WHITE PAPER

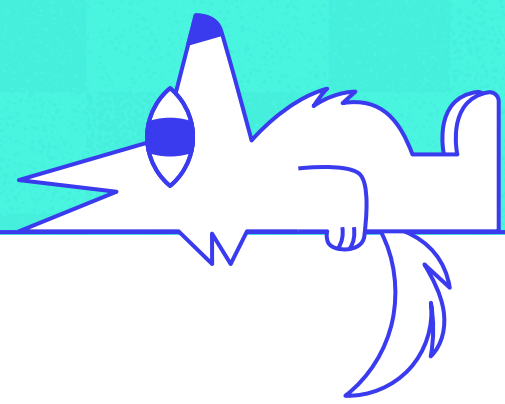
Run Rules: QA Wolf's declarative coordination model

A practical answer to testing cross-system complexity

QA WOLF

Table of Contents

Abstract



Every modern software application depends on linked user journeys—different people and systems acting across devices, services, and time to achieve a shared result.

Consider a rideshare app

- A rider requests a trip from their phone.
- A dispatch service matches an available driver.
- The driver accepts the request and navigates to the pickup point.
- When the driver arrives and the dispatch system confirms the rider's presence, the trip begins.

In this system, each participant acts independently, on different devices and timelines, yet their actions depend on one another. But today's testing tools can't represent that complexity. They assume a single actor, a single timeline, and a shared execution context. There's no built-in support for concurrent actors, no way to express dependencies across workflows, and no faithful model of distributed timing.

Run Rules—QA Wolf's test orchestration system—solves that problem. It enables end-to-end (E2E) tests that mirror how real systems behave: distributed, asynchronous, and multi-actor. Each participant's behavior is represented as an independent workflow, executed in isolation but coordinated declaratively via a dependency model rather than a control script. Run Rules compiles these declarations into a directed acyclic graph (DAG) that represents causal and concurrent relationships among workflows. It evaluates data dependencies in real time, launches workflows as soon as their inputs resolve, and preserves causal order without shared state or fragile sequencing.

This paper explains how Run Rules makes linked user journeys testable at scale and what it took to build a coordination model that faithfully reflects real-world behavior.

Definitions

Workflow
A sequence of steps representing one actor's behavior in the system (e.g., "rider requests trip," "dispatch matches driver," "driver accepts trip").

Actor
A participant in a workflow such as a rider, driver, or dispatch service (API).

Directed acyclic graph (DAG)
The structure Run Rules uses to model workflow dependencies. Each node is a workflow; each edge defines a declared relationship (e.g., "`driver_accept` depends on `dispatch_match`").

Coordination
The synchronization of multiple actors that may act sequentially or concurrently. Run Rules supports two primary models:

- Turn-based coordination: one workflow's outputs unlock another's (e.g., `rider_request` → `dispatch_match` → `driver_accept`).
- Real-time coordination: concurrent workflows synchronize on shared conditions (e.g., `driver_arrival` and `rider_presence` before ride start).

Run Rules
QA Wolf's declarative coordination system. Run Rules defines when workflows become eligible to run and what data they consume or produce. Our system interprets these rules to determine execution order dynamically.

Container
A sandboxed runtime provisioned for a single workflow. Containers isolate environments completely—no shared memory, cookies, or session state—mirroring the real-world isolation between real-world actors.

Modeling linked user journeys

Many modern software features involve interactions between multiple participants or systems. A rideshare trip is one example:

