JEPSEN

Capela dda5892

Kyle Kingsbury

2025-08-07

Capela is an unreleased, distributed, general-purpose programming environment. Capela and Jepsen worked together to test several development builds prior to Capela's initial release. We report four issues in the Capela language, including loops that did not iterate; fourteen crashes or non-fatal panics, including double-borrow errors and corrupting allocator memory; severe performance degradation after roughly a minute of operation; and three safety issues, including partitions ignoring their initial values, sporadically vanishing, and losing committed writes. Most issues occurred in healthy clusters. Capela fixed two of the language issues—all others remain under investigation. This research was funded by Capela Inc., and conducted in accordance with the Jepsen ethics policy.

1 Background

Much of modern software involves application logic written in some programming language, which stores state in one or more distributed, fault-tolerant databases. The storage system and application logic usually represent data in different ways, requiring additional languages to query and transform data: SQL, stored procedures, etc. There are also usually translation layers between application logic and storage: client libraries, query builders, Object-Document Mappers, and so on.

Capela is an unreleased, distributed, general-purpose programming environment which aims to simplify application development by unifying processing and storage in a single distributed system. Capela hopes that their integrated model will make programs shorter and easier to understand for both human engineers and Large Language Models.[1]

Capela programs are written in a dialect of Python, augmented with a sophisticated type system.[2] Capela allows programmers to write distributed classes whose instances are automatically replicated and persisted to disk. Method invocation is transactional, with Strong Serializable semantics for pure functions and those which mutate Capela state. Capela also plans to allow functions to perform external I/O, like making calls to HTTP services. Support for external I/O, and its intended consistency model, are still under development.

Readers familiar with distributed languages like SR, MPD, Erlang, Oz, or Bloom will recognize elements of Capela. As with Linda's Tuple Spaces, Capela models distribution not in terms of message passing, but as access to transparently distributed data structures. Like Smalltalk and other image-based languages, Capela persists program state directly, and allows programs to be modified over time. Indeed, Capela feels somewhat like an object-oriented database with stored procedures. More recently, blockchain systems like Ethereum have re-popularized transactional execution over replicated persistent state, viz. smart contracts.

Capela is divided into *partitions*. Each partition is a logically single-threaded state machine whose state is an instance of a user-provided class. Each partition elects a leader to coordinate writes; that partition's throughput is limited by the speed of a single node. A cross-partition transaction protocol allows methods and queries to interact with multiple partitions. Like single-partition operations, cross-partition operations (without external I/O) should ensure Strong Serializability.

Methods on Capela objects can be called via an HTTP JSON API, which provides arguments and returns the method's return value. For example, consider a class which provides a simple map of string keys to integer values:

[1] Large Language Models (LLMs), often referred to as "AI", are statistical models which predict plausible completions of a string of tokens, like an English sentence or a program. When developers generate code with assistance from LLMs, they are often limited by the model's *context window*—the number of tokens the model can work with at once. LLMs are generally less effective on a large codebase, because less of the code fits within the context window. Capela therefore aims to make codebases smaller and more tractable for LLMs.

[2] Capela's type system is still under development, but their plans are ambitious. Capela intends to statically analyze not only arguments and return values, but also to constrain values via dependent types, and to control side effects via session types and an effect system. Static analysis of dataflow and effects allows sophisticated optimizations. For example, Capela may be able to prove when ordering constraints can be elided, because two pieces of code do not interact with the same state. When one variable in a list changes due to a concurrent transaction's update, Capela may be able to re-evaluate only the part which changed, and so on.

```
class KV(Node):
    state: Dict[str, int] =
      Field(default_factory=dict)

    def set(self, key: str, value: int) -> int:
        self.state[key] = value
        return value

    def get(self, key: str) -> Optional[int]:
        return self.state.get(key, None)
```

Users can call `set` and `get` by making HTTP POST requests, specifying the ID of an instance of this class. Each instance's `state` map is automatically persisted and replicated across nodes. To record in partition `!1234...` that country singer Kacey Musgraves has won seven CMA awards, one might call:

```
POST /d/!1234.../set {"key": "Kacey Musgraves",
                      "value": 7}
=> 7
```

Capela engaged Jepsen early in the development process to verify the safety and fault-tolerance properties of their system. During our engagement, Capela had not yet been released and had no public marketing or documentation. Its website launched on 2025-08-06, just prior to the release of this report. Instead, we based our understanding of Capela's intended guarantees on informal documentation and conversations with Capela's team. We stress that the issues we discuss in this report are typical of early, unreleased software, and had (as of this writing) no user-facing impact.

