

How to become a Staff AI Engineer

Chapter 7: Distributed Systems for AI Engineering

Celso Sousa

LinkedIn: <https://www.linkedin.com/in/celso-sousa/?locale=en-US>

Portfolio: <https://celsosousa.com.br/project-portfolio/>

Ebook: <https://celsosousa.com.br/ebook-staff-ai-engineer/>

Chapter Objective

Production AI systems rarely run on a single machine. Models, databases, queues, retrieval mechanisms, APIs, caches, pipelines, and agents are typically distributed across multiple processes, services, hosts, regions, and providers.

When an architecture crosses the network, problems emerge that do not exist in the same way in local systems. Messages can be delayed, nodes can partially fail, networks can experience partitions, different components can observe different versions of state, and the same operation can be executed more than once.

For a Staff AI Engineer, Distributed Systems is a fundamental discipline because it determines whether an architecture will remain correct, available, and economically sustainable as it grows in volume, criticality, and number of dependencies.

The objective of this chapter is not to memorize patterns or classify technologies, but to develop a mental model for managing distributed uncertainty. Several principles will recur throughout the chapter:

- **Partial failures are inevitable:** Some components may continue operating while others fail, creating intermediate states that must be handled explicitly;
- **Consistency has a cost:** Stronger guarantees typically require greater coordination, potentially increasing latency or reducing availability;
- **Retries can duplicate operations:** Recovering from transient failures requires idempotency, deduplication, and clearly defined retry limits;
- **Replication increases availability:** However, it introduces synchronization, divergence, stale reads, and additional cost;
- **Sharding increases capacity:** However, it complicates global queries, rebalancing, workload distribution, and recovery;
- **Asynchrony increases decoupling and resilience:** But it requires explicitly handling eventual consistency, intermediate state, and distributed observability;
- **Horizontal scale increases capacity:** However, it also increases the number of network calls, failure modes, and coordination mechanisms required.

The primary mindset shift is recognizing that, in distributed systems, controlling the logical state of the software is not enough. It is also necessary to control its behavior in the presence of delays, duplication, unavailability, divergence, and uncertainty about the outcome of remote operations.

7.1. Distributed Systems Fundamentals and Consistency

A distributed system consists of multiple independent components that cooperate through a network. From the user's perspective, these components may appear to form a single application. Internally, however, each service has its own state, resources, lifecycle, and possibility of failure. This characteristic fundamentally changes how correctness, reliability, and architecture must be analyzed.

7.1.1. Partial Failures and Network Partitions

A partial failure occurs when only part of the architecture fails while other components remain operational. This property fundamentally distinguishes distributed systems from local programs. In a local application, a healthy process can generally continue executing its local operations. In a distributed architecture, a service can be perfectly healthy and still be unable to complete a request because a remote dependency does not respond.

This condition creates several forms of uncertainty:

- **A healthy service may be unable to complete a task:** A database, retrieval service, model provider, or another dependency may be unavailable;
- **An operation may have been executed even when its response never arrived:** The client cannot determine from the timeout alone whether the call should be retried;
- **A region may become isolated while the rest of the platform remains operational:** The architecture must determine which capabilities can continue functioning;
- **A component may become slow without becoming completely unavailable:** Slowdowns consume resources while new requests continue accumulating;
- **A locally observed failure may originate in another component:** Diagnosis requires understanding the complete request path.

Consider an agent that requests the creation of an order in an external system. The service executes the operation, but the response is lost. The agent receives a timeout.

At this point, there is a critical distinction between not receiving confirmation and knowing that the operation did not occur. Retrying the call without protective mechanisms may create a second order. Partial failure, therefore, is not merely an availability problem. It is also a correctness problem.

A fundamental Staff-level question is: **what happens if the destination executes the action, but the caller never receives confirmation?**

Answering this question correctly leads directly to idempotency, deduplication, reconciliation, and state recovery.

Network partitions amplify this problem. A partition occurs when parts of a system can no longer communicate even though their individual components remain operational.

The causes can vary:

- **Physical or logical failures can interrupt communication:** Hosts remain active but stop exchanging messages;
- **Routing problems can isolate parts of the infrastructure:** Services continue processing local traffic without access to other regions;
- **Regional incidents can separate parts of the topology:** An entire region may lose connectivity;

- **Misconfigurations can produce isolation:** Firewalls, ACLs, DNS, or service discovery may prevent communication;
- **Failures may affect only specific paths:** Not all dependencies necessarily become unavailable simultaneously.

During a partition, different components may continue accepting requests and modifying state. A fundamental decision then emerges: **should the system continue operating without synchronization, or should specific operations stop to prevent divergence?**

A recommendation system may continue using slightly stale information. A financial system may not be able to authorize conflicting operations simultaneously across isolated regions. The decision must be driven by the semantics of the operation, not merely by the technology being used.

7.1.2. Time, Causality, and Distributed Coordination

Distributed machines do not share a perfectly synchronized clock. Even small differences can create problems when timestamps are used as evidence of ordering, ownership, or causality.

Timestamps appear in many architectural decisions:

- **Event ordering may depend on timestamps:** However, different clocks do not guarantee causal ordering;
- **Expiration may depend on local time:** Tokens, leases, and caches may expire at different moments;
- **Ownership may depend on time intervals:** Clock divergence can create conflicting interpretations;
- **Distributed locks may use leases:** Clock drift can alter the perception of whether ownership is retained or lost;
- **Versioning may use timestamps:** Concurrent events may appear ordered even when no causal relationship exists.

In AI systems, these differences can affect document freshness, event ordering, data versions, and auditability. In environments where event reconstruction is important, a mature architecture typically combines several mechanisms:

- **Clock synchronization reduces divergence:** However, it does not turn distributed clocks into a perfect source of causality;
- **Correlation IDs connect related operations:** They make it possible to follow a request across multiple components;
- **Event IDs identify events independently of time:** Identity remains stable even when clocks diverge;
- **Structured logs preserve context:** Relevant information can be searched and correlated;
- **Distributed tracing reconstructs the request path:** Operational causality can be observed across services.

