

Temporal Workflows

Durable Orchestration for Microservices and Agents

Trex Team

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2026 by N O B T R E X LLC. All rights reserved.

For permissions and other inquiries, write to:
P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

Contents

1	Platform Mental Model and Core Components	5
1.1	Choosing Durable Execution over Queues, Cron, and Workflow State Machines	5
1.2	Modeling the Temporal Server, Cluster Roles, and Namespace Isolation	12
1.3	Scaling Workers with Task Queues: Polling, Throughput Shaping, and Routing	19
1.3.1	The Mechanics of Polling and Backpressure . .	20
1.3.2	Differentiating Workflow Tasks and Activity Tasks	21
1.3.3	Task-Queue Topology and Workload Isolation .	22
1.3.4	The Scaling Playbook: Observing and Adjusting Capacity	24
1.3.5	Common Anti-Patterns in Worker Scaling . . .	25
1.4	Understanding Workflow Executions, Runs, and Event History as the Source of Truth	26

2	Workflow Execution Semantics and Determinism	33
2.1	Replay mechanics: reconstructing workflow state from event history	34
2.2	Determinism rules: what must be stable under replay and how it breaks	41
2.3	Designing replay-safe workflow logic: explicit state, stable branching, and deterministic concurrency	49
2.4	Controlling history growth with Continue-As-New: rollover patterns and state handoff	56
3	Activities: Side Effects, Timeouts, Retries, and Idempotency	63
3.1	Placing Side Effects Correctly: The Workflow/Activity Determinism Boundary	64
3.2	Engineering Activity Timeouts and Retries as a Reliability Contract	72
3.2.1	The Timeout Taxonomy and Failure Detection	74
3.2.2	Computing Maximum Wall-Clock Exposure	76
3.2.3	Designing for Interruption and Probabilistic Availability	77
3.2.4	Operational Runbook: Diagnosing Timeout Spikes	79
3.3	Implementing Idempotency for Activities Across Databases, RPCs, and Third-Party APIs	82
3.4	Orchestrating Workflow-Level Backoff, Hedged Requests, and Fallback Providers	88

3.5	Choosing Between Retry, Compensate, and Human Escalation: A Failure Classification Playbook	96
3.5.1	Encoding Reconciliation and Compensation . .	100
3.5.2	Structuring Human Escalation	101
4	Orchestration Patterns for Microservices	105
4.1	Implementing Durable Timers, Deadlines, and Wait Conditions Without Resource Locking	105
4.2	Building Fan-out/Fan-in Aggregations with Bounded Concurrency and Backpressure	114
4.3	Designing Child Workflows for Decomposition, Team Ownership, and Fault Isolation	120
4.4	Encoding Sagas and Compensation Plans as Auditable Workflow State	128
4.5	Establishing Task Queue and Namespace Conventions for Routing, Isolation, and Multi-Team Operations . . .	136
5	Interacting with Running Workflows: Signals, Queries, and Updates	145
5.1	Choosing Between Signals, Queries, and Updates: Semantics, Consistency, and Failure Surfaces	146
5.2	Designing Workflow Interaction APIs: Stable Contracts and Evolvable State Machines	154
5.3	Implementing Update-With-Start: Atomic <i>Create If Absent</i> + <i>Mutate</i> Interaction	161

5.4	Building Callback-Driven Ingestion: Webhooks, Message Buses, Deduplication, and Ordering	169
5.5	Human-in-the-Loop Interactions	179
6	Operating Temporal in Production: Visibility, Schedules, and Tooling	187
6.1	Visibility versus Event History: Building Fast Listing and Operational Queries	188
6.2	Designing Search Attributes for Multi-Dimensional Observability and Routing	195
6.3	Implementing Metrics, Logs, and Traces for Workflows and Worker Fleets	202
6.4	Operating Schedules: Server-Managed Cron, Safe Catch-Up, and Controlled Backfills	209
6.5	Using the Temporal CLI for Local Iteration, Introspection, and Admin Runbooks	215
7	Safe Evolution: Testing, Replay, and Worker Versioning	225
7.1	Implementing replay-first test suites to prevent non-deterministic upgrades	225
7.1.1	Mechanics of Replay Verification	226
7.1.2	Curating the History Corpus	228
7.1.3	Designing a Resilient Test Harness	230
7.1.4	Taxonomy of Replay Failures	231
7.1.5	Repair Strategies and Actionable Fixes	233

7.2	Running incident investigations using workflow history as primary evidence	234
7.3	Choosing upgrade strategies: backward compatibility, pinning, and multi-fleet rollouts	242
7.3.1	Backward-compatible workflow code paths . . .	242
7.3.2	Pinned workers and queue-based isolation . . .	244
7.3.3	Multi-fleet rollouts and progressive load shifting	246
7.3.4	Defining the operational contract	247
7.4	Applying Worker Versioning: build IDs, ramp policies, and rollback-safe execution routing	249
7.5	Evolving workflow contracts and metadata: schemas, search attributes, and update/signal APIs	255
8	Durable Agents with Temporal: Reliable LLM and Tool Orchestration	263
8.1	Mapping Agent Control Loops to Workflows and Tool Calls to Activities	264
8.2	Engineering Semantic Failure Handling: Structured Retries, Fallback Tools, and Compensations	270
8.2.1	The Semantic Failure Taxonomy	271
8.2.2	Orchestrating the Repair Loop	273
8.2.3	Durable Compensations for Partial Effects . . .	274
8.2.4	Observability and Circuit Breakers	276
8.3	Implementing Human Feedback and Policy Gates with Signals and Updates	277

8.4	Building Long-Running Autonomy: Timers, Schedules, Deadlines, and Catch-Up Semantics	284
8.5	Choosing Cross-Boundary Durable Operations: Nexus vs Direct Calls vs Activity-Based Integration	292
8.5.1	Evaluating the Three Integration Models	294
8.5.2	Operational Ownership and Observability	297
8.5.3	Security and Authorization Boundaries	298
8.5.4	Failure Semantics and Graceful Degradation	299

Introduction

Distributed systems inevitably encounter partial failures, network latency, and process crashes. Coordinating state across microservices typically relies on a combination of message queues, database transactions, ad-hoc state machines, and scheduled jobs. These distinct tools address individual failure modes but force developers to write defensive, complex boilerplate to manage retries, timeouts, and state recovery for long-running operations.

Temporal provides a distinct compute model: durable execution. It abstracts the complexity of distributed state management by guaranteeing that code executes to completion regardless of transient failures or infrastructure outages. This is achieved through an append-only event history log and deterministic replay. When a worker process fails, a new worker process consumes the recorded events, replays the compiled code, and resumes the function exactly where it halted. The state of local variables, the program counter, and the call stack are implicitly preserved without requiring manual database checkpoints.

This book provides a comprehensive technical examination of the Temporal platform and its application to modern system architecture. It begins by establishing a precise mental model of Temporal's core components: the server, namespaces, task queues, and workers. The text explicitly defines the boundary between deterministic workflow exe-

cution and non-deterministic activities, which isolate external side effects, API requests, and database operations. With this foundation established, the book details explicit control-flow patterns for distributed microservices. It covers timeout taxonomies, idempotency strategies, concurrency controls, and the use of sagas for managing distributed transactions and structured compensations.

Because durable executions can run for months or years, they require continuous external interaction and high operational visibility. The text explains how to safely read and mutate the state of active workflows using queries, signals, and updates. It examines production operations, detailing how search attributes and visibility indexing enable efficient monitoring, listing, and system introspection. Furthermore, deploying changes to long-running code introduces unique versioning requirements. The book outlines safe evolution techniques, including replay-driven testing, incident debugging from event histories, and worker versioning mechanisms, ensuring that new code deployments do not violate the determinism constraints of in-flight executions.

Finally, the text applies the durable execution model to the design of autonomous agents. Coordinating Large Language Models and external tool integrations introduces significant non-determinism and complex semantic failure modes. By mapping agent control loops to Temporal workflows and mapping tool executions to Temporal activities, developers can enforce strict timeout contracts, implement structured fallbacks, and facilitate explicit human-in-the-loop approvals. The inclusion of the Nexus framework further extends this capability, allowing for durable operations across distinct organizational or infrastructure boundaries.

The objective of this text is to equip software engineers and system architects with the practical patterns and operational knowledge required to utilize Temporal effectively. By adhering to the strict constraints of deterministic replay and leveraging Temporal's native prim-

itives, teams can build reliable, scalable, and highly auditable distributed applications.

Chapter 1

Platform Mental Model and Core Components

Temporal is easiest to misuse when you treat it as “just a queue” or “just a scheduler.” This chapter makes a precise, operational mental model: the server persists an append-only history and drives tasks and timers; workers are stateless interpreters of that history; and replay is the mechanism that turns crashes, restarts, and long waits into normal control flow. With this model, later chapters can talk about determinism, retries, idempotency, and upgrades without hand-waving.

1.1. Choosing Durable Execution over Queues, Cron, and Workflow State Machines

Engineers building distributed systems inevitably need to execute work outside the context of a synchronous user request. The standard toolkit

for background processing usually begins with message queues for discrete jobs, cron for time-based scheduling, and explicit, database-backed state machines for multi-step processes. While these tools are foundational, they force you to implement orchestration logic, retry handling, and state recovery manually when business processes span multiple steps, require routing, or involve waiting for external events.

To understand when an event-sourced, replay-driven model is the right abstraction, you must first define the operational boundaries and failure modes of the traditional alternatives.

Consider a common operational requirement: scheduling a recurring process that must execute reliably, prevent overlapping runs, and allow for historical backfills. Using traditional cron, you configure a system-level daemon or a cloud scheduler. If the process crashes, the scheduler is unaware; if a run takes longer than the interval, a second instance starts concurrently, potentially corrupting data.

In a durable execution system, you model this not as a blind trigger, but as a managed orchestration. For example, using the Temporal CLI, you define a schedule that guarantees exactly one active workflow execution per interval:

```
temporal schedule create \  
  --schedule-id "monthly-billing-run" \  
  --interval "30d" \  
  --workflow-id "billing-generator" \  
  --task-queue "billing-tasks" \  
  --type "BillingWorkflow" \  
  --overlap-policy "Skip"
```

This command delegates the responsibility of trigger timing, overlap prevention (Skip), and durable tracking to the platform. The underlying execution maintains its state regardless of worker restarts. Recognizing when to leverage this abstraction requires evaluating the limits of standard background processing patterns.

Message Queues and Job Workers Message brokers (such as RabbitMQ, Kafka, or Amazon SQS) provide at-least-once delivery of discrete payloads. They excel at decoupling producers from consumers and buffering traffic spikes. When using a queue, the unit of work is typically short-lived and stateless.

The primary operational contract of a queue is delivery, not execution progress. If a consumer crashes halfway through processing a message, the visibility timeout expires, and the broker re-delivers the message to another worker. This creates the classic “invisible retry” failure mode: any external side effects (like charging a credit card or sending an email) performed before the crash will be executed again unless the consumer implements strict idempotency checks against an external database.

Furthermore, queues lack native primitives for multi-step routing. Orchestrating a process with five distinct steps requires maintaining five separate queues, passing the accumulated state in the message payloads, and building external tracking mechanisms to monitor the overall progress of the distributed transaction. When complex routing, fan-out, or conditional branching is required, relying solely on queues quickly degrades into an unmanageable mesh of dead-letter queues and undocumented implicit state.

Time-Based Schedulers (Cron) Cron and distributed task schedulers execute code at specific temporal intervals. Their primary guarantee is best-effort triggering. They are stateless by design; the scheduler does not track the internal progress of the script it invokes.

The most common failure mode for time-based schedulers is the overlap problem. If a nightly synchronization job takes twenty-six hours to complete, the next invocation will start before the first finishes, leading to race conditions unless you implement custom distributed locks. Additionally, standard cron lacks native backfill capabilities. If the sched-

uler is down for three hours, you must manually calculate the missed intervals and execute custom scripts to reconcile the missing data.

Explicit Workflow State Machines When multi-step durability is required, engineers frequently build explicit state machines. This typically involves a database table with a `status` column (e.g., `PENDING`, `PROCESSING`, `COMPLETED`), combined with worker processes that poll the database for actionable rows using `SELECT ... FOR UPDATE` semantics.

While this provides strong consistency and centralizes state, the operational burden is immense. You own the concurrency control, the polling fleet scaling, and the schema migrations. As the business logic evolves, adding a new intermediate state to an existing process requires complex SQL migrations to update inflight records. The control flow of the business process becomes fragmented, buried across SQL queries, polling loop logic, and disparate worker scripts.

Defining the Durable Execution Problem Statement Temporal's problem statement is precisely scoped: the durable orchestration of long-lived, multi-step processes where progress must be preserved across failures, where waiting is cheap, and where control flow can branch, retry, fan out, and block on external input without holding compute resources.

The platform achieves this not by storing the literal memory state of a process, but by persisting an authoritative event history. The Temporal server acts as an append-only ledger of decisions and outcomes. Workers—which host your application code—reconstitute the state of a workflow by deterministically replaying this history from the beginning.

When a workflow needs to wait for thirty days, it does not consume a thread or hold a database lock. The worker instructs the server to create a durable timer and drops the workflow instance from memory.

Thirty days later, the server enqueues a wake-up task. A worker picks up the task, fetches the history, replays the code to the exact line where it paused, and continues execution. System correctness hinges on a strict architectural boundary: orchestrating decisions (workflow code) must be fully deterministic, while all fallible, non-deterministic operations (calling APIs, querying databases) must be isolated in activities.

Decision Criteria and Trade-offs Selecting the appropriate tool requires evaluating your workload against the guarantees provided by each architecture. Use the following criteria to define your system boundaries.

Use a queue when the unit of work is short, stateless, and idempotency is trivial to implement. High-throughput data ingestion, log processing, or simple image resizing tasks are perfectly suited for message brokers. If the task fails, discarding it or retrying from scratch carries minimal penalty.

Use cron when the only durable requirement is “start something periodically,” and the execution is safe to skip or overlap. Cleaning up temporary directories, flushing local caches, or emitting heartbeats are tasks where the overhead of durable execution is unnecessary.

Use an explicit state machine when you require a shared-memory model across multiple disparate writers. If dozens of independent microservices need to concurrently mutate the exact same entity state via rapid, ad-hoc transactions, a traditional relational database with strict ACID locks remains the correct tool. Durable execution models optimize for a single authoritative orchestrator advancing state, not massive concurrent mutation of a single record.

Use Temporal when you need durable progress with rich control flow. If your process spans multiple microservices, requires compensating transactions (the saga pattern), waits for human approvals, or relies on complex exponential backoffs for failing APIs, durable execution is

the objectively better abstraction. You gain complete auditability of the execution history natively, provided you can design your workflow code to adhere to strict determinism constraints.

Capability	Message Queues	Cron	Explicit State Machine	Temporal
Durable progress across steps	Manual / External DB	None	Manual (DB Schema)	Native (Event History)
Waiting without resource lock	Requires delay queues	Native	Requires active polling	Native (Durable Timers)
Complex control flow (fan-out)	High complexity (DLO mesh)	N/A	High complexity (SQL)	Native (Code constructs)
Auditability	External logging required	Logs only	Current state only	Complete execution log
Programming model complexity	Low for single steps	Low	High (Custom orchestration)	Moderate (Requires determinism)
Primary scaling unit	Consumers / Partitions	Scheduler nodes	Database IOPS / Locks	Task Queues / Pollers

Architectural Anti-Patterns When migrating to a durable execution model, developers frequently carry over assumptions from queues and explicit state machines. Recognizing these anti-patterns prevents severe systemic issues.

Executing side effects in workflow code. This is the most critical failure mode in replay-driven systems. If you place a network call, a random number generator, or a database write directly inside the workflow definition, the workflow becomes non-deterministic. When the worker restarts and replays history to recover state, it will execute the network call again. If the result differs from the original execution, the worker detects a state mismatch and blocks the workflow. All side effects must be wrapped in activities, which record their results durably in the event history upon completion.

Using Temporal as a generic distributed lock or high-frequency cache. While Temporal provides strong consistency guarantees for a given workflow execution, it relies on database transactions under the hood for every state transition. Rapidly signaling a single workflow thousands of times per second to update a counter will throttle the names-

pace and cause history growth limits to be breached. Use Redis or Memcached for ephemeral, high-throughput locking and counting; use Temporal to orchestrate the broader lifecycle.

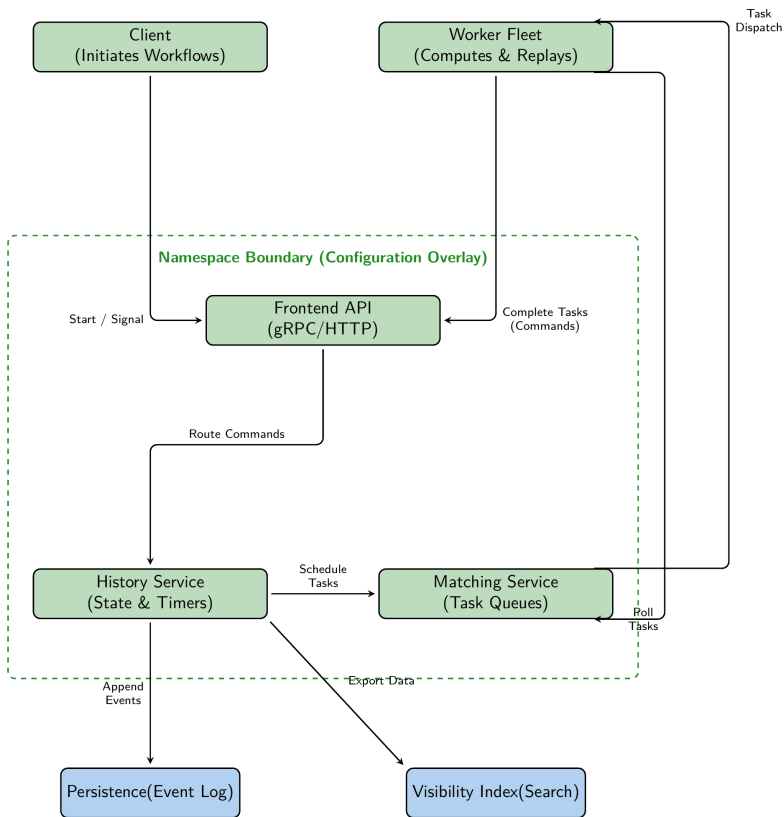
Treating activity retries as a substitute for idempotency. Temporal automatically retries failed activities based on your configured policies. However, if an activity sends an HTTP POST request to a payment gateway, the network connection might drop after the gateway has processed the charge but before the Temporal worker receives the response. The system will retry the activity. If the downstream payment gateway does not enforce idempotency (e.g., via a unique idempotency key provided by the activity), you will double-charge the customer. Durable execution guarantees that the activity will eventually succeed or exhaust retries; it does not absolve you from designing idempotent downstream APIs.

Modeling single, microscopic jobs as individual workflows. If you need to process one million standalone rows from a CSV file where each row takes 10 milliseconds to parse and insert into a database, spawning one million separate workflows introduces massive overhead. Each workflow generates multiple persistence round-trips to the Temporal server to record its start, task scheduling, and closure. For massive fan-out of independent, microscopic, non-durable tasks, a standard message queue or a batch processing framework is significantly more efficient. Reserve durable execution for the parent process that orchestrates the batch job, tracks its overall completion, and handles the subsequent business logic.

1.2. Modeling the Temporal Server, Cluster Roles, and Namespace Isolation

The fundamental server-side contract in Temporal relies on a strict division of responsibilities: the Temporal cluster durably records your workflow's history, and your worker processes evaluate that history to drive execution. A common early misconception is that the Temporal server executes workflow code. It does not. The server acts as an authoritative, highly available event ledger and task scheduler. Workers are stateless compute nodes that pull from this scheduler, interpret the event history through the lens of your application code, and return commands detailing what must happen next.

Understanding the mechanics of the server and how it isolates workloads is critical for designing scalable systems, setting appropriate data retention policies, and establishing multi-tenant architectures.



Cluster Roles and Responsibilities To interact correctly with the platform, you must distinguish between what the cluster persists, derives, schedules, and enforces. The Temporal server acts as a distributed orchestrator composed of several distinct logical services, typically backed by a persistent data store (like Cassandra, PostgreSQL, or MySQL) and a search backend (like Elasticsearch).

- *Persisted (The Authority)*: The History Service owns the workflow state. It durably records every action as an append-only se-

quence of events in the primary database. If a database transaction commits an event, it is globally guaranteed to have occurred. This event history is the absolute source of truth for the system.

- *Derived (The Index)*: Workflow metadata—such as execution status, start times, and custom search attributes—is asynchronously projected into a visibility store. This index powers operational queries (e.g., “list all failed workflows in the last hour”). Because visibility is derived, it is eventually consistent and decoupled from the strong consistency guarantees of the event history.
- *Scheduled (The Queues)*: The Matching Service brokers tasks. When the History Service determines that code must execute (a workflow needs to advance, or an activity needs to run), it instructs the Matching Service to enqueue a task. Matching maintains the task queues and pairs incoming tasks with polling workers. When a worker requests work, Matching dispatches the task.
- *Enforced (The Policies)*: The cluster enforces timeouts, rate limits, and isolation boundaries. If a worker pulls a task but crashes, the History Service’s timers dictate when that task times out and must be enqueued for another worker. The cluster aggressively protects its own availability by shedding load and throttling requests that exceed configured quotas.

Namespace Isolation and Configuration A Namespace is the fundamental unit of isolation, routing, and configuration within a Temporal cluster. You can conceptualize a namespace as a hybrid between a Kubernetes namespace, a database schema, and a Service Level Objective (SLO) domain. Namespaces apply an operational overlay on top of the underlying physical cluster, enforcing boundaries around visibility, retention, and access control.

To establish these boundaries, operators register namespaces with spe-

cific configuration constraints. The following CLI command demonstrates the creation of a production namespace with a strict retention policy and archival enabled:

```
temporal operator namespace create \  
  --retention 30d \  
  --history-archival-state enabled \  
  --history-uri "s3://my-company-bucket/temporal-archival/" \  
  --description "Production payment processing" \  
  payments-prod
```

This configuration defines the rules of engagement for any workflow executed within the `payments-prod` namespace.

The `retention` parameter is the most critical architectural decision at the namespace level. It dictates how long a workflow's event history is retained in the primary database after the workflow reaches a closed state (completed, failed, canceled, or timed out). Open workflows are never deleted by the retention policy. Retention directly impacts cluster storage costs, database performance, and operational debuggability. Depending on the server version and whether you are operating self-hosted or Temporal Cloud, the default retention limit generally defaults between 72 hours and 30 days.

When audit requirements demand that histories be kept longer than the live retention period, `history-archival-state` allows closed histories to be continuously exported to cold storage (such as Amazon S3 or Google Cloud Storage) before they are scrubbed from the high-performance primary database.

Namespaces also scope Custom Search Attributes. Because the visibility backend maps search attributes to strongly typed database indices, a schema change in one namespace does not corrupt the visibility index of another.

For multi-tenancy, you must choose between a *tenant-per-namespace* model and a *tenant-in-attributes* model.

- Use *tenant-per-namespace* when tenants require different retention policies, dedicated rate limiting, separate encryption keys, or distinct access-control lists (authorization). This provides the highest degree of isolation but carries the operational overhead of managing numerous namespaces via infrastructure-as-code.
- Use *tenant-in-attributes* when tenants share identical SLOs, compliance requirements, and worker fleets. In this model, all executions run in a shared namespace, and tenant identity is attached as a Custom Search Attribute. You filter by tenant exclusively during API list queries.

The Operational Flow on the Wire Understanding how state translates into network operations allows you to debug effectively when the system behaves unexpectedly. Consider the lifecycle of a workflow execution from the perspective of the server-side platform:

- *Initiation:* A client application invokes a gRPC `StartWorkflowExecution` request against the Frontend Service, specifying the target namespace, workflow type, and task queue.
- *History Append:* The Frontend routes the request to the History Service. History opens a new mutable state record in the persistence layer, appending a `WorkflowExecutionStarted` event. Immediately, it also appends a `WorkflowTaskScheduled` event.
- *Task Dispatch:* The History Service notifies the Matching Service that a workflow task is pending on the designated task queue.
- *Worker Polling:* A fleet of workers continuously long-polls the Matching Service. Matching pairs the pending task with an available worker and dispatches it over the network.

- *Compute and Respond:* The worker receives the task, which includes the event history. The worker's SDK interprets the history to reconstitute the local state of the workflow code, executes the necessary business logic until it reaches a blocking operation (like calling an activity or setting a timer), and generates a set of commands. The worker sends these commands back to the server via a `RespondWorkflowTaskCompleted` request.
- *State Transition:* The History Service receives the completion. It appends a `WorkflowTaskCompleted` event to the history ledger. It then processes the generated commands—for instance, writing an `ActivityTaskScheduled` event—and tells Matching to enqueue the new activity task.
- *Iteration:* The system repeats this cycle of queuing tasks, awaiting worker execution, and appending results to the ledger until the worker eventually issues a command to complete or fail the workflow.

At no point during this flow does the Temporal cluster evaluate your application code or maintain your workflow's variables in its own memory. The server's memory solely tracks task routing, database connections, and timers. This separation is why worker fleets can be aggressively scaled, restarted, or even crash repeatedly without corrupting the state of long-running business processes.

Isolation Failures and Anti-Patterns Treating namespaces as mere metadata tags—rather than robust configuration boundaries—leads to predictable operational failures.

A primary anti-pattern is sharing a single namespace across multiple development environments (e.g., placing staging and production workloads in a single default namespace). Because namespaces are

the boundary for quotas and limits, a load test executed by staging workers can exhaust the namespace-level API request limits, inadvertently throttling production clients attempting to start critical workflows. Similarly, task queues are scoped to namespaces. If staging and production share a namespace and a task queue name, a staging worker will eventually poll and execute a production task, polluting production data with staging configuration. Always map deployment environments to separate namespaces.

Another critical failure mode involves conflating the consistency guarantees of the event history with the visibility index. Because workflows operate within a namespace, engineers often write external applications that query the namespace for workflow status (e.g., querying for workflows with a specific search attribute) and immediately act upon the result. While the event history possesses read-after-write consistency, the visibility index does not. If a client starts a workflow and immediately queries the visibility API for that workflow's existence, the query may return empty. Platform observability and indexing rely on asynchronous replication. Application logic must be driven by synchronous signals, queries, and history events rather than by polling the visibility index for state transitions.

Namespaces dictate your operational blast radius. By structuring namespaces around specific lifecycle retention needs, explicit operational boundaries, and multi-tenant constraints, you isolate failures to the smallest possible domain. The cluster is responsible for durably holding the state within those domains; your task is to ensure the domains are drawn around the correct boundaries.

1.3. Scaling Workers with Task Queues: Polling, Throughput Shaping, and Routing

Workers in a Temporal architecture act as horizontally scalable, stateless compute units that execute your application code. Unlike traditional message brokers or webhook-driven systems that push work to consumers, Temporal operates on a strict pull-based model. Workers actively poll the Temporal server for tasks, execute the corresponding workflow or activity code, and report the results back. This architecture inherently provides backpressure: if your worker fleet is saturated, workers naturally stop polling, allowing tasks to queue safely within the Temporal server until compute capacity becomes available.

To govern this behavior, worker configurations expose explicit controls for local concurrency and polling behavior. The following Go SDK example demonstrates the primary configuration surfaces used to tune a worker's throughput profile:

```
workerOptions := worker.Options{
    // Local concurrency limits: how many tasks run simultaneously
    MaxConcurrentActivityExecutionSize: 1000,
    MaxConcurrentWorkflowTaskExecutionSize: 100,

    // Poller limits: how many concurrent long-polls are open to the
    // server
    MaxConcurrentActivityTaskPollers: 20,
    MaxConcurrentWorkflowTaskPollers: 5,
}

w := worker.New(client, "ecommerce-payments-queue", workerOptions)
```

These four parameters act as the first-order controls over your system's throughput and schedule-to-start latency. Tuning them correctly requires understanding the mechanical separation between polling and executing, as well as the fundamental differences between workflow tasks and activity tasks.

1.3.1. The Mechanics of Polling and Backpressure

When a worker starts, it initializes dedicated goroutines (or threads, depending on the SDK) called pollers. These pollers issue gRPC requests—specifically `PollWorkflowTaskQueue` and `PollActivityTaskQueue`—to the Temporal cluster’s Frontend service. The Frontend routes these requests to the Matching service, which maintains the authoritative state of the Task Queues.

If a task is available on the queue, the Matching service dispatches it immediately to the polling worker. If the queue is empty, the Matching service holds the gRPC connection open in a long poll, typically for up to one minute. When a new task arrives during this window, it is dispatched to one of the waiting long-poll connections.

Understanding the distinction between execution size limits and poller limits is critical for performance tuning. The execution size (e.g., `MaxConcurrentActivityExecutionSize`) dictates the maximum number of tasks the worker is allowed to process simultaneously in memory. The poller count (e.g., `MaxConcurrentActivityTaskPollers`) dictates how many concurrent fetch requests the worker maintains against the server.

If your worker is configured with a high execution limit but only a single poller, throughput will be artificially capped by network latency. Even if a worker can process 1,000 concurrent activities, a single poller must complete its network round trip, hand the task to the internal execution pool, and issue a new poll request before the next task can be fetched. By increasing the number of pollers, you allow the worker to fetch multiple tasks in parallel, keeping the internal execution pool saturated.

Conversely, setting the poller count too high relative to the execution size provides no throughput benefit and needlessly consumes network connections and Temporal server resources. The benchmark-workers

load-testing harnesses provided by Temporal typically run with explicitly tuned poller counts to test throughput shaping, demonstrating that optimizing this ratio is a prerequisite for high-performance deployments.

1.3.2. Differentiating Workflow Tasks and Activity Tasks

Although you specify a single Task Queue name when starting a workflow or configuring a worker, the Temporal server actually maintains two distinct physical queues under the hood: one for workflow tasks and one for activity tasks. This bifurcation exists because the execution profiles, resource requirements, and scaling semantics of workflows and activities are fundamentally different.

Workflow Tasks

Workflow tasks drive the deterministic advancement of a workflow's state. When a worker receives a workflow task, it executes your workflow code to process new events, update local state, and schedule new commands (like starting an activity or setting a timer). Because workflow code is tightly constrained-it cannot perform network I/O or block on external resources-workflow tasks are highly CPU-efficient and execute rapidly. A single worker pod can often process thousands of workflow tasks per second. High concurrency limits for workflow tasks are generally safe, provided the worker has sufficient memory to cache the workflow event histories.

Activity Tasks

Activity tasks represent the side-effecting work done in the outside world: querying databases, invoking third-party APIs, processing files, or executing heavy machine-learning models. Activity execution times are variable and heavily dependent on external systems. Consequently, activity workers are typically bound by network I/O, downstream API

rate limits, or intensive CPU usage. Activity concurrency limits must be strictly tuned to protect downstream systems from overwhelming load. While Temporal can trivially dispatch 10,000 activity tasks per second, your legacy relational database may instantly crash if 10,000 concurrent connections attempt to write to it simultaneously.

Because workflow tasks and activity tasks are decoupled at the queue level, an exhausted activity pool does not block workflow progression. If all activity execution slots are occupied waiting on a slow third-party API, the worker will stop polling the activity task queue. However, the worker's workflow task pollers remain unaffected. The worker continues fetching workflow tasks, allowing it to process cancellation requests, handle incoming signals, and enforce timeout policies, even while the activity side of the house is entirely gridlocked.

1.3.3. Task-Queue Topology and Workload Isolation

Task queues serve as the primary routing mechanism between the logical work orchestrated by Temporal and the physical compute infrastructure that executes it. Designing your task-queue topology is a critical architectural exercise that dictates how your system scales, how failures are isolated, and how teams take ownership of specific domains.

Service-Boundary Queues

The most common and effective topology aligns task queues with service deployment boundaries. If a dedicated team owns an “Inventory” service, that service should expose an `inventory-service` task queue. Other services interact with the inventory domain by scheduling activities on this queue. This pattern eliminates code-sharing bottlenecks. The calling workflow only needs to know the name of the queue and the signature of the activity; it does not need the actual activity implementation in its codebase.

Isolating Workloads by Performance Tier

When a single service handles wildly different traffic profiles, routing all traffic through a single task queue introduces the risk of head-of-line blocking. Consider a reporting service that generates real-time dashboards for active users and also processes massive, low-priority backfills overnight. If the backfill process floods the shared queue with millions of tasks, real-time dashboard requests will be stuck waiting for compute capacity.

To solve this, provision separate queues for distinct workload tiers, such as `reports-realtime` and `reports-batch`. You can then deploy independent worker pools configured specifically for each tier. The real-time worker pool might run on high-CPU instances with high concurrency limits to ensure millisecond responsiveness, while the batch worker pool runs on cheaper spot instances with strict concurrency limits to prevent the backfill from saturating the underlying database.

Separating Workflow and Activity Deployments

As systems scale, you may find that workflow executions and activity executions require conflicting infrastructure. Workflow workers require stable deployments and generous memory to maintain sticky execution caches, reducing the need to repeatedly fetch event histories from the server. Activity workers, particularly those handling CPU-intensive transformations or media encoding, may need dynamic autoscaling, GPU instances, or access to specialized subnets.

You can physically separate these pools by configuring specialized workers. Deploy one set of containers that registers only workflow definitions and polls for workflow tasks. Deploy a completely separate set of containers that registers only activity definitions and polls for activity tasks. This topological split allows you to scale the stateless, compute-heavy activity layer elastically while keeping the stateful, cache-sensitive workflow layer stable.

1.3.4. The Scaling Playbook: Observing and Adjusting Capacity

When throughput bottlenecks occur, the resolution path requires translating cluster metrics into specific worker configuration adjustments. The definitive signal of worker starvation is an increase in schedule-to-start latency. This metric tracks the time a task spends sitting in the Temporal server's queue waiting for a worker to poll it.

Monitor `temporal_workflow_task_schedule_to_start_latency` and `temporal_activity_schedule_to_start_latency` independently. When these metrics begin to rise, follow a structured diagnostic flow:

- *Check Worker Resource Utilization:* Inspect the CPU and memory consumption of your worker pods. If the pods are hitting 100% CPU utilization or exhausting their memory limits, the internal worker pools are fully saturated. The correct action is horizontal scaling-deploy more worker replicas to the cluster.
- *Check Downstream Saturation:* If worker CPU is low but activity latency is high, inspect the downstream dependencies. If your workers are waiting on a saturated database or an API rate limit, adding more workers will only exacerbate the outage. In this scenario, high schedule-to-start latency is working exactly as designed: Temporal is buffering the excess load. Do not scale up the workers; instead, address the downstream bottleneck or accept the queueing behavior.
- *Tune Poller and Concurrency Limits:* If worker CPU is low, downstream dependencies are healthy, but schedule-to-start latency remains high, your workers are under-polling relative to the backlog. They have unused compute capacity but are artificially constrained by their configuration. Increase the

`MaxConcurrent...Pollers` limit to allow more concurrent fetches, or increase the `MaxConcurrent...ExecutionSize` to allow more simultaneous local execution.

1.3.5. Common Anti-Patterns in Worker Scaling

Misunderstanding the relationship between task queues, pollers, and workers frequently leads to self-inflicted hotspots.

A common operational mistake is assuming that increasing poller counts is a universal fix for queue latency. If your worker's execution pool is already full, adding more pollers only pulls tasks off the server faster than the worker can process them, leading to local queueing inside the worker process, increased local timeouts, and an overall degradation of system predictability. Poller counts should only be increased when workers exhibit idle compute capacity during periods of high queue depth.

Another destructive pattern is unbounded dynamic queue generation. Because task queues are dynamically created by the Temporal server the first time they are referenced, engineers sometimes embed dynamic routing logic directly into their workflows, generating unique queue names per user, per request, or per database shard (e.g., `user-12345-queue`). Workers must be explicitly configured to poll specific queue names at startup. If your application generates thousands of dynamic queue names, you must dynamically provision workers to poll those queues, creating an operational nightmare of service discovery and lifecycle management. A task queue should represent a class of compute or a service boundary, not a specific row in a database. Keep the cardinality of your task queues low and handle tenant-specific routing within the payload of the activity task itself.

Finally, relying on generic rate limiters inside your application code

instead of leveraging worker concurrency settings undermines the platform's architectural benefits. If an activity starts, consumes a local worker thread, and then immediately goes to sleep because a local rate-limiter tells it to back off, it is actively consuming execution capacity while doing no work. By applying strict `MaxConcurrentActivityExecutionSize` limits at the worker configuration level, you ensure that tasks remain durably queued in the Temporal cluster-consuming zero worker memory or CPU-until actual execution capacity is guaranteed.

1.4. Understanding Workflow Executions, Runs, and Event History as the Source of Truth

To design resilient applications on Temporal, you must master the precise semantics of its core durable objects. A workflow in Temporal is not a running process in the traditional operating system sense. It is a durable, logical entity represented by an append-only log of events managed by the Temporal cluster. Workers execute code to derive the state of the workflow from this log. Understanding the boundaries between executions, runs, events, and code execution is non-negotiable for writing correct, fault-tolerant workflows.

Executions Versus Runs

A *workflow execution* represents the complete, end-to-end lifespan of a logical business process. It is uniquely identified within a namespace by a Workflow ID (e.g., `order-fulfillment-912`). Temporal guarantees that only one execution with a given Workflow ID can be running at any specific time.

A *run* is a single, continuous incarnation of that execution, identified by a globally unique Run ID (a UUID). A single workflow execution can span multiple runs. If a workflow fails and is retried according to its

retry policy, or if it explicitly restarts itself to truncate its history via the Continue-As-New mechanism, the Temporal server creates a new Run ID while preserving the original Workflow ID. Conflating executions and runs leads to operational confusion. When tracking business logic progress over time, you query by Workflow ID. When debugging a specific crash or a distinct segment of history, you isolate the Run ID.

To observe this structure in practice, inspect the event history of a running execution using the Temporal CLI. Run the following command to retrieve the history in a tabular format:

```
temporal workflow show \
  --workflow-id "order-fulfillment-912" \
  --namespace "production" \
  --fields "EventId,EventType,Details"
```

EVENTID	EVENTTYPE	DETAILS
1	WorkflowExecutionStarted	{WorkflowType: {Name: OrderWorkflo
w}, RunId: 5a2b...}		
2	WorkflowTaskScheduled	{TaskQueue: {Name: orders}, Attemp
t: 1}		
3	WorkflowTaskStarted	{RequestId: 8c1d...}
4	WorkflowTaskCompleted	{}
5	ActivityTaskScheduled	{ActivityId: 1, ActivityType: {Nam
e: ChargeCard}}		
6	ActivityTaskStarted	{Attempt: 1}
7	ActivityTaskCompleted	{Result: {"status":"success"}}
8	WorkflowTaskScheduled	{TaskQueue: {Name: orders}, Attemp
t: 1}		

Event History as the Unambiguous Source of Truth

The output above exposes the underlying reality of the Temporal platform: the event history is the state. The Temporal cluster does not store a JSON blob representing the variables in your workflow code. Instead, it durably persists every meaningful transition—start parameters, scheduling decisions, side-effect outcomes, and external signals—as an ordered sequence of events.

Events carry universal attributes (an incrementing Event ID, a times-

tamp, and an Event Type) alongside type-specific payload data. When a client passes arguments to a workflow, those arguments are serialized and stored inside the `WorkflowExecutionStarted` event. When an activity completes successfully, its return value is serialized into the `ActivityTaskCompleted` event.

Because the history is append-only and strictly ordered, it provides an exact audit trail of what transpired from the cluster's perspective. You never need to build ad-hoc database tables to track the lifecycle stages of a Temporal workflow; the event history inherently provides this capability.

Replay: Reconstituting State from History

Since the server only stores the event log, the worker must interpret that log to resume execution. This interpretation process is called *replay*.

When a Temporal worker polls a task queue and receives a `WorkflowTask`, the cluster includes the workflow's event history in the payload. The worker loads the specific workflow function, allocates a fresh thread (or coroutine), and begins executing the code from the very first line.

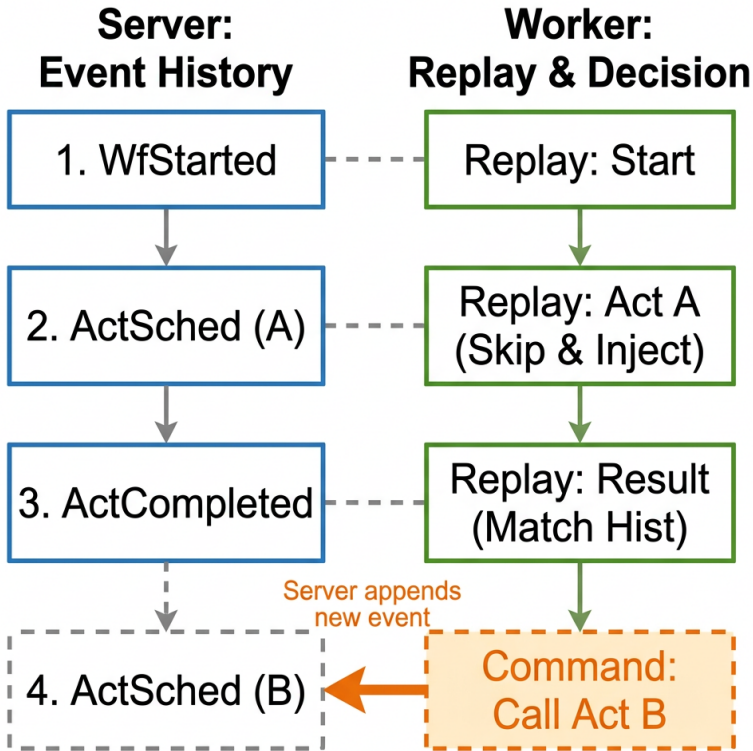
As the code executes, it inevitably issues requests to the platform, such as scheduling an activity, starting a timer, or yielding for a signal. In a standard programming model, these would be immediate network calls or system interruptions. During replay, the worker intercepts these requests and checks them against the provided event history.

If the workflow code attempts to schedule the `ChargeCard` activity, the worker examines the history. If it finds a matching `ActivityTaskScheduled` event followed by an `ActivityTaskCompleted` event (Events 5 and 7 in the CLI output), the worker does not execute the activity again. Instead, it

extracts the serialized result from Event 7, feeds it back into the workflow code as the activity's return value, and allows the code execution to advance.

The worker continues this process—matching code execution against recorded history—until it reaches the end of the log. At this point, the worker's in-memory state precisely matches the state at the time the last workflow task completed. If the workflow code then requests a new operation that is *not* in the history (e.g., sending a receipt email), the worker captures this request as a new *Command*. The worker packages all newly generated Commands and sends them back to the Temporal cluster in a `RespondWorkflowTaskCompleted` RPC. The cluster validates these Commands and translates them into new Events (e.g., `ActivityTaskScheduled`), appending them to the history.

The figure below illustrates an event history timeline mapped to replayed workflow code checkpoints, showing where a crash occurs and how replay resumes.



Platform-Level Failure Analysis

The combination of an append-only log and execution via replay introduces specific operational characteristics and failure modes that govern how you design production workloads.

Because replay demands strict agreement between the historical events and the worker code, non-determinism is treated as a severe correctness failure. If a worker replays history and the workflow code diverges—for instance, if the history indicates an `ActivityTaskScheduled` event for `ChargeCard`, but the current code

attempts to execute `SendEmail` instead—the worker safely halts execution of that task. It throws a non-determinism error, dropping the workflow task rather than corrupting the state. The cluster will continuously retry the workflow task, causing the execution to stall until you deploy code that accurately maps to the persisted history.

History size limits directly dictate workflow lifecycle design. The Temporal cluster imposes a hard limit of 50,000 events per execution and issues operational warnings when a history exceeds 10,000 events. Because workers reconstruct state by downloading and evaluating the entire log, an excessively large history bloats network payloads, incurs heavy CPU costs during replay, and leads to “replay storms”—scenarios where workers spend all available compute cycles replaying massive histories rather than making forward progress.

To prevent replay storms, limit the amount of data written to the history. Pass large payloads (like images or large JSON datasets) by reference using blob storage URIs rather than embedding them directly into activity inputs or outputs. For workloads requiring unbounded iterations, such as processing millions of records in a batch job or running an infinite monitoring loop, use the `Continue-As-New` mechanism. `Continue-As-New` safely truncates the history by capturing the required state, shutting down the current Run, and immediately starting a new Run with an empty history and the carried-over state.

Finally, align your data retention strategies with the reality that history is the primary source of truth. Temporal namespaces allow you to configure retention periods (e.g., 30 days) that dictate how long a closed workflow execution’s event history remains queryable. Once the retention period expires, the history is permanently deleted from the active cluster to reclaim storage. Ensure your cluster retention policies satisfy your compliance, support, and auditability requirements before deploying to production. Do not build parallel, database-driven audit logs from within your workflow code; leverage the exact sequence of

events natively persisted by Temporal, and use archival features if you must retain logs indefinitely.

Chapter 2

Workflow Execution Semantics and Determinism

Temporal workflows are not “running processes” in the traditional sense—they are deterministic functions repeatedly re-executed to a well-defined point by replaying an event history. Once you internalize that semantic shift, you can write orchestration logic that survives crashes, deploys safely over time (with versioning covered in Chapter 7), and scales without turning reliability into an application-level science project.

2.1. Replay mechanics: reconstructing workflow state from event history

The Temporal service does not execute your workflow code. It acts as an append-only log of events and a highly available queue. Your application workers run the code. To bridge the gap between a remote event log and a stateful, running process, Temporal relies on event sourcing and a process called replay.

To understand how a workflow progresses, you must first examine its single source of truth: the Event History.

Retrieve the history of any running or completed workflow using the Temporal CLI:

```
temporal workflow show --workflow-id user-onboarding-123
```

The output reveals a linear sequence of events rather than a snapshot of variables or memory:

```
1 WorkflowExecutionStarted {WorkflowType:{Name:OnboardingWorkflow}, TaskQueue...}
2 WorkflowTaskScheduled   {TaskQueue:{Name:onboarding-queue, Kind:Normal}...}
3 WorkflowTaskStarted      {RequestId:5a9e3b..., Identity:worker-host-1}
4 WorkflowTaskCompleted    {Identity:worker-host-1}
5 ActivityTaskScheduled     {ActivityId:5, ActivityType:{Name:SendWelcomeEmail}...}
6 ActivityTaskStarted       {Identity:worker-host-2, Attempt:1}
7 ActivityTaskCompleted     {Result:{"status": "delivered"}}
8 WorkflowTaskScheduled     {TaskQueue:{Name:onboarding-queue, Kind:Normal}...}
```

Every action the workflow takes, and every external input it receives, is memorialized in this log. To convert this log into running code, Temporal SDKs rely on three fundamental concepts:

History

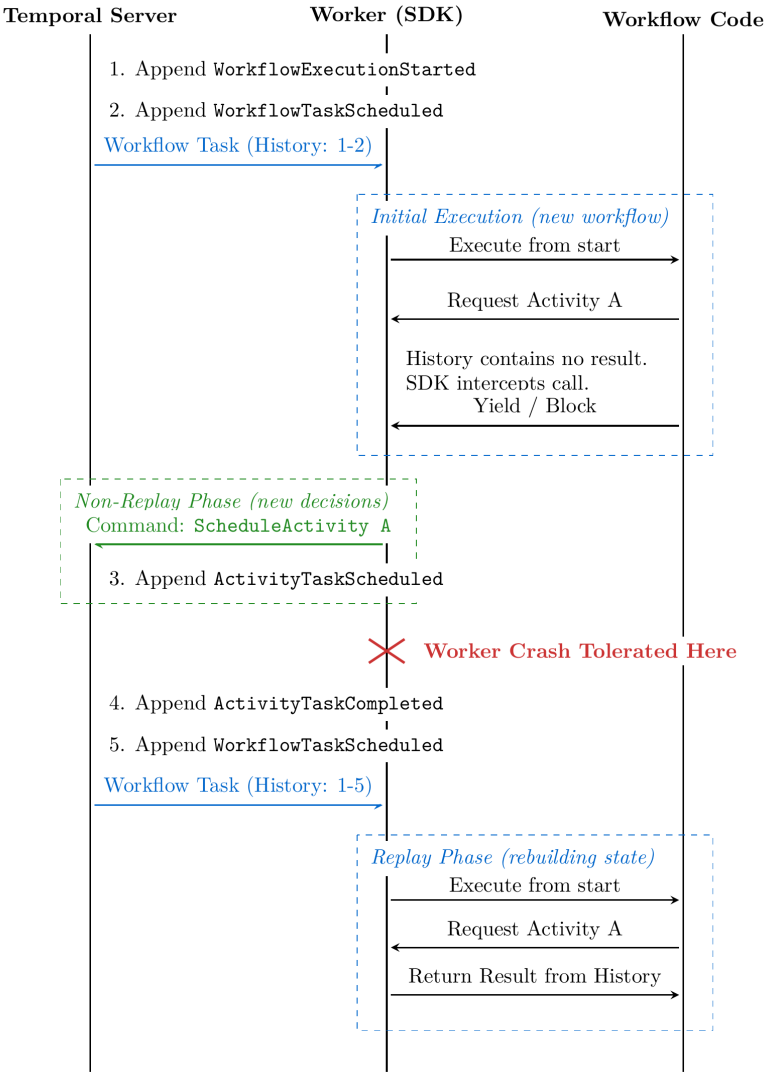
The immutable, ordered log of events durably stored on the Temporal Server. It contains the workflow's inputs, scheduling actions (like starting timers or activities), and the outcomes of those actions. History is the absolute ground truth.

Command

A proposed next step emitted by your workflow code (e.g., `ScheduleActivity`, `StartTimer`). Commands are intermediate and ephemeral. They only become permanent when the worker transmits them to the server, and the server validates and appends them to the history as events (e.g., `ActivityTaskScheduled`, `TimerStarted`).

Replay

The mechanical process of re-executing your workflow code from the very beginning of the function, using the recorded history to reconstruct the exact variable state and program counter. During replay, the SDK intercepts any calls to activities or timers. Instead of actually executing them, the SDK looks at the history. If the history says the activity already completed, the SDK returns the recorded result to the code immediately. If the history shows the activity is still pending, the SDK yields execution.