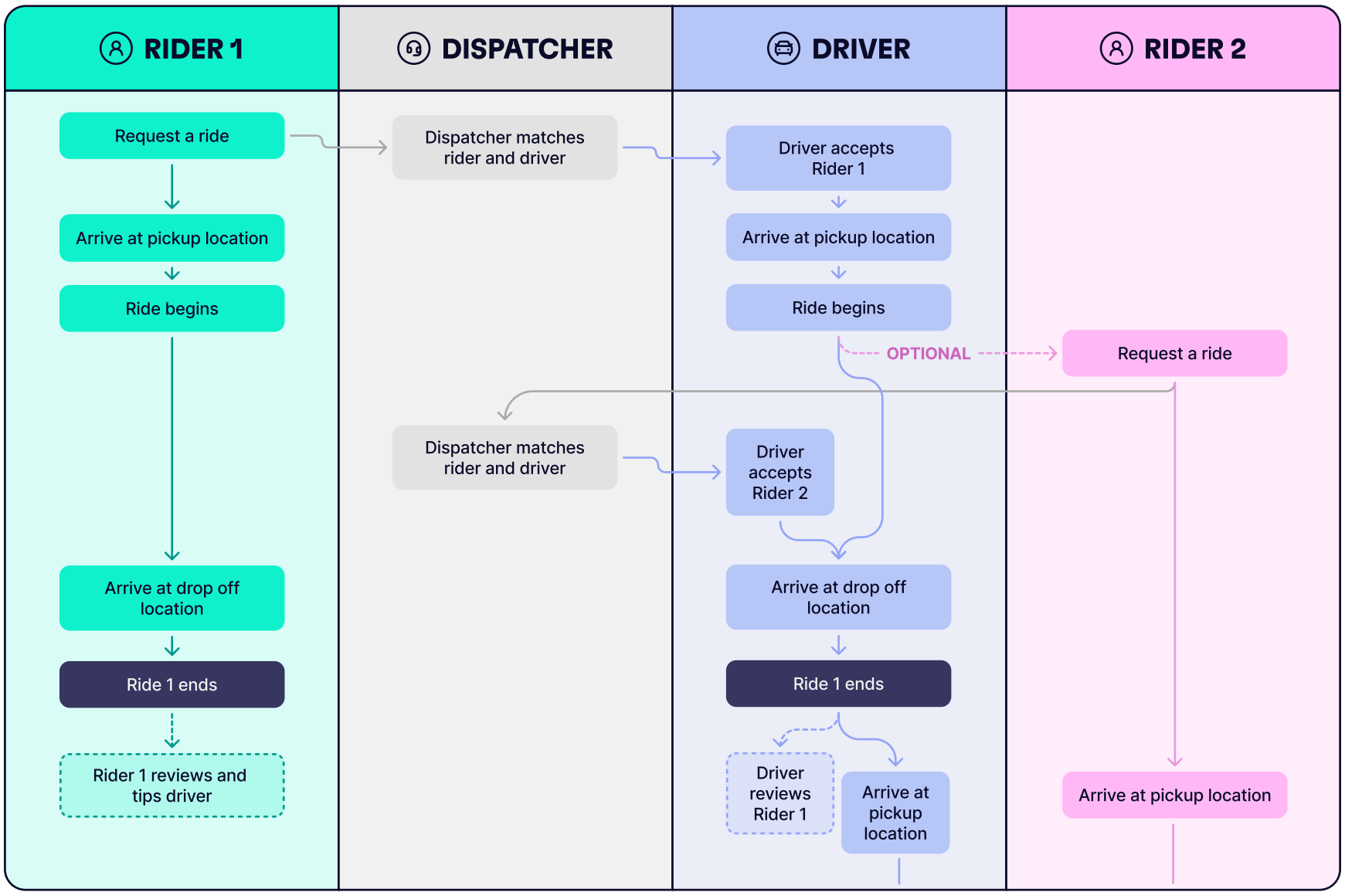


Fig. 1 — Individual actors participate in separate workflows, which are linked together by a common goal.

1. The **rider** requests a trip from their phone.
2. The **dispatch service** receives the request and assigns a **driver**.
3. The **driver** accepts the assignment from another device.
4. The driver arrives at the pickup point while the dispatch system confirms the rider is present using location data.
5. Once both conditions are met, the trip begins.
6. The rider can tip the driver. The driver can accept another assignment.
7. Once they reach the destination, the trip ends.
8. The rider can review the driver. The driver can review the rider.

Each participant operates in its own environment and interacts with the system at different times. Their actions are related through shared data—trip requests, driver assignments, arrival updates—not through a single session or runtime.

The overall trip emerges from these exchanges: one actor's output becomes another's input, and timing determines how the system responds. Linked user journeys like this involve both sequence and overlap. Some actions depend on earlier results, while others occur in parallel.

