environment Parameters):** This encompasses the system configuration, including model weights, version, quantization, and hardware setup.
- **r (The Randomness/Stochastic Term):** This captures sources of variability not explicitly modeled in x or θ , including stochastic sampling, hardware-level non-determinism, and other uncontrolled or unobserved factors. It includes factors like:
 - The sampling seed used by the LLM.
 - The GPU floating-point accumulation order (which is non-associative).
 - The ANN traversal path in a vector database.

- External state or "world state" at the exact moment of a tool call.

In practice, some sources of variability—such as retrieved context or tool outputs—may be incorporated into the effective input x when they are observable and logged. Controlling non-determinism in a PCN means constraining r (the randomness term) where possible, observing and logging it where feasible, and designing the system to behave reliably across the range of outputs that P can produce.

3.2 The Probabilistic Compute Graph (PCG): Building Blocks

Real AI systems chain multiple PCNs and DCNs together. We model this as a Probabilistic Compute Graph (PCG): a directed graph $G = (V, E)$ where each node $v \in V$ is either a DCN or a PCN, and each edge $(u, v) \in E$ represents a data dependency. Figure 3 shows a PCG for a simple RAG system.

Figure 3 illustrates the sequential pipeline of a basic RAG system modeled as a PCG. The user query enters the graph and passes through five nodes in order:

- (1) **Embed query node (DCN — deterministic embedding given fixed model and environment)**, which converts the query into a vector representation;
- (2) **Retrieve — vector DB node (PCN — ANN search and index state variability)**, which fetches the top-k relevant documents;
- (3) **Rerank node (DCN or PCN depending on implementation — deterministic cross-encoder or stochastic LLM-based reranker)**, which reorders documents;
- (4) **Prompt constructor node (typically a DCN)**, which assembles the final prompt;
- (5) **LLM inference node (PCN — model stochasticity and infrastructure variability)**, which generates the answer.

Each PCN in the chain introduces its own variability and this variability propagates and can amplify across downstream nodes.

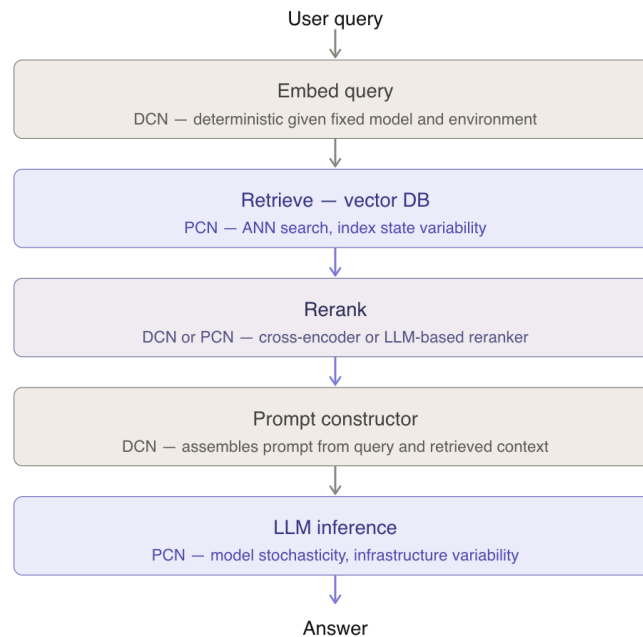


Figure 3 — A PCG for a simple RAG system

The figures below show other building blocks in state of the art AI systems.

Router Node (Figure 4):

A router node implements conditional branching within the PCG, selecting the next node or sub-graph to execute based on the output of the current node. The router produces a discrete routing decision (e.g., selecting between multiple execution paths). It can be implemented as either a DCN (rule-based routing, such as “if intent = classification, route to path A”) or a PCN (LLM-based routing, such as “classify this query and decide which agent to invoke”).

When the routing decision is made by a PCN, the selected branch is stochastic: small differences in input or intermediate outputs can lead to different routing decisions, and identical queries may be routed to different downstream sub-graphs across executions. This introduces path-dependent variability, where the overall non-determinism of the system depends not only on the variance of individual nodes but also on the execution path taken.

To enable debugging and reproducibility, systems must capture the realized execution path at each router node, allowing the full sequence of decisions to be reconstructed.

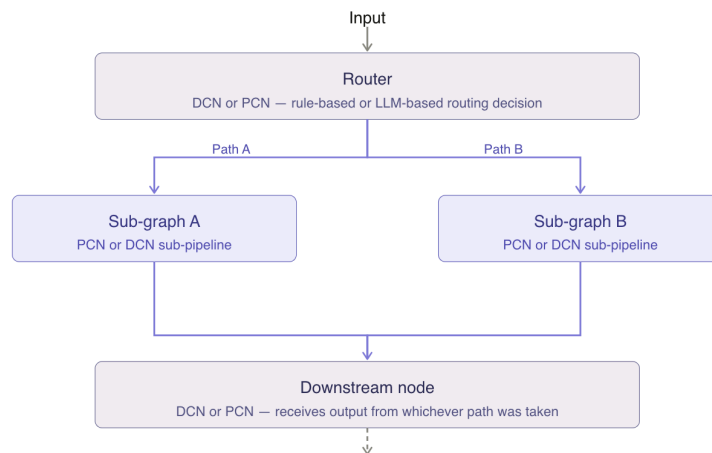


Figure 4 — A Router Node

Fan-out and Fan-in Nodes (Figure 5):

A fan-out node dispatches the same input to multiple sub-graphs (PCGs) that execute in parallel. Each parallel PCG may contain its own mix of PCNs and DCNs. A fan-in join node typically waits for all (or a configured subset of) parallel sub-graphs to complete and merges their outputs before passing the combined result downstream.

This pattern introduces additional sources of non-determinism beyond the variance within each individual sub-graph. First, the outputs of each parallel sub-graph are themselves probabilistic, so the merged context at the fan-in node combines multiple samples (often correlated) from their respective output distributions. Second, the order in which sub-agent results arrive at the fan-in node is non-deterministic and depends on network timing and compute load. When this order is preserved in the assembled context, it can affect how the downstream LLM interprets the combined inputs.

Snapshot isolation at the retrieval layer — pinning all sub-graphs to the same index snapshot ID — is a primary mitigation for context inconsistency across parallel branches.

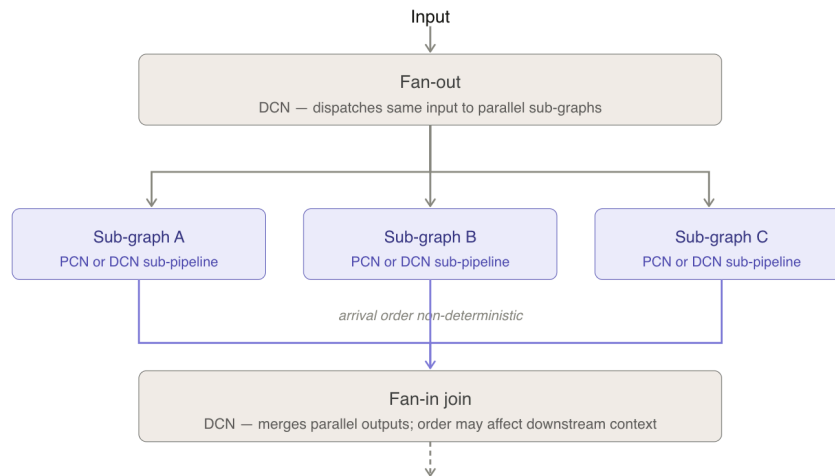


Figure 5 — Parallel PCG execution using Fan-in and Fan-out nodes

Cyclic PCG (Figure 6):

Cycles are possible in agentic systems that loop until a termination condition is met — the canonical example being ReAct-style agents (Yao et al., 2023) that follow a Plan → Act → Observe loop and continue iterating until a router node determines that the task is complete. In a cyclic PCG, the graph contains a directed cycle: the output of one or more PCNs is fed back as input to an earlier node in the graph.

Because each iteration may pass through one or more PCNs, cycles can introduce additional accumulated variance over time. A ReAct agent that takes multiple reasoning steps will traverse several PCN sampling operations, and small differences in early steps can propagate through subsequent iterations. The number of iterations itself may also vary across executions, introducing further variability.

Cycles with PCN nodes require special handling in the Reliability Plane. Logic/Flow guardrails (Principle 5) should enforce maximum iteration counts to prevent runaway loops caused by agents that fail to reach or recognize a termination condition. The version manifest must capture the full iteration trace, not just the final output, to enable meaningful replay and debugging.