The End-to-End Workflow Task Loop

When you start a workflow, the Temporal server writes a `WorkflowExecutionStarted` event and places a Workflow Task into the task queue. A worker polling that queue receives the task, which includes the event history.

Because this is a new workflow, the history is short. The worker instantiates the workflow function and begins executing it line by line. The code evaluates inputs, sets local variables, and perhaps calls an activity:

```
@workflow.defn
class OnboardingWorkflow:
    @workflow.run
    async def run(self, user_id: str):
        # Executes locally in the worker
        setup_data = initialize_user_struct(user_id)

        # Code requests an activity; SDK intercepts
        result = await workflow.execute_activity(
            send_welcome_email,
            setup_data,
            start_to_close_timeout=timedelta(minutes=5),
        )

        await workflow.sleep(timedelta(days=30))
        return result
```

When the code reaches `execute_activity`, the SDK checks the provided history. There is no `ActivityTaskCompleted` event yet. Recognizing that the workflow cannot proceed, the SDK pauses the workflow function's execution. It collects the pending action into a Command (`ScheduleActivityTask`) and sends a `RespondWorkflowTaskCompleted` API call back to the Temporal Server, attaching the command.

The server validates the command, appends an `ActivityTaskScheduled` event to the history, and routes the activity task to the appropriate queue. The Workflow Task is now completely finished. The worker destroys the workflow object from memory (or caches it, but conceptually, the heap is discarded).

There is no thread actively waiting. The workflow state is entirely at rest in the database.

When the activity eventually succeeds, the server appends `ActivityTaskCompleted` to the history. Because a workflow was waiting on this outcome, the server immediately schedules a new Workflow Task.

A worker picks up the new Workflow Task, receiving the updated history (events 1 through 7 in the CLI output example). The SDK spawns a fresh instance of the workflow function and starts executing it from line one. This is the replay phase.

The code runs `initialize_user_struct(user_id)` again. Then it reaches `execute_activity`. This time, the SDK observes an `ActivityTaskCompleted` event in the history. It does not re-schedule the activity. Instead, it extracts the JSON payload from the history event, deserializes it, and returns it directly to the variable `result`.

The workflow unblocks and proceeds to the next line: `workflow.sleep(timedelta(days=30))`. The SDK intercepts the sleep, finds no `TimerFired` event in history, and yields execution again. It emits a `StartTimer` command to the server. The workflow heap is once again discarded. Thirty days later, the server will append `TimerFired`, schedule a Workflow Task, and the replay cycle will happen a third time, stepping seamlessly over the activity and the sleep, finally returning the result.

Contrasting the Durable Execution Mental Model

In a traditional request/response architecture, process memory is the source of truth. If you execute a function that makes three synchronous HTTP calls and the process crashes on the second call, the orchestration state is permanently lost. You must build custom database tables to track progress and background cron jobs to resume abandoned

work.

In a classic in-memory orchestrator or message queue system, you manage state by explicitly chaining callbacks or structuring DAGs (Directed Acyclic Graphs). The code structure itself must be inverted to match the state machine constraints of the underlying engine.

Temporal provides “Durable Execution.” It guarantees that a standard language function will execute to completion regardless of infrastructure failures, as long as it behaves deterministically. It achieves this by shifting the source of truth from volatile process memory to the durable Event History.

This model explains why “sleeping for days” is perfectly safe and incurs virtually no infrastructure cost. A `sleep` is simply an instruction to the Temporal Server to persist a timer and wake the workflow later. No worker threads are consumed during the sleep duration. A workflow can sleep for a decade, surviving hundreds of worker container restarts, network partitions, and database failovers, because the workflow’s state is safely persisted as a sequence of historical events.

Fault Tolerance and At-Least-Once Delivery

The replay mechanism is the foundation of Temporal’s fault tolerance. Assume a worker container loses power exactly halfway through executing a workflow task. The worker never sends the `RespondWorkflowTaskCompleted` API call to the server.

From the server’s perspective, the workflow task timed out (governed by the `WorkflowTaskTimeout`, which defaults to 10 seconds). The server simply places a new Workflow Task back into the queue. A different, healthy worker picks it up, receives the exact same history, and replays it. The new worker reconstructs the identical heap state and resumes execution perfectly.

This architecture fundamentally relies on at-least-once task delivery

from the server, combined with deterministic replay in the SDK. As long as your workflow code behaves predictably during replay, at-least-once task delivery results in exactly-once logical execution. You do not need to build idempotency checks for your orchestration logic; the platform provides it inherently. However, external side effects executed within activities still require idempotency, as they are subject to separate retry policies.

What Does Not Happen During Replay

Understanding what workflows cannot do is just as critical as understanding what they do. Because workflow code may be executed dozens or hundreds of times during its lifecycle to rebuild state, it must not cause unintended external side effects.

A workflow function does not perform network I/O, write to local disks, or mutate external databases directly. If a workflow initiated a direct HTTP POST request to a payment gateway, it would execute that request every time a worker replayed the history, resulting in multiple charges. All interactions with the outside world must be delegated to Activities. Activities guarantee that the outcome is recorded in the history exactly once, ensuring that subsequent replays merely read the recorded result instead of re-executing the side effect.

From Incident to Root Cause Using First-Principles Replay

The replay model gives you a highly precise methodology for debugging stuck or failing workflows. When an incident occurs, you do not need to guess what the workflow is currently doing; the history tells you exactly where the instruction pointer is paused.

If a workflow is reported as “stuck,” retrieve its history and locate the final `WorkflowTaskCompleted` event. This event represents the exact moment the worker last successfully executed the workflow code and yielded commands.

Look at the events immediately following it.

- If you see `ActivityTaskScheduled` but no `ActivityTaskStarted`, the workflow logic is healthy. It evaluated the code, requested an activity, and yielded correctly. The root cause lies in your worker fleet-either there are no workers polling the specific activity task queue, or the workers are saturated.
- If you see `ActivityTaskStarted` without completion, the activity is actively executing, hanging, or silently failing and retrying in a loop.
- If you see `TimerStarted` but no `TimerFired`, the workflow is intentionally asleep. Check the timer duration against the timestamps.
- If the final event is `WorkflowExecutionSignaled`, but there is no subsequent `WorkflowTaskCompleted`, the workflow received an external signal, but the worker failed to process it. This typically indicates a panic or a determinism error crashing the workflow task repeatedly.

By mentally tracing the history against your source code, you simulate the replay process yourself. You can point to the exact line of code where the SDK encountered an unfulfilled history event and yielded. This deterministic relationship between code and history is the cornerstone of operating Temporal workloads at scale.

2.2. Determinism rules: what must be stable under replay and how it breaks

Workflow determinism in Temporal hinges on a single, uncompromising rule: given the same event history, your workflow code must pro-

duce the exact same sequence of commands. When a workflow executes, it is not merely running top-to-bottom; it is dynamically building a state machine. Every time the worker processes a workflow task, it replays the historical events from the server to reconstruct the workflow's state. If the code deviates during this replay—attempting to schedule a different activity, fire a different timer, or change the order of operations—the state machine breaks.

To understand how determinism breaks in practice, examine a simple workflow that dynamically generates a tracking identifier and passes it to an activity.

```
func OnboardUserWorkflow(ctx workflow.Context, user User) error {
    // DANGER: Non-deterministic native library call
    trackingID := uuid.New().String()

    // The command generated here embeds the non-deterministic string
    err := workflow.ExecuteActivity(ctx, SendWelcomeEmail,
        user.Email, trackingID).Get(ctx, nil)
    if err != nil {
        return err
    }
    return nil
}
```

When this workflow runs for the first time, `uuid.New().String()` generates a unique value, such as `A1B2`. The SDK buffers a `ScheduleActivityTask` command containing `A1B2` in its payload. The worker sends this command to the server, and the server appends an `ActivityTaskScheduled` event to the workflow history.

Suppose the worker crashes before the activity completes. The server dispatches a new workflow task to another worker. This new worker begins replaying the workflow from the start, using the existing history. The workflow code executes again, and `uuid.New().String()` generates a completely new value, `C3D4`. The code then requests to schedule the `SendWelcomeEmail` activity with the payload `C3D4`.

At this exact moment, the SDK's internal state machine detects a fatal

divergence. It compares the command proposed by the replaying code (`ScheduleActivityTask` with C3D4) against the next event in the historical record (`ActivityTaskScheduled` with A1B2). Because the payloads do not match, the SDK refuses to proceed and panics the workflow task.

```
go.temporal.io/sdk/internal.NonDeterministicWorkflowError:
Nondeterminism: history event is ActivityTaskScheduled,
but workflow code generated ActivityTaskScheduled with different arguments.
```

This failure mechanism is intentional and protective. If the SDK allowed the divergent command to pass, the workflow's state would decouple from its durable record, leading to data corruption, duplicated external side effects, and unrecoverable execution states.

Mechanics of command validation To diagnose and prevent these errors, you must understand how the SDK validates commands during replay. The SDK maintains a pointer into the event history. As the replaying workflow code executes, it invokes SDK APIs (like `ExecuteActivity` or `NewTimer`). These APIs do not immediately execute network calls; instead, they act as interceptors. During replay mode, the interceptor checks the current history pointer. If the proposed command matches the historical event—by command type, target name, and exact payload—the pointer advances, and the code is allowed to continue. If there is a mismatch, a `NonDeterministicWorkflowError` is raised immediately.

This mechanical strictness means that non-determinism is not inherently a problem of code structure; it is a problem of data consistency between the workflow's memory and the server's database.

Taxonomy of non-determinism

The sources of determinism violations generally fall into four categories: time, randomness, external I/O, and concurrency. Understanding how each manifests during replay allows you to proactively identify unsafe code before it reaches production.

System time and calendars Wall-clock time constantly moves forward. Calling `time.Now()` or `System.currentTimeMillis()` directly in workflow code is universally unsafe. If a workflow branches based on the current time (e.g., checking if an SLA has expired), the initial execution might take the “SLA met” branch, while a replay triggered hours later might take the “SLA breached” branch.

Instead, you must extract time exclusively from the workflow context. SDKs provide deterministic time APIs, such as `workflow.Now(ctx)` in Go or `Workflow.currentTimeMillis()` in Java. These APIs do not read the host’s hardware clock. Instead, they return the timestamp of the workflow task that originally triggered the execution phase. Because this timestamp is baked into the immutable history, replays consistently return the exact same value they returned during the initial run.

Time zones and daylight saving time (DST) introduce subtler determinism risks. Even if you use the deterministic workflow clock, converting that UTC timestamp to a local time zone using the host machine’s configuration is dangerous. If the workflow originally ran on a server configured for `America/New_York` and later replays on a server configured for UTC, date-boundary calculations will shift. Pass time zone offsets explicitly as workflow inputs or activity results, and never rely on the worker OS environment.

Randomness and unique constraints Like wall-clock time, random number generators (RNG) and UUID generators yield novel outputs on every invocation. If workflow control flow, or the payloads passed to activities, rely on these native functions, the resulting commands will drift during replay.

When you require unique identifiers or random distributions within workflow logic, use the SDK’s deterministic equivalents. The SDKs provide pseudo-random number generators seeded by the workflow

run ID. Because the run ID never changes for a given execution, the sequence of random numbers generated during replay perfectly mirrors the original sequence.

When you must integrate a native random function, such as a cryptographic nonce generator from a third-party library, wrap the execution in the SDK's `SideEffect` primitive. `SideEffect` runs the provided function once, records its output into the event history as a marker, and immediately returns that output. During subsequent replays, `SideEffect` bypasses the function entirely and simply returns the recorded value from history. This transforms a non-deterministic operation into a durable, replay-safe constant.

I/O and environment dependence Workflow logic runs in a vacuum. Any attempt to read the external environment directly—whether fetching an environment variable, reading a local file, or invoking a REST API—creates a dependency on volatile state.

Feature flags are the most common trap in this category. Consider a workflow that checks a global feature flag registry to decide whether to route an order to a legacy processing system or a new processing system. If the workflow executes on Monday while the flag is disabled, it routes to the legacy system. If the flag is toggled on Tuesday, and the Monday workflow is forced to replay (perhaps due to a worker restart), the workflow code will evaluate the flag as enabled. It will attempt to route to the new system, contradicting the history, and crash with a nondeterminism error.

Treat the workflow code as a pure orchestration function. If a decision requires external data, that data must either be passed in as the original workflow argument or fetched via an Activity. Activities execute exactly once, record their results in history, and return those recorded results during replay.

Concurrency and iteration order Implicit ordering assumptions

are a frequent source of invisible non-determinism. Standard hash maps in modern programming languages (such as Go's `map` and Python's `dict` prior to 3.7) intentionally randomize their iteration order to prevent hash-collision denial-of-service attacks. If you iterate over a map of items to dynamically schedule parallel activities, the sequence of `ScheduleActivityTask` commands emitted to the SDK will randomly shuffle on every replay.

To fix this, either sort the keys deterministically before iteration or use ordered collection types.

Similarly, you cannot rely on language-native threading or asynchronous event loops. Spawning raw goroutines, Java `Threads`, or Python `asyncio.Tasks` leaves the execution interleaving up to the operating system's CPU scheduler. If two threads modify a shared workflow variable, the final state depends entirely on a race condition. SDKs provide deterministic concurrency primitives (such as workflow-scoped goroutines and promises) that execute purely within the boundary of a single workflow task, ensuring that logical concurrency behaves identically across replays.

Furthermore, floating-point arithmetic can introduce hardware-level non-determinism. Different CPU architectures (e.g., x86 vs. ARM) or compiler optimization levels can produce micro-variations in floating-point precision. While rare in standard line-of-business applications, scientific or financial workflows that perform complex floating-point math directly in the workflow body should offload those calculations to activities to ensure the final calculated value is locked into the event history.

Language SDK variations and sandboxing Enforcement of determinism rules varies by language SDK. The Java and TypeScript SDKs provide robust runtime sandboxing. The TypeScript SDK utilizes V8 isolates to explicitly block access to standard library modules like

`Math.random()`, `Date.now()`, and native filesystem APIs from within workflow code. The Java SDK employs thread-local storage and custom constraints to warn against or outright block non-deterministic system calls.

In contrast, the Go and Python SDKs do not natively sandbox workflow code. They rely entirely on developer discipline and static analysis tools. In these languages, the compiler will not prevent you from calling `time.Now()` or spinning up a raw thread. You must enforce these rules via code reviews, linting, and rigorous testing. Regardless of the language, the server's history validation remains identical. The safety net is always there, but sandboxed languages catch the error at execution time, whereas non-sandboxed languages catch it at replay time.

Diagnosing determinism failures When a determinism violation occurs, the worker does not fail the entire workflow execution. Instead, the workflow task fails, the worker logs the exception, and the server schedules a retry of the workflow task. Because the underlying code and history mismatch remains, the task will repeatedly fail and retry indefinitely, effectively stalling the workflow execution.

This stall is an operational feature. It prevents the workflow from taking destructive actions based on a corrupted state. To resolve the incident, you must diagnose the source of the divergence.

First, identify whether the mismatch originates from your direct code, a third-party dependency, or an environment shift. Inspect the `NonDeterministicWorkflowError` message. It will specify exactly which command was expected and which command was generated.

If the error indicates that history expected a `TimerStarted` event with a duration of 3600s, but the code generated a `TimerStarted` event with a duration of 7200s, the diagnosis is straightforward: someone changed the timer duration in the workflow code without using a versioning API. The deployment of the new code caused in-flight workflows to replay

against old histories using the new logic.

If the error indicates a command ordering mismatch—for example, history expects `ActivityTaskScheduled(ActivityA)` but the code generated `ActivityTaskScheduled(ActivityB)`—look for changes in `if/else` branching, loop structures, or hash map iterations.

Third-party dependencies require deeper scrutiny. If you import a library to validate a payload, and that library internally calls a time function or spawns a background thread, it contaminates the workflow. Always audit external libraries invoked directly within workflow functions. If a library is not guaranteed to be pure and deterministic, wrap its invocation in an activity.

To debug a complex violation locally, use the SDK's Replayer utility. The Replayer allows you to download the JSON event history of the stalled workflow from the production server and feed it directly into your local testing environment.

```
import "go.temporal.io/sdk/worker"

func TestReplayHistory(t *testing.T) {
    replayer := worker.NewWorkflowReplayer()
    replayer.RegisterWorkflow(ProcessInvoiceWorkflow)

    // Load the JSON history exported from the production cluster
    err := replayer.ReplayWorkflowHistoryFromJSONFile(nil, "history.
        json")
    if err != nil {
        t.Fatalf("Determinism failure reproduced locally: %v", err)
    }
}
```

Executing the Replayer against the production history will reliably crash at the exact line of code causing the divergence. By stepping through the Replayer execution with a debugger, you can inspect the state variables just before the mismatch occurs, allowing you to pinpoint exactly where the workflow logic deviated from the historical record.

Once you isolate the cause, you have two paths to remediation. If the non-determinism was introduced by a recent deployment changing the workflow structure, you must either roll back the deployment or wrap the new logic in the SDK's versioning APIs (which isolate new logic paths from old histories). If the non-determinism stems from an unsafe call (like an accidental `time.Now()`), you must patch the code to use the deterministic SDK equivalent. Upon deploying the fix, the stalled workflow tasks will automatically succeed on their next retry, and the workflow execution will seamlessly resume.

2.3. Designing replay-safe workflow logic: explicit state, stable branching, and deterministic concurrency

When writing workflow logic, you must abandon the assumption that your function executes sequentially from start to finish in a single unbroken process. Instead, structure your workflow as a pure function that transforms explicit state and immutable history events into a sequence of new commands. This perspective shift requires specific coding disciplines: managing state explicitly, branching only on stable conditions, and coordinating concurrency using specialized deterministic primitives rather than standard library threading.

Consider a practical example of an order fulfillment workflow written in Java. This implementation demonstrates how to isolate state, wait for durable signals, and orchestrate concurrent tasks deterministically:

```
public class OrderWorkflowImpl implements OrderWorkflow {
    private enum State { PENDING, APPROVED, PACKED, SHIPPED,
        CANCELLED }

    private State currentState = State.PENDING;
    private final FulfillmentActivities activities =
        Workflow.newActivityStub(FulfillmentActivities.class,
            ActivityOptions.newBuilder())
```

```

        .setStartToCloseTimeout(Duration.ofMinutes(10))
        .build());

@Override
public void processOrder(String orderId) {
    // Explicit state machine blocking on a condition
    Workflow.await(() ->
        currentState == State.APPROVED || currentState == State.
CANCELLED
    );

    // Stable branching based on explicit state
    if (currentState == State.CANCELLED) {
        return;
    }

    // Deterministic concurrency: fan-out
    Promise<Boolean> packPromise =
        Async.function(activities::packOrder, orderId);
    Promise<Boolean> labelPromise =
        Async.function(activities::printLabel, orderId);

    // Deterministic concurrency: fan-in join
    Promise.allOf(packPromise, labelPromise).get();

    // State update based on durable activity results
    if (packPromise.get() && labelPromise.get()) {
        currentState = State.PACKED;
        activities.shipOrder(orderId);
        currentState = State.SHIPPED;
    } else {
        currentState = State.CANCELLED;
    }
}

@Override
public void approveOrder() {
    if (this.currentState == State.PENDING) {
        this.currentState = State.APPROVED;
    }
}

@Override
public void cancelOrder() {
    this.currentState = State.CANCELLED;
}
}

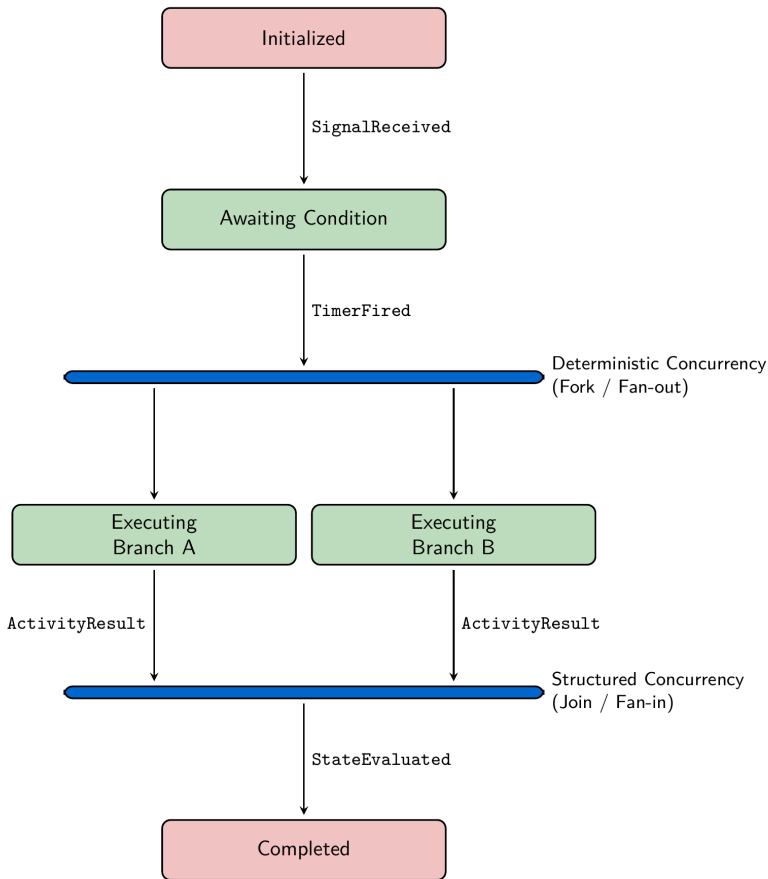
```

Explicit state machines

Standard application logic frequently buries state within the call stack—in local variables, deeply nested conditional blocks, or closure environments. In a replay-based environment, relying on implicit call-stack state makes code brittle. Refactoring a nested `if` statement or extracting a method can unintentionally alter the sequence of commands generated during replay, resulting in a nondeterminism error.

To protect against this, model workflows as explicit state machines. Define a distinct set of operational states—such as `PENDING`, `APPROVED`, or `SHIPPED`—and store the current state in a workflow-scoped variable. Drive every transition between these states with a durable history event, such as an activity result, a fired timer, or an incoming signal.

When you pair explicit state variables with SDK-provided blocking constructs—like `Workflow.await` in Java or `workflow.Await` in Go—you decouple the workflow’s logical progression from its control-flow nesting. The workflow simply blocks until an event mutates the state variable to satisfy the wait condition. During a workflow task, the SDK replays history, evaluates the condition against the reconstructed state, and either continues execution or suspends the workflow, yielding the thread back to the worker pool.



Structuring logic around explicit state greatly reduces the surface area for nondeterminism. Because transitions map cleanly to recorded history events, determining why a workflow is stalled becomes an exercise in reading the current state variable and correlating it with the last recorded event, rather than reverse-engineering a complex call stack.

Stable branching

Every `if`, `switch`, or loop condition in your workflow code is a decision point. For a workflow to be replay-safe, every decision point must be stable: it must resolve to the exact same boolean or discrete outcome every time the code executes against a given history.

To achieve stable branching, condition your decisions strictly on one of three sources:

- Workflow input arguments.
- Previously recorded history events (such as activity results or signal payloads).
- Explicit workflow state that was mutated deterministically during replay.

A common mistake is computing a branch condition using volatile external data. For instance, querying a database directly or checking a live API endpoint inside an `if` statement violates determinism because the external system's response will inevitably change over time. When replay encounters that branch days or weeks later, the live check will return a different value, redirecting the control flow and causing a non-determinism exception when the newly generated commands mismatch the historical commands.

Push volatile reads out of the workflow logic and into activities. If your workflow needs to branch based on whether a user has a premium subscription, execute a `checkSubscriptionStatus` activity, and branch on its recorded result. The activity execution acts as a durable snapshot of the external world at that moment in time. When the workflow replays, the SDK bypasses executing the activity again, injects the recorded result from history, and ensures the branch evaluates exactly as it did originally.

For operations that compute a value locally but involve sources of non-

determinism (such as generating a random UUID to use as an idempotency key, or accessing a system property), use the SDK's `SideEffect` primitive. `SideEffect` executes a block of code exactly once and records its return value in the event history. On subsequent replays, the block is skipped and the recorded value is returned. Reserve `SideEffect` for short, unfailing, and purely local computations; for anything that interacts with the network or risks failure, use an activity.

By restricting branch conditions to durable inputs, you create highly auditable code. When examining a workflow execution, you can answer precisely why a particular code path was taken by simply looking at the recorded event history, without guessing at the transient state of external systems at the time the decision was made.

Deterministic concurrency

Workflow executions frequently need to orchestrate multiple tasks concurrently, such as kicking off several independent activities and waiting for all of them to complete. However, you cannot implement concurrency using standard language mechanisms like OS threads, thread pools, native Goroutines, or standard library futures. Doing so decouples the execution order from the SDK's deterministic replay loop. If standard threads are used, the operating system scheduler dictates the order in which concurrent operations emit commands, resulting in interleaved command sequences that vary from one execution to the next.

Instead, enforce structured concurrency using the deterministic primitives provided by the Temporal SDK. In Java, this means using `Async.function` or `Async.procedure` to return `Promise` objects. In Go, it means using `workflow.Go` combined with `workflow.Channel` or `workflow.Future`. In TypeScript, standard native `Promise` constructs are instrumented and mapped to deterministic execution contexts under the hood.

When you use SDK-provided primitives, you define a deterministic concurrency graph. You specify logical concurrency (which operations may proceed independently) and dependencies (where operations must wait), creating explicit fork and join points. The SDK runtime, rather than the OS, schedules these operations. It ensures that concurrent branches execute in a strictly defined order based on the arrival of history events.

Consider the fan-out/fan-in pattern shown in the earlier Java example. Using `Async.function`, the workflow forks two independent activity executions. By passing both promises to `Promise.allOf().get()`, the workflow creates a join point, suspending execution until both activities complete. Regardless of which activity finishes first in the real world, the workflow code wakes up and resumes execution deterministically, processing the history events in the exact sequence they were recorded by the server.

Iteration order is another subtle source of concurrency nondeterminism. If you iterate over an unordered collection-like a standard `HashMap` in Java or a `map` in Go-to spawn concurrent tasks, the order in which those tasks are scheduled will vary across executions. Always sort collections or use ordered data structures (such as `LinkedHashMap` or a simple `List`) before iterating over them to spawn concurrent work.

Structure your deterministic concurrency with cancellation boundaries in mind. When you fan-out multiple concurrent operations, you must account for what happens if one operation fails or if the workflow receives a cancellation request. SDK concurrency primitives propagate cancellation tokens automatically to child tasks and activities within their scope. By binding concurrent operations to these deterministic scopes, you ensure that cancellation cascades predictably, cleaning up parallel branches without stranding resources or leaving the workflow in an undefined state.

2.4. Controlling history growth with Continue-As-New: rollover patterns and state handoff

Event histories are not merely storage; they dictate the computational cost of workflow execution. Every workflow task dispatch requires the worker to evaluate history events against your compiled workflow code to reconstruct state. In short-lived workflows, this cost is invisible. In long-lived or highly active workflows—such as continuous IoT telemetry processors, ongoing user subscription lifecycles, or eternal background monitors—history accumulation becomes a critical operational variable.

As a workflow accumulates events (timers firing, activities completing, signals arriving), the effort required to rebuild its state grows. Workers must fetch larger history payloads over the network, allocate more memory to hold the deserialized events, and spend more CPU cycles iterating through the replay loop. To prevent unbounded resource consumption, Temporal enforces strict limits on history size: a workflow execution issues warnings when history reaches 10 MB or 10,240 events, and is forcefully terminated if it reaches 50 MB or 51,200 events.

To maintain workflows that operate indefinitely, you must explicitly sever the history while preserving the logical orchestration. This is achieved through Continue-As-New, a platform command that atomically completes the current workflow run and starts a fresh run with the same Workflow ID, a new Run ID, and an empty history.

Triggering Continue-As-New

Continue-As-New is initiated from within the workflow code. Instead of returning a standard completion or an error, the workflow returns a specialized Continue-As-New command containing the inputs for the next run.

Consider a tracking workflow that continuously monitors a user's loyalty points via signals. To prevent the signal history from growing indefinitely, the workflow enforces a rollover threshold.

```
package loyalty

import (
    "context"
    "go.temporal.io/sdk/workflow"
)

type LoyaltyState struct {
    CustomerID      string
    TotalPoints     int
    EventsHandled   int
}

func LoyaltyTrackerWorkflow(ctx workflow.Context, state LoyaltyState)
    error {
    logger := workflow.GetLogger(ctx)

    signalChan := workflow.GetSignalChannel(ctx, "add_points")

    // Rollover threshold to prevent history limits
    const maxEventsPerRun = 1000

    for state.EventsHandled < maxEventsPerRun {
        var pointsToAdd int
        // Block until a signal is received
        signalChan.Receive(ctx, &pointsToAdd)

        state.TotalPoints += pointsToAdd
        state.EventsHandled++

        logger.Info("Points updated", "Total", state.TotalPoints)
    }

    // Reset the event counter for the new run, but keep the total
    points
    state.EventsHandled = 0

    // Return the ContinueAsNew error to trigger the rollover
    return workflow.NewContinueAsNewError(ctx, LoyaltyTrackerWorkflow,
        state)
}
```

When this code executes and the event threshold is met, the SDK emits

a `ContinueAsNew` command back to the server. The server responds by writing a final event to the current run's history and immediately creating the new run. You can observe this transition directly via the command line interface by inspecting the execution history:

```
$ temporal workflow show --workflow-id customer-123

...
3004 WorkflowExecutionSignaled {SignalName: add_points, Input: 50}
3005 WorkflowTaskScheduled     {}
3006 WorkflowTaskStarted       {}
3007 WorkflowTaskCompleted     {}
3008 WorkflowExecutionContinuedAsNew {
      NewExecutionRunId: "a1b2c3d4-5678-90ab-cdef-1234567890ab",
      Input: {"CustomerId": "customer-123", "TotalPoints": 15000, "Events
Handled": 0}
}
```

The logical workflow remains active and addressable by the same Workflow ID `customer-123`, but the underlying history payload is reset.

The cost of eternal workflows and recognizing unhealthy growth

Understanding why large histories degrade performance requires examining worker mechanics. Workers cache workflow state in memory to optimize subsequent workflow tasks. If a workflow receives a signal while its state is cached, the worker only needs to process the incremental events. However, caches are bounded and routinely evicted due to memory pressure, idle timeouts, or worker restarts (such as during a deployment).

When a worker receives a workflow task for an evicted execution, it must perform a full replay. If the execution has 40,000 events, the worker must download the entire history from the server. Because network protocols paginate data, this requires multiple round trips, compounding latency. Once downloaded, the worker steps through

all 40,000 events, allocating memory for every local variable, promise, and internal SDK object created along the way.

This creates operational symptoms long before the server enforces the 51,200 event termination limit:

- *Slow wake-ups after deployments:* Because a deployment restarts workers and flushes the cache, all active workflows must be fully replayed. Workflows with massive histories can take seconds to rebuild state, causing a spike in workflow task latency.
- *Memory spikes:* The object graph required to represent thousands of loop iterations or branching paths strains garbage collection.
- *High worker CPU:* Parsing and matching tens of thousands of history events against the workflow code consumes significant compute cycles.

Monitor server metrics such as `history_size` and `history_count`, and establish alerts well below the physical limits—typically around 5,000 events. Common culprits for rapid growth include high-frequency polling loops utilizing short timers, chatty signals from IoT devices or webhooks, and retry loops that bypass standard activity retry policies.

State handoff patterns

Because Continue-As-New starts a completely new execution, local variables and memory state are lost. You must deliberately serialize the operational state into the Input of the new run. Designing this state payload requires treating the handoff as an internal API contract. Several patterns solve different architectural needs:

Snapshot-and-roll is the most common approach, utilized in the code example above. The workflow maintains an internal data structure rep-

representing the current aggregate state (such as a points total, a user profile, or a configuration map). On rollover, the workflow discards the transaction log (the individual signals) and passes only the aggregate snapshot to the next run. The constraint here is payload size. The state payload must fit within standard gRPC limits (typically 2 MB). If your snapshot represents an unbounded list of items—such as keeping an array of every order ID a customer has ever placed—you will eventually hit payload limits. Unbounded collections must be pushed to external storage, leaving only reference IDs or compact aggregates in the workflow state.

Windowed processing binds the rollover to time rather than event volume. This is highly effective for reporting, aggregation, or batching workloads. A daily metric aggregator workflow might capture signals continuously but check the workflow clock upon each signal. When the workflow clock crosses midnight, the workflow executes an activity to persist the daily rollup to a database, resets its internal counters, and continues-as-new into the next day. This explicitly binds the execution history to a known temporal window, making debugging predictable (e.g., “Run ID X corresponds to Tuesday’s data”).

Accumulator with compaction handles scenarios where detailed event data must be tracked temporarily but can be compressed before the handoff. For example, a workflow tracking user navigation paths might accumulate individual click events. Before calling Continue-As-New, the workflow executes a compaction algorithm—grouping identical repeated actions or pruning stale intermediate steps—passing only the optimized path to the next run.

Design constraints and concurrency drainage

While Continue-As-New preserves the Workflow ID and guarantees continuity, the transition is destructive to in-flight operations. It is not a silent background mechanism; it represents the termination of the

current function invocation.

If you return a Continue-As-New command while an activity is currently executing, the current workflow run closes immediately. The pending activity continues running on its worker, but when it attempts to report completion, the server will discard the result because the Run ID that scheduled it is already closed. The new run knows nothing about the activity.

To maintain idempotency and correctness, you must drain pending operations before rolling over. If your workflow utilizes parallel activities or asynchronous futures, the code must block and wait for all outstanding promises to resolve before returning the Continue-As-New command.

```
// Assume 'futures' is a slice of workflow.Future tracking pending
// activities
if state.EventsHandled > maxEventsPerRun {
    // 1. Stop accepting new logical work (application specific logic
    //    )
    state.AcceptingNewWork = false

    // 2. Wait for all in-flight activities to complete
    for _, f := range futures {
        var result ActivityResult
        err := f.Get(ctx, &result)
        if err != nil {
            return err // Handle activity failure appropriately
        }
        state.ApplyResult(result)
    }

    // 3. All state is settled; safe to rollover
    return workflow.NewContinueAsNewError(ctx, MyWorkflow, state)
}
```

Signals require different handling. The server guarantees that if a signal is sent to a workflow during the exact moment it is continuing-as-new, the signal is not lost. The server appends it to the history of the newly created run. However, inside your workflow code, you must ensure you do not inadvertently drop signals that have already been deliv-

ered to the worker but remain unread in the channel. Always perform a final non-blocking read on signal channels to capture any buffered signals, updating your snapshot state before executing the handoff.

Architectural boundaries: Continue-As-New vs Child Workflows

Continue-As-New is strictly an optimization for maintaining a single, long-lived orchestration thread. It replaces the current execution. When designing complex systems, developers sometimes confuse Continue-As-New with the need to spawn independent or parallel life-cycles.

If a workflow processes a file and needs to trigger 1,000 independent record-processing tasks, rolling over does not solve the concurrency problem. Continue-As-New operates sequentially. If you need distinct operational boundaries, separate failure domains, or the ability to run massive fan-out operations, you must decompose the logic into Child Workflows. A parent workflow orchestrating many child workflows keeps its own history small because the heavy execution logic and event appending happen within the isolated histories of the children.

Use Continue-As-New when the workflow represents a single logical entity (like a single user, a single device, or a single recurring cron job) whose lifetime simply outpaces the physical constraints of an event history. By consciously managing state handoffs, enforcing payload limits, and ensuring clean operational drainage, you convert unbound resource consumption into a stable, indefinitely scalable orchestration pattern.

Chapter 3

Activities: Side Effects, Timeouts, Retries, and Idempotency

Workflows are durable because they replay; activities are safe because they don't. This chapter teaches you to design the seam between deterministic orchestration and non-deterministic side effects so that retries become a feature rather than a source of duplicate charges, phantom emails, and irreversible data corruption. By the end, you'll be able to specify timeouts and retry policies as explicit contracts, implement idempotency that survives restarts, and choose when to let the SDK retry versus orchestrating more intelligent recovery in the workflow.

3.1. Placing Side Effects Correctly: The Workflow/Activity Determinism Boundary

The foundational rule of durable execution is that workflow code is replayed, while activity code is not. Internalizing this distinction requires understanding what replay means operationally. When a worker process crashes or a task is suspended, the orchestrator does not resume the workflow from a captured memory dump. Instead, the orchestrator reconstructs the workflow's state by re-executing its code from the beginning, feeding it the exact same sequence of recorded events—the history—it observed during its original run.

Because of this reconstruction process, any non-deterministic operation inside a workflow is structurally incompatible with replay. Reading the system clock, generating a random identifier, inspecting an environment variable, or making a direct network call all produce varying results across different executions. If the workflow code generates a new identifier or observes a different network response during a replay, its execution path diverges from the durably recorded history. The orchestrator detects this divergence, categorizes it as a non-determinism error, and halts the workflow to prevent state corruption.

To isolate this risk, you must strip workflows of direct side effects and push all non-deterministic logic into activities. Workflows act as pure orchestrators, while activities execute I/O. Consider the following Java implementation, which highlights the critical difference between a native system call and an orchestration-safe equivalent:

```
// BAD: Non-deterministic workflow (will fail on replay)
public class OrderWorkflowImpl implements OrderWorkflow {
    @Override
    public OrderResult processOrder(OrderRequest req) {
        // ERROR: System.currentTimeMillis() returns a new value on
        // replay
        long startTime = System.currentTimeMillis();

        // ERROR: Direct network I/O inside the workflow
    }
}
```


3.1. PLACING SIDE EFFECTS CORRECTLY: THE WORKFLOW/ACTIVITY DETERMINISM BOUNDARY

```
        RestTemplate rest = new RestTemplate();
        String receipt = rest.postForObject("https://api.stripe.com/
charge", req, String.class);

        return new OrderResult(receipt, startTime);
    }
}

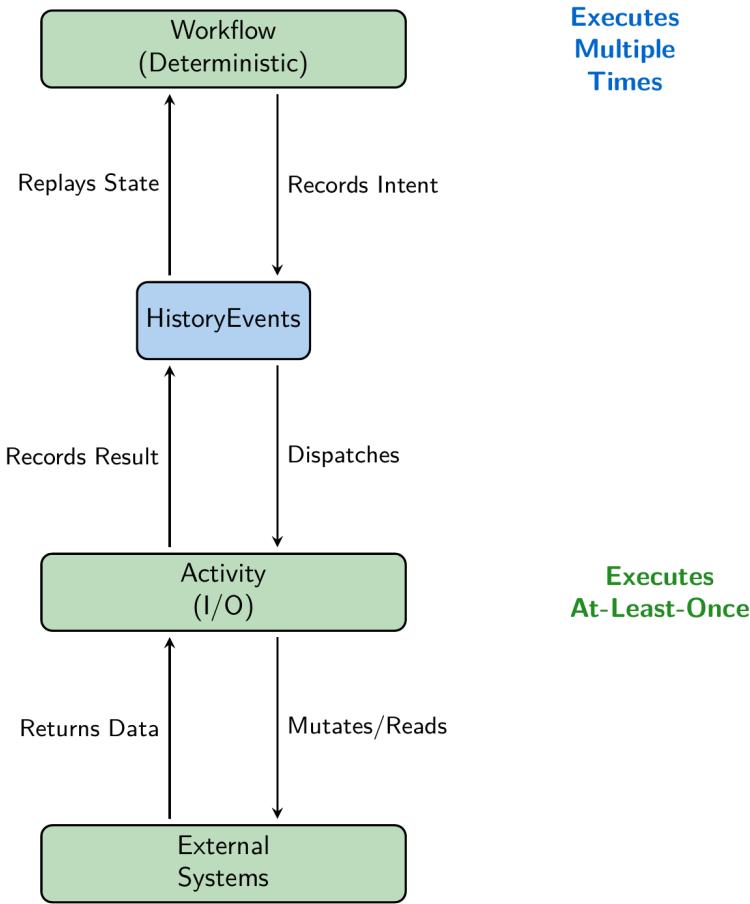
// GOOD: Deterministic workflow orchestrating an activity
public class OrderWorkflowImpl implements OrderWorkflow {
    // Activity stub proxies calls to the activity worker
    private final PaymentActivities activities = Workflow.
newActivityStub(
        PaymentActivities.class,
        ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(10))
            .build()
    );

    @Override
    public OrderResult processOrder(OrderRequest req) {
        // SAFE: Workflow.currentTimeMillis() replays
        deterministically
        long startTime = Workflow.currentTimeMillis();

        // SAFE: Side effect delegated to an activity
        String receipt = activities.charge(req);

        return new OrderResult(receipt, startTime);
    }
}
```

In the flawed implementation, the orchestrator has no visibility into the network call. If the worker crashes immediately after the HTTP request succeeds, the orchestrator will retry the workflow from scratch, executing the payment charge a second time. In the corrected implementation, the `activities.charge(req)` invocation does not execute a local function. Instead, it instructs the orchestrator to schedule an external task.



The Mechanics of Intent and Observation

To architect reliable workflows, you must build an accurate mental model of what actually happens across this boundary. When a workflow schedules an activity, it records an `ActivityTaskScheduled` intent into the event history and yields control back to the orchestra-

tor. The orchestrator places this intent on a task queue. A separate worker picks up the task, executes the effectful activity code, and upon completion, reports back to the orchestrator, which records an `ActivityTaskCompleted` or `ActivityTaskFailed` event. The workflow then resumes, reading the recorded completion event to continue its execution.

You can inspect this mechanism directly using the platform CLI. By dumping the execution history, you will see that the actual source code is never serialized; only the orchestrator's state transitions are captured:

```
temporal workflow show \  
  --workflow-id order-processing-123 \  
  --fields "EventId,EventType,ActivityTaskScheduledEventAttributes.  
    ActivityType"
```

```
1 WorkflowExecutionStarted  
2 WorkflowTaskScheduled  
3 WorkflowTaskStarted  
4 WorkflowTaskCompleted  
5 ActivityTaskScheduled      ActivityType: ChargeCustomer  
6 ActivityTaskStarted  
7 ActivityTaskCompleted  
8 WorkflowTaskScheduled
```

This asynchronous handoff yields two strict operational realities. First, activity invocations are structurally at-least-once with respect to the real world. Network partitions, worker crashes, or temporary service outages will trigger activity retries, causing the underlying code to execute multiple times.

Second, returning from an activity represents an observation boundary. You must treat the returned result as potentially originating from a prior, abandoned attempt. If an activity executes successfully but the network drops the acknowledgment before it reaches the orchestrator, the platform will retry the activity. The external system must deduplicate the request and return the exact same correlation data. The work-

flow must be prepared to receive this correlated response and trust it as the final observed state.

Designing Orchestration-Safe APIs

Treat the boundary between workflows and activities as a formal API design problem rather than a framework limitation. When designing this boundary, you face an immediate decision regarding granularity: coarse-grained versus fine-grained activities.

A coarse-grained activity encapsulates multiple sequential steps into a single function, such as fetching a user record, validating a balance, and writing an audit log. This approach reduces the total number of events recorded in the history, lowering storage overhead and minimizing scheduling latency. However, it forces the activity code to manage its own internal partial failures. If the audit log write fails, the activity must either roll back the balance update itself or risk leaving the system in an inconsistent state when the orchestrator retries the entire block of code.

Fine-grained activities map one distinct I/O operation to one activity. This delegates all retry and compensation logic back to the workflow, maintaining the orchestrator as the single source of truth for the execution state. While this increases event history size, it cleanly isolates failures. In practice, align your activity boundaries with external transactional boundaries. Group operations that succeed or fail together atomically into a single coarse activity. Separate operations that cross system boundaries, require different timeout constraints, or demand independent retry policies into fine-grained activities.

Request and Response Shapes

Avoid primitive arguments and unstructured maps at the activity boundary. Define explicit request and response objects for every activity. This is critical because the workflow must pass deterministic

context to the non-deterministic activity to ensure safe retries.

The request object must carry correlation IDs and idempotency keys generated by the workflow. Because the workflow environment provides deterministic random generation (e.g., `Workflow.randomUUID()`), you must generate the idempotency key in the workflow and pass it into the activity as a payload.

```
public class ChargeRequest {
    public final String customerId;
    public final long amountCents;
    public final String idempotencyKey;

    public ChargeRequest(String customerId, long amountCents, String
        idempotencyKey) {
        this.customerId = customerId;
        this.amountCents = amountCents;
        this.idempotencyKey = idempotencyKey;
    }
}

public class ChargeResponse {
    public final String transactionId;
    public final String status;

    public ChargeResponse(String transactionId, String status) {
        this.transactionId = transactionId;
        this.status = status;
    }
}
```

By generating the `idempotencyKey` within the workflow, you guarantee that every retry of the activity attempt uses the exact same key. The activity extracts this key and passes it to the external payment gateway, which uses it to silently deduplicate the repeated requests.

State Versus References

When crossing the determinism boundary, you must serialize data into the activity request payload. Avoid passing large datasets directly if the data is already available in an external data store. Instead, pass stable references.

For example, instead of passing a 10-megabyte document directly into the activity arguments, pass an object storage URI or a database primary key. The activity then fetches the payload directly from the external source. This prevents the event history from hitting payload size limits and keeps the orchestrator's underlying database lightweight.

However, you must guarantee that the referenced data is logically immutable for the duration of the workflow. If the activity fetches a database row that has been updated by another process since the workflow originally started, the activity will operate on a different state than the workflow expected. When passing references across the boundary, enforce versioning or strict immutability on the external data.

Preserving Context for Compensation

Activity results must preserve sufficient context to enable subsequent workflow decisions, particularly compensations. Returning a generic boolean success flag or a void response discards critical execution state. When an activity modifies an external system, the response object must contain the external identifier of the mutated resource, the timestamp of the operation, and any provider-specific status codes.

Consider a workflow that provisions cloud infrastructure. If the creation activity returns only `true`, the workflow has no way to identify the specific virtual machine if a later step fails and triggers a rollback. The activity must return the specific instance ID generated by the cloud provider. By capturing this metadata explicitly at the boundary, you decouple the execution of the side effect from the logic required to undo it, retaining all necessary state securely within the workflow history.

Sandboxing Variances Across SDKs

The enforcement mechanism for the determinism boundary varies significantly depending on your runtime environment. You must understand how your chosen SDK isolates workflow code, as this dictates the

level of testing discipline required from your engineering team.

The TypeScript SDK provides strict runtime sandboxing by executing workflow code inside a dedicated V8 isolate. It proactively intercepts and blocks non-deterministic calls. If you attempt to invoke `Math.random()` or `Date.now()` within a TypeScript workflow, the sandbox intercepts the call and injects a deterministically seeded value. If the operation involves restricted I/O modules like `fs` or `http`, it throws a terminal error before the code executes.

The Python SDK utilizes an Abstract Syntax Tree (AST) analyzer and trace hooks to restrict unsafe module imports and intercept thread-spawning calls. It aggressively prevents workflows from interacting with the operating system directly, catching violations at startup or execution time.

Conversely, the Java and Go SDKs rely heavily on developer discipline. While the Java SDK intercepts time calls like `Workflow.currentTimeMillis()` by returning the timestamp of the orchestrator's most recent `WorkflowTaskStarted` event, it does not prevent a standard `java.net.HttpURLConnection` from compiling or executing. The Go SDK employs linters to catch goroutine leaks, but a standard `http.Get` call inside a workflow will execute cleanly on its first run. In these languages, the failure only becomes apparent during a replay, when the external state has changed and the workflow path diverges. You must verify the specific sandboxing guarantees of your target SDK version and implement deterministic testing frameworks to catch violations before deployment.

Anti-Patterns in the Workflow Boundary

Several common engineering practices fundamentally violate the determinism boundary and manifest as intermittent workflow corruption.

The most frequent anti-pattern is embedding side effects inside logging

statements or metrics emitters. Because workflow code re-executes during replay, any network-bound logger placed inside the workflow core will fire multiple times for the same logical event. If you use a standard logging framework that flushes directly to an external aggregation service, a single worker crash might cause the workflow to recover and emit duplicate log entries. You must use the orchestration-safe logging utilities provided by the SDK, which check the orchestrator's replay flag and automatically suppress output during recovery phases.

Another critical anti-pattern is utilizing in-memory caches that observe external state. Developers frequently introduce static dictionaries or time-to-live (TTL) caches into workflow code to optimize repeated lookups. If a workflow reads from a local cache that is asynchronously refreshed by another thread, or if cache keys expire based on local wall-clock time, the read operation becomes non-deterministic. A workflow might make a decision based on a cached value during its initial run, crash, and then read a different value upon replay after the cache has expired. The workflow will throw a non-determinism error and stall indefinitely.

All cache lookups must be completely pushed behind the activity boundary. If you need to observe external state efficiently, schedule a fine-grained activity to fetch it. Once the activity returns the value, it is durably recorded in the event history. Subsequent replays will deterministically read the cached value from the history, bypassing the external system entirely and ensuring perfect execution consistency.

3.2. Engineering Activity Timeouts and Retries as a Reliability Contract

Timeout and retry configurations dictate how an orchestration engine behaves when external dependencies degrade, networks partition, or

worker nodes fail. Leaving these values to broad defaults or treating them as generic safety nets masks fundamental architectural decisions. Every timeout configures a failure detection window. Every retry policy shapes the load applied to downstream systems during an outage. Together, they form a strict reliability contract between your orchestration logic and the external world.

To establish this contract, begin by defining explicit bounds for your activities. The following example demonstrates configuring a comprehensive timeout and retry policy for an RPC call using the Temporal Go SDK:

```
activityOptions := workflow.ActivityOptions{
    // Maximum time the task can sit in the queue waiting for a
    // worker
    ScheduleToStartTimeout: 5 * time.Minute,

    // Maximum execution time for a single attempt
    StartToCloseTimeout:    10 * time.Second,

    // Total time budget spanning all attempts, backoffs, and queue
    // wait times
    ScheduleToCloseTimeout: 1 * time.Hour,

    // Maximum time between heartbeat pings (0 disables heartbeating)
    HeartbeatTimeout:      0,

    RetryPolicy: &temporal.RetryPolicy{
        InitialInterval:    1 * time.Second,
        BackoffCoefficient: 2.0,
        MaximumInterval:    30 * time.Second,
        MaximumAttempts:    0, // Unlimited attempts, constrained by
        // ScheduleToCloseTimeout
        NonRetryableErrorTypes: []string{
            "InvalidRequestFormat",
            "InsufficientFundsError",
        },
    },
}

ctx = workflow.WithActivityOptions(ctx, activityOptions)
err := workflow.ExecuteActivity(ctx, ProcessPayment, req).Get(ctx, &
    res)
```

This configuration is not an arbitrary collection of limits; it explicitly

models the expected behavior of the `ProcessPayment` dependency, the acceptable queue latency, and the upstream business deadline.