To represent them accurately, a model must describe the relationships between participants—the flow of data, the timing of actions, and the conditions that connect one step to another.

Procedural coordination and its technical limitations

Traditional testing frameworks coordinate procedurally: they run a fixed sequence of actions—run A, then run B, then run C. That approach works for single-user tests but doesn't reflect how modern systems behave when several actors operate independently on different devices or timelines.

Traditional frameworks (Playwright, Cypress) lack any concept of coordination, requiring QA engineers to collapse multi-actor flows into a single control thread to test them. Even when used through external test runners, the lack of any programmatic coordination layer forces test developers to pass state imperatively through scripted steps rather than through declared dependencies between workflows. The design (or, really, lack thereof) makes it impossible to represent actions that happen in parallel or depend on data produced elsewhere.

As a result, procedural test code that attempts to validate linked user journeys inevitably creates one or more of the following technical problems:

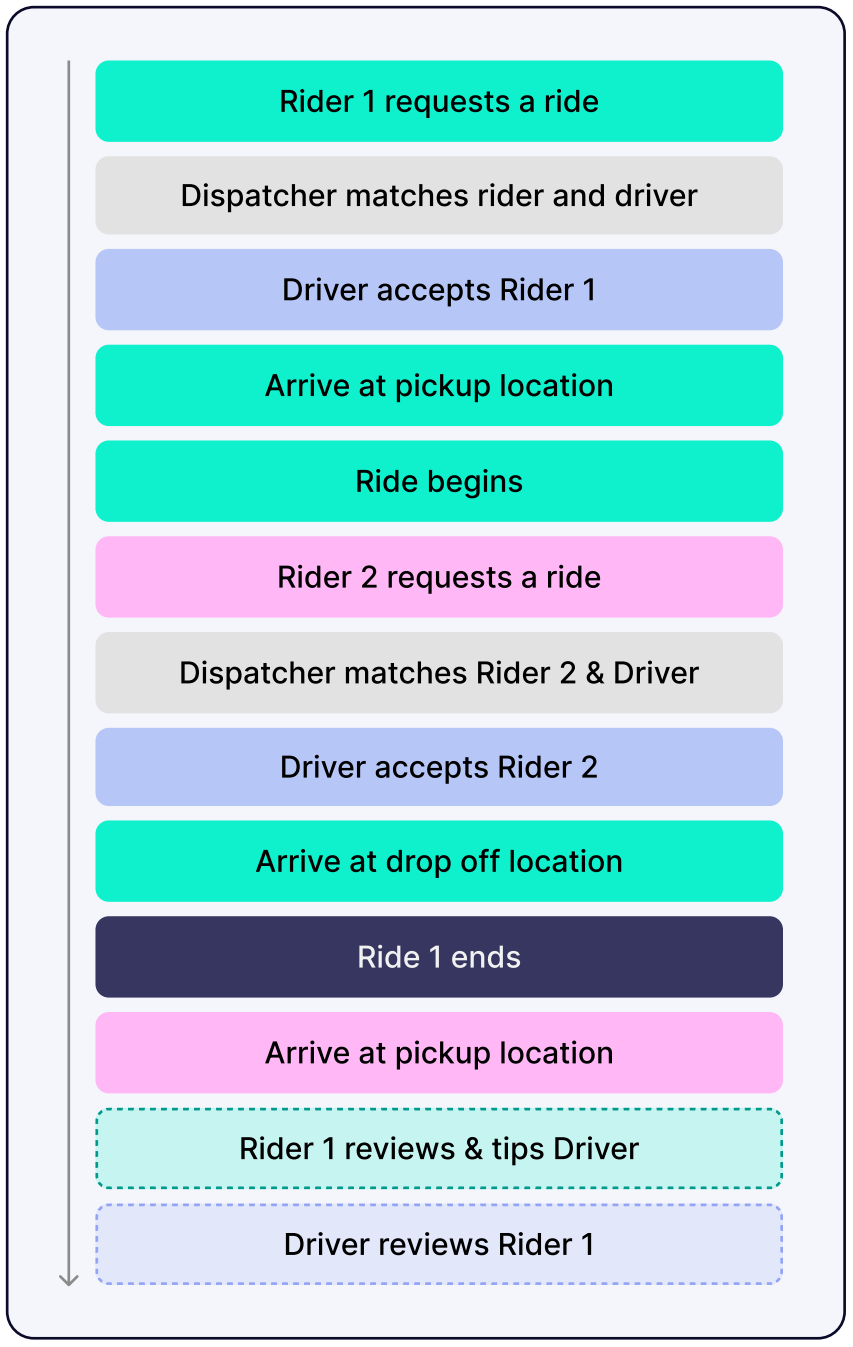


Fig. 2 — A procedural version of a linked user journey.

1. Concurrent execution can't be represented.

Procedural frameworks control every workflow through one execution thread, so each actor's actions must run in order. Actors that should operate at the same time—like a rider and a driver—end up running one after another. This single-threaded structure prevents the system from modeling independently scheduled actors or exposing issues that depend on true parallel behavior, such as synchronization errors and timing drift between devices.

2. Tests use static waits, which don't accurately reflect real system timing.

Procedural tests often rely on fixed sleeps or polling loops to simulate the pauses between actions. These waiting strategies ignore variable network latency, backend processing time, and user response delays, making timing-sensitive issues hard to detect or reproduce.