The central principle is that an isolated timestamp should not automatically be treated as absolute proof of causality. Distributed coordination becomes necessary when multiple participants must agree on a shared decision. This occurs, for example, in leader election, distributed locking, work assignment, version ownership, or exclusive updates.

However, strong coordination comes at a cost:

- **Distributed locks increase latency:** Operations become dependent on additional communication;
- **Coordinators can become bottlenecks:** A central component begins influencing the entire system;
- **Failures can make ownership ambiguous:** Leases, timeouts, and fencing may be required;
- **Coordination reduces parallelism:** Participants must wait for collective decisions;

- **Blast radius can increase:** Independent workloads become dependent on the same coordination infrastructure.

A powerful strategy is to reduce the amount of coordination required. Partitioning work so that each worker has exclusive ownership over a set of data may be better than continuously coordinating all workers.

The principle is: **distributed coordination is expensive. When the domain allows it, eliminate the need for coordination rather than merely building more sophisticated coordination mechanisms.**

7.1.3. Consistency Models, CAP, and PACELC

Consistency determines which versions of state can be observed by different participants. In local applications, we usually assume that a completed write will be available to subsequent reads. In distributed systems, replication, caches, indexes, and asynchronous processing create different representations of the same state. The architecture must define which guarantee is actually required.

Strong consistency simplifies part of the application's reasoning:

- **Fewer intermediate states need to be handled:** Developers can reason about a more uniform view;
- **Decisions based on stale versions become less likely:** This is important for sensitive operations;
- **Business rules can become simpler:** Less reconciliation is delegated to the application;
- **Conflicts become more constrained:** The infrastructure assumes greater responsibility for ordering and synchronization.

This logical simplicity, however, transfers complexity to the infrastructure:

- **Greater coordination may be required:** Replicas must confirm or order operations;
- **Latency may increase:** Writes may wait for communication among participants;
- **Availability may decrease during partitions:** Operations may be rejected to prevent divergence;
- **Throughput may decrease:** Coordination limits parallelism for certain workloads.

Strong consistency may be required for balances, critical inventory, authorization, access policies, or irreversible transactions. Applying it indiscriminately to telemetry, analytics, caches, and recommendations, however, may add cost without producing proportional value.

Eventual consistency temporarily allows different versions of state, provided that they converge later. A RAG pipeline naturally illustrates this behavior. A document may be stored immediately, while parsing, chunking, embedding generation, and indexing take seconds or minutes. During this interval, the document exists in the source of truth but cannot yet be retrieved.

Eventual consistency provides several advantages:

- **Availability tends to increase:** Components can operate without waiting for global synchronization;
- **Scalability improves:** Replicas and workers can operate with greater independence;
- **Write latency may decrease:** Not every participant needs to confirm immediately;
- **Asynchronous pipelines become natural:** Different representations are progressively updated.

On the other hand, the application must explicitly handle the consequences:

- **Stale reads may occur:** The user may observe an earlier version;
- **Requests issued close together may produce different results:** Replicas may be at different stages;

- **Conflicts may require reconciliation:** Concurrent changes require resolution rules;
- **The user experience may become confusing:** Changes that are still being processed need to be represented;
- **Troubleshooting becomes more complex:** The system must distinguish errors from expected delays.

Eventual consistency does not mean the absence of guarantees. It means consciously accepting a window of divergence. The relevant question is: **what inconsistency window can the business tolerate?**

The CAP theorem helps structure this decision during network partitions. When components lose communication, the architecture must determine which operations can remain available without violating the required consistency guarantees.

It is not useful to treat CAP merely as a permanent classification of technologies into "CP" or "AP". At Staff level, the more relevant question is: **during a concrete partition, which operation can continue and which one must stop?**

The same product may make different decisions depending on the data. Recommendations may remain available using stale information, while authorization changes may require much stronger guarantees.

PACELC extends this reasoning by showing that the trade-off does not disappear when the network is healthy. Even without partitions, stronger consistency typically requires more coordination and, consequently, higher latency:

- **Waiting for more participants can strengthen guarantees:** However, it increases the critical path;
- **Responding faster reduces latency:** But it creates a window in which replicas observe different versions;
- **Geographic replication reduces distance to users:** However, it increases the synchronization path;
- **Global guarantees simplify the application:** But they may make interactions significantly slower.

In many scenarios, the correct guarantee does not need to be global strong consistency. Read-after-write consistency, for example, can guarantee that a user immediately observes their own update without requiring every system replica to be synchronized.

Some strategies include:

- **Sticky routing:** The user remains temporarily associated with a replica that knows about their write;
- **Primary reads after an update:** Critical reads query a more up-to-date source;
- **Version awareness:** The system verifies whether the queried representation has incorporated a specific version;
- **Explicit processing state:** The interface indicates that an update is still being indexed.

This is an important Distributed Systems principle: use the minimum guarantee that correctly satisfies the requirement rather than indiscriminately applying the strongest guarantee available.

7.2. Reliable Processing and Resilience

Distributed messages can be lost, repeated, or processed multiple times. Therefore, an architecture must explicitly define which delivery and effect guarantees it provides. Likewise, failures should not be treated as extraordinary exceptions. They are part of the normal operation of distributed systems.

7.2.1. Delivery Semantics, Idempotency, and Deduplication

At-most-once, at-least-once, and exactly-once represent different trade-offs. At-most-once means that an operation will be executed no more than once. If confirmation or processing fails, the message may be lost. This behavior reduces overhead but accepts loss as a possibility. It may be sufficient for certain telemetry events, but it is typically inappropriate for payments or critical processing.

At-least-once attempts to guarantee processing at least once. It typically uses retries, which means that duplicates become possible.

Consider an embedding pipeline. A worker processes a document but fails before acknowledging the message. The broker delivers the same message again. The consumer must detect that the work has already occurred or execute it again safely.

This model has several important properties:

- **Silent loss becomes less likely:** Unacknowledged operations can be retried;
- **Transient failures can be recovered:** Workers can continue after periods of unavailability;
- **The infrastructure remains relatively simple:** It is not necessary to physically prevent every duplicate;
- **Complexity is transferred to the consumer:** Processing must be idempotent or deduplicated.

