

Orkes Conductor 5.3.2

Kyle Kingsbury
2026-10-20

Orkes Conductor is a workflow orchestration platform which runs user-defined tasks and routes data between them. We tested Orkes Conductor versions 5.3.2 through 5.5.0-rc2 and found unexpectedly frequent concurrent execution of individual tasks, one issue that could significantly delay workflow execution, four minor issues with `startWorkflowErrors` or null IDs, and three serious problems which caused Conductor to permanently abandon or forget workflow executions, or to mark them completed even though their tasks were never run. Orkes has released version 5.5.6 with patches for all serious safety issues, as well as reducing concurrent execution and fixing two `startWorkflowErrors`. The other two `startWorkflowErrors` are under investigation. Delayed execution is consistent with Conductor’s design parameters and need not be addressed. Orkes has published [a companion blog post](#) to this work. This research was funded by Orkes Inc., and conducted in accordance with the [Jepsen ethics policy](#).

1 Background

A **workflow orchestrator** is a software system which executes computations made up of small *tasks*. Tasks are connected in a *workflow*, which routes the outputs of some tasks to the inputs of others, determines which tasks may be scheduled and when, and so on. The orchestrator manages workflows and specific executions of them, runs their tasks, persists state, and tries to drive each execution to completion.

Conductor is a workflow system **originally built by Netflix** and open sourced in 2016.¹ The open source version, **Conductor OSS**, is now maintained by the **Conductor OSS Foundation**. This foundation is led by **Orkes**, whose founders helped build Conductor at Netflix. Orkes also offers a **hosted version of Conductor** with additional features, called *Orkes Conductor*. This analysis focuses on Orkes Conductor, which we call *Conductor* for short.

In Conductor, a *workflow definition* is a JSON structure that defines which tasks must be executed, their order, what their inputs and outputs should be, and so on. Each workflow definition is identified by a name and a version. For example, a workflow definition could specify that to adopt a cat, one must first complete an application, then schedule a vet visit, and receive vaccination records from the vet. A workflow may also accept an input (say, the name of the cat) and produce an output (like a certificate of adoption).

A workflow definition may be run many times—say, once for each cat to be adopted. Each instance is called a *workflow execution*, but we often call an execution a “workflow” for short. Each execution is identified by a unique *workflow ID*. During an execution a task may also be attempted multiple times (e.g. in case of

failure), and each attempt receives a unique *execution ID*.

Some tasks are built into Conductor. Some are *system tasks*, like making HTTP requests or submitting events to a log like Kafka. Others are called *operators*—these provide control flow like conditionals, fork/join, loops, and so on. Operators may execute EC-MAScript code to perform user-defined logic. Finally there are *worker tasks*, which are written by users to perform the “business logic” of the system.

To start an execution of a workflow, one can make an API call to Conductor, submit an event through a message broker, or use Conductor’s built-in cron-style scheduler. Once acknowledged, Conductor schedules the workflow’s tasks and attempts to drive it to completion.

1.1 Architecture

Conductor **has five major components**. Conductor servers are stateless processes which manage workflows and schedule execution. They provide an API and user interface, interact with external services, and track state. The definitions of workflows, the state of their executions and the results of tasks are stored in a database like Postgres, Cassandra, or Redis. Conductor also uses a queue—often Redis—to dispatch work to various processes, and to store short-lived state. Finally, there are two pools of stateless workers: standard workers which execute Conductor’s built-in tasks, and custom workers, which are written by users using **SDKs for various languages**. All workers **connect to a Conductor server**, poll the server for tasks, execute them, and send results back to the server.

¹There are two unrelated orchestration systems with similar names: one from Microsoft, also named **Conductor**, and one called **Postgres Conductor**. There is also an unrelated **AI marketing service** called Conductor.

Each component may be scaled independently. Any number of servers and workers can run concurrently, and nodes may be killed and restarted at any time. In version 5.3.2, the database and queue were both required to preserve all acknowledged writes. However, version 5.4.0 introduced a new option which enabled the use of a lossy storage system for all queues except for the main *decider queue*.