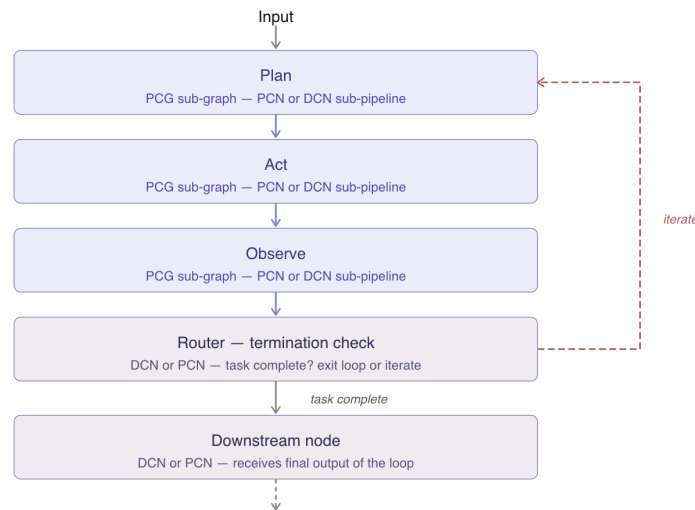


Figure 6 — Cycles in PCG

Most AI systems — including dynamic multi-agent workflows using MCP, A2A, or future coordination protocols — can be modeled by combining the above building blocks. Understanding which building block each component maps to helps reveal its non-determinism profile and the appropriate reliability mechanisms to apply.

Three properties follow from this structure. First, non-determinism propagates and can amplify: variability at early nodes can influence all downstream computations. Second, parallel execution introduces additional variability — when merge order is preserved in the assembled context, the order in which fan-in joins receive sub-agent outputs can affect downstream behavior. Third, router nodes create path-dependent variability: different executions of the same query may traverse different subgraphs, making overall behavior dependent on the execution path taken.

The topology determines the debugging strategy: the question is not which component is non-deterministic (many may be), but at which node, and along which execution path, a particular run diverged from expected behavior.

3.3 AI Failure Modes Map to Distributed Systems Concepts

The PCG model maps naturally onto structures that distributed systems researchers have studied for decades. This mapping allows us to import existing theory and tooling directly into the problem of AI reliability. The following mappings are analogical rather than exact equivalences, intended to transfer intuition from distributed systems to AI pipelines.

Distributed Systems Concept	What It Means (with example)	AI Systems Equivalent
Partial node failure	A server returns wrong answers while appearing healthy (e.g. a replica serving stale data).	Hallucination: the LLM returns a confident, well-formatted response that is factually wrong. The system signals no error.
Stale replica	A backup copy has not received the latest updates (e.g. read replica 30s behind primary).	Stale context: the retrieval index has not been updated with the latest documents,

Distributed Systems Concept	What It Means (with example)	AI Systems Equivalent
		so the LLM reasons from outdated information.
Network partition	Two parts of the system cannot communicate (e.g. a split-brain database cluster).	Agent communication failure: When an agent cannot reach a required sub-agent or tool server, it faces the same choice as a partitioned distributed system: proceed with incomplete information (availability over consistency) or halt and return no answer (consistency over availability). In MCP and A2A-style deployments, if a tool server is unreachable, the orchestrating agent must explicitly make this tradeoff.
Inconsistent replicas	Two replicas return different values for the same query due to unsynchronized writes.	Run-to-run inconsistency: the same query can return contradictory answers across executions, even when the underlying model and data have not changed (analogous to inconsistent replicas).
Byzantine fault	A node behaves arbitrarily or maliciously, returning plausible-but-incorrect data to mislead peers.	Adversarial prompt injection (input-level Byzantine behavior): malicious content in retrieved documents instructs the LLM to ignore its system prompt.
Clock skew	Different nodes have slightly different clocks, causing events to appear out of order.	Temporal non-determinism: tool calls return different results depending on when they are made, reflecting time-dependent behavior analogous to clock skew.
Cascading failure	One slow component triggers failures in all downstream services.	Error propagation / compounding variance: a bad retrieval result → bad prompt → bad LLM output → wrong agent action.
Eventual consistency	Replicas are temporarily out of sync but converge given time and no new writes.	Semantic drift: the system's output distribution shifts over time as underlying components change, eventually stabilizing at a new steady state — analogous to eventual consistency, where the system converges over time, but to a different behavior rather than a single fixed value.

This mapping allows us to apply established distributed systems techniques, such as versioning, tracing, quorum, and consistency models, to AI pipelines.

4. Design Principles for Reliable AI Systems

This section translates the distributed systems toolkit into five concrete design principles. Each is grounded in a specific distributed systems precedent, addresses a specific class of non-determinism from Section 2, and can be implemented with current tooling. These principles are not all-or-nothing: each introduces tradeoffs between performance, cost, accuracy, and complexity, as analyzed in Section 4.7.

4.1 Principle 1: Version Everything

Every component that can materially affect the output of an AI pipeline should be versioned or recorded, and that version must be captured at runtime alongside every inference request. This includes the LLM model version, prompt template (stored as a content hash), context source snapshot ID, embedding model version and quantization level, inference framework version, and the random seed used for each call (when available).

This principle is directly inspired by immutable infrastructure and content-addressable storage systems such as Git. In distributed systems, every write is tagged with a version identifier (e.g., timestamps, log sequence numbers, or version vectors), allowing the system state at any past moment to be reconstructed. AI systems require the same capability for reproducibility and debugging.

A minimal version manifest:

```
{
  "request_id": "req_7f3a9c",
  "timestamp": "2026-04-15T09:23:11Z",
  "llm_model": "provider/model-family-version",
  "prompt_template_hash": "a3f8b2d1",
  "embedding_model": "provider/embedding-model-version",
  "context_snapshot": "idx_20260415_0900",
  "inference_framework": "vllm-0.4.1",
  "random_seed": 42,
  "pcg_topology_hash": "f9c3e1a7"
}
```

In systems with router nodes and dynamic branching, the topology of the PCG may vary across executions. Capturing a hash of the realized execution trace (`pcg_topology_hash`)—which nodes were executed and which branches were taken—is essential for replay, debugging, and drift monitoring.

Version history must be retained, not just the current version. A tiered storage strategy is effective: hot storage for recent requests (e.g., 30 days), warm object storage for medium-term retention (30–365 days), and cold archival storage for long-term analysis beyond one year.

Without comprehensive versioning, reproducibility degrades from a deterministic process to a best-effort approximation.

4.2 Principle 2: Causal Semantic Tracing

Every inference request should produce a complete (or as complete as practical) execution trace, recording the inputs, outputs, and timing of every node in the pipeline, with causal relationships between nodes preserved. Standard distributed tracing systems (Sigelman et al., 2010) capture timing and service boundaries. AI systems require more: the content at each node boundary, including what was retrieved, what prompt was constructed, what the model produced, and what each tool call returned (or references to that content, such as document IDs or hashes), subject to privacy and cost constraints.

We call this **causal semantic tracing**. It transforms debugging from speculation into diagnosis.

In parallel PCGs, the tracing model must extend beyond single-threaded causal chains. When a planning agent fans out to multiple sub-agents executing concurrently, each sub-agent produces its own causal subtrace. These subtraces are linked to the parent trace via a shared trace ID and annotated with their fan-out origin. At the fan-in join node, the trace records the merge order (when it affects downstream behavior) and how aggregation was performed. This enables near-complete reconstruction of the execution tree, including the timing and ordering of parallel branches.

This directly aligns with causal consistency: the trace store should preserve causal ordering, ensuring that causally related operations are observed in the correct order—if operation A causally precedes B, an engineer inspecting the trace observes A before B, even across parallel branches and process boundaries.

Traces should be stored in a dedicated, append-only trace store. A tiered storage model applies: full semantic traces in hot storage for 7–30 days, compressed summaries in warm storage for up to a year, and aggregated statistical baselines in cold archival storage beyond that. The hot storage window here is shorter (7–30 days vs. 30 days in Principle 1). This is intentional since full semantic traces are larger and more expensive than version manifests.

To control storage costs, trace data should use content-addressable storage wherever possible. Instead of storing large payloads (e.g., a 100KB prompt) in every trace, store a content hash (e.g., `prompt_template_hash`) and deduplicate identical artifacts across traces. This significantly reduces total cost of ownership (TCO) while preserving the semantic integrity and replayability of the trace.

Without causal semantic tracing, AI systems reduce to opaque pipelines where failures can be observed but not explained.