2 Test Design

We designed a test suite for Capela using the Jepsen testing library. From 2025-04-07 through 2025-05-07, we tested versions `dda5892` (2025-04-08) through `599e9cb` (2025-04-28), running on clusters of three to five Debian nodes. We ran our clusters both in LXC and on multiple EC2 VMs. Our Python programs were uploaded to the local directory of each node prior to startup.

Our test harness injected a variety of faults, including killing and pausing processes, partial and total network partitions, setting node clocks forward and back by milliseconds to hundreds of seconds, and strobing clocks rapidly between two times. We took snapshots of and later restored chunks of Capela's data files, and also introduced random single-bit errors into data files. In some tests we ran Capela on top of LazyFS, a FUSE filesystem which can forget un-fsynced data on demand; we coupled that data loss with process crashes to simulate power failures.

Our tests ran several different workloads, which follow.

2.1 Write-Read Registers

As a simple test of Capela's transaction semantics, we wrote a simple key-value store backed by a single dictionary. In addition to direct writes and reads on individual keys, we implemented a small transaction system which took a list of `[function, key, value]` micro-operations and applied them sequentially, returning a completed transaction. For example, here is a transaction which intends to read key `1`, set key `1` to `2`, then read key `1` again:

```
[[:r 1 nil] [:w 1 2] [:r 1 nil]
```

Our `txn` method took that transaction structure, applied it to state in Capela, and returned that same transaction structure, filling in the values read. Assuming the initial value of key `1` was `0`:

```
[[:r 1 0] [:w 1 2] [:r 1 2]]
```

We used the Elle consistency checker to analyze histories of these transactions. For this workload Elle inferred partial version orders for keys by assuming writes follow reads within a single transaction. Given Capela's goal of Strong Serializability, we also assumed individual keys were Linearizable in reconstructing version orders. Elle used those version orders to build a transaction dependency graph, and searched for cycles which would violate Strong Serializability.

Elle also checked for a number of non-cyclic anomalies. For instance, we searched for internal anomalies within a single transaction, like a read which failed to observe the most recent write. We also checked for G1a (Aborted Read), G1b (Intermediate Read), and P4 (Lost Update).

We designed two variants of this workload. The simple variant, *wr*, stored all data in a single partition. A second, *multi-wr*, created several partitions, each with their own dictionary, and hashed each key to a specific partition. We submitted transactions to a single coordinator partition, and indicated which partition each particular key belonged to. This allowed us to test Capela's cross-partition transaction protocol.

2.2 List Append

We designed a closely related transactional workload, list-append. Instead of overwriting values in place, our writes appended a unique integer element to a list identified by some key. From the order of those elements, Elle inferred the order of transactions which produced that version. This allowed us to infer almost all of the write-write, write-read, and read-write dependencies which must have been present in a history, detecting additional anomalies.

As with the write-read register workloads, we wrote both *append* and *multi-append* variants, which stored their keys in a single partition, or in multiple partitions, respectively.

2.3 Partition Set

Transactions in the write-read register and list-append tests often failed because the partitions they interacted with did not exist. This was particularly surprising as Capela intended to ensure that once created, a call to `select(partition-key)` would always return that partition, rather than `None`.

Our *partition-set* workload verified this claim by performing two types of operations. First, it could create a single partition in Capela, storing a unique integer value, and remember the key assigned to that partition. Second, the test would try to read all partitions by issuing a `select(partition-key)` query for each known partition. Once a partition was acknowledged as created, it should appear in every read beginning later.

2.4 Ad Hoc Queries

As we encountered bugs in the Capela language, we built up a collection of handcrafted programs to explore behavior and serve as a regression test. We submitted each of these programs to Capela's `/query` endpoint, which evaluates an arbitrary string of code, and compared its return value against an expected result.

2.5 Generative Python

We also devised a small, experimental workload called gen-py, which generates simple Python programs and compares how each program runs in Jython (a Python interpreter), to how it runs in Capela. We used test.check, a property-based testing library, to generate random programs and to shrink faulty programs to smaller examples.

We spent only one day on this workload, so it could generate only simple programs: literals, variable assignments, function definitions, and function calls. Nevertheless, it uncovered several unusual behaviors in Capela.

