

Transparent Continuation Checkpointing

Recovering long-running programs from committed execution state

Omar Abdelrahman · Trigora · September 2026

CONTENTS

Abstract

1. Durable execution and history reconstruction
2. Motivation: long-running dynamic software
3. Transparent Continuation Checkpointing
4. From source program to resumable execution
5. Continuation representation and commit semantics
6. Portable TCC engine architecture
7. Host protocol and durable recovery
8. Programming and execution model
9. External effects and nondeterminism
10. Waiting, timers, children, cancellation, and control flow
11. Recovery semantics
12. Code and artifact evolution
13. Evaluation methodology
14. Recovery versus execution history
15. Recovery versus live state
16. Healthy-path cost
17. Semantic and crash-recovery validation
18. What TCC changes
19. Limitations

20. Related work and alternative recovery architectures

21. Conclusion

Acknowledgments

References

Abstract

Replay-based reconstruction can make recovery depend on accumulated execution history. Transparent Continuation Checkpointing (TCC) is a durable execution architecture that captures resumable program state at durable boundaries, allowing supported executions to recover without reconstructing their current position through accumulated prefix replay.

Language frontends compile declared TypeScript and Python subsets into a common versioned TCC program artifact. A host-independent execution core runs that artifact, commits a logical continuation at durable boundaries, and restores it into a fresh runtime. The continuation is not a dump of process, heap, or WebAssembly memory. Recovery loads the artifact named by the continuation and hydrates saved execution state. Completed prefix work is not re-executed.

In controlled experiments on a frozen research prototype, at fixed live state, TCC recovery remained approximately 0.6–0.9 ms as execution-history depth increased from 10 to 1,000, while the evaluated history-replay reconstruction baseline increased from approximately 61 ms to 1.7 s. No semantic failures were observed across 50,000 generated cases within that prototype’s evaluated grammar. A portable Rust engine, embeddable natively and through WebAssembly, executes artifacts produced by the TypeScript and Python frontends against the same continuation and host-protocol semantics. Fresh-process crash tests validate recovery against a durable reference host. Those crash tests are architectural and semantic validation. They are not new latency measurements.

These recovery timings are local-harness measurements on a synthetic depth workload. They characterize a scaling law. They are not production-server performance

guarantees, and they do not claim constant-time recovery, exactly-once external I/O, or arbitrary-language support.

1. Durable execution and history reconstruction

Durable execution is the problem of running a program that may stop, wait, crash, or outlive the process that began it, and later continue as if the interruption were part of ordinary control flow. The program might call an external API, wait hours for a human decision, spawn child work, or resume after the machine that held it is gone. Between those events the operator still needs a coherent answer to a simple question: *where is this program?*

A widely deployed answer is **history replay**. The runtime records an append-only log of workflow events. Recovery loads that log and re-executes the program, substituting recorded results for completed work, until control flow arrives again at the wait that was in progress.

Execution:

A → B → C → D → WAIT
 x

Recovery:

A → B → C → D → WAIT
—————→
reconstruct

Replay is attractive for good reasons. It is close to event sourcing: the log is an authoritative transcript, and the present is a function of that transcript. Deterministic workflow code plus recorded activity results yields reproducibility. The semantics are mature. Temporal [1] is the most fully realized industrial instance of this model, and the measurements in this paper treat it as such — as evidence that replay is an extremely successful architecture, not as a claim that replay has failed.

events is a place the process might die, and a place a replay-based runtime would typically record another slice of history.

TCC is not an agent algorithm. Agents are the wedge: they turn durable execution from an optional workflow feature into part of the execution substrate. The same question applies to any long-running dynamic program — approval loops, multi-step integrations, programs that suspend on timers and events.

The research question is narrower than “how should agents persist?”

Can durable execution preserve *where a program currently is* rather than reconstructing that position from everything that happened before?

That is the question this paper evaluates.

3. Transparent Continuation Checkpointing

A **continuation**, in the sense used here, is not a transcript and not a debugger snapshot of the entire heap. It is sufficient resumable execution state to continue a supported program from a durable boundary.

```
supported source program
  ↓
language frontend
  ↓
versioned TCC program artifact
  ↓
explicit execution state
  ↓
durable operation
  ↓
logical continuation commit
  ↓
process may disappear
  ↓
fresh runtime
  ↓
load exact artifact
  ↓
hydrate continuation
  ↓
resume saved execution state
  ↓
continue without replaying completed prefix
```

A durable boundary is a commit point. After it commits, it is legal for the process to die. Recovery continues from that boundary instead of replaying the prefix that led there. Before it commits, recovery returns to the previous committed boundary.

The continuation has to carry, at a high level:

control position — enough to resume at the boundary, not at the start of the program;

required live program state — the values still needed to continue, not every value the program has ever held;

execution and version identity — which program, and which deployed artifact, this continuation belongs to;

references required for durable operations — outstanding effects, waits, and child executions that recovery must re-arm rather than recreate.

That is the logical object recovery uses. Sections 4–7 show how a frontend produces it, what a committed continuation contains, and how a fresh process restores it. Serialization layout and storage keys are omitted.

TCC is **transparent** relative to a declared language subset: the developer writes ordinary supported source code together with explicit durable operations, while the language frontend and runtime own the resumable execution representation underneath. The current frontends target declared subsets of TypeScript and Python. It is not transparent with respect to arbitrary host-language programs. Unsupported constructs are compile-time errors. Transparency is a property of the supported surface, not a claim that every program in the source language is a durable execution.

Languages are how developers express programs. A versioned program artifact is how the engine understands them. TCC is how that artifact is executed durably.

The continuation is **not**:

- the V8 heap;
- a JavaScript closure serialization;
- the Rust stack;
- an OS or process image;
- a dump of WebAssembly linear memory;
- accumulated workflow history.