3.2.1. The Timeout Taxonomy and Failure Detection

Each timeout category protects a different segment of the activity life-cycle. Understanding what each timeout bounds is critical for engineering an effective reliability contract. Choosing inappropriate values leads directly to degraded performance or silent system failures.

Schedule-to-Start Timeout This timeout bounds worker availability and queue latency. It measures the time from when the orchestrator places the activity task into a task queue until a worker pulls it. A spike in Schedule-to-Start latency indicates worker exhaustion, a downed worker fleet, or an influx of tasks exceeding processing capacity.

Set this timeout based on the upstream caller's tolerance for staleness. If an activity fulfills a synchronous user request, the user will likely abandon the session after 30 seconds. Setting a Schedule-to-Start timeout of 10 minutes in this scenario forces workers to waste compute cycles processing requests that the upstream client has already discarded. Conversely, for asynchronous background processing, Schedule-to-Start timeouts can safely span hours or days to smooth out traffic spikes without dropping work.

Start-to-Close Timeout The Start-to-Close timeout is the most critical configuration in your activity design. It bounds a single execution attempt. Crucially, this timeout acts as the primary failure detection mechanism for infrastructure crashes. If a worker host loses power, experiences a kernel panic, or runs out of memory, it cannot

notify the orchestrator of the failure. The orchestrator only detects the failure when the Start-to-Close timeout expires.

If you set this timeout to 1 hour for an API call that typically takes 200 milliseconds, a worker crash will result in the orchestrator waiting a full hour before scheduling a retry. To minimize recovery latency, set the Start-to-Close timeout to the strict P99 latency of the external dependency, plus a strict margin for network overhead and framework serialization.

Heartbeat Timeout When an activity genuinely requires a long execution time—such as a database index rebuild, video transcoding, or a bulk file download—setting a tight Start-to-Close timeout is impossible. This creates a conflict: a long Start-to-Close timeout severely delays failure detection, but a short one artificially kills legitimate long-running work.

Heartbeat timeouts resolve this conflict. By configuring a Heartbeat timeout of 30 seconds and a Start-to-Close timeout of 2 hours, you require the activity worker to explicitly report progress to the orchestrator at periodic intervals. If a worker crashes, the heartbeat ping ceases, the orchestrator detects the failure within 30 seconds, and the activity is immediately retried on a healthy node. Heartbeats also serve as a checkpointing mechanism; you can attach a progress payload to the heartbeat, allowing the next attempt to resume exactly where the failed attempt left off.

Schedule-to-Close Timeout This defines the absolute end-to-end boundary for the activity, encompassing the initial queue wait, execution time, all retry backoff intervals, and all subsequent execution attempts. It represents the ultimate business deadline. When this timeout triggers, the orchestrator abandons the activity and returns a timeout error to the workflow. Model this value on the maximum time bud-

get allocated by the overarching business process, keeping in mind that exceeding it means failing the parent orchestration flow.

3.2.2. Computing Maximum Wall-Clock Exposure

Retries and timeouts function together as a unified system. Treating them as independent settings leads to operational surprises where activities retry endlessly or terminate prematurely. You must calculate the maximum wall-clock exposure of your retry policy to understand the worst-case scenario for your system's performance.

When a transient failure occurs, the orchestrator calculates the delay before the next attempt using the retry policy. The backoff interval generally follows an exponential curve:

$$\text{Interval} = \min (\text{InitialInterval} \times \text{BackoffCoefficient}^{\text{Attempt}-1}, \text{MaximumInterval})$$

Assume an activity has a `StartToCloseTimeout` of 10 seconds, an `InitialInterval` of 1 second, a `BackoffCoefficient` of 2.0, a `MaximumInterval` of 30 seconds, and `MaximumAttempts` of 6.

To determine the absolute worst-case wall-clock duration before the workflow registers a final failure, compute the sum of the execution timeouts and the backoff intervals for each attempt:

1. **Attempt 1:** 10s execution. Fails. (Total: 10s)
2. **Backoff 1:** 1s wait. (Total: 11s)
3. **Attempt 2:** 10s execution. Fails. (Total: 21s)
4. **Backoff 2:** 2s wait. (Total: 23s)
5. **Attempt 3:** 10s execution. Fails. (Total: 33s)
6. **Backoff 3:** 4s wait. (Total: 37s)

7. **Attempt 4:** 10s execution. Fails. (Total: 47s)
8. **Backoff 4:** 8s wait. (Total: 55s)
9. **Attempt 5:** 10s execution. Fails. (Total: 65s)
10. **Backoff 5:** 16s wait. (Total: 81s)
11. **Attempt 6:** 10s execution. Fails. (Total: 91s)

The absolute worst-case exposure is 91 seconds. If you configure a `ScheduleToCloseTimeout` of 60 seconds, Attempt 5 and Attempt 6 will never execute; the orchestrator will terminate the activity after 60 seconds regardless of the remaining attempts allocated in the retry policy. Always ensure your `ScheduleToCloseTimeout` is large enough to accommodate the desired number of retry cycles, including the maximum execution time of each attempt.

Orchestration frameworks automatically apply randomization, or jitter, to the calculated backoff intervals—typically a 20% variance. This means a theoretical 10-second interval will resolve to a random delay between 8 and 12 seconds. Jitter prevents synchronized “retry storms”. If a downstream service drops offline, hundreds of workflows will fail simultaneously. Without jitter, those workflows would awake at the exact same millisecond for their next attempts, slam the dependency, and instantly overwhelm it again. Jitter distributes the retry load across the time window, giving the recovering downstream service a smoother traffic gradient.

3.2.3. Designing for Interruption and Probabilistic Availability

Retries act as a probabilistic availability tool. If a downstream service has an availability of 99% (roughly 14 minutes of downtime per day),

executing a single attempt yields a 99% success rate. If failures are uncorrelated, retrying once pushes the theoretical success rate to 99.99%. However, in distributed systems, failures are rarely uncorrelated; infrastructure outages span minutes. Exponential backoff stretches the retry attempts across the outage window, maximizing the probability that at least one attempt lands after the dependency has recovered.

This mechanism fundamentally assumes the activity can be safely interrupted and re-executed. Worker restarts and rolling deployments introduce deliberate interruptions into this flow. When you deploy new worker code, the orchestrator gracefully stops existing worker processes. In-flight activities receive a cancellation signal.

To honor this interruption contract safely, your activity implementation must listen for context cancellation. In Go, this is achieved by monitoring the context's `Done()` channel:

```
func ProcessPayment(ctx context.Context, req PaymentRequest) (
    PaymentResponse, error) {
    resultChan := make(chan PaymentResponse, 1)
    errChan := make(chan error, 1)

    go func() {
        res, err := callExternalAPI(req)
        if err != nil {
            errChan <- err
            return
        }
        resultChan <- res
    }()

    select {
    case <-ctx.Done():
        // Worker is shutting down or activity timed out.
        // Clean up resources and return the cancellation error.
        return PaymentResponse{}, ctx.Err()
    case err := <-errChan:
        return PaymentResponse{}, err
    case res := <-resultChan:
        return res, nil
    }
}
```

If an activity catches this cancellation and exits cleanly, the orchestrator immediately hands the retry to a newly deployed worker. If the activity ignores the context, it becomes a zombie thread, and the orchestrator is forced to wait for the `StartToCloseTimeout` to forcibly terminate the connection, unnecessarily delaying recovery.

Because interruptions and transient network faults trigger retries, the external dependency may still be processing the first attempt when the second attempt arrives. The retry policy is not just a configuration for your workflow; it is a contract with the downstream system. If the downstream API lacks idempotency or request deduplication, aggressive retries will amplify duplicate side effects. High maximum attempt counts demand robust idempotency strategies.

Furthermore, differentiate between infrastructure failures and semantic failures. Retrying an HTTP 503 Service Unavailable error is necessary. Retrying an HTTP 400 Bad Request or a business-level authentication failure is actively harmful. Use the `NonRetryableErrorTypes` configuration to explicitly exempt semantic failures from the retry loop. When the activity worker encounters one of these specified errors, it surfaces the error immediately to the workflow, terminating the activity contract and allowing the workflow to execute permanent compensating logic.

3.2.4. Operational Runbook: Diagnosing Timeout Spikes

When operational alerts fire for elevated activity timeout rates, the immediate response must follow a structured diagnostic path. Timeouts are symptoms of underlying resource constraints, not root causes. Jumping straight to increasing timeout values typically exacerbates the underlying instability. Use the following sequence to isolate the failure domain.

Step 1: Inspect Execution History and Identify the Timeout

Type Use your orchestrator’s CLI or dashboard to inspect a failing workflow execution. Extract the event history to determine exactly which timeout was breached.

```
$ temporal workflow show --workflow-id payment-processing-123
...
3 ActivityTaskScheduled {ActivityId: 1, ActivityType: ProcessPayment}
4 ActivityTaskStarted   {Attempt: 1}
5 ActivityTaskTimedOut  {FailureType: StartToCloseTimeout}
6 ActivityTaskScheduled {ActivityId: 1, ActivityType: ProcessPayment}
...
```

If `ScheduleToStart` timeouts are spiking, the problem lies within your worker fleet, not the external dependency. This indicates workers are saturated, scaling policies have failed to provision enough nodes, or network routing from the workers to the orchestrator is broken. Mitigation involves horizontally scaling the worker deployment or shedding load upstream.

If `StartToClose` timeouts are spiking, workers are successfully acquiring tasks but failing to complete them. Proceed to Step 2.

Step 2: Inspect Worker Health and Network Saturation

A spiking `StartToClose` timeout often implicates the downstream dependency, but it can also indicate worker-side resource exhaustion. Check the CPU, memory, and file descriptor usage on the worker nodes. If workers are starved for CPU, operating system thread scheduling delays will cause local network requests to time out before they even reach the network interface. Similarly, inspect SNAT port exhaustion or DNS resolution latencies on the worker hosts. If infrastructure metrics are healthy, the bottleneck is external.

Step 3: Correlate with External Dependency Latency

Analyze the P90 and P99 response times of the downstream service.

If the downstream service latency has exceeded your configured `StartToClose` timeout, every attempt will forcefully time out locally, even if the external service eventually processes the request successfully.

This scenario creates a dangerous failure mode: the external service is slow but succeeding, while your workers time out, assume failure, and issue a retry. This retry amplifies the load on an already degraded dependency, accelerating its collapse.

Step 4: Execute Mitigations Depending on the root cause discovered in the previous steps, execute one of the following mitigations:

- **Dependency Overload:** If the downstream system is saturated, adding more workers or increasing retry aggressiveness will worsen the outage. Mitigate by temporarily reducing the task queue processing rate-using concurrent execution limits within your worker configuration-or by pausing the workflow generation upstream.
- **Mismatched Contract:** If the external system's baseline latency has permanently shifted (for instance, due to significant dataset growth), the `StartToClose` timeout contract is fundamentally broken. You must deploy an updated configuration that increases the `StartToClose` limit to accommodate the new P99 latency.
- **Global Budget Exhaustion:** If attempts are exhausting the `ScheduleToClose` global budget, verify whether the external service is returning retryable errors or just dropping connections. If it is returning structured errors, consider shifting traffic to a fallback provider using workflow-level orchestration rather than endlessly spinning in a local activity retry loop.

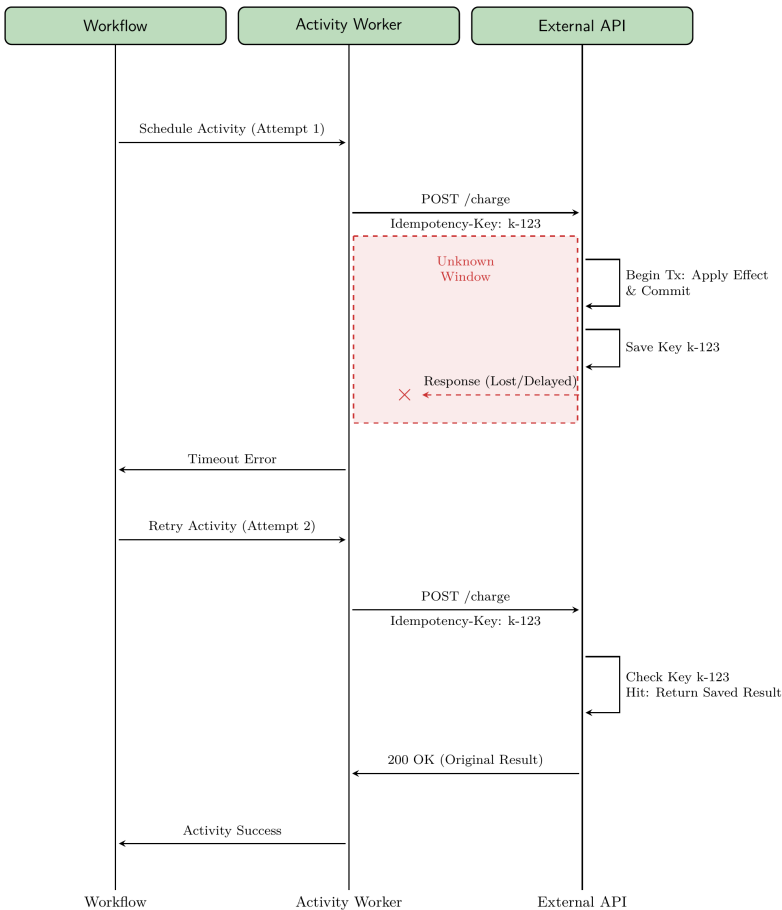
Treating timeouts and retries as a formalized contract forces you to define exactly how your system degrades under stress. By meticulously aligning timeout boundaries with dependency behavior and defining explicit rules for retries, you isolate failures to the smallest possible domain and prevent transient disruptions from collapsing the overarching orchestration pipeline.

3.3. Implementing Idempotency for Activities Across Databases, RPCs, and Third-Party APIs

Because activity invocations execute at-least-once with respect to the real world, any network partition, worker crash, or upstream orchestration retry can cause the activity code to re-execute. Consequently, every effectful activity must either be strictly idempotent, or be paired with a compensating strategy and an explicit model for resolving ambiguity.

When an activity attempts to apply a side effect over a network, the outcome falls into exactly one of three states. First, *Success (Confirmed)* indicates the external system processed the request and the activity successfully received the acknowledgement. Second, *Failure (Confirmed)* indicates the target system actively rejected the request—such as a schema validation failure or a constraint violation—guaranteeing the effect was not applied. Third, the *Unknown* state occurs when the request entered the network, but the worker failed to receive an acknowledgement. Network timeouts, target service crashes during processing, and worker node failures all fall into this window. In the unknown state, the effect may or may not have occurred, and a retry is structurally indistinguishable from a duplicate request unless you provide the external system with a mechanism to detect it.

3.3. IMPLEMENTING IDEMPOTENCY FOR ACTIVITIES ACROSS DATABASES, RPCS, AND THIRD-PARTY APIS



Implementing Idempotency Keys

When interacting with modern third-party APIs (such as payment gateways or cloud infrastructure providers) or well-designed internal RPCs, idempotency keys are the standard deduplication mechanism. An idempotency key is a unique string submitted alongside the request payload. The receiving system binds the key to the outcome of the op-

eration. If a subsequent request arrives with the exact same key, the server bypasses the operational logic and returns the cached result of the original execution.

To use idempotency keys safely, the key must be deterministic and stable across retries of the *same logical activity attempt*, but globally unique across different workflows or different activities within the same workflow. Orchestration SDKs expose execution context specifically for this purpose. Combining the workflow execution ID with the activity ID provides a highly stable key.

```
import requests
from temporalio import activity

@activity.defn
def charge_customer(payload: dict) -> dict:
    # Extract stable deterministic context provided by the worker
    info = activity.info()

    # Construct a key that remains identical across retries
    # of this specific activity execution.
    idem_key = f"{info.workflow_run_id}-{info.activity_id}"

    headers = {
        "Authorization": "Bearer sk_test...",
        "Idempotency-Key": idem_key,
        "Content-Type": "application/json"
    }

    # The timeout here governs the HTTP request, independent
    # of the activity's start-to-close timeout.
    response = requests.post(
        "https://api.external-provider.com/v1/charges",
        json=payload,
        headers=headers,
        timeout=10.0
    )

    response.raise_for_status()
    return response.json()
```

If the HTTP request times out and the orchestration framework retries the activity, the exact same Idempotency-Key is generated. The up-

stream provider identifies the duplicate, halts any redundant processing, and safely returns the existing record.

Building Internal Deduplication Stores

When you control the downstream database but the operation is not naturally idempotent (such as debiting an account balance or appending an event), you must build deduplication mechanics yourself. Natural idempotency applies only to absolute state transitions, such as `UPDATE status = 'ACTIVE'`. Relative transitions, such as `UPDATE balance = balance - 100`, strictly require deduplication.

The most robust approach is a dedicated deduplication table in the same database where the business entities reside. This table typically consists of `idempotency_key` (primary key), `response_payload`, and `created_at`.

Execute the business logic and the insertion into the deduplication table within a single, atomic database transaction. If the transaction violates the primary key constraint on the deduplication table, the request is a retry of a previously successful operation. You then roll back the duplicate transaction and retrieve the original `response_payload` to return to the caller.

```
CREATE TABLE idempotency_store (  
    idempotency_key VARCHAR(255) PRIMARY KEY,  
    status VARCHAR(50) NOT NULL,  
    response_payload JSONB,  
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP  
);
```

When using this pattern, configure an aggressive TTL (Time-To-Live) or a background cleanup job to purge keys older than the maximum possible retry window of the calling workflows. If your workflows max out at a 30-day run duration, purging keys after 45 days safely minimizes storage overhead without risking duplicate execution.

Conditional Writes and Compare-and-Swap

For high-throughput systems where maintaining a separate deduplication table creates unacceptable contention, conditional writes (often implemented as Compare-and-Swap or CAS) offer a lightweight alternative. CAS relies on versioning the rows in your primary tables.

When an activity reads the current state, it retrieves the row's `version` number. The subsequent write operation includes this version in its `WHERE` clause, incrementing it upon success:

```
UPDATE accounts
SET balance = balance - 100, version = version + 1
WHERE id = 42 AND version = 7;
```

If the database returns 0 rows affected, the activity knows the state has changed since the read. In a retry scenario, the activity reads the state again. If it observes that the balance was already reduced (detectable if you recorded the expected transition or stored a transaction ID alongside the balance), it returns success. If the state indicates a concurrent modification by an entirely different process, the activity can fail safely, allowing the workflow to decide how to handle the conflict.

The Transactional Outbox Pattern

One of the most persistent sources of duplicated side effects is the “dual-write” problem: updating a database record and simultaneously publishing a message to an event broker (like Kafka or RabbitMQ) within the same activity. Because these are two distinct systems, a distributed transaction is usually impractical. If the database commits but the Kafka publish times out, a retry will apply the database update again.

The Transactional Outbox pattern eliminates the dual-write entirely by shifting the publish intent into the database itself. You create an outbox table in the same database as your business entities.

Inside the activity, open a single database transaction. Apply the busi-

ness state change and insert the corresponding domain event into the outbox table. Commit the transaction. Because both operations occur in a single local transaction, they are strictly atomic.

A separate, independent relay mechanism—most commonly a Change Data Capture (CDC) tool like Debezium—monitors the database’s write-ahead log (WAL) and reliably pushes the outbox events to the message broker.

By pushing the dual-write into a durable log, the activity’s responsibilities contract. The activity no longer worries about Kafka timeouts; it simply writes to the database idempotently. The CDC system provides at-least-once delivery to the message broker, pushing the requirement for deduplication to the downstream consumer (often using the Inbox pattern, where the consumer tracks processed event IDs).

Handling Non-Idempotent Legacy APIs and Ambiguity

You will inevitably integrate with legacy systems or external APIs that offer neither idempotency keys nor transactional guarantees. When you invoke a non-idempotent legacy API, such as a legacy FTP file drop or an older SOAP billing service, and the request falls into the Unknown state due to a timeout, you cannot blindly retry.

Instead, you must design for partial-idempotency through reconciliation. When the external system does not provide deduplication, the workflow must actively verify the external state before attempting the operation again.

This introduces a read-before-write reconciliation loop. If an activity fails with a timeout, the workflow routes execution to a separate verification activity. The verification activity queries the external system using deterministic parameters (e.g., querying the billing service for charges applied to a specific Customer ID within the last five minutes matching the exact amount).

If the verification activity finds the target state already applied, it re-constructs the success payload and returns it to the workflow, allowing execution to proceed. If it explicitly confirms the state is missing, the workflow can safely issue a retry.

To make this feasible, activities must pre-allocate identifiers whenever possible. If the legacy system allows client-generated IDs, generate a UUID in the workflow and pass it into the activity. If the activity times out, the verification step searches for that exact UUID.

When external APIs strictly generate their own IDs and return them only in the response payload, the Unknown state creates deep ambiguity: the resource might exist, but you do not know its identifier. In these cases, your activity must capture any available correlation metrics-timestamps, payload hashes, or associated customer IDs-and return them as part of an enriched error object. The workflow can then use this metadata to either drive an automated compensation activity (like a bulk cancellation query) or park the workflow and signal an operator for manual resolution. Designing activities to capture and surface this metadata ensures that when idempotency fails, the resulting ambiguity is explicitly managed rather than silently ignored.

3.4. Orchestrating Workflow-Level Backoff, Hedged Requests, and Fallback Providers

Activity retries provide a robust default mechanism for handling transient faults, but they are inherently generic. Standard retry policies blindly execute the same operation with standard exponential backoff until a timeout is reached. When your recovery strategy depends on business context, specific error classes, time-of-day constraints, customer tiers, or the observed health of a dependency, you must elevate the retry logic out of the activity framework and into the workflow.

By orchestrating retries explicitly, you gain the ability to change providers, adjust parameters dynamically, implement circuit-breaker delays, and execute hedged requests. The workflow becomes the active decision-maker rather than a passive observer waiting for an activity to either succeed or exhaust its retries.

Workflow-Driven Fallback Example

Consider a payment processing scenario where you prefer a primary provider for its lower fees, but require a fallback to a secondary provider if the primary experiences an outage. Relying solely on activity retries would stall the workflow until the primary recovers or the activity times out. Instead, you can configure the primary activity with a strict, short retry policy and orchestrate the fallback directly in the workflow.

```
func ProcessPaymentWorkflow(ctx workflow.Context, req PaymentRequest)
    (Result, error) {
    options := workflow.ActivityOptions{
        StartToCloseTimeout: 5 * time.Second,
        RetryPolicy: &temporal.RetryPolicy{
            MaximumAttempts: 3, // Fail fast to trigger fallback
        },
    }
    ctx = workflow.WithActivityOptions(ctx, options)

    var result Result
    err := workflow.ExecuteActivity(ctx, CallPrimaryProvider, req).
        Get(ctx, &result)

    if err != nil {
        // Evaluate error type to decide on fallback
        var appErr *temporal.ApplicationError
        if errors.As(err, &appErr) && appErr.Type() == "
            ProviderUnavailable" {
            workflow.GetLogger(ctx).Info("Primary provider failed,
                switching to secondary")

            // Secondary provider might have a different timeout and
            retry profile
            fallbackOptions := workflow.ActivityOptions{
                StartToCloseTimeout: 15 * time.Second,
                RetryPolicy: &temporal.RetryPolicy{
```

```
                InitialInterval: 2 * time.Second,
                MaximumAttempts: 5,
            },
        },
        fallbackCtx := workflow.WithActivityOptions(ctx,
        fallbackOptions)
        err = workflow.ExecuteActivity(fallbackCtx,
        CallSecondaryProvider, req).Get(fallbackCtx, &result)
    }
}

if err != nil {
    return Result{}, err
}

return result, nil
}
```

This structure transforms a rigid failure into a dynamic recovery path. You retain durable state tracking of which provider succeeded and which failed, enabling accurate auditing and precise compensation paths later.

The Decision Checklist for Workflow Orchestration

Before implementing custom retry logic in a workflow, evaluate whether standard activity retries suffice. Move orchestration into the workflow only when you answer “yes” to any of the following questions:

- *Do you need to change request parameters based on attempt number or error type?* If a service rejects a request due to rate limiting, you might need to parse the `Retry-After` header and dynamically wait, or reduce the payload size on the next attempt.
- *Do you need to switch providers?* Transitioning from a degraded primary service to a secondary service requires workflow-level awareness.
- *Do you need to apply circuit breaking or global rate limiting?* Pausing executions across multiple steps or shifting to a de-

graded mode requires coordination that an isolated activity cannot perform.

- *Do you need to coordinate multiple activities to recover from a compound failure?* If resetting a connection requires calling a separate authentication or token-refresh activity before retrying the original call, the workflow must orchestrate that sequence.

Implementing Durable Backoff and Retry Windows

When you implement custom retries, you must avoid blocking worker threads with synchronous sleeps. Never embed dynamic sleep or back-off logic inside activity code. Doing so wastes worker capacity and bypasses the workflow engine's ability to durably persist the sleep state. If a worker crashes during an in-activity sleep, the timer resets.

Instead, execute a single activity attempt (disabling standard retries) and manage the backoff loop using workflow timers.

```
func DynamicBackoffWorkflow(ctx workflow.Context, req DataRequest)
    error {
        maxAttempts := 10
        baseWait := 1 * time.Second

        for attempt := 1; attempt <= maxAttempts; attempt++ {
            // Single-attempt activity
            opts := workflow.ActivityOptions{
                StartToCloseTimeout: 5 * time.Second,
                RetryPolicy: &temporal.RetryPolicy{ MaximumAttempts: 1 },
            }
            ctxOpts := workflow.WithActivityOptions(ctx, opts)

            err := workflow.ExecuteActivity(ctxOpts, FetchData, req).Get(
                ctx, nil)
            if err == nil {
                return nil // Success
            }

            if attempt == maxAttempts {
                return err // Exhausted retries
            }

            // Calculate exponential backoff with jitter
```

```
        sleepDuration := calculateBackoffWithJitter(baseWait, attempt
    )

    // Durable workflow sleep
    workflow.Sleep(ctx, sleepDuration)
}
return nil
}
```

When calculating backoff intervals, always incorporate jitter. Without jitter, workflows that fail simultaneously due to a transient network blip will wake up and retry in lockstep, creating a “retry storm” that continually overwhelms the downstream dependency. Add randomness to the calculated interval, typically $\pm 20\%$ of the exponential delay, to spread the load.

In many operational scenarios, you want to bound retries not by an attempt count, but by a strict deadline—a “retry window.” If the workflow must return a response to a user within 30 seconds, looping for 10 attempts is irrelevant if 30 seconds have elapsed. Check the workflow time (using the determinism-safe time API provided by your SDK) and break the loop to trigger a fallback or return a degraded response once the deadline passes.

Hedged Requests for Tail Latency Reduction

In high-throughput systems, median latency is often stable, but p99 tail latency spikes due to localized network drops, JVM garbage collection pauses, or momentary database contention. A hedged request mitigates tail latency by issuing a secondary request to the same service (or a replica) after a short delay, taking the result of whichever request returns first.

Hedging differs from retrying. A retry waits for a definitive failure or timeout before trying again. A hedged request is dispatched while the first request is still in flight.

Hedging is only safe for read-only operations or strictly idempotent writes to systems that support request deduplication. If you hedge a non-idempotent write, you guarantee duplicate side effects. You must also pair hedging with explicit load-shedding guardrails. If a service is globally saturated, issuing a second request for every slow response will double the load on an already failing system, amplifying an outage.

To implement a hedged request, you start the primary activity asynchronously, wait for a short hedging delay, and if the primary has not completed, start the secondary activity. You then wait for the first successful completion and cancel the outstanding request.

```
public String fetchWithHedging(String input) {
    // 1. Start primary request asynchronously
    Promise<String> primary = Async.function(activities::fetchData,
        input);

    // 2. Wait for the hedging delay (e.g., 95th percentile latency)
    boolean primaryCompletedFast = Workflow.await(
        Duration.ofMillis(200), primary::isCompleted);

    if (primaryCompletedFast) {
        return primary.get(); // Completed before hedge delay
    }

    // 3. Primary is slow, launch hedge
    Promise<String> hedge = Async.function(activities::fetchData,
        input);

    // 4. Wait for either to complete successfully
    Promise<String> firstCompleted = Promise.anyOf(primary, hedge);

    // Handle errors and ensure the loser is canceled via context
    // propagation
    return firstCompleted.get();
}
```

This pattern requires robust cancellation propagation. When `Promise.anyOf` resolves, the workflow must signal the engine to cancel the still-running activity. The activity implementation must listen for the cancellation context and gracefully terminate its network connections to free resources on the external service.

Multi-Provider Fallbacks and Semantic Drift

When orchestration moves to a fallback provider, you are not simply retrying an identical operation. Secondary providers often exhibit different performance profiles, different data schemas, and different consistency guarantees.

Treat provider choice as a durable, auditable decision. When the workflow transitions to a fallback, it must record which provider was ultimately used. If the primary provider returns an ambiguity error (e.g., a timeout where the request might have been processed), you cannot blindly switch to the secondary provider without risking duplicate execution across two separate systems.

Before invoking the fallback, the workflow must first verify or compensate the primary provider's state. If the primary provider timed out on a charge authorization, the workflow should execute a cancellation activity against the primary provider to ensure the authorization is voided before charging the secondary provider.

Furthermore, configure distinct timeout and retry profiles for the secondary provider. A degraded mode service might be geographically distant, requiring longer timeouts, or it might be a lower-capacity backup that requires more aggressive rate limiting and longer backoff intervals to prevent immediate saturation.

Workflow-Level Circuit Breakers

Standard SDK-level circuit breakers exist in memory on individual worker instances. If a worker restarts, the circuit breaker resets. Furthermore, in a distributed worker pool, each worker maintains its own localized view of dependency health, which can lead to slow tripping and uneven load distribution.

You can orchestrate circuit-breaker semantics directly within the workflow. While global circuit breaker state (tracking error rates across

thousands of concurrent workflows) usually requires external storage or a dedicated signals/queries mechanism, individual long-running workflows can manage their own localized circuit breakers.

When an activity fails with an error indicating dependency overload (e.g., HTTP 429 Too Many Requests or HTTP 503 Service Unavailable), the workflow can transition into a “circuit open” state. Instead of tight-loop retrying, the workflow durably sleeps for a defined reset interval (e.g., 5 minutes) before attempting a “half-open” probe activity.

```
// Workflow fragment demonstrating localized circuit breaker
err := workflow.ExecuteActivity(ctx, CallExternalAPI).Get(ctx, nil)
if err != nil && isOverloadError(err) {
    workflow.GetLogger(ctx).Warn("Dependency overloaded, opening
        circuit")

    // Durable sleep pauses workflow execution without consuming
    // worker threads
    workflow.Sleep(ctx, 5 * time.Minute)

    // After sleep, attempt probe or transition to degraded mode
    err = workflow.ExecuteActivity(ctx, CallExternalAPI).Get(ctx, nil)
}
if err != nil {
    // Route to permanent fallback
    return executeFallback(ctx)
}
```

This approach prevents the workflow from continuously hammering a failing dependency. By shifting the delay out of the activity retry loop and into a durable workflow sleep, you free up the activity worker to process other tasks, isolating the resource impact of the downstream outage. The workflow persists the exact state of the failure, ensuring that operators can inspect the history, understand why the workflow is delayed, and intervene if necessary by sending a signal to force the circuit closed or redirect the execution path.

3.5. Choosing Between Retry, Compensate, and Human Escalation: A Failure Classification Playbook

Executing an activity across a network is fundamentally an exercise in navigating incomplete information. When an activity returns an error, you rarely know for certain that the underlying side effect failed; you only know that your observation of the side effect failed. Treating all failures as equivalent—typically by applying a generic retry policy until a timeout is reached—destroys the operational semantics of your system. It masks permanent configuration errors behind hours of backoff, amplifies load on struggling downstream dependencies, and turns ambiguous timeouts into dangerous data corruption risks.

To build reliable workflows, you must reframe failure as a routing problem. Every error returned by an activity provides a set of observable signals—error type, timeout class, attempt count, and partial response payloads. By inspecting these signals, you can route the failure to a durable remediation path: automated retry, graceful compensation, explicit reconciliation, or human escalation.

The following Go snippet demonstrates how to encode this classification directly into a workflow. Instead of allowing activity errors to implicitly fail the workflow, the code catches the error, inspects the failure class, and routes the execution to the correct remediation strategy.

```
func OrderFulfillmentWorkflow(ctx workflow.Context, req OrderRequest)
    (OrderResult, error) {
    var result OrderResult

    // Execute activity with a baseline retry policy for transient
    faults
    err := workflow.ExecuteActivity(ctx, ChargeCustomer, req).Get(ctx
    , &result)
    if err != nil {
        // 1. Inspect the error to determine the failure class
        var appErr *temporal.ApplicationError
```



```
        if errors.As(err, &appErr) {
            switch appErr.Type() {
                case "OverloadError":
                    // E.g., HTTP 429. Route to fallback or workflow-
                    level backoff.
                    return RouteToFallbackProvider(ctx, req)

                case "SemanticError":
                    // E.g., HTTP 400 or Insufficient Funds. Do not retry
                    .

                    // Trigger compensations for any previous steps.
                    err := RunCompensations(ctx, req.OrderID)
                    return OrderResult{}, fmt.Errorf("order rejected: %w
", err)

                case "AuthError":
                    // E.g., HTTP 401/403. Retrying is useless without
                    credential rotation.
                    return RouteToHumanEscalation(ctx, req, "Invalid
                    credentials")
                }
            }

            // 2. Handle ambiguous outcomes (Timeouts)
            var timeoutErr *temporal.TimeoutError
            if errors.As(err, &timeoutErr) {
                // The charge might have succeeded. We must verify before
                retrying or failing.
                status := ReconcilePaymentStatus(ctx, req.IdempotencyKey)
                if status == PaymentConfirmed {
                    return OrderResult{Status: "Success"}, nil
                } else if status == PaymentFailed {
                    // Now we know it failed cleanly; safe to compensate
                    or escalate
                    return RouteToHumanEscalation(ctx, req, "Charge
                    failed after timeout")
                }
            }

            // 3. Fallback for unclassified or exhausted transient errors
            return RouteToHumanEscalation(ctx, req, "Exhausted retries on
            unknown error")
        }

        return result, nil
    }
}
```

This code transforms raw exceptions into business decisions. To implement this pattern effectively, you must map the errors generated by your external dependencies into five distinct failure classes.

Transient Infrastructure Failures

Transient failures represent temporary environmental anomalies: DNS resolution drops, TCP connection resets, ephemeral network partitions, or unexpected worker pod terminations. These errors are strictly infrastructure-level and carry no semantic payload. Because they are stateless and temporary, the universally correct response is to retry the exact same request.

Activity-level retries are designed precisely for this class. You configure your retry policy to absorb these anomalies by specifying a relatively aggressive initial interval (e.g., 500ms) and relying on exponential back-off. If a transient error bubbles up to the workflow level, it means the retry policy has been exhausted, and the failure has escalated from a transient blip to a persistent outage. At that point, the workflow should route the failure to a fallback provider or human escalation.

Dependency Overload and Backpressure

Overload failures occur when the external dependency is functioning but lacks the capacity to process your request. Typical signals include HTTP 429 (Too Many Requests), HTTP 503 (Service Unavailable), database connection pool exhaustion, or explicit backpressure payloads.

Treating overload failures as generic transient failures is a severe anti-pattern. If you aggressively retry an HTTP 429, you contribute to a retry storm that further degrades the struggling dependency. When your activity encounters an overload signal, it should immediately fail the activity attempt and pass the overload context to the workflow. The workflow then orchestrates a delayed backoff, applies a circuit breaker

pattern, or hedges the request against a secondary provider. The remediation path must prioritize protecting the dependency over aggressively completing the workflow.

Semantic and Business Rule Failures

Semantic failures indicate that the request is structurally or logically invalid. These include HTTP 400 (Bad Request), database constraint violations, insufficient funds errors, or invalid state transitions. A semantic error guarantees that the external system rejected the side effect, and crucially, it guarantees that sending the identical request again will yield the exact same rejection.

Never apply automated retries to semantic failures. In Temporal and similar orchestration frameworks, semantic errors should be flagged as non-retryable in the activity retry policy. When the workflow catches a semantic error, it must abandon the current execution path. If previous activities in the workflow have already executed side effects (e.g., reserving inventory before attempting the failed payment), the workflow must transition to a compensation path to undo those effects.

Authorization and Contract Failures

Authorization failures (HTTP 401 Unauthorized, HTTP 403 Forbidden) and contract mismatch errors (unsupported API versions, missing mandatory fields) occur when your worker lacks the permissions or the correct schema to interact with the external system. Like semantic errors, these are permanent for the lifetime of the request.

However, unlike semantic errors which represent user-driven invalid state (like an empty bank account), contract failures represent a systemic configuration error. Your API token expired, your OAuth rotation failed, or the downstream provider deprecated an endpoint. Routing these to automated compensation is usually unsafe, as the compensation activities will likely rely on the same invalid credentials and fail

identically. The correct remediation path for contract failures is human escalation. The workflow should pause, trigger an operational alert, and wait for an operator to rotate the credentials or deploy a worker update before resuming.

Ambiguous and Unknown Outcomes

Ambiguity is the most dangerous failure class in distributed systems. An ambiguous outcome occurs when the activity worker loses communication with the external system after transmitting the request but before receiving the response. The primary signal for ambiguity is an activity timeout (specifically a Start-To-Close timeout) or a 504 Gateway Timeout.

When an activity times out, the outcome is a tri-state: Success, Failure, or Unknown. You cannot blindly retry, because if the original request succeeded, you will duplicate the side effect. You cannot blindly compensate, because if the original request failed, compensating will corrupt the system state in the opposite direction.

The mandatory remediation path for an ambiguous failure is explicit reconciliation. You must schedule a verification activity that queries the downstream system to determine the actual outcome of the original request.

3.5.1. Encoding Reconciliation and Compensation

Reconciliation activities depend entirely on the idempotency mechanisms established at the activity boundary. When a primary activity times out, the workflow invokes a corresponding `Verify` activity, passing the exact idempotency key or correlation ID used in the original request.

If the downstream system exposes an explicit read API, the verifica-

tion activity polls that endpoint. If the downstream system confirms the transaction succeeded, the workflow discards the timeout error and proceeds normally. If the downstream system confirms the transaction is unknown or failed, the workflow can safely issue a retry.

When a workflow encounters a verified semantic failure or an intentional abort condition, it must execute compensations. Compensations are activities designed to offset or reverse a previously completed side effect. To execute a compensation safely, the workflow must possess the exact identifiers generated by the original external system.

If your `CreateUser` activity returns a user ID, that ID must be durably recorded in the workflow state and explicitly passed to the `DeleteUser` compensation activity. Never attempt to compensate using only the original request parameters. If you ask a downstream system to “delete the user named Alice,” you risk deleting a different record created by a concurrent process. Always compensate by exact external identifier.

Furthermore, compensations are themselves activities that execute over unreliable networks. They can fail. If a compensation activity encounters a transient failure, it must retry. If a compensation activity encounters a permanent or semantic failure (e.g., the user record to be deleted is suddenly locked), you have entered a failure chain. The automated recovery process has broken down, and the workflow must transition immediately to human escalation.

3.5.2. Structuring Human Escalation

Human escalation is not merely firing an alert or writing to a dead-letter queue; it is a durable, first-class state within the workflow execution. When automation cannot safely proceed—because credentials have expired, compensations have failed, or reconciliation is impossible—the workflow must pause and wait for operator interven-

tion without losing its execution context.

To implement durable escalation, structure your workflow to block on an external signal.

```
func RouteToHumanEscalation(ctx workflow.Context, req OrderRequest,
    reason string) (OrderResult, error) {
    // 1. Emit metrics and trigger operational alerts (PagerDuty,
    // Slack, etc.)
    err := workflow.ExecuteActivity(ctx, AlertOperations,
        EscalationPayload{
            WorkflowID: workflow.GetInfo(ctx).WorkflowExecution.ID,
            Reason:    reason,
            Context:    req,
        }).Get(ctx, nil)

    if err != nil {
        workflow.GetLogger(ctx).Error("Failed to alert operations", "
            Error", err)
    }

    // 2. Block the workflow until an operator provides a resolution
    // signal
    var resolution OperatorCommand
    signalChan := workflow.GetSignalChannel(ctx, "operator-
        intervention")

    // Wait durably for the signal. This can pause for days or weeks
    // without consuming resources.
    signalChan.Receive(ctx, &resolution)

    // 3. Apply the operator's decision
    switch resolution.Action {
    case "RETRY":
        // Operator fixed the downstream issue; attempt the exact
        // same execution path
        return OrderFulfillmentWorkflow(ctx, req)
    case "OVERRIDE_AND_PROCEED":
        // Operator manually completed the step and provided the
        // necessary external ID
        return OrderResult{Status: "Success", ExternalID: resolution.
            OverrideData}, nil
    case "ABORT":
        // Operator demands a hard stop; run compensations
        RunCompensations(ctx, req.OrderID)
        return OrderResult{}, fmt.Errorf("workflow aborted by
            operator")
    default:

```

```
        return OrderResult{}, fmt.Errorf("unknown operator command")
    }
}
```

This pattern guarantees that operational intervention is auditable and synchronized with the workflow state machine. The operator is presented with a stable remediation command set. Because the workflow state is durable, the operator can take the time needed to investigate the downstream system, verify logs, or rotate credentials before sending the intervention signal.

Avoid the anti-pattern of escalating without preserving evidence. If you force an on-call engineer to guess what the workflow was attempting to do, you invite human error. Ensure that the alert payload includes the specific idempotency keys, the last known successful receipt, and the exact error classification that triggered the escalation.

The Failure Classification Matrix

To standardize failure handling across teams, encode the following decision matrix into your shared workflow libraries and operational runbooks. This matrix maps the observable failure classes to their strictly recommended remediation paths.

Failure Class	Primary Signal	Remediation Path
Transient	TCP Resets, DNS drops, 500 Internal Error	Automated Retry: Activity-level bounded exponential backoff.
Overload	HTTP 429, 503, Connection Pool Exhaustion	Backoff / Fallback: Workflow-level delayed retry, circuit breaking, or routing to secondary providers.
Semantic	HTTP 400, Constraint Violations, Validation Errors	Compensate / Fail: Do not retry. Abandon the execution path and reverse prior side effects.
Contract / Auth	HTTP 401, 403, Schema Mismatch	Human Escalation: Pause execution. Alert operators to rotate credentials or patch worker code.
Ambiguous	Activity Start-To-Close Timeout, HTTP 504	Reconcile: Schedule a verification activity using the original idempotency key before attempting retry or compensation.

Table 3.1: *Decision matrix mapping failure classes to their durable remediation paths.*

By formalizing these classifications, you decouple the orchestration of failure recovery from the execution of the side effects themselves. The activities remain focused purely on interacting with the external world, while the workflows retain absolute authority over how to handle the inevitable collapse of those interactions. Applying this playbook ensures that when failures occur, the system degrades predictably, preserves strict consistency, and provides operators with exact levers for safe intervention.

Chapter 4

Orchestration Patterns for Microservices

Microservices excel at local autonomy but struggle with end-to-end business processes: timeouts become contracts, retries become policies, and “eventual consistency” becomes a source of ambiguity. In this chapter you’ll model those realities directly in Temporal workflows—using durable waiting, controlled parallelism, child workflow boundaries, and sagas—so cross-service coordination becomes explicit code with explicit failure semantics.

4.1. Implementing Durable Timers, Deadlines, and Wait Conditions Without Resource Locking

Building microservices often requires orchestrating processes that span hours, days, or months. Traditional distributed systems

implement delays using leased database locks, delayed message queues, in-memory schedulers, or external cron jobs. Each of these mechanisms introduces operational fragility: in-memory schedulers fail during deployments, database polling wastes compute cycles, and distributed cron jobs inevitably drift from the actual state of the business process they intend to control.

Temporal solves this by treating time as a first-class, durable control flow construct. You can instruct a workflow to suspend execution for an explicit duration or until a specific condition is met, without holding OS threads, network connections, or infrastructure resources.

To ground this, examine how durable waiting is implemented in the Java SDK using the `Workflow.sleep` and `Workflow.await` primitives:

```
public class OrderFulfillmentWorkflowImpl implements
    OrderFulfillmentWorkflow {
    private boolean paymentReceived = false;
    private boolean cancellationRequested = false;

    @Override
    public void fulfillOrder(String orderId) {
        // 1. Simple durable sleep: suspend for a fixed time
        Workflow.sleep(Duration.ofHours(24));

        // 2. Deadline race: Wait for payment, cancellation, OR a 48-
        hour timeout
        boolean conditionMet = Workflow.await(Duration.ofDays(2),
            () -> paymentReceived || cancellationRequested);

        if (!conditionMet) {
            // The timer expired before the conditions were true
            escalateToHuman(orderId);
            return;
        }

        if (cancellationRequested) {
            processCancellation(orderId);
            return;
        }

        proceedWithShipping(orderId);
    }
}
```

```
@Override
public void signalPayment() {
    this.paymentReceived = true;
}

@Override
public void signalCancel() {
    this.cancellationRequested = true;
}
}
```

When `Workflow.sleep` or `Workflow.await` executes, the SDK does not block a thread. Instead, it emits a `TimerStarted` command to the Temporal Server and suspends the workflow execution. The Server records this command in the event history and persists the timer in its internal database. The worker process is now entirely free to execute other tasks. When the physical time elapses, the Server writes a `TimerFired` event to the history and queues a new workflow task. A worker picks up this task, replays the history to reconstruct the workflow state, and resumes execution exactly where it left off.

Execution Model Constraints for Time

Because workflows must be strictly deterministic (as detailed in Chapter 2), you cannot read the system wall-clock time directly (e.g., using `System.currentTimeMillis()` in Java or `time.Now()` in Go). If a workflow reads system time, its logic would change during a replay weeks later. Instead, SDKs provide deterministic time APIs, such as `Workflow.currentTimeMillis()` or `workflow.Now(ctx)`. These APIs return the timestamp of the workflow task that is currently being processed. This time is recorded in the event history and remains frozen during replays, guaranteeing that time-based conditionals branch exactly as they did during the original execution.

Deadline Races and Conditional Awaits

The most common orchestration requirement is not a blind sleep,

but a deadline race: waiting for an external condition while maintaining a strict timeout. The `Workflow.await` method (or `workflow.AwaitWithTimeout` in Go) implements this cleanly.

The condition provided to the `await` function is evaluated deterministically. It is checked immediately when the method is called, and then re-evaluated every time a new workflow task is processed—typically triggered by an incoming Signal or Update (covered in Chapter 5). If the condition evaluates to `true`, the workflow resumes immediately. If the timeout duration elapses first, the method returns `false` (or an error in Go), allowing you to handle the expiry explicitly.

This pattern is fundamentally different from a polling loop. The workflow does not wake up repeatedly to check the variables; it remains completely dormant until an external event or the timer triggers a state transition.

Multi-Stage SLAs and Escalations

Organizational SLOs (Service Level Objectives) and SLAs (Service Level Agreements) rarely map to a single binary timeout. A robust business process typically involves a grace period, escalating reminders, and a final hard deadline. Implementing this in a single workflow provides total visibility into the lifecycle of an SLA.

Design these time-based orchestration flows using explicit state transitions. Attach persistent state labels to the workflow so operators can query the exact stage of the process.

```
public void manageInvoiceSla(Invoice invoice) {
    this.slaStage = "GracePeriod";
    updateSearchAttributes("SlaStage", this.slaStage);

    // Stage 1: Grace Period
    boolean paid = Workflow.await(Duration.ofDays(14), () -> isPaid()
    );
    if (paid) return;

    // Stage 2: Escalating Reminders
```

```
this.slaStage = "Reminders";
updateSearchAttributes("SlaStage", this.slaStage);

for (int i = 0; i < 3; i++) {
    sendReminderActivity(invoice.getId(), i + 1);
    paid = Workflow.await(Duration.ofDays(3), () -> isPaid());
    if (paid) return;
}

// Stage 3: Final Deadline
this.slaStage = "Breached";
updateSearchAttributes("SlaStage", this.slaStage);
suspendAccountActivity(invoice.getAccountId());
}
```

When defining SLA thresholds, rely on business metrics rather than infrastructure defaults. Incident postmortems frequently reveal that “retry storms” occur when technical timeouts (e.g., 30-second network limits) are mistakenly used to govern business SLAs (e.g., waiting for an upstream vendor to process a batch). Use activities for short, technical retries, and use workflow timers for long-running, business-aligned deadlines.

Time-Window Correlation

Sometimes you need to collect multiple events over a fixed period before taking action. For example, a system might accept order modifications for 30 minutes after placement, batching them together to minimize downstream shipping recalculations.

Implement time-window correlation using a fixed timer combined with a collection state.

```
func BatchModificationsWorkflow(ctx workflow.Context, orderID string)
    error {
    var modifications []string

    // Listen for incoming modification signals
    signalChan := workflow.GetSignalChannel(ctx, "modify_order")

    // Define the correlation window
    windowTimer := workflow.NewTimer(ctx, 30*time.Minute)
```

```

selector := workflow.NewSelector(ctx)
selector.AddReceive(signalChan, func(c workflow.ReceiveChannel,
more bool) {
    var mod string
    c.Receive(ctx, &mod)
    modifications = append(modifications, mod)
})

// Process signals until the timer fires
for {
    selector.Select(ctx)
    if windowTimer.IsReady() {
        break // Window closed
    }
}

return ProcessBatchActivity(ctx, orderID, modifications)
}

```

This pattern guarantees that any events arriving exactly at the 30-minute mark are processed deterministically. Late-arriving signals (received after the timer fires and the workflow completes or transitions) are either rejected by the Temporal Server or can be handled by a subsequent workflow depending on your ID reuse policy.

Encoding Human-Facing Expectations

Orchestration often intersects with human schedules: “process on the next business day” or “resume after vendor maintenance”. Do not calculate complex calendar logic, holidays, or business days dynamically inside the workflow using external APIs mid-sleep. Doing so risks non-determinism and introduces hidden failure points.