4.3 Principle 3: Deterministic Replay

Given a stored trace and version manifest, it should be possible to replay a past inference request and produce an output with measurable fidelity to the original. Full replay pins all component versions, fixes the random seed where supported, and uses stored context directly (skipping retrieval) to eliminate context variability. Partial replay re-executes selected nodes or subgraphs while holding others fixed, enabling targeted testing of prompt changes, model updates, or workflow modifications against known failures.

This principle is inspired by event sourcing (Fowler, 2005) and deterministic simulation testing, a technique used in production distributed systems to validate behavior under controlled conditions. Perfect replay is not achievable due to factors such as GPU floating-point non-determinism and the potential deprecation of model versions by providers. The goal is approximate replay with measurable fidelity, evaluated using task-appropriate metrics such as semantic similarity or downstream correctness.

For parallel PCGs, replay must re-execute sub-agents using their stored subtrace contexts and merge their outputs in the recorded fan-in order to preserve structural fidelity. While exact timing may differ across runs, maintaining causal structure ensures that the replayed execution remains comparable to the original.

4.4 Principle 4: Quorum-Based Validation

For high-stakes requests, the system should execute multiple independent inference runs and aggregate the results before returning a response. Outputs that do not meet a defined consistency threshold should be flagged for review or excluded from the final response. This converts inherently stochastic outputs into confidence-weighted decisions.

This approach is analogous to quorum read patterns in distributed databases and is inspired by Byzantine fault tolerance: assume any individual run may be incorrect, and seek agreement across multiple runs as a heuristic for reliability. Unlike classical quorum systems, however, agreement here is semantic rather than exact—outputs are compared based on meaning rather than bitwise equality.

A practical quorum validation pipeline consists of:

1. **Generate multiple samples:** Run k independent inference samples (e.g., using different random seeds where supported, or independent requests under identical configurations).
2. **Measure semantic similarity:** Compare outputs using metrics such as BERTScore (Zhang et al., 2020), LLM-as-judge approaches (Zheng et al., 2023), or task-specific evaluation metrics where available.
3. **Identify consensus:** Cluster outputs under a defined similarity threshold and identify the largest consensus group.
4. **Handle outliers:** Flag responses outside the consensus cluster for inspection, logging, or human review.
5. **Return result:** Return a representative response from the consensus cluster (e.g., centroid, highest-confidence sample) and log all intermediate outputs for traceability.

Quorum-based validation is not appropriate for every request. It multiplies inference cost and can increase latency. It should be applied selectively: high-stakes domains (e.g., finance,

healthcare, legal), requests historically associated with high variance or workflows explicitly requiring high confidence or auditability.

The effectiveness of quorum validation depends on diversity across samples. Identical configurations may produce correlated errors, reducing the benefit of aggregation. Introducing controlled variation (e.g., different prompts, models) can improve robustness.

Tail-Latency Management

A key challenge with quorum validation is tail latency. When issuing k parallel inference requests, response time is typically gated by the slowest of the required responses.

To mitigate this, systems can adopt **hedged requests**, a technique from distributed systems. Instead of issuing k requests, issue $k + r$ parallel requests and return as soon as k responses arrive, discarding the rest. The effective latency becomes the k -th fastest response out of $k + r$, significantly improving p99 latency at the cost of additional compute.

For example, for a quorum size of $k = 3$, issuing 5 parallel requests and returning the first 3 results reduces sensitivity to stragglers. This tradeoff is justified for high-stakes tiers where latency predictability and reliability are more important than raw cost.

Limitations and Tradeoffs

Quorum-based validation provides no formal correctness guarantees. If all samples are biased or incorrect, the consensus will also be incorrect. It is therefore best understood as a probabilistic reliability mechanism rather than a correctness proof.

The tradeoffs include:

- **Cost:** Linear increase with the number of samples (k)
- **Latency:** Increased unless mitigated with hedging
- **Complexity:** Additional infrastructure for clustering, scoring, and orchestration

Despite these costs, quorum validation is a powerful tool for reducing variance and increasing confidence in probabilistic systems.

4.5 Principle 5: Guardrails as Consistency Constraints

Every AI pipeline should enforce explicit constraints on what it is allowed to output and what actions it is allowed to take. These guardrails must be treated as first-class system components, not afterthoughts.

This is closely analogous to integrity constraints in relational databases. A **NOT NULL** or **FOREIGN KEY** constraint does not prevent application code from attempting to write invalid data; it enforces at the system level that invalid data cannot persist. AI guardrails play the same role: they act as last-line-of-defense enforcement mechanisms that are explicit, logged, versioned, and designed to be non-bypassable by the application layer.

This principle also draws from established distributed systems patterns such as circuit breakers (Nygard, 2007). If guardrail violation rates exceed a threshold, the system should stop routing

requests to the failing component and degrade gracefully (e.g., fallback responses, simplified execution paths, or human escalation). Guardrail policies themselves should be versioned and included in the version manifest (Principle 1), ensuring that enforcement behavior is reproducible and auditable.

Guardrails can be categorized based on where they operate within the pipeline:

Type	Focus	Examples
Input Rails	Protecting the model	Prompt injection detection, PII redaction, intent validation, malformed input rejection.
Output Rails	Protecting the user	Hallucination checks, factual grounding verification, tone and safety filtering, schema validation, PII detection in responses.
Tool/Action Rails	Protecting the system	Rate limiting, role-based access control for APIs, human-in-the-loop approval for irreversible actions (e.g. deleting a record), cost threshold enforcement.
Logic/Flow Rails	Protecting the process	Preventing infinite loops in agent reasoning, enforcing maximum step counts, budget and token caps, detecting circular A2A delegation.

In PCG-based systems, guardrails should be applied at critical node boundaries, not only end-to-end. This ensures that invalid or unsafe intermediate outputs do not propagate downstream and amplify through the graph.

This is particularly important in parallel and cyclic PCGs:

- **Fan-out / Fan-in patterns:** Each sub-agent branch should be independently validated. At fan-in join nodes, guardrails should validate aggregated results and enforce constraints on merge behavior.
- **Cyclic execution (agent loops):** Logic/Flow guardrails must enforce maximum iteration counts and detect failure modes such as non-termination or circular A2A delegation.
- **Planning agents:** Tool/Action guardrails must constrain what actions a planning agent is allowed to trigger, preventing a hallucinating planner from issuing harmful or irreversible commands across multiple branches.

Guardrails should also be applied at the boundary of every pipeline step. Output from step N —for example, a structured document containing sensitive information—must be validated and sanitized (e.g., PII redaction) before being passed as input to step $N+1$. This prevents contamination from propagating through the pipeline.

Guardrail enforcement should be logically centralized and versioned, even if physically distributed across services (e.g., middleware layers, sidecars, or policy engines). Centralized policy definition ensures that updates propagate consistently across all enforcement points.

All guardrail decisions—passes, failures, and escalations—must be logged as part of the execution trace (Principle 2), enabling auditing, debugging, and policy evaluation over time.

Guardrails do not eliminate non-determinism; they bound its impact. In PCG terms, guardrails define the permissible output and action space of each node, constraining the range of acceptable outcomes from probabilistic computation.

They also introduce tradeoffs. Overly strict guardrails may produce false positives, rejecting valid outputs and degrading user experience. Looser guardrails may allow unsafe or incorrect outputs to pass through. Effective systems calibrate guardrail policies based on risk tolerance and continuously refine them using observed traces.

In high-stakes workflows, guardrails can be combined with quorum-based validation (Principle 4), requiring that responses both satisfy policy constraints and achieve consensus before being accepted.

4.6 Consistency Models for AI Pipelines

We extend the distributed systems mapping with a formal grounding in the Jepsen consistency hierarchy (Kingsbury <https://jepsen.io/consistency> ; Bailis et al., 2013). The Jepsen models were developed to precisely characterize what guarantees a distributed system provides about the visibility and ordering of reads and writes across time. AI pipelines have directly analogous requirements: what does an inference call "see" about the world, and is that view consistent across calls?

We apply these consistency models as an analogical framework rather than a formal equivalence. The semantic nature of LLM outputs means that "consistency" must be interpreted probabilistically rather than exactly: two outputs are "consistent" if they are semantically equivalent, not bitwise identical. With that caveat explicit, the mapping is both precise and practically useful.

Understanding Consistency and Isolation in Distributed Systems

The Jepsen framework organizes distributed system guarantees into two separate categories: consistency models and isolation levels. These are often confused, but they answer different questions. Understanding both is necessary to reason about what guarantees an AI pipeline actually provides.