In many systems, controlled duplication is a more manageable problem than silent loss. Exactly-once, in turn, must be interpreted carefully. Guaranteeing that a message is physically delivered only once in the presence of network failures is extremely difficult. However, systems can be built in which the logical effect of an intent is observed only once.

This usually combines several mechanisms:

- **Transactions group related changes:** State is committed or rolled back within the available guarantees;
- **Deduplication identifies repeated messages:** A new delivery does not necessarily produce a new effect;
- **Idempotent consumers enable replay:** The same intent can be processed again safely;
- **Transactional logs preserve sequence and state:** The application can reconstruct what was actually applied.

A payment may reach a service twice. If both requests carry the same idempotency key, the system can recognize that they represent the same intent and produce only one financial effect. In practice, at-least-once delivery + idempotency is often a more realistic way of achieving exactly-once-equivalent behavior. Idempotency therefore represents one of the most valuable properties of a distributed architecture.

Several strategies are particularly useful:

- **Idempotency keys identify a logical operation:** Repetitions using the same identifier do not produce multiple effects;
- **Upserts make certain writes repeatable:** The same resource converges to the same state;
- **State checks verify previous effects:** The application queries persisted state before modifying it again;
- **Deduplication tables record processed operations:** Repeated messages can be recognized;
- **Deterministic keys stabilize identity:** The same input produces the same persisted identity.

In agentic systems, this becomes even more important. An agent may execute a tool, receive a timeout, and retry the action. If the tool only performs a read, the risk may be small. If it creates resources, sends messages, makes purchases, or transfers money, duplication can be severe. The more operations that can be safely repeated, the fewer exceptional states the architecture needs to handle.

Deduplication can operate at different levels:

- **Message-level deduplication:** Identifies messages that have already been processed;
- **Business-level deduplication:** Detects different requests that represent the same business intent;
- **Content-level deduplication:** Identifies equivalent documents using hashes or similar mechanisms;
- **Event-level deduplication:** Repeated events are discarded before producing new effects.

In RAG, deduplication also affects quality. Multiple copies of the same document consume storage, embeddings, and processing resources and may dominate the top retrieval results, reducing the diversity of evidence presented to the model. In this case, an operational technique improves both system efficiency and quality.

7.2.2. Retries, Backoff, and Jitter

Retries are appropriate when a failure may be transient. However, adding them indiscriminately can transform a small incident into a systemic failure.

A mature policy must account for several principles:

- **Not every error should be retried:** Authentication failures, validation errors, or invalid arguments will typically continue failing;
- **The number of attempts must be limited:** Indefinite repetition consumes resources without guaranteeing recovery;
- **The operation must tolerate duplication:** Retries over non-idempotent actions may generate incorrect effects;
- **The retry budget must fit within the latency budget:** A successful response may have no value if it arrives too late;
- **Transient and permanent errors must be distinguished:** Each category requires a different response.

AI platforms may have retries across model providers, vector databases, storage, external APIs, and other components. A particularly dangerous anti-pattern is allowing independent policies across several layers.

If the SDK, application service, proxy, and gateway each perform three attempts, a single original request may generate a much larger number of downstream calls. Retries must be treated as an end-to-end policy.

Exponential backoff gradually reduces the frequency of new attempts, giving a degraded dependency time to recover capacity:

- **Pressure decreases during incidents:** Clients stop continuously hammering the dependency;
- **Retry storms become less likely:** The downstream system gains room to recover;
- **Transient failures can be absorbed:** Small periods of unavailability no longer immediately become errors.

Backoff, however, consumes time. A policy that is acceptable for asynchronous workloads may completely violate the latency budget of an interactive application.

Jitter complements backoff by adding randomness to intervals. Without jitter, thousands of clients that failed at the same instant may retry at exactly the same time. The dependency receives new synchronized waves of traffic and fails again.

This behavior demonstrates that distributed incidents do not necessarily occur because individual clients are incorrect. Thousands of clients correctly executing the same strategy can collectively produce destructive behavior.

7.2.3. Circuit Breakers, Bulkheads, and DLQs

Circuit breakers temporarily stop calls to a dependency exhibiting persistent failures. Instead of consuming connections, threads, tokens, and latency budget on calls with a low probability of success, the system begins failing fast or uses an alternative path.

Conceptually, the mechanism has three states:

- **Normal operation allows calls:** The dependency is considered healthy;
- **Open circuit stops calls:** The recent failure rate indicates that continuing to try is counterproductive;
- **Controlled recovery probes the downstream system:** Some requests verify whether the dependency has recovered.

The benefits extend beyond the component being directly protected:

- **Fault containment reduces propagation:** A failing downstream dependency does not consume unlimited caller resources;
- **Fast failure preserves latency budget:** The application can make another decision quickly;
- **Fallback can be activated:** An alternative can assume part of the processing;
- **Downstream protection reduces pressure:** The dependency gains an opportunity to recover.

In AI systems, an unavailable model provider may lead the system to use another model, return reduced functionality, transform the task into asynchronous processing, or declare partial unavailability. Bulkheads complement this strategy by isolating resources across workloads.

This isolation can use:

- **Separate connection pools:** One workload does not exhaust all connections;
- **Independent thread pools:** Slow tasks do not block critical operations;
- **Per-tenant quotas:** One customer does not consume all shared capacity;
- **Dedicated workers:** Different workloads use separate resources;
- **Independent queues:** Low-priority backlogs do not block interactive paths.

Consider an AI platform that shares the same workers between interactive chat and large batch workloads. An intense batch workload may occupy all capacity and drastically degrade the online experience.

Bulkheads connect reliability directly to business priority. Dead-letter queues solve another problem: messages that continue failing even after the expected attempts.

A DLQ can provide:

- **Isolation:** Problematic items stop blocking the main flow;
- **Investigation:** Evidence remains available for analysis;
- **Replay:** Messages can be reprocessed after a fix;
- **Auditability:** The organization knows which operations failed.