Instead, calculate the exact target time (e.g., `Instant` or `time.Time`) at the moment the wait begins, either using a deterministic library within the workflow or by invoking a fast activity that queries a calendar service. Once the target absolute time is resolved, subtract `Workflow.currentTimeMillis()` to determine the exact `Duration` to sleep.

If the wait condition depends on a human action that is suddenly delayed—such as extending a deadline—expose a Signal handler that overrides the target time. The workflow can wake up, observe the new extended time, and go back to sleep.

Cancellation and Timer Interruption

When a workflow is cancelled, any active `Workflow.sleep` or timer will immediately abort, throwing a cancellation exception (e.g., `CanceledFailure` in Java, or returning a cancelled error in Go). This prevents cancelled workflows from lingering in a sleeping state.

However, if you need to perform cleanup activities (like sending a “cancellation confirmed” email) after a timer is interrupted, you must execute those activities in a detached cancellation scope. If you attempt to schedule an activity in the same cancelled scope, the SDK will refuse to execute it.

```
try {
    Workflow.sleep(Duration.ofDays(7));
} catch (CanceledFailure e) {
    // Timer interrupted by workflow cancellation
    CancellationScope detachedScope = Workflow.
        newDetachedCancellationScope();
    detachedScope.run(() -> {
        cleanupActivity.revertPendingChanges();
    });
    throw e; // Rethrow to allow the workflow to complete as
        cancelled
}
```

Operational Safety and History Management

Timers write events to the workflow history. A single sleep writes two events: `TimerStarted` and `TimerFired`. If you design a workflow that uses short sleep intervals (e.g., 5 seconds) in an unbounded loop, you will rapidly hit Temporal’s history size limits (typically 50,000 events or 50 MB), causing the workflow to terminate.

Timers should not be used for high-frequency polling. If you must poll

an external system every 5 seconds, encapsulate that loop inside an Activity, allowing the Activity to execute the polling locally and only return to the workflow when the condition is met or a meaningful time-out is reached. If long-running timer loops are structurally necessary, you must use Continue-As-New (detailed in Chapter 2) to periodically truncate the event history.

To make time-based workflows operable, surface the wait states using Search Attributes (detailed in Chapter 6). When a workflow suspends for 30 days, operations teams need to know *why* it is waiting and *when* it will resume. Update a WaitReason or SlaStage search attribute immediately before calling Workflow.sleep.

Operator Playbook: Managing Time-Based Workflows

Because time is modeled durably, operators have immense control over inflight waits without touching application databases. Equip your operations team with runbooks leveraging the Temporal CLI (temporal):

- **Identify stalled SLAs:** Find all workflows currently waiting in a specific SLA stage.

```
temporal workflow list \  
  --query "SlaStage = 'GracePeriod' AND ExecutionStatus = '  
    Running'"
```

- **Force-expire a wait condition:** Instead of waiting days for a timer, operators can send a signal to immediately bypass the wait condition, forcing the workflow down the expiry path for testing or administrative overrides.

```
temporal workflow signal \  
  --workflow-id "invoice-123" \  
  --name "force_expire"
```

(This requires your workflow code to explicitly listen for the force_expire signal within its Workflow.await condition.)

- **Audit timer history:** Inspect the event history to determine exactly when a wait started and when it is scheduled to fire.

```
temporal workflow show --workflow-id "invoice-123"
```

Operators can look for the `TimerStarted` event, which explicitly lists the `startToFireTimeout`.

Trade-offs and Anti-patterns

When designing time-based flow, consider the trade-off between responsiveness and history churn. Shorter timers allow the system to react faster to timeouts but increase the load on the Temporal cluster and worker replay cycles. Default to longer, business-meaningful timers at the workflow level, delegating rapid, transient retries to the Activity layer.

Avoid these critical anti-patterns:

- **Thread Blocking:** Never use `Thread.sleep()`, `time.Sleep()`, or any construct that blocks the OS thread inside workflow code. This consumes worker capacity, prevents the worker from processing other tasks, and completely subverts Temporal's durability guarantees. If the worker crashes during an OS-level sleep, the progress is lost.
- **External Cron Drift:** Avoid triggering business workflow progression via external cron jobs (e.g., an AWS CloudWatch event that pings an API). The external cron has no awareness of the workflow's actual state. If the workflow was delayed or paused, the cron will fire out of sync, requiring complex reconciliation. Use Temporal's native timers or schedules instead.
- **Untyped Timers:** Never place a `Workflow.sleep(30 days)` in code without surrounding it with explicit state updates or search attributes. An unannotated 30-day sleep makes the workflow a

black box to operators, leading to confusion during incident response when someone asks, “Why is this order not shipping?” Explicitly transition the state before sleeping.

4.2. Building Fan-out/Fan-in Aggregations with Bounded Concurrency and Backpressure

Fan-out/fan-in orchestration frequently degrades into a brittle web of asynchronous callbacks, thread pools, and queue backlogs. While Temporal provides deterministic concurrency constructs that make executing parallel tasks trivial, the architectural challenge lies in controlling the downstream impact. When parallel execution scales without constraint, the resulting surge in requests transforms a well-intentioned orchestration into a distributed denial-of-service attack on your own infrastructure.

The Conceptual Model for Controlled Fan-out

Treating fan-out as an explicit execution plan requires defining four parameters for every parallel operation: the unit of work, the concurrency limit, the aggregation semantics, and the failure policy. Activity retry and timeout mechanics handle transient infrastructure failures, but the orchestration layer must dictate how many activities are allowed to run simultaneously and how their collective results are synthesized.

The foundational pattern for controlled parallelism is bounded concurrency. Rather than iterating over a list of items and dispatching them all simultaneously, the workflow employs a deterministic semaphore to cap the number of inflight activities. This ensures the workflow generates tasks at a predictable rate, preventing both task queue flooding and history bloat.

Consider the following Python SDK implementation that processes a

batch of items with a strict concurrency limit:

```
from temporalio import workflow
import asyncio
from datetime import timedelta
from typing import List, Dict, Any

@workflow.defn
class BatchProcessingWorkflow:
    @workflow.run
    async def run(self, item_ids: List[str]) -> Dict[str, Any]:
        concurrency_limit = 50
        # A deterministic semaphore safe for workflow execution
        semaphore = asyncio.Semaphore(concurrency_limit)
        results = {}

        async def process_item(item_id: str):
            async with semaphore:
                try:
                    # Execute the downstream call
                    result = await workflow.execute_activity(
                        "process_single_item",
                        item_id,
                        start_to_close_timeout=timedelta(seconds=10),
                    )
                    return item_id, {"status": "success", "data":
result}
                except Exception as e:
                    # Trap activity failures to prevent the entire
                    batch from failing
                    return item_id, {"status": "failed", "error": str
(e)}

        # Dispatch all tasks; the semaphore restricts inflight
        execution to 50
        tasks = [process_item(item_id) for item_id in item_ids]
        completed = await asyncio.gather(*tasks)

        for item_id, outcome in completed:
            results[item_id] = outcome

        return results
```

In this pattern, the workflow evaluates the loop immediately, but the `asyncio.Semaphore` suspends the coroutines before they schedule the activity. At any given moment, no more than 50 `ScheduleActivity` commands are written to the workflow history. As activities complete,

the semaphore releases, allowing the next pending coroutine to schedule its activity. This creates a smooth, predictable pipeline of execution regardless of whether the input list contains 100 or 10,000 items.

Staged Fan-out for Massive Datasets

When the input set scales to hundreds of thousands of items, instantiating a coroutine for every item simultaneously will exceed memory constraints and workflow history limits, even if a semaphore restricts the active execution count. For massive datasets, you must implement staged fan-out.

Staged fan-out partitions the total workload into discrete batches—for example, processing 5,000 items at a time. The workflow executes the bounded concurrency pattern over the first batch, aggregates the results, and writes the partial state to external storage (such as an Amazon S3 bucket) rather than holding the entire accumulated map in workflow memory. Once the batch is complete and the results are persisted, the workflow calls `Continue-As-New` to reset its history, passing the external storage reference and the offset for the next batch. This design guarantees the orchestration can process millions of records while maintaining a flat, predictable memory and history profile for each execution phase.

Aggregation Semantics and Result Schemas

Once the concurrency is controlled, you must define the aggregation semantics. The default behavior of `asyncio.gather` or a `Promise.allOf` is strict aggregation: if a single child operation fails, the entire future resolves to an error, discarding the results of the successful operations. In microservice architectures, strict aggregation is rarely desirable for bulk operations.

Instead, favor best-effort aggregation with error budgets. As demonstrated in the earlier code snippet, the coroutine traps the activity ex-

ception and maps it into a structured outcome schema. The workflow returns an aggregate payload that preserves per-item outcomes: success, timeout, or failed. This allows the invoking service or human operator to parse the result, identify exactly which downstreams failed, and determine the root cause without spelunking through distributed logs.

When you require redundancy rather than bulk processing, use quorum aggregation. You might dispatch identical read requests to five different replicas but only require three successful responses to proceed. You can implement this by tracking the number of successful completions in a loop using primitives like `asyncio.as_completed` (or Go's `workflow.Selector`). Once the quorum condition is met, the workflow explicitly cancels the remaining inflight activities. This trades minor compute overhead for significantly reduced latency and higher availability.

Tying Workflow Parallelism to System Capacity

Workflow-level concurrency dictates how many tasks are inserted into the task queue, but physical downstream throughput is governed by the worker fleet. If a workflow specifies a concurrency limit of 500, but the worker pool consists of 10 nodes each configured with a maximum concurrent activity execution size of 10, the workers will fully saturate at 100 concurrent executions. The remaining 400 tasks will sit in the task queue, aging until worker slots become available.

This separation between orchestration logic and physical execution capacity is a critical architectural feature. It allows the workflow to express its ideal parallelism while the worker configuration enforces the hard physical limits of the infrastructure. However, if a single task queue is used for all activities, a massive fan-out workflow can completely exhaust the worker fleet's slots, starving latency-sensitive transactional workflows that share the same queue.

To prevent this blast radius expansion, assign heavy fan-out workloads to dedicated, bulk-processing task queues serviced by an isolated worker fleet. This guarantees that background data processing cannot disrupt user-facing requests.

Establishing a Backpressure Contract

While worker capacity tuning acts as a static defense against system saturation, dynamic downstream outages require an active orchestration response. If the downstream API enforces strict rate limits, standard activity retry policies will cause tasks to sleep on the server and retry exponentially. While this avoids overwhelming the worker threads, a large fan-out operation will eventually accumulate thousands of pending retries. When the downstream service recovers, these synchronized retries can immediately trigger a secondary failure wave—a retry storm.

To prevent retry amplification, move the backpressure handling into the orchestration layer. Configure the activity to fail fast when it encounters a 429 Too Many Requests or similar saturation signal, disabling the standard retry policy for that specific error. When the workflow catches the rate-limit exception from an activity, it intercepts the fan-out loop and executes a durable timer (e.g., `await asyncio.sleep(60)`). This pauses the scheduling of all new activities across the entire workflow, effectively halting the fan-out process and providing the downstream service a guaranteed cool-down window. Once the timer elapses, the workflow resumes scheduling tasks.

Operational Flow: Managing Downstream Outages During Fan-out

When a downstream service experiences an extended outage, an active fan-out workflow will begin accumulating failed items. To manage this operationally, follow a structured incident playbook:

1. **Observe Saturation:** Monitor the activity failure metrics and task queue backlog. The structured result schema will surface the failing dependencies directly in the workflow's state.
2. **Halt Execution:** If the workflow exposes a Signal handler for configuration changes, emit a Signal to dynamically reduce the concurrency limit to zero, pausing further task generation.
3. **Isolate and Cancel:** If the orchestration is no longer relevant due to the delay, cleanly terminate the workflow. Because bounded concurrency limits the inflight tasks, cancellation only affects the currently active subset, leaving the rest untouched.
4. **Re-drive:** Once the downstream service recovers, inspect the persisted result schema from the workflow. Extract the IDs of the items marked as failed or timed out and initiate a targeted re-drive workflow, entirely bypassing the items that previously succeeded.

Trade-offs and Anti-patterns

Designing robust aggregations requires balancing latency against blast radius. Higher concurrency reduces overall batch latency but increases the variance of individual requests and the likelihood of correlated failures.

Avoid the anti-pattern of unbounded fan-out. Developers often write a loop that creates thousands of asynchronous tasks, assuming the underlying framework will magically manage the load. In orchestration engines, this behavior translates to massive history payloads and degraded worker performance.

Equally dangerous is the practice of aggregating unbounded data in workflow memory. Workflows must remain lightweight coordinators; accumulating gigabytes of payload data inside the workflow state will eventually crash the worker process and corrupt the execution history.

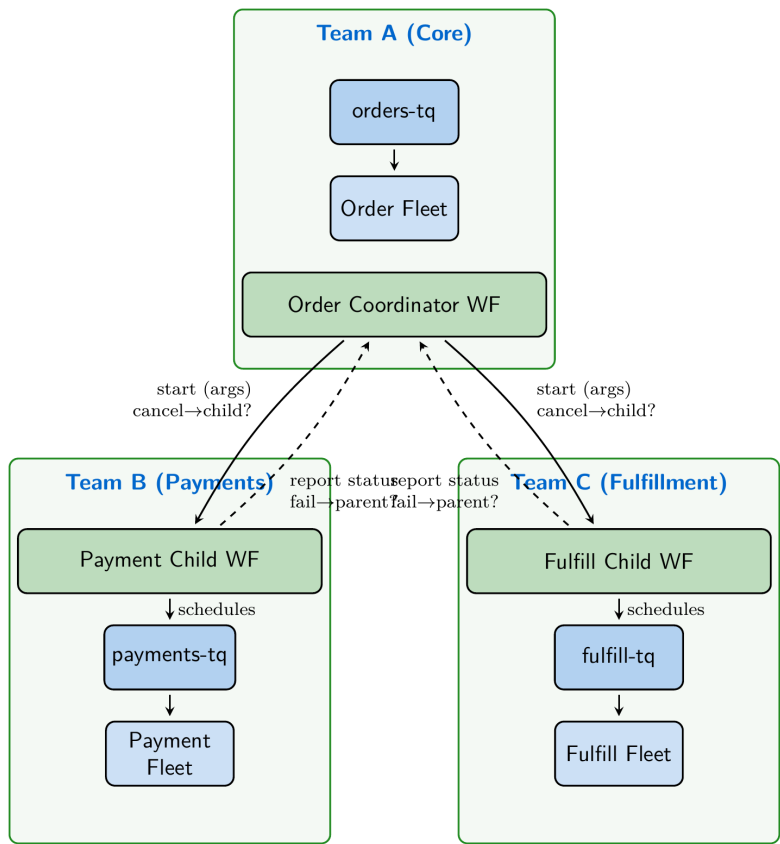
Finally, do not emit opaque failure statuses from a fan-out process. A workflow that orchestrates 1,000 items and simply returns a generic workflow exception forces operators to comb through execution histories and external logs to deduce which specific items caused the failure. Always capture exceptions at the item level and return a deterministic, auditable dataset that maps every input to its explicit outcome.

4.3. Designing Child Workflows for Decomposition, Team Ownership, and Fault Isolation

Microservice architectures naturally fracture business processes across domain boundaries. An e-commerce order involves inventory allocation, payment capture, fulfillment, and customer notification. When orchestrating these processes, modeling the entire end-to-end transaction as a single workflow with dozens of direct activity calls quickly creates a monolithic orchestrator. This orchestrator becomes tightly coupled to the data structures, SLAs, and deployment lifecycles of every downstream dependency.

Child workflows provide a structural solution to this decomposition problem. While an activity represents a single external side effect or I/O operation, a child workflow represents a composable unit of orchestration. A child workflow can wait on durable timers, react to external signals, maintain long-lived state, and execute its own internal sequence of activities. Most importantly, it executes with its own event history and can be routed to an entirely different worker fleet.

By modeling domain segments as child workflows, you establish explicit organizational interfaces. The parent workflow coordinates the high-level business transaction, while child workflows encapsulate the specific operational mechanics of each domain.



Defining the Contract and Execution Boundary To configure a child workflow, you declare a rigorous contract defining how it should start, how it handles cancellation, and where it executes. The following Go SDK example demonstrates an order parent workflow invoking a payment child workflow. Notice the explicit configuration of organizational and lifecycle boundaries.

```
func OrderCoordinatorWorkflow(ctx workflow.Context, orderID string)
    error {
```

```

// 1. Define the child workflow execution boundary
childOpts := workflow.ChildWorkflowOptions{
    // Route execution to the Payment team's infrastructure
    TaskQueue: "payments-tq",

    // Enforce idempotency: prevent accidental duplicate
    executions
    WorkflowID: "payment-child-" + orderID,
    WorkflowIDReusePolicy: enums.
    WORKFLOW_ID_REUSE_POLICY_REJECT_DUPLICATE,

    // Define what happens to the child if the parent is
    cancelled or fails
    ParentClosePolicy: enums.PARENT_CLOSE_POLICY_REQUEST_CANCEL,

    // Isolate SLA limits from the parent
    WorkflowExecutionTimeout: 2 * time.Hour,
}

ctx = workflow.WithChildOptions(ctx, childOpts)

// 2. Start the child workflow
paymentInput := PaymentRequest{OrderID: orderID, Amount: 150.00}
childFuture := workflow.ExecuteChildWorkflow(ctx, "
    PaymentChildWorkflow", paymentInput)

// Wait for the child workflow to start successfully (state
    durably recorded)
var childExecution workflow.Execution
if err := childFuture.GetChildWorkflowExecution().Get(ctx, &
    childExecution); err != nil {
    return fmt.Errorf("failed to start payment child workflow: %w
", err)
}

// 3. Wait for the child workflow to complete
var paymentResult PaymentResponse
if err := childFuture.Get(ctx, &paymentResult); err != nil {
    // Handle isolated failure domain
    var childErr *temporal.ChildWorkflowExecutionError
    if errors.As(err, &childErr) {
        workflow.GetLogger(ctx).Error("Payment domain failed", "
Reason", childErr.Unwrap())
        return initiateCompensations(ctx, orderID)
    }
    return err
}

```

```
    return nil  
}
```

This code snippet establishes more than just asynchronous execution. By defining `TaskQueue`, `ParentClosePolicy`, and `WorkflowIDReusePolicy`, the parent explicitly designs the fault isolation and routing parameters for the downstream process.

Enforcing Team Boundaries with Task Queues When you specify `TaskQueue: "payments-tq"` in the `ChildWorkflowOptions`, you decouple the execution of the sub-process from the parent's worker fleet. This routing mechanism directly translates Conway's Law into orchestration architecture.

The Order team maintains the parent workflow and deploys workers polling `orders-tq`. The Payments team maintains the `PaymentChildWorkflow` implementation and deploys workers polling `payments-tq`. Because the child workflow definition (its function signature and argument types) acts as a stable API contract, the Payments team can iterate on their internal orchestration—adding new activities, refactoring wait conditions, or altering retry loops—without requiring the Order team to redeploy or retest the parent workflow.

This decoupling works gracefully within a single namespace. For cross-namespace or cross-cluster composition, where strict administrative isolation is required between business units, you rely on Nexus endpoints (detailed in Chapter 8) rather than direct child workflow invocation. However, within a shared administrative boundary, task queue routing remains the most efficient method for segmenting ownership.

To maintain these boundaries safely, establish strict contract testing rules (the mechanics of which are covered in Chapter 7). Changes to the child workflow's input parameters, return types, or emitted signals

constitute an orchestration API break. Internal changes to how the child achieves its output do not.

Controlling Failure and Cancellation Propagation Fault isolation requires explicitly defining what happens when a workflow hierarchy encounters an error. By default, a child workflow encapsulates its own activity retries. If an activity within the child fails transiently, the child retries it without surfacing the churn to the parent. However, if the child exhausts its retries or explicitly fails due to a terminal business rule (e.g., “Customer Account Suspended”), the failure propagates back to the parent as a `ChildWorkflowExecutionError`.

You must design the parent to anticipate these domain-level failures. Rather than allowing the parent workflow to crash and retry, use typed domain errors to catch the failure and make a deterministic routing decision. The parent can then trigger alternative flows, such as pausing for human intervention, falling back to a degraded mode, or executing a compensation saga.

Equally important is the `ParentClosePolicy`, which dictates how cancellation propagates downward. You select this policy based on the child’s responsibility:

- *TERMINATE* (Default): The child workflow is immediately killed if the parent completes, fails, or is cancelled. Use this for purely read-only or inherently safe sub-processes where abrupt termination leaves no stranded external state.
- *REQUEST_CANCEL*: The parent sends a polite cancellation token to the child. The child workflow code catches this cancellation, executes necessary cleanup activities (like voiding an auth or releasing a locked resource), and then exits gracefully. This is mandatory for child workflows managing external financial or physical side effects.

- *ABANDON*: The child workflow becomes entirely detached from the parent's lifecycle. If the parent workflow completes or fails, the child continues running to its own completion.

When using the *ABANDON* policy, you must ensure the child has successfully launched before the parent concludes. As shown in the code example, always await the `GetChildWorkflowExecution()` promise. If the parent workflow completes before the server durably records the child start event, the child workflow will never execute.

Containing Blast Radius and Event History Growth Beyond team ownership, child workflows act as physical firewalls for workflow state. A core operational constraint of Temporal is the event history limit. Every activity scheduled, started, and completed writes events to the workflow history. If an orchestration requires polling an external system via an activity every 5 minutes for a month, or iterating over a batch of 10,000 items, placing these activities directly inside the parent workflow will quickly trigger a `HistorySizeExceeded` failure.

Child workflows provide a durable state boundary. No matter how many activities, timers, or child-level `Continue-As-New` iterations occur inside the child, the parent workflow's event history only records the `ChildWorkflowExecutionStarted` and `ChildWorkflowExecutionCompleted` (or `Failed`) events.

Use child workflows to isolate noisy, high-churn subsystems. If an automated background check involves hundreds of micro-interactions and rapid retries, encase that logic in a child workflow. The parent workflow remains pristine, containing only the high-level milestones of the business process. This separation heavily improves observability; an operator debugging the parent workflow (using visibility tools covered in Chapter 6) sees a clean progression of domain milestones, while an operator debugging the child workflow sees the exhaustive

granular details of the specific task.

Operational Playbook: Managing Degraded Child

Workflows When executing workflows that cross team boundaries, operational failures often occur asynchronously. A common incident involves a downstream child workflow becoming severely degraded—for instance, the Fulfillment team’s API provider suffers an outage, causing Fulfillment child workflows to stall on activity retries.

Because child workflows isolate execution, the parent workflows will not crash. Instead, they will pause, awaiting the completion of the child future. To manage this safely, build operational degradation paths into the parent workflow:

- *Detect saturation:* When invoking a child workflow that interfaces with unstable external domains, do not wait on the future indefinitely. Wrap the child execution in a parent-side timer or deadline race.
- *Graceful degradation:* If the child fails to complete within an expected SLA (e.g., 4 hours), the parent workflow catches the timeout. The parent can then emit a metric, tag its own searchable metadata with `status=degraded_fulfillment`, and continue processing other unrelated parallel branches.
- *Targeted re-drives:* Because the child workflow was isolated, operators can manage the Fulfillment team’s backlog independently. Once the downstream API recovers, operators can selectively reset or unpause the affected child workflows. The parent workflows dynamically resume their orchestration once the children finally report success.

Trade-offs and Anti-patterns The decision to decompose a process into a child workflow carries a performance trade-off. Starting a child workflow requires multiple trips to the server to persist the new execution state, create a new event history, and route tasks to a new queue. This overhead is negligible for long-running business processes, but prohibitive for high-throughput, low-latency micro-operations.

Choose an activity when the operation is a single, cohesive external side effect that requires straightforward retries and timeout boundaries. Choose a child workflow when the operation requires durable waiting, complex multi-step state transitions, explicit encapsulation from the parent's event history, or execution on a team-specific worker fleet.

Avoid the anti-pattern of wrapping every single activity in its own child workflow. Over-composition leads to deep, fragmented execution chains (e.g., Parent → Child → Grandchild → Great-Grandchild) that are exceedingly difficult to debug and monitor. Flatten your orchestration where possible, utilizing child workflows strictly at the boundaries of domain ownership, SLA isolation, and logical encapsulation.

Furthermore, ensure you establish clear failure contracts. A child workflow that fails with a generic, un-typed error message forces the parent workflow to parse string details to determine if compensation is necessary. Always design child workflows to return or fail with strongly typed domain objects, enabling the parent workflow to execute deterministic and auditable recovery logic.

4.4. Encoding Sagas and Compensation Plans as Auditable Workflow State

Distributed transactions often fail in the gaps between services. A downstream dependency accepts a payload but drops the confirmation, a subsequent processing step violates a business constraint, or an administrator manually aborts a process midway. Because you cannot hold database locks across independent microservices, you must rely on sagas: ordered sequences of local transactions, paired with a defined rollback plan.

Implementing a saga requires treating the compensation plan as an explicit, durable state machine. Instead of relying on ad-hoc cleanup scripts or trusting that downstream services will auto-expire pending resources, you encode the exact rollback steps into the workflow's history. This transforms the saga from an abstract architectural pattern into a concrete, observable operational artifact.

The Compensation Stack Pattern

The foundation of a reliable saga is the compensation stack. You maintain a dynamic list of rollback operations, pushing a new compensation onto the stack only after its corresponding forward action confirms success. If a forward action fails, you execute the accumulated compensations in reverse order.

This approach separates three distinct concerns: activity-level retries for transient failures, orchestration-level rollback for business rule violations, and human intervention for unrecoverable states.

Consider the following Temporal Go SDK implementation of a multi-step onboarding saga. It uses Go's defer statement and a slice of functions to guarantee that compensations run on failure.

```
func OnboardingSaga(ctx workflow.Context, req OnboardingRequest) (error) {
```



```
var compensations []func(workflow.Context) error

// Ensure compensations execute if the workflow returns an error
// or is canceled.
defer func() {
    if err != nil {
        // Use a disconnected context so compensations can run
        even if
        // the parent context was canceled.
        compCtx, _ := workflow.NewDisconnectedContext(ctx)

        // Execute compensations in reverse order (LIFO).
        for i := len(compensations) - 1; i >= 0; i-- {
            compErr := compensations[i](compCtx)
            if compErr != nil {
                // Log compensation failure; this often requires
                human escalation.
                workflow.GetLogger(ctx).Error("Compensation
                failed",
                    "Error", compErr)
            }
        }
    }
}()

// Step 1: Provision the user account
var accountID string
err = workflow.ExecuteActivity(ctx, ProvisionAccount, req).Get(
    ctx, &accountID)
if err != nil {
    return err // Fails the workflow, executing any registered
    compensations (none yet)
}

// Register compensation for Step 1
compensations = append(compensations, func(c workflow.Context)
    error {
        return workflow.ExecuteActivity(c, DeleteAccount, accountID).
        Get(c, nil)
    })

// Step 2: Charge the payment method
err = workflow.ExecuteActivity(ctx, ChargeCard, accountID, req.
    Amount).Get(ctx, nil)
if err != nil {
    return err // Fails the workflow, triggering the
    DeleteAccount compensation
}
```

```
// Register compensation for Step 2
compensations = append(compensations, func(c workflow.Context)
    error {
        return workflow.ExecuteActivity(c, RefundCard, accountID, req
            .Amount).Get(c, nil)
    })

// Step 3: Send welcome kit (cannot be easily reversed)
err = workflow.ExecuteActivity(ctx, ShipWelcomeKit, accountID).
    Get(ctx, nil)
if err != nil {
    return err // Triggers RefundCard, then DeleteAccount
}

return nil
}
```

This structure ensures that compensations are strictly tied to recorded history. You only register the `DeleteAccount` compensation because `ProvisionAccount` successfully returned an `accountID`. If `ProvisionAccount` fails after five retries, the workflow terminates without executing compensations, because there is nothing to undo.

Compensation Semantics and Time-Sensitive Rollbacks

Real-world compensations rarely map to a simple “delete” operation. You must design compensation activities to be highly reliable and strictly idempotent, as they may be retried extensively if the network partitions during the rollback phase.

Compensations often degrade into partial rollbacks or time-sensitive variants. For example, reversing a credit card transaction within the first few minutes can be performed via a “Void” operation, which prevents the charge from appearing on the customer’s statement. If the transaction has already settled, the compensation must change to a “Refund”, which incurs processing fees and alters the customer experience.

You can encode these time-sensitive semantics by leveraging durable

timers within the compensation logic itself.

```
compensations = append(compensations, func(c workflow.Context) error
{
    elapsed := workflow.Now(c).Sub(chargeCompleteTime)
    if elapsed < 5*time.Minute {
        return workflow.ExecuteActivity(c, VoidCharge, accountID).Get(
            c, nil)
    }
    return workflow.ExecuteActivity(c, RefundCharge, accountID).Get(c,
        nil)
})
```

By capturing the `workflow.Now()` timestamp immediately after the forward action succeeds, the compensation block becomes completely deterministic. When the workflow replays to execute the rollback, it evaluates the exact time that elapsed between the forward action and the failure, ensuring the correct compensation path is chosen consistently.

Classifying Failures with Domain-Typed Errors

Not all errors should trigger a saga rollback. If a downstream service is temporarily unavailable, the correct action is usually to pause and retry the forward operation. Rollbacks should be reserved for terminal conditions: business validation failures, permanent resource exhaustion, or explicit cancellation.

You must design your activities to return domain-typed errors that the orchestrator can inspect. By classifying failures, the workflow makes auditable decisions about whether to retry, compensate, or escalate.

```
err = workflow.ExecuteActivity(ctx, ChargeCard, accountID, amount).
    Get(ctx, nil)
if err != nil {
    var appErr *temporal.ApplicationError
    if errors.As(err, &appErr) {
        switch appErr.Type() {
            case "InsufficientFundsError", "CardDeclinedError":
                // Terminal business failure: trigger compensation
                return err
            case "FraudLockoutError":
                // Escalate to human review without immediately
                compensating
```

```
        return TriggerManualReview(ctx, accountID)
    }
}
// For unknown errors, rely on activity retry policies.
// If retries exhaust, the activity returns a TimeoutError,
// triggering compensation.
return err
}
```

This classification guarantees that transient infrastructure issues do not accidentally unravel complex business processes, while legitimate business rejections are handled swiftly.

Executing Compensations Under Cancellation

One of the most frequent operational mistakes in saga design is failing to protect the rollback phase from cancellation requests. When an operator or an upstream system cancels a workflow, the workflow context is immediately marked as canceled. Any standard activity execution attempted with a canceled context will fail instantly without making a network request.

If a cancellation request arrives while the saga is mid-flight, you must ensure the workflow halts forward progress but still runs its required compensations. This requires executing the compensation activities in a disconnected context—a context that inherits the workflow’s state but ignores the parent’s cancellation signal.

In the Temporal SDKs, this is achieved via `workflow.NewDisconnectedContext` (Go) or `Workflow.newDisconnectedCancellationScope` (Java). Without this explicit scope, a canceled saga leaves the system in a permanently inconsistent state, with forward actions committed and compensations bypassed. The compensation scope itself should typically have a hard deadline applied to it, ensuring that a stuck compensation does not block the workflow from eventually closing.

Operational Visibility into In-Flight Sagas

Sagas can span hours or days. During an incident, operators must be able to inspect a workflow and answer critical questions: What phase is this saga in? How many steps are currently irreversible? Which compensations are pending?

If this data is buried in the workflow history's binary payload, incident response stalls. You must project the saga's internal state into searchable metadata and queryable structures.

Define a structured query handler that returns the exact state of the compensation plan.

```
type SagaState struct {
    CompletedSteps      []string
    PendingCompensations int
    IsIrreversible      bool
}

// Inside the workflow definition:
workflow.SetQueryHandler(ctx, "GetSagaState", func() (SagaState,
    error) {
    return SagaState{
        CompletedSteps:      completedStepNames,
        PendingCompensations: len(compensations),
        IsIrreversible:      hasPassedIrreversibleStep,
    }, nil
})
```

Additionally, update the workflow's Search Attributes as it crosses critical thresholds. For instance, when the saga executes a step that cannot be compensated—such as mailing a physical check—update a `SagaPhase` search attribute to `Irreversible`. This allows operations teams to bulk-filter workflows and safely apply batch cancellations only to sagas that can cleanly roll back.

Irreversibility and Mitigation Strategies

A saga eventually reaches a point of no return. Some side effects cannot be logically compensated. A sent email cannot be unsent; a dispatched

physical shipment cannot always be intercepted.

When designing a saga, you must identify the boundary of irreversibility. Workflows should be structured to perform all compensatable actions first, deferring irreversible actions to the absolute end of the process.

When a failure occurs after an irreversible step, you cannot blindly execute the compensation stack. Instead, you transition from compensation to mitigation. Mitigation involves executing alternative workflows or alerting human operators to resolve the physical or external inconsistency.

For example, if a fulfillment center has already loaded a package onto a delivery truck, the `CancelShipment` compensation activity will fail with a domain-specific error indicating the package is in transit. The workflow must catch this specific failure and execute a mitigation strategy, such as dispatching a `SendReturnLabelEmail` activity to the customer and creating a ticket in the customer support system.

You can also design external APIs to be more compensatable. Instead of treating a payment as a single `Charge` action, split it into `Authorize` and `Capture`. The `Authorize` step locks the funds and is easily voidable during the compensation phase. You only invoke `Capture` at the very end of the saga, crossing the irreversibility boundary only when all other business rules have passed.

Operational Playbook: Intervening in a Stuck Saga

Even with exhaustive retry policies and durable compensations, a saga can become irrevocably stuck. A downstream API might introduce a breaking schema change, causing a compensation activity to fail continuously. Because the workflow is programmed to retry the compensation until it succeeds, the saga will remain active indefinitely.

Operations teams require a defined playbook to resolve these dead-

locks. You must provide mechanisms to intervene in the saga's execution without deploying new code.

The standard operational flow for a stuck saga proceeds as follows:

- *Identify the block:* Use the `GetSagaState` query to determine exactly which compensation is failing. Inspect the workflow history to view the specific activity error.
- *Evaluate the external state:* Check the downstream system to determine if the compensation actually succeeded but the network response was lost, or if it is genuinely failing due to a data constraint.
- *Inject an administrative override:* Expose a signal handler in your workflow that allows an operator to bypass a specific compensation.

```
var skipNextCompensation bool
workflow.SetSignalHandler(ctx, "ForceSkipCompensation", func() {
    skipNextCompensation = true
})

// Inside the compensation loop:
for i := len(compensations) - 1; i >= 0; i-- {
    if skipNextCompensation {
        workflow.GetLogger(ctx).Warn("Operator force-skipped
        compensation",
            "Index", i)
        skipNextCompensation = false
        continue
    }
    // Execute compensation...
}
```

- *Audit the intervention:* When the operator fires the `ForceSkipCompensation` signal, the signal payload and the subsequent logger warning are recorded permanently in the workflow history. This provides a perfect audit trail showing

exactly who intervened, when, and which step was bypassed, ensuring that manual overrides do not create silent data corruption.

By encoding the compensation plan explicitly within the workflow state, mapping out time boundaries, and exposing control surfaces to operators, the saga transitions from a fragile theoretical model into a robust, observable engine for distributed state management.

4.5. Establishing Task Queue and Namespace Conventions for Routing, Isolation, and Multi-Team Operations

Orchestration platforms often fail at scale not because of theoretical throughput limits, but because workloads from different teams and priority tiers collide in shared execution pools. Task queues and namespaces are the physical manifestation of your architectural boundaries. They define who owns which workloads, how noisy neighbors are isolated, and how fleets scale independently. Treating these routing primitives as an explicit architectural contract ensures that your orchestration remains operable during outages, hotfixes, and hyper-growth.

When starting a worker process, you explicitly bind it to a task queue. This binding forms a consumption contract. The following Java configuration demonstrates how to map different workload classes to distinct worker pools, applying tailored concurrency limits to each queue:

```
WorkerFactory factory = WorkerFactory.newInstance(client);

// Critical, user-facing workload: high concurrency, dedicated fleet
WorkerOptions criticalOptions = WorkerOptions.newBuilder()
    .setMaxConcurrentWorkflowTaskExecutionSize(200)
    .setMaxConcurrentActivityExecutionSize(500)
    .build();
Worker criticalWorker = factory.newWorker("prod-fulfillment-critical", criticalOptions);
```



```
// Batch/Backfill workload: strict concurrency bounds to protect
// downstream
WorkerOptions batchOptions = WorkerOptions.newBuilder()
    .setMaxConcurrentWorkflowTaskExecutionSize(50)
    .setMaxConcurrentActivityExecutionSize(20)
    .build();
Worker batchWorker = factory.newWorker("prod-fulfillment-batch",
    batchOptions);

factory.start();
```

This code physically separates the execution of critical workflows from bulk operations, even if they execute the same underlying activity implementations.

The Hierarchy of Isolation: Namespaces and Task Queues

Isolation requires understanding the operational scope of both namespaces and task queues. Namespaces are the primary administrative boundary within Temporal. They provide strict, multi-tenant separation at the storage and API layers. Workflow IDs are unique only within a namespace, and cluster-level configurations—such as history retention periods and archival settings—are scoped per namespace.

Crucially, namespace-level constraints dictate your absolute capacity ceilings. In Temporal Cloud environments, Actions per Second (APS) limits are enforced at the namespace level. If a runaway load test or an unthrottled batch job exhausts the namespace APS quota, the Temporal service will throttle all requests within that namespace, degrading both critical and non-critical workflows alike. Because of this strict failure domain, namespaces should be partitioned by environment (e.g., dev, staging, prod) and by high-level tenant or business unit.

Task queues, by contrast, are dynamic routing and execution boundaries within a namespace. They are lightweight, created automatically on first use, and serve as the matching point between the Temporal server and your application workers. While namespaces isolate storage

and API quotas, task queues isolate compute resources. By dividing workloads across task queues, you dictate exactly which worker fleets process which tasks, controlling the blast radius of CPU spikes, memory leaks, or thread-pool exhaustion.

Topology Patterns for Routing and Isolation

Designing your task queue topology requires mapping your organizational boundaries and workload profiles to dedicated execution paths.

Per-Team Ownership Queues. When multiple teams contribute to a shared parent workflow, force execution boundaries using task queues. If the Checkout team invokes a fraud-check child workflow owned by the Risk team, the child workflow should be routed to the Risk team's task queue.

```
ChildWorkflowOptions fraudCheckOptions = ChildWorkflowOptions.  
    newBuilder()  
        .setTaskQueue("prod-risk-eval")  
        .setWorkflowId("fraud-check-" + transactionId)  
        .build();  
FraudCheckWorkflow child = Workflow.newChildWorkflowStub(  
    FraudCheckWorkflow.class,  
    fraudCheckOptions  
);
```

This routing contract ensures that the Risk team can independently deploy, scale, and monitor their worker fleets without coordinating with the Checkout team. If a bad deployment crashes the Risk workers, tasks will buffer safely in the `prod-risk-eval` queue. The Checkout workers remain entirely unaffected, continuing to process other parent workflow logic or routing compensating actions if the child workflow times out.

Per-Workload Queues for Noisy Neighbor Isolation. Mixing latency-sensitive operations (e.g., synchronous user checkout) and high-throughput bulk operations (e.g., end-of-month reconciliation) on the same task queue is a severe anti-pattern. A sudden influx of

batch tasks will saturate the worker pool's poller threads, causing user-facing workflows to experience unacceptable scheduling latency. Separate these workload classes by appending a tier identifier to the queue name (e.g., `payments-tier1` vs `payments-batch`). Route batch workflows to a worker fleet configured with lower concurrency limits, naturally smoothing the load on downstream databases without starving tier-1 operations.

Shard-by-Tenant Queues. In B2B SaaS platforms, a single massive customer can generate enough activity to overwhelm shared worker fleets, degrading performance for smaller tenants. Implement tenant-sharded queues when fairness guarantees are contractually required. Instead of routing all data ingestion to `prod-ingestion`, you route traffic dynamically: `prod-ingestion-tenantA` for enterprise clients and `prod-ingestion-shared` for the long tail. This allows you to provision dedicated compute resources for premium tenants and forcefully throttle aggressive tenants by capping the worker pool size on their specific queue.

Task Queue Naming Conventions as Architectural Contracts

Task queue names are string identifiers, making them susceptible to drift and fragmentation. Establish a strict, parsable naming convention across your engineering organization. A robust format follows a predictable schema:

```
[environment]-[domain]-[service]-[workload_class]
```

For example, `prod-finance-ledger-critical` or `stage-logistics-shipping-batch`.

This nomenclature pays immediate dividends during an incident. When an alert fires for high schedule-to-start latency on `prod-logistics-shipping-batch`, the on-call engineer instantly knows the environment, the owning team, the impacted service, and

the fact that it is a lower-priority bulk workload.

Do not embed ephemeral identifiers—such as deployment hashes, Kubernetes pod names, or version strings—into task queue names. Task queues represent stable routing destinations. Embedding ephemeral data forces you to constantly update parent workflows or API clients with new routing targets on every deployment, breaking the decouple-by-queue design.

Worker Sizing, Concurrency, and Throughput Planning

A task queue is only as fast as the worker fleet polling it. Worker-level throughput relies on the interplay between the task queue backlog and the worker's internal concurrency settings.

When you configure `setMaxConcurrentWorkflowTaskExecutionSize` and `setMaxConcurrentActivityExecutionSize`, you are defining the maximum number of threads (or coroutines) the worker will dedicate to those tasks. If you set the activity concurrency to 100 on a worker, and deploy 10 replicas of that worker, the total system concurrency for that task queue is strictly bounded at 1,000 parallel activities.

To build autoscaling worker fleets, do not scale based on standard CPU or memory metrics alone, as workers often spend the majority of their time awaiting I/O from downstream dependencies. Instead, scale on the combination of task queue backlog size and worker slot utilization. Emit metrics for `temporal_worker_task_slots_available`; when the available slots across the fleet drop near zero, the fleet is saturated and must scale out. Conversely, if downstream systems are failing due to rate limits, scaling the worker fleet out will only amplify the damage. In backpressure scenarios, you must constrain the worker pool size or dynamically lower the worker concurrency limits to relieve pressure on the failing downstream service.

Deploy Safety and Worker Versioning

As multi-team coordination grows, shared task queues become risky during deployments. If two versions of a workflow implementation poll the same queue, tasks will be routed non-deterministically between the old and new code, causing replay failures.

Worker Versioning mechanisms (which entered public preview in mid-2025) fundamentally change how task queues handle deployments. By assigning a Build ID to your worker fleet and opting into versioning, the Temporal routing layer ensures that a workflow execution is permanently pinned to the version of the code where it started.

When you deploy a new fleet with an updated Build ID to the exact same task queue, the routing engine enforces the split: new workflow executions route to the new Build ID fleet, while existing open workflows continue routing to the older Build ID fleet. This enables safe, side-by-side blue/green deployments on a single task queue without requiring complex queue migration logic. You must leave the old worker fleet running until all workflows bound to that prior version have completed.

Operational Playbook: Mitigating Noisy Neighbors

When an upstream system floods your orchestration layer, or a runaway script generates an unexpected volume of heavy workflows, you must quickly isolate the traffic to protect critical paths. Execute the following playbook to contain noisy neighbor incidents:

- **Identify the Saturation Point:** Use metrics dashboards to locate the bottleneck. If `temporal_namespace_aps_limit_exceeded` is spiking, the entire namespace is under threat. If schedule-to-start latencies are climbing on specific task queues while APS remains healthy, the worker fleet is saturated.
- **Halt Polling on Non-Critical Queues:** If the noisy workload

shares a database or downstream dependency with critical workflows, suspend the workers polling the batch/non-critical task queues. Workflows on those queues will stall safely without failing, instantly shedding load from your downstreams.

- **Clamp Worker Concurrency:** If the noise originates on a critical queue that cannot be paused, push a configuration change to lower `setMaxConcurrentActivityExecutionSize` on the affected worker fleet. This applies immediate backpressure, stretching the execution over a longer time horizon without dropping tasks.
- **Isolate via Routing:** If the incident is caused by a specific workflow type flooding a shared queue, modify the starter code (or child workflow options) to route that specific workload to a temporary, isolated task queue (e.g., `prod-domain-incident-sink`). Deploy a minimal worker fleet to process that sink queue at a heavily restricted rate.

Operational Playbook: Queue Migration and Workload Refactoring

When architectural changes necessitate moving a workflow type from one team's ownership to another, or splitting a monolithic queue into workload-specific queues, follow a strict drain-and-replace migration sequence to prevent task loss or routing blackholes:

- **Provision the Destination:** Define the new task queue (e.g., `prod-newteam-service`). Deploy the new worker fleet configured to poll this destination queue. Verify via metrics that the workers are successfully polling, even though the queue is empty.
- **Update the Callers:** Modify all upstream clients, cron schedulers, and parent workflows to route new executions to `prod-newteam-service`. Deploy this routing change.

- **Observe the Split:** Monitor both queues. New executions will populate the new queue. Existing, long-running workflows will continue executing their subsequent workflow tasks and activities on the old queue.
- **Drain the Origin:** Do not tear down the old worker fleet. You must wait for all open workflows initiated on the old queue to reach a closed state. Query the system for open workflows filtering by the old task queue.
- **Decommission:** Once the old queue has zero open executions and no new tasks arriving, decommission the old worker fleet and remove the origin queue configuration from your infrastructure as code.

Anti-Patterns in Routing and Topology

Relying on a single global task queue (e.g., `prod-all-workflows`) destroys fault isolation. A poison-pill workflow that continuously panics the worker thread pool will crash the entire fleet, halting all business processes simultaneously. Global queues also make autoscaling impossible to tune, as you cannot optimize a single thread pool for both CPU-bound state machines and highly concurrent I/O-bound activities.

Attempting to use namespaces as a substitute for team boundaries is equally destructive. Namespaces impose strict isolation; workflows in Namespace A cannot use standard SDK primitives to synchronously signal, query, or spawn child workflows in Namespace B. If your teams build interdependent services, placing them in separate namespaces forces you to reinvent reliable cross-namespace communication using external messaging buses or raw API calls, sacrificing the deterministic transaction guarantees of the orchestration layer. Namespaces should segment environments and strict trust domains; task queues should segment teams and workloads.

Finally, performing migrations without a drain-and-backfill plan inevitably orphans in-flight state. If you abruptly shut down workers for a task queue that still contains open workflows, those workflows will wait indefinitely for a poller that will never arrive. The orchestration engine guarantees the durability of the state, but it is your architectural topology that guarantees its progress.

Chapter 5

Interacting with Running Workflows: Signals, Queries, and Updates

Temporal workflows are durable, but they're not sealed boxes: production systems must interrogate and steer them while they run. This chapter teaches a rigorous interaction model—queries for safe reads, signals for asynchronous event ingestion, and updates for validated, state-changing calls—then shows how to compose these primitives into real operational patterns like callback correlation and human approvals without fragile polling.

5.1. Choosing Between Signals, Queries, and Updates: Semantics, Consistency, and Failure Surfaces

Interaction with a running Temporal workflow does not resemble a traditional remote procedure call to an in-memory object. When you interact with a workflow, you are instructing the Temporal service to record an external intent into a durable history log and to schedule a workflow task. The worker process polling the corresponding task queue eventually dequeues this task, replays the execution history to reconstruct the workflow's state, and invokes the appropriate handler. This fundamental decoupling between the client's request and the worker's execution dictates the semantics, consistency guarantees, and failure modes of the three interaction primitives: signals, queries, and updates.

To ground these concepts, consider how the three primitives are defined within a standard workflow contract. The following Go snippet illustrates a shopping cart workflow exposing a query, an update, and a signal:

```
package shoppingcart

import (
    "errors"
    "go.temporal.io/sdk/workflow"
)

type CartState struct {
    Items  map[string]int
    Status string
}

func CartWorkflow(ctx workflow.Context, cartID string) (CartState,
    error) {
    state := CartState{Items: make(map[string]int), Status: "open"}

    // Query: Read-only snapshot, no history mutation
    err := workflow.SetQueryHandler(ctx, "getCart", func() (CartState
```

```
, error) {
    return state, nil
})
if err != nil {
    return state, err
}

// Update: Validated state transition with client acknowledgement
err = workflow.SetUpdateHandlerWithOptions(ctx, "addItem",
    func(ctx workflow.Context, itemID string, qty int) error {
        state.Items[itemID] += qty
        return nil
    },
    workflow.UpdateHandlerOptions{
        Validator: func(ctx workflow.Context, itemID string, qty
int) error {
            if qty <= 0 {
                return errors.New("quantity must be positive")
            }
            if state.Status != "open" {
                return errors.New("cart is closed")
            }
            return nil
        },
    },
)
if err != nil {
    return state, err
}

// Signal: Asynchronous event injection
checkoutChan := workflow.GetSignalChannel(ctx, "checkout")
var checkoutEvent string
checkoutChan.Receive(ctx, &checkoutEvent) // Blocks until signal
arrives

state.Status = "checking_out"
// Proceed to downstream activities...
return state, nil
}
```

Signals: Asynchronous Event Injection

Treat signals as durable, fire-and-forget messages delivered to a specific workflow execution. When a client sends a signal, the Temporal service appends a `WorkflowExecutionSignaled` event to the work-

flow’s history and immediately returns a success response. The client acknowledgement means exclusively that the Temporal service has persisted the event. It does not indicate that the workflow code has processed the signal, nor does it guarantee that the workflow state has changed in any meaningful way.

Because signals bypass workflow-level validation at the time of ingestion, you must handle structural and logical validation entirely within the workflow code. If a signal delivers an invalid payload, the workflow cannot reject it back to the caller; it must either discard the payload, log a warning, or fail the workflow execution entirely.

Ordering expectations for signals must be managed carefully. The Temporal service guarantees that signals are appended to history in the exact sequence they are received by the server. However, network realities and client-side retries mean that the arrival order at the server rarely matches the strict causal order generated by upstream systems. If your domain logic requires strict ordering, you must include sequence numbers or causal identifiers in the signal payload and buffer out-of-order signals within your workflow state.