3. Shared state makes results unreliable.

A single test runner reuses the same memory, session, and/or device state to act as multiple users. That overlap allows one actor's data to leak into another's environment, producing nondeterministic results and making test failures difficult to isolate.

4. Workflow relationships are buried in code.

Rules about which workflow depends on which output are embedded inside procedural logic. When the product changes, QA engineers must trace those dependencies manually through the test code, increasing maintenance time and the risk of error.

5. Failures trigger unnecessary full-suite reruns.

When one workflow fails, procedural suites must restart the entire sequence from the beginning. This re-executes successful workflows, rebuilds valid state, and consumes unnecessary runtime resources.

6. Causal relationships can't be observed.

Procedural tests log events in order, but don't record the data dependencies between them. When something breaks, QA engineers can see what happened but not why—for example, which workflow was waiting on which input, or what condition allowed the next step to run. Without that causal context, debugging distributed workflows becomes slow and error-prone.

Coordinating workflows declaratively

In distributed systems, coordination naturally follows the data: processes run when the information they depend on becomes available. Run Rules brings that same approach to testing. Instead of scripting when each flow should run, QA engineers describe what data each test produces and what data is required by others. With that information, Run Rules works out the execution order and starts each flow the moment its inputs arrive.

The foundation of this model—the ephemeral suite-scoped variable space—and the Run Rules govern how workflows discover dependencies, become eligible to run, and synchronize—either turn by turn or in real time.

Ephemeral suite-scoped variables: the foundation of Run Rules

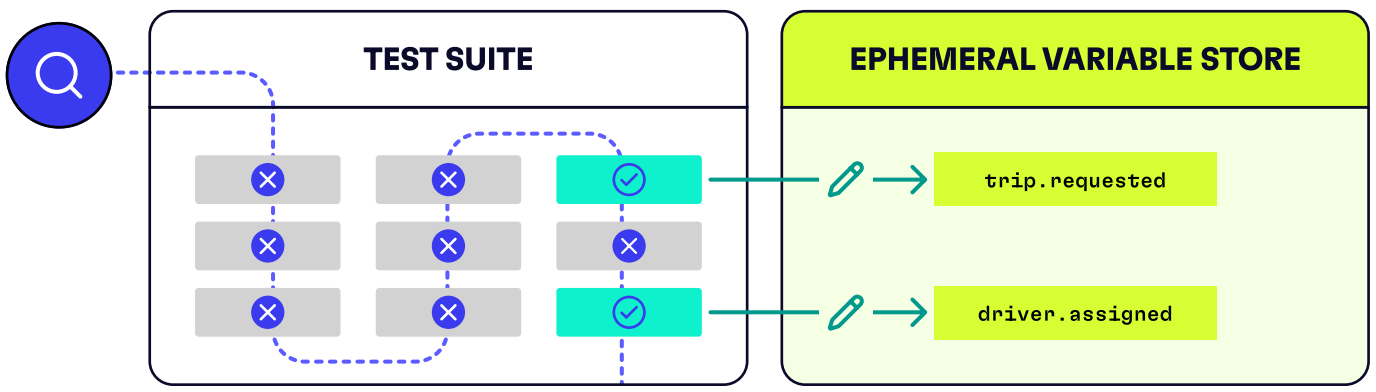


Fig. 3.1 — Run Rules searches the test suite for keys and writes them to the variable store.