In AI pipelines, messages may fail because of corrupted documents, unexpected formats, unavailable model providers, incorrect parsing, or validation failures. However, having a DLQ is not enough. Without ownership, alerts, and a remediation process, it becomes merely a permanent location for storing forgotten failures. Resilience must be an operational property, not merely a topological one.

7.3. Scalability, Partitioning, and Replication

Scalability is the ability to increase users, data, or processing without disproportionate degradation in performance, reliability, or cost. A scalable system is not one that simply uses many machines. It is one in which additional capacity can be incorporated predictably and where bottlenecks can be identified and addressed.

7.3.1. Horizontal and Vertical Scaling

Horizontal scaling adds instances to distribute work. This strategy works particularly well with stateless application servers or when the required state is externalized.

It provides several important advantages:

- **Elasticity follows demand:** Instances can be added or removed;
- **Fault tolerance increases:** The loss of one instance can be absorbed by the others;
- **Growth can be incremental:** Capacity is added progressively;
- **Load balancing distributes traffic:** Requests are routed across multiple instances.

The problem is that adding application servers does not automatically eliminate other bottlenecks. If twenty servers depend on the same already-saturated database, adding a twenty-first instance increases pressure on the limiting component.

The Staff-level question is: **which shared bottleneck limits end-to-end throughput?**

Scaling the wrong component adds cost without increasing capacity. Vertical scaling increases the resources of a single machine, such as CPU, RAM, VRAM, or faster storage.

In some scenarios, this is the most appropriate solution:

- **Distributed complexity is lower:** No additional nodes or partitions are introduced;
- **Coordination is reduced:** State remains more concentrated;
- **Workloads that are difficult to partition may benefit:** Large models are one example;
- **The operational change may be small:** The logical architecture remains similar.

However, vertical scaling has physical and economic limits:

- **Larger hardware tends to become progressively more expensive:** Cost does not necessarily grow linearly;
- **A machine cannot grow indefinitely:** Capacity limits exist;
- **Blast radius may increase:** A failure removes a larger share of capacity;
- **Upgrades may require interruption:** Depending on the infrastructure.

In model serving, vertical scaling may be unavoidable when no smaller machine has enough VRAM to load a replica. Horizontal and vertical scaling are therefore complementary strategies.

7.3.2. Partitioning and Sharding

Partitioning divides data or work into subsets so that different participants process different portions of the workload. The partition key is a structural decision because it affects balancing, locality, isolation, security, and operations.

Several alternatives illustrate these trade-offs:

- **User can provide natural locality:** Related data remains close together;
- **Tenant can facilitate isolation:** However, large customers may create skew;
- **Region can satisfy locality and data residency requirements:** But volume may be geographically uneven;
- **Hash tends to distribute more evenly:** However, it makes certain range queries more difficult;
- **Range simplifies certain queries:** But it may produce hotspots;
- **Document can be appropriate for content pipelines:** Provided that the distribution is sufficiently uniform.

A good partition key must reflect the actual access pattern, not merely the conceptual distribution of the data. A multi-tenant platform partitioned by tenant may achieve excellent isolation. However, if one customer represents half of the traffic, that customer may also represent half of the workload of a shard.

A decision that is positive for tenant isolation may be negative for load balancing. Sharding applies partitioning to storage by distributing subsets of data across nodes. It is common in distributed databases, search engines, and vector databases.

Its benefits are clear:

- **The dataset can exceed the resources of a single machine:** Total storage scales horizontally;
- **Throughput can increase:** Independent queries execute in parallel;
- **New shards enable progressive expansion:** Capacity follows growth.

However, distribution introduces new costs:

- **Cross-shard queries are more expensive:** The application queries multiple participants;
- **Rebalancing moves data:** Changing the number of shards can be operationally complex;
- **Hot shards limit the platform:** Uneven distribution reduces the benefits;
- **Global operations require aggregation:** Local results must be combined;
- **Fan-out increases:** One logical operation produces multiple physical calls.

In vector search, for example, a query may reach several shards. Each shard returns its best local candidates, and a subsequent stage must produce a global ranking. The number of shards must balance capacity, parallelism, and fan-out cost.

7.3.3. Replication, Quorums, and Leadership Models

Replication maintains multiple copies of the same state. It can serve different purposes:

- **High availability:** Another replica continues serving when a node fails;
- **Read scaling:** Reads can be distributed;
- **Disaster recovery:** Additional copies reduce the risk of permanent loss;
- **Geographic distribution:** Data can remain closer to users.

Every write, however, must be propagated. Synchronous and asynchronous replication produce different trade-offs:

- **Synchronous replication provides stronger guarantees:** The write waits for additional acknowledgments;
- **Synchronous replication increases latency:** More communication enters the critical path;
- **Asynchronous replication responds faster:** Not every copy needs to be updated immediately;
- **Asynchronous replication creates stale reads:** A window of divergence exists.

Each replica also adds storage, compute, and networking. Additional reliability is not free.

Quorums allow a sufficient subset of replicas to confirm a given operation, avoiding dependence on every node. The objective is to balance availability, latency, and consistency. More important than memorizing specific configurations is understanding that different guarantees depend on how many participants must agree before the operation can proceed.

In leader-based replication, a primary node coordinates or receives specific writes. This approach has advantages:

- **Ordering becomes simpler:** There is a primary authority for establishing sequence;
- **Consistency can be easier to reason about:** Concurrent conflicts are more controlled;
- **Reconciliation decreases:** Replicas follow a primary authority.

But new trade-offs emerge:

- **The leader can become a bottleneck:** Writes depend on its capacity;
- **Failover can cause unavailability:** Another participant must take over;
- **Election must be safe:** Participants cannot maintain conflicting interpretations of leadership;
- **Write-heavy workloads can saturate the primary path:** Read replicas do not solve write throughput.

In leaderless approaches, multiple replicas may accept operations. This can improve availability in some scenarios, but it transfers more complexity to reconciliation. The application may need to handle concurrent versions, merging, versioning, and quorums.

There is no universally superior choice. The decision depends on the meaning of the data and the properties that must be preserved.