A common anti-pattern is attempting to use signals as makeshift remote procedure calls. Developers sometimes design an interaction where a client sends a “request” signal and then polls a query-or waits for a “response” signal-to determine the outcome. Under network partitions or client crashes, the client may retry the “request” signal, resulting in duplicate events in the workflow history. Because the signal primitive provides no native deduplication based on business intent, the workflow must manually track which requests it has already processed to remain idempotent.

Queries: Read-Only Views

Queries provide a mechanism to inspect the internal state of a running or completed workflow. The defining property of a query is that it per-

forms no history mutation. When a worker receives a query task, it runs the workflow code up to the most recent event, evaluates the query handler against the current variable state, and returns the result to the caller. Because queries leave no trace in the workflow history, they are exceptionally cheap to execute and can be invoked frequently.

Query handlers must be strictly deterministic and side-effect free. You cannot schedule activities, mutate variables, or initiate child workflows from within a query handler. Attempting to do so violates the determinism constraints enforced by the SDK and will result in a worker panic or query failure.

The most subtle failure surface of a query involves data freshness. A query always executes against the state of the workflow as of its last processed event. If a workflow is actively processing a backlog of tasks, or if the worker fleet is saturated, the query result reflects a mathematically valid historical state that may lag behind real-world time.

When designing your interaction surfaces, classify your query usage into a strict taxonomy of freshness requirements:

- **Debug and Visibility:** High tolerance for staleness. Operators querying workflows to understand stuck states are unaffected by milliseconds or seconds of lag.
- **UX Display:** Medium tolerance. Displaying a status to an end user via a web interface can tolerate slight delays, provided the UI is designed to handle eventual consistency.
- **Routing Decisions:** Low tolerance. Using a query to decide whether an external load balancer should route traffic to a specific workflow is dangerous. The state may transition immediately after the query returns.
- **Correctness Gates:** Zero tolerance. Never use a query to check if a workflow is in a specific state before sending a signal (the

“check-then-act” race condition). Use an update to validate the state and apply the mutation atomically.

Queries also possess an explicit failure surface at the API layer. Clients can configure a `QueryRejectCondition` to instruct the server to reject the query if the workflow is in a specific state (for example, closed or failed). Your client code must always anticipate and handle query rejections rather than assuming the network call will definitively yield a business result.

Updates: Validated State Transitions

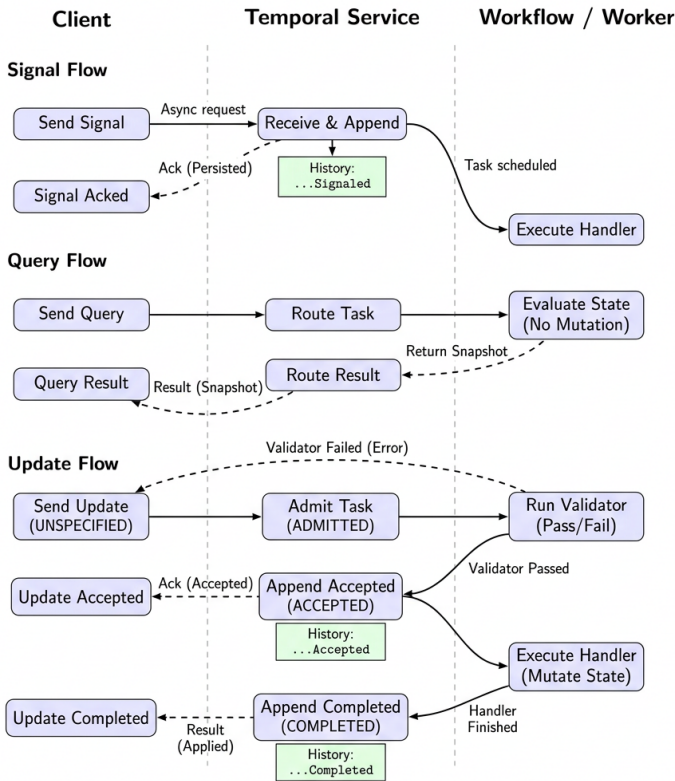
Updates provide synchronous, state-changing calls with strict acknowledgement guarantees. They exist to bridge the gap between the speed of signals and the safety of RPCs, giving callers a durable acknowledgement that a specific transition was validated and applied according to the workflow’s business logic. Note that semantics and availability of workflow updates depend on the specific Temporal Server and SDK versions in use; verify server compatibility and ensure any required feature flags are enabled before standardizing on this primitive.

An update request progresses through a strictly ordered lifecycle maintained by the Temporal server: `UNSPECIFIED` → `ADMITTED` → `ACCEPTED` → `COMPLETED`. When a client issues an update, the server admits it and creates an update task for the worker. The worker intercepts this task and executes the update’s validator function.

The validator is the crucial distinction between an update and a signal. The validator function executes strictly as a read-only precondition check. It must not mutate state or schedule commands. If the validator returns an error, the worker reports the rejection to the server, the server returns the error to the client, and absolutely no record of the update is appended to the workflow history. This protects long-running workflows from accumulating history bloat due to invalid external requests.

If the validator passes, the update is **ACCEPTED**. The Temporal service durably appends a `WorkflowExecutionUpdateAccepted` event to the history. At this exact moment, the workflow state transition is guaranteed to execute, even if the worker crashes immediately afterward. The workflow handler then applies the state mutation and can optionally schedule activities. Once the handler returns, the update is **COMPLETED**, and a `WorkflowExecutionUpdateCompleted` event is written to history.

Client retry semantics for updates are drastically simplified compared to signals. Every update request includes a unique update ID. If a network partition causes a client to retry an update request, the Temporal server recognizes the duplicate ID. If the original update was already accepted, the server will not execute the validator or handler again; it will simply attach the client to the pending update's lifecycle and wait for its completion. This enforces exactly-once logical execution for the mutation without requiring you to build custom deduplication logic inside the workflow.



Trade-offs and Operational Failure Modes

Choosing the correct interaction primitive requires balancing latency against certainty, and throughput against per-call cost.

Latency versus Certainty: Signals offer the lowest possible latency for the caller because the request completes as soon as the Temporal server writes to its persistence layer. However, this trades away all certainty regarding business constraints. The caller does not know if the cart was open, if the item was in stock, or if the payload was formatted correctly. Updates incur higher latency because the request must

traverse the server, wait in a task queue, be evaluated by a worker, and be written back to the server before the client receives an acknowledgement. This wait provides absolute certainty that the business constraints were met and the state transition occurred.

Throughput versus Cost: Signals and queries represent the most efficient use of resources. A signal generates only one event in history. A query generates zero events and requires no disk I/O. Updates are more computationally and storage intensive. A single successful update appends at least two events (`WorkflowExecutionUpdateAccepted` and `WorkflowExecutionUpdateCompleted`) to the workflow history and demands multiple round-trips between the server and the worker. If your workload involves streaming thousands of telemetry data points per second to a workflow, signals combined with internal batching are required to maintain performance. If you are handling human-driven approvals or financial transactions, the cost of updates is strictly justified by the structural safety they provide.

API Ergonomics versus State Machine Complexity: When you rely heavily on signals, you push complexity into the workflow. The workflow must be designed to tolerate redundant signals, parse malformed payloads, and explicitly reject transitions that violate state rules—usually by recording an internal error state that a subsequent query can expose. When you use updates, you push complexity to the client interface. The workflow code remains pristine, handling only valid transitions, while the client must be prepared to catch specific validation errors and execute compensation logic.

You must design your update handlers to execute rapidly. While the update handler executes, the client's RPC request to the Temporal service remains blocked. If an update handler schedules an activity and blocks waiting for its result, and that activity takes thirty seconds to run, the client's network call will likely time out. The correct pattern is to mutate workflow variables immediately within the update handler,

return a success result to unblock the client, and allow the main workflow loop to observe the mutated state and schedule the long-running activity asynchronously. The update proves the request was accepted; it does not promise that downstream asynchronous work has finished.

5.2. Designing Workflow Interaction APIs: Stable Contracts and Evolvable State Machines

Treat your workflow as an API surface with a lifecycle far longer than any single deployment. Because workflow executions can span months or years, the shape of the signals, queries, and updates they accept forms a versioned contract. When you upgrade your worker code, in-flight executions created under older code must still process new inputs safely, and new clients must be able to interact with older in-flight executions.

Designing interaction surfaces for long-running processes requires you to think beyond simple method signatures. You are defining a state machine that must survive restarts, tolerate redundant inputs, and provide unambiguous responses to external callers.

To ground these concepts, consider the following Go implementation of an order workflow. It demonstrates an explicit command vocabulary, strict separation between validation and state transition, and deterministic state management using Temporal's Update API.

```
package workflows

import (
    "errors"
    "fmt"
    "go.temporal.io/sdk/workflow"
)

type OrderState string

const (
```

```

    StatePending      OrderState = "PENDING"
    StateAcknowledged OrderState = "ACKNOWLEDGED"
    StateCanceled     OrderState = "CANCELED"
)

// Request and response envelopes form the stable contract
type AcknowledgeRequest struct {
    OperatorID string
    Notes      string // Optional field, added safely without
                    breaking signatures
}

type AcknowledgeResponse struct {
    Success      bool
    Transitioned bool
}

var ErrInvalidState = errors.New("invalid state transition")

func OrderWorkflow(ctx workflow.Context, orderID string) (string,
    error) {
    currentState := StatePending

    // Register an Update handler with an explicit validator
    err := workflow.SetUpdateHandlerWithOptions(
        ctx,
        "AcknowledgeOrder",
        func(ctx workflow.Context, req AcknowledgeRequest) (
            AcknowledgeResponse, error) {
            // State transition (mutates workflow state
            deterministically)
            currentState = StateAcknowledged
            return AcknowledgeResponse{Success: true, Transitioned:
            true}, nil
        },
        workflow.UpdateHandlerOptions{
            Validator: func(ctx workflow.Context, req
            AcknowledgeRequest) error {
                // Validation (read-only, no side effects)
                if currentState != StatePending {
                    return fmt.Errorf("%w: cannot acknowledge from %s",
                    ErrInvalidState, currentState)
                }
                if req.OperatorID == "" {
                    return errors.New("OperatorID is required")
                }
                return nil
            },
        },

```

```
    },
  )
  if err != nil {
    return "", err
  }

  // Workflow blocks here, waiting for state changes driven by
  // external interactions
  err = workflow.Await(ctx, func() bool {
    return currentState == StateAcknowledged || currentState ==
    StateCanceled
  })

  return string(currentState), err
}
```

Defining the Command Vocabulary

Every signal, query, and update requires a defined payload shape. Always use an envelope object (such as a struct or a class) rather than a primitive type. If your signal takes a single string reason today, changing it later to accept both a reason and an operatorId will change the method signature. If you rely on generated code or strictly typed SDKs, altering the signature breaks compatibility with in-flight executions that expect the older shape. By using an envelope like `CancelRequest`, you can safely add new fields to the object over time.

Distinguish clearly between required and optional fields. Optional fields are not free; they create a long-term compatibility obligation. Your workflow code must be prepared to handle the absence of an optional field deterministically. Never use system clocks (e.g., `time.Now()` or `new Date()`) to backfill missing timestamps during payload parsing, as this injects non-determinism into the workflow execution. Instead, rely on `workflow.Now(ctx)` to apply time-based defaults, or force the upstream client to supply the timestamp.

Embed idempotency keys or correlation identifiers directly into your command payloads if the operation relies on them. While Temporal enforces idempotency for the transport of updates and signals—

meaning a retried request with the same underlying message ID will not execute twice—application-level deduplication often outlives the transport layer’s awareness. Passing an explicit `requestID` allows your workflow state machine to reject logically duplicate business commands even if they arrive as distinct network requests.

Separating Validation from Transition

The introduction of workflow updates fundamentally changed how interactions are modeled in Temporal. Unlike signals, which are appended to history asynchronously, updates provide a synchronous validation hook. This splits your interaction logic into two distinct phases: precondition checking and state mutation.

Configure your update handlers with an explicit validator. The validator runs against the current in-memory state of the workflow before the update event is written to the execution history. Because it executes before consensus, the validator must be completely pure. If you attempt to mutate workflow variables or execute random number generators inside a validator, you violate the SDK’s determinism contract. During a worker restart or workflow replay, the Temporal SDK will detect this side effect and throw a non-determinism error, potentially stalling the workflow. The validator’s only job is to evaluate whether the requested operation is legal given the workflow’s current state, and either accept or reject the payload.

Emit stable, machine-readable error codes from your validators. When a validator returns an error, Temporal immediately returns that error to the client, and the update is never recorded in the workflow history. Because clients will build retry and routing logic around these rejections, your error taxonomy forms part of the interaction contract. A rejection such as `ErrInvalidStateTransition` tells the client that retrying the same operation against the current state is futile. Conversely, if your validator rejects a payload due to a temporary internal policy

(e.g., an operational freeze window), the error code should instruct the client to queue or retry the command later.

Explicit State Machines for Workflows

Do not manage workflow lifecycle by sprinkling boolean flags like `isCanceled`, `isAcknowledged`, or `hasShipped` throughout your code. As workflows grow, ad-hoc flags inevitably combine into impossible composite states, making it difficult to determine which external interactions are legally permitted at any given moment.

Treat your workflow as a protocol endpoint. Define a closed set of explicit states and maintain a strict transition table. When a command arrives, consult the current state to determine if the transition is allowed. Implementing a transition table prevents defensive programming from cluttering your business logic. Instead of writing conditional checks inside every handler, your handler simply checks the matrix.

Current State	Update: Acknowledge	Signal: Ship	Signal: Cancel
PENDING	Transition to ACKNOWLEDGED	Reject (Invalid State)	Transition to CANCELED
ACKNOWLEDGED	Reject (Already Acked)	Transition to SHIPPED	Transition to CANCELED
SHIPPED	Reject (Already Shipped)	Reject (Already Shipped)	Reject (Too Late)
CANCELED	Reject (Canceled)	Reject (Canceled)	Ignore (Idempotent)

Table 5.1: *State Transition Matrix for an Order Workflow*

This design forces you to confront illegal transitions deterministically. With updates, invalid transitions are trivial to handle: the validator rejects them, the client receives the error, and the workflow history remains clean.

Signals, however, are fire-and-forget from the perspective of the workflow history; by the time your signal handler executes, the signal has

already been recorded. Because you cannot return an error to a signal caller, you must decide how the state machine reacts to an illegal signal. Silently ignoring invalid signals is a common anti-pattern that obscures client bugs. Instead, log a warning, increment a metric for invalid transitions, and optionally record a “signal rejected” marker in a dedicated workflow variable so that queries can surface the misbehavior. Never mutate core business state in response to an illegal signal.

API Evolution Strategies

As your business requirements shift, your interaction APIs will evolve. Because workflows are long-lived, you will frequently find yourself operating in a mixed environment: old workflows processing new commands, and new workflows processing old commands.

Apply backward-compatible additions cautiously. You can safely add new optional fields to a request object, but you must configure your client and worker serialization settings to tolerate unknown fields. If your JSON deserializer uses strict parsing (failing when it encounters unrecognized fields), sending a new payload shape from an updated client will crash older workers attempting to process the command for in-flight executions. Always prefer lenient parsing for workflow inputs.

Backward-compatible additions require careful handling of semantic defaults. When a client sends a payload missing a newly added optional field, the deserializer will populate it with the language’s zero value. Your workflow must detect this zero value and apply the correct semantic default deterministically. If the default behavior depends on the time the workflow processes the command, use `workflow.Now(ctx)`. Never default to external configuration values that might change between replays, as this will lead to divergent histories.

When you need to deprecate a signal or update, do not delete the handler from your workflow code. Removing a handler breaks deterministic replay for any closed or in-flight workflow that previously received

that command. Instead, guide clients away from the deprecated endpoint gradually. For updates, modify the validator to return a descriptive deprecation error, forcing clients to migrate. Keep the underlying handler structure intact until you are absolutely certain no active workflow histories contain the deprecated interaction.

To coordinate complex migrations across distributed clients, expose a capabilities query. Implement a standard query, such as `GetSupportedCommands` or `GetCapabilities`, that returns a list of features the current workflow execution understands.

```
err = workflow.SetQueryHandler(ctx, "GetCapabilities", func() ([]string, error) {
    // Return features based on the workflow's version or internal state
    return []string{"AcknowledgeOrder_v2", "CancelOrder"}, nil
})
```

Upstream systems can issue this query before sending a critical update, dynamically routing their requests to the older or newer command shape based on what the specific workflow execution supports. This decouples client deployments from worker deployments and provides a reliable mechanism for state machine feature flagging.

Decoupling Queries from Internal State

Do not couple the shape of your query responses to your workflow's internal state representation. A common mistake is to return the entire internal state object directly from a query handler. If you do this, any internal refactoring—such as renaming a field or normalizing a nested struct—will instantly break upstream clients expecting the old query shape.

Define dedicated query response types. Map your internal state to these response objects inside the query handler. When you change how the workflow stores data internally, you simply update the mapping logic in the handler to populate the stable response object. This

isolates clients from your internal engineering decisions and preserves the query contract.

Compatibility and Versioning Constraints

When standardizing on workflow updates, you must account for infrastructure compatibility. Updates rely on specific server-side capabilities and SDK implementations that evolved significantly in recent Temporal releases. Verify that your Temporal Server deployment meets the minimum version requirements for updates, and ensure that feature flags enabling update registries are active in your environment if required by your specific release line. Document these minimum versions for your operations team, and run contract tests against the precise server versions deployed in your production environment.

Treat every interaction endpoint as a published API. By combining explicit command shapes, pure validators, and strict state machines, you guarantee that your workflows remain robust, predictable, and fully evolvable over their entire lifecycle.

5.3. Implementing Update-With-Start: Atomic *Create If Absent* + *Mutate* Interaction

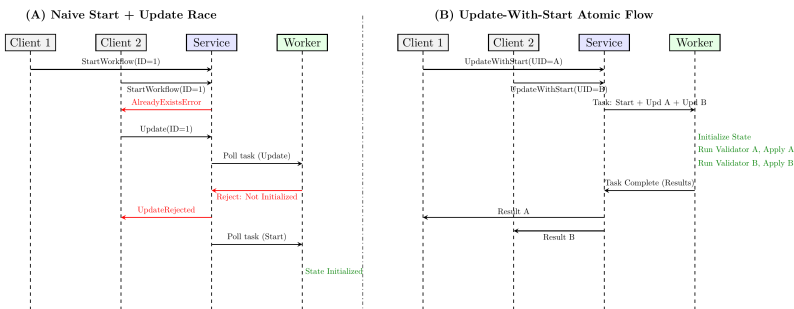
Distributed architectures frequently encounter a specific concurrency requirement: an external caller must ensure a stateful entity exists, apply a mutation to it, and receive a definitive acknowledgement. When multiple clients or upstream events attempt this simultaneously, they race to create the orchestration.

If you build this interaction using a naive sequence of `StartWorkflow` followed by `UpdateWorkflow`, you introduce an unavoidable race condition. Client A and Client B might both attempt to start the workflow. Client A succeeds. Client B receives a

`WorkflowExecutionAlreadyStartedError` and immediately sends an `UpdateWorkflow` request. However, if Client A's start request has not yet been processed by a worker—meaning the workflow code has not initialized its state or registered its handlers—Client B's update may be rejected or hang.

Using `SignalWithStart` resolves the creation race but introduces a semantic gap: signals are asynchronous. A client using `SignalWithStart` receives no synchronous validation failure if the payload is invalid, nor does it receive a return value indicating the business outcome of the signal. Retries on the client side can result in duplicate signals being appended to the workflow history.

`Update-With-Start` collapses these phases. It provides an atomic entry-point that guarantees the workflow is created if it does not exist, applies a strongly consistent update to the workflow, and returns the result of that update to the caller.



Implementing the Client Entrypoint

To execute an `Update-With-Start`, the client must furnish the parameters required to start the workflow alongside the parameters required for the update. The Temporal Service evaluates the workflow ID. If the workflow does not exist, the service uses the start parameters to create

it, then immediately applies the update. If the workflow already exists, the service ignores the start parameters and simply routes the update to the running execution.

Below is a Python SDK example of a unified entrypoint for a shopping cart. It adds an item to a cart, creating the cart orchestration if this is the user's first interaction.

```
import asyncio
from temporalio.client import Client, WorkflowUpdateHandle

async def add_item_to_cart(client: Client, user_id: str, item_id: str
    , quantity: int):
    # The workflow ID acts as the concurrency lock for this specific
    # user's cart
    workflow_id = f"cart-{user_id}"
    update_id = f"add-item-{item_id}"

    result = await client.execute_update_with_start(
        "CartWorkflow",
        id=workflow_id,
        task_queue="cart-task-queue",
        # Update details
        update="add_item",
        update_args=[item_id, quantity],
        update_id=update_id,
        # Start details (ignored if cart-user_id already exists)
        args=[user_id],
    )

    return result
```

This operational primitive is also exposed for manual intervention and operational tooling via the Temporal CLI:

```
temporal workflow update \
  --workflow-id cart-user123 \
  --name add_item \
  --input '"item-456"' --input '1' \
  --update-id add-item-456 \
  --start-workflow-type CartWorkflow \
  --start-task-queue cart-task-queue \
  --start-input '"user123"'
```

Atomicity Boundaries and the Update Lifecycle

When relying on Update-With-Start, you must rigorously understand its atomicity guarantees to avoid overpromising exactly-once end-to-end processing. The word “atomic” in this context applies to the workflow state transition, not to the external side effects or downstream activities.

Temporal processes updates through three strict lifecycle stages: ADMITTED, ACCEPTED, and COMPLETED.

When you issue an Update-With-Start, the Temporal server ensures the update is written durably to history (ADMITTED). If the workflow is created, the server packs both the `WorkflowExecutionStarted` and the `WorkflowExecutionUpdateAdmitted` events into the very first workflow task dispatched to the worker.

The worker executes the workflow’s initialization code, immediately followed by the update validator. If the validator passes, the update transitions to ACCEPTED. The worker then executes the update handler, which mutates the workflow state. Once the handler returns, the update is COMPLETED.

The client call to `execute_update_with_start` blocks until the update reaches a specified lifecycle stage—typically COMPLETED. This means the caller receives a definitive response confirming that the mutation was applied to the state machine. However, if the update handler schedules activities (e.g., charging a credit card), the update completes *before* the activities finish, unless you explicitly block the update handler on activity execution.

Blocking update handlers on long-running activities is an anti-pattern. It couples the client’s HTTP connection lifecycle to the activity’s execution time and external system availability. Instead, design your update handlers to mutate workflow state, schedule the activities in the background (or rely on the workflow’s main loop to observe the state change and schedule them), and return a state acknowledgement to the caller.

Client Design and Idempotency Strategies

Update-With-Start relies on two distinct identifiers to guarantee idempotent processing during network partitions and client retries: the Workflow ID and the Update ID.

The Workflow ID guarantees entity-level uniqueness. By mapping the external correlation key deterministically to the workflow ID (e.g., `cart-<user_id>`), you ensure that concurrent start requests do not create duplicate orchestrations.

The Update ID deduplicates the specific mutation. If a client receives a network timeout while waiting for the update result, it cannot know whether the server accepted the update, rejected the update, or never received it. The client should retry the identical `execute_update_with_start` call using the exact same Update ID. The Temporal server evaluates the Update ID against the workflow's history. If the server observes that this specific update has already been COMPLETED, it intercepts the request and instantly returns the cached result payload without re-executing the update handler on the worker.

You must generate Update IDs deterministically based on the business intent. If the client generates a random UUID for the Update ID on every retry, the Temporal server treats each retry as a novel business command, applying the mutation multiple times. Conversely, if you hardcode the Update ID, the client can only ever apply that mutation once for the lifespan of the workflow.

For example, when a webhook delivers a payment event, use the provider's message ID as the Update ID. When a user clicks "Add to Cart," use a combination of the session ID and the item ID.

Workflow-Side Structure and State Initialization

Workflows handling Update-With-Start require deliberate structuring

around initialization. Because the Temporal service ignores the start parameters if the workflow already exists, you cannot rely on the workflow's entry arguments to represent the total sum of the entity's state.

Instead, treat initialization as the establishment of an empty container. The workflow boots, sets up its internal data structures based on the start arguments, and then waits for updates to drive state transitions.

```
from temporalio import workflow

@workflow.defn
class CartWorkflow:
    def __init__(self) -> None:
        self.user_id = ""
        self.items = {}
        self.status = "OPEN"

    @workflow.run
    async def run(self, user_id: str) -> str:
        self.user_id = user_id
        # The main loop keeps the workflow open until checkout
        await workflow.wait_condition(lambda: self.status != "OPEN")
        return "Cart finalized"

    @workflow.update(name="add_item")
    def add_item(self, item_id: str, quantity: int) -> dict:
        self.items[item_id] = self.items.get(item_id, 0) + quantity
        return {"total_items": sum(self.items.values())}

    @add_item.validator
    def validate_add_item(self, item_id: str, quantity: int) -> None:
        if self.status != "OPEN":
            raise ValueError("Cannot add items to a closed cart")
        if quantity <= 0:
            raise ValueError("Quantity must be positive")
```

Notice what happens if the client sends an Update-With-Start with invalid arguments (e.g., a quantity of -1). The server creates the workflow, writes the start event to history, and dispatches the task. The worker initializes the workflow class and runs `validate_add_item`, which raises a `ValueError`.

The update is rejected, and the client receives an error. However, *the*

workflow continues to run. The start operation cannot be rolled back. The entity now exists in its initialized, empty state (`self.items = {}`). Your workflow design must account for this by ensuring that workflows can gracefully exist in an empty state or by implementing idle timeouts in the main loop to terminate orchestrations that receive no valid updates within a reasonable window.

Operational Flow and State Machine Coordination

Consider a complete end-to-end orchestration for order processing: `OrderOrchestration`. The goal is to create the order workflow and apply a `PaymentAuthorized` update atomically.

- **Ingestion:** The client receives a callback from a payment gateway. It derives the Workflow ID (`order-987`) and the Update ID (`pay-auth-abc`).
- **Submission:** The client calls `Update-With-Start`. The start arguments contain the basic order metadata, and the update arguments contain the authorization token.
- **Service Routing:** The Temporal Service checks the `order-987` execution. If absent, it creates the execution and writes the `Admitted` event.
- **Worker Execution:** The worker polls the task queue. It executes the order initialization, binding the order metadata to state variables. It then runs the `PaymentAuthorized` validator.
- **Transition & Application:** The validator confirms the order is not already paid. The update handler runs, changing the internal state machine from `PENDING` to `AUTHORIZED`, and returns the new status.
- **Acknowledgement:** The worker responds to the service, and the service releases the waiting client with the result.

- **Downstream Processing:** The workflow's main run method, which was suspended awaiting a state change to `AUTHORIZED`, unblocks and proceeds to schedule fulfillment activities.

This separation of concerns—updates for state mutation and the main loop for side-effect execution—preserves determinism and keeps update latency low.

Trade-offs and Common Pitfalls

Update-With-Start introduces specific trade-offs compared to asynchronous signals or simple start invocations.

Updates incur higher service and database load than signals because they require synchronous validation and bi-directional communication between the client, server, and worker. If you have an extremely high-throughput ingestion pipeline where strict entity creation guarantees are not required immediately (e.g., streaming telemetry data), raw signals are more appropriate.

A severe anti-pattern is attempting to dynamically change start parameters across retries or subsequent calls. If Client A calls Update-With-Start with start arguments `["user_A"]`, and Client B concurrently calls it with start arguments `["user_B"]`, whichever call arrives first dictates the immutable start parameters for that workflow execution. The Temporal server silently ignores the start parameters of the second call and only routes its update. If your application logic relies on the second call updating the baseline configuration, you must explicitly pass that configuration as part of the update payload, not the start payload.

Furthermore, do not embed initialization side effects (like external API calls) directly inside the workflow constructor or before the update handler executes. Workflows must remain deterministic. Any required external I/O to “hydrate” the workflow state must be modeled as an Activity. Since update handlers should not block on activities, hydra-

tion requirements generally necessitate accepting the update, writing the hydration request to internal state, and letting the main loop execute the hydration activity before accepting subsequent business mutations.

5.4. Building Callback-Driven Ingestion: Webhooks, Message Buses, Deduplication, and Ordering

External systems like webhook providers and message brokers operate on transport protocols that prioritize delivery over strict coordination. When consuming events from a billing platform, an inventory system, or a Kafka topic, you must assume at-least-once delivery, uncoordinated retries, and occasional reordering. Translating this chaotic stream into correct, exactly-once business logic requires treating the Temporal workflow as a durable, stateful inbox. Rather than building ad-hoc database tables to track message identifiers, sequence numbers, or processing timestamps, you embed ingestion rules directly into the workflow's state machine.

To ground these concepts, consider an HTTP handler that receives asynchronous events from a third-party webhook provider. The handler acts as a stateless translation layer, extracting identifiers and forwarding the event to a Temporal workflow using an atomic Update-With-Start operation.

```
func HandleBillingWebhook(w http.ResponseWriter, r *http.Request) {
    // Authenticate and parse the incoming webhook
    payload, err := io.ReadAll(r.Body)
    if err != nil {
        http.Error(w, "Bad request", http.StatusBadRequest)
        return
    }

    event, err := provider.ConstructEvent(payload, r.Header.Get("
        Signature"), secret)
```

```

if err != nil {
    http.Error(w, "Invalid signature", http.StatusBadRequest)
    return
}

// Derive a deterministic Workflow ID for correlation
workflowID := fmt.Sprintf("billing-customer-%s", event.CustomerID
)

// Issue an atomic Update-With-Start to route the event
updateOpts := client.UpdateWorkflowWithOptionsRequest{
    UpdateID: event.ID, // Leverage provider's event ID for
transport deduplication
    WorkflowID: workflowID,
    UpdateName: "ApplyBillingEvent",
    Args:       []interface{}{event},
    WorkflowOptions: client.StartWorkflowOptions{
        ID:           workflowID,
        TaskQueue: "billing-tasks",
        // Ensure late events do not silently overwrite completed
workflows
        WorkflowIDReusePolicy: enums.
WORKFLOW_ID_REUSE_POLICY_REJECT_DUPLICATE,
    },
}

// Execute the call with a strict context timeout
ctx, cancel := context.WithTimeout(r.Context(), 8*time.Second)
defer cancel()

updateHandle, err := temporalClient.UpdateWorkflowWithOptions(ctx
, updateOpts)
if err != nil {
    // A 503 instructs the webhook provider to back off and retry
later
    http.Error(w, "Service unavailable", http.
StatusServiceUnavailable)
    return
}

// Block until the workflow validates and acknowledges the
transition
var result ApplyEventResult
if err := updateHandle.Get(ctx, &result); err != nil {
    // Semantic validation failures return 400 to permanently
halt provider retries
    http.Error(w, err.Error(), http.StatusBadRequest)
    return
}

```

```
}  
  
    // 200 OK acknowledges the event and stops retries  
    w.WriteHeader(http.StatusOK)  
}
```

Choosing the Ingestion Primitive

When integrating an external event source, you must decide how the ingestion service communicates with Temporal. The primary decision is whether the external system requires synchronous validation feedback.

Webhook providers generally enforce a strict contract: you must return an HTTP 2xx status code to acknowledge the event, and HTTP 4xx or 5xx codes will trigger a retry schedule. If you use Temporal signals for webhook ingestion, the Temporal service acknowledges the signal as soon as it is appended to the workflow's history. The ingestion service immediately returns a 200 OK to the webhook provider. However, if the workflow logic subsequently evaluates the event and determines it violates a business rule, the webhook provider has already discarded the event. You lose the ability to push backpressure or semantic errors up the chain.

Updates invert this dynamic. By invoking an update, the ingestion service blocks until the workflow worker pulls the task, executes the update validator, and applies the state transition. If the event is malformed, the workflow rejects the update. The ingestion service catches this deterministic error and translates it into an HTTP 400 Bad Request, permanently halting the provider's retry loop. If the Temporal workers are saturated and the update times out, the ingestion service returns an HTTP 503, leveraging the provider's built-in retry mechanics to absorb the load spike.

While updates provide robust synchronization, they impose a higher throughput cost than signals. Processing an update requires the Tem-

poral service to maintain an open gRPC stream with the client, route the task, and wait for the worker's validation phase to complete before responding. If your ingestion pipeline processes tens of thousands of telemetry events per second from a message bus where individual message-level HTTP acknowledgements are unnecessary, this synchronous overhead becomes a bottleneck. In such high-throughput, low-latency scenarios, signals are the superior primitive. A signal durably injects the event into the workflow history in a fire-and-forget manner, allowing the ingestion service to immediately commit the broker offset.

Correlation Modeling and Routing

Before an event can be processed, it must be routed to the correct workflow execution. This requires a deterministic mapping from the external entity's identifiers to a stable Temporal Workflow ID. The webhook payload contains an identifier—like an Order ID, Customer ID, or Ticket ID—which you interpolate into a strict string format, such as `order-12345`.

Because you do not control the external system's timing, events often arrive before the local system has explicitly created the orchestration. A payment gateway might send a `payment_authorized` event milliseconds before your application logic issues a standard `StartWorkflow` command. If the ingestion service attempts to send an update to a non-existent workflow, the call fails.

The Update-With-Start pattern bypasses this race condition. By bundling the initialization parameters with the event payload, the Temporal service guarantees that exactly one workflow execution exists to receive the event. The workflow itself becomes the authoritative correlation boundary.

You must also account for events arriving after a workflow has completed. If an order is closed but a `refund_issued` webhook arrives a

week later, Temporal's default behavior allows a new workflow execution to start with the same ID. Depending on your business logic, this may be disastrous (silently resetting the order to a default state) or intended (spinning up a distinct refund-handling execution). Control this behavior explicitly using the `WorkflowIDReusePolicy`. Set it to `RejectDuplicate` if late events should bounce, or configure the workflow to remain open in a dormant state, utilizing `workflow.Await` to catch trailing events until an absolute duration expires.

State-Level Deduplication

External systems guarantee at-least-once delivery, meaning duplicate events are a certainty. A single network partition between your ingestion service and the webhook provider will cause the provider to resend the exact same payload.

Relying on the transport layer or the Temporal client to deduplicate is insufficient. While the Temporal SDK automatically deduplicates in-flight updates using the `UpdateID` (which you should populate with the provider's unique event ID), this cache is transient. Once an update completes, or if the ingestion service crashes and retries with a new request context, the Temporal cluster will route the duplicate event to your workflow.

True deduplication must be enforced within the workflow's state machine. Define what constitutes a duplicate based on the provider's semantics. Typically, the provider supplies an immutable Event ID. The workflow maintains a registry of processed identifiers and consults this registry during update validation.

```
type BillingState struct {
    ProcessedEvents map[string]bool
    // other domain state fields...
}

func (w *BillingWorkflow) ProcessEventUpdate(ctx workflow.Context,
    event provider.Event) (ApplyResult, error) {
    if w.state.ProcessedEvents[event.ID] {
```

```
        // Return a distinct status without returning an error.
        // Errors would cause the HTTP handler to return 4xx/5xx,
        // triggering more retries.
        return ApplyResult{Status: "IgnoredDuplicate"}, nil
    }

    // Apply the state transition
    w.state.ProcessedEvents[event.ID] = true

    // ... process business logic ...
    return ApplyResult{Status: "Applied"}, nil
}
```

Notice that a duplicate event does not return a Go error. If the workflow returns an error for a duplicate, the HTTP handler translates that into a failure response, prompting the webhook provider to retry endlessly. By returning a successful payload indicating the duplicate was safely ignored, the ingestion service returns a 200 OK, neutralizing the retry loop.

When defining deduplication rules, you must distinguish between transport retries and semantic updates. If a webhook provider resends an event due to a timeout, the payload is identical. However, some upstream systems emit updates for the same entity using the same Event ID but with mutated internal fields (like an updated `modified_at` timestamp). If your deduplication logic relies solely on the Event ID, the workflow silently discards the mutated payload. To prevent this, compute a deterministic hash of the payload's semantic fields and store the hash alongside the Event ID. If an event arrives with an existing ID but a novel hash, the workflow recognizes it as a legitimate semantic update rather than a transport duplicate.

Maintaining an unbounded `ProcessedEvents` map introduces a subtle failure mode: memory and workflow history bloat. Over months of execution, accumulating thousands of event IDs increases the payload size of state snapshots and slows down replay. You must enforce eviction rules. Because webhook providers maintain a finite retry window

(often 72 hours to a week), you only need to store an Event ID until that window expires.

Furthermore, a workflow that processes thousands of events per day will rapidly accumulate a massive event history. Temporal limits history size to maintain replay performance. To mitigate this, periodically truncate the history using `ContinueAsNew`. When calling `ContinueAsNew`, you pass the active `ProcessedEvents` map as a parameter to the new workflow execution. This resets the history length to zero while seamlessly preserving the deduplication barrier across the execution boundary.

Managing Out-of-Order Deliveries

Network topologies and concurrent dispatchers guarantee that events will occasionally arrive out of order. An `invoice.paid` event might reach your workflow before the corresponding `invoice.created` event. If the workflow assumes strict sequential ordering, it will either crash or reject the out-of-order event, causing data loss.

When consuming from message brokers like Kafka, ordering guarantees are strictly bound to partition keys. Kafka guarantees total order only within a single partition. If the upstream publisher misconfigures its partition routing, or if a topic rebalance temporarily splits event processing across multiple consumer instances, your ingestion service will dispatch events to Temporal out of order. Because Temporal workflows are keyed by Workflow ID rather than partition offsets, the workflow itself becomes the ultimate arbiter of ordering.

If the broker payload includes an incremental sequence number (e.g., an event version), the workflow enforces order mathematically, buffering future events until the missing dependencies arrive.

```
func (w *BillingWorkflow) ApplyEventUpdate(ctx workflow.Context,
    event provider.Event) (ApplyResult, error) {
    if event.Version <= w.state.CurrentVersion {
        return ApplyResult{Status: "IgnoredStale"}, nil
    }
}
```

```

    }

    if event.Version > w.state.CurrentVersion + 1 {
        // Future event; buffer it in state
        w.state.OutOfOrderBuffer[event.Version] = event
        return ApplyResult{Status: "Buffered"}, nil
    }

    // Apply the next sequential event
    w.applyEventLogic(event)
    w.state.CurrentVersion = event.Version

    // Recursively drain any buffered events that are now unblocked
    for w.state.OutOfOrderBuffer[w.state.CurrentVersion+1] != nil {
        nextEvent := w.state.OutOfOrderBuffer[w.state.CurrentVersion
+1].(provider.Event)
        w.applyEventLogic(nextEvent)
        w.state.CurrentVersion = nextEvent.Version
        delete(w.state.OutOfOrderBuffer, nextEvent.Version)
    }

    return ApplyResult{Status: "Applied"}, nil
}

```

This pattern isolates your business logic from underlying broker topologies. The broker can arbitrarily reorder deliveries, and the workflow state machine autonomously reassembles the correct sequence.

When sequence numbers are not available, utilize time-bounded buffering with `workflow.Await`. When a workflow needs an entity to be initialized before it can process subsequent mutations, decouple the ingestion of the event from the execution of the side effects. The interaction handler appends the out-of-order event to a local array and returns success. The main workflow routine pauses, waiting for the prerequisite state to resolve.

```

func (w *BillingWorkflow) Execute(ctx workflow.Context, input
WorkflowInput) error {
    // Main execution loop blocks until the invoice is created AND
    there are pending payments
    for {
        err := workflow.Await(ctx, func() bool {
            return w.state.InvoiceCreated && len(w.state.
PendingPayments) > 0

```



```
    })
    if err != nil {
        return err
    }

    // Drain the buffer and process payments
    for _, payment := range w.state.PendingPayments {
        err = workflow.ExecuteActivity(ctx,
            ProcessPaymentActivity, payment).Get(ctx, nil)
        // ... handle activity outcomes ...
    }

    w.state.PendingPayments = nil
}
}
```

This pattern is highly effective but introduces the risk of indefinite suspension. If the expected `invoice.created` event is permanently lost by the upstream system, the buffered payment will sit in memory forever. To mitigate this, wrap the wait condition in `workflow.AwaitWithTimeout`. If the timeout expires before the prerequisite event arrives, the workflow executes an escalation activity, such as alerting an operator or querying the external system's REST API to manually fetch the missing invoice data.

End-to-End Operational Flow and Backpressure

Operating a callback-driven ingestion pipeline requires aligning the timeouts and failure modes of three distinct layers: the external provider, the stateless ingestion service, and the Temporal workers.

Consider a traffic spike where a provider blasts thousands of webhooks concurrently. The stateless ingestion service scales horizontally to accept the HTTP requests, rapidly translating them into `Update-With-Start` calls. However, if the Temporal worker fleet processing the `billing-tasks` queue is underscaled, task execution latency increases.

Because update handlers block until the worker executes them, the ingestion service's calls to Temporal begin to pend. If the webhook

provider expects a response within 10 seconds, and the worker queue depth causes updates to take 15 seconds, the ingestion service will drop the HTTP connection and the provider will register a timeout.

To handle this elegantly, configure explicit context timeouts on the ingestion service's Temporal client calls. Set the timeout slightly shorter than the provider's HTTP timeout limit (e.g., 8 seconds for a 10-second webhook limit). When the context deadline is exceeded, the `UpdateWorkflowWithOptions` call returns a timeout error. The ingestion service intercepts this and returns an HTTP 503 `Service Unavailable` or 429 `Too Many Requests`.

Returning a 503 is critical. It signals to the provider that the payload is valid, but the system cannot currently process it. The provider applies exponential backoff, effectively turning its outbound queue into a shock absorber for your system. Once the Temporal workers drain the backlog, the ingestion service naturally resumes returning 200 OKs.

Conversely, rigorously manage poison messages—events that are semantically invalid and will never succeed. If an event payload cannot be unmarshaled, or if the update validator within the workflow returns a deterministic rejection (such as an invalid transition for the current state machine phase), the ingestion service receives a clear error from Temporal. The HTTP handler must translate this into an HTTP 400 `Bad Request`. Returning a 500 `Internal Server Error` for a validation failure is a severe anti-pattern; it communicates a transient failure to the webhook provider, causing them to retry the poison message until their maximum retry window expires, wasting bandwidth and masking real operational metrics.

By carefully modeling ingestion interactions as updates, enforcing state-level deduplication, buffering out-of-order deliveries, and translating workflow outcomes into precise HTTP semantics, you transform unreliable external event streams into a perfectly ordered, strictly con-

sistent workflow execution log.

5.5. Human-in-the-Loop Interactions

Human decisions—approvals, manual reviews, policy exceptions, and emergency interventions—introduce unbounded wait times and arbitrary state mutations into otherwise automated processes. Implementing these interactions by polling external databases for flag changes or relying on long-running HTTP connections inevitably leads to fragile control loops. Instead, Temporal allows you to encode human interventions directly into the workflow execution as durable, versioned interaction protocols.

Modeling Human Decisions as Workflow State

A human-in-the-loop process requires the workflow to halt progression until an external condition is met, while simultaneously exposing its current state to user interfaces. You achieve this by defining an explicit approval state machine within the workflow and exposing state transitions via workflow updates.

Consider an order fulfillment workflow that requires manual manager approval for high-value transactions. Rather than executing an activity that polls an external system, the workflow registers an update handler and pauses execution using a deterministic wait condition.

```
type ApprovalDecision struct {
    ApproverID string
    Action      string // "APPROVE" or "REJECT"
    Reason      string
}

type OrderWorkflowState struct {
    Status      string
    DecisionLog []ApprovalDecision
}

func HighValueOrderWorkflow(ctx workflow.Context, order Order) error
```

```

{
    state := &OrderWorkflowState{Status: "PENDING_APPROVAL"}

    // Register the update handler for human decisions
    err := workflow.SetUpdateHandlerWithOptions(
        ctx,
        "SubmitApproval",
        func(ctx workflow.Context, decision ApprovalDecision) error {
            state.DecisionLog = append(state.DecisionLog, decision)
            if decision.Action == "APPROVE" {
                state.Status = "APPROVED"
            } else {
                state.Status = "REJECTED"
            }
            return nil
        },
        workflow.UpdateHandlerOptions{
            Validator: func(ctx workflow.Context, decision
ApprovalDecision) error {
                if state.Status != "PENDING_APPROVAL" {
                    return fmt.Errorf("order is already %s", state.
Status)
                }
                if decision.ApproverID == "" {
                    return errors.New("ApproverID is required")
                }
                // Validate role/claims if passed in payload or
context
                return nil
            },
        },
    )
    if err != nil {
        return err
    }

    // Wait for the status to change, or timeout after 48 hours
    approvalTimeout := 48 * time.Hour
    notPending := func() bool {
        return state.Status != "PENDING_APPROVAL"
    }

    ok, err := workflow.AwaitWithTimeout(ctx, approvalTimeout,
notPending)
    if err != nil {
        return err
    }
}

```

```
    if !ok {
        // Timeout occurred: initiate escalation
        state.Status = "ESCALATED"
        return escalateApproval(ctx, order)
    }

    if state.Status == "REJECTED" {
        return performCancellation(ctx, order)
    }

    // Proceed with approved order...
    return fulfillOrder(ctx, order)
}
```

This structure models the approval as a rigorous protocol rather than a simple pause. The `ApprovalDecision` payload explicitly captures who made the decision, what the decision was, and why it was made. The `AwaitWithTimeout` API blocks the workflow deterministically without occupying a worker thread or network connection. During this wait, the workflow can continuously receive queries to serve the user interface, reflecting the `PENDING_APPROVAL` status.

Choosing Updates versus Signals for Decisions

When capturing human input, you must choose between signals and updates. Updates are almost always the correct primitive for access-controlled or state-dependent decisions because they provide synchronous validation.

If you implement the approval via a signal, a user interface submitting an invalid decision—such as an approval from an unauthorized user, or an approval for an order that was already rejected—will receive a successful acknowledgement from the Temporal service because the signal was successfully written to history. The workflow code would then silently discard the signal, leaving the UI out of sync and the user unaware that their decision was rejected.

Updates resolve this by running the validation function before append-

ing the event to history. In the example above, if a second manager attempts to approve an already-approved order, the update validator immediately returns a precise error (`order is already APPROVED`). The client receives this error synchronously, allowing the UI to display a meaningful message to the user. Signals should be reserved for asynchronous hints or supplementary data ingestion where the client does not require definitive confirmation that the workflow accepted the state transition.

Time Windows, Expirations, and Escalation

Human operators are inherently unreliable. They go off-shift, miss notifications, or simply forget to click buttons. Therefore, every human-in-the-loop interaction must enforce a strict time window. Unbounded waits eventually lead to forgotten workflow executions accumulating in the system, complicating operations and increasing storage costs.

To enforce deadlines, wrap the wait condition in a timeout mechanism. When the timeout expires, the workflow must take a decisive action: auto-approve, auto-reject, or escalate. Escalation typically involves transitioning the workflow into an `ESCALATED` state and executing an activity to page a secondary team or send high-priority alerts.

If you have a layered escalation policy—for instance, notifying a direct manager after 24 hours, and an executive after 48 hours—you can chain timeouts or compute deadlines dynamically based on the execution time. Because Temporal timers are durable and backed by the service’s scheduling infrastructure, they survive worker crashes and deployments. The UI can query the workflow to read the computed deadline and display a countdown to the user, ensuring the human operator and the workflow engine share the exact same understanding of time.

Interventions and Break-Glass Overrides

Beyond routine approvals, complex orchestrations frequently require operator intervention when external systems misbehave. An external API might report a persistent failure, or a downstream dependency might enter an unrecoverable state. In these scenarios, on-call engineers need a way to override workflow state without manually editing databases or forcefully terminating the execution.

You build these break-glass operations by defining explicit override commands in your interaction API. An override is a specialized update handler designed to bypass standard business logic while enforcing strict auditability.

```
type OverrideCommand struct {
    OperatorID string
    TargetState string
    ReasonCode string
    TicketURL string
}

// Inside workflow execution:
err := workflow.SetUpdateHandlerWithOptions(
    ctx,
    "ForceStateOverride",
    func(ctx workflow.Context, cmd OverrideCommand) error {
        // Record the override in the audit log
        state.DecisionLog = append(state.DecisionLog,
            ApprovalDecision{
                ApproverID: cmd.OperatorID,
                Action:      "OVERRIDE_TO_" + cmd.TargetState,
                Reason:       cmd.ReasonCode + " - " + cmd.TicketURL,
            })

        // Mutate the state machine
        state.Status = cmd.TargetState
        return nil
    },
    workflow.UpdateHandlerOptions{
        Validator: func(ctx workflow.Context, cmd OverrideCommand)
            error {
                if cmd.TicketURL == "" || cmd.OperatorID == "" {
                    return errors.New("overrides require OperatorID and
TicketURL")
                }
                if !isValidTargetState(cmd.TargetState) {
                    return fmt.Errorf("invalid override target: %s", cmd.
```

```
        TargetState)
    }
    return nil
},
},
)
```

By surfacing interventions through an update rather than an out-of-band database modification, the override becomes an immutable part of the workflow history. It respects the rules of determinism and replay. When the workflow resumes, it evaluates the overridden state exactly as if it had arrived there through standard operations.

Crucially, you must prevent override handlers from performing direct I/O or accessing external databases to verify permissions. Authorization logic—such as verifying if `OperatorID` belongs to the “On-Call” group—must happen upstream in your application API layer or gateway before the update is sent to Temporal. The workflow trusts the payload it receives, provided the payload passes the deterministic validation rules.

Auditable Trails and Forensics

When human operators alter the course of an automated process, reconstructing the timeline of intent becomes critical for compliance, security, and debugging. Workflow history inherently logs the arrival of every update and signal, including timestamps and raw payloads. However, treating the raw event history as your primary audit log is an anti-pattern. History events are optimized for execution replay, not for querying or compliance exports, and history is eventually deleted or archived based on your namespace retention policy.

Instead, build an explicit audit trail directly into your workflow state. Design payloads to require identity correlation (`OperatorID`, `Role`), context correlation (`TicketURL`, `SessionID`), and justification (`ReasonCode`). Every time a human-driven update mutates the state

machine, append a structured record to an internal `DecisionLog` array.

To make this audit trail accessible, expose it via a dedicated query handler.

```
err = workflow.SetQueryHandler(ctx, "GetAuditTrail", func() ([]
    ApprovalDecision, error) {
    return state.DecisionLog, nil
})
```

When a decision is finalized, you can also execute a fast-running activity to flush the internal `DecisionLog` to a permanent data warehouse or security information and event management (SIEM) system. This dual-layered approach uses the workflow memory to maintain the immediate, strongly consistent state of the interaction, while using activities to project that state outward for long-term retention.