1.2 Guarantees

Conductor guarantees “**at least once message delivery**”, ensuring that all messages are persistent and will be delivered to task workers one or more times. Conductor also guarantees that workflows eventually terminate. “If a workflow has started, it will run to completion as long as all its tasks are completed.” This does not necessarily mean that workflows and tasks execute successfully. **Tasks can run out of retry attempts**, fail with non-retryable errors, or be explicitly canceled—and a similar state machine applies to workflow executions. However, workflows should eventually converge on a **terminal status**: COMPLETED, FAILED, TIMED_OUT, or TERMINATED.

When tasks are executed, each attempt receives a unique execution ID, and also a unique *retry count*, which **indicates the number of attempts already performed**.

Orkes says that for each workflow execution, each instance of a task has at most one visible successful execution. Tasks may execute successfully multiple times—for instance, if a Conductor server loses its connection to the database just before persisting the completion of the task execution. However, once the output of a successful task becomes visible, no task, workflow, or API call will observe a different output.²

2 Test Design

To verify these properties, we built a **test harness** for Orkes Conductor using the **Jepsen testing library** and the **Conductor Java SDK** at version 5.0.1. We used this harness to test Orkes Conductor 5.3.2, 5.4.0, and 5.5.0-rc1 and -rc2.

Orkes typically deploys Conductor in cloud environments with Postgres (or a Postgres-compatible database, like their in-house **ParticleDB**) as the primary database, and Redis (or a compatible service like **Valkey**) as the queue. To approximate these deployments, we set up clusters of ten Debian machines, both in containers and as separate EC2 VMs: a single Postgres node for the database, either a single Redis node or Redis Cluster for the queue, and the remaining nodes split evenly between Conductor servers, Conductor’s standard workers, and a **custom worker** we wrote using the Java SDK.

Our **test workload** defined a single workflow with one task, handled by our custom worker. The task slept

up to 30 seconds to simulate work being done, then **returned a randomly generated output**. It also **logged its execution** with the test harness, so we could tell exactly which tasks had run and when, along with their observed inputs and intended outputs.

Our test proceeded in three phases. During the **main phase**, the test harness called `startWorkflow()` with unique inputs (to distinguish executions from one another), **kept track** of which executions were in-progress and which were terminal, and checked the status of specific executions using `getWorkflow()`. To discover executions when the call to `startWorkflow()` timed out or failed, we periodically called `getRunningWorkflow()` to list all in-progress executions. During the main phase we also injected **a variety of faults**, including pausing and killing processes, as well as network partitions between Conductor servers and Postgres, Redis, and worker nodes.

During the **recovery phase**, the test ended network partitions, restarted and unpaused processes, etc., then let the cluster recover. To figure out when the system had converged on a steady state, we polled the depth of Conductor’s decider queue. When this queue reached zero, Conductor had nothing more to do and the test moved to the final phase.

During the **final phase**, the test read the state of every workflow execution it had ever seen. We then searched for **a variety of unexpected behaviors**:

Backend Error: Some request returned an unexpected backend error.

Concurrent Execution: Two or more workers attempted to execute the same task in a single workflow execution at the same time.

Different Task Input: Two views of the same task in the same workflow execution had different inputs, or a task’s input was inconsistent with the workflow’s input.

Different Workflow Input: A single workflow execution had two different inputs.

Different Workflow Output: A single workflow execution had two different outputs.

Duplicate Workflow ID: Two calls to `startWorkflow` returned the same workflow execution ID.

G1a: A `startWorkflow` request returned what appeared to be a definite failure, but a workflow with that requested input was later visible to some reader.

Impossible Output: A workflow execution’s output was inconsistent with the outputs of its tasks.

Incomplete: A workflow execution was not in state COMPLETED during the final phase.

Missing: A read failed to observe a workflow execution after we knew it existed.

Never Started: A workflow came into existence with no corresponding `startWorkflow` request.

Null Workflow ID: A `startWorkflow` call returned null, rather than a workflow ID string.

Task Never Executed: None of a workflow’s tasks were ever executed.