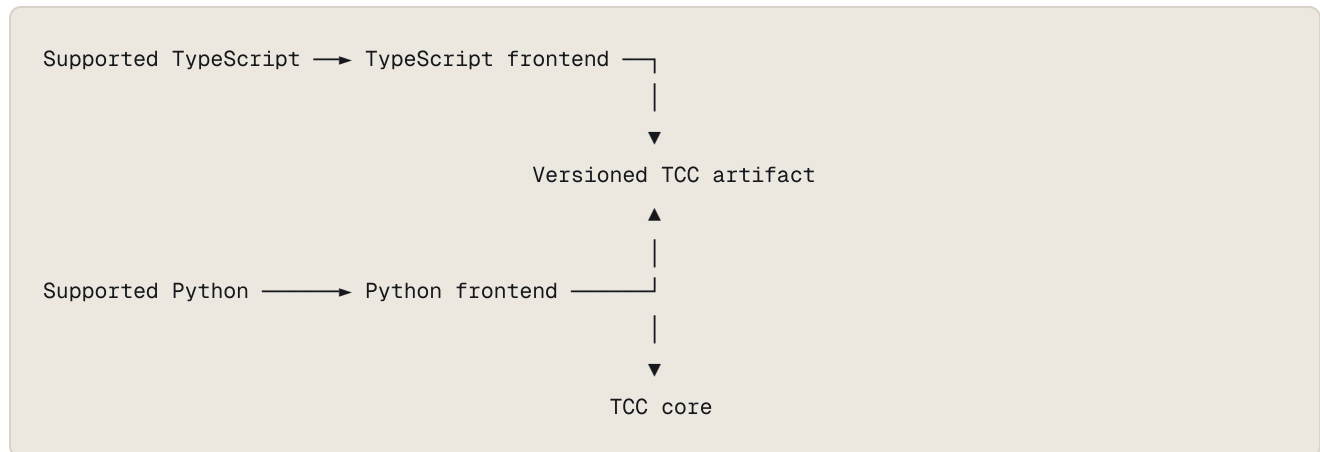
History may still be retained for audit and observability. It is not the recovery primitive.

4. From source program to resumable execution

TCC separates source-language semantics from durable execution semantics. Each language frontend accepts a declared subset of its source language and lowers it into the same versioned TCC program artifact consumed by the execution core. The artifact is an explicit execution representation: functions, an entry point, and a resumable instruction sequence the core can step. Source text is not an input to the engine at run

time. Both frontends target the same **TCC artifact model and engine instruction semantics**. Equivalent TypeScript and Python programs need not compile to byte-for-byte identical artifacts.

Figure A. Source languages to a common execution representation.



A minimal TypeScript program in the current authoring model:

```
import { effect, waitForEvent } from "@trigora/sdk";

export default async function run() {
  const result = await effect("generate", async () => generateSomething());
  const approval = await waitForEvent("approved");

  return { result, approval };
}
```

The corresponding Python program:

```
from trigora import effect, wait_for_event

async def run():
    result = await effect("generate", generate_something)
    approval = await wait_for_event("approved")
    return {"result": result, "approval": approval}
```

Durable operations are those imported from the frontend's declared authoring surface. TypeScript aliases are recognized:

```
import { effect as durableEffect } from "@trigora/sdk";
```

A locally declared `function effect`, or an unsupported construct, does not become a TCC durable operation. Effect callback bodies are not compiled into the artifact; the host performs the external work for a string-literal key.

Each frontend:

- parses the supported program;
- resolves durable-operation imports for that language;
- identifies durable operations by resolved SDK identity, not merely by function name;
- lowers supported control flow into an explicit TCC execution representation;
- assigns stable durable-operation identities;
- determines which values and control state must survive durable boundaries;
- emits a versioned TCC program artifact (frontend identity, language-semantics version, engine format version, required engine features, required host capabilities, and the program).

Both programs lower conceptually to the same execution:

```
State 0:
  request effect("generate")
  suspend

State 1:
  result = effect result
  request wait("approved")
  suspend

State 2:
  approval = event result
  return { result, approval }
```

Conceptual representation; implementation serialization details omitted.

```
Conceptual artifact

entry run

state 0:
  EFFECT key="generate"
  -> state 1

state 1:
  STORE result
  WAIT_EVENT name="approved"
  -> state 2

state 2:
  STORE approval
  RETURN { result, approval }
```

Illustrative lowering only; this is not the artifact wire format or serialized instruction encoding.

```
different source semantics
    ↓
common execution semantics
    ↓
common continuation model
```

The developer still wrote an ordinary async function with `await`. The compiler produced a resumable state machine. That is what “transparent” means in TCC. It does not mean arbitrary TypeScript or Python is supported.

With a host that answers effect key `generate` with `42` and later delivers event `"approved"` with payload `"ok"`, the completed result is `{ result: 42, approval: "ok" }`.

5. Continuation representation and commit semantics

A continuation is the explicit program state required to resume an execution. After the example above has completed `effect("generate")` and committed the wait, the logical object looks like this:

```
Continuation
-----
artifact:      <artifact hash>
next state:    State 2

live values:
  result = 42

control state:
  status = suspended
  waiting for "approved"

durable references:
  effect "generate" = completed
  wait = <stable wait identity>
-----
```

Logical fields aligned with the frozen continuation contract. Not a wire format.

For a suspended durable operation, the saved next-state identifier denotes the state entered when the outstanding operation is resolved; the outstanding durable reference prevents that state from advancing prematurely.

Figure B. What a continuation holds.

```
state identifier
+ live values
+ control state
+ artifact identity
+ durable references
```

A committed continuation includes, as applicable:

control position / next execution state;

- live values needed by future execution;
- structured control state (active `try` / `catch` / `finally` handlers);
- execution identity;
- artifact identity, engine format version, and language-semantics version;
- a host-confirmed revision;
- outstanding wait, timer, or child operation required by the execution model;
- a terminal result when the execution has completed.