For operational indexing—allowing support staff to locate workflows overridden by a specific engineer—you should bind human interaction events to Temporal Search Attributes. When an override update executes, the workflow can call `workflow.UpsertSearchAttributes` to write the `OperatorID` or `ReasonCode` into the visibility store. This makes it trivial to query the Temporal API for all workflows forcibly overridden by a specific user within a given time frame, tying the operational intervention directly to your observability and incident response systems. By treating human decisions as explicitly typed, carefully validated, and heavily audited workflow updates, you eliminate the split-brain problem between user interfaces and backend processes. The workflow remains the authoritative source of truth, waiting patiently and deterministically for humans to guide it.

Chapter 6

Operating Temporal in Production: Visibility, Schedules, and Tooling

Temporal's event history is the ground truth, but operators rarely have the time (or permissions) to read histories for everything-so production success depends on the “secondary systems” around it: visibility indexes, search attributes, metrics, traces, schedules, and CLI workflows. This chapter shows how to intentionally design those surfaces so that on-call can answer “what’s happening, to whom, and what do we do next?” in minutes, not hours.

6.1. Visibility versus Event History: Building Fast Listing and Operational Queries

Operating Temporal reliably requires a mental reset regarding how you query the state of your system. Operations demands two different “truths” operating at two different speeds. The first is Event History: the authoritative, replayable, audit-grade record of every state transition that occurred inside a workflow run. The second is Visibility: an indexed projection of workflow metadata designed specifically for high-cardinality listing, filtering, and operational dashboards.

Confusing these two subsystems is the root cause of many brittle operational runbooks. Operators often attempt to use Event History to answer fleet-wide questions, or they treat Visibility data as a flawless, real-time mirror of granular execution state. To build sustainable operational practices, you must map the questions you ask during an incident to the least expensive, most appropriate data source that can answer them reliably.

To ground this distinction, consider the primary tool operators use to inspect the system: the Temporal CLI. When triaging an incident, the initial operational query targets the Visibility store using a SQL-like List Filter:

```
temporal workflow list \  
  --namespace production-orders \  
  --query "WorkflowType = 'ProcessOrder' AND ExecutionStatus = '  
    Running' AND OrderState = 'PendingFulfillment'"
```

This command interrogates the secondary index (such as Elasticsearch, OpenSearch, or a SQL-backed visibility table) to rapidly return a bounded set of workflow executions matching your criteria. It does not touch the underlying event histories. The output provides a high-level summary of the matching executions:

6.1. VISIBILITY VERSUS EVENT HISTORY: BUILDING FAST LISTING AND OPERATIONAL QUERIES

WORKFLOW ID	RUN ID	START TIME
STATUS		
order-88392-xyz	123e4567-e89b-12d3-a456-426614174000	2026-10-14T08:12:0
0Z Running		
order-88394-abc	987fcdeb-51a2-43d7-9012-3456789abcde	2026-10-14T08:13:2
2Z Running		

Once you identify a specific, problematic workflow from this list, you shift contexts to Event History to diagnose the exact sequence of events that led to its current state:

```
temporal workflow show \  
  --namespace production-orders \  
  --workflow-id "order-88392-xyz"
```

This command fetches the authoritative, ordered log of events directly from the primary persistence store (e.g., Cassandra, PostgreSQL, or MySQL), bypassing the Visibility index entirely to provide forensic truth.

Mapping Operational Questions to the Right Source

An operational query in Temporal is a bounded, index-friendly search over visibility fields. It returns workflow execution summaries and key identifiers, not full histories. Knowing which source to query dictates the speed and safety of your incident response.

When operators ask, “Which workflows are stuck in a retry loop?”, or “Show all tenant X orders in state Y”, they are asking selection and cohorting questions. These questions require scanning thousands or millions of records. They must be routed to the Visibility store via List Filters leveraging custom Search Attributes (like TenantId or OrderState).

Conversely, when an operator asks, “Why did order #88392 fail exactly at the payment step?”, or “Did the third activity attempt receive the correct JSON payload?”, they are asking causal, forensic questions. These require the Event History.

The rule of thumb for Temporal operations is absolute: Visibility selects and shortlists; History explains and proves.

The Cost of Misaligned Queries

Failing to separate these concerns leads to two severe anti-patterns that degrade cluster stability and operator efficiency.

The first anti-pattern is treating Event History as a queryable database. Because Event History acts as the source of truth for workflow state, operators sometimes write scripts that pull down hundreds of event histories via the SDK or CLI, parse the JSON or protobuf payloads in memory, and search for specific failure patterns across the fleet. This is exceptionally dangerous. Primary persistence stores append histories in batches as binary blobs. They are optimized for high-throughput appends and single-execution point reads, not arbitrary scans. Reading thousands of histories to locate a handful of stuck workflows will exhaust primary database read capacity, trigger rate limits, and starve active worker fleets of the IOPS they need to progress healthy workflows.

The second anti-pattern is treating Visibility as a perfect, real-time mirror of History. Visibility is a projected view. While modern Temporal Server versions (v1.20+) offer SQL-based visibility that is highly consistent with primary persistence, deployments utilizing Elasticsearch or OpenSearch (required for advanced full-text search and very high cardinality prior to newer backend capabilities) operate with an index refresh interval. This introduces eventual consistency. If a workflow transitions to a new state and immediately fails, a visibility query executed in the same millisecond might not reflect the updated state. Relying on Visibility data to orchestrate transactional control flows or assuming it contains the absolute latest microsecond of execution progress creates false confidence and race conditions during automated remediation.

Furthermore, Visibility only surfaces fields that you explicitly configure it to index. If a piece of data is hidden deep within an Activity payload, it is invisible to operational queries unless explicitly extracted.

Designing the Minimum Viable Operational Record

Because Visibility only indexes what it is told to index, operational query design is fundamentally a schema design problem. You cannot wait until an incident occurs to decide how to search for impacted workflows. You must instrument a “Minimum Viable Operational Record” (MVOR) for each workflow type.

The MVOR consists of standard identifiers and explicitly promoted metadata that allow operators to slice the workflow population without falling back on history scans. Implementing the MVOR requires discipline at the workflow code level to emit stable Search Attributes.

A robust MVOR includes:

- *Stable Workflow ID Conventions:* Workflow IDs should be deterministic and semantically meaningful. A format like `[entity-type]-[tenant-id]-[unique-id]` allows operators to use the `STARTS_WITH` operator in List Filters to isolate resources rapidly.
- *State Labels:* Do not force operators to guess the current business state by inferring it from standard Temporal execution status (Running, Completed, Failed). Workflows should explicitly upsert a custom Search Attribute (e.g., `CustomBusinessState`) at major milestones.
- *Tenant and Region Identifiers:* In multi-tenant systems, blast radius containment is paramount. An indexed `TenantId` ensures you can isolate queries to a noisy neighbor without scanning the entire namespace.

- *Lifecycle Timestamps*: Beyond the default `StartTime` and `CloseTime`, exposing SLA deadlines or next-action timestamps as Search Attributes enables queries that identify workflows at risk of breaching compliance windows.

Consider how the State Label is maintained within the workflow code. Rather than relying on the timing of Activity executions, the workflow explicitly updates the Visibility store:

```
// Go SDK Example: Upserting an operational state label
func ProcessOrderWorkflow(ctx workflow.Context, order Order) error {
    // Promote the business state to Visibility for operational
    querying
    err := workflow.UpsertSearchAttributes(ctx, map[string]interface{}{
        "CustomBusinessState": "PendingValidation",
        "TenantId":             order.TenantID,
    })
    if err != nil {
        return err
    }

    err = workflow.ExecuteActivity(ctx, ValidateOrder, order).Get(ctx, nil)
    if err != nil {
        // Update visibility to reflect the failure state for
        incident triaging
        workflow.UpsertSearchAttributes(ctx, map[string]interface{}{
            "CustomBusinessState": "ValidationFailed",
        })
        return err
    }

    // Continue workflow execution...
    workflow.UpsertSearchAttributes(ctx, map[string]interface{}{
        "CustomBusinessState": "PendingFulfillment",
    })
    // ...
}
```

By explicitly modeling `CustomBusinessState` and `TenantId` as Search Attributes (after registering them in the Temporal cluster via the CLI), you empower the Visibility store to answer high-value operational

queries instantaneously.

List Filters and Query Validation

The syntax powering these queries is the List Filter. It supports a subset of SQL, including equality (`=`, `!=`), comparisons (`>`, `<`, `>=`, `<=`), boolean logic (`AND`, `OR`, parentheses), and operators like `BETWEEN`, `IN`, and `STARTS_WITH`.

Because Search Attribute names are case-sensitive and operator support can vary slightly depending on the visibility backend (e.g., Elasticsearch handles text tokenization differently than standard SQL backends), you must validate your operational queries in staging. Runbooks should not rely on operators crafting complex `AND/OR` chains under pressure. Instead, establish a canonical set of saved queries for your incident response teams.

For example, a validated runbook query to find high-value orders that have breached their internal SLA without failing out of the system might look like this:

```
WorkflowType = 'ProcessOrder'
  AND ExecutionStatus = 'Running'
  AND CustomBusinessState = 'PendingFulfillment'
  AND OrderSlaDeadline < "2026-10-14T12:00:00Z"
```

Note the time resolution constraints: while Elasticsearch backends historically resolve time to the nanosecond, SQL backends typically resolve to the microsecond. When crafting automated remediation scripts that query by `StartTime` or `CloseTime`, ensure your time boundaries account for these backend-specific precision floors.

The Repeatable Triage Loop

Once the distinction between Visibility and History is codified into your system's schema, incident response becomes a highly repeatable flow: List, Filter, Sample, Explain, Intervene.

The loop begins by *listing* a broad cohort to understand the scope of the anomaly. If alerting indicates a spike in task queue latency or a drop in fulfillment rates, operators first query the Visibility store by `WorkflowType` and `ExecutionStatus`.

Next, they *filter* the list down. Using the MVOR design, they append conditions like `CustomBusinessState = 'ValidationFailed'` to narrow the population to the exact cohort experiencing the failure. This yields the total blast radius: “We have 4,500 workflows stuck in the `ValidationFailed` state.”

With the cohort identified, the operator *samples* two to three executions from the Visibility results.

Only at this point do they pull the *Event History* for those specific exemplars. By examining the `ActivityTaskFailed` events or the unhandled workflow panics in those two or three histories, the causal mechanism becomes clear. They use history to prove the hypothesis—perhaps an external fraud detection API is returning a 503 `Service Unavailable`.

Finally, armed with the causal explanation and the exact bounds of the impacted cohort derived from the Visibility index, the operator decides the *intervention scope*. They can safely issue a batch command, targeting only the executions identified by the validated List Filter, to signal a retry, update the state, or terminate the runs. By relying on Visibility to define the targets and History to define the cause, operational interventions remain fast, targeted, and completely isolated from the performance risks of full-database scans.

6.2. Designing Search Attributes for Multi-Dimensional Observability and Routing

Search attributes transform the visibility store from a simple execution directory into a robust operational database. While workflow IDs and run IDs provide point lookups, search attributes enable set-based operations. They allow you to slice your running workflows across multiple independent dimensions, isolate noisy neighbors, measure SLA compliance, and execute targeted batch interventions. Treat search attributes as a strict production interface—a defined schema that sits at the boundary between your workflow code and your operational tooling.

Registering and using a search attribute involves configuring the Temporal cluster and then emitting the attribute from your workflow code. Because search attributes dictate how operators will interact with live systems, define them early in the deployment lifecycle.

```
# Register a custom search attribute at the cluster level
temporal operator search-attribute create \
  --name TenantId \
  --type Keyword \
  --namespace production

# Verify the registration
temporal operator search-attribute list --namespace production
```

Custom search attributes:
TenantId (Keyword)

Once registered, workflow code applies the attribute. Below is the mechanism in Go, demonstrating how a workflow declares its tenancy and initial operational state immediately upon starting.

```
func ProvisionTenantWorkflow(ctx workflow.Context, input
ProvisionInput) error {
  // Upsert attributes immediately to ensure visibility during
  early execution
```

```
attributes := map[string]interface{}{  
    "TenantId":      input.TenantID,  
    "BusinessState": "ProvisioningResources",  
}  
err := workflow.UpsertSearchAttributes(ctx, attributes)  
if err != nil {  
    return err  
}  
  
// ... workflow logic ...  
}
```

A Taxonomy of Operational Dimensions

Effective operational routing requires slicing workflow populations along distinct axes. Group your candidate search attributes into four functional categories: identity, topology, lifecycle, and debugging.

Identity attributes associate a workflow execution with a specific business entity. Standard examples include `TenantId`, `AccountId`, or `OrderNumber`. These fields are typically immutable for the lifetime of the execution. When defining identity attributes, strictly prefer the `Keyword` type over `Text`. The `Text` type subjects the value to tokenization by the underlying visibility store (like `Elasticsearch`). For example, a `Text` field containing `cust-456-abc` might be tokenized into `cust`, `456`, and `abc`, leading to false positives during wildcard queries. The `Keyword` type ensures exact-match semantics, making it the only safe choice for opaque UUIDs and system identifiers.

Topology attributes describe where the workflow logically runs and what constraints apply to it. Examples include `Region`, `DeploymentCell`, or `DataResidencyZone`. While `Temporal`'s built-in `TaskQueue` attribute handles worker routing, custom topology attributes allow operators to quickly quantify the blast radius of an infrastructure outage. If a specific database shard goes down, filtering by `DatabaseShardId` immediately isolates the affected executions.

Lifecycle attributes track the progression of the workflow against

business and operational milestones. The most critical of these is `BusinessState`. Instead of relying on raw workflow status (Running vs. Completed), `BusinessState` maps to specific operational phases like `AwaitingHumanApproval`, `PaymentPending`, or `ProvisioningHardware`. Another critical lifecycle attribute is `SlaClass` or `SlaDeadline`, which enables dashboards to query for workflows that are actively breaching their expected completion windows.

Debugging attributes are transient markers used during incidents or migrations. `IncidentTag` or `BuildIdCohort` allow operators to tag specific executions that were impacted by a bug. Once tagged, you can easily monitor these workflows through remediation or exclude them from standard SLA reporting.

Dimensional Modeling and Indexing Criteria

Not every internal variable deserves to be a search attribute. The visibility store is an operational projection, not a replica of your application database. Excessive indexing degrades visibility performance and increases storage costs. Apply four strict criteria when deciding whether to index a field: selectivity, cardinality, stability, and actionability.

Selectivity measures how effectively the attribute narrows down a population. A boolean attribute like `IsInternalCustomer` with a 99% to 1% distribution provides poor selectivity for the majority class but excellent selectivity for the minority class. Prioritize attributes that divide your workflows into operationally distinct cohorts.

Cardinality refers to the number of unique values the attribute will hold. High-cardinality fields (like `WorkflowId`, which is natively supported) require significant index space. Avoid indexing continuously shifting identifiers or high-entropy fields as custom attributes unless you have a concrete operational query that demands them.

Stability dictates the update frequency of the attribute. Calling `UpsertSearchAttributes` appends an `UpsertWorkflowSearchAttributes` event to the Event History. If a workflow uses a search attribute as a granular progress bar—updating it from 1 to 100 sequentially—it will bloat the execution history, increase history payload size, and cause write-thrashing in the visibility store. Restrict search attribute updates to coarse-grained phase changes. If you need a granular progress indicator, implement a query handler; do not use the visibility store.

Actionability is the ultimate test of a search attribute. If you can query an attribute but cannot envision a scenario where you would pause, cancel, signal, or route a remediation based on that query, the attribute is purely analytical. Analytical data belongs in an external data warehouse, exported via custom worker interceptors or the Event Export facility, not in the live operational visibility index.

Backend Mechanics and Version Constraints

The behavior of search attributes heavily depends on your Temporal Server version and the configured visibility backend.

If your cluster uses standard SQL visibility (MySQL or PostgreSQL), custom search attributes are scoped at the namespace level. However, SQL-based visibility imposes strict limits on the number of custom search attributes you can register, as each attribute typically maps to a column in the underlying relational schema.

If your cluster uses Elasticsearch or OpenSearch, custom search attributes are natively global across the cluster, even if you define them with a specific namespace in mind. This global scope creates a collision risk: if Team A uses a `State` attribute for integer-based state machine steps, and Team B registers `State` as a keyword for string-based labels, the index mapping will conflict. Enforce strict namespace prefixing for custom attributes in shared clusters, such as

`AcmeCorp_BusinessState`.

Version boundaries also dictate list semantics. In Temporal Server versions 1.20 and later, the `Keyword` type is strictly single-valued. If you need to attach an array of tags to a workflow (e.g., a list of affected resources), you must use the `KeywordList` type. Earlier server versions allowed arrays within standard `Keyword` fields, but this led to index mapping inconsistencies. Ensure your attribute registration scripts explicitly declare `KeywordList` for multi-valued fields.

Finally, search attributes bypass the standard Temporal payload codec encryption. Because visibility data must be read and indexed by the database engine (SQL or Elasticsearch), search attributes are stored in plain text. Never index Personally Identifiable Information (PII) such as email addresses, phone numbers, or unhashed account numbers. If you must search by a user's email, index a deterministic hash of the email address instead.

Schema Discipline and Anti-Patterns

Treating search attributes as unstructured storage severely damages operability. The most common anti-pattern is stringifying JSON blobs and dumping them into a `Text` attribute. This circumvents the schema, prevents efficient targeted queries, and places the parsing burden entirely on the operator's mental model during an incident. If an attribute contains a JSON object, decompose the actionable fields into distinct `Keyword` attributes and leave the remainder in the workflow payload.

Another anti-pattern is ambiguous state tracking. If `BusinessState` drifts from the actual control flow—for instance, if the workflow transitions to a rollback phase but fails to update the attribute-operators will execute runbooks against the wrong cohort of executions. Bind `UpsertSearchAttributes` directly to your state machine transitions. Ensure that any code path updating the logical state immediately updates the visibility state.

Avoid mixing internal and external identifiers in the same field. If you interact with a legacy billing system and a modern Stripe integration, do not use a single `BillingId` attribute. Create `LegacyBillingId` and `StripePaymentIntentId`. Explicit schemas prevent operational scripts from mistakenly sending Stripe commands to a legacy cohort due to an accidental ID collision.

Governance and Namespace Evolution

Once multiple dashboards, incident runbooks, and automated scripts depend on a search attribute, it becomes a frozen API. You cannot simply rename it without breaking downstream tooling. Implement a disciplined lifecycle for search attribute evolution at the namespace boundary.

When introducing a new operational dimension, begin with a design review focusing on naming consistency and cardinality. Register the new attribute via the cluster operator CLI or your Infrastructure-as-Code pipeline (e.g., Terraform).

If you are migrating from an old attribute to a new one (e.g., from `Status` to `BusinessState`), update the workflow code to dual-write both attributes.

```
// Dual-write during migration phase
attributes := map[string]interface{}{
    "Status":      "pending_approval",    // Deprecated, maintained
    for old dashboards
    "BusinessState": "AwaitingApproval",  // New standardized
    attribute
}
workflow.UpsertSearchAttributes(ctx, attributes)
```

Allow the dual-write to run until all long-running workflows utilizing the old schema have completed, or execute a batch update to backfill the new attribute onto open executions. Update your dashboards, alerting rules, and CLI runbooks to query the new attribute. Finally, remove the old attribute from the workflow code and submit a request to un-

register the attribute from the cluster.

Routing Interventions via the Control Plane

The true value of a disciplined search attribute schema emerges during large-scale operational interventions. Search attributes act as the control plane for routing signals, cancellations, and terminations to precisely defined cohorts of workflow executions.

Consider an incident where an external provisioning API begins returning 500 Internal Server Error responses, but only for the `us-east-2` region. Workflows fail their activities and enter a backoff-retry loop. Once the downstream API is restored, you do not want to wait for the exponential backoff to elapse naturally. You want to forcefully signal the affected workflows to retry immediately.

Because you implemented topology and lifecycle attributes, isolating the cohort is trivial. You construct a Temporal SQL-like visibility query:

```
WorkflowType = 'ProvisionResource'
  AND Region = 'us-east-2'
  AND BusinessState = 'Provisioning'
```

You can test this query safely using the standard list command to verify the blast radius before taking any destructive or mutative action.

```
# Validate the cohort size and sample the executions
temporal workflow list \
  --query "WorkflowType = 'ProvisionResource' AND Region = 'us-east-2' AND BusinessState = 'Provisioning'" \
  --limit 5
```

Once verified, you use the exact same query substrate to route an administrative batch command. This operation creates a batch job on the Temporal server that iterates over the visibility index and asynchronously routes the specified signal to every matching execution.

```
# Route a reset/retry signal to the isolated cohort
```

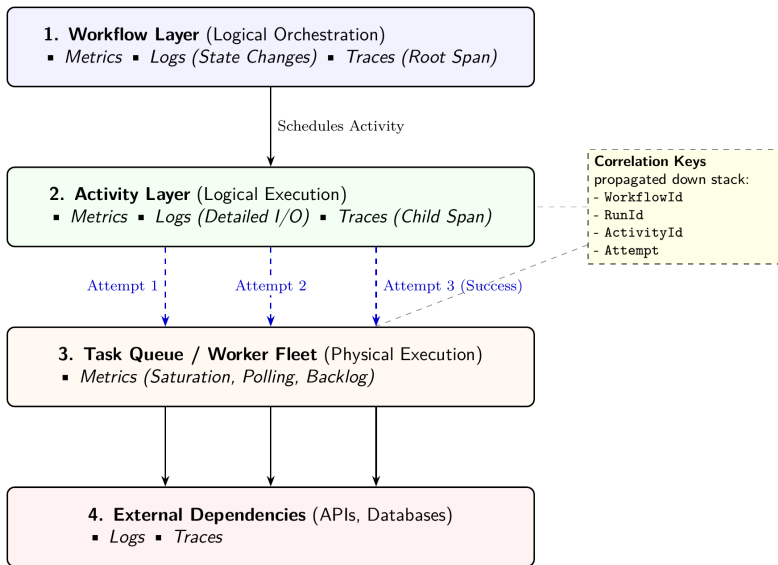
```
temporal batch start \  
  --query "WorkflowType = 'ProvisionResource' AND Region = 'us-east  
    -2' AND BusinessState = 'Provisioning'" \  
  --type signal \  
  --signal-name ForceRetrySignal \  
  --reason "Incident INC-1234 mitigation"
```

Batch job "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d" started.

This workflow intervention loop-query, sample, route, verify-relies entirely on the accuracy of the search attributes. If the Region attribute was missing, the operator would be forced to signal all provisioning workflows globally, potentially overwhelming the restored API. If BusinessState was not accurately maintained, the batch job might signal workflows that have already moved on to the billing phase, triggering out-of-order execution logic. Proper dimensional modeling ensures that when incidents occur, your routing and recovery actions are surgical, safe, and highly predictable.

6.3. Implementing Metrics, Logs, and Traces for Workflows and Worker Fleets

Production observability in Temporal is not about collecting every function call; it is about collecting the signals that change operational decisions. A well-instrumented Temporal deployment provides distinct layers of telemetry, allowing operators to separate infrastructure saturation from downstream dependency failures and business logic bugs. You must establish a strict observability contract defining what is measured at the worker fleet, task queue, activity, and workflow levels, and how to correlate those signals across process boundaries.



To capture actionable telemetry, you must explicitly configure your SDK to export metrics and propagate trace context. Temporal emits metrics locally within the worker process, but they require an exporter to reach your time-series database.

Configure your worker to expose a Prometheus endpoint while enforcing strict label boundaries to prevent cardinality explosions. In the Go SDK, you configure the telemetry interceptor as part of the `client.Options` when initializing your connection.

```

import (
    "go.temporal.io/sdk/client"
    "go.temporal.io/sdk/interceptor"
    "go.temporal.io/sdk/telemetry"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
)

func initializeTemporalClient(tracerProvider *sdktrace.TracerProvider
    ) (client.Client, error) {
    // Configure Prometheus exporter listening on port 9090

```

```

metricsHandler := telemetry.NewPrometheusReporter(telemetry.
    PrometheusReporterOptions{
        ListenAddress: "0.0.0.0:9090",
    })

// Initialize client with metrics and tracing interceptors
return client.Dial(client.Options{
    HostPort: "temporal.system.local:7233",
    Namespace: "payments",
    MetricsHandler: metricsHandler,
    Interceptors: []interceptor.ClientInterceptor{
        // Propagate OpenTelemetry headers seamlessly across task
        // queues
        opentelemetry.NewTracingInterceptor(opentelemetry.
            TracerOptions{
                Tracer: tracerProvider.Tracer("temporal-workers"),
            },
        ),
    },
})
}

```

Defining the Units of Health

Production observability breaks down into four specific units of health, each demanding a different measurement strategy.

First, cluster and namespace health dictate whether the Temporal server can accept and schedule work. Server-side metrics like `temporal_request_latency` and `temporal_request` (partitioned by operation) tell you if the control plane is functioning. As an application operator, you monitor these to rule out infrastructure-wide outages.

Second, task queue health measures the friction between scheduled work and actual execution. This is quantified by wait times.

Third, worker fleet health measures the saturation and failure rates of your compute layer. You track poller utilization, memory consumption, and task slot availability.

Finally, workflow-level health evaluates whether the business process

is meeting its Service Level Agreement (SLA). This layer abstracts away retries and transient errors, focusing entirely on end-to-end duration and terminal states.

The Schedule-to-Start Latency Signal

The most critical early-warning indicator in any Temporal deployment is queue latency, specifically `temporal_activity_schedule_to_start_latency` and `temporal_workflow_schedule_to_start_latency`.

Error rates are lagging indicators. If an external payment gateway degrades, your activities will eventually fail, but only after exhausting their defined timeouts. If a database query locks up, the resulting error might not surface until a 30-second context timeout expires. By contrast, queue latency reacts instantly. If your downstream dependencies slow down, worker threads block waiting for I/O. As threads block, the worker runs out of concurrent execution slots, and pollers stop pulling new tasks from the server. The moment polling stops, the `schedule_to_start` latency on the task queue begins to climb.

Create alerts based on the 95th percentile (P95) of schedule-to-start latency. A sudden spike indicates either a total worker crash or a severe downstream bottleneck saturating your concurrency limits.

```
# PromQL: P95 Activity Schedule-to-Start Latency by Task Queue
histogram_quantile(0.95, sum by (le, task_queue) (
  rate(temporal_activity_schedule_to_start_latency_bucket{namespace="
    payments"}[2m])
))
```

Worker Saturation and Poller Utilization

To distinguish between a compute shortage (you need more worker instances) and a downstream bottleneck (threads are stuck waiting on an API), monitor worker saturation using `temporal_worker_task_slots_available`.

When this metric approaches zero for activity slots, the worker is saturated. If saturation correlates with a drop in CPU utilization on the worker nodes, your activities are blocked on external I/O. Scaling the worker fleet in this scenario will merely generate more concurrent requests against an already-failing dependency, worsening the incident. Shed load or pause the affected schedules. Conversely, if task slots are depleted while worker CPU is pegged near 100%, you are compute-bound and should scale the fleet horizontally.

Correlating Across Retries

Retries severely complicate trace topology. A single logical activity execution (e.g., “ChargeCreditCard”) might represent five physical attempts spread across three different worker instances over several minutes.

To make retries visible without turning dashboards into unstructured noise, you must standardize correlation IDs. Temporal automatically provides `WorkflowID`, `RunID`, and `ActivityID`. When writing structured logs inside an activity, extract the `Attempt` counter from the activity context.

Do not aggregate alerts based on raw attempt volume. A transient network blip might cause 500 fast failures and retries within two seconds, skewing raw error rates despite zero user impact. Instead, aggregate metrics by the logical operation and only alert on terminal failures (when all retries are exhausted) or when the `Attempt` count exceeds a threshold that risks violating the business SLA.

When analyzing traces in tools like Jaeger or Datadog, Temporal’s OpenTelemetry interceptors automatically group retries. The root span represents the workflow run, child spans represent logical activities, and nested spans represent individual retry attempts. You can collapse the nested attempts to see the logical flow, or expand a long-running activity span to discover that it failed four times due to

ConnectionReset before succeeding on the fifth.

Telemetry Anti-Patterns and Cost Traps

The determinism constraints of Temporal workflows dictate strict rules for observability. Because workflow code is replayed to rebuild state after a worker crash or eviction, standard logging libraries will emit duplicate logs during replay. This creates a misleading operational narrative, implying that live execution is occurring when the system is merely catching up.

Always use the SDK's replay-aware logger inside workflow code. In Go, this is `workflow.GetLogger(ctx)`; in Java, it is `Workflow.getLogger()`. These wrappers inspect the workflow context and suppress log emission when `isReplaying` is true.

```
func ProcessOrderWorkflow(ctx workflow.Context, order Order) error {
    logger := workflow.GetLogger(ctx)

    // This log will only emit once, even if the workflow replays 10
    // times.
    logger.Info("Starting order processing", "OrderID", order.ID)

    // ANTI-PATTERN: Do not call Prometheus client directly here!
    // rawPrometheusCounter.Inc() // Will increment during replays!

    // Instead, use the SDK's scoped metrics handler:
    metricsHandler := workflow.GetMetricsHandler(ctx)
    metricsHandler.Counter("order_processing_started").Inc(1)

    // ... continue orchestration
}
```

A secondary anti-pattern is cardinality explosion. Metrics exporters in Temporal allow you to attach custom tags to SDK metrics. It is highly tempting to attach business identifiers—such as `tenant_id`, `order_id`, or `user_id`—to activity execution metrics. Because Prometheus and similar time-series databases allocate separate time series for every unique combination of label values, unbounded business IDs will rapidly exhaust database memory and inflate observability costs.

Use metrics exclusively for bounded, system-level dimensions: `task_queue`, `activity_type`, `error_class`, and `namespace`. When you need to slice operational data by high-cardinality business identifiers, use Search Attributes (as covered previously) and rely on the Visibility API rather than time-series metrics.

Constructing the Minimal Dashboard Set

Effective dashboards drive the triage loop. An operator on call should be able to scan four specific panels to isolate any Temporal incident:

- *Task Queue Backlog and Latency:* Visualizes `schedule_to_start` latency. This answers “Is work piling up?”
- *Worker Fleet Utilization:* Visualizes `worker_task_slots_available` and host-level CPU/Memory. This answers “Are workers starved for capacity or blocked on I/O?”
- *Activity Outcome Rates by Error Class:* Visualizes the rate of `temporal_activity_execution_failed` grouped by `error_type` (e.g., `TimeoutError`, `ApplicationError`). This answers “Which dependency is rejecting requests?”
- *Workflow SLA / End-State Distribution:* Visualizes end-to-end workflow duration (`temporal_workflow_execution_latency`) and custom business metrics marking logical completion. This answers “Is user experience actually impacted?”

If an incident triggers, the operational flow follows these panels sequentially. You detect a rise in queue latency (Panel 1). You identify the specific task queue backing up. You consult utilization (Panel 2) to determine if slots are available. If slots are exhausted, you look at activity outcomes (Panel 3) to see if a specific error class (like a downstream 503 Service Unavailable) is consuming all retries and satu-

rating threads. Armed with this isolation, you can confidently decide whether to scale the worker fleet, shed load by temporarily disabling a scheduled workflow, or declare a downstream dependency incident.

6.4. Operating Schedules: Server-Managed Cron, Safe Catch-Up, and Controlled Backfills

Traditional cron schedulers introduce a fragile dependency into distributed architectures. They fire triggers blindly, offer no native visibility into whether the previous job finished, lack replay capabilities for missed intervals, and fail to provide backpressure when downstream systems degrade. Temporal Schedules replace external cron daemons by promoting periodic execution to a first-class, durable server entity. Because a Schedule is managed by the Temporal Server independently of the workers, you can inspect its state, pause it, configure concurrency rules, and execute controlled historical backfills without modifying application code.

To establish a baseline for how Schedules are managed, consider the creation of a daily billing aggregation job via the Temporal CLI:

```
temporal schedule create \  
  --schedule-id "daily-billing-job" \  
  --type "BillingWorkflow" \  
  --task-queue "finance-queue" \  
  --interval 24h \  
  --overlap-policy BufferAll \  
  --catchup-window 1h \  
  --search-attribute "ScheduleName=daily-billing"
```

```
Schedule daily-billing-job created successfully.  
Next execution: 2023-10-25T00:00:00Z
```

This command defines an independent state machine on the Temporal Server. The Schedule evaluates time based on the cluster's

clock, and when the interval arrives, it issues a command to start the `BillingWorkflow`. It completely decouples the timing logic from the worker process. The worker only needs to listen to the `finance-queue` for the resulting workflow tasks.

Modeling Schedule Identity and Ownership

Treat schedules as independent operational primitives. A `Schedule` is identified by a unique ID within a namespace, separate from the workflow executions it creates. When designing schedule IDs, apply strict naming conventions that reflect domain ownership and frequency, such as `finance-billing-daily` or `infra-cert-rotation-weekly`. This enforces namespace hygiene and makes CLI introspection predictable.

A fundamental contract between a `Schedule` and the workflows it spawns is parameterization. Workflows started by a `Schedule` must not rely on the physical wall-clock time at the moment they execute. Queue latency, worker unavailability, or historical backfills guarantee that the actual execution time will frequently drift from the intended schedule time.

Instead, extract the scheduled time provided by the Temporal context. Temporal automatically injects metadata into the scheduled workflow, allowing your application logic to read the exact timestamp the execution was *supposed* to run. Use this logical timestamp to define data boundaries (e.g., querying a database for records created before the scheduled time) to ensure the workflow remains perfectly idempotent regardless of when the workers actually process it.

Managing Concurrency with Overlap Policies

When a `Schedule` fires, it must decide what to do if the workflow from the previous interval is still running. In a standard cron system, scripts either silently pile up and exhaust memory, or blindly abort based on

rudimentary lock files. Temporal handles this via explicit Overlap Policies, executed entirely on the server side before any worker is engaged.

Select the overlap policy that strictly matches your business use case:

- **Skip:** The default policy. If the previous run is active, the new run is dropped. Use this for stateless synchronization jobs where missing an interval is acceptable because the next successful run will eventually reconcile the system state.
- **BufferOne / BufferAll:** The Schedule queues the new start requests. When the active workflow completes, the buffered workflow is immediately started. `BufferAll` is essential for sequential, ordered data processing where no time window can be skipped. However, if your workers process runs slower than the schedule interval, `BufferAll` will cause unbounded backlog growth. Monitor the `schedule_buffer_overruns` metric to detect when your buffer exceeds its limits.
- **AllowAll:** The Schedule starts the new workflow concurrently alongside the existing one. Choose this only when you have strict data isolation guarantees—for instance, if the workflow is parameterized by the scheduled hour, and parallel executions will not cause database lock contention or duplicate side effects.
- **CancelOther / TerminateOther:** The Schedule forcefully terminates or cancels the currently running workflow and starts the new one. This is ideal for “latest state wins” polling jobs, where completing a delayed previous run provides no business value.

Handling Missed Intervals and Catch-Up Behavior

Operational realities dictate that intervals will occasionally be missed. This occurs primarily when a Schedule is manually paused during an incident or if the Temporal Server itself experiences downtime. Note that

if only your *workers* go down, intervals are not technically missed; the Temporal Server continues to evaluate the schedule and places workflow tasks into the task queue, where they wait safely until workers return.

Catch-up behavior specifically addresses intervals missed because the schedule evaluation itself was halted. The `catchup-window` defines how far back in time Temporal should look to retroactively execute missed triggers once the schedule resumes.

If a schedule with a one-hour interval is paused for 24 hours, and the `catchup-window` is set to 2 hours, resuming the schedule will instantly fire the 2 missed intervals from the past two hours, dropping the older 22 intervals. This bounds your operational blast radius. Without a catch-up window, an operator unpausing a long-dormant schedule could inadvertently unleash a flood of queued executions, crushing downstream dependencies. Set the catch-up window to match the maximum backlog your downstream database or API can absorb without rate-limiting.

Executing Controlled Backfills

Schedules natively support backfilling, which is the process of executing a schedule's logic over a specific historical (or future) time range as if time had naturally passed. Backfilling is heavily utilized for data reprocessing, disaster recovery, or populating new analytical models with historical data.

A backfill generates workflow starts for every interval within the specified time bounds. Because backfilling evaluates overlap policies just like natural execution, running a backfill with a `Skip` overlap policy usually results in only a single workflow running while the rest are instantly skipped. To ensure all historical periods are processed, you must explicitly override the overlap policy during the backfill operation to `AllowAll` or `BufferAll`.

```
temporal schedule backfill \  
  --schedule-id "daily-billing-job" \  
  --overlap-policy AllowAll \  
  --start-time "2023-09-01T00:00:00Z" \  
  --end-time "2023-09-30T23:59:59Z"
```

Executing a backfill safely requires careful planning to bound the blast radius. A 30-day backfill of an hourly schedule instantly generates 720 workflow executions. If you route these to your primary production task queue, you will immediately starve your real-time traffic, creating a massive latency spike across all operations sharing that worker fleet.

Apply the following staging procedures for safe backfills:

- **Isolate the compute pool.** If the backfill requires significant processing time, consider dynamically routing the backfilled executions to a dedicated task queue. Update the schedule configuration to target a `finance-queue-backfill` task queue, spin up a targeted worker fleet with strict concurrency limits, and execute the backfill. This isolates the historical load from real-time operations.
- **Tag the backfill runs.** Leverage visibility to maintain auditability. Update the schedule to inject a specific search attribute (e.g., `IncidentTag="backfill-ticket-123"`) before initiating the backfill. If the backfill logic proves flawed or corrupts data, you can use the CLI to query and bulk-terminate exactly those 720 executions without guessing which runs belonged to the real-time schedule and which were historical.
- **Stage the rollout.** Never execute a massive backfill across months of data in a single command. Run a dry run by backfilling a single hour. Verify the workflow completes successfully and produces the correct side effects. Then execute a canary backfill over a single day. Monitor your worker saturation and down-

stream dependency metrics. Only proceed with the full backfill once the canary proves your concurrency limits are holding.

Schedule Observability and Anti-Patterns

To safely operate schedules at scale, you must monitor both schedule-specific metrics and visibility states. Relying solely on workflow execution metrics obscures schedule health. A schedule might be actively failing to create workflows due to payload serialization errors, leaving no workflow execution history to inspect.

Monitor the `schedule_action_success` metric to guarantee schedules are successfully bridging the gap from schedule evaluation to workflow creation. Alert on `schedule_buffer_overruns` to detect when a `BufferAll` policy is accumulating executions faster than workers can process them.

Temporal automatically injects search attributes into workflows started by a `Schedule`. You can query your namespace for `TemporalScheduledById = "daily-billing-job"` to immediately filter your visibility views to runs originating from a specific schedule. Furthermore, querying `TemporalSchedulePaused = true` provides a fast operational dashboard of all schedules currently suspended across the namespace, ensuring that incident responders do not forget to unpause schedules after an outage is resolved.

Avoid two major anti-patterns when dealing with periodic execution in Temporal:

First, avoid the legacy `CronSchedule` field located on `WorkflowOptions`. In older Temporal deployments, cron strings were attached directly to the workflow start command. This legacy approach offers no explicit pause functionality, provides no backfill capability, and lacks granular overlap control. It also pollutes the workflow execution history with continuous `ContinueAsNew` events.

Always migrate legacy cron workflows to the top-level `Schedule` primitive to unlock explicit observability and lifecycle management.

Second, prohibit the use of infinite `sleep` loops within workflow code for periodic execution. While a workflow that runs a `while(true)` loop with a `Workflow.sleep(24h)` statement is technically durable, it is operationally hostile. Such workflows cannot be paused without injecting custom signals, they cannot be backfilled, and they accumulate massive histories that eventually degrade system performance. Schedule intervals belong in the server-managed `Schedule` configuration, not inside application control flow.

By treating Schedules as durable entities, strictly enforcing idempotency at the workflow boundary, and applying rigorous backfill staging, you convert time-based execution from a source of operational fragility into a highly auditable, deterministic production system.

6.5. Using the Temporal CLI for Local Iteration, Introspection, and Admin Runbooks

Production orchestration demands operational tools that are automatable, reviewable, and usable under incident pressure. The modern temporal CLI fulfills this role, unifying workflow operation, namespace administration, and local development into a single executable. Operating a Temporal cluster reliably relies on transitioning from ad-hoc debugging to deterministic, CLI-centered runbooks.

The `temporal` command-line interface directly consumes the visibility indexes, search attributes, and schedule configurations established in your production design. By scripting and standardizing CLI invocations, you create a verifiable audit trail for triage and intervention.

Establishing Local Dev Parity

Before writing production runbooks, validate execution logic and queries locally. The CLI includes an embedded development server that bundles the Temporal Server, a SQLite persistence layer, and the Temporal Web UI.

Start the local development server with file-backed persistence to test cross-restart behavior, schedule catch-ups, and long-running retries:

```
temporal server start-dev \  
  --db-filename ./temporal-dev.sqlite \  
  --namespace local-dev \  
  --ui-port 8080
```

By default, `start-dev` runs entirely in memory. Supplying the `--db-filename` flag forces SQLite to flush state to disk, allowing you to stop the server, modify workflow code, restart, and observe how your workers handle non-deterministic changes or pending schedule intervals. Note that this local SQLite file format is optimized for development speed and its schema is tightly coupled to the specific CLI version; do not attempt to share this database file across different CLI versions or use it in shared CI environments without ephemeral resets.

To avoid passing the target address and namespace to every command during an incident, define CLI environments. Environments prevent the catastrophic operator error of executing a command in production when staging was intended.

```
# Define a production environment context  
temporal env set prod.address temporal-api.production.internal:7233  
temporal env set prod.namespace core-billing  
temporal env set prod.tls-cert-path /etc/certs/temporal.pem  
temporal env set prod.tls-key-path /etc/certs/temporal.key  
  
# Use the environment  
temporal env use prod
```

Introspection: The Triage Loop

When an alert fires for queue latency or workflow failure, the triage loop begins with visibility filtering and ends with targeted introspection. Do not use the CLI to read full event histories across a large cohort; instead, use visibility queries to isolate a candidate, then describe its current state.

List executions matching your operational criteria using the `--query` flag. This leverages the search attributes and minimum viable operational records defined in your namespace:

```
temporal workflow list \  
  --query "WorkflowType = 'ProcessPayment' AND ExecutionStatus = '  
    Running' \  
      AND PaymentSLAStatus = 'AtRisk'" \  
  --limit 5
```

Once you identify a target `WorkflowId`, retrieve its current operational state using the `describe` command. The `describe` output skips the immutable event history and instead projects the immediate state of the workflow, including any pending activities, pending child workflows, and their exact retry states.

```
temporal workflow describe \  
  --workflow-id "pay-req-99482"
```

The output immediately highlights where the workflow is blocked:

```
{  
  "executionConfig": {  
    "taskQueue": {  
      "name": "payments-queue",  
      "kind": "Normal"  
    },  
    "workflowExecutionTimeout": "0s",  
    "workflowRunTimeout": "0s",  
    "defaultWorkflowTaskTimeout": "10s"  
  },  
  "workflowExecutionInfo": {  
    "execution": {  
      "workflowId": "pay-req-99482",  
      "runId": "a1b2c3d4-e5f6-7890-1234-56789abcdef0"  
    },  
    },
```

```

    "type": {
      "name": "ProcessPayment"
    },
    "startTime": "2023-10-25T14:32:01Z",
    "status": "Running",
    "historyLength": "14"
  },
  "pendingActivities": [
    {
      "activityId": "5",
      "activityType": {
        "name": "ChargeCreditCard"
      },
      "state": "Started",
      "heartbeatDetails": null,
      "lastHeartbeatTime": "2023-10-25T14:32:05Z",
      "lastStartedTime": "2023-10-25T14:32:04Z",
      "attempt": 4,
      "maximumAttempts": 10,
      "lastFailure": {
        "message": "Gateway timeout",
        "source": "PaymentGateway",
        "applicationFailureInfo": {
          "type": "NetworkError",
          "nonRetryable": false
        }
      }
    }
  ]
}

```

This output drives the intervention decision. The `pendingActivities` block reveals that the `ChargeCreditCard` activity is on its 4th attempt and failing with a `NetworkError`. Because the failure is `nonRetryable: false`, the operator knows the worker is actively retrying and the cluster is healthy; the external payment gateway is the fault domain.

Designing Deterministic Interventions

Once an issue is isolated, interventions such as signaling, canceling, or terminating executions must be executed safely. Bulk operations are natively supported by the CLI but carry significant blast-radius risks if the visibility query is too broad.

A cancellation is a cooperative request. It records a `WorkflowExecutionCancelRequested` event in the workflow history, giving the workflow code an opportunity to catch the cancellation exception, run cleanup activities (like reversing a partial ledger entry), and exit gracefully. Termination is a hard kill. It halts the workflow immediately, records a `WorkflowExecutionTerminated` event, and bypasses all application-level cleanup logic.

Always prefer cancellation over termination unless the worker fleet is actively saturated by an unbounded loop that cannot process the cancellation request.

Apply a targeted cancellation using a precise visibility query. Require a `--reason` flag so the intervention is permanently stamped into the event histories of the affected workflows:

```
temporal workflow cancel \  
--query "CustomIncidentTag = 'INC-1049' AND ExecutionStatus = '  
    Running'" \  
--reason "Canceling stuck cohort per INC-1049 instructions"
```

Operating Schedules via CLI

Schedules are durable entities managed by the Temporal Server. Unlike raw cron strings embedded in application code, schedules can be inspected, paused, triggered out-of-band, and backfilled. Administrative runbooks should manage schedule states explicitly during outages and data repair operations.

During a downstream outage, you may need to pause a schedule to prevent task queue saturation.

```
temporal schedule toggle --schedule-id "daily-report-gen" --pause \  
--reason "Database maintenance window"
```

When the maintenance window ends, simply unpausing the schedule resumes standard interval generation. However, if you need to execute the workflow for the intervals missed during the downtime, use the

backfill command.

Backfills explicitly generate workflow executions for past time windows. You must define the overlap policy to dictate behavior if a previous run is still executing. For data repair, `AllowAll` pushes all intervals onto the task queue, while `CancelOther` ensures older stalled runs are canceled before the newer interval begins.

```
temporal schedule backfill --schedule-id "daily-report-gen" \  
  --start-time "2023-10-20T00:00:00Z" \  
  --end-time "2023-10-25T00:00:00Z" \  
  --overlap-policy AllowAll
```

Monitor the execution of scheduled runs using the `describe` command, which details the recent actions and upcoming scheduled times:

```
temporal schedule describe --schedule-id "daily-report-gen"
```

```
{  
  "schedule": {  
    "action": {  
      "workflow": {  
        "workflowId": "report-gen-workflow",  
        "workflowType": { "name": "GenerateReport" },  
        "taskQueue": { "name": "reporting" }  
      }  
    },  
    "spec": {  
      "cronString": ["0 8 * * *"]  
    }  
  },  
  "info": {  
    "runningWorkflows": [],  
    "recentActions": [  
      {  
        "expectedTime": "2023-10-24T08:00:00Z",  
        "actualTime": "2023-10-24T08:00:00Z",  
        "startWorkflowResult": {  
          "workflowId": "report-gen-workflow-2023-10-24T08:00:00Z",  
          "runId": "c4d5e6f7..."  
        }  
      }  
    ],  
    "nextActionTimes": [  
      "2023-10-25T08:00:00Z",  
    ]  
  }  
}
```

```
    "2023-10-26T08:00:00Z"  
  ]  
}  
}
```

Scripting and Audit-Friendly Operations

Manual CLI execution is error-prone during high-severity incidents. Wrapping Temporal CLI commands in shell scripts standardizes preconditions, forces safety checks, and formats outputs for external logging systems.

When parsing output in a script, never rely on the default tabular or text representations, as these can shift between CLI versions. Always enforce JSON output using `--output json` or `-o json` and parse the results with tools like `jq`. Pin the CLI version in your execution environments to guarantee stable API interaction.

The following runbook script demonstrates a safe bulk termination pattern. It requires an explicit namespace, executes a count query to evaluate the blast radius, prompts the operator for confirmation, and limits the termination scope.

```
#!/usr/bin/env bash  
set -euo pipefail  
  
NAMESPACE="${1:-}"  
QUERY="${2:-}"  
REASON="${3:-}"  
  
if [[ -z "$NAMESPACE" || -z "$QUERY" || -z "$REASON" ]]; then  
    echo "Usage: $0 <namespace> <query> <reason>"  
    exit 1  
fi  
  
echo "Evaluating target cohort..."  
# Force JSON output to reliably extract the count value  
COUNT_JSON=$(temporal workflow count \  
    --namespace "$NAMESPACE" \  
    --query "$QUERY" \  
    -o json)
```

```

TARGET_COUNT=$(echo "$COUNT_JSON" | jq -r '.count')

if [[ "$TARGET_COUNT" -eq 0 ]]; then
    echo "No workflows match the specified query."
    exit 0
fi

if [[ "$TARGET_COUNT" -gt 1000 ]]; then
    echo "ERROR: Query matches $TARGET_COUNT workflows."
    echo "Bulk operations exceeding 1,000 executions require manual
        escalation."
    exit 1
fi

echo "WARNING: You are about to TERMINATE $TARGET_COUNT executions."
echo "Namespace: $NAMESPACE"
echo "Query: $QUERY"
echo "Reason: $REASON"
read -p "Type 'CONFIRM' to proceed: " CONFIRM_INPUT

if [[ "$CONFIRM_INPUT" != "CONFIRM" ]]; then
    echo "Aborting."
    exit 1
fi

echo "Executing bulk termination..."
temporal workflow terminate \
    --namespace "$NAMESPACE" \
    --query "$QUERY" \
    --reason "$REASON"

echo "Termination requests submitted. Verifying..."
sleep 2

POST_COUNT_JSON=$(temporal workflow count \
    --namespace "$NAMESPACE" \
    --query "$QUERY AND ExecutionStatus = 'Running'" \
    -o json)

POST_COUNT=$(echo "$POST_COUNT_JSON" | jq -r '.count')
echo "Remaining running targets: $POST_COUNT"

```

This scripted approach codifies operational policy. By restricting the batch size to 1,000, enforcing a reason string for auditability, and verifying the state post-execution, the tool shifts from a raw command executor to a reliable administrative guardrail. Store these scripts along-

side your infrastructure code, ensuring that the same version-control and code-review disciplines applied to worker deployments are applied to your operational tooling.