²While operations within Conductor should see only a single successful task execution, tasks may have visible side effects on systems outside Conductor. Well-behaved tasks should therefore be idempotent and deterministic.

3 Results

3.1 Tasks Execute Concurrently (#1)

Under normal operation, healthy Conductor clusters occasionally executed the same task for the same workflow execution on two workers concurrently. This is not surprising, since **mutual exclusion is impossible in asynchronous networks**. However, it is worth keeping in mind that even very short-lived tasks may be executed at the same time! In **this test run**, workflow 3919784b-7691-11f1-a292-00163e983f22's single task began on worker n9 at 70.110 seconds, then on worker n10 at 70.113 seconds. It completed on n9 at 70.114 seconds, and on n10 less than a millisecond later.

The documentation for Conductor OSS describes a distributed locking system which “prevents race conditions when multiple server instances evaluate the same workflow concurrently”. Orkes Conductor also has a locking mechanism whose lease time is configured by `conductor.workflow.lockLeaseTime`. While this lock can reduce the frequency of concurrent execution, it cannot prevent it entirely.

With our single-task workflows, the default lease time of one minute, two Conductor servers, three workers, near-zero network latency, short-lived tasks (p50 ~500 µs, p95 ~5 ms, p99 ~10 ms), and roughly 1,100 workflows per second, we observed concurrent execution roughly once every six hours in healthy clusters. Reducing the lease time to one second allowed concurrent execution every few minutes. Production frequency may vary.

This problem was exacerbated by a **time of check to time of use** bug in the sweeper process which re-enqueues stuck tasks. Imagine a sweeper reads a task in state SCHEDULED. A worker then executes that task, changing its state in the database to IN_PROGRESS and removing the task message from the queue. The sweeper then checks the queue and finds no message for the (no longer) scheduled task; it re-enqueues it, leading to duplicate delivery. Orkes reports this issue (#1) is fixed in version 5.5.6.

Jepsen also suggests that Orkes update the Conductor documentation to make it clear that tasks may execute concurrently. It may also be possible for a worker to observe `retryCount = 0` (suggesting it was the first to run) even though some other worker had already successfully executed that task.

3.2 Tasks Stall Before Execution (#2)

When using a Postgres primary DB and Redis as the queue, Orkes Conductor 5.3.2 often allowed a handful of workflows to stall prior to task execution. While stalled, each workflow would show state RUNNING, and the workflow's task would appear as if it were running on a specific worker. However, the worker's `execute()` method was never called for that task. This state would persist until the task's response timeout triggered, at which point Conductor would reschedule the

task. We observed this behavior in response to a broad variety of faults—for instance, if Redis were killed or paused; or Postgres were killed; or a server were killed or partitioned from Redis, Postgres, or a worker; or if a worker were killed.

For example, **consider this test run**, in which two out of 20 workflow executions stalled. Workflow execution 3f71d24a-7619-11f1-b911-00163eb95b8c was acknowledged by server n3 at 08:23:43.588, its sole task was scheduled at 08:23:43.593, and recorded as IN_PROGRESS on worker n10 at 08:23:44.615. However, worker n10 never called `Worker.execute(task)` with that workflow ID. Server n3 appeared to execute normally until 08:24:42, when it logged a read timeout against Redis.

In general, a pause, crash or partition isolating a Conductor server from Redis would reliably stall a handful of workflows by an hour—the default response timeout. Users could lower the `responseTimeoutSeconds` parameter to shorten the delay.

Since the tasks were eventually executed within the time implied by the response timeout window, this does not violate Conductor's claimed guarantees. Orkes opted not to change this behavior. (#2).

3.3 Null Workflow IDs (#3)

When creating a workflow, the Java client's `startWorkflow()` method normally returned a workflow ID as a string. However, it rarely returned null, as in **this test run**, which involved killing the Redis process.

Orkes is investigating this issue (#3). While the root cause is unknown, version 6.0.2 of the Java client includes a mitigation.