3 Results

We begin with four issues in Capela's Python-compatible language. We then discuss fourteen crashes or non-fatal panics, and a problem with performance degradation after roughly a minute of operation. We finish with three safety errors.

Jepsen filed issues in an empty Github repository Capela set up, with the understanding that it would be made public before the release of this report. However, as of our scheduled publication time, the issue tracker remained private. All issue links will return 404 errors initially, but we hope this will be resolved shortly.

3.1 `for` Loops Don't (#1)

In version `ebf0f3e`, `for` loops in Capela silently failed to evaluate their bodies. For example, consider the following program which appends elements from one list to another:

```
acc = [0]
xs = [1,2,3]
for x in xs:
    acc.append(x)
acc
```

When run in Python, this returns `[0, 1, 2, 3]`. In Capela, it returned `[0]`: the loop was never evaluated. Capela reports that they resolved this issue (#1) in version 25c4b96.

3.2 `match` is Unimplemented (#7)

Version `ebf0f3e` did not support Python's `match` expressions, which are analogous to what many languages call `switch` or `case`. For example, here is a test run which performed a simple match query.

```
x = 2
match x:
    case 1:
        '1'
    case 2:
        '2'
    case _:
        'other'
```

In Python, this would normally return `'2'`. In Capela, this threw `expected an indented block`. Since the error message does not mention that `match` is unimplemented, and does not provide a line number or other pointer to the expression, programmers could have trouble figuring out what had gone wrong. We recommended that Capela provide a more informative error message until `match` was ready (#7).

3.3 Destructuring Bind Leaks Internal Structures (#2)

In version `ebf0f3e` and below, destructuring a list leaked internal Capela structures into runtime variables. For example, consider this program, which destructures the list `[1, 2]`, and should result in `y = 2`:

```
x, y = [1, 2]
y
```

In Capela, this program returned `{:object_type "builtins.Effect"}`, rather than 2. We found a number of internal representations leaked into return values, including `builtins.Effect` and `builtins.Projection`. The cause was simple: while destructuring bind created an effect, it was never applied to the right-hand side of the statement. Capela fixed this issue (#2) in version `599e9cb`.

3.4 Semi-Lazy Assignment Order (#6)

Unlike Python, Capela's variable assignment semantics were somewhat, but not entirely, lazy. This allowed programs which would crash in Python to silently succeed in Capela, and vice-versa. In version `ebf0f3e`, we observed several interesting behaviors. For example, take the program…

```
def f():
    x
'ok'
```

This compiles and returns `'ok'` in Python. Python allows variables to be defined after the functions which use them; `f()` throws at call time, not compile time. In Capela, this program threw `name not found: x`; the compiler required that variables referred to in a function be defined beforehand. On the other hand, Capela allowed the use of unbound variables *outside* a function:

```
x = y
y = x
x
```

In Python, this throws `name 'y' is not defined`. In Capela, it returned the JSON object `{:object_type "builtins.Thunk"}`—a representation of an internal structure representing lazy evaluation. This laziness allowed programs to read values "from the lexical future":

```
y = x
x = 2
y
```

This returned `2`, rather than Python's `name 'x' is not defined`. On the other hand, sometimes assignment in Capela did occur in lexical order:

```
y = x
x = 1
z = x
x = 2
[x, y, z]
```

This returned `[2, 2, 1]`: `x` observed `y`'s later value, even though it was assigned before `y` existed. However, `z` observed `y`'s earlier value, even though it was assigned after `x`!

One might assume that Python programmers would simply not write programs like this. However, programmers do make mistakes and use variables out of order by accident. In Python, the language's (mostly) eager assignment order catches these kinds of errors early. In Capela, several of our test programs inadvertently fell afoul of lazy evaluation, silently returning incorrect results. Debugging these issues took considerable effort.

Capela states their compiler "should codegen IR that preserves the expected Python behavior." We opened #6 for these divergences from Python.

3.5 Unset Env Panics (#16)

When attempting to create a new partition with initial values, or when executing transactions, version `1cde55b` would often return an HTTP 500 error complaining a task had panicked:

```
POST /partitions {"object_type": "foo.Foo",
                  "id": 0}
=> HTTP 500 'task 824 panicked with message
   "Env is not set for the current thread"'
```

These panics appeared constantly in node logs, even in healthy clusters. In some cases the node continued running; in others, it crashed. The most common source was `ext_serde.rs:57`. Capela is investigating this issue (#16).