Run Rules coordinates workflows through a shared variable space that exists only for the lifetime of a run. This ephemeral suite-scoped variable space is created when the run begins and is discarded when it ends. Workflows read and write values into this space using two custom methods that are injected into the runtime environment:

- **`setOutput()`** is used by producer workflows to create a key-value pair in the suite scope.
- **`getWorkflowInput()`** is used by consumers to read from key-value pairs in the suite scope.

This way, workflows never share memory, sessions, or runtime state.

This model eliminates global fixtures, cross-workflow setup code, and implicit state. Each run receives its own isolated namespace, preventing data from leaking across runs or retry attempts. Because the variable space lives outside any individual test runner, workflows can execute in parallel across machines, devices, or environments while still coordinating through declared data.

Values published with **`setOutput()`** appear in the variable space immediately. These values become synchronization points for other workflows, replacing brittle timing logic and hard-coded orchestration. Workflow execution is based on data availability—not when someone else finished.

This minimal abstraction—publish a value, consume a value—is the foundation of all coordination in Run Rules.

Run Rules: analysis at setup and eligibility at runtime

Run Rules operates at two levels: provisioning-time dependency analysis and runtime eligibility enforcement.

Provisioning-time dependency analysis

Before a run begins, Run Rules analyzes every workflow and extracts its calls to **`getWorkflowInput()`**. From these calls, Run Rules constructs the workflow's concrete dependency set—its **needs**. The system assembles these dependencies into a directed graph and validates it. Cycles with no seed values, missing producers, and other invalid structures are detected during this provisioning phase, before any workflow executes.

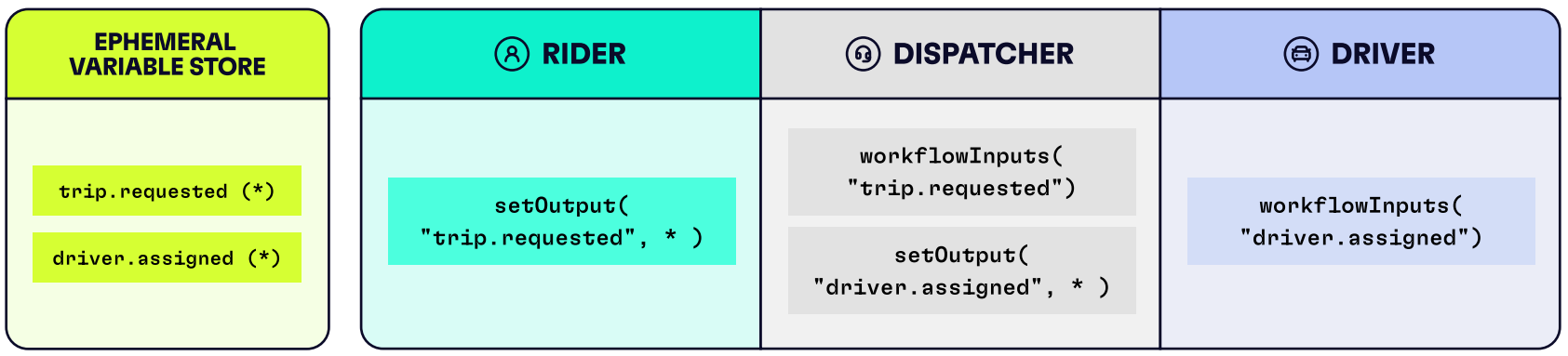


Fig. 3.2 — At provisioning time, Run Rules understands each workflow's dependencies in order to prevent runtime errors.