Consistency models describe what order of operations a single object (such as a single key in a key-value store) is guaranteed to have when read by different processes. They answer the question: if I write a value and then read it from a different machine, is what I see guaranteed to reflect my write? The three main Jepsen consistency models — Linearizability, Sequential Consistency, and Causal Consistency — each give a different, progressively weaker answer to this question. Eventual consistency is not part of the formal Jepsen hierarchy, but it is widely used in practice and is included here because semantic drift in AI systems maps naturally to it.

Isolation levels describe what a single inference request (acting like a transaction) sees when it reads multiple pieces of context that may be changing concurrently. They answer the question: if another process is updating the retrieval index at the same time my request is running, which version of the data does my request see? The four ANSI SQL isolation levels — Snapshot Isolation, Repeatable Read, Read Committed, and Read Uncommitted — each give a different answer. Stronger isolation prevents more anomalies but requires more coordination and adds cost.

In AI pipelines, both categories apply. Consistency models govern the ordering and agreement of inference outputs across multiple calls to the pipeline. Isolation levels govern what context a single inference call sees during its execution. The two tables below map each category to its AI pipeline equivalent. These mappings reinterpret consistency guarantees over *behavior and semantics* rather than exact state.

Table 1: Consistency Models Applied to AI Pipelines

Consistency Model	Classical Guarantee	AI Pipeline Equivalent	Enforced By
Linearizability	Every operation appears to take effect atomically at a single point in real time. All nodes agree on both the result and the order of operations.	Quorum validation returns a response only when k-of-n independent runs produce semantically equivalent outputs. Two independent calls to the same pipeline must agree before a result is returned. This is approximate: agreement is measured by meaning, not by identical bytes. This provides agreement but does not guarantee real-time ordering. The limitation is semantic rather than temporal: the ordering constraint is relaxed because outputs are compared by meaning equivalence rather than by a global clock.	Quorum-Based Validation (Principle 4).
Sequential Consistency	All operations appear to execute in some consistent global order. Every node agrees on the same sequence, even if that sequence does not exactly match real-world clock time.	Each pipeline step consumes the versioned output of the previous step, and all component versions are pinned in the version manifest. Two runs of the same request with the same manifest will follow the same sequence of operations and see the same component versions. This ensures a consistent execution sequence per request.	Version Everything + Causal Semantic Tracing (Principles 1 and 2).
Causal Consistency	If operation A causally caused operation B (for example, a retrieval caused a subsequent LLM call), then every process that observes B must also have observed A first. Operations with no causal relationship may be seen in any order.	The trace store records and exposes the full causal chain of every inference in the order steps actually caused each other: retrieval then prompt construction then LLM output then tool call. An engineer inspecting the trace always sees steps in causal order, and this holds across parallel branches.	Causal Semantic Tracing (Principle 2).
Eventual Consistency (not in formal Jepsen hierarchy)	All replicas will eventually converge to the same value if no new writes arrive. There is no guarantee about how long convergence takes or what is seen in the meantime.	After a component update (a new model checkpoint, a rebuilt retrieval index, a new embedding model), the system's output distribution shifts and then gradually stabilizes. This is semantic drift. The system eventually converges on a new stable behavior, but it may produce inconsistent outputs throughout the transition period.	Longitudinal drift monitoring via CI/CD (Section 5.6). Detected but not prevented.

Table 2: Isolation Levels Applied to AI Pipelines

Isolation Level	Classical Guarantee (across multiple objects in a transaction)	AI Pipeline Equivalent	Enforced By
Snapshot Isolation	All reads within a transaction see a consistent snapshot of the database taken at the start of the transaction. Changes committed by other transactions after the snapshot started are invisible to this transaction.	A single inference request is pinned to a specific retrieval index snapshot taken at request start. Any index updates that happen while the request is being processed are invisible to it. Every document lookup within that request sees the same consistent view of the data, even if the index is being updated concurrently.	Context snapshot pinning in Version Everything (Principle 1).
Repeatable Read	If you read a row once within a transaction, reading it again returns the same value. No other transaction can change a value you have already read. New rows inserted by other transactions may still appear (phantom reads are allowed).	If the same retrieval query is issued multiple times within a single request pipeline (for example, by two parallel sub-agents), both return the same document set. The index snapshot ID in the version manifest enforces this. New documents added to the index mid-request are not visible.	Index snapshot ID in Version Everything (Principle 1).
Read Committed	You only ever read data that has been fully committed. You will never see half-written data. However, if another transaction commits a change between two of your reads, your second read may return a different value (non-repeatable reads are allowed).	Retrieved documents always come from a fully committed index state — no partially ingested documents are visible. However, two requests made seconds apart may retrieve different documents if the index was updated between them. This is the default behavior of most production RAG systems without snapshot pinning.	Baseline of most production RAG systems without Principle 1.
Read Uncommitted	No isolation guarantees. A transaction can read data that another transaction has written	No context pinning, no version manifests, no snapshot isolation. Each inference call retrieves	No Reliability Plane.

Isolation Level	Classical Guarantee (across multiple objects in a transaction)	AI Pipeline Equivalent	Enforced By
	but not yet committed (a dirty read). If that other transaction rolls back, the data you read never actually existed in a stable state.	whatever documents happen to be in the index at that exact moment, with no guarantee that the view is stable or consistent across calls. This is the default state of most production AI systems that have no Reliability Plane.	

The most important observation from this table is the bottom row. A production AI system deployed without a Reliability Plane — no context pinning, no version manifests, no snapshot isolation — often operates effectively at Read Uncommitted. Every inference call gets an arbitrary, unstable view of the world. Engineers building such systems are, in effect, running a distributed database with no isolation guarantees and no understanding of what consistency model they are providing to their users.

The practical target for most production AI systems should be Snapshot Isolation for context (guaranteed by context snapshot pinning in Principle 1) combined with Causal Consistency for tracing (guaranteed by Principle 2). For high-stakes requests, Quorum-Based Validation (Principle 4) pushes toward approximate Sequential Consistency or Linearizability, at the cost of increased latency and inference spend.

Parallel fan-out and fan-in patterns in the extended PCG introduce additional consistency challenges beyond the sequential case. When multiple sub-agents execute concurrently and their outputs are merged at a fan-in join node, the aggregated context may exhibit anomalies analogous to non-repeatable reads: two sub-agents that retrieved the same underlying documents at slightly different times may have seen different index states, producing context that is internally inconsistent. Snapshot isolation at the retrieval layer — pinning each sub-agent to the same index snapshot ID — is the correct mitigation.

4.7 Tradeoffs: Latency, Accuracy, Determinism, and TCO

- The five principles above introduce competing forces that practitioners must balance. In production AI systems, reliability is not a binary property but a point in a multidimensional design space defined by latency, accuracy, determinism, and total cost of ownership (TCO). Optimizing one dimension typically degrades another.
- **Low latency** minimizes the time from request receipt to response delivery. It is achieved through single-shot inference (no quorum), minimal or asynchronous guardrail checks, in-memory version lookups, and lightweight output validation. These optimizations prioritize responsiveness but reduce opportunities to detect or correct errors.
- **High accuracy** maximizes the probability that each response is factually correct, task-appropriate, grounded in retrieved context, and semantically appropriate. It is achieved through quorum-based validation (Principle 4), grounding and validation checks in output guardrails (Principle 5), and retrieval confidence scoring. These mechanisms improve reliability but increase both latency and compute cost.
- **High determinism** minimizes variance across repeated runs on the same input. It is achieved through context snapshot pinning (Principle 1), fixed random seeds (where supported), and controlled execution via deterministic replay (Principle 3). Causal semantic tracing (Principle 2) enables measurement and diagnosis of non-determinism but does not itself enforce it. Increasing determinism often reduces output diversity and limits the model's ability to explore alternative reasoning paths.

These tradeoffs give rise to common operating configurations:

Configuration	Tradeoff	Best For
High Determinism + High Accuracy	High latency and TCO: quorum validation, full guardrail enforcement, and context verification increase both response time and inference cost.	High-stakes, lower-frequency decisions (e.g., compliance review, medical advisory, financial risk assessment).
High Expected Accuracy + Low Latency	Lower confidence and determinism: single-shot inference may produce accurate results in expectation, but lacks validation and is sensitive to variance across runs.	Low-stakes, high-frequency use cases (e.g., conversational assistants, drafting tools, general Q&A).
Low Latency + High Determinism	Lower accuracy: greedy decoding and rigid templates reduce variance but limit flexible reasoning and may degrade output quality.	Narrow, structured tasks (e.g., classification, data extraction, format conversion).