Chapter 7

Safe Evolution: Testing, Replay, and Worker Versioning

Temporal makes "shipping" fundamentally different: old executions keep running, and your new code must faithfully replay years of recorded history. This chapter teaches a practical, production-grade discipline for evolving workflows safely-treating history as test fixtures and evidence, and using worker versioning plus rollout mechanics to make upgrades boring.

7.1. Implementing replay-first test suites to prevent non-deterministic upgrades

Replay testing is the central engineering control for evolving Temporal workflows. You cannot effectively unit test determinism; instead, you

must continuously prove it against the recorded reality of past executions. A replay test asserts a rigid contract: given a fixed event history, your workflow code must make the exact same decisions and issue the exact same commands in the exact same sequence. If the code deviates, the SDK detects a state machine mismatch and raises a non-determinism error, mimicking the workflow task failure that would occur in production.

To prevent these failures from reaching live environments, you must treat workflow histories as your primary regression artifacts. Building a replay-first testing strategy requires curating a corpus of representative histories, designing a test harness that integrates tightly with your CI pipeline, and establishing disciplined practices for interpreting and fixing replay failures.

7.1.1. Mechanics of Replay Verification

At runtime, a Temporal worker executes workflow code until it yields (e.g., by scheduling an activity, setting a timer, or completing). The SDK translates these operations into Commands, which the Temporal Server validates and persists as Events in the execution's history. During a replay, the worker is fed this same event history. The SDK intercepts the workflow's operations and compares the newly generated Commands against the historically recorded Events. Any divergence—whether an extra timer, a missing activity, or even a modified activity timeout—violates determinism.

To embed this verification into your development lifecycle, you use the SDK's native replayer utilities. These utilities parse JSON-formatted event histories and route them through the exact same workflow registration process used by your production workers.

Below is a concrete example using the Go SDK to construct a replay

test harness. This test iterates over a directory of exported history files, verifies them against the current workflow implementation, and fails the build if any non-determinism is detected.

```
package workflows_test

import (
    "context"
    "os"
    "path/filepath"
    "testing"

    "go.temporal.io/sdk/worker"
    "github.com/stretchr/testify/require"

    "myapp/workflows"
)

func TestWorkflowReplay(t *testing.T) {
    replayer := worker.NewWorkflowReplayer()

    // Register the exact workflow implementations used in production
    replayer.RegisterWorkflow(workflows.OrderFulfillmentWorkflow)
    replayer.RegisterWorkflow(workflows.PaymentProcessingWorkflow)

    // Load history fixtures from the local testdata directory
    matches, err := filepath.Glob("testdata/histories/*.json")
    require.NoError(t, err)
    require.NotEmpty(t, err, "No history fixtures found")

    for _, match := range matches {
        t.Run(filepath.Base(match), func(t *testing.T) {
            historyFile, err := os.Open(match)
            require.NoError(t, err)
            defer historyFile.Close()

            // Replay the history against the registered workflows
            err = replayer.ReplayWorkflowHistoryFromJSONFile(nil,
                match)

            // A non-nil error indicates a determinism violation
            require.NoError(t, err, "Determinism failure on history:
%s", match)
        })
    }
}
```

This harness is intentionally simple but provides a strict verification gate. By forcing all workflow code changes to pass through `ReplayWorkflowHistoryFromJSONFile`, you shift determinism checks to the developer's local machine and the initial CI pull request.

7.1.2. Curating the History Corpus

A replay harness is only as effective as the histories it consumes. Overfitting to a few trivial executions provides a false sense of security. You must curate a history corpus that spans the operational reality of your system, capturing diverse code paths, prolonged retry loops, and complex interleavings of external interactions.

Acquiring histories

Histories are extracted directly from the Temporal cluster. During active development or incident triage, you can export a specific execution's history using the Temporal CLI. The `--output json` flag formats the event sequence exactly as the SDK replayer expects:

```
temporal workflow show \  
  --workflow-id "order-7382" \  
  --run-id "a1b2c3d4-e5f6-7890-1234-56789abcdef0" \  
  --output json > testdata/histories/order-7382.json
```

For automated corpus building, use the SDK's client API to query closed workflows and download their histories programmatically. An automated nightly job can query executions matching specific search attributes or failure states, sample them, and commit them to your corpus repository.

Stratification and sampling

Do not attempt to store every production history; the storage overhead and test execution time will quickly become unmanageable. Instead, sample across distinct execution profiles:

- *Happy paths*: Standard executions that complete without retries or timeouts, establishing the baseline command sequence.
- *Deeply nested failures*: Executions containing activity failures, heartbeat timeouts, and extensive retry loops. These validate that error handling paths remain deterministic.
- *Signal and Update interleavings*: Executions where signals were received out of order, immediately prior to workflow completion, or simultaneously with activity completions. Concurrency in workflow state machines is a frequent source of accidental non-determinism.
- *Long-lived boundaries*: Executions captured immediately before and after a Continue-As-New transition. These ensure that the state carried across run boundaries via the Continue-As-New arguments remains compatible with the new code.

Sanitization and sensitive data

Histories contain the raw payloads of workflow inputs, activity results, and signals. If your system handles Personally Identifiable Information (PII) or sensitive financial data, dumping raw production histories into a Git repository violates security policies.

Address this by leveraging Temporal's Data Converter framework. If your production workers use a custom Data Converter to encrypt payloads before they leave the worker, the exported JSON histories will only contain ciphertexts. To successfully replay these histories in CI, your test harness must register the identical Data Converter and have access to the necessary decryption keys. Alternatively, you can generate synthetic histories by driving test workflows in a dedicated, isolated staging namespace and exporting those safe histories for the CI corpus.

7.1.3. Designing a Resilient Test Harness

An effective replay test suite must minimize false positives while ruthlessly exposing actual determinism breaks. This requires mirroring the production worker environment as closely as possible within the test harness.

Pinning execution configurations

Replay tests can fail if the test environment handles serialization differently than production. Ensure your replay harness registers the same custom payload converters, failure interceptors, and workflow options. If your production workers run on specific SDK versions, pin those versions in your CI environment. Changes in the underlying SDK's command generation logic or internal state machine can occasionally alter replay behavior, so SDK upgrades should be tested against your corpus before being deployed.

Stratifying test execution

Running thousands of replays on every commit creates friction. Split your test suite into two tiers:

- *The PR Smoke Gate*: A curated set of 20 to 50 highly representative histories that run in seconds on every pull request. This catches obvious determinism violations immediately.
- *The Nightly Regression Corpus*: A massive collection of historical executions run asynchronously every night. This acts as the final release gate, ensuring that subtle edge cases introduced by recent merges do not compromise in-flight workflows.

7.1.4. Taxonomy of Replay Failures

When a replay test fails, the SDK outputs a non-determinism error detailing the divergence. Typically, this takes the form of a mismatched command. For example, the history might expect a `ScheduleActivityTask` command, but the worker code generated a `StartTimer` command instead. Understanding the root cause of these mismatches dictates how you repair the code.

Accidental use of wall-clock time

Workflow code must evaluate time consistently across replays. If a developer uses language-native time functions (e.g., `time.Now()` in Go, or `Instant.now()` in Java), the workflow will generate a different timestamp during replay than it did in production. *The fix:* Replace native time calls with SDK-provided deterministic time functions, such as `workflow.Now(ctx)`. The SDK caches the timestamp of the workflow task that originally generated the event, guaranteeing that the workflow observes the exact same time during replay.

Unsafe randomness

Generating a random UUID or executing a randomized backoff within workflow code causes branch divergence during replay. If a random value determines which activity to call, the replay will fail when the newly generated random value routes the workflow down an unexpected path. *The fix:* Use the SDK's deterministic random number generators (e.g., `workflow.GetSideEffect()` or `workflow.GetRandom()`). Alternatively, generate the random values inside an activity and return them to the workflow, as activity results are durably recorded in the history and skipped during replay.

Map iteration order

Languages like Go randomize map iteration order by design. If you

iterate over a map of items and schedule an activity for each, the sequence of `ScheduleActivityTask` commands will vary across runs. If the history recorded scheduling Activity A then Activity B, but the replay schedules Activity B then Activity A, the SDK will flag a mismatch. *The fix:* Extract the map keys, sort them deterministically, and iterate over the sorted slice. This guarantees the exact same sequence of commands is generated every time.

Environmental dependency injection

Injecting external dependencies—such as database clients, feature flag SDKs, or network calls—directly into workflow code is a critical anti-pattern. If a feature flag toggles a code path, and that flag changes state between the initial execution and the replay, the workflow will diverge. *The fix:* Move all interactions with external state into activities. If you must evaluate a feature flag to decide workflow control flow, do so within an activity that returns a boolean. The boolean result is recorded in the history, locking in the decision for all future replays.

Changes in activity scheduling decisions

Not all determinism failures stem from language anti-patterns. Often, they result from legitimate application evolution. Modifying an activity's `ScheduleToClose` timeout, changing the target Task Queue, or altering the retry policy changes the parameters of the `ScheduleActivityTask` command. The SDK compares the command generated by the new code against the command recorded in the history. Because the parameters differ, the SDK rejects the command. *The fix:* Recognize that modifying scheduling parameters is a breaking change for in-flight executions. To repair this, you must explicitly version the workflow code to apply the new parameters only to new executions, while preserving the old parameters for historical replays.

7.1.5. Repair Strategies and Actionable Fixes

When a replay test fails due to intentional application changes—such as adding a step to a business process or modifying an activity timeout—you cannot simply “fix” the test by deleting the offending history fixture. Deleting the fixture masks the problem; the in-flight executions represented by that fixture will still fail in production. Instead, you must alter your workflow code to handle both the historical reality and the new behavior safely.

To safely introduce non-deterministic changes, use the SDK’s versioning APIs (often called patching). Patching introduces stable branching into your workflow code by recording a marker in the event history the first time a specific code path is evaluated.

When you introduce a new activity, you wrap the change in a version check. During replay, the SDK detects whether the version marker exists in the history. If the history is from an old execution that predates the change, the SDK forces the workflow down the legacy code path, bypassing the new activity and maintaining determinism. If the history is from a new execution, the SDK routes it down the new code path. This ensures that a single worker binary can deterministically replay years-old histories while simultaneously executing the latest logic for newly started workflows.

Once the in-flight executions bound to the older logic have completed or reached their retention limits, you can deprecate the legacy code paths and remove the version markers. The replay test suite serves as the arbiter of this cleanup process: if you remove a patch too early, the nightly regression corpus will fail on the older histories, preventing a dangerous deployment.

7.2. Running incident investigations using workflow history as primary evidence

When an execution misbehaves, you do not start by tailing logs or guessing at concurrency conditions. You start by exporting the exact event sequence that the Temporal cluster recorded. This history is the authoritative forensic record of the execution's lifecycle, stripping away the noise of distributed system timing and providing a deterministic timeline of what actually occurred.

Begin your investigation by retrieving the execution's state and exporting its history as a JSON artifact:

```
# Export the complete event history for local replay and analysis
temporal workflow show \
  --workflow-id "billing-run-9381" \
  --output json > history.json

# Retrieve current status, pending activities, and the worker build
  ID
temporal workflow describe \
  --workflow-id "billing-run-9381"
```

Analyze the output of the describe command. It provides the `workerId` (typically a combination of hostname and process ID) and the `buildId` (if worker versioning is active). Use these identifiers to filter your external log aggregation tools (e.g., Elasticsearch, Datadog) to the precise worker instance that executed the failing task. Do not search globally for the workflow ID; search specifically for the `workerId` combined with the `ActivityId` or the attempt number extracted from the describe output.

Triaging the Failure Class

Categorize the failure by inspecting the tail end of the history or the pending state output. Your immediate goal is to classify the incident into one of three distinct failure domains:

- *Workflow task failures:* The control plane rejected the state transition. Look for a repeating cycle of `WorkflowTaskScheduled`, `WorkflowTaskStarted`, and `WorkflowTaskFailed`. This indicates that the workflow code itself is crashing due to an unhandled panic, or it is violating determinism by generating commands that contradict the previously recorded history.
- *Activity failures:* The data plane is failing. The history shows `ActivityTaskScheduled` followed by `ActivityTaskFailed` or `ActivityTaskTimedOut`. The workflow logic is correct, but the external side effect cannot complete successfully.
- *Stuck executions:* The workflow is suspended indefinitely. The history ends cleanly with a `WorkflowTaskCompleted`, but no activities or timers are pending. The execution is blocked on a condition or waiting for an external signal that never arrived.

Disciplined Causality: Reading the Event Ladder

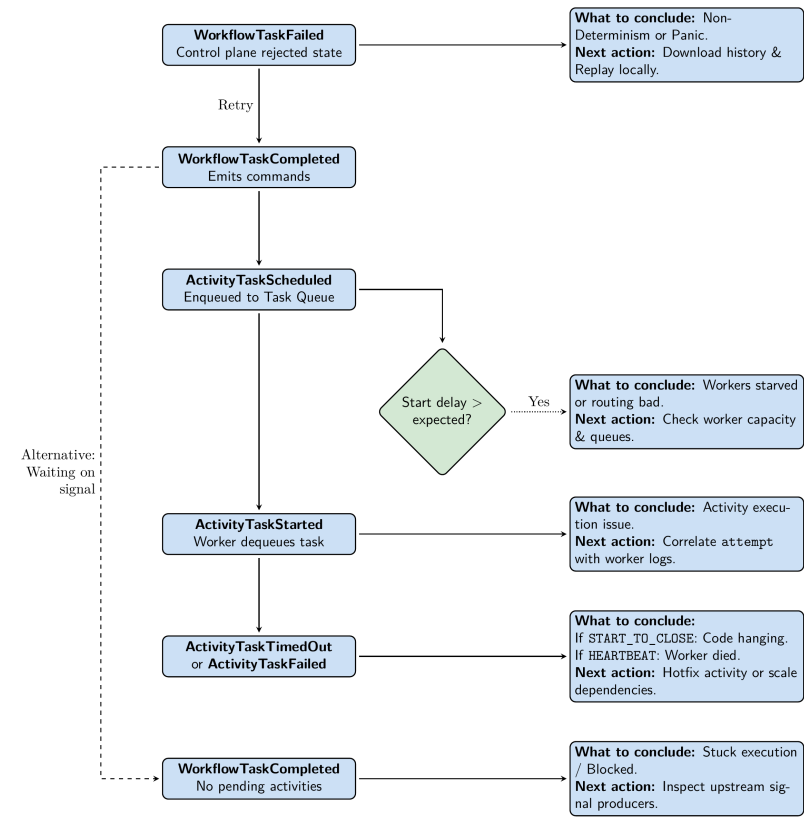
Read the JSON history file as a causal chain. Each event represents a discrete state transition accepted by the cluster. By isolating specific event sequences, you decouple workflow decisions from activity side effects.

A workflow execution progresses through evaluations called workflow tasks. A standard state transition appears as a triplet: `WorkflowTaskScheduled`, `WorkflowTaskStarted`, and `WorkflowTaskCompleted`. The `WorkflowTaskCompleted` event is the critical inflection point; it confirms that the worker successfully advanced the state machine and returned commands to the cluster, such as `ScheduleActivityTask` or `StartTimer`.

Examine the latency between events to diagnose infrastructural bottlenecks. The time difference between `ActivityTaskScheduled` and `ActivityTaskStarted` dictates queue latency. The scheduled event

records the exact millisecond the cluster placed the task on a Task Queue. The started event records when a worker successfully dequeued it. If this gap stretches from milliseconds to seconds or minutes, you are dealing with worker starvation, capacity exhaustion, or queue routing misconfigurations—not a bug in the activity code itself.

Inspect the attempt fields within activity execution events. When an activity fails, Temporal increments the attempt counter and schedules a new execution based on your configured retry policy. By correlating the `ActivityTaskScheduled` event ID and attempt number with your application logs, you isolate the exact environment and execution context that produced the failure.



Identifying Telltale Event Patterns

You can rapidly diagnose common failure modes by recognizing their specific event signatures:

The Poison Pill Workflow: A **WorkflowTaskFailed** event with a cause of **WORKFLOW_TASK_FAILED_CAUSE_NON_DETERMINISTIC_ERROR** means a worker running a newer, incompatible version of the workflow code attempted to process a history generated by an older version. The cluster correctly rejected the state transition, dropping the workflow task

to prevent state corruption.

The Hanging Activity: A sequence where `ActivityTaskStarted` occurs on time, but is repeatedly followed by `ActivityTaskTimedOut` with a `timeoutType` of `TIMEOUT_TYPE_START_TO_CLOSE`. The worker picked up the task but never reported completion or failure. This points to a deadlock in the activity code, a blocked network call missing a client-side timeout, or a worker crash (such as an Out-Of-Memory kill) that prevented it from emitting a failure back to the cluster.

Non-Retryable Escapes: An `ActivityTaskFailed` event where the failure payload contains an application-level error type that your retry policy was supposed to ignore, yet the history shows a subsequent `ActivityTaskScheduled` event. This indicates a string-matching error between the error type returned by the activity and the exact type specified in the `NonRetryableErrorTypes` configuration.

Idempotency Failures: The history shows an `ActivityTaskCompleted` event, meaning Temporal recorded the activity as successful and progressed the workflow. However, downstream systems report duplicate database inserts or double-charging. This reveals that a previous attempt of the activity executed the side effect successfully but failed to report completion to the cluster (often due to a network partition), prompting a retry. The fault lies in the activity's lack of idempotency keys, not the workflow engine.

Reproducing the Issue via Replay

When investigating workflow task failures or non-determinism, rely on the SDK's replay testing facilities rather than deploying speculative fixes. Replay runs the downloaded JSON history against your local workflow code, allowing you to step through the exact execution path in a local debugger.

Use the replayer in your local test environment, passing the exported

history file:

```
package forensic_test

import (
    "testing"
    "go.temporal.io/sdk/worker"
    "mycompany/billing/workflows"
)

func TestIncidentReplay(t *testing.T) {
    replayer := worker.NewWorkflowReplayer()

    // Register the exact workflow type involved in the incident
    replayer.RegisterWorkflow(workflows.BillingCycleWorkflow)

    // Replay the history captured from production
    err := replayer.ReplayWorkflowHistoryFromJSONFile(nil, "history.
    json")
    if err != nil {
        t.Fatalf("Replay failed: %v", err)
    }
}
```

When a replay fails due to non-determinism, the SDK surfaces a highly specific error detailing the exact divergence between the recorded history and the commands generated by the current code path.

```
Replay failed: nondeterministic workflow: history event is of type
ActivityTaskScheduled for activity 'ChargeCustomer', but workflow code
generated TimerStarted
```

This output immediately isolates the bug: the historical record dictates that the workflow scheduled the `ChargeCustomer` activity next, but the current code attempts to start a timer (e.g., a `workflow.Sleep` call was introduced before the activity).

To pinpoint the origin of a regression, bisect your version control history. Replay the same JSON file against older commits until the replay succeeds. The commit that flips the replay from failure to success contains the unsafe, unversioned change.

Executing Mitigation Strategies

Once you identify the root cause, mitigate the impact based on the failure class. Apply these interventions in order of safety:

- *Pause task routing or worker polling*: If an external dependency goes down, prevent new workflows from starting and stop workers from picking up pending activities. You can scale activity worker deployments to zero or dynamically adjust polling configurations to halt processing. Executions will pause without losing their state, and retries will safely suspend until you restore capacity.
- *Hotfix activities*: Activity code is never replayed. If an activity is failing due to a logic bug, you can safely modify the code, deploy the new worker binary, and allow Temporal's retry mechanisms to execute the new logic on the next attempt. In-flight workflows will seamlessly pick up the fixed activity execution without any versioning overhead.
- *Hotfix workflows*: Modifying workflow logic during an incident requires strict determinism controls. If you must patch a workflow, you cannot simply change the code if the change alters the sequence of commands generated for existing histories. You must wrap the fix in a version marker or deploy it exclusively to a new Build ID via worker versioning.
- *Reset executions*: When a workflow has progressed down a fundamentally incorrect logical path due to a bug, fixing the code will not automatically rewind the execution state. You must reset the workflow.

Use the `temporal workflow reset` command to truncate the event history at a specific `WorkflowTaskCompleted` event. This drops all subsequent events—including any erroneous activity schedules or child

workflow starts—and forces the workflow to resume execution from the reset point using the currently deployed code. Identify the target event ID by inspecting the history for the last safe `WorkflowTaskCompleted` event prior to the divergence.

```
# Reset the workflow to the last successful workflow task before the
  error
temporal workflow reset \
  --workflow-id "billing-run-9381" \
  --event-id 15 \
  --reason "Rewinding prior to invalid account charge"
```

A reset effectively rewrites the future of the workflow. Before executing a reset, verify that any side effects generated by activities after the reset point are either inherently idempotent or have been manually compensated in the external systems. Resetting the Temporal state does not magically undo database commits or API calls that have already occurred in the real world.

Resolving Stuck Executions

For workflows that are stuck waiting on external input, the history will show no active timers and no pending activities. The workflow is entirely blocked on a `workflow.GetSignalChannel` call or a condition variable evaluation. In these scenarios, mitigation involves inspecting the external systems responsible for sending the missing signal. If the upstream system permanently dropped the event, you can manually unblock the workflow by sending the required signal payload via the CLI:

```
temporal workflow signal \
  --workflow-id "billing-run-9381" \
  --name "ApprovalSignal" \
  --input '{"approved": true, "reason": "incident-override"}'
```

By enforcing this forensic playbook, you stop relying on circumstantial evidence. You extract the deterministic history, pinpoint the exact point of divergence or failure, reproduce the anomaly locally via

replay, and apply targeted mitigations that preserve the integrity of the durable execution state.

7.3. Choosing upgrade strategies: backward compatibility, pinning, and multi-fleet rollouts

Workflow executions in Temporal are durable and can span days, months, or years. Because workers reconstruct workflow state by replaying history, deploying new code to workers that handle in-flight executions introduces a fundamental constraint: any change that alters the sequence of decisions previously recorded in the history will cause a non-determinism error. In contrast, activities execute outside the deterministic sandbox. You can deploy changes to activity implementations using standard continuous delivery practices, provided the activity signature and expected side-effect semantics remain compatible.

When you must change workflow logic—whether to introduce a new feature, fix a bug, or refactor a state machine—you must ensure that in-flight executions either continue processing on code that understands their existing history, or that the new code is explicitly written to handle both old and new histories. Navigating this constraint requires selecting an upgrade strategy aligned with your deployment infrastructure, the lifespan of your workflows, and your tolerance for maintaining legacy code paths.

7.3.1. Backward-compatible workflow code paths

The most direct approach to evolving workflow logic is to modify the code in a way that preserves backward compatibility with all existing event histories. Under this strategy, a single fleet of workers runs the

latest version of your workflow code, handling both new starts and the continuation of in-flight executions. To prevent non-determinism errors when replaying old histories through new logic, you use the SDK's patching (or versioning) APIs to explicitly branch the execution based on whether a specific change was present when the workflow originally executed that step.

Consider a scenario where you want to add a new notification activity to an existing order fulfillment workflow. If you simply insert the activity call, replaying an in-flight execution that has already passed this point will fail, because the history lacks the corresponding `ActivityTaskScheduled` event. You bridge this using a patch marker:

```
func OrderFulfillmentWorkflow(ctx workflow.Context, orderID string)
    error {
        // ... previous steps ...

        // The GetVersion API records a marker in the history for new
        // executions,
        // and returns the recorded version for replaying executions.
        v := workflow.GetVersion(ctx, "AddedShippingNotification",
            workflow.DefaultVersion, 1)
        if v == 1 {
            err := workflow.ExecuteActivity(ctx, SendNotificationActivity
                , orderID).Get(ctx, nil)
            if err != nil {
                return err
            }
        }

        // ... subsequent steps ...
        return nil
    }
```

When this code processes a new execution, `GetVersion` writes a `MarkerRecorded` event to the history with the patch identifier and the version 1, then executes the new activity. When it replays an older execution that originally ran before the patch was introduced, `GetVersion` does not find the marker in the history, returns `workflow.DefaultVersion`, and skips the activity call.

Managing compatibility debt

The primary advantage of patching is operational simplicity: you only deploy and manage one worker fleet. Rollbacks are straightforward, provided you do not roll back to a build that lacks the patch markers already written by the new build.

However, this strategy shifts the burden from infrastructure to code complexity. As workflows evolve, patch branches accumulate. You must maintain these branches until you can guarantee that no in-flight executions remain that depend on the old paths. For long-lived workflows, this “compatibility window” can stretch into months or years. Removing a patch too early results in immediate non-determinism failures for any lingering executions.

To govern this safely, you must pair code-level patching with robust replay testing. A comprehensive history corpus ensures that no branch is accidentally broken or removed prematurely. Use this strategy when making minor additive changes, adjusting timeouts, or introducing conditional steps. For extensive state machine refactors or fundamentally incompatible data structure changes, patching becomes unwieldy, necessitating infrastructure-level isolation.

7.3.2. Pinned workers and queue-based isolation

When workflow changes are too complex to manage via inline branching, or when a refactor fundamentally alters the decision logic, you can isolate the new code from the old code. The pinned worker strategy routes new workflow starts to a distinct task queue serviced exclusively by workers running the new code, while a subset of workers running the old code remains active on the original task queue to drain in-flight executions.

Implementing task queue routing

Implementing this strategy requires a routing mechanism at the point of workflow initiation. You configure the workflow starter (e.g., an API gateway or an event consumer) to send new execution requests to a versioned task queue name.

```
// Start the workflow on a dynamically versioned task queue
options := client.StartWorkflowOptions{
    ID:         "order-" + orderID,
    TaskQueue:  "orders-v2",
}
we, err := temporalClient.ExecuteWorkflow(context.Background(),
    options, "OrderWorkflow", req)
```

Simultaneously, you configure your deployment orchestration to maintain two separate worker deployments. The v1 deployment listens on the orders-v1 task queue, while the v2 deployment listens on the orders-v2. Because in-flight executions are inherently bound to the task queue on which they started, they naturally continue polling orders-v1.

Operational requirements and capacity planning

Pinning simplifies workflow code by eliminating patch branches; the code reflects only the current business logic. However, it significantly increases infrastructure overhead. You must provision, monitor, and scale multiple worker fleets simultaneously.

Capacity planning for the older fleets requires specific attention. As the v1 task queue drains, its aggregate load decreases, but the requirement for timely processing remains. You cannot scale the v1 fleet down to zero until the absolute final execution completes. This necessitates observability into the backlog and drain rate of legacy queues. If an execution enters a long-running sleep or awaits a human-in-the-loop signal, the v1 fleet must remain provisioned—perhaps for months—just to process a handful of eventual wake-ups.

Additionally, you must maintain the old worker binaries and their asso-

ciated dependencies. If a severe security vulnerability mandates patching an underlying library in the old code, you must rebuild and redeploy the v1 worker fleet without altering its workflow decision logic.

Pinning is highly effective for workflows with predictable, bounded lifespans (e.g., processes that guarantee completion within hours or days). You calculate the maximum drain time, retain the legacy fleet for that duration plus a safety margin, and then decommission it. For unbound or infinitely long-lived workflows, pinning leads to an ever-expanding graveyard of legacy task queues.

7.3.3. Multi-fleet rollouts and progressive load shifting

Multi-fleet rollouts extend the concept of isolation by running parallel worker fleets and progressively shifting traffic based on strict containment boundaries. Unlike static pinning, where an entire queue cuts over to a new version instantly, multi-fleet strategies distribute risk by limiting the blast radius of a new deployment to specific tenants, regions, or execution profiles.

Designing containment boundaries

To implement a multi-fleet rollout, you construct a routing layer that evaluates business criteria before assigning a task queue or directly routing the request. For example, in a multi-tenant SaaS application, you might migrate traffic by tenant ID.

You deploy a v2 worker fleet listening on a new set of task queues. The API layer intercepts workflow start requests, checks a feature flag or configuration map keyed by tenant ID, and directs the start to either the stable fleet or the canary fleet.

This strategy ensures that if the new workflow code contains a logical flaw or interacts poorly with downstream services, the impact is iso-

lated. In-flight executions for migrated tenants continue on the new fleet, while unmigrated tenants remain completely unaffected on the old fleet.

Handling rollbacks safely

The critical distinction between deploying standard microservices and deploying Temporal workflows lies in rollback execution. In a stateless microservice, rolling back involves routing traffic back to the previous deployment. In Temporal, if you route a workflow that has already made decisions on the v2 fleet back to a v1 fleet that lacks those code paths, the v1 worker will fail to replay the v2 history.

When executing a multi-fleet rollout, your rollback plan must distinguish between new starts and in-flight executions. If the v2 deployment exhibits issues, you immediately revert the routing rules for new starts, sending them back to the v1 fleet. However, executions that started on v2 are now “poisoned” from the perspective of the v1 code. You cannot simply shut down the v2 fleet. You must keep it running to allow those in-flight executions to complete, or you must explicitly build a forward-fix into v2 to gracefully terminate or migrate them.

This dynamic requires strict discipline around activity dependencies. If a workflow bug is actually an activity bug, rolling back the activity deployment is generally safe across all fleets. But if the workflow logic itself is flawed, you must stabilize the v2 fleet enough to drain its trapped executions.

7.3.4. Defining the operational contract

Selecting between backward compatibility, pinning, and multi-fleet rollouts is not purely an architectural decision; it dictates the operational contract your team must uphold. Each strategy demands specific observability configurations, routing discipline, and incident response

procedures.

To support backward-compatible code paths, your operational contract relies heavily on CI/CD rigor. The mandatory components include a comprehensive suite of replay tests integrated into every pull request, strict code review policies prohibiting the removal of patch markers without querying production for lingering open executions, and alerting on non-determinism (`WorkflowTaskFailed`) metrics.

To support pinned workers, the contract shifts toward infrastructure monitoring. You must implement dashboards that track task queue backlog, schedule-to-start latency, and active execution counts across all deployed task queue versions. You establish a formal decommissioning procedure: an engineer verifies that open executions for `orders-v1` equal zero before terminating the legacy deployment and deleting the repository branch.

To support multi-fleet rollouts, the contract centers on routing control and progressive delivery tooling. You need dynamic configuration management to adjust routing weights or tenant assignments without requiring code deployments. Your incident response playbooks must explicitly forbid shutting down a failed canary fleet until its associated task queue is confirmed empty.

Heuristics for strategy selection

Evaluate the nature of the change and the lifespan of the workflow to select the appropriate strategy:

- Use backward-compatible patching when the behavioral change is minor, additive, or applies conditionally. If the workflow lifespan is short, the patch can be removed relatively quickly. If the workflow lifespan is long, you commit to maintaining the patch indefinitely, which is often still cheaper than maintaining a parallel deployment for years.

- Use pinned workers when introducing breaking changes, significant state machine refactors, or altering input/output data structures. This is highly effective if the maximum execution lifespan is strictly bounded (e.g., 30 days). The cost of running the old fleet is amortized over a known, finite period.
- Use multi-fleet rollouts when the operational risk of the new code is high and requires progressive validation against production traffic. This provides the tightest control over blast radius, at the expense of requiring sophisticated routing and configuration management infrastructure.

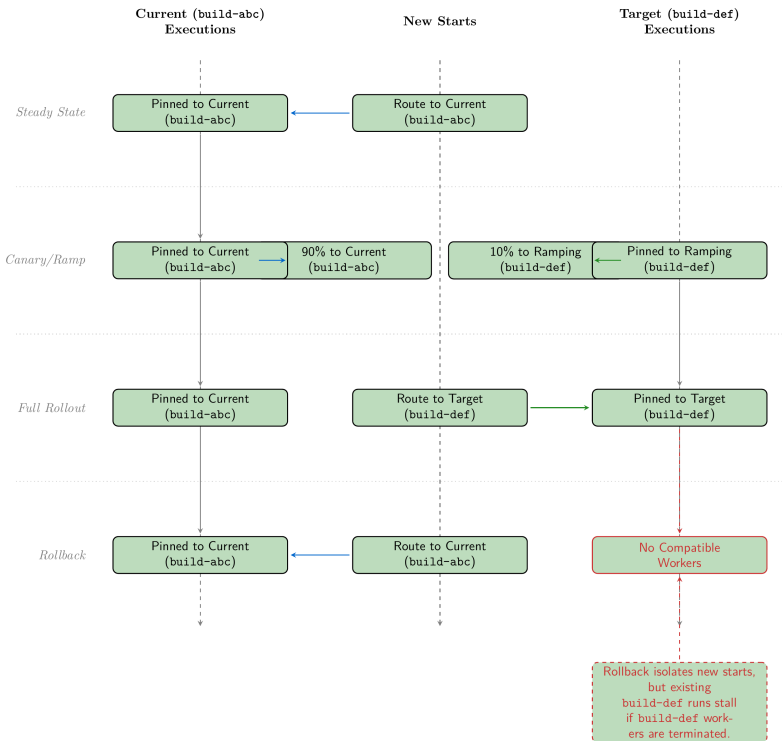
For many systems, these strategies are used in combination. A team might use patching for day-to-day feature additions to avoid infrastructure churn, while reserving a pinned task-queue migration for a major architectural overhaul planned once a year. Regardless of the chosen path, the fundamental rule remains: you must design your deployments to guarantee that every in-flight execution always has access to worker code capable of replaying its exact history.

7.4. Applying Worker Versioning: build IDs, ramp policies, and rollback-safe execution routing

Worker Versioning formalizes the relationship between a specific compiled workflow binary and the executions that depend on it. In durable execution, the code that starts a workflow must remain available to replay its history until that execution completes. Relying solely on strict backward compatibility to satisfy this requirement creates immense operational friction. Worker Versioning shifts this burden from code structure to execution routing, allowing you to deploy breaking workflow changes safely by directing tasks to specific, compatible worker

deployments.

A Worker Deployment Version is explicitly identified by a tuple of a deployment name and a build ID. The deployment name acts as a logical grouping for a service or application (e.g., `checkout-service`), while the build ID uniquely identifies the compiled artifact, such as a Git SHA, a semantic version, or a container image digest. When versioned workers poll the Temporal server, they report their deployment version. The server uses this information to partition the task queue and route workflow tasks exclusively to workers capable of successfully executing them.



Establishing deployments and routing policies

To control how workflows are assigned to workers, you manipulate the deployment routing policy via the Temporal CLI or SDK. The policy revolves around three key concepts: the Current version, the Ramping version, and the Target version.

The *Current version* is the default destination for new workflow starts. The *Ramping version* is a new deployment configured to receive a specific percentage [0, 100] of new workflow starts. The *Target version* is the dynamically selected build ID assigned to an incoming start request based on the evaluation of the ramp percentage against the Current and Ramping versions.

Execute the following CLI commands to create a deployment and register it as the Current version. This ensures all newly started workflows bind to build-abc:

```
# 1. Register the initial worker deployment
temporal worker deployment create \
  --name "checkout-service" \
  --build-id "build-abc"

# 2. Set the initial deployment as the Current version
temporal worker deployment rollout set-current \
  --name "checkout-service" \
  --build-id "build-abc"
```

When a workflow starts, the server resolves the assignment rule, binds the execution to build-abc, and places the first workflow task into the build-abc partition of the task queue. The workflow is now permanently pinned to this deployment version. Subsequent workflow tasks—triggered by signals, activities completing, or timers firing—will bypass assignment rules and route directly to the build-abc partition.

Executing a progressive ramp

When deploying a new worker binary with modified workflow logic (build-def), you do not immediately replace the Current version. In-

stead, you introduce it as the Ramping version. Ramping allows you to strictly control the blast radius of new, potentially flawed logic while verifying determinism, activity execution, and business metrics.

```
# 1. Register the newly compiled worker artifact
temporal worker deployment create \
  --name "checkout-service" \
  --build-id "build-def"

# 2. Configure 10% of new starts to route to the new build
temporal worker deployment rollout set-ramping \
  --name "checkout-service" \
  --build-id "build-def" \
  --percentage 10
```

Under this configuration, 10% of new workflows evaluate `build-def` as their Target version, while 90% continue to target `build-abc`. Existing workflows started prior to this configuration change are unaffected; their tasks continue routing to `build-abc` based on their pinned state.

During the ramp phase, you must monitor workflow task failure metrics. A workflow task failure indicates that the worker could not successfully process the task, commonly due to a determinism violation (e.g., a non-backward-compatible change was accidentally evaluated against an old history) or a severe application panic. Because you control the ramp percentage, you limit the number of new executions exposed to this risk.

Queue blocking is a documented failure mode during ramp phases. If the Ramping version is allocated a percentage of traffic but its worker fleet is down, misconfigured, or drastically under-provisioned, the server will queue tasks in the ramping partition. Due to task queue partitioning mechanics, backpressure from an unserved ramping queue can occasionally cascade, preventing the Current version from receiving its expected share of tasks. Always verify that workers reporting the new deployment version are actively polling before initiating a ramp.

You can verify the active routing configuration and poller status at any

time:

```
$ temporal worker deployment describe --name "checkout-service"
Current Version: build-abc
Ramping Version: build-def
Ramp Percentage: 10%
```

```
Active Pollers:
  build-abc: 45 nodes
  build-def: 5 nodes
```

Designing rollback-safe routing

In standard stateless microservices, a rollback is executed by terminating the new pods and redirecting network load balancers back to the old pods. In durable execution, applying this infrastructure-level rollback pattern causes catastrophic task stalling. If you terminate the build-def workers during a rollback, all workflows that started during the ramp phase and pinned to build-def will have nowhere to execute. Their tasks will sit in the build-def partition indefinitely, manifesting as a “no compatible workers” state.

Rollback safety in Temporal requires decoupling the *routing rollback* from the *infrastructure rollback*. A blue-green or rainbow deployment strategy is mandatory.

To abort a failing ramp, you immediately reset the routing rules so that build-abc recaptures 100% of new starts. This halts the bleeding. However, you must leave the build-def worker fleet running until all executions pinned to build-def either complete naturally or are manually terminated/reset.

```
# Revert routing logic: new starts drop back to build-abc exclusively
temporal worker deployment rollout set-current \
  --name "checkout-service" \
  --build-id "build-abc"

# The ramping rule is implicitly cleared, halting traffic to build-
  def.
# DO NOT terminate build-def workers yet!
```

By utilizing this routing-first rollback, your workflow logic does not need to be backward-compatible with the aborted deployment. Because `build-def` runs in isolation, its corrupted state or flawed decisions are confined to its own bounded execution pool.

To safely decommission `build-def`, query the Temporal visibility store to ensure no open executions remain bound to the aborted deployment version. Once the execution count reaches zero, the underlying infrastructure can be safely torn down.

Version assignment overrides and migration

While pinning workflows to the deployment version that initiated them is the safest default, there are scenarios where you must move in-flight executions to a newer build. This typically occurs when an actively running, long-lived workflow encounters a critical bug, or when you wish to consolidate worker fleets to reduce infrastructure costs.

You accomplish this by establishing an explicit compatible relationship, effectively telling the Temporal server: “Any execution pinned to `build-abc` is permitted to process its next task on `build-xyz`.”

When you configure this redirect, you take on the engineering burden of guaranteeing that `build-xyz` is strictly backward-compatible with all possible histories generated by `build-abc`. If `build-xyz` encounters an event sequence it cannot deterministically reproduce, the workflow task will fail, and the execution will stall. Version overrides should be reserved for emergency hotfixes on long-running workflows or carefully orchestrated fleet consolidations where compatibility has been proven against a comprehensive replay test corpus.

By mastering Worker Versioning, you transform deployment risk from an unpredictable determinism gamble into a controllable routing operation. Build IDs bind executions to code, ramp policies limit exposure, and routing rollbacks provide an instant escape hatch that protects the

integrity of in-flight states. Maintain strict discipline over your worker fleet lifecycles—never deleting a build ID fleet while executions depend on it—and progressive delivery becomes entirely safe for durable execution.

7.5. Evolving workflow contracts and metadata: schemas, search attributes, and update/signal APIs

Shipping durable execution code safely involves more than routing workflow tasks to deterministic workers. Workflows interact with the outside world through a set of coupled contracts: the internal state schemas that govern payload serialization, the external interaction APIs (signals and updates) that allow clients to mutate in-flight state, and the operational metadata (search attributes) used to cohort and monitor executions. Evolving these surfaces requires strict sequencing to prevent breaking in-flight executions, losing operational visibility, or trapping workflows in irrecoverable states.

To ground this, consider a concrete scenario where an e-commerce workflow must evolve. You need to migrate from an old `AddItems` signal to a newer, synchronous `AmendOrder` Update API, while simultaneously updating an execution's indexing behavior to support a new dashboard.

```
temporal workflow update \  
  --workflow-id "order-123" \  
  --name "AmendOrder" \  
  --input '{"ItemIds": ["xyz-789"], "Reason": "customer request"}' \  
  --meta "encoding=json/plain"
```

Within the worker code, the workflow must define the target state of this contract while maintaining backward compatibility with executions that started before the migration:

```

func OrderWorkflow(ctx workflow.Context, input OrderInput) (
    OrderResult, error) {
    // 1. Evolve Operational Metadata: Upserting search attributes
    // records a marker in the workflow history.
    err := workflow.UpsertSearchAttributes(ctx, map[string]interface{}{
        "CustomerTier": input.Tier, // Must map to a registered
        Keyword
    })
    if err != nil {
        return OrderResult{}, err
    }

    // 2. Evolve Interaction APIs: Registering the new Update handler
    err = workflow.SetUpdateHandler(ctx, "AmendOrder",
        func(ctx workflow.Context, req AmendRequest) error {
            // Apply strongly-typed state mutation
            return applyAmendment(ctx, req)
        },
        workflow.UpdateHandlerOptions{
            Validator: func(ctx workflow.Context, req AmendRequest)
            error {
                if len(req.ItemIds) == 0 {
                    return fmt.Errorf("validation failed: missing
items")
                }
                return nil
            },
        })

    // 3. Maintain Old Contract: Leave a tombstone handler for the
    deprecated Signal
    workflow.SetQueryHandler(ctx, "AddItems", func(req OldAddItemReq)
    error {
        // Forward legacy inputs to the new internal state machine
        logic
        return applyAmendment(ctx, translateToAmendRequest(req))
    })

    // ... workflow execution logic
}

```

This code demonstrates a point-in-time compatibility window. To reach this state-and eventually clean it up-you must manage the life-cycle of each contract surface independently.

Evolving workflow state schemas

The state schema encompasses all data serialized into the workflow history. This includes the workflow input, activity results, signal payloads, and memo fields. Because history is immutable, a workflow must be able to deserialize and interpret payloads written days, months, or years ago.

When evolving payloads, treat your workflow decoders as tolerant readers. Additive changes to payload structures (like adding a `DiscountCode` string to `OrderInput`) are generally safe, provided your Data Converter and language serialization libraries do not panic when encountering unknown fields in older histories, and default the new field deterministically. If the workflow behavior branches based on the presence of this new field, ensure the branch logic resolves safely for old executions where the field evaluates to null or an empty string.

Conversely, removing fields or changing their types is inherently unsafe if those fields are already written to the history of open executions. If an old history contains a `PaymentMethod` object, and the new worker code expects a `PaymentToken` string in that exact position, deserialization fails during replay, blocking the workflow task. To enforce schema stability, manage your workflow payloads as you would an external public API: use Protocol Buffers, Thrift, or strictly versioned JSON schemas, and rely on explicit schema registries if your Data Converter supports them.

Pay close attention to payload metadata. As shown in the CLI example, input payloads carry encoding overrides (e.g., `encoding=json/plain` or `encoding=protobuf`). The worker's Data Converter inspects this metadata to select the correct deserialization strategy. This mechanism allows you to roll out serialization changes—for instance, migrating from JSON to Protobuf—by updating the client to send the new encoding metadata, while the worker retains the ability to decode both

formats based on the headers recorded in the history.

Evolving operational metadata and search attributes

Search attributes dictate how executions are indexed in the cluster's visibility store (typically Elasticsearch or OpenSearch) and are crucial for dashboards, alerting, and batch operations. Unlike arbitrary workflow payloads, search attributes have explicit, strictly enforced types at the Temporal cluster level.

A common schema evolution trap involves the distinction between `Keyword` and `Text` search attributes. If you store a `UUID` in a `Text` attribute, the underlying search engine tokenizes it using standard text analyzers (often splitting on hyphens). Consequently, an exact-match query for `ExecutionID = "123e4567-e89b-12d3-a456-426614174000"` will fail or return spurious matches. You must use `Keyword` types for identifiers to guarantee exact string matching.

When you need to change a search attribute's type or meaning, you cannot simply alter the cluster configuration in-place without disrupting in-flight queries. Instead, follow a dual-write migration strategy. First, register the new search attribute (e.g., `OrderStateCode` as an `Int`) at the cluster level. Next, deploy worker code that upserts both the old `OrderState` (`Keyword`) and the new `OrderStateCode`.

Crucially, calling `UpsertSearchAttributes` writes an `UpsertWorkflowSearchAttributes` event to the workflow history. This means metadata changes are not just ephemeral side effects; they act as a contract-as-history audit trail. You can replay a workflow and definitively prove exactly when it updated its operational visibility state. Because of this determinism requirement, do not remove the old search attribute upsert logic from your workflow code until all open executions that rely on it have closed, or until you are certain that replaying past upserts of the old attribute will not break your

current indexing invariants.

Evolving external interaction contracts (Signals and Updates)

Workflows expose their interaction surfaces through Signal and Update handlers. While both allow external clients to pass data into a running execution, Updates provide a synchronous response and run a distinct validation phase before writing to history. Note that Workflow Updates are a newer feature in the Temporal ecosystem; when relying on them, verify the feature maturity and behavioral guarantees in your specific SDK version.

Removing a Signal or Update handler is an irreversible state transition if there are in-flight executions whose histories contain that event. If an execution received an `AddItems` signal yesterday, the `WorkflowExecutionSignaled` event is permanently etched into its history. If you deploy a new worker today that lacks the `AddItems` handler, the SDK will encounter the signal in history during replay, fail to find the corresponding handler, and throw a non-determinism error. The execution becomes permanently stuck.

To deprecate a handler, you must replace it with a “tombstone” handler—a function that accepts the old payload schema but performs a no-op, or translates the payload into the new state machine logic. You must retain this “tombstone” handler until all executions containing the old signal have naturally closed or been forcefully terminated.

Updates introduce an additional evolution constraint: the validation handler. When a client sends an Update, the worker executes the validation logic. If it passes, the Update is accepted, and an `WorkflowExecutionUpdateAccepted` event is recorded. If you change the validation logic to be strictly narrower (e.g., rejecting an Update if `ItemIds` is empty, where previously it was allowed), you risk breaking in-flight clients. However, tightening validation logic does *not* break

replay for Updates that were already accepted, because replay skips the validation phase for historically accepted Updates and proceeds directly to the execution phase. The contract breakage here is strictly between the new worker and old clients currently issuing requests.

The coordinated migration playbook

Because these three surfaces-internal schemas, operational metadata, and interaction APIs-are coupled, evolving them requires a sequenced migration playbook. Attempting to change the client, the worker, and the visibility dashboards simultaneously invariably leads to race conditions.

Execute migrations using a defined sequence of compatibility windows.

- **Phase 1: Worker preparation.** Deploy the new worker code first. The worker must be configured to tolerate both old and new states. It should register “tombstone” handlers for deprecated signals, accept both old and new payload schemas, and dual-write to both old and new search attributes. Monitor the deployment to ensure workflow task failure rates remain flat, proving that the new worker deterministically replays all old histories.
- **Phase 2: Operational cutover.** With the worker dual-writing search attributes, update your operational dashboards, alerts, and batch scripts to query against the new search attributes. If backfilling is required for older executions that haven’t received the new upsert, run a batch job to update their search attributes using the server API. Verify that your operational visibility is fully reliant on the new metadata.
- **Phase 3: Client cutover.** Update the downstream clients (web backends, cron jobs, API gateways) to invoke the new Update APIs or send the new payload schemas. You can use the Tempo-

ral CLI to inject test payloads with specific encoding metadata to verify the worker handles the transition correctly before shifting live client traffic. Monitor the invocation metrics on the worker to confirm that traffic to the deprecated handlers is dropping to zero.

- **Phase 4: Compatibility cleanup.** You cannot remove the compatibility code immediately after client traffic shifts. You must wait for the “long tail” of open executions to close. Identify the maximum possible lifetime of the workflows running the deprecated contracts. Only when you can cryptographically prove—via cluster visibility queries—that zero open executions require the old signal handlers or the old schema fields, is it safe to delete the “tombstone” handlers and single-write the new search attributes.

Avoiding irrecoverable state transitions

Even with a strict migration playbook, certain structural changes to a workflow’s state machine can trap executions. This occurs when a migration inadvertently deletes a “bridge state.”

Consider a workflow that transitions from `Pending` to `PaymentAuthorized` to `Shipped`. If you refactor the workflow logic to completely remove the `PaymentAuthorized` step (perhaps integrating authorization directly into the checkout activity), you must still account for workflows currently sitting in the `PaymentAuthorized` state. If the new code skips this step entirely, it will attempt to schedule the `Shipped` activity immediately during replay. The SDK will compare this new command against the history, see that the history originally recorded an authorization activity, and fail the task due to a command mismatch.

To avoid this, maintain bridge states using explicit versioning markers (discussed in detail alongside Worker Versioning). Do not rewrite the history-generating control flow for in-flight executions. Instead,

use compatibility branching to ensure that executions already in the `PaymentAuthorized` state can gracefully complete that step, while new executions follow the streamlined path.

Treat your workflow contracts as append-only. Evolve by extending schemas, dual-writing metadata, and adding new handlers, using explicit compatibility windows to bridge the gap until old executions drain.

Chapter 8

Durable Agents with Temporal: Reliable LLM and Tool Orchestration

Agentic systems fail in ways that don't look like ordinary distributed systems failures: the model is stochastic, tools are flaky, side effects are hard to unwind, and “waiting for a human” breaks most runtimes. This chapter shows how to treat an agent as a durable, inspectable, policy-governed program-where every step can be retried, audited, paused, rolled forward, or rolled back-without pretending the LLM is deterministic.