Values that survive a checkpoint must be representable in the continuation's language-semantics version. For the current TypeScript subset that includes `undefined`, `null`, booleans, IEEE-754 binary64 numbers, strings, objects with string keys, and arrays without cycles. The Python subset maps supported values (including `None`, booleans, numbers, strings, dicts, and lists without cycles) onto the representations defined for its language-semantics version. Unknown value kinds are not coerced.

Continuations are interpreted under the language-semantics version recorded by the producing artifact. TypeScript and Python therefore share the TCC continuation model without requiring their source-language value semantics to be identical. A frontend is responsible for mapping its supported values and control constructs into the representations defined for its language-semantics version.

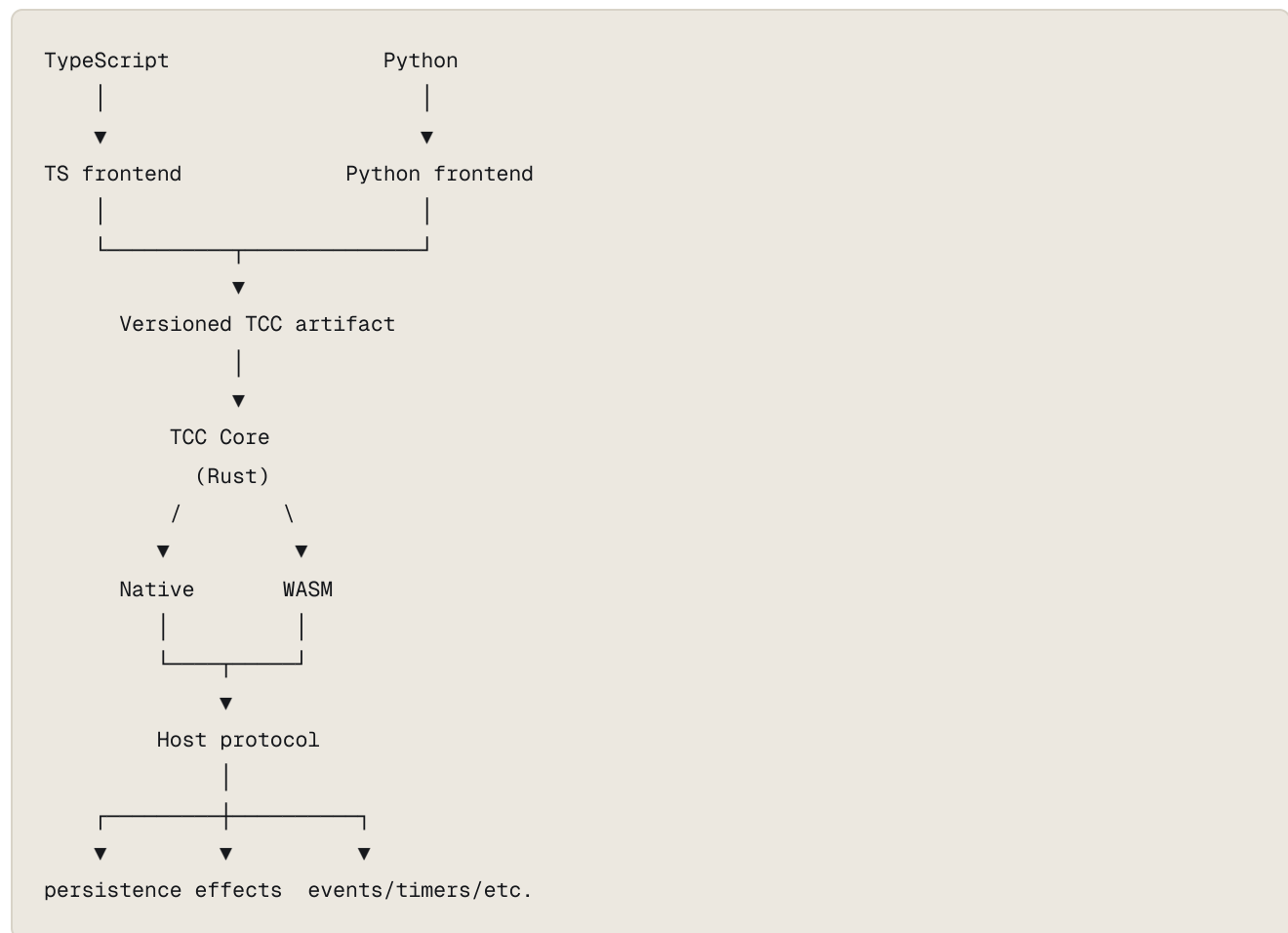
A continuation is **committed** when the host confirms persistence of that revision. Unconfirmed in-memory updates are not recoverable state. If a host reconstructs continuation bytes from a log or from deltas, the result must still be a continuation in this logical format. That reconstruction is not execution of completed program instructions.

Resume must fail if the stored artifact identity does not match the artifact being loaded. The engine does not rewrite a continuation onto a different compilation of the program.

6. Portable TCC engine architecture

The portable engine separates language frontend, execution core, embedding, and host.

Figure C. Portable engine.



Three things TCC keeps separate

Source language: TypeScript, Python, future frontends

Execution representation: versioned TCC program artifact + logical continuation

Engine embedding: native Rust, WebAssembly

A new language does not require a new recovery architecture. A new engine embedding does not require a new continuation format.

TCC separates language portability from runtime portability. Frontends determine which source languages can produce TCC artifacts. Engine embeddings determine where

those artifacts can execute. TypeScript and Python therefore do not require separate durable-execution engines; both target the same core semantics.

The core defines execution semantics: stepping the artifact, owning continuation state, and emitting host requests. It does not own SQLite, Cloudflare, or any other store. It does not depend on a particular operating system or cloud provider. It builds natively and to `wasm32-unknown-unknown`. The WebAssembly target does not use WASI networking, filesystems, or threads.