3.6 Replication Lag Crash (#4)

In our initial tests of version `ebf0f3e`, we found that Capela reliably crashed after roughly a minute of operation. After a few hundred transactions a node would often panic, complaining that it `Failed to receive block from BlockExchange: Lagged`. For instance, take this short test, on which node `n1` crashed after about ten seconds:

```
thread 'netdev' panicked at
uvm_syn/src/net/libp2p/driver/blocks.rs:199:25:
Failed to receive block from BlockExchange:
Lagged(266)
```

The node then entered a shutdown process, and went through a cascade of additional errors: unwrapping `Result`s which contained `Err` values, attempting to access a `txn` attribute on a `NoneType` object, and finally panicking in a destructor.

This issue (#4) appeared with as few as five transactions per second. Capela is investigating.

3.7 Fast Sync Timeout Crash (#22)

In version `599e9cb`, freshly started clusters without faults could have nodes spontaneously crash due to a timeout while syncing the history of a partition. In this test run, one node logged the following, then exited:

```
netdev ThreadId(18)
uvm_syn::net::stsync::behaviour: Query timed
out: FastSyncPartitionHistory { partition:
!7ee0...07d, from_block: Some(BlockReference
{ block_no: 1, block_hash: "897...f22" }),
to_block: None, limit: Some(32) }
query=QueryId(3v39)
```

The test harness automatically restarted the node, whereupon it crashed immediately. Before crashing, it logged that `uvm_syn::mvcc::partition::run_loop` had encountered an error: `Block out of order: 3 != 2`. Capela is investigating this issue (#22).

3.8 Double-Borrow Error in `vm_task.rs` (#8)

In version `ebf0f3e`, we observed nodes in healthy clusters spontaneously crash after a few seconds of transactions with a `BorrowMutError` panic in `vm_task.rs`.

```
thread 'tokio-runtime-worker' panicked at
uvm_syn/src/mini/env/vm_task.rs:165:40:
already borrowed: BorrowMutError
```

Capela is investigating this issue (#8).

3.9 Double-Borrow Error in `tx.rs` (#17)

In version `599e9cb`, nodes in healthy clusters occasionally panicked with an apparently non-fatal borrow error in `tx.rs`. For example, take this test run, in which node `n1` logged:

```
thread 'tokio-runtime-worker' panicked at
uvm_syn/src/mini/builtins/tx.rs:264:69:
already mutably borrowed: BorrowError
```

Capela is investigating this issue (#17).

3.10 Double-Borrow Error in `reloc.rs` (#18)

We also observed `already borrowed` panics in `reloc.rs`. These errors occurred in healthy clusters, but they did not cause the node as a whole to crash. For example:

```
thread 'tokio-runtime-worker' panicked at
uvm_syn/src/mini/env/reloc.rs:266:48:
already borrowed: BorrowMutError
```

Capela is investigating this issue (#18).

3.11 Unimplemented Panic in `election.rs` (#11)

In healthy clusters under light load, Capela `ebf0f3e` would occasionally log `not yet implemented` panic messages from the election module. For instance, in this run, node `n2` logged…

```
thread 'tokio-runtime-worker' panicked at
uvm_syn/src/mvcc/partition/run_loop/
election.rs:1507:17: not yet implemented
```

Capela is investigating this issue (#11).

3.12 Double-Free or Corruption Crash (#12)

In version `ebf0f3e`, we found nodes in freshly-created, healthy clusters occasionally crashed with a `double-free or corruption (out)` message, while creating partitions.

```
thread 'tokio-runtime-worker' panicked
at uvm_syn/src/mini/ext_serde.rs:57:9:
Env is not set for the current thread
double free or corruption (out)
exception_to_error: task 835 panicked
with message "Env is not set for the
current thread"
```

Capela is investigating this issue (#12).

3.13 Corruption of Memory Allocator State (#21)

In version `599e9cb`, fresh clusters would frequently crash during setup, logging `malloc(): unaligned tcache chunk detected`, `corrupted size vs prev_size`, `corrupted double-linked list`, or `malloc(): invalid size (unsorted)`.

These errors (#21) suggest that Capela often corrupted the internal state of the memory allocator, possibly via an out of bounds write, buffer overrun, or other memory error. Valgrind suggests some memory leaks, but we were unable to determine the cause. Capela is investigating.