These dimensions are fundamentally in tension. Increasing determinism often reduces diversity of outputs, while increasing accuracy through redundancy (e.g., quorum validation) increases both latency and cost.

Total Cost of Ownership (TCO) must be considered explicitly alongside latency.

Quorum-based validation increases inference cost linearly with the number of samples. For example, routing 15% of requests through a quorum of $k = 3$ (assuming a baseline of single-shot inference) results in approximately a 30% increase in total inference cost. Hedged requests (Section 4.4) further increase cost while reducing tail latency. Storage costs also grow with the reliability plane: version manifests, semantic traces, and replay artifacts must be retained, though tiered storage with TTL policies keeps this manageable.

The appropriate operating point depends on the risk profile of the application. High-stakes systems should invest in accuracy, determinism, and auditability, accepting higher latency and cost. Low-stakes systems can prioritize latency and cost, accepting higher variance and weaker guarantees. In practice, many systems adopt **tiered reliability**, dynamically applying more expensive mechanisms (e.g., quorum validation, full tracing) only to requests that require higher confidence.

4.8 Semantic Idempotency

For AI components, we define *semantic idempotency*: an operation is semantically idempotent if repeating it with the same input across independent executions (e.g., different random seeds where supported) produces outputs that are semantically equivalent under a task-specific metric—preserving intent, downstream decision, and materially consistent facts, even if the exact wording differs. This is analogous to idempotency in distributed systems, but defined over semantic equivalence rather than exact state equality.

In parallel PCGs, semantic idempotency must be evaluated not only at the node level but for the full graph execution. Small variations at upstream nodes (e.g., router decisions) can lead to different downstream paths, producing a structural form of non-idempotency that per-node analysis will miss. Systems exhibiting frequent path divergence under repeated execution are structurally high-variance, even if individual nodes appear stable.

Quorum-based validation (Principle 4) provides empirical samples that can be used to estimate semantic idempotency using the same task-appropriate metrics described there such as BERTScore (Zhang et al., 2020) or LLM-as-judge approaches (Zheng et al., 2023), enabling data-driven routing decisions (e.g., escalating high-variance requests to more robust execution modes).

5. Reference Architecture

5.1 The Intelligence Plane and the Reliability Plane

The reference architecture separates the AI system into two planes, drawing inspiration from the data plane and control plane separation in storage and networked systems.

The **Intelligence Plane** is the inference pipeline—the components that process a request and produce a response. In systems with parallel sub-agents, the Intelligence Plane is best modeled as a Probabilistic Compute Graph (PCG), containing fan-out, router, and fan-in nodes rather than a simple sequential chain.

The **Reliability Plane** is the infrastructure that wraps, observes, and actively governs the Intelligence Plane. It includes components such as the version registry, trace collector, replay

engine, quorum validator, guardrail enforcer, and a policy layer that determines when and how these mechanisms are applied.

The Intelligence Plane optimizes for capability and throughput; the Reliability Plane optimizes for correctness, consistency, and safety.

Many teams running AI systems in production today have elements of an Intelligence Plane and partial reliability mechanisms, but lack a unified Reliability Plane. This gap is what the reference architecture addresses.

5.2 Architecture Overview

Figure 7 illustrates the two-plane separation at the heart of the reference architecture. Input enters from the top and flows through the Intelligence Plane, while the Reliability Plane operates alongside it, intercepting and governing execution.

The Reliability Plane, shown on the left, contains five primary components: the Version Registry (which versions all components), the Trace Collector (which logs execution at node boundaries), the Replay Engine (which reproduces past requests), the Quorum Validator (which aggregates multiple runs to estimate consensus), and the Guardrail Enforcer (which applies input, output, and flow constraints). A policy layer coordinates when and how these mechanisms are applied.

The Intelligence Plane, shown on the right, contains the Probabilistic Compute Graph (PCG): any combination of DCNs, PCNs, router nodes, context retrievers, LLM inference calls, tool executors (MCP, A2A, API), agents, parallel fan-out/fan-in nodes, acyclic or cyclic sub-PCGs, and nested PCGs of arbitrary depth.

The two planes communicate via a bidirectional interface. The Reliability Plane intercepts inputs and outputs at critical node boundaries of the Intelligence Plane, enforcing guardrails, capturing traces, and looking up version manifests, while minimizing changes to core inference logic. Feedback signals from the Reliability Plane (e.g., validation failures or policy triggers) can influence execution decisions in the Intelligence Plane.

Output exits from the bottom after passing through the Intelligence Plane under the governance of the Reliability Plane.

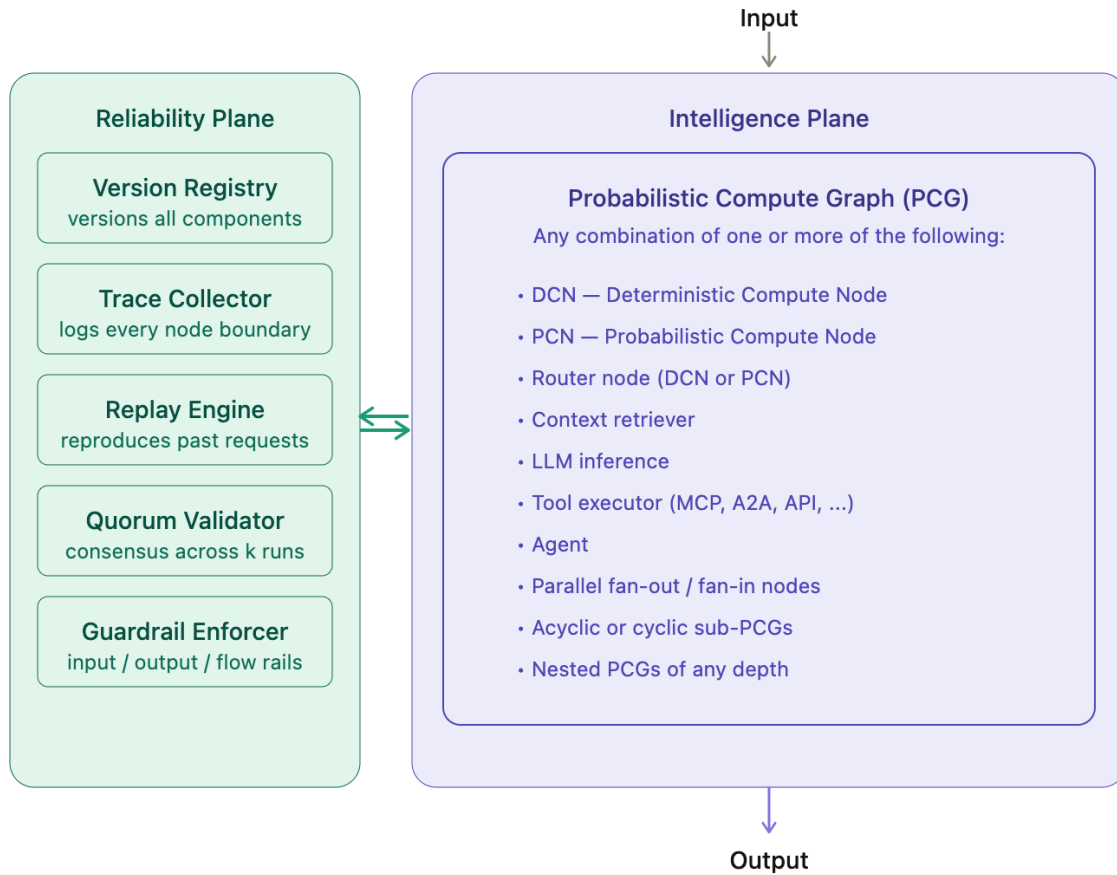


Figure 7 — Intelligence Plane and Reliability Plane Separation

Plane	Components	Role
Reliability Plane	Version Registry, Trace Collector, Replay Engine, Quorum Validator, Guardrail Enforcer, Policy Layer	Observability, versioning, execution governance, consistency enforcement, and replay capability.
Intelligence Plane	PCG nodes — DCNs, PCNs, router nodes, fan-out/fan-in nodes, and sub-PCGs of arbitrary depth and topology. Instantiated as any combination of: context retrievers, LLM inference nodes, tool executors, MCP clients, A2A coordinators, and nested agent sub-graphs.	Request execution across any PCG topology: routing, context retrieval, inference, action execution, and inter-agent coordination.