The JavaScript host embedding uses the WebAssembly build. The Python host embedding uses the native Rust engine. Both expose the same logical engine/host interaction model. WASM is therefore not a requirement for TCC execution. It is one portable embedding of the Rust core. Native and WASM execution are required to preserve the same externally observable TCC semantics, including for artifacts produced by the Python frontend.

WASM is not the persisted continuation format. WASM linear memory is never checkpointed as the TCC continuation.

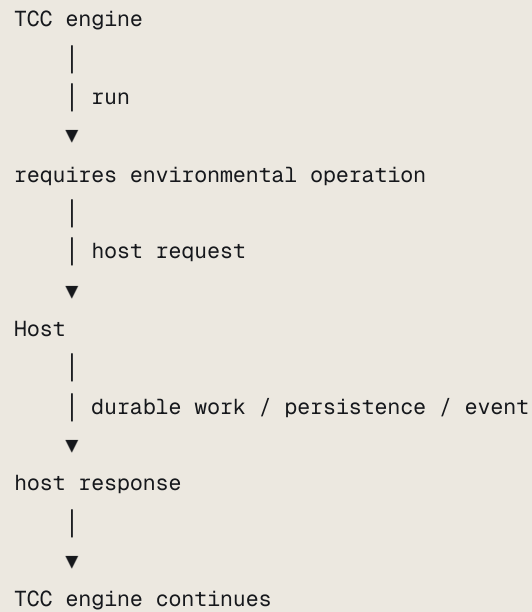
TCC is not built on WASM snapshots. The TCC engine is implemented in Rust and can be embedded natively or through WASM, while persisted continuations remain versioned logical execution state independent of the runtime's process memory.

Native and WASM builds execute the same core. For a given artifact, continuation, and host responses they must produce the same validation result, continuation encoding, and step outcomes.

The current TypeScript and Python subsets have been compiled to this core. That is implementation evidence for the architecture in this section. It is not a substitute for the timed recovery comparison in Section 14, which used the earlier research prototype.

7. Host protocol and durable recovery

The engine runs until it needs the environment, then yields.



The current WebAssembly embedding exposes a coarse-grained ABI for starting or resuming an execution, advancing it until environmental interaction is required, and supplying the corresponding host response. JSON is used by the present reference binding; neither JSON nor the specific ABI representation is essential to TCC. The ABI is not claimed to have zero overhead. `SharedArrayBuffer` and raw-memory checkpointing are not part of the model.

Host requests cover continuation persistence, effects, waits, timers, child execution, and artifact retrieval. The host replies with confirmations, effect results, event payloads, timer firings, child results, cancellation, or artifacts. A response that does not match the outstanding request is an error. The engine's in-memory outstanding request is not recoverable; the host owns durable state.

Reference host

A Node/SQLite host is a **reference implementation** of that protocol. It is not part of the core and not Trigora Cloud. It durably stores artifacts, continuations, effect journal state and results, waits, and events. A Python host embedding uses the same logical protocol against the native engine.

Persist-before-ack. Persistence is confirmed only after the relevant durable store transaction commits. An uncommitted acknowledgement does not make a continuation revision authoritative. A process death before commit cannot make an uncommitted continuation recoverable.

Artifact pinning. Recovery loads the artifact identified by the stored artifact identity. A continuation does not silently resume against the newest compilation.

Effect journaling. Completed effect outcomes are reused on recovery. If an effect is known completed, the provider is not called again for that durable identity. If failure occurs in the ambiguous external-effect window (Section 9), the existing retry and idempotency semantics apply.

Wait recovery. A waiting continuation survives process death. Recovery restores and re-registers the wait rather than replaying the execution that originally reached it.

Leases. The reference host enforces that at most one worker may advance a given execution at a time. A second worker cannot concurrently advance it while another worker owns a live lease. Continuation writes use a revision check.

Duplicate delivery. Duplicate events after terminal completion are no-ops under the tested semantics.

Figure F. Crash windows.

Failure point	Recovery behavior
Before checkpoint commit	Resume previous committed continuation
After checkpoint commit	Resume new committed continuation
Before effect starts	Effect may be attempted
After committed effect result	Reuse result; do not invoke provider
After external success but before durable result	Retry/reconcile using stable identity
While waiting	Restore continuation and re-register wait

Fresh-process restore

Figure D. Process death and resume.

```
Process A

start artifact
  ↓
effect("generate")
  ↓
42
  ↓
waitForEvent("approved")
  ↓
commit continuation
  ↓
SIGKILL
  ×

durable store

Process B

open durable store
  ↓
load artifact by stored hash
  ↓
load committed continuation
  ↓
create fresh TCC runtime
  ↓
hydrate continuation
  ↓
resume from saved state
  ↓
deliver "approved"
  ↓
return { result: 42, approval: "ok" }
```

A fresh runtime hydrates the committed continuation against the artifact named by it, and continues from the saved state.

The completed prefix and completed `effect("generate")` are not re-executed during continuation recovery.

Workers in the reference-host tests are killed with `SIGKILL` at persistence, effect-provider and journal, wait-registration, and wait-checkpoint hooks. A completely new process opens the same durable store and resumes. Tests check that a committed wait checkpoint does not replay the previously completed effect; that duplicate post-completion events are no-ops; that a second owner cannot steal a live lease; that restore uses the stored artifact identity; and that a crash before the first checkpoint restarts from the artifact while still honoring an already completed effect journal entry when one exists.

These tests are not a formal proof of correctness.

8. Programming and execution model

A Trigora program is a long-running function whose progress can be persisted and resumed. TypeScript and Python are the current authoring languages. The examples in Section 4 are the authoring model, not a hidden second API.

Ordinary control flow between durable operations is ordinary. The program is not a state machine the developer authors as data. Recovery must still have an explicit state machine underneath; a declared language subset exists because of that requirement.

Durable boundaries are the operations that may stop and later continue without replaying committed work. In the current portable frontends they include:

`effect` — durable external work;

`waitForEvent` / `wait_for_event` — suspend without holding compute;

`sleep` — a timer wait;

`invoke` — a durable child execution.