7.4. Transactions, Distributed State, and Agentic Systems

As workflows cross multiple services, preserving the abstraction of a single transaction becomes increasingly difficult. AI systems and agentic systems intensify this problem because an operation may involve models, external tools, databases, queues, and actions that last much longer than a traditional transaction.

7.4.1. Distributed Transactions and Sagas

Distributed transactions attempt to preserve atomicity across multiple resources. Consider an operation that must update a database, publish an event, and reserve a resource. If only two of those steps are completed, the application may remain in an intermediate state.

There are several strategies for handling this:

- **Strong coordination can approximate a global transaction:** However, it increases dependency among participants;
- **Local operations can use compensation:** Intermediate states become explicitly accepted;
- **Outbox or transactional logs can synchronize state and publication:** Event delivery becomes part of the persistent flow;
- **Reconciliation can detect divergence later:** Not every inconsistency must be resolved synchronously.

In microservices, transforming every component in a workflow into one large global ACID transaction often creates more complexity than value. A mature architecture may accept intermediate states as long as they are explicit, observable, and recoverable.

Sagas follow this principle. The process consists of multiple local operations, and completed steps may have compensating actions.

Consider a process that reserves a product, charges the customer, and schedules delivery. If the charge fails after the reservation, another operation can release the item.

A saga does not attempt to pretend that the entire process happened instantaneously. It models the workflow as a sequence of recoverable states and actions. This is especially important for agents.

Critical tools must consider several properties:

- **Reversibility:** Can the operation be undone?;
- **Compensation:** Is there a semantically correct action that can reduce or repair the effect?;
- **Explicit status:** Does the workflow know which steps have been completed?;
- **Idempotency:** Can recovery safely repeat operations?;
- **Auditability:** Can decisions and effects be reconstructed?

An agent that can act autonomously but cannot recover from an intermediate failure has a fragile architecture.

7.4.2. Eventual Consistency in RAG

RAG is an excellent example of an eventually consistent distributed system. An update typically crosses several representations:

- **Storage preserves the original document:** The source begins to exist;
- **Parsing extracts structure and text:** Different formats must be processed;
- **Chunking creates retrieval units:** The document is divided;
- **Embedding generates vector representations:** New derived data appears;
- **Indexing publishes content to the search mechanism:** The document becomes retrievable;
- **Replication propagates the index:** Different replicas progressively receive the version.

These steps do not need to happen simultaneously. However, the application must maintain an explicit state model:

- **Stored does not necessarily mean searchable:** The product must distinguish the two conditions;
- **The user must know the processing state:** Received and indexed do not represent the same thing;
- **Intermediate failures must be visible:** An embedding error should not appear as a successful completion;
- **Reprocessing must be safe:** Retry and replay should be part of the design;
- **Old versions must be replaced correctly:** The index cannot retain inconsistencies indefinitely.

States such as **received**, **processing**, **indexed**, and **failed** make the pipeline easier to understand. This model simultaneously improves UX, observability, recovery, and governance.

7.4.3. Durable State in Agentic Systems

Long-running agents create another problem: execution may outlive the worker that initiated it. An agent may begin on one instance, perform several tool calls, wait hours for human approval, and later resume on another worker.

If all state exists only in the memory of the first instance, any failure destroys the progress. Durable agents must externalize the information required for resumption.

This state may include:

- **Current objective:** Preserves the purpose of the execution;
- **Position in the workflow or plan:** Makes it possible to determine what should happen next;
- **Relevant history:** Preserves the necessary context;
- **Tool outputs:** Prevents unnecessary repetition of operations;
- **Approvals:** Preserves human decisions and controls;
- **Checkpoint:** Defines a safe resumption point;
- **Identifiers of executed actions:** Support idempotency and deduplication.

Checkpointing also involves a trade-off. Persisting after every step increases durability but adds storage, latency, and complexity. Persisting infrequently reduces overhead but increases the amount of work lost after a failure. The appropriate frequency depends primarily on the cost of recomputation and the risk of repeating external effects.

7.4.4. Distributed Locks

Distributed locks are necessary when only one participant should own a specific resource or execute an operation. They can be useful for singleton jobs, exclusive updates, and some leader-election scenarios.

However, they introduce challenges of their own:

- **The lock holder may fail:** Another participant must assume ownership;
- **The lease may expire before the work is complete:** Two participants may execute simultaneously;
- **Network partitions create different perceptions:** Multiple nodes may believe they hold authority;
- **Clock issues affect leases:** Distributed time introduces ambiguity;
- **Contention reduces throughput:** Work that could be parallel becomes serialized.

For critical operations, fencing tokens can prevent a former holder from continuing to produce effects after losing ownership. The best question is not "which distributed lock should we use?" but: **is it actually necessary to coordinate this operation?** Partitioning, explicit ownership, and idempotency often eliminate the need for locks entirely.

7.5. Operability, Security, and Economics of Distributed Systems

Distributing computation and state creates scalability and availability benefits, but it also expands the operational surface. Caches, observability, security, data residency, and costs cannot be added later as independent concerns. They are part of the distributed architecture from the beginning.

7.5.1. Caching

Distributed caches reduce latency and load on source systems. In return, they introduce multiple representations of the same state.

An architecture must explicitly answer several questions:

- **Who invalidates the cache:** There must be a clear condition for considering an entry stale;
- **How long stale data is acceptable:** TTL must reflect the semantics of the data;
- **What happens if the cache is unavailable:** Application behavior must remain predictable;

- **Which component is the source of truth:** The cache must not accidentally acquire authority;
- **How hot keys are handled:** A small number of entries may concentrate a large share of traffic.

A cache should normally be reconstructable from the authoritative source. If losing it eliminates data that exists nowhere else, it is no longer functioning merely as a cache and has become a persistent state. A cache stampede is a classic example of distributed risk. When a popular entry expires, thousands of requests may attempt to recompute it simultaneously.

Several strategies reduce the problem:

- **Single-flight or locking limits concurrent recomputation:** One request refreshes the value while others wait;
- **Staggered expiration reduces synchronization:** Related items do not expire simultaneously;
- **Early refresh updates before expiration:** The current version continues serving traffic;
- **Stale-while-revalidate preserves availability:** Old data can be used while the new version is computed.