8.1. Mapping Agent Control Loops to Workflows and Tool Calls to Activities

An autonomous agent operates through a continuous control loop: it evaluates its current context against a goal (plan), selects an operation to perform (act), captures the result of that operation (observe), and integrates that result into its context (revise). In a standard script, this loop is fragile. A crash during a long-running external API call destroys the in-memory context, requiring the agent to start over from scratch, potentially re-executing non-idempotent side effects.

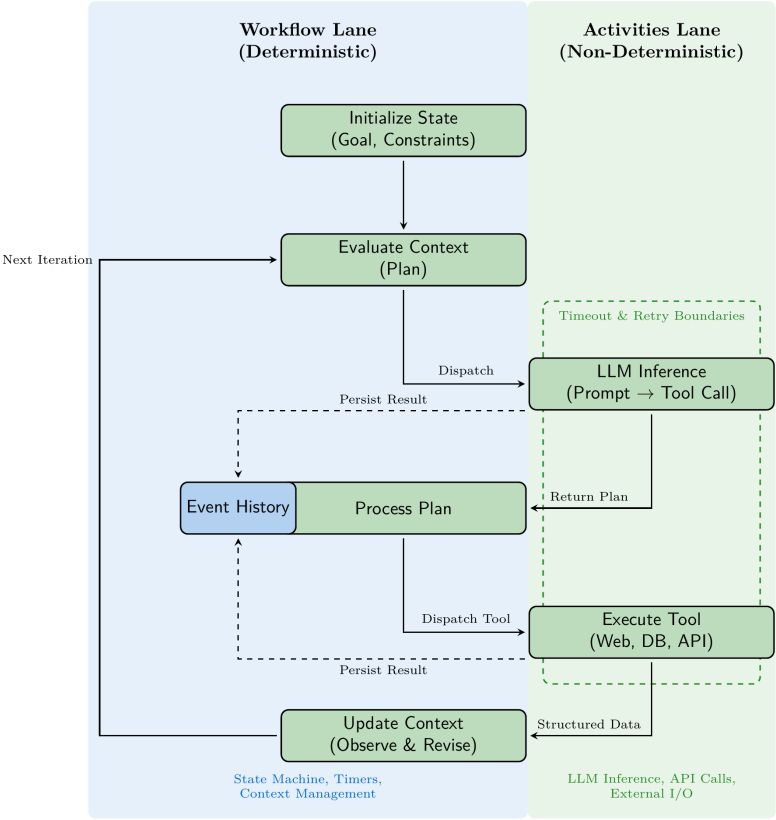
By modeling this loop as a Temporal execution, you transform the fragile script into a durable state machine. The fundamental architectural decision in this model is drawing a strict boundary between deterministic orchestration and non-deterministic execution.

The Determinism Mapping Rule Temporal requires workflow code to be strictly deterministic. If a worker process crashes and restarts, the Temporal server replays the execution's Event History through the workflow code to reconstruct the exact same state. If the code behaves differently during replay, the workflow fails with a non-deterministic error.

This dictates a rigid mapping for agent architectures:

- *Workflows own the control loop and context.* The workflow maintains the durable state machine, the iteration limits, the sequence of operations, and the authoritative transcript of decisions.
- *Activities own all model inference and external I/O.* Every call to a Large Language Model (LLM), database, or external tool must execute inside an activity. Activities execute at most once for a

given step in the workflow history; their results are recorded in the Event History and returned identically during replay.



Implementing the Replay-Safe Agent Kernel Translating this rule into code requires encapsulating the agent’s ongoing context into a strictly typed data structure. Avoid managing state implicitly through a continuously growing raw string of conversation history. Instead, maintain structured variables for the initial goal, operational con-

straints, the current iteration count, and an array of normalized observations.

Below is a Python implementation of an agent kernel utilizing the Temporal Python SDK. It implements a bounded loop that coordinates LLM reasoning and tool execution.

```
from dataclasses import dataclass, field
from datetime import timedelta
from typing import List, Optional
from temporalio import workflow, activity
from temporalio.common import RetryPolicy

@dataclass
class Observation:
    action_type: str
    target: str
    result_summary: str

@dataclass
class AgentState:
    goal: str
    iteration: int = 0
    max_steps: int = 10
    history: List[Observation] = field(default_factory=list)
    status: str = "running"

@dataclass
class PlanResult:
    is_complete: bool
    tool_name: Optional[str]
    tool_payload: Optional[dict]
    final_answer: Optional[str]

@workflow.defn
class AutonomousAgentWorkflow:
    @workflow.run
    async def run(self, goal: str) -> str:
        # Initialize durable context
        state = AgentState(goal=goal)
        workflow.set_query_handler("get_state", lambda: state)

        while state.iteration < state.max_steps:
            state.iteration += 1

            # 1. Plan: Call LLM Activity
            plan: PlanResult = await workflow.execute_activity(
```

```

        "llm_reasoning_activity",
        state,
        start_to_close_timeout=timedelta(minutes=2),
        retry_policy=RetryPolicy(
            initial_interval=timedelta(seconds=2),
            maximum_attempts=5,
            non_retryable_error_types=["
ContextLengthExceededError"]
        )
    )

    # 2. Check for completion
    if plan.is_complete:
        state.status = "completed"
        return plan.final_answer or "Goal achieved."

    # 3. Act: Execute Tool Activity
    tool_output = await workflow.execute_activity(
        "execute_tool_activity",
        plan.tool_payload,
        start_to_close_timeout=timedelta(seconds=30),
        retry_policy=RetryPolicy(maximum_attempts=3)
    )

    # 4. Observe and Revise
    observation = Observation(
        action_type=plan.tool_name,
        target=plan.tool_payload.get("id", "unknown"),
        result_summary=tool_output
    )
    state.history.append(observation)

    state.status = "failed_budget_exceeded"
    return "Failed: Maximum steps reached."

```

This structure ensures that if the worker executing the `execute_tool_activity` is terminated abruptly, the Temporal server will transfer the workflow task to another worker. The new worker replays the workflow from the beginning. During replay, `workflow.execute_activity("llm_reasoning_activity", ...)` does not invoke the LLM again. Instead, the SDK intercepts the call, retrieves the previously recorded `PlanResult` from the Event History, and immediately yields it. The workflow state reconstruction

is functionally identical to the original execution.

Treating the Model as a Fallible Dependency A common anti-pattern in early agent development is treating the LLM invocation as a reliable utility function, often calling it directly within the workflow. Because LLMs exhibit high latency, frequently encounter rate limits (e.g., HTTP 429), and occasionally fail due to transient provider degradation, invoking them directly breaks replay logic and ties the orchestration layer’s availability to the provider’s uptime.

By moving the LLM call strictly behind an activity boundary, you expose it to Temporal’s retry mechanics. The `RetryPolicy` applied to `llm_reasoning_activity` in the previous example is critical. It defines a capped exponential backoff. You configure `maximum_attempts` to establish a cost and latency ceiling for reasoning steps, preventing infinite retry loops that silently consume platform budgets.

Similarly, explicitly defining `non_retryable_error_types` forces terminal failures for semantic or structural errors. If an LLM activity throws a `ContextLengthExceededError`, retrying the exact same payload will persistently fail. Passing it back to the workflow as an exception allows the workflow to implement fallback logic—such as spawning a summarization activity to compress the history array before attempting the reasoning step again.

Defending Against History Bloat and Nondeterminism

When integrating LLMs, payloads grow rapidly. If an agent converses back and forth, appending raw HTTP transcripts (including system prompts, few-shot examples, and verbose model outputs) to the state on every iteration, the Temporal Event History will quickly expand.

Temporal has hard limits on Event History: 50,000 events and 50MB in total size. Even before hitting these limits, bloated histories incur se-

vere performance penalties during replay because large payloads must be deserialized repeatedly.

To defend against history bloat, enforce strict normalization within your activities. The LLM activity must not return the raw JSON response received from OpenAI or Anthropic. It should parse the output, extract only the structural intent (the tool call parameters or the final response), and return a highly compressed representation (like the `PlanResult` class). The activity itself constructs the full prompt by injecting the workflow's compact `AgentState` into a template. The raw prompt string and the raw completion string live and die entirely within the activity worker's memory; they never pollute the durable Event History.

If an external tool returns massive output—such as downloading a 10MB PDF or scraping a heavy DOM—the tool activity must truncate, summarize, or extract the relevant semantic facts before returning. If the raw artifact is needed for downstream steps, the activity should upload the payload to an object store (e.g., Amazon S3) and return only the URI. The workflow durably tracks the URI, keeping the payload out of the Event History while preserving determinism.

Operationalizing the Agent Loop Once deployed, the agent kernel behaves as a highly observable background process. Because the state is explicitly structured and updated sequentially, you can interact with the live agent using Temporal Queries and Signals (detailed extensively in Chapter 5).

In the code example, the line `workflow.set_query_handler("get_state", lambda: state)` exposes the agent's internal progress. Operators or customer-facing dashboards can query this handler at any time to inspect the current iteration, the specific goal, and the exact sequence of history observations without blocking the ongoing loop

or waking up an activity worker.

When a failure occurs—whether it’s a terminal exception in a tool or a constraint violation—the operator does not need to guess what the LLM was attempting. Because the workflow cleanly separated the deterministic decisions from the I/O, you can use the Temporal UI or CLI to inspect the Event History. The history provides a precise, time-stamped ledger: the inputs provided to the `llm_reasoning_activity`, the structured `PlanResult` it returned, and the subsequent inputs passed to `execute_tool_activity`. This strict observability boundary transforms unpredictable agent behavior from a black-box machine learning problem into a standard distributed systems debugging process.

8.2. Engineering Semantic Failure Handling: Structured Retries, Fallback Tools, and Compensations

In traditional distributed systems, failures are typically infrastructure exceptions: network partitions, database timeouts, or HTTP 503 errors. You handle these by relying on standard exponential backoff. In agentic systems, the most dangerous failures arrive wrapped in an HTTP 200 OK. A language model successfully generates a response, but hallucinates a required parameter, selects a destructive tool, or violates a strict system prompt constraint. If you treat these semantic failures as standard exceptions and allow a default retry policy to blindly re-execute them, you create infinite loops that burn API tokens, exacerbate rate limits, and fail to advance the system state.

To build reliable agents, you must construct an error model that explicitly distinguishes between infrastructure faults and semantic flaws, driving a dynamic repair loop governed directly by the workflow.

8.2.1. The Semantic Failure Taxonomy

A robust agent orchestration layer requires a strict taxonomy of failure modes. You classify errors into three distinct categories to determine whether the failure belongs to the infrastructure layer or the agent's reasoning layer.

- **Transport and Transient Faults:** Timeouts, rate limits, and service unavailability. The agent's intent was correct, but the execution environment failed. These are handled purely by Temporal's native `RetryPolicy`. The activity raises an exception, and Temporal automatically re-executes it without executing any workflow logic.
- **Deterministic Validation Failures:** The model output violates a JSON schema, or a mandatory tool argument is missing. Retrying blindly without modifying the input is futile. The system must short-circuit native retries, capture the validation error, and feed it back into the model's context window.
- **Semantic Failures:** The tool executed successfully from a network perspective, but the business context is invalid. For example, the agent attempts to grant administrative privileges to a guest user, violating a policy check. These failures require branching workflow logic to switch tools, change models, or compensate prior actions.

When designing tool activities, you face an architectural choice: signal semantic failures by raising an `ApplicationError` marked as non-retryable, or return a structured data object indicating failure. While raising an exception is idiomatic for static microservices, a semantic failure in an agentic system is fundamentally an observation. The tool successfully interacted with the environment, and discovering that an action is prohibited is a valid data point the agent needs to see.

Returning a structured observation prevents your workflow execution history from filling with `ActivityTaskFailed` events for routine business logic rejections. You define a rich result schema that standardizes these observations across all tools.

```
from temporalio import activity
from pydantic import BaseModel, Field
from typing import Optional, Dict, Any

class AgentObservation(BaseModel):
    success: bool
    data: Optional[Dict[str, Any]] = None
    error_class: Optional[str] = None
    error_message: Optional[str] = None
    provenance: str = Field(default="unknown")
    effect_descriptor: str = Field(default="none")

@activity.defn
async def execute_customer_refund(refund_amount: float, customer_id:
    str) -> AgentObservation:
    try:
        # Network calls raise standard exceptions
        customer = await db_client.get_customer(customer_id)
    except TimeoutError as e:
        # Transport failure: handled invisibly by Temporal
        RetryPolicy
        raise e

    if customer.balance < refund_amount:
        # Semantic failure: returned as a successful activity
        execution
        # carrying a failed business observation
        return AgentObservation(
            success=False,
            error_class="SEMANTIC_BUSINESS_RULE",
            error_message=f"Cannot refund {refund_amount}. Balance is
            {customer.balance}.",
            provenance="internal:billing_api/v2",
            effect_descriptor="none"
        )

    await db_client.process_refund(customer_id, refund_amount)
    return AgentObservation(
        success=True,
        data={"refunded": refund_amount},
        effect_descriptor="mutated"
    )
```


8.2.2. Orchestrating the Repair Loop

Instead of failing the activity, the structured `AgentObservation` routes control flow back to the deterministic workflow. The workflow evaluates the `error_class` and executes a stateful repair loop.

If the error is a deterministic validation failure, the repair loop appends the exact schema mismatch string to the agent's context and re-invokes the language model. If the error is a semantic policy violation, reprompting the same model often yields a stubborn repetition of the hallucination. In this scenario, the workflow executes a fallback strategy: it switches to a more capable, reasoning-heavy provider, or forces the execution of a retrieval tool to gather more context.

```
from temporalio import workflow
from datetime import timedelta

@workflow.defn
class AgentWorkflow:
    @workflow.run
    async def run(self, context: dict) -> str:
        max_repair_attempts = 3
        attempt = 0
        current_model = "primary-fast-model"

        while attempt < max_repair_attempts:
            # Plan generation is an activity using current workflow
            state
            plan = await workflow.execute_activity(
                generate_plan,
                {"context": context, "model": current_model},
                start_to_close_timeout=timedelta(minutes=1)
            )

            observation = await workflow.execute_activity(
                execute_tool,
                plan.tool_call,
                start_to_close_timeout=timedelta(seconds=30)
            )

            if observation.success:
                return "Goal Achieved"

        # Strategy: Reprompt with exact error
```

```

        if observation.error_class == "VALIDATION":
            context["history"].append({
                "role": "system",
                "content": f"Schema error: {observation.
error_message}. Correct the JSON."
            })
            attempt += 1
            continue

        # Strategy: Provider Fallback
        if observation.error_class in ["SEMANTIC_BUSINESS_RULE",
"POLICY"]:
            current_model = "fallback-reasoning-model"
            context["history"].append({
                "role": "system",
                "content": f"Action rejected: {observation.
error_message}. Formulate new strategy."
            })
            attempt += 1
            continue

        # Transition to an escalation state when budget is exhausted
        return await self.escalate_to_human(context)

```

By embedding the fallback logic directly into the workflow, you guarantee that the transition between providers is durable. If the worker crashes immediately after the primary model fails, the workflow replays, sees the logged validation error, and seamlessly initiates the fallback request upon recovery.

8.2.3. Durable Compensations for Partial Effects

Agents rarely execute single, isolated actions; they formulate multi-step plans. When an agent mutates external state in the first step, but subsequently encounters an unrecoverable semantic failure in the second step, the environment is left in a partially complete, inconsistent state. Because the failure fundamentally altered the agent's intention, you cannot rely on downstream APIs to automatically roll back the transaction. You must execute explicit, durable compensations.

A compensation stack is a list of reversal parameters maintained in workflow state. Whenever an activity returns an observation with `effect_descriptor="mutated"`, the workflow immediately pushes the corresponding compensation instructions onto the stack. If the plan breaks down, the workflow iterates through the stack and executes the compensations in reverse order.

```
@workflow.defn
class AgentWorkflow:
    # ...
    async def execute_plan(self, plan: PlanRequest) -> bool:
        compensations = []
        try:
            for step in plan.steps:
                observation = await workflow.execute_activity(
                    execute_tool,
                    step,
                    start_to_close_timeout=timedelta(seconds=30)
                )

                if observation.success and observation.
effect_descriptor == "mutated":
                    # Store exact parameters required to revert this
action
                    compensations.insert(0, step.compensation_params)

                    elif not observation.success:
                        workflow.logger.error(f"Semantic failure: {
observation.error_message}")
                        await self.run_compensations(compensations)
                        return False

            return True

        except Exception as e:
            # Transport failure exhausted native Temporal retries
            workflow.logger.error("Infrastructure failure during plan
execution.")
            await self.run_compensations(compensations)
            raise e

    async def run_compensations(self, compensations: list):
        for comp in compensations:
            # Compensations must succeed; use infinite retries
structurally
            await workflow.execute_activity(
```

```

        undo_tool,
        comp,
        start_to_close_timeout=timedelta(minutes=2),
        retry_policy=RetryPolicy(maximum_attempts=0)
    )

```

When designing compensation activities, strictly enforce idempotency. A compensation might be interrupted by network failure midway through execution and retried. If the agent's initial action was `CreateTicket`, the compensation is `CloseTicket(id)`. If `CloseTicket` is called twice for the same ID, the second invocation must cleanly return success rather than throwing a “Ticket already closed” error, which would stall the compensation loop indefinitely.

8.2.4. Observability and Circuit Breakers

Because agents self-heal and alter their strategies dynamically, observing standard activity metrics is insufficient. You need to know why an agent changed course. Every time the workflow catches an error class, switches to a fallback model, or executes a compensation, you append a structured event to a `remediation_trace` list managed within the workflow state.

This trace is exposed via a Temporal Query handler. Operators rely on it during postmortems to identify which internal tools consistently trigger validation errors or which prompts cause semantic drift.

```

$ temporal workflow query --workflow-id agent-run-1234 --type get_remediation_trace
[
  {
    "timestamp": "2023-10-24T10:15:30Z",
    "attempt": 1,
    "action": "execute_customer_refund",
    "outcome": "SEMANTIC_BUSINESS_RULE",
    "details": "Cannot refund 50.0. Balance is 30.0.",
    "remediation_strategy": "FALLBACK_TO_REASONING_MODEL"
  },

```

```
{
  "timestamp": "2023-10-24T10:16:05Z",
  "attempt": 2,
  "action": "fetch_billing_history",
  "outcome": "SUCCESS",
  "details": "Found pending deposit of 20.0.",
  "remediation_strategy": "NONE"
}
```

Autonomous repair loops carry the risk of runaway API utilization. A persistent hallucination will cause the agent to endlessly reprompt the model, burning tokens without advancing the system state. To prevent this, enforce tight execution budgets at the workflow level.

You maintain accumulators for `total_prompt_tokens`, `total_completion_tokens`, and a wall-clock `start_time`. Before initiating any iteration of the repair loop, the workflow compares these accumulators against predefined limits. When a budget is exhausted—or a semantic failure is flagged as explicitly unrecoverable by a policy gate—the workflow triggers a circuit breaker. It halts autonomous execution, marks its state as requiring intervention, and enters a dormant phase. The context is safely preserved in Temporal’s durable state, fully prepared to resume execution once external instructions arrive.

8.3. Implementing Human Feedback and Policy Gates with Signals and Updates

When an autonomous agent reaches a boundary condition—whether due to high-risk side effects, ambiguous model outputs, or explicit business rules—it must pause safely and await external direction. Treating human-in-the-loop interactions as a correctness primitive ensures that the agent yields control deterministically without consuming compute resources, polling fragile endpoints, or losing context. Temporal pro-

vides two mechanisms for receiving external input: Signals and Updates. Signals act as asynchronous, fire-and-forget messages, while Updates provide synchronous, tracked requests with a strict validation phase. For policy gates and approvals, Updates are heavily preferred. Their validation phase allows you to inspect the workflow state and reject state-mutating requests that arrive out of order or reference stale plans, cleanly pushing the error back to the client without polluting the workflow history.

```
from datetime import timedelta
from temporalio import workflow
import asyncio

@workflow.defn
class PolicyGatedAgent:
    def __init__(self) -> None:
        self.pending_approval = False
        self.current_plan_hash = ""
        self.approved = False
        self.agent_paused = False
        self.terminated = False
        self.agent_loop_scope = None

    @workflow.update
    async def approve_plan(self, plan_hash: str) -> str:
        self.pending_approval = False
        self.approved = True
        return f"Plan {plan_hash} approved."

    @approve_plan.validator
    def validate_approve_plan(self, plan_hash: str) -> None:
        if not self.pending_approval:
            raise ValueError("Agent is not awaiting approval.")
        if plan_hash != self.current_plan_hash:
            raise ValueError(f"Stale plan hash. Expected {self.current_plan_hash}")

    @workflow.signal
    def kill_switch(self) -> None:
        self.terminated = True
        if self.agent_loop_scope:
            self.agent_loop_scope.cancel()

    @workflow.run
    async def run(self, request: dict) -> dict:
```

```
self.agent_loop_scope = workflow.CancellationScope()
try:
    async with self.agent_loop_scope:
        return await self.agent_loop()
except asyncio.CancelledError:
    await workflow.execute_activity(
        "CleanupActivity",
        schedule_to_close_timeout=timedelta(minutes=5)
    )
    return {"status": "terminated_by_operator"}

async def agent_loop(self) -> dict:
    # 1. Agent formulates an intent
    self.current_plan_hash = "plan_7b8x9"
    self.pending_approval = True

    # 2. Block until approval, pause release, or timeout
    try:
        await workflow.wait_condition(
            lambda: (not self.pending_approval) and (not self.
agent_paused),
            timeout=timedelta(hours=24)
        )
    except asyncio.TimeoutError:
        self.pending_approval = False
        await workflow.execute_activity(
            "EscalateToOperator",
            schedule_to_close_timeout=timedelta(minutes=5)
        )
        return {"status": "escalated"}

    if self.approved:
        # 3. Proceed with plan execution
        pass

    return {"status": "completed"}
```

Interaction Contracts and Concurrency

Establishing a durable contract between the human operator and the agent requires handling concurrency safely. Temporal executes workflows in a single-threaded deterministic sandbox, but message handlers—both Signals and Updates—are invoked as coroutines that interleave with the main workflow thread at explicit yield points. Because handlers execute synchronously during these yields, the work-

flow state can mutate between two consecutive lines of code if an `await` separates them.

Design your update handlers to mutate state monotonically. An agent moves strictly forward: from planning, to pending approval, to approved, to executing. The update validator enforces this by rejecting approvals if the workflow is not actively in the pending state. Because the validator executes on the worker and throws an exception directly back to the calling client, invalid transitions are blocked before they ever execute. Furthermore, rejections leave no trace in the workflow execution history. This mechanism protects the history size limits from being exhausted by aggressive client retries or automated systems blindly submitting approvals.

You must also account for exactly-once application guarantees. When an operator clicks “Approve” on a sluggish cellular connection, the browser might retry the request multiple times. Because Updates use Temporal’s native protocol, the client SDK attaches a unique Update ID to the request. The Temporal cluster deduplicates the incoming requests automatically. The approval applies exactly once, and subsequent client retries safely receive the result of the already-completed update without triggering the handler a second time.

Pre-Action Gates and Plan Binding

Pre-action gates enforce a strict sequence before the agent executes a tool with meaningful side effects. The core vulnerability in approval systems is intent drift: the operator approves one action, but the agent executes another. To prevent this, the workflow must explicitly couple the operator’s approval to the exact cryptographic state of the proposed action.

Calculate a deterministic hash of the proposed tool parameters, tool names, and target execution environment. Ensure keys are sorted to guarantee a stable JSON serialization order before hashing. The

update payload from the operator must include this exact hash. If the workflow experiences a timeout or a non-deterministic state shift that causes the agent to formulate a new plan, the stored `current_plan_hash` changes. When the delayed approval arrives, the validator compares the operator's hash against the newly generated hash, detects the mismatch, and rejects the update. This guarantees the operator and the agent remain perfectly synchronized on the identical intent.

Time-bounded waits are equally critical. An agent blocking indefinitely for human input poses a silent operational risk, tying up business processes and obscuring stalled executions. Enforce an SLA using the timeout parameter of `workflow.wait_condition`. If the SLA expires, catch the resulting timeout exception and execute a dedicated escalation path. Escalation typically involves executing an activity to page an on-call engineer, release any external locks held by the agent, and transition the agent to a terminal state or a safe standby loop.

Policy Enforcement Gates

Moving beyond explicit human review, agents require programmatic guardrails to evaluate constraints that cannot be statically validated by a schema. For example, a policy gate might evaluate whether an email draft contains sensitive user data, or if a proposed infrastructure change violates compliance rules.

Because business policies evolve independently of workflow code, policy evaluation must be executed as an external activity. Do not embed external policy logic or make direct HTTP calls to an Open Policy Agent (OPA) server from within the workflow code. If the workflow replays weeks later after a policy update, an embedded check would evaluate the new rules and yield a different result, violating determinism and failing the replay.

Instead, the workflow calls an `EvaluatePolicy` activity. The

outcome—whether allowed or denied, along with the semantic reasons—is recorded durably in the workflow history. The agent loop observes the result. If denied, the agent uses the semantic failure reason to formulate a new plan, effectively utilizing the policy gate as constraint injection.

Persist the policy context as first-class workflow state. Create a structured data type that captures the `rule_set_version`, the `decision_id`, and the `evaluator_identity`. Store these records in a workflow-level audit list. When the workflow completes, this list serves as a cryptographically verifiable ledger. Security teams can rely on this ledger because Temporal’s execution model guarantees that the recorded context precisely matches what the workflow evaluated to proceed.

Intervention Controls: Pause, Resume, and Constraint Injection

Long-running autonomy requires mid-flight intervention. Operators must be able to pause execution or inject new constraints without restarting the agent. Implement pause and resume functionality using a state flag checked at predefined safe points. A workflow cannot preempt an activity midway through execution from the orchestrator’s perspective. Instead, define an interruptible loop. Before initiating the next planning phase or tool execution, the workflow evaluates a condition that blocks if the `agent_paused` flag is true. Operators send a pause signal or update, flipping the flag. The workflow safely halts at the boundary of the next step.

Constraint injection allows operators to steer the agent dynamically. Expose an update handler that appends text or structured rules to an `injected_constraints` list. The workflow explicitly passes this list into the payload of the next LLM planning activity. Because the constraint injection is captured as an update, the new instructions become part of the durable state and survive replays, seamlessly altering the

agent's behavior on its next iteration.

For emergency stops, implement a kill switch using cancellation scopes. When a `kill_switch` signal arrives, the workflow flips a termination flag and explicitly cancels the active cancellation scope wrapping the agent loop. Inside the workflow execution, catch the `asyncio.CancelledError`. This routes the workflow into a cleanup block containing compensation activities—reverting partial database writes, closing open tickets, or notifying a channel that the agent was terminated—before cleanly exiting the execution.

End-to-End Operational Flow

Integrating these gates into a production environment requires connecting the workflow state to operator interfaces. Begin by having upstream systems invoke the agent using an `Update-With-Start` call. This mechanism either creates the workflow if it does not exist or routes the request to the running instance, eliminating race conditions around workflow initialization.

When the agent reaches a gated plan, it upserts a search attribute—such as `AgentStatus = "PendingApproval"`. Search attributes expose internal workflow state to the Temporal visibility store, making the workflow discoverable via SQL-like queries. Immediately after updating the search attribute, the workflow executes an activity to notify an external queue, posting a message to Slack or updating a database row that powers an operator dashboard.

Operators query the visibility store to populate their dashboards with agents awaiting review. Keep in mind that the visibility store is eventually consistent. While the dashboard relies on the visibility store for discovery, the actual approval action relies on the `Update` request. When the operator clicks approve, the UI backend sends the update with the plan hash. This request routes directly to the worker hosting the workflow execution, bypassing the visibility store cache and guaranteeing

strong consistency. The update validator confirms the hash, the handler updates the state, the main workflow thread unblocks, clears the search attribute, and the agent safely resumes its execution.

8.4. Building Long-Running Autonomy: Timers, Schedules, Deadlines, and Catch-Up Semantics

To build agents that persist for days or weeks without consuming continuous compute resources, time must be treated as a first-class orchestration primitive. Rather than relying on standard loop constructs paired with thread sleeps—which block worker processes and lose state upon restart—you bind agent execution to durable timers and server-managed schedules. This allows an agent to suspend entirely, freeing worker resources, and reliably wake up only when a specific duration has passed, a condition is met, or an external signal arrives.

Sleep-and-Recheck Loops

A common agent pattern is waiting for external world state to change. Polling external APIs continuously from within an activity is an anti-pattern; it burns compute, inflates activity execution times, and complicates crash recovery. Instead, delegate the wait to the workflow orchestrator using durable timers combined with signals or condition evaluations.

Consider an agent orchestrating an infrastructure rollout that must pause until a manual safety check clears or a 24-hour SLA expires.

```
from datetime import timedelta
from temporalio import workflow

@workflow.defn
class InfrastructureRolloutAgent:
    def __init__(self) -> None:
        self._safety_cleared = False
```

```
@workflow.signal
def clear_safety_check(self) -> None:
    self._safety_cleared = True

@workflow.run
async def run(self) -> str:
    # Agent decides to wait for external clearance
    workflow.logger.info("Waiting for safety clearance.")

    # Suspend workflow until the condition is true or 24 hours
    pass
    met = await workflow.wait_condition(
        lambda: self._safety_cleared,
        timeout=timedelta(hours=24)
    )

    if met:
        # Execute next agentic step
        return await workflow.execute_activity(
            apply_rollout,
            schedule_to_close_timeout=timedelta(minutes=10)
        )
    else:
        # Fallback behavior
        return await workflow.execute_activity(
            escalate_timeout,
            schedule_to_close_timeout=timedelta(minutes=5)
        )
```

When `workflow.wait_condition` (or a standard durable sleep) is invoked, the workflow code blocks. Under the hood, the SDK emits a `TimerStarted` command to the Temporal server. The server stores this event in the workflow history and tracks the timer in its database. At this point, the worker process evicts the workflow state from its memory. No CPU threads remain blocked waiting for the timer. Hours or days later, when the server's clock reaches the deadline, it appends a `TimerFired` event to the history and pushes a new workflow task to the queue. A worker picks up the task, replays the history to reconstruct the workflow state, and resumes execution exactly where the agent left off.

If the `clear_safety_check` signal is received before the timer fires, the server immediately wakes a worker, injects the signal into the history, and evaluates the condition. This architecture makes it economically feasible to run millions of concurrently sleeping agents with zero active CPU utilization.

Server-Managed Schedules for Periodic Autonomy

While `wait_condition` is ideal for event-driven pauses, many agents require periodic re-planning. An agent monitoring a portfolio or scanning for security vulnerabilities must wake up on a strict cadence to pull new data and reassess its environment.

You could implement this by placing an infinite loop with a timer inside a single workflow. However, unbounded loops generate continuously growing workflow histories, eventually hitting the server's event limits and severely degrading replay performance. While you can mitigate history bloat by periodically calling `Continue-As-New`-truncating the event log and passing state to a fresh workflow run-relying purely on self-perpetuating workflows for cron-like behavior introduces fragility. If the workflow encounters an unhandled terminal error and fails, the periodic cycle stops permanently.

Server-managed schedules provide a robust alternative. A schedule is an independent identity on the Temporal server that starts workflow executions at specific times. The schedule guarantees the cadence, while the workflow encapsulates the logic for a single tick.

You configure a schedule to trigger an agent periodically using the CLI or an SDK client.

```
temporal schedule create \  
  --schedule-id "portfolio-monitor-daily" \  
  --cron "0 9 * * 1-5" \  
  --workflow-id "portfolio-monitor-agent" \  
  --task-queue "agent-tq" \  
  --type "PortfolioMonitorWorkflow" \  
  --overlap-policy Skip \  

```

```
--catchup-window 1h
```

Because the schedule operates independently of the worker processes, the Temporal cluster continues tracking the next firing time even if your entire worker fleet goes offline. When workers return, the server dispatches the pending workflow tasks according to your configured policies.

Catch-Up Semantics and Overlap Policies

When decoupling the schedule from the workflow, you must define how the system behaves when executions overlap or when downtime causes missed triggers. Agents typically interact with mutable external environments, meaning the context at the time a trigger was supposed to fire may no longer be valid.

The `OverlapPolicy` dictates what happens if a scheduled agent run takes longer than the schedule interval, or if a manual trigger overlaps with a scheduled run.

- **Skip:** If a previous run is still active, the new run is dropped. This is the default and is highly recommended for monitoring agents where the newest data supersedes older data. If the market data agent is still processing yesterday's tick, skipping today's tick prevents a stampeding herd of concurrent workflows analyzing redundant state.
- **BufferOne / BufferAll:** The server queues the new run(s) to start immediately after the current one finishes. Use this for reconciliation agents where every tick represents a distinct partition of work that must strictly execute in sequence.
- **CancelOther / TerminateOther:** The server halts the currently running workflow and starts the new one. This suits "latest intent wins" agents, such as an orchestration loop that syncs the

latest state from an external system. The in-flight sync is obsolete if a new sync triggers.

- **AllowAll:** The server starts the new workflow concurrently. This is only safe if the agent's actions are entirely isolated or idempotent, otherwise you risk concurrent state mutation.

When the Temporal cluster or the worker fleet experiences an outage, schedules may miss their firing times. The `CatchupWindow` defines the maximum acceptable delay for a missed schedule action to still be executed once the system recovers. The default is one year, meaning a workflow offline for a day will immediately attempt to backfill all missed ticks upon recovery.

For many agentic workloads, automatically running missed ticks is dangerous. If a daily notification agent is offline for a week, you do not want it sending seven identical emails in rapid succession when it recovers. To prevent this, configure a narrow `CatchupWindow` (e.g., 10 minutes or 1 hour). If the server evaluates a missed tick and finds it exceeds the window, it discards the action entirely and waits for the next scheduled time.

If a schedule is paused during an outage and the `CatchupWindow` prevents automatic execution, you can manually trigger a backfill once you verify the environment is safe. The backfill operation accepts a time range and generates workflow executions for all schedule intervals that fall within that range, injecting them into the task queue while respecting the defined `OverlapPolicy`. This gives operators precise control over recovery when an agent's external dependencies have experienced prolonged downtime.

Time-Boxed Intent: Activity Deadlines and Budgets

Beyond waking up reliably, an agent must bound the time it spends attempting to interact with the world. Timeouts are critical guardrails

against infinite loops of semantic failures or permanently hung API calls.

Activity timeouts compose a precise deadline hierarchy. When calling an LLM or an external tool, the `StartToCloseTimeout` is the most critical parameter. It defines the maximum time a single execution attempt of an activity can take. If a worker process crashes mid-generation, or an HTTP connection hangs indefinitely without dropping, the server relies exclusively on the `StartToCloseTimeout` to detect the failure and schedule a retry. Without it, a crashed worker might leave an activity in a pending state until a broader deadline fires, which could take hours. Setting a tight `StartToCloseTimeout` forces the server to promptly re-queue the activity to another healthy worker, enabling rapid self-healing.

The `ScheduleToCloseTimeout` bounds the total time for the activity, encompassing queue time, execution time, and all retry attempts. This represents the agent’s “time-boxed intent.” If the agent decides it must fetch a document to answer a user’s question, but the document retrieval service is degraded, the agent should not retry indefinitely. Setting a `ScheduleToCloseTimeout` of 10 minutes ensures that if the document cannot be retrieved within that window, the activity fails, returning a `TimeoutError` to the workflow. The agent can catch this error and transition to a fallback plan.

```
from temporalio.exceptions import ActivityError, TimeoutError

# Inside the workflow run method:
try:
    summary = await workflow.execute_activity(
        summarize_document,
        args=[doc_id],
        start_to_close_timeout=timedelta(minutes=2),
        schedule_to_close_timeout=timedelta(minutes=10),
        retry_policy=RetryPolicy(maximum_attempts=5)
    )
except ActivityError as e:
    if isinstance(e.cause, TimeoutError):
        workflow.logger.warning("Summarization timed out, switching
```

```
tactics.")
    return await execute_fallback_strategy()
```

At the workflow level, agents should also maintain operational budgets. While Temporal can run a workflow indefinitely, a poorly prompted agent might get stuck in an unresolvable reasoning loop, continuously attempting different invalid tool parameters. Enforce an iteration budget explicitly stored in workflow state: increment a counter on every plan-act-observe cycle. If the counter exceeds the budget, the agent terminates cleanly or routes itself to a human intervention queue, preventing unbounded LLM token expenditure.

Operational Flow: Scheduled Monitors and State Persistence

To synthesize these concepts, consider a scheduled daily portfolio monitor agent. This agent wakes up every morning, invokes activities to pull market data, uses an LLM activity to evaluate portfolio drift against a predefined policy, and requests human approval if a reallocation is required.

To maintain continuity across daily runs without accumulating an infinite workflow history, you must pass state from the concluding run to the next. Because a schedule triggers the workflow execution with a fixed input payload configured at schedule creation, you cannot pass arbitrary arguments directly between scheduled runs.

Instead, leverage the `LastCompletionResult`. When a scheduled workflow completes, it can return a serializable result. The server captures this value at the start time of the run. When the schedule triggers the next run, the workflow can query the schedule context to retrieve the result of the previous execution.

```
from temporalio import workflow
from datetime import timedelta

@workflow.defn
```

```
class PortfolioMonitorAgent:
    @workflow.run
    async def run(self) -> dict:
        # Retrieve state from the previous schedule tick
        last_result = workflow.info().last_completion_result
        if last_result:
            previous_drift = last_result.get("calculated_drift", 0.0)
            last_rebalance = last_result.get("last_rebalance_date")
        else:
            previous_drift = 0.0
            last_rebalance = None

        # 1. Fetch current market data
        market_data = await workflow.execute_activity(
            fetch_market_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

        # 2. Evaluate drift via LLM
        drift_analysis = await workflow.execute_activity(
            evaluate_drift,
            args=[market_data, previous_drift],
            start_to_close_timeout=timedelta(minutes=2)
        )

        current_drift = drift_analysis["drift_percentage"]

        # 3. Apply semantic logic and deadlines
        if current_drift > 5.0:
            # Requires rebalancing - trigger approval gate
            await workflow.execute_activity(
                request_human_approval,
                args=[drift_analysis],
                schedule_to_close_timeout=timedelta(hours=24)
            )
            last_rebalance = workflow.now().isoformat()

        # 4. Return state for the next scheduled tick
        return {
            "calculated_drift": current_drift,
            "last_rebalance_date": last_rebalance
        }
```

When designing state hand-offs, be strictly aware of payload limits. The Temporal server enforces a maximum blob size for event payloads, defaulting to 2 MiB, with an additional ~1 KiB overhead limitation

specifically for tracking the `LastCompletionResult`. If your agent attempts to return an exhaustive transcript of all past LLM conversations or raw tool outputs, the workflow will fail to complete and the schedule will halt. You must aggressively summarize and discard transient data, preserving only the durable, high-level intent, timestamps, and aggregated metrics necessary for the next run.

For agents that require continuous, uninterrupted loops rather than discrete periodic ticks, `Continue-As-New` remains the correct mechanism. However, calling `Continue-As-New` prevents access to `LastCompletionResult` in the new run; instead, you pass the compressed state directly as arguments to the new run's invocation. You must choose the operational model based on the agent's rhythm: use schedules for discrete, time-spaced check-ins, and `Continue-As-New` for continuous event consumption.

By combining schedules for periodic execution, precise overlap and catch-up policies to handle downtime safely, and structured timeouts for non-deterministic activities, you construct an agent kernel that operates reliably over vast timescales. The orchestrator guarantees that the agent wakes up when required, acts within its assigned budgets, and gracefully recovers from transient failures and extended outages without duplication or side-effect drift.

8.5. Choosing Cross-Boundary Durable Operations: Nexus vs Direct Calls vs Activity-Based Integration

When an autonomous agent needs to enact changes outside its immediate operational domain, it crosses a boundary of trust, availability, and schema ownership. An IT operations agent running in a corporate namespace cannot—and should not—have direct database access to re-

boot services owned by an SRE platform. It must request that action through an interface. If that interface fails, drops the connection, or processes the request ambiguously, the agent's durable control loop loses its synchronization with the real world.

To maintain correctness at scale, you must formalize how agents invoke external capabilities. Temporal provides three distinct paradigms for crossing these boundaries: direct client calls via activities, activity-based task queue integration, and Temporal Nexus durable operations.

To ground this, consider an IT ops agent that diagnoses a failing application and decides to restart its deployment. The restart procedure is owned by a remote SRE team. Using Temporal Nexus, you expose the SRE team's restart capability as a durable operation. The agent workflow invokes this operation directly, abstracting away whether the SRE team fulfills the request via a quick script or a multi-day saga.

```
package agent

import (
    "time"
    "go.temporal.io/sdk/workflow"
    "go.temporal.io/sdk/nexus"
)

// RestartRequest defines the contract shared between the Agent and
// SRE teams
type RestartRequest struct {
    ServiceName string
    ClusterID   string
    Reason      string
}

func ITOpsAgentWorkflow(ctx workflow.Context, alertID string) error {
    // ... (Agent planning and diagnosis steps omitted) ...

    req := RestartRequest{
        ServiceName: "payment-gateway",
        ClusterID:   "prod-us-east",
        Reason:      "Automated remediation: CPU saturation",
    }

    // Configure the Nexus operation invocation
```

```

    opts := workflow.NexusOperationOptions{
        Endpoint: "sre-platform-services", // Route configured in
        Temporal Server
        Operation: "RestartService",
        ScheduleToCloseTimeout: 2 * time.Hour,
    }
    ctx = workflow.WithNexusOperationOptions(ctx, opts)

    // Execute the durable operation across the boundary
    var operationResult string
    future := workflow.ExecuteNexusOperation(ctx, nexus.NewClient(),
        "RestartService", req)

    err := future.Get(ctx, &operationResult)
    if err != nil {
        // Handle semantic failure, fallback to opening a Jira ticket
        return escalateToHuman(ctx, req, err)
    }

    // Incorporate the successful result into the agent's observation
    state
    workflow.GetLogger(ctx).Info("Remote operation completed", "
        result", operationResult)
    return nil
}

```

This code defines a strict contract. The caller does not know how `RestartService` is implemented. It only knows that the Temporal server will route the request to the `sre-platform-services` endpoint, guarantee at-least-once delivery, and suspend the caller workflow durably until the SRE team's system resolves the operation.

8.5.1. Evaluating the Three Integration Models

Choosing the right integration model depends on your operational pre-requisites, latency requirements, and failure semantics.

Direct Client Calls

In this model, the agent workflow schedules a local activity. That activity executes a standard HTTP, gRPC, or GraphQL call to a remote

service.

The primary advantage is zero operational coupling: the remote team does not need to run Temporal, nor do they need to coordinate with your infrastructure. The integration is entirely standard. However, this places the entire burden of durability on the agent's local activity. Because external API calls are inherently non-deterministic, your activity must explicitly handle transport faults, rate limits, and ambiguous timeouts.

If the activity sends a POST request and the connection drops before receiving a response, the remote service might have executed the request. When Temporal retries the activity, it will send the request again. Therefore, direct client calls mandate the use of strict idempotency keys. You derive these keys from the workflow's deterministic context:

```
func TriggerRemoteRestartActivity(ctx context.Context, req
    RestartRequest) error {
    actInfo := activity.GetInfo(ctx)
    // Combine Workflow ID and Activity ID for a globally unique,
    // stable key
    idempotencyKey := fmt.Sprintf("%s-%s", actInfo.WorkflowExecution.
        ID, actInfo.ActivityID)

    // Inject the key into the remote HTTP request headers
    httpClient := &http.Client{Timeout: 30 * time.Second}
    httpReq, _ := http.NewRequest("POST", "https://sre.api.corp/
        restart", body)
    httpReq.Header.Set("X-Idempotency-Key", idempotencyKey)

    resp, err := httpClient.Do(httpReq)
    // ... response handling and error categorization ...
}
```

Use direct client calls when integrating with third-party SaaS providers or legacy internal services that cannot adopt Temporal.

Activity-Based Integration

In a shared infrastructure environment, the agent workflow can sched-

ule an activity on a Task Queue that a different team’s worker fleet listens to. The caller and the callee share the same Temporal cluster (and typically the same namespace, unless cross-namespace routing is specifically configured).

This model provides native Temporal retries and allows the remote team to implement the execution logic inside a Temporal worker. However, it tightly couples the two domains. The remote team’s activity execution events—including attempt counts, heartbeats, and specific activity error types—pollute the calling agent’s workflow history. If the remote activity type signature changes, it breaks the caller. Furthermore, the remote team cannot easily decompose their activity into a more complex workflow without forcing the caller to change their invocation from an Activity to a Child Workflow.

Use activity-based integration only within a single organizational boundary where teams share a unified deployment pipeline and code repository.

Nexus Durable Operations

Temporal Nexus is designed specifically for cross-boundary integration. A Nexus Operation is a formalized RPC mechanism backed by Temporal’s durable state machine. When a workflow calls `workflow.ExecuteNexusOperation`, the request is sent to a Server-managed Nexus Endpoint. The Server routes the request to the target namespace (which can be in an entirely different Temporal cluster) and queues it for the handler.

Nexus provides a queue-based worker architecture. If the handler service (the SRE team) is completely down, the caller workflow (the IT agent) does not fail; the operation request remains queued until the handler workers reconnect.

Furthermore, Nexus supports a unified synchronous/asynchronous

completion model. When the SRE worker picks up the request, it can complete it inline (like a fast activity), or it can return an asynchronous callback token and start its own long-running SRE workflow to fulfill the request. The calling agent workflow is completely unaware of this distinction; it simply blocks durably on the operation's `Future` until the final result is delivered.

8.5.2. Operational Ownership and Observability

Cross-boundary operations fundamentally shift how you observe and debug agent execution.

When you use direct client calls or activity-based integrations, the boundary is transparent. The agent's workflow history contains the raw input, the exact number of retry attempts, the delay between those attempts, and every intermediate exception thrown by the remote service. While this provides high visibility to the agent operator, it violates encapsulation. The SRE team handling the restart request might want to transparently switch from a direct API call to a complex rolling-restart saga without the caller noticing.

Nexus explicitly partitions observability. The caller workflow history only records that a Nexus Operation was scheduled, successfully started, and eventually completed or failed. The specific retries, child workflows, and internal activities used by the target namespace to fulfill the operation are entirely hidden from the caller.

This creates clean operational ownership. If an operation fails, the agent owner checks the agent workflow and sees a well-typed Nexus operation failure containing a standard error code. The SRE team checks their own namespace's workflow history to diagnose the internal mechanical failure that led to that error. You eliminate the friction of granting cross-team read-only access to workflow histories just to

debug integration issues.

8.5.3. Security and Authorization Boundaries

Granting an autonomous agent the ability to execute operations across namespaces requires stringent security controls. Without explicit boundaries, an agent with excessive permissions could become a vector for blind lateral movement—for example, an LLM misinterpreting a user prompt and deciding to invoke a `DropDatabase` operation in another cluster.

Nexus Endpoints act as policy enforcement gateways. You define endpoints at the Temporal Server level, not just in client code. An endpoint maps a specific caller identity to a target namespace and task queue.

```
# Example Temporal CLI command to create a secure Nexus endpoint
temporal operator endpoint create \
  --name sre-platform-services \
  --target-namespace sre-prod \
  --target-task-queue operations-q \
  --allowed-caller-namespaces it-agents-prod
```

By explicitly allowing only `it-agents-prod` to call the `sre-platform-services` endpoint, you establish a zero-trust boundary at the orchestration layer. The infrastructure rejects requests from unapproved namespaces before they ever reach the target worker fleet.

Inside the target worker's Nexus handler, you can extract standard headers populated by the Temporal Server to implement fine-grained business authorization. You can inspect the caller's namespace and apply policy logic: validating that the IT agent namespace is permitted to execute `RestartService` against development environments, but explicitly rejecting requests targeting production infrastructure without human approval.

8.5.4. Failure Semantics and Graceful Degradation

While cross-boundary durability guarantees that requests are not lost in transit, it introduces the risk of false confidence. A durable operation is not a guarantee of immediate execution. If the remote boundary is experiencing an extended outage, or if the SRE team's fulfillment workflow is stuck waiting on a manual approval gate, your agent workflow will suspend indefinitely.

You must design agents with explicit “give up and page” thresholds. Never execute a cross-boundary durable operation without configuring a strict timeout that aligns with the agent's business SLA.

When invoking a Nexus operation, apply the `ScheduleToCloseTimeout`. If this timeout expires before the target system fulfills the operation, the `Future.Get` call returns a timeout error. Your workflow must catch this error and transition cleanly into a fallback state.

```
// Configure a strict 15-minute SLA for the cross-boundary operation
opts := workflow.NexusOperationOptions{
    Endpoint:           "sre-platform-services",
    Operation:          "RestartService",
    ScheduleToCloseTimeout: 15 * time.Minute,
}
ctx = workflow.WithNexusOperationOptions(ctx, opts)

future := workflow.ExecuteNexusOperation(ctx, nexus.NewClient(), "
    RestartService", req)

var result string
err := future.Get(ctx, &result)
if err != nil {
    if temporal.IsTimeoutError(err) {
        workflow.GetLogger(ctx).Warn("Remote operation SLA breached,
            escalating")
        // The operation might still be running remotely, but the
        agent must move on.
        // Transition to an alternative tool: open an incident ticket
        .
        ticketReq := IncidentTicket{
            Title: "Agent SLA breached waiting for automatic restart
```

```
    ",
        Severity: "High",
    }
    return workflow.ExecuteActivity(ctx,
        CreateIncidentTicketActivity, ticketReq).Get(ctx, nil)
}
// Handle other semantic failures from the remote service
return err
}
```

If the agent decides to abandon the operation and move on to a fall-back tool, you should explicitly attempt to cancel the in-flight operation. Temporal propagates workflow cancellation requests across the Nexus boundary. When you cancel the context used to invoke `ExecuteNexusOperation`, the Temporal server sends a cancellation signal to the target handler.

However, enforcement is up to the target system. The remote SRE workflow must be designed to listen for cancellation signals and safely abort its internal tasks. If the remote workflow is already in the middle of a critical, non-interruptible section (such as actively draining connections from a pod), it may ignore the cancellation and complete the operation anyway. Your agent must be designed to tolerate scenarios where it timed out and escalated, but the side effect eventually completed on the remote system hours later. Define compensations—such as sending a follow-up Slack message clarifying that the delayed restart was finally observed—rather than assuming an aborted operation left no external trace.