The two frontends need not expose identical source-language syntax. They must preserve the corresponding TCC durable-execution semantics for constructs they declare supported.

Effects run application code against the outside world. Their outcomes, once committed, are reused on recovery rather than re-executed. Section 9 treats the window in which that commitment has not yet happened.

Waits persist the continuation and a wait record, then drop the live process. An event, a timeout, or cancellation resolves the wait. Duplicate deliveries after a terminal wait status are no-ops.

Child execution is a separate durable execution: own identity, own continuation, own effects and waits. The parent suspends until the child reaches a terminal outcome. A logical invoke creates at most one child and delivers that outcome at most once.

Cancellation is a durable transition, not a best-effort signal on a live worker. The host delivers it. It takes effect at the next durable boundary. It does not magically interrupt an in-flight external call. A cancelled execution does not resume ordinary work.

Current TypeScript subset

A construct counts as supported in the portable engine only under the conformance rule in Section 17. Durable operations are identified through resolved imports from the corresponding TCC authoring surface, and unsupported constructs are rejected at compile time.

Control flow. `if / else`; `while`; simple `for (init; cond; assignment)`; unlabeled `break` and `continue`; nested blocks.

Durable execution. Multiple durable operations; frozen identities so a loop that re-hits the same effect reuses a completed journal entry; `effect`; `waitForEvent`; `sleep`; `child invoke`.

Values and operators. Objects and arrays (no spread, no cycles, no object identity); `===` / `!==`; unary `!`; numeric `<` `<=` `>` `>=`. Literals: `undefined`, `null`, booleans, numbers,

strings. Property read `obj.key`.

Errors. `throw`; `try`; `catch`; `finally`. Host-side failed-effect retry uses the same idempotency key.

Unsupported constructs — including labeled `break` / `continue`, `for-in` / `for-of`, closures, nested functions, arithmetic, logical `&&` / `||`, destructuring, optional chaining, `Promise.all` / `Promise.race`, and parameters on the entry function — fail at compile time rather than silently changing semantics. Effect keys, event names, and invoke names must be string literals.

The earlier research prototype evaluated a broader grammar, including two-branch structured `Promise.all` of child invokes. That work is part of the generated-test evidence in Section 17. It is not claimed as current portable-engine surface.

Current Python subset

The same conformance rule applies. Authoring uses resolved `trigora` imports (`effect`, `wait_for_event`, `sleep`, `invoke`). A locally declared `effect` is not a durable operation.

Control flow. `if` / `else`; `while`; simple `for` over a range-style condition; unlabeled `break` and `continue`; nested blocks.

Durable execution. Multiple durable operations; frozen identities with the same journal-reuse rule; `effect`; `wait_for_event`; `sleep`; child `invoke`.

Values and operators. Dicts and lists (no cycles, no object identity); equality and inequality; boolean `not`; numeric comparisons. Literals: `None`, booleans, numbers, strings. Key/attribute reads on supported objects.

Errors. `raise`; `try`; `except`; `finally`. Host-side failed-effect retry uses the same idempotency key.

Unsupported constructs fail at compile time rather than silently changing semantics. Closures, nested functions, classes, generators, arbitrary CPython libraries, and

`asyncio.gather` are not in the declared subset. Effect keys, event names, and invoke names must be string literals. This paper is not a complete compatibility matrix.

9. External effects and nondeterminism

Long-running programs do not stay inside the runtime. They call tools, charge cards, send mail, write to other systems. Those calls are not functions of the continuation. They are effects on the world.

```
Program
  ↓
effect boundary
  ↓
external system
  ↓
record durable outcome
  ↓
continue
```

TCC gives each effect a stable identity and a journaled lifecycle: pending, started, completed or failed. Once an outcome is committed, recovery reuses it. The application effect handler is not run again for that identity.

Continuation checkpointing does not eliminate a fundamental distributed-systems ambiguity. If the process dies after the external call succeeds and before the outcome is committed, the engine cannot universally know whether the side effect occurred. Recovery may retry with the same identity. Downstream systems that honor an idempotency key return the original result. Systems that do not may see the call twice. Idempotency is part of the programming model where the callee supports it. Magical exactly-once I/O for arbitrary third-party APIs is not. Trigora does not rewrite unknown SDK calls to inject keys.

This logging-and-retry pattern is not a TCC invention. Olive [2] showed that logging intents against cloud storage can give snippets of code exactly-once semantics despite failures and duplicate execution. Beldi [3] adapted that log-based approach to stateful

serverless workflows. TCC uses the same family of idea — durable identity, recorded outcome, retry with that identity — inside a continuation-checkpointed execution. The contribution evaluated here is the recovery architecture around those effects, not a new theory of exactly-once I/O.

Where idempotency is unavailable, the remaining tool is reconciliation: the program must be written so that a duplicate call is detectable or harmless. That is an application obligation. The runtime does not erase it.

10. Waiting, timers, children, cancellation, and control flow

Continuation checkpointing would be a toy if it only covered a straight line of local steps. The executions that motivate the work suspend, fan out, and cancel.

External waits. The program reaches an event wait (or a timer). TCC commits the continuation and the wait, then releases the process.

```
execute → suspend → [hours] → event → restore → continue
```

Nothing in the model holds a worker for the duration of the wait. Wake is an event, a timeout, or a cancel. Exactly one of those terminal transitions wins. A runtime restart while waiting does not, by itself, resume the program. Recovery re-registers the wait.

Child execution. `invoke` starts a separate durable execution. The parent's continuation records that it is waiting on that child, then suspends. The child has its own continuation. Nested invoke is a tree, not a call stack that recovery must replay.