Again, the problem is not a single incorrect request, but emergent synchronization among thousands of correct requests.

7.5.2. Distributed Observability

Distributed troubleshooting requires reconstructing behavior across multiple services. Isolated local logs do not adequately represent an operation that crosses an API, retrieval, database, model inference, and external tools.

An observable architecture must combine different signals:

- **Correlation IDs connect events:** Components propagate the logical identity of the operation;
- **Distributed tracing represents the call chain:** Duration and dependencies can be understood;
- **Structured logs preserve searchable context:** Tenant, workload, model, request ID, and status can be correlated;
- **Per-dependency metrics reveal localized degradation:** Latency, errors, and throughput must be observed individually;
- **End-to-end metrics represent the final outcome:** Individually healthy components do not guarantee task success.

A request may spend only a few milliseconds in the API and several seconds in retrieval, the database, model inference, or tool execution. Without tracing, each team sees only its own component, and incidents become manual investigations across several teams. Observability is not an optional capability added after the architecture. It is a requirement for operating a distributed system.

7.5.3. Cybersecurity, Governance, and Data Residency

Distribution increases the attack surface because every service, communication channel, replica, and trust boundary creates another opportunity for abuse. Several risks are particularly relevant:

- **Service impersonation can falsify identity:** Service-to-service communication requires authentication;
- **Lateral movement increases the impact of compromise:** Compromising one service should not automatically provide access to the entire platform;
- **Replication increases points of exposure:** Every copy must be protected;
- **Message tampering threatens integrity:** Critical events require appropriate protection mechanisms;

- **Replay attacks reuse valid messages:** Idempotency does not necessarily replace anti-replay mechanisms;
- **Cross-tenant leakage can occur in caches or shards:** Isolation must be preserved throughout the architecture;
- **Compromised workers can generate malicious results:** Internal nodes should not receive unlimited trust merely because they are inside the network.

Controls may include service identity, TLS, authorization, signed tokens, tenant isolation, audit logs, idempotency, and replay protection. The more distributed the platform becomes, the more important it is to know the location and ownership of data.

The organization needs to know:

- **Where each piece of data is stored:** Including replicas and backups;
- **How many copies exist:** Replication changes the exposure surface;
- **What the source of truth is:** Without clear authority, recovery and incident response become ambiguous;
- **Who has operational ownership:** Every component requires defined responsibility;
- **How data crosses trust boundaries:** Security review depends on this lineage.

Data residency adds another dimension. A global architecture cannot determine its topology based solely on performance and availability:

- **Replication may move data across countries or regions:** Localization restrictions must be considered;
- **Backups are also part of the footprint:** Secondary copies are not outside governance;
- **Logs may contain PII:** Observability may also be subject to regulatory requirements;
- **Model providers may receive sensitive data:** The end-to-end flow must be evaluated;
- **Failover may change the processing location:** The secondary region may be subject to different requirements.

A decision that is excellent from an availability perspective may be inappropriate from a governance perspective.

7.5.4. The Economics of Reliability

Scalability and reliability have a cost. Every additional redundancy, recovery, and observability mechanism increases resource consumption and operational complexity.

Costs can appear in different forms:

- **Replication increases storage:** Every copy must be maintained;
- **Multi-region increases networking:** Data travels between regions;
- **High availability requires redundant capacity:** Some resources may remain idle;
- **Queues add infrastructure:** Messages must be stored and operated;
- **Retries increase workload:** A single request can generate several inference operations or calls;
- **Observability has its own cost:** Logs, metrics, and traces must be stored;
- **Strong coordination consumes latency and resources:** Additional guarantees are not free.

The architectural question should not be: **how do we achieve maximum availability?**

The more useful question is: **what level of availability does the business actually need, and how much are we willing to pay for it?**

A payment platform and a low-criticality internal workflow do not need the same topology. Reliability is an economic decision.

7.6. Resilient Architectures at Scale

After establishing state, processing, and security guarantees, the remaining question is how the system behaves under degradation, geographic distribution, and highly asymmetric workloads.

7.6.1. Graceful Degradation

Mature systems do not need to have only two states: fully functional or unavailable. Graceful degradation makes it possible to preserve part of the value even when certain capabilities fail.

In AI systems, there are several examples:

- **An unavailable primary model can activate a fallback:** Quality decreases, but the service continues;
- **An unavailable reranker may result in retrieval without reranking:** The pipeline operates in a reduced mode;
- **An unavailable recommendation service may return popular items:** Personalization is lost, but the experience remains useful;
- **An unavailable agent may be replaced by a manual workflow:** Users can still complete the activity;
- **Unavailable personalization may result in a generic experience:** The failure does not take down the entire product;
- **Optional dependencies can be skipped:** The critical path continues functioning.

For every dependency, one question should be asked: **does this failure really need to take down the entire experience?**

Often, it does not. Graceful degradation can provide greater perceived availability at a much lower cost than attempting to make every dependency individually infallible.

7.6.2. Multi-Region Architecture

Distributing a platform across regions can improve availability, geographic latency, and disaster recovery. However, every additional region expands the complexity of consistency, traffic routing, failover, observability, data residency, and cost. Active-passive keeps one region as the primary while another remains prepared to take over.

This strategy has specific characteristics:

- **The topology is relatively simple:** Most writes have a primary region;
- **Concurrent conflicts decrease:** Multiple regions do not necessarily modify the same state simultaneously;
- **Failover must be reliable:** The secondary region must have sufficiently recent state;
- **Capacity may remain idle:** Redundancy creates cost even when it is not serving users.

Active-active allows several regions to process traffic simultaneously. It presents a different set of trade-offs:

- **Users can use nearby regions:** Geographic latency decreases;
- **Redundant capacity is actively used:** Less infrastructure remains completely idle;
- **Concurrent writes may occur:** The domain must tolerate or reconcile conflicts;

- **Replication becomes critical:** State must continuously circulate;
- **Failover becomes less binary:** Traffic can be redirected among already-active regions.

The decision must consider Recovery Time Objective, Recovery Point Objective, required consistency, data residency, and the financial impact of downtime.