3.4 Syntax Error (#4)

Occasionally, in healthy clusters without faults, calls to `startWorkflow` would return a syntax error. There seemed to be nothing unusual about the requests: they provided exactly the same kind of input as every other call to `startWorkflow()`. Here's a **typical example**:

```
{:type      :invoke,
 :process   12,
 :f         :start-workflow,
 :value     {:name "greetings",
             :version 1,
             :input {"name" "jepsen 10"}}}
```

This threw `BACKEND_ERROR - ERROR: syntax error at or near "\"null\""\n Position: 13.`

Conductor stores workflow executions in a table partitioned by date. When a new partition was created or rotated, a race condition allowed the partition manager to observe a temporary empty map of shards to tables. Conductor then looked up the shard in that empty map, and constructed an SQL statement like `INSERT INTO null ...`. Postgres threw a syntax error, and that error message was relayed back to the

client. In version 5.5.6 Conductor ensures that the shard map is initialized before accepting writes, which should prevent the issue (#4).

3.5 Duplicate Key Error (#8)

In the release candidate build 5.5.0-rc1, starting a workflow could very rarely return a Postgres duplicate key error. For example, take [this test run](#), with Postgres kills and pauses. A single call to `startWorkflow` returned this error:

```
BACKEND_ERROR - ERROR: duplicate key value
violates unique constraint
\"workflow_archive_shard_00_pkey\"\\n
Detail: Key (org_id, workflow_id)=
(0000, 72e49555-81eb-11f1-8ccb-00163eb95b8c)
already exists.
```

This seems to have been caused by a retry mechanism added to 5.5.0-rc1: when a Conductor server failed to insert a new workflow execution record into Postgres, it would try again, and could conflict with its previous write. That error message was then relayed to the client. In version 5.5.6, Conductor should ignore duplicate ID errors on retries (#8).

3.6 Aborted Read (#6)

In 5.3.2, 5.4.0, 5.5.0-rc1, and 5.5.0-rc2, using a Postgres database and a Redis queue, killing Conductor servers often caused calls to `startWorkflow()` to throw an error like Failed to connect to n3/10.0.3.103:8080. This sounds like a definite failure—after all, if the client couldn't open a connection, how could it possibly start the workflow? However, many of these workflows were actually executed by workers and visible to `getWorkflow` calls. In [this short run](#), 1,752 out of 25,920 failed calls to `startWorkflow` were actually successful, like this one:

```
{:index 65834,
 :time 11146587755,
 :type :fail,
 :process 24,
 :f :start-workflow,
 :value {:name "greetings",
         :version 1,
         :input {"name" "jepsen 8568"}},
 :error
[:failed-to-connect
 "Failed to connect to n3/10.0.3.103:8080"]}
```

The unique input for this workflow, "jepsen 8568", appeared in later reads of workflow 559cb7ea-7f07-11f1-9dce-00163eb95b8c. In fact, the client had issued a successful request to create this workflow, but perhaps due to a client-side timeout, opted to retry the request internally. When that retry failed to open a connection, it returned only that most recent error, rather than one which reflected the earlier request timeout. Orkes changed the Java client to

[avoid retrying non-idempotent requests](#), and is exploring whether to add an `isDefinite()` method to client errors (#6).

3.7 Abandoned Workflows (#5)

In versions 5.3.2, 5.4.0, and 5.5.0-rc1, with a Postgres database and a Redis queue, a variety of faults could cause Conductor to permanently abandon some workflow executions. These workflows would remain in the RUNNING state indefinitely, with their tasks never executed. However, both the decider queue and the queue for that task type would be empty. We observed this when killing a single Conductor server, or isolating a server from Postgres or Redis, or killing Postgres. Neither waiting nor restarting the nodes would cause Conductor to re-execute the abandoned workflows.

Consider [this test run](#), where during the 15-second main phase we killed server n4 once, then restarted it. This caused workflow 70ee12a3-7bd8-11f1-8029-00163e983f22 to be abandoned. Even hours after the test, the cluster listed that workflow in state RUNNING, but no worker ever attempted to execute its task.