Structured joins (research-prototype semantics). A structured join of child executions is a barrier:

```
parent
├── child A
└── child B
  ↓
  join
  ↓
  continue
```

The parent does not resume on the first success. It resumes when the join's aggregate outcome is determined. Fail-fast cancellation of siblings does not rewrite a terminal aggregate that has already been observed. Recovery does not create a duplicate child for work that already exists. The research prototype evaluated a two-branch `Promise.all` of child invokes under those rules. The current portable TypeScript subset does not yet compile `Promise.all`.

Cancellation is observed at the next durable boundary. It does not interrupt an in-flight effect. A cancelled execution does not resume ordinary work.

Those are the semantics recovery has to preserve. Generated tests on the research prototype, and compile/native/WASM/ `SIGKILL` conformance on the portable engine, check the constructs each frontend actually supports.

11. Recovery semantics

The headline property is exact:

Recovery from a committed continuation checkpoint does not require prefix replay.

A continuation checkpoint is **committed** when it has become the authoritative resumable state for that execution. Recovery loads that state and continues. Completed effect outcomes are reused. Child identities stay stable. Waits are re-armed, not recreated.

If the process dies **after** a boundary commits, recovery resumes from the new committed state. If it dies **before**, recovery resumes from the previous committed state. There is no third, half-applied continuation. The program may re-enter the work that sat between those two commits. That is why durable operations have identities, and why the effect window in Section 9 exists. Section 7 tabulates those crash windows.

This is **not** a claim of:

exactly-once external effects — the engine cannot universally know whether an external API succeeded if the process dies after the side effect and before the completion record;

constant-time recovery — recovery work is associated with live continuation state, not with a constant;

zero recovery work — restore and resume are real work;

arbitrary-language support — each frontend is a declared subset;

immunity from code and version compatibility — a continuation resumes against the artifact that produced it. Changing the program under a live continuation is a deployment problem, not something recovery silently solves.

Those limits are part of the model, not footnotes on an otherwise unlimited claim.

12. Code and artifact evolution

Two problems are easy to collapse and should not be.

Recovery against the same artifact

Supported:

```
Continuation V1
  ↓
Artifact V1
  ↓
resume
```

The continuation stores the artifact identity that created it. Restore loads that blob. It does not substitute whatever source is currently compiled.

Migrating an in-flight continuation to new code

Not currently solved automatically:

```
graph TD; A[Continuation V1] --> B[?]; B --> C[Artifact V2]
```

Continuation V1
↓
?
↓
Artifact V2

Artifact pinning makes recovery correct against the creating program. It does not migrate a live continuation onto incompatible code. Explicit continuation migration or version transformation is possible future work, not a current capability.

13. Evaluation methodology

The measurements below are from a frozen local research prototype (TypeScript compiler and runtime in the Trigora research lab). They are intended to make a scaling claim interpretable, not to reproduce the engine. They were **not** collected on the portable Rust/WASM implementation or the SQLite reference host. Those components provide the architectural and crash-recovery evidence in Sections 4–7 and 17. They do not replace the numbers in Sections 14–16.

Environment. All reported numbers are from a single-machine harness on Apple silicon (arm64, macOS), not a multi-node production cluster. The Temporal comparison uses the TypeScript SDK 1.23 against Temporal Server 1.31 (CLI 1.8) [1]. One Temporal series uses the SDK’s in-process test environment. A second series uses a local Postgres-backed Temporal server with a freshly started worker. Those two series are not merged into one line. They differ in persistence, RPC, and worker lifecycle.

The TCC side of the recovery comparison restores from the prototype's committed continuation store on the same machine. It does not go through Temporal, and it does not use a production object store. Matching the *workload* (depth, live size, blocked wait) is the fairness condition. Matching the *storage stack* is not claimed.

Workload. The flagship comparison is a matched synthetic depth sweep: the same application-shaped work at durable-boundary depths 10, 100, and 1,000, with TCC live continuation state held at approximately 4 KB and no artificial activity latency. History event counts on the Temporal side were 64, 604, and 6,004 at those depths.

What is timed.

TCC recovery latency: time from loading a committed continuation until reaching the verified suspended state.

Temporal reconstruction latency: time from worker readiness until replay reaches the equivalent verified suspended state. Worker creation is excluded. Including it would mix process-lifecycle cost into reconstruction.

Runs and statistics. Each cell is three runs. Reported values are min / median / max. Quartiles and high percentiles are not reported: $n=3$ is too small for stable estimates. Medians are the headline; ranges are in the figure bands.

Fresh-worker condition. Temporal reconstruction is measured on a worker that was not the sticky owner of the in-flight execution. Default sticky routing on the server-backed path can introduce a multi-second handoff floor that masks replay scaling; the server series used for comparison bounds that handoff so reconstruction cost is visible. That choice isolates replay. It is not how every production Temporal deployment is operated.

Healthy-path cost (Section 16) is a different experiment: TCC continuation persistence versus a matched history-prefix persistence baseline, at the same durability and hardware budget, with concurrency 1, 8, and 32 and three runs. Throughput is committed transitions per wall-clock second. This is not a Temporal comparison and not a recovery-semantics claim. The baseline exists so healthy-path cost can be discussed without mixing Temporal's worker and RPC stack into a persistence argument.

Scope. These measurements are not a production Temporal Cloud benchmark, not a cross-hardware study, not a representative agent trace, and not a claim about Trigora Cloud. A reader should be able to judge fairness from the above. The scripts that implement the prototype are not required for that judgment, and they are not published as part of this paper.

14. Recovery versus execution history

Figure 1 is the central result. It reports the frozen research prototype, not the portable Rust engine.