3.14 B-Tree Height Assertion Crash (#15)

In version `ebf0f3e`, nodes in a healthy cluster, without faults, could crash following an assertion failure involving a B-tree height invariant. In this test run, node `n1` crashed with:

```
thread 'tokio-runtime-worker' panicked at
/rustc/854f22563c8daf92709fae18ee6aed52953835cd
/library/alloc/src/collections/btree/node.rs:
685:9: assertion failed: edge.height ==
self.height - 1
```

Capela is investigating this issue (#15).

3.15 Index Out of Bounds in `reloc.rs` (#23)

In `599e9cb`, nodes in a freshly started cluster could crash during partition creation, without any fault injection, due to an index out of bounds error in `reloc.rs`. For example, this run had a node crash with:

```
thread 'tokio-runtime-worker' panicked at
uvm_syn/src/mini/env/reloc.rs:546:28:
index out of bounds: the len is 17 but the
index is 19
```

Capela is investigating this issue (#23).

3.16 Unreachable Code Panic in `latches-0.2.0` (#13)

In version `ebf0f3e`, following an election, Capela could log a non-fatal panic like:

```
thread 'tokio-runtime-worker' panicked
at /root/.cargo/registry/src/
index.crates.io-1949cf8c6b5b557f/
latches-0.2.0/src/task/waiters/mod.rs:51:22:
internal error: entered unreachable code:
update non-existent waker
```

We were able to reproduce this issue (#13) with either process pauses or kills. Capela is investigating.

3.17 Crashes on Startup With `.sst` File Bitflips (#10)

Capela uses sorted strings tables (`.sst` files) to store data on disk. In version `ebf0f3e`, introducing a handful of single-bit errors into one of these `.sst` files caused the node to crash immediately on startup, logging `Corruption: block checksum mismatch`. For example, take this run, where corruption in `n1`'s `storage/00031.sst` caused `n1` to crash with:

```
thread 'main' panicked at
uvm_syn/src/runtime/host.rs:366:73:
called `Result::unwrap()` on an `Err` value:
RocksDB(Error { message: "Corruption: block
checksum mismatch: stored(context removed) =
2870321000, computed = 3921368871, type = 4
in /opt/capela/data/storage/000031.sst
offset 873 size 937 The file
/opt/capela/data/storage/MANIFEST-000034
may be corrupted." })
```

Note that this log line mis-identifies the corrupted file—this test only corrupted the `.sst` files in `storage/`, not the `MANIFEST` files. Corruption in different directories caused slightly different error messages.

Many databases crash when their on-disk files are corrupted. Only a few, like Riak and TigerBeetle, are designed to tolerate and repair disk errors. Capela's engineers intend for Capela to handle disk errors gracefully, so we reported this as #10.

3.18 Crash in `libp2p-rendezvous` With File Snapshot/Restore (#14)

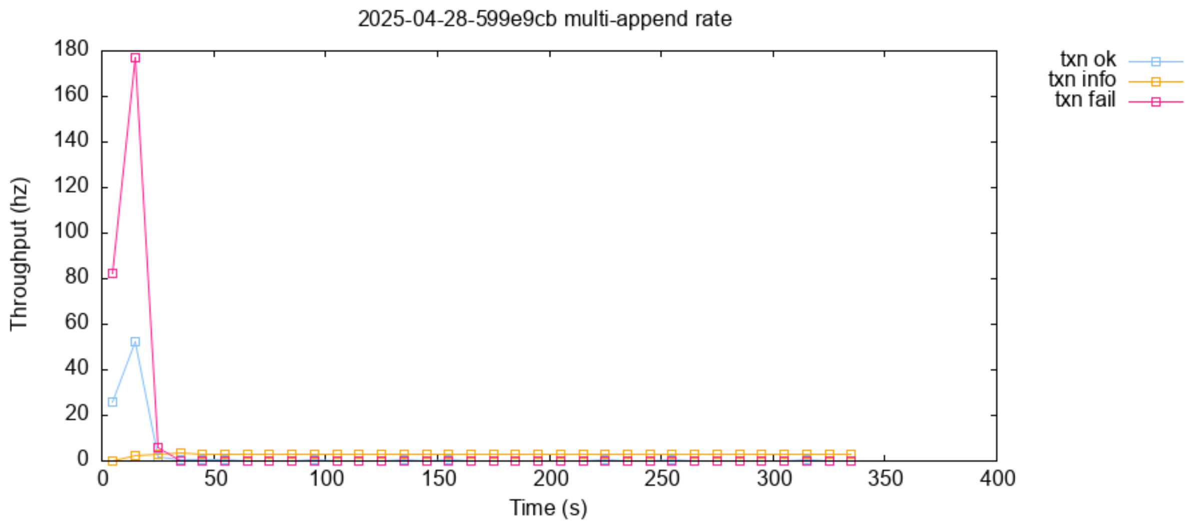
In version `ebf0f3e` we encountered a rare crash caused by a panic in `libp2p-rendezvous`. Our only example of this crash involved process kills, combined with restoring snapshots of `.sst` and `.blob` files.