5.3 Application to Deployment Patterns

Retrieval-Augmented Generation (RAG):

- The Version Registry pins the retrieval index snapshot ID and embedding model version.
- The Trace Collector logs retrieved document IDs (and optionally content or hashes), similarity scores or relevance signals, and the constructed prompt.
- Low-confidence retrievals (based on thresholds over similarity or recall signals) trigger modified prompting (e.g., instructing the LLM to express uncertainty, abstain, or request clarification).
- The Guardrail Enforcer (output rails) applies a heuristic grounding check using LLM-as-judge evaluation (e.g., Zheng et al., 2023) or task-specific metrics, flagging or filtering responses that are not supported by retrieved context.

Parallel Multi-Agent Systems:

- Each sub-agent can be modeled as a PCG, and the inter-agent communication layer forms a higher-level graph whose edges carry messages and context between agents.
- Crucially, the Trace Collector must record fan-out structure, the timing of each sub-agent’s execution, causal relationships across branches, the merge order at fan-in joins, and the realized topology hash. This enables near-complete reconstruction of the execution tree from the trace store.
- The Quorum Validator can be applied at critical decision nodes, not only at the final output, to reduce variance in routing, planning, or tool-selection decisions.
- Snapshot isolation at the retrieval layer (where applicable) ensures that all concurrently executing sub-agents observe the same index state, preventing non-repeatable-read anomalies in the assembled context. Even under snapshot isolation, downstream divergence may still occur due to model stochasticity or tool variability, reinforcing the need for tracing and validation mechanisms.

Model Context Protocol (MCP):

- The Version Registry extends to include MCP server version and tool schema version, including compatibility guarantees (e.g., backward and forward compatibility). Tool schema changes are a significant source of workflow variability and should trigger re-evaluation of action guardrail policies.
- The Guardrail Enforcer (tool/action rails) validates MCP responses against expected schemas before passing them to downstream components, preventing malformed responses from propagating into LLM context. Schema validation ensures structural correctness; additional semantic validation may be required for correctness and completeness.
- The Trace Collector records tool inputs, outputs, and associated schema versions, enabling reproducibility, debugging, and drift analysis across tool interactions.

Agent-to-Agent (A2A) Coordination: In A2A patterns, a planning agent decomposes a task and delegates sub-tasks to specialized agents.

- The Reliability Plane manages this by treating each A2A message as a traceable edge in the PCG, carrying context and metadata between agents. Trace identifiers and parent-child span relationships propagate with each delegation, preserving causal relationships and enabling reconstruction of the complete execution tree.
- Logic/Flow guardrails on the planning agent enforce maximum delegation depth, fan-out limits, and cycle detection, preventing runaway or circular A2A delegation.

- Action guardrails constrain what actions, tools, and resources a planning agent is permitted to invoke on downstream agents, ensuring that delegation remains within defined policy boundaries.
- Failures in delegated sub-tasks (e.g., timeouts or invalid responses) are captured in the trace and handled through retries, fallback strategies, or escalation, maintaining system stability under partial failures.

5.4 Performance, Cost, and Management Tradeoffs

The following estimates assume a single-region deployment with lightweight guardrail models and moderate request complexity; actual overhead varies by architecture and model size.

Component	Latency Overhead	Why / Optimization
Version Registry lookup	~1–2ms	A network call on a cache miss to a key-value store. Near-zero with an in-memory cache (short TTL ~5s).
Trace Collector (async)	Near-zero on the critical path under non-blocking buffering	Trace events are buffered in memory and written asynchronously. Flush interval (e.g., 100ms) is configurable.
Input Guardrail	~5–20ms	Pattern matching and schema validation: low single-digit ms. LLM-based input classification (e.g., prompt injection detection): 10–20ms. Assumes lightweight or specialized models.
Output Guardrail	~10–50ms	Schema validation is fast. Grounding checks via LLM-as-judge add 30–50ms. Can run async for non-blocking validation paths.
Quorum Validation (send 5, return on 3)	~1.5–2x inference latency (p50) depending on model variance; better p99 vs naive k=3	5 requests sent in parallel; wait for 3rd fastest of 5. TCO: ~67% higher than naive k=3 quorum (5 vs 3 calls), and 5× the cost of single-shot inference for covered requests. Mitigate by applying only to high-risk traffic (10–20%)
Replay Engine	Offline only	No runtime overhead. ~50KB per trace; ~55GB/year at 3,000 reqs/day (50KB × 3,000 × 365). Tiered storage (hot 30d, warm 1yr, cold archival) keeps cost manageable.

For low-risk requests, the Reliability Plane adds approximately 15 to 70ms, a small fraction of typical LLM inference latency of 500ms to 3,000ms. For high-risk requests using the k+r hedge strategy, p50 latency roughly doubles but p99 improves meaningfully.

Aggregate inference cost depends on the fraction of traffic routed through quorum validation. Assuming a baseline of single-shot inference:

- For **k = 3 quorum**, each covered request incurs 2 additional model calls (3 – 1).
 - If 10% of traffic uses quorum: $0.10 \times 2 = 20\%$ total cost increase.
 - If 20% of traffic uses quorum: $0.20 \times 2 = 40\%$ total cost increase

- For **hedged quorum (send 5, return on 3)**, each covered request incurs 4 additional calls (5 – 1):
 - 10% traffic → 40% increase
 - 20% traffic → 80% increase

Hedged quorum should be applied selectively to the highest-risk subset of requests to balance cost and tail-latency benefits.

Trace storage costs scale with both request volume and PCG complexity, as each node boundary generates additional trace events.

In practice, the marginal cost of reliability is small relative to the base cost of inference, making selective application of the Reliability Plane economically viable.

5.5 What This Architecture Does Not Solve

The architecture manages and bounds non-determinism; it does not eliminate it.

Quorum validation improves confidence in outputs but cannot compensate for a fundamentally poor model or systematically incorrect inputs (e.g., flawed retrieval or biased training data).

Replay is approximate rather than exact, due to infrastructure variability, external dependencies, and potential changes in model behavior or provider implementations over time.

Semantic drift can be detected and monitored, but not fully prevented, as it arises from cumulative changes across models, data, and external systems.

Guardrails provide important safety and consistency constraints, but they are not a complete defense against sophisticated adversarial inputs and may introduce false positives that degrade usability.

Systems with highly dynamic PCG topologies—where graph structure is determined at runtime—require extensions to the tracing model (e.g., dynamic topology capture and runtime graph reconstruction) as discussed in Section 8.2.

Finally, stronger reliability guarantees come with inherent tradeoffs in latency and cost; achieving higher confidence or determinism requires additional computation, validation, and infrastructure.

5.6 CI/CD for AI Systems

CI/CD for AI systems extends traditional pipelines with AI-specific validation and reliability stages:

- **Model version pinning and promotion gates:** New model versions are evaluated in staging against representative query sets using heuristic metrics (e.g., LLM-as-judge scoring). Multiple samples per query should be used to measure both average quality and variance. Promotion to production requires meeting configured thresholds.
- **Prompt regression testing:** Every prompt template change triggers automated evaluation against a baseline (typically the previous production version under the same version manifest). Degraded output quality or increased variance blocks deployment.

- **PCG topology regression testing:** For systems with router nodes and conditional branching, the pipeline verifies expected branch coverage and routing distribution across a representative query set. Metrics such as branch coverage percentage and routing diversity detect unintended collapse (e.g., the system always selecting a single path).
- **Context index validation:** When the retrieval index is updated, validation jobs verify that retrieval quality metrics (e.g., recall, mean reciprocal rank) have not degraded. These checks must be coupled with embedding model versioning, as embedding changes directly affect retrieval behavior.
- **Guardrail policy testing:** Changes to guardrail policies are tested against adversarial inputs and benign inputs to measure both detection effectiveness and false positive rates before deployment.
- **Longitudinal evaluation baselines:** Golden query sets with human-verified reference responses are executed on every deployment. Semantic drift is measured using task-appropriate similarity or scoring metrics; deviations above a configured threshold trigger alerts.
- **Trace-based replay testing:** Historical traces are replayed against new configurations (models, prompts, or policies) to evaluate behavior changes on real-world inputs before rollout.
- **Rollback capability:** Every component in both planes supports rapid rollback. Version manifests—including `pcg_topology_hash`—enable rollback of topology, model, prompt, index, and policy changes as a consistent unit.

5.7 Prompt Engineering, Context Engineering, and Evaluation Strategies

The five architectural principles manage the reliability infrastructure around AI pipelines. Complementary to these, prompt engineering and context engineering directly shape the behavior of LLM components, influencing output quality, reducing variance, and improving groundedness.