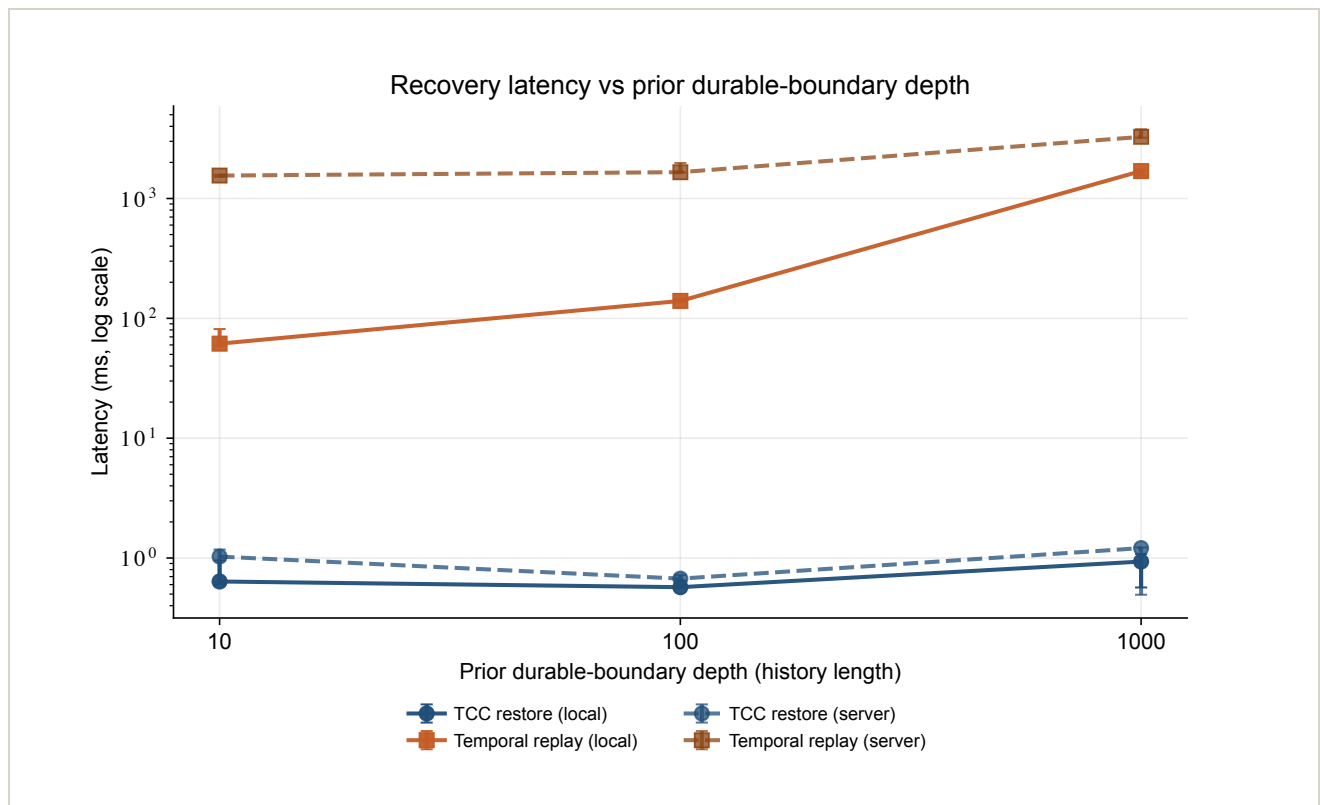


Figure 1. Recovery latency versus prior durable-boundary depth. TCC: time from loading a committed continuation until the verified suspended state. Temporal: time from worker readiness until replay reaches the equivalent state (worker creation excluded). Live continuation $\approx$ 4 KB. Solid: in-process Temporal test environment. Dashed: Postgres-backed Temporal with bounded sticky handoff. Markers: median of 3 runs; bars: min-max.

At depths 10, 100, and 1,000, TCC recovery medians were 0.64 ms, 0.57 ms, and 0.93 ms (ranges 0.60–0.96, 0.56–0.58, 0.57–1.08). Temporal in-process reconstruction medians were 61 ms, 140 ms, and 1.69 s (ranges 59–81, 138–145, 1.56–1.74 s).

The reading is architectural:

History-replay reconstruction grew with retained history in the evaluated configuration, while TCC recovery remained primarily tied to live continuation state.

One curve moves with history; the other does not, once live state is held fixed.

The server-backed Temporal series is higher in absolute terms (medians 1.55 s, 1.66 s, 3.27 s at the same depths) and still increases with history when sticky-worker handoff is bounded. TCC recovery on that sweep stayed in the 0.5–1.2 ms band. The two Temporal series answer slightly different operational questions; both are consistent with history-sensitive reconstruction.

The same history-insensitive TCC shape, still at approximately 4 KB parent live state, reproduced on three richer local patterns that are not plotted here: external suspend/wake, parent-child invoke with child history held fixed, and a two-child join with both child histories held fixed. Temporal fresh-worker reconstruction still increased with parent history in those matched local workloads (on the order of ~50–70 ms at depth 10 through ~1.7 s at depth 1,000). They are corroboration that Figure 1 is not an artifact of a single sequential loop.

15. Recovery versus live state

If recovery does not track history depth, it still tracks something. In TCC that something is the state required to continue.

A mathematically cleaner version includes the execution environment E , because storage and runtime implementation affect absolute latency:

$$T_{\text{restore}}(S, E) \approx T_{\text{load}}(S, E) + T_{\text{decode}}(S, E) + T_{\text{hydrate}}(S, E) + T_{\text{resume}}(E)$$

Where:

S = committed live continuation state

E = storage, hardware, locality, and runtime environment

H = accumulated execution-history depth

The central TCC property can then be expressed as:

$$\left. \frac{\partial T_{\text{restore}}}{\partial H} \right|_{S,E} \approx 0$$

In plain English:

At fixed live state and under the same environment, increasing completed execution depth should not materially increase TCC restoration work.

Live-state size does matter:

$$\frac{\partial T_{\text{restore}}}{\partial S} > 0$$

That does not imply a perfectly linear relationship—only that larger continuations require more loading, decoding, and hydration work.

At fixed workflow depth, increasing live continuation size from the compact ~4 KB used in Figure 1 to 1 MB raised recovery to approximately 5 ms. An earlier full-checkpoint characterization, not the Figure 1 harness, rose from approximately 1.8 ms at 1 KB to approximately 6 ms at 1 MB.