When a workflow is started, Conductor first writes a workflow record to Postgres, then enqueues the workflow in the decider queue. If there were to be a failure during one of those writes, the workflow might be successfully written to Postgres, but be missing from the queue; it would consequently never be executed. Surprisingly, the [sweeper process](#) which periodically monitors running workflows relies purely on the decider queue, not the database. This meant Conductor never recovered from these partial writes. In version 5.5.0, Orkes added a background process which re-enqueues abandoned executions (#5).

3.8 Lost Workflows (#7)

In versions 5.3.2, 5.4.0, and 5.5.0-rc1, network partitions between a Conductor server and Postgres, or killing Postgres, could infrequently cause Conductor to forget about the existence of workflow executions which were previously visible to clients. Calls to the Java SDK's `getWorkflow()` method would return null, and HTTP requests for `/api/workflow/<id>` would return a 404 error. This state persisted indefinitely; lost workflows were not recoverable.

For example, take [this test of version 5.4.0](#) using Postgres as the DB and decider queue, and Redis as the normal queue. Workflow 0c426286-80e9-11f1-bb95-00163e983f22 was discovered 239 seconds into the test via a call to `getRunningWorkflow()`. Conductor insisted this workflow was in state RUNNING until 313 seconds. From 313 seconds until the end of the test at 677 seconds, every call to `getWorkflow` for that ID returned null.

This workflow execution was committed to Postgres, but the server was unable to append it to the decider queue—perhaps due to a network partition. Given this failure, the server then decided to undo its write

to Postgres by deleting the workflow record. In the intervening 74 seconds, that doomed workflow was visible to clients. In version 5.5.0, Conductor no longer deletes executions when a step fails (#7).

3.9 For Sale: Completed Workflow, Never Run (#9)

When Postgres was killed, we occasionally found workflows in state `COMPLETED` whose tasks had never actually run.³ In this test of version 5.5.0-rc2, three workflows were completed but never executed. At the end of the test, workflow 774977a0-8324-11f1-911d-00163e983f22 looked like this:

```
{
  "tasks" : [{
    "pollCount" : 1,
    "reasonForIncompletion" : "responseTimeout:
      10 exceeded for the taskId:
      77499eb5-8324-11f1-911d-00163e983f22
      with Task Definition: greet",
    "retried" : true,
    "taskDefinition" : {
      "retryDelaySeconds" : 5,
      "rateLimitFrequencyInSeconds" : 1,
      "responseTimeoutSeconds" : 10,
      "retryCount" : 2147483647,
      "timeoutPolicy" : "TIME_OUT_WF",
      ...
    },
    "retryCount" : 0,
    "status" : "TIMED_OUT",
    "executed" : false,
    "workerId" : "n7",
    ...
  }],
  "status" : "COMPLETED",
  ...
}
```

```
]],
"status" : "COMPLETED",
...
}
```

While this workflow's sole task was supposedly assigned to worker n7, neither n7 nor any other worker ever logged it: it seems that `Worker.execute` was never called for this task. The task ended in status `TIMED_OUT`, and its retry policy was `TIME_OUT_WF`, which, per the [Conductor Error Handling documentation](#), should have caused the workflow to also have status `TIMED_OUT`. Instead the workflow was marked as `COMPLETED`, despite never being executed.

This workflow was affected by the task stall problem discussed earlier (#2): worker n7 polled the decider queue and Conductor believed n7 *would* execute the task, but (perhaps due to a network failure) n7 never began execution. The sweeper task on server n4 then marked the task as `TIMED_OUT`, and tried to schedule a retry. That retry process makes two separate writes to Postgres: one which inserts a new task record, and a second which marks the original task row as superseded. Postgres failed between these two writes, but Conductor ignored that failure and marked the original task superseded without a new task row. This explains why the task was never retried, even though the task definition said it should have been. Server n3's sweeper process then discovered the superseded task, and incorrectly concluded that since all tasks had a terminal status, the workflow was `COMPLETED`.

A database transaction around these two writes would have prevented this issue (#9). However, the two writes came from different layers of the code, and wrapping the two in a transaction would have required significant changes. In version 5.5.6, Conductor no longer ignores a failed write of a retry task record.