Prompt Engineering

Prompt engineering is the practice of designing and iterating on inputs to a language model to reliably produce accurate, grounded, and structured outputs (Sahoo et al., 2024). Key techniques include:

- **Chain-of-Thought (CoT) prompting** instructs the model to produce intermediate reasoning steps before giving a final answer, improving performance on complex multi-step tasks and making intermediate reasoning steps visible (though not necessarily faithful), complementing Causal Semantic Tracing.
- **Few-shot prompting** provides examples of desired input-output behavior within the prompt, often narrowing the output distribution and reducing response variance.
- **Prompt chaining** decomposes complex tasks into a sequence of simpler sub-prompts, mapping naturally onto the PCG model and making each step independently testable and replayable.
- **Output format constraints** (e.g., JSON schemas, structured formats) reduce the space of valid outputs and lower variance for downstream consumers, reinforced by output guardrails.

Prompt templates should be versioned as first-class artifacts (Principle 1), and changes should trigger automated prompt regression tests in the CI/CD pipeline.

Context Engineering

Context engineering is the systematic design, management, and optimization of all information provided to an LLM at inference time—system instructions, retrieved documents, tool definitions, conversation history, and task metadata. Key reliability practices include:

- **Token budgeting:** managing information density to prioritize high-relevance content, mitigate position-related attention effects, and avoid exceeding effective context limits.
- **Provenance tags:** annotating retrieved content with source identifiers and retrieval timestamps, enabling grounding verification and auditability.
- **Context versioning and observability:** logging the exact context supplied to each inference call as part of causal semantic tracing.
- **Context snapshots:** pinning the retrieval index snapshot per request, enabling replay and enforcing snapshot isolation (Section 4.6).

Evaluation Strategies

Prompt and context changes are among the most common sources of silent behavioral regression. Key testing strategies include:

- **A/B evaluation:** running existing and proposed prompt or context configurations against the same representative queries and comparing outputs using automated scoring.
- **LLM-as-judge evaluation:** using a separate LLM instance as a structured evaluator to score responses for accuracy, groundedness, and coherence (Zheng et al., 2023). These methods are heuristic and depend strongly on well-defined evaluation criteria.
- **Behavioral test suites:** maintaining golden queries with human-verified reference outputs to detect output drift and regressions over time.
- **Variance testing:** executing multiple runs per query to measure output stability and estimate semantic idempotency (Section 4.8).
- **Adversarial prompt testing:** including prompt injection attempts and edge cases to verify that guardrails remain effective and that prompt or context changes do not weaken defenses.

6. Design Walkthrough: Financial Services AI Pipeline

We illustrate the framework with a representative financial services AI system combining RAG, MCP-based tool access, and A2A coordination.

System Overview

An analyst-facing assistant:

- RAG over internal filings and reports
- MCP integration for live regulatory data
- A planning agent delegating to (A2A):
 - a document analysis agent (RAG)
 - a regulatory lookup agent (MCP)
- Results merged at a fan-in node and synthesized

Observed Failure Modes (System V1: No Reliability Plane)

- **Inconsistent responses:** identical queries yield different answers due to index updates and model stochasticity
- **Ungrounded outputs:** LLM produces plausible but unsupported claims due to weak retrieval
- **Silent behavioral drift:** model/provider updates change output characteristics without visibility
- **Cascading failures:** malformed tool output propagates through parallel agents and contaminates final response

System V2: With Reliability Plane

- **Versioning + tracing:** index snapshots and retrieved documents are logged, enabling root-cause analysis of inconsistencies
- **Retrieval-aware prompting + grounding checks:** low-confidence retrieval triggers uncertainty-aware responses; guardrails enforce grounding
- **Model version tracking + drift detection:** checkpoint-level versioning and longitudinal evaluation detect silent behavioral changes
- **Tool validation + graceful degradation:** schema validation prevents malformed tool outputs from entering the pipeline; failures trigger retries and fallback behavior

The Reliability Plane does not eliminate failures—it makes them observable, diagnosable, and containable.

7. Related Work

7.1 Distributed Systems Foundations

This paper draws on five decades of distributed systems research. Lamport (1978) introduced causal ordering, which we draw on and apply to semantic tracing in AI pipelines. The FLP impossibility result (Fischer, Lynch & Paterson, 1985) established fundamental limits on achieving consensus in asynchronous systems, motivating designs that manage rather than eliminate failure. The CAP theorem (Gilbert & Lynch, 2002) demonstrated inherent tradeoffs between consistency and availability. Amazon’s Dynamo (DeCandia et al., 2007) and Google’s Spanner (Corbett et al., 2012) illustrate how these principles are realized in production systems.

The Byzantine Generals Problem (Lamport, Shostak & Pease, 1982) provides a model for adversarial fault tolerance, which we apply analogically to classify prompt injection as an AI-specific form of Byzantine behavior. Distributed tracing systems such as Dapper (Sigelman et al., 2010) and OpenTelemetry provide the foundation for observability, which we extend with semantic content capture. Kingsbury’s Jepsen framework and Bailis et al. (2013) on Highly Available Transactions formalize the consistency hierarchy that we reinterpret for AI pipelines in Section 4.6.

Gap: These systems were designed for environments where correctness is defined over system state and failures arise from node crashes, partitions, or adversarial behavior. AI pipelines introduce a distinct failure mode: components that are operationally correct at the infrastructure level but produce probabilistically incorrect or inconsistent outputs. This requires extending distributed systems reasoning from deterministic state transitions to probabilistic computation.

7.2 ML Reliability and Reproducibility

Sculley et al. (2015) identified hidden technical debt in machine learning systems, highlighting the complexity of maintaining production ML pipelines. Pineau et al. (2021) focused on reproducibility in machine learning research, emphasizing the need for consistent training and evaluation practices. Zaharia et al. (2018) introduced MLflow, providing infrastructure for model tracking, versioning, and lifecycle management. Paleyes et al. (2022) surveyed challenges in deploying and maintaining machine learning systems in production.

Gap: This literature largely focuses on training reproducibility, model lifecycle management, and deployment pipelines. While these works address reliability at the level of model development and system integration, they do not directly address runtime non-determinism in inference pipelines or the consistency properties of systems where model outputs are probabilistic. In particular, they do not provide a framework for reasoning about variance, reproducibility, and consistency in multi-stage or multi-agent AI systems.

This paper extends the notion of ML reliability from training-time reproducibility to runtime behavior, framing inference pipelines as probabilistic distributed systems and introducing mechanisms to manage non-determinism at execution time.

7.3 RAG, Agents, and Emerging Protocols

Lewis et al. (2020) introduced Retrieval-Augmented Generation (RAG) as a method for combining parametric and non-parametric knowledge. Subsequent work has characterized its limitations: Gao et al. (2023) identify common failure modes in RAG systems, including retrieval errors and hallucination. Shi et al. (2023) demonstrate context distraction effects, motivating techniques such as retrieval confidence estimation and guardrails. ReAct (Yao et al., 2023) established multi-step reasoning and tool-use patterns for agent-based systems, and the RAGAS framework (Es et al., 2023) provides evaluation metrics for assessing RAG output quality.

These works primarily focus on improving model behavior, prompt design, retrieval quality, and evaluation methodologies.

Gap: This literature largely does not address system-level reliability infrastructure for AI pipelines. In particular, it does not provide mechanisms for versioning, tracing, replay, or consistency management across multi-stage or multi-agent systems. Nor does it address the consistency properties of parallel execution, including coordination challenges and fan-in join anomalies that arise when multiple probabilistic components are composed (Section 4.6).

This paper complements existing work by introducing a distributed-systems-inspired framework for managing non-determinism and ensuring reliability at the level of the full system, rather than individual components.

7.4 Prompt Engineering and Context Engineering

Wei et al. (2022) established chain-of-thought prompting as a mechanism to improve multi-step reasoning. Sahoo et al. (2024) provided a systematic survey of prompt engineering techniques. Mei et al. (2025) formalized context engineering as a discipline encompassing the retrieval, processing, and management of the full information payload provided to an LLM at inference time.

These works focus primarily on improving model behavior through prompt design, context selection, and information structuring.

Gap: This literature largely does not address integration with system-level reliability infrastructure. In particular, it does not provide mechanisms for versioning prompt and context configurations, tracing and auditing their effects, replaying executions under controlled conditions, or evaluating their impact on system-level consistency and variance.

This paper complements existing work by connecting prompt and context engineering to a broader reliability framework, where these techniques reduce variance at the component level while system-level mechanisms (e.g., tracing, replay, and quorum validation) manage variability across the full pipeline.