Multi-region is a tool, not an objective. For many workloads, its complexity may exceed the value it produces.

7.6.3. Hotspots, Skew, and Real Capacity

Uniform distribution is a dangerous assumption. Users, documents, tenants, and keys rarely generate the same traffic volume. Skew occurs when the actual workload distribution differs significantly from the average.

Several examples are recurring:

- **A large customer may concentrate a significant share of traffic:** Partitioning by tenant creates a hot shard;
- **A popular document may concentrate queries:** A small number of items dominate reads;
- **An inappropriate partition key may concentrate writes:** Data distribution appears healthy, but workload distribution does not;
- **Regional spikes may saturate part of the infrastructure:** The global average still appears normal.

Mitigations may include repartitioning, selective replication, caching, or hot-key splitting. The principle is: **capacity planning must analyze the actual workload distribution, not merely its average.**

Average load often hides exactly the cases that determine incidents.

7.6.4. Fan-Out, Fan-In, and Tail Latency

Fan-out occurs when a request is distributed across multiple components. Fan-in occurs when those results are aggregated again. The pattern frequently appears in distributed search. A query may reach several shards, each returns its best candidates, and another layer combines the results.

This parallelism provides advantages:

- **Recall can increase:** More portions of the corpus participate in the search;
- **Work can happen simultaneously:** Different shards process the query in parallel;
- **Total capacity increases:** Each node operates on only part of the data.

However, fan-out also has a cost:

- **Network calls increase:** One logical request generates several physical operations;
- **Tail latency becomes more relevant:** Slow participants can delay aggregation;
- **Failure probability increases:** More components enter the critical path;
- **Cost per request increases:** More infrastructure participates;
- **Observability becomes more complex:** The trace develops multiple branches.

In billion-scale retrieval, sending every query to every shard may become economically prohibitive. The architecture must limit fan-out to what is necessary to satisfy recall and quality requirements. This behavior also explains why average latency is insufficient in distributed systems.

When an operation depends on multiple calls, the slowest participant may determine the total duration. The greater the fan-out, the greater the probability of encountering a component in the slow tail. This is why p95 and p99 are fundamental.

Several strategies can reduce tail latency:

- **Timeouts limit waiting:** Slow components stop blocking indefinitely;
- **Hedged requests may help in specific cases:** A second request is initiated when the first exceeds a defined threshold;
- **Replicas provide alternative paths:** Another instance may serve the read;
- **Caching eliminates expensive calls:** Fewer dependencies enter the critical path;
- **Reducing fan-out decreases exposure to the slow tail:** Fewer participants need to respond.

At Staff level, performance must be analyzed as a distribution, not merely as an average.

7.7. Trade-Offs, Anti-Patterns, and Business Impact

Distributed Systems enable scale, isolation, availability, and independence across components. These capabilities, however, are obtained by introducing new forms of complexity. The role of a Staff AI Engineer is not to maximize any single property in isolation. It is to determine which guarantees are actually required and which costs can be accepted.

7.7.1. Key Trade-Offs

The main architectural trade-offs can be organized as follows:

- **Consistency vs. Availability:** During partitions, preserving consistency may require rejecting certain operations;
- **Consistency vs. Latency:** Strong guarantees generally require greater coordination;
- **Replication vs. Cost:** More copies increase availability but also increase storage, compute, and networking;
- **Horizontal Scale vs. Distributed Complexity:** More machines increase capacity but also network calls, coordination, and failure modes;
- **At-Least-Once vs. Duplicates:** Reducing loss requires consumers prepared for repeated processing;
- **Strong Coordination vs. Throughput:** Locks and consensus can limit parallelism;
- **Synchronous Processing vs. Resilience:** Synchrony simplifies the flow, while asynchronous processing improves decoupling;
- **Freshness vs. Availability:** Requiring information to always be current may reduce fault tolerance;
- **Sharding vs. Query Simplicity:** Distribution increases capacity but makes global operations more complex;
- **Multi-Region Reliability vs. Operational Complexity:** Geographic redundancy increases availability and also expands the operational surface.

The mature response is not to always choose the strongest guarantee. It is to identify which guarantee each workload requires and avoid paying for consistency, redundancy, or coordination without proportional value.

7.7.2. Important Anti-Patterns

Several mistakes repeatedly appear in distributed architectures and should be recognized quickly:

- **Assuming the network is reliable:** Every remote call can fail, be delayed, or lose its acknowledgment;
- **Implementing retries without idempotency:** Critical operations may produce duplicate effects;
- **Adding retries independently across multiple layers:** SDKs, proxies, gateways, and applications can multiply traffic;
- **Using global locks indiscriminately:** Excessive coordination reduces throughput and creates large failure domains;
- **Creating distributed transactions unnecessarily:** Compensation and eventual consistency may be sufficient;
- **Sharding prematurely:** Structural complexity is introduced before volume justifies it;
- **Ignoring hot keys and skew:** Healthy averages hide saturated partitions;
- **Adopting eventual consistency without considering UX:** Technically correct behavior may appear defective to the user;
- **Assuming exactly-once is an automatic infrastructure property:** Exactly-once effects typically depend on idempotency, transactions, and deduplication;
- **Replicating data without considering governance:** Availability increases together with data exposure;
- **Creating a DLQ without an operational process:** Messages stop blocking the pipeline but remain forgotten;
- **Operating without distributed tracing:** Diagnosis across multiple services becomes slow;
- **Maximizing reliability without analyzing cost:** An architecture can be robust yet economically unviable.

The common pattern behind these mistakes is optimizing a local property without considering its systemic effect.

7.7.3. Business Impact

Distributed Systems directly affect availability, growth capacity, experience, risk, and cost. When well designed, the architecture can produce important benefits:

- **Replication reduces downtime:** Individual failures no longer eliminate the service entirely;
- **Horizontal scaling follows growth:** Capacity increases as traffic and volume grow;
- **Async processing absorbs spikes:** Input rate no longer needs to instantaneously match processing capacity;
- **Idempotency reduces operational errors:** Retries become safer;
- **Graceful degradation preserves value during incidents:** Part of the experience remains available;
- **Sharding enables large-scale workloads:** Data no longer depends on a single machine;
- **Multi-region can increase global availability:** It may also reduce geographic latency;
- **Observability reduces MTTR:** Incidents can be located and understood more quickly.