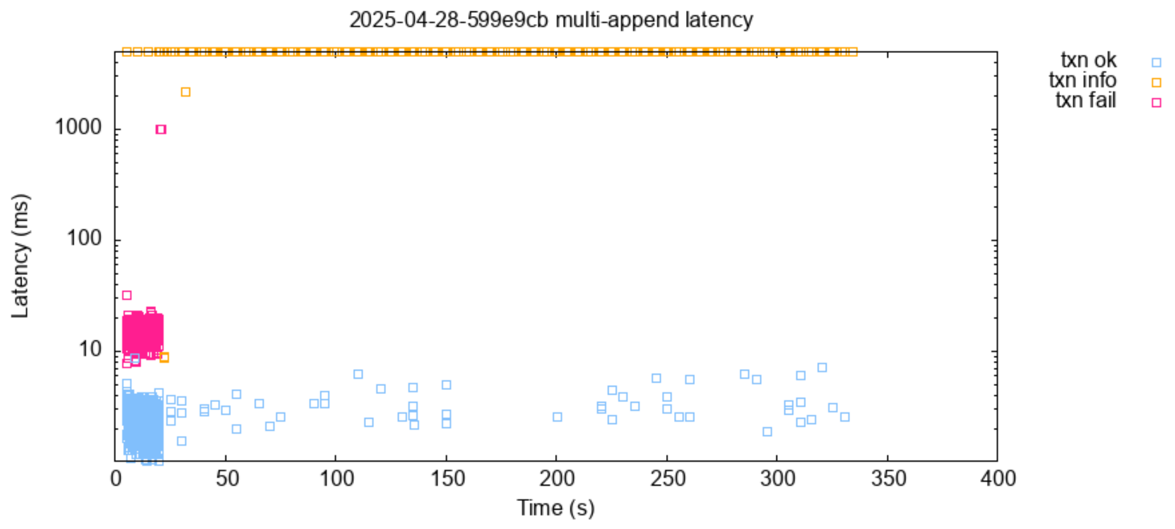
```
thread 'netdev' panicked at
/root/.cargo/registry/src/index.crates.io-
1949cf8c6b5b557f/libp2p-rendezvous-0.16.0/
src/server.rs:194:38:
Send response: DiscoverResponse(Ok(([], Cookie
{ id: 5474065259777396154, namespace:
Some(Namespace("uvm-sync")) })))
```

Capela is investigating this issue (#14).

3.19 Performance Degradation (#19)

After roughly 20 to 100 seconds of operation, healthy clusters typically flipped into a degraded state. Goodput dropped from roughly 50 to about 0.2 transactions per second. Latencies on most nodes jumped from roughly five milliseconds to over five seconds (our client socket timeout). Here are typical plots of latency and throughput over time, for a five-node cluster running version `599e9cb`.





In this test, the performance degradation was contemporaneous with node `n4` crashing due to a `BlockExchange: Lagged` panic. The test suite detected the crash and restarted `n4`, but the cluster never recovered. From that point on, only requests against node `n1` succeeded. All others timed out.

Capela is investigating this issue (#19).

3.20 New Partitions Ignore Provided Fields (#3)

When creating a partition, one provides a JSON map with the `object_type` to be instantiated, and any fields one wishes to set on the resulting object. For example, consider a dog with a `name` field:

```
class Dog(Node):
    name: str
```

To create a particularly noble dog, one might make an HTTP request like…

```
POST /partitions
{"object_type": "dog.Dog",
 "name": "Baron Frederick von Puppington III"}
```

In version `1cde55b` and below, this created a `Dog` with no name: the `get_or_create_partition` function ignored any fields provided and hardcoded an initial payload of `None`. Moreover, since partitions are uniquely identified by their state, making multiple POST requests to create different dogs would, in fact, return the same empty dog each time. This issue (#3) was fixed in version `ebf0f3e` (2024-04-10), but returned in `599e9cb` (2024-04-28). Capela is investigating.