Runtime eligibility enforcement

At runtime, Run Rules enforces dependencies dynamically as workflows execute. Each run begins with an empty suite-scoped variable space. Workflows start immediately in their isolated environments and run normally until they reach a declared dependency via **`getWorkflowInput()`**. If the required variable already exists, execution continues; if not, the workflow pauses at that point. There is no polling loop or custom waiting logic—Run Rules resumes the workflow automatically when the variable becomes available.

Producers run before consumers only when the user declares a dependency between them. Run Rules does not infer ordering from data alone; it enforces the relationships expressed through **`getWorkflowInput()`**. Once a workflow declares that it needs a value from another workflow, Run Rules ensures it does not proceed past that point until the value has been published.

If a workflow fails, only that workflow is retried. Outputs from successful workflows persist across attempts, allowing retries to reuse upstream values without re-running the entire suite. Each retry sees the same inputs as the original attempt, preserving the intended causal sequence.

Run Rules therefore provides deterministic, data-driven coordination: workflows advance exactly when their declared dependencies allow—never earlier, never out of order, and never through orchestration code embedded in test logic.

Coordination patterns supported by Run Rules

Run Rules supports both turn-based and real-time coordination.

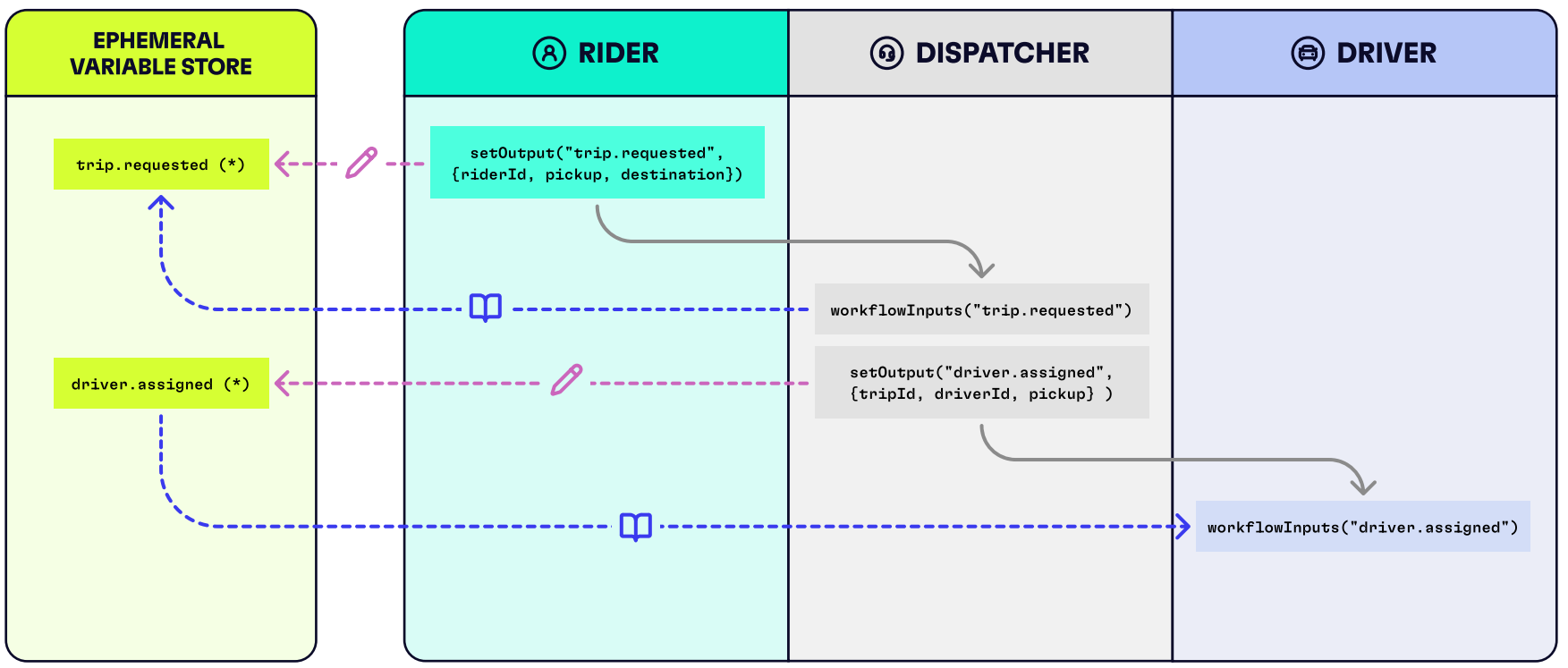


Fig. 3.3 — Each actor can produce or consume variables that are in the ephemeral value store. Any actor that consumes a value from the store can not begin until the variable it needs exists in the store.