7.5 AI Safety and Guardrails

Bai et al. (2022) introduced Constitutional AI as a model-level approach to improving safety through guided training and self-critique. Rebedea et al. (2023) introduced NeMo Guardrails for inference-time policy enforcement. Perez & Ribeiro (2022) characterized prompt injection attacks against LLMs, which this paper maps analogically to Byzantine fault behavior in the PCG model (Section 3.3).

These works focus on improving safety through model design, prompt constraints, and policy enforcement mechanisms, often treating safety as a set of isolated checks (e.g., input/output filtering or tool access control).

Gap: Existing guardrail frameworks largely do not address integration with system-level reliability infrastructure. In particular, they do not incorporate versioning, tracing, replay, or quorum-based validation as part of a unified architecture. Nor do they explicitly address how guardrails should operate across multi-stage and parallel execution patterns, including enforcement at node boundaries and fan-in joins where multiple probabilistic outputs are combined.

This paper extends prior work by treating guardrails as first-class components within a broader reliability architecture, integrating them with versioning, tracing, and validation mechanisms, and applying them systematically across the full PCG, including parallel and multi-agent execution.

8. Limitations and Future Work

8.1 Limitations

- Non-determinism cannot be eliminated. The framework manages and bounds its consequences, not its underlying causes.
- Replay is approximate rather than exact. GPU floating-point non-determinism, infrastructure variability, and potential changes in model versions or provider behavior mean that even fixed seeds cannot guarantee token-level reproduction. If the original model version is no longer available, exact replay becomes impossible.
- Semantic similarity metrics are imperfect. Methods such as BERTScore and LLM-as-judge evaluations are themselves probabilistic, task-dependent, and subject to evaluator bias. As a result, quorum validation reliability is bounded by the quality and stability of the evaluation metric.
- Dynamic topology limitations. The PCG model assumes a relatively stable graph structure. Systems where topology is fully determined at runtime require extensions to tracing and replay mechanisms, including explicit capture of dynamic graph construction and router decisions.
- Consistency model mapping is analogical. The consistency hierarchy adapted from Jepsen (Section 4.6) provides a framework for reasoning, not a formal equivalence. Because LLM outputs are semantic and probabilistic, consistency must be understood in distributional terms rather than exact state equality.
- Observability and privacy tradeoffs. Causal semantic tracing requires capturing inputs, outputs, and context, which introduces storage overhead and potential privacy or compliance concerns. Practical deployments must balance trace fidelity with data minimization and access control.
- Implementation cost. Building and operating the Reliability Plane introduces non-trivial engineering and operational complexity. The framework is most applicable to production systems at scale, where the benefits of improved reliability justify the additional infrastructure and cost.

8.2 Future Work

- **Formal consistency models for AI pipelines:** Developing formal specifications of semantic consistency for PCGs—analogueous to linearizability and causal consistency in distributed systems—would enable reasoning about guarantees such as output stability, bounded variance, and reproducibility across executions. Such models could form the basis for formal verification of AI system behavior.
- **Causal semantic tracing for dynamic topologies:** When graph structure is determined at runtime, tracing infrastructure must dynamically discover and record execution topology, including router decisions, fan-out/fan-in relationships, and causal ordering. Supporting replay and analysis in such systems requires new mechanisms for capturing and reconstructing evolving execution graphs.
- **Automated semantic idempotency profiling:** Continuous tooling to measure semantic idempotency across query types and execution paths would enable detection of high-variance regions in the PCG. These signals could drive adaptive routing decisions, selective quorum application, or prompt and context adjustments.
- **Empirical evaluation across deployment contexts:** Large-scale empirical studies are needed to quantify the impact of the Reliability Plane on production systems, including

reductions in hallucination rates, output variance, debugging time, and drift detection latency across different domains and workloads.

- **Formal analysis of fan-in consistency anomalies:** The non-repeatable-read-like anomalies that arise at fan-in join nodes, where multiple probabilistic outputs are merged, deserve formal study analogous to isolation anomalies in distributed databases. This includes modeling the impact of merge order, partial failures, and asynchronous completion on downstream behavior.
- **Cost-aware reliability optimization:** Future work should explore adaptive strategies that balance reliability, latency, and cost e.g., selectively applying quorum validation, guardrails, or tracing depth based on predicted risk or query characteristics.

9. Conclusion

AI pipelines are not deterministic software systems that occasionally behave unexpectedly. They are probabilistic distributed systems—networks of components whose outputs depend on inputs, model state, and stochasticity—and they should be designed and operated accordingly. In multi-agent systems, this structure generalizes to a directed graph with parallel fan-out, conditional routing, and fan-in joins, introducing consistency challenges that sequential pipeline models do not capture.

This paper has argued that managing AI non-determinism does not require entirely new tools. Many of the necessary concepts already exist in distributed systems and transfer effectively, with adaptation, to AI pipelines. We formalized AI pipelines as Probabilistic Compute Graphs supporting parallel execution; mapped AI failure modes onto distributed systems concepts; derived five design principles grounded in established precedents; and instantiated these in a reference architecture separating the Intelligence Plane from the Reliability Plane.

The consistency model mapping—from Read Uncommitted for unpinned production systems, to Snapshot Isolation for context pinning, to Causal Consistency for tracing, and approximate Linearizability for quorum validation—provides practitioners with a vocabulary for reasoning about the guarantees their AI systems provide. Many production AI systems today operate at effectively Read Uncommitted consistency without explicitly recognizing it.

The distributed systems community learned that reliability cannot be added after the fact; it must be designed into the system. AI systems are now encountering the same reality. The Reliability Plane is one way to make that principle concrete for probabilistic computation.

References

1. Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The Curious Case of Neural Text Degeneration. ICLR.
2. PyTorch. (2025). Reproducibility. <https://pytorch.org/docs/stable/notes/randomness.html>.
3. Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. NeurIPS.
4. Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. IEEE TPAMI.
5. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Hopkins, M., Luck, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. TACL.
6. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. ICLR.
7. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... & Shanbhag, C. (2010). Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report.
8. Fowler, M. (2005). Event Sourcing. <https://martinfowler.com/eaDev/EventSourcing.html>
9. Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2020). BERTScore: Evaluating Text Generation with BERT. ICLR.
10. Zheng, L., Chiang, W. L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., ... & Stoica, I. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. NeurIPS.
11. Nygard, M. (2007). Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf.
12. Kingsbury, K. Jepsen: Distributed Systems Safety Analysis. <https://jepsen.io/consistency> Accessed April 2026
13. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J. M., & Stoica, I. (2013). Highly Available Transactions: Virtues and Limitations. Proceedings of the VLDB Endowment, 7(3), 181–192.
14. Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927.
15. Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. CACM, 21(7), 558–565.
16. Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of Distributed Consensus with One Faulty Process. JACM, 32(2), 374–382.
17. Gilbert, S., & Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News, 33(2), 51–59.
18. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's Highly Available Key-Value Store. SOSP.
19. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., ... & Woodford, D. (2012). Spanner: Google's Globally Distributed Database. OSDI.
20. Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. ACM TOCS, 4(3), 382–401.
21. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., ... & Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. NeurIPS.
22. Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché Buc, F., ... & Larochelle, H. (2021). Improving Reproducibility in Machine Learning Research. JMLR, 22(242), 1–20.
23. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., ... & Talwalkar, A. (2018). Accelerating the Machine Learning Lifecycle with MLflow. CIDR.
24. Paleyes, A., Urma, R. G., & Lawrence, N. D. (2022). Challenges in Deploying Machine Learning: A Survey of Case Studies. ACM Computing Surveys.
25. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS.

26. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... & Wang, H. (2023). Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997.
27. Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E. H., ... & Zhou, D. (2023). Large Language Models Can Be Easily Distracted by Irrelevant Context. ICML.
28. Es, S., James, J., Espinosa-Anke, L., & Schockaert, S. (2023). RAGAS: Automated Evaluation of Retrieval Augmented Generation. arXiv:2309.15217.
29. Mei, L., et al. (2025). A Survey of Context Engineering for Large Language Models. arXiv:2507.13334
30. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., ... & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. NeurIPS.
31. Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., ... & Kaplan, J. (2022). Constitutional AI: Harmlessness from AI Feedback. Anthropic Technical Report.
32. Rebedea, T., Dinu, R., Sreedhar, M., Erikson, C., & Kirk, J. (2023). NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications. EMNLP.
33. Perez, F., & Ribeiro, I. (2022). Ignore Previous Prompt: Attack Techniques for Language Models. NeurIPS ML Safety Workshop.
34. Anthropic. (2024). Model Context Protocol. <https://www.anthropic.com/news/model-context-protocol>
35. Google. (2025). A2A: A New Era of Agent Interoperability. <https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/>