However, high availability only produces ROI when the financial and operational impact of downtime justifies its cost. For internal or low-criticality workloads, a simpler architecture may be economically superior. Distributed Systems must therefore be treated simultaneously as technical and economic engineering.

Chapter Summary

Distributed Systems is the discipline of building reliable systems despite independent components, imperfect networks, replicated state, and inevitable failures. For AI Engineering, this is particularly important because modern applications combine databases, vector stores, model providers, retrieval, caches, agents, queues, and external tools into a single logical flow.

The central principles can be summarized as follows:

- **Assume partial failures from the design stage:** Services can fail independently, and responses may disappear after an operation has already executed;
- **Define consistency according to data semantics:** Not all information requires strong consistency;
- **Use eventual consistency consciously:** The divergence window must be compatible with the product and the business;
- **Design idempotent operations:** This makes retries, recovery, and replay significantly safer;
- **Accept duplicates when they are easier to control than loss:** At-least-once often provides a practical trade-off;
- **Use retries, backoff, and jitter with limits:** Recovery mechanisms should not generate more load than the original failure;
- **Isolate failures:** Circuit breakers, bulkheads, and DLQs reduce cascading failures;
- **Scale the actual bottleneck:** Horizontal scaling of non-limiting components only increases cost;
- **Shard only when volume justifies it:** Premature distribution creates complexity that is difficult to remove;
- **Use replication proportionally to criticality:** Additional availability has financial and consistency costs;
- **Observe tail latency:** The slow tail frequently determines the actual experience;
- **Treat distributed state explicitly:** This is critical for long-running agents, workflows, and pipelines;
- **Reduce coordination whenever possible:** Ownership and partitioning often scale better than global locks;
- **Consider security and governance in the topology:** Where state and data exist matters as much as how they are processed;
- **Connect reliability to economic value:** The architecture should provide the guarantees that are necessary, not the theoretical maximum.

The central lesson is that scaling a system does not simply mean adding machines. It means managing coordination, consistency, failures, state, and costs across components that never possess a perfectly synchronized view of the world.

Practical Framework for Analyzing a Distributed System

When given a System Design problem, a Staff AI Engineer can use a relatively systematic sequence to discover which guarantees are necessary before selecting technologies. First, determine where state and remote calls exist:

- **Identify stateful components:** Distinguish persistent state, cache, transient state, and derived representations;
- **Determine the source of truth:** Every piece of information must have a clear authority;
- **Map network boundaries:** Every remote call adds latency and the possibility of partial failure;
- **Identify failure domains:** Determine which parts can fail independently;
- **Classify critical operations:** Different paths do not necessarily require the same guarantees.

Next, define consistency requirements:

- **Determine whether read-after-write is required:** This guarantee may be sufficient without global strong consistency;
- **Determine whether stale data is acceptable:** The answer influences the possible level of decoupling;
- **Evaluate concurrent writes:** Can different regions or participants modify the same resource simultaneously?;
- **Determine whether conflicts can be reconciled:** Some domains allow merging, while others do not;
- **Identify effects that must never be duplicated:** These require special attention to idempotency, deduplication, and transactions.

Next, analyze reliable processing:

- **Define whether messages may be delivered more than once:** This determines consumer requirements;
- **Verify whether consumers are idempotent:** Replay must be safe;
- **Determine where deduplication occurs:** Message IDs and business IDs solve different problems;
- **Create a path for unrecoverable items:** DLQs or equivalent mechanisms must have ownership;
- **Verify whether replay is possible:** Recovery frequently depends on reprocessing.

Then, design the resilience policy:

- **Define timeouts explicitly:** No remote call should consume resources indefinitely;
- **Coordinate retries, backoff, and jitter:** The policy must consider the end-to-end path;
- **Evaluate circuit breakers:** Degraded dependencies should not produce cascading failures;
- **Use bulkheads when workloads compete for capacity:** Failure domains must be controlled;
- **Design graceful degradation:** Not every downstream failure needs to take down the entire experience.

Next, analyze scale:

- **Find the component that saturates first:** This is the bottleneck that actually needs to be scaled;
- **Evaluate whether vertical scaling is sufficient:** Distribution should not be added unnecessarily;
- **Determine whether partitioning is actually required:** Volume and throughput must justify the complexity;
- **Choose partition keys based on the workload:** Data distribution does not guarantee traffic distribution;
- **Determine the need for replication:** Additional reliability requires justification;
- **Analyze fan-out:** A query distributed across many participants may turn tail latency into the primary bottleneck.

Next, examine the distributed state:

- **Determine whether workers need to be stateless:** Local state makes horizontal scaling and failover more difficult;
- **Define where checkpoints are persisted:** Long-running workflows must survive worker loss;
- **Ensure agent resumption:** Critical information should not exist only in memory;
- **Question every distributed lock:** Coordination should exist only when the domain truly requires it;
- **Look for opportunities to eliminate coordination:** Partitioning, ownership, and idempotency are often better alternatives.

Finally, connect the architecture to business, security, and governance:

- **Determine how much downtime is acceptable:** Availability targets must reflect actual requirements;
- **Quantify the cost of inconsistency:** A stale read may be irrelevant in one context and catastrophic in another;
- **Define how much redundancy is justified:** Reliability is also an economic decision;
- **Map every copy of the data:** Replicas, caches, and backups are part of the governance surface;
- **Verify data residency:** Distributed topology must comply with applicable requirements;
- **Evaluate tenant isolation:** The workload or incident of one customer should not compromise others;
- **Calculate the cost of the proposed guarantees:** Availability, consistency, and redundancy must remain sustainable at scale.

This is the mental model expected from a Staff AI Engineer: **explicitly define which guarantees the system actually requires and build the minimum amount of coordination, redundancy, and complexity necessary to deliver them reliably, securely, and economically sustainably.**