3.21 Partitions Vanish And Reappear (#24)

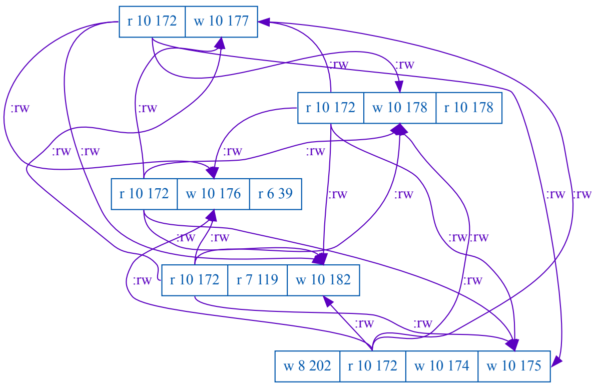
After their creation has been acknowledged, partitions in Capela should always appear to readers. However, in version `ebf0f3e`, we found that partitions sporadically disappeared after being created. This occurred even in healthy clusters under minimal load. We observed this problem frequently in our list-append tests—where it caused transactions to abort—and designed the partition-set workload to confirm.

For example, take this test run, which attempted to create eight partitions, each storing a single unique integer. Seven of those attempts (1, 2, 4, 5, 6, 7, and 8) succeeded. However, only partition 5 was reliably visible to all readers at the end of the test. For the other six partitions, calls to `select(partition-key)` returned `None` on some nodes.

Capela is investigating this issue (#24).

3.22 Lost Update (#5)

In version `ebf0f3e`, transactions involving reads and writes to our single-partition key-value store appeared to exhibit P4 (Lost Update). For instance, here is a 20-second test run which performed 1,633 successful transactions against a healthy three-node cluster, without faults. It found 50 instances of Lost Update: two or more committed transactions which read the same version of some key, then wrote that key.



Here are five of those transactions, all of which read key 10's value as 176, wrote some other value for key 10, and committed. None of them observed the others' effects, which is why there are read-write (`:rw`) dependencies between them. Under Update Atomic, at most one of these transactions may commit. This history therefore violates Update Atomic, Parallel Snapshot Isolation, Snapshot Isolation, Serializability, and so on.

Transactions also observed divergent timelines of writes to individual keys. For example, here are the first few non-empty reads of key `12` from a short list-append test:

Process	Observed Value
11	1
6	1, 2
11	1, 2, 3
1	1
1	1
11	1, 4, 6
1	1, 4, 7, 8
1	1, 4, 7, 8
6	1, 4, 6
11	1, 4, 7, 8, 9
1	1, 4, 7, 8, 9, 10
6	1, 4, 7, 8, 9, 10

After this time, the elements `[1, 4, 7, 8, 9, 10]` were stable, though later elements continued to fluctuate. Since transactions only ever append elements, every read of key `12` should have been consistent—namely, a prefix of the longest version of the list. Instead, transactions observed elements which appeared for some time, then vanished permanently. These lost elements could also be seen by multiple transactions. The first read of `[1, 2]` above was performed by the transaction which appended `2`:

```
[[:append 12 2]
 [:r 12 [1 2]]
 [:append 12 3]]
```

However, this append of `2` was also observed by a concurrent, read-only transaction, before it vanished permanently.

```
[[:r 6 [1 2 3 ...]]
 [:r 6 [1 2 3 ...]]
 [:r 7 [4 6 7 ...]]
 [:r 12 [1 2 3]]]
```

Prior to version `599e9cb`, transactions never aborted. In `599e9cb`, transactions often returned HTTP 500 errors with the message `Transaction aborted`, and we no longer observed Lost Update or incompatible orders in list-append tests. However, we continued to see them with write-read registers. There may have been two separate bugs, or perhaps the change to transaction error handling only affected the list-append workload.

Capela is unsure what caused this bug (#5), or why their changes might have affected it. Investigation is ongoing.