No	Summary	Event Required	Fixed in
#1	Unexpectedly concurrent task execution	None	5.5.6
#2	Tasks stall on most failures	Pause, or crash, or partition	As designed
#3	Null workflow IDs	Redis crash	Unresolved
#4	<code>startWorkflow</code> returns syntax error	None	5.5.6
#8	<code>startWorkflow</code> returns duplicate key error	Postgres crash and pause	5.5.6
#6	Aborted read	Server crash, or Redis/Postgres crash	Unresolved
#5	Abandoned workflows	Server or Postgres crash, or partition	5.5.0
#7	Missing workflows	Postgres crash or partition	5.5.0
#9	Never-executed workflows marked as completed	Postgres crash and partition	5.5.6

4 Discussion

Our tests of Orkes Conductor found that tasks ran concurrently more often than necessary (#1), and that execution could be significantly delayed by single-node failures (#2). We also found four minor errors in `startWorkflow` (#3, #4, #6, #8), and three serious safety errors where acknowledged workflows

would never execute (#5), disappear entirely (#7), or be marked as `COMPLETED` despite never running (#9). Since we were able to reproduce many of these issues with single process failures, or in some cases observe them in healthy clusters, it is conceivable that production users of Orkes Conductor may have been affected.

Conductor automatically recovered from task stalls

³We discovered this issue just as our work on Conductor was coming to a close; regrettably, we have not been able to test with other versions or narrow down the failure modes that might induce it.

(#2) after the default one-hour timeout; users should not have to take any action to ensure affected workflows complete. Issues #3, #4, and #8 caused `startWorkflow` to return an error or a null workflow ID. It seems likely that client code would notice these errors and retry or propagate them appropriately. Issue #6 caused successful calls to `startWorkflow` to return apparently definite “failed to connect” errors; if clients retried these errors automatically, they may have inadvertently created multiple copies of a single workflow. In the absence of documentation, it may be safest for users to assume that all Conductor errors are indefinite.

For abandoned workflows (#5), users may be able to recover by searching their Conductor clusters for `RUNNING` workflows which have not been executed in some time, and manually restarting them. Likewise, users may be able to search for workflows which are in state `COMPLETED`, but whose tasks did not complete (#9), and restart them as well. Missing workflows (#7) might be recoverable by consulting an external system which is known to have information about workflow executions before they were sent to Conductor—for example, a log maintained by an upstream system.

We were able to test up to version 5.5.0-rc2, which included fixes for issues #5 and #7. Orkes subsequently released version 5.5.6 with fixes for #1, #4, #8, and #9. We recommend users upgrade to version 5.5.6, since it addresses all three serious safety issues we found. Task stalls (#2) were consistent with Conductor’s promised behavior and need not be addressed. Null workflow IDs (#3) and a misleading “failed to connect” error message (#6) are still under investigation.

In other respects, Conductor appeared to function as one would expect. Workflow and task inputs and outputs were consistent with one another, task outputs appeared to be stable despite being executed multiple times, and so on. As always, Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove correctness.

4.1 Concurrency

Users should be aware that within a single workflow, a single task may be executed by multiple worker threads concurrently (#1). In our tests, we observed this behavior every few minutes to hours in healthy clusters, though it affected a relatively small proportion of workflows. This is an inherent property of any distributed execution system, and cannot be prevented—though its frequency can be reduced.

Even if mutual exclusion were possible, it would not prevent the effects of workers from being reordered or interleaved with one another. Since those effects might be routed through an asynchronous network, they could be arbitrarily delayed or lost. This is why distributed systems often use **fencing tokens** or **consensus** instead of mutual exclusion. In any workflow system, users should take care to ensure that tasks

are idempotent and commute—not only with other attempts of the same task, but with other tasks, and with tasks executed by **failure workflows**.

For example, imagine a workflow which books a flight. Conductor begins the book-flight task, but the worker times out, causing Conductor to try the book-flight task again. That too times out, and Conductor initiates a failure workflow to cancel the flight. The failure workflow executes a task which cancels the flight, receives confirmation from the flight service that the flight has been canceled, and terminates. Then one of the requests sent by the book-flight task arrives at the flight service, and the flight is booked again—even though the workflow apparently canceled it.