Turn-based coordination

A downstream workflow depends on values produced by an upstream workflow. Run Rules enforces this ordering automatically: the downstream workflow cannot begin until the upstream workflow's outputs exist. This models sequential behavior—such as "the rider requests a trip before the driver sees it"—without explicit waits or control flow.

Real-time coordination

Independent workflows can run concurrently but synchronize on specific conditions. A workflow becomes eligible the moment its required variables appear, even while other

workflows are still executing. This enables true parallelism, where actors proceed independently but meet on shared conditions such as presence, proximity, or matched IDs.

Turn-based and real-time coordination require no different logic: both emerge from the same eligibility rules applied to the ephemeral variable space.

Observability through declarative coordination

Because dependencies are explicit and workflows run in isolation, Run Rules can reconstruct the causal execution sequence. Each workflow records its own logs, traces, and artifacts. Run Rules correlates these per-workflow timelines with the variable-space events to produce a suite-level causal trace: which workflow waited on which inputs, when each dependency appeared, and how failures propagated.

This makes multi-workflow behavior observable, debuggable, and explainable—qualities that procedural orchestration often complicates.

Why it matters

Declarative coordination shifts responsibility from test authors to the system:

- **True parallel execution reveals real concurrency issues.** Workflows operate independently and start as soon as their inputs exist—no sequential bottlenecks, no shared timeline.
- **Isolation ensures deterministic behavior at any scale.** Each workflow runs in its own container with no shared session or state. The same suite produces the same result regardless of the level of parallelism.
- **Targeted retries preserve causal integrity and reduce time.** When a workflow fails, Run Rules retries only the node that failed, reusing upstream outputs. No full-suite reruns, no corrupted state.
- **Declarative relationships make coordination transparent.** Dependencies are visible in the generated graph, not hidden in orchestrated test code. Adding or modifying a workflow requires updating its declarations, not rewriting control flow.
- **Causal traces expose how and why failures occurred.** Engineers can inspect the full chain of events: which workflow produced which value, when dependencies were resolved, and what triggered each start.

Run Rules models coordination the way real systems behave—independent actors respond to changing data, and system timing emerges from propagation and constraints. This extends end-to-end testing beyond linear paths into distributed, data-driven scenarios that match the systems teams build and depend on today.

The impact of coordination-aware testing

Run Rules expands the scope of what automated testing can verify. By coordinating isolated workflows across users, devices, and systems under test, it changes the way QA engineers approach the testing of linked user journeys, moving from procedural sequences to explicit, reusable components built into the system and supported by the testing framework. QA engineers can now represent real behavior—where one actor begins a flow, another joins later, and both operate under different timing and platform conditions—without flattening those interactions into a single script.

Because coordination is declared rather than scripted, suites become adaptive and resilient. When a workflow fails, Run Rules retries only the affected node, reusing valid upstream outputs. When dependencies are unmet, it holds related workflows instead of re-executing the entire suite. That makes coverage faster to iterate on and failures easier to isolate.

The same declarative structure enables deep observability. Every workflow maintains its own timeline, while Run Rules reconstructs a causal trace showing how dependencies are resolved over time, across roles, and across platforms. Engineers can trace cause and effect directly—seeing not just what failed, but why and in what context.

Patent Pending

PATENT APPLICATIONS HAVE BEEN FILED COVERING ASPECTS OF THE TECHNOLOGY DESCRIBED HEREIN.

Ownership: QA Wolf, Inc.

Inventors:

Brendan Downing
Eric Dobbertin
Eric Eidelberg
Erkan Erol
Jaden Lemmon
Jon Perl
Michael Price
Nishant Shukla