№	Summary	Event Required	Fixed in
#1	`for` loops don't loop	None	25c4b96
#7	`match` is unimplemented	None	Unresolved
#2	Destructuring bind leaks internal structures	None	599e9cb
#6	Semi-lazy assignment order	None	Unresolved
#16	Unset `Env` panic	None	Unresolved
#4	Replication lag crash	None	Unresolved
#22	Fast sync timeout crash	None	Unresolved
#8	Double-borrow error in `vm_task.rs`	None	Unresolved
#17	Double-borrow error in `tx.rs`	None	Unresolved
#18	Double-borrow error in `reloc.rs`	None	Unresolved
#11	Unimplemented panic in `election.rs`	None	Unresolved
#12	Double-free or corruption crash	None	Unresolved
#21	Corruption of memory allocator state	None	Unresolved
#15	B-tree height assertion crash	None	Unresolved
#23	Index out of bounds in `reloc.rs`	None	Unresolved
#13	Unreachable code panic in `latches-0.2.0`	Pause	Unresolved
#10	Crash on startup due to `.sst` file corruption	File bitflips	Unresolved
#14	Crash in `libp2p-rendezvous`	File snapshot/restore	Unresolved
#19	Performance degradation	None	Unresolved
#3	New partitions ignore provided fields	None	Unresolved
#24	Partitions vanish and reappear	None	Unresolved
#5	Lost Update	None	Unresolved

4 Discussion

Capela engaged Jepsen early in the development process to help validate safety and fault tolerance. During our collaboration in spring 2025, Capela was still an unreleased prototype with no documentation. Core features such as iteration and side effects had not yet been implemented. Builds routinely crashed after a few seconds, and throughput generally fell to just a few transactions per second after a minute or so. We observed frequent data loss. We stress that these are normal behaviors for a system in early development—Capela intends to resolve safety issues and crashes prior to their first public release. Capela also aims to support most, if not all, of Python's semantics before the release.

While language semantics were not the focus of our investigation, we found four issues in Capela's Python compiler: unsupported syntax (#7), semi-lazy variable assignment (#6), `for` loops which quietly did nothing (#1), and internal language structures leaking out of destructuring bind (#2). Issues #1 and #2 were resolved in `25c4b96` and `599e9cb` respectively; the others have yet to be addressed.

We found fourteen crashes or non-fatal panics in Capela. Nodes panicked due to missing `Env` structures (#16), slow replication (#4, #22), double-borrow errors (#8, #17, #18), unimplemented code (#11), double-free or other memory corruption errors (#12, #21), B-tree height invariant errors (#15), an index out of bounds (#23), unreachable code (#13), and file corruption (#10, #14). Many took down the node entirely. All but the last three occurred in healthy clusters. We also found severe performance degradation in almost every test run. After roughly a minute, transaction throughput would drop by two orders of magnitude, and RPC requests to most nodes would simply time out (#19). Capela is investigating these issues.

Finally, we found three safety issues in Capela. Newly created partitions silently ignored the initial values provided for their fields (#3). Partitions would randomly vanish and reappear after creation (#24). Transactions also frequently lost writes. We observed both P4 and incompatible versions of keys in list-append tests, which suggests versions temporarily diverged before being lost (#5). All three issues remain extant.

Since Capela had no external users during our collaboration, these bugs had no real impact on users. Capela will continue to invest in testing and resolving bugs prior to their initial release.

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. In particular, our tests were limited by Capela's frequent crashes and performance degradation. Capela could have additional correctness bugs which did not appear in our tests because we could not perform enough transactions to encounter them.

4.1 Future Work

We found a number of additional crashes or panics which, for want of time, we omit from this report. Several places in the Capela code attempt to call `Result::unwrap()` on an `Err` value—for example, issue #9. It also seems likely that the Capela language has more features which are unimplemented or incorrect, such as `with` expressions. Capela plans to expand their language test coverage prior to release. Additional generative testing of programs may prove fruitful.

We have several plans for additional Capela tests. We would like to create partitions dynamically during the `list-append` workload, rather than up-front. We would like to store data in multiple instance variables per object. Finally, we believe it would be useful to test other data structures—both those provided by Capela and ones we implement ourselves. Instead of using the built-in list type, we could summon (so to speak) a list from the void.

This work would not have been possible without the assistance of the Capela team, including Gabriel Guimaraes, Jonathan Lima, and Derek Stavis. Our thanks also to Coda Hale and Ben Linsay for their help in investigating Rust errors. As always, we are grateful to Irene Kannyo for her editorial support. This research was funded by Capela Inc., and conducted in accordance with the Jepsen ethics policy.