The solution is to ensure that the book-flight operation and the cancel-flight operation commute—that is to say, they should have the same effects when executed in either order. It is not sufficient for the flight service to cancel an existing flight: it must also prevent any number of *future* requests to book that flight. Downstream services can use **idempotency keys** and logical clocks to provide these commutative semantics.

These challenges are not unique to Conductor, but apply to any distributed orchestration system. The original **Saga paper** was intended for single-node database systems. When **generalized to distributed systems**, new failure modes arise.

4.2 Storage Safety

Conductor is only as safe as the storage systems it relies on. Conductor 5.3.2 required that the systems used for `conductor.db.type` and `conductor.queue.type` not lose acknowledged writes. In 5.4.0, the system used for `conductor.queue.type` could theoretically lose acknowledged writes without impairing Conductor safety, so long as `conductor.queue.decider-queue.type` did not. This was not quite true (#7), but the core idea remains.

We tested both single-node Redis and Redis Cluster deployments, and found that when using Redis for either the primary DB or the queue in 5.3.2, Conductor could lose acknowledged workflow executions when Redis nodes were killed or partitioned from one another. We did not test Postgres data loss, but it seems reasonable to assume similar effects. The question for Orkes and users who deploy Conductor themselves is: how safe are the Redis and Postgres systems being used?

A single Redis instance is of course a single point of failure: it may lose data if the node’s disks fail, or if it is configured with any setting weaker than **`appendfsync always`** and the node loses power. **Redis Cluster** and **Redis Sentinel** can both lose acknowledged writes when networks partition. Managed Redis-compatible services vary in their safety guarantees. **ElastiCache Valkey** and **ElastiCache Redis OSS** both **warn of write loss**, but Amazon says **this should not occur** with `--durability sync`. Google Cloud’s **Memorystore** for Redis may also **lose acknowledged writes**, as can **Azure Managed Redis** with **persistence** and **with high availability**.

Single-node Postgres deployments are similarly vulnerable to catastrophic node failures, like fires or filesystem corruption. Some Postgres replication schemes are unsafe: as [Bin Wang demonstrated](#), Patroni clusters can lose committed writes in some situations, [even with synchronous_mode_strict](#). Citus Cloud claims “you won’t lose any data”, but seems to rely on an asynchronous WAL storage system which could lose up to 60 seconds or 16 MB of writes. Google’s Cloud SQL for PostgreSQL documentation [warns about the possibility of split-brain and data loss](#) in some configurations. However, there are several distributed databases which provide Postgres-compatible interfaces with stronger safety guarantees, like [CockroachDB](#). In short, one’s mileage may vary.

While we only tested the proprietary Orkes Conductor, it seems likely that Conductor OSS depends similarly on the safety of its backing storage services. To avoid lost or abandoned workflow executions, we recommend users carefully evaluate their choice of `conductor.db.type` and `conductor.queue.type`, and the safety of those respective systems.

4.3 Future Work

While these tests were run on Orkes Conductor, they used the standard Conductor Java

client. With a little work, the test suite could measure Conductor OSS behavior as well. We focused on `conductor.db.type=postgres` and `conductor.queue.type=redis_standalone`, but Conductor supports [a variety of storage systems](#). Future research could explore behavior with Cassandra, MySQL, and SQLite.

Our tests only evaluated workflows with a single task using our custom worker. We have not yet explored multi-step workflows, workflows which use Conductor’s built-in or control-flow tasks, or failure workflows. All of these might be useful avenues for further testing.

While we tested with process pauses, crashes, and network partitions, we have not yet explored high network latency or clock skew.

Our thanks to the Orkes team for their invaluable support, including Viren Baraiya, Manan Bhatt, Dale Brady, Jeu George, Dilip Lukose, and Miguel Prieto. Jepsen also wishes to thank [Irene Kannyo](#) for her editorial support. This research was funded by Orkes Inc., and conducted in accordance with the [Jepsen ethics policy](#).