Within the tested range, recovery tracks checkpoint and live-state size much more closely than accumulated execution-history depth.

A continuation that carries a megabyte of live values is more expensive to restore than one that carries four kilobytes. That is expected. It is also the intended cost driver:

replay is history-sensitive; TCC is live-state-sensitive. Executions that keep a small live working set, even after many durable steps, are the ones for which the Figure 1

shape appears. Executions that accumulate live state without bound will pay for that state on every restore.

16. Healthy-path cost

A fair objection to Figure 1 is that recovery looks cheap because the architecture paid during ordinary execution — by writing a continuation at every boundary.

That cost is real. The research prototype was measured against a matched history-prefix persistence baseline, at the same durability and hardware budget, so the objection could be answered in persistence terms rather than as another cross-engine ratio.

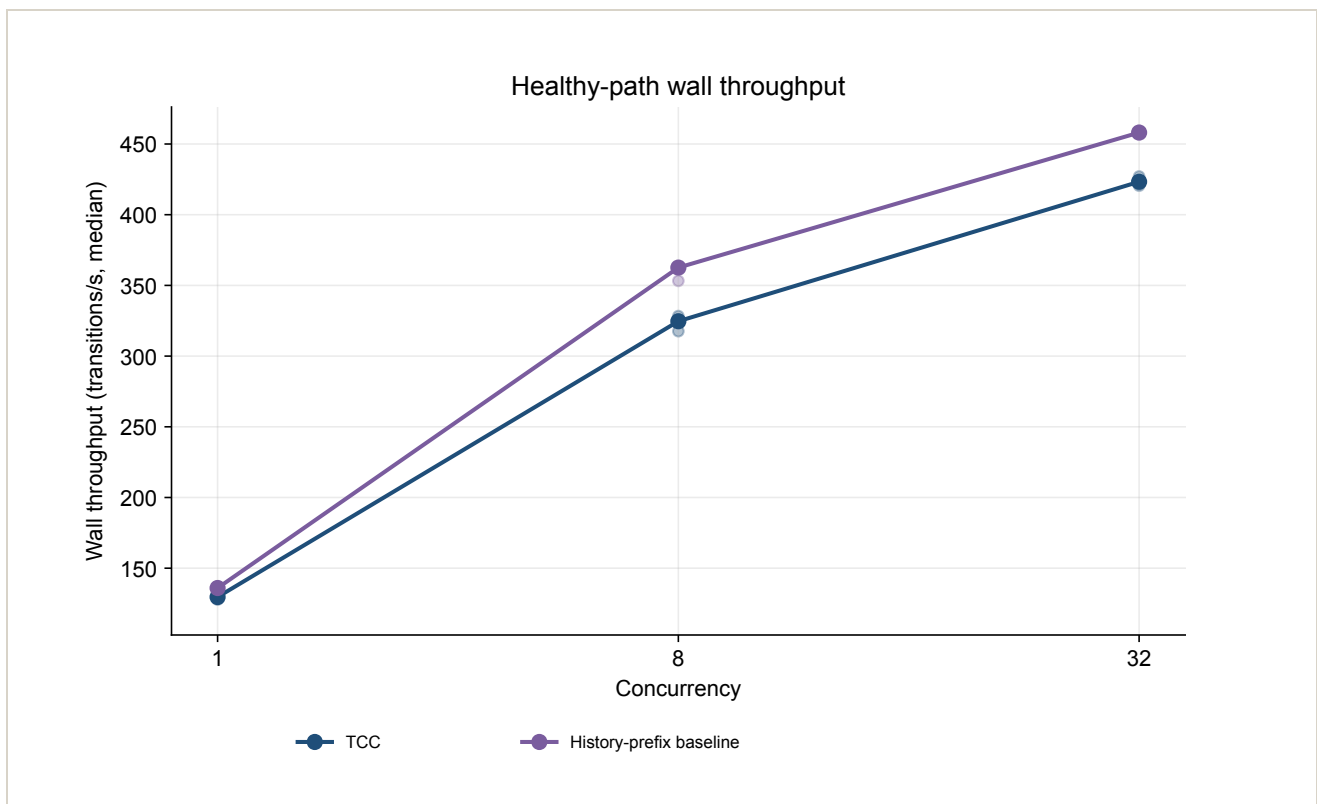


Figure 2. Healthy-path wall throughput: TCC continuation persistence versus a matched history-prefix persistence baseline. Throughput is committed transitions per wall-clock second. Concurrency 1, 8, 32; n=3. Frozen research prototype.

At concurrency 8, the prototype delivered 10.5% lower wall throughput than the matched baseline (medians 325 vs 363 transitions/s).

TCC currently trades measurable healthy-path execution cost for direct continuation recovery.

Profiling indicates that the remaining difference is dominated by prototype implementation and coordination overhead rather than simply by the volume of persisted state. That is as far as the evidence goes. It is not a claim that the healthy-path cost has been eliminated, and it is not a measurement of the portable Rust engine.

Short executions with small histories may gain little from this trade. The architecture is a bet that for some long-running dynamic programs, paying a modest common-path premium is preferable to reconstructing position from a growing prefix.

17. Semantic and crash-recovery validation

A recovery architecture that never runs the supported language is not an execution system. There are two complementary forms of evidence. They are not one statistic.

Research-prototype generated validation

The frozen TypeScript research prototype includes a compiler and runtime for its declared subset, crash injection at durable-boundary windows, and generated tests.

The strongest generated campaign reported here: **50,000 cases, zero semantic failures in the evaluated suite**. Separate campaigns covered sequential control flow with waits and exceptions, nested invoke, cancellation, and structured joins. Each campaign was preceded by a clean 10,000-case pass on the same grammar.

No semantic failures were observed across 50,000 generated cases within the supported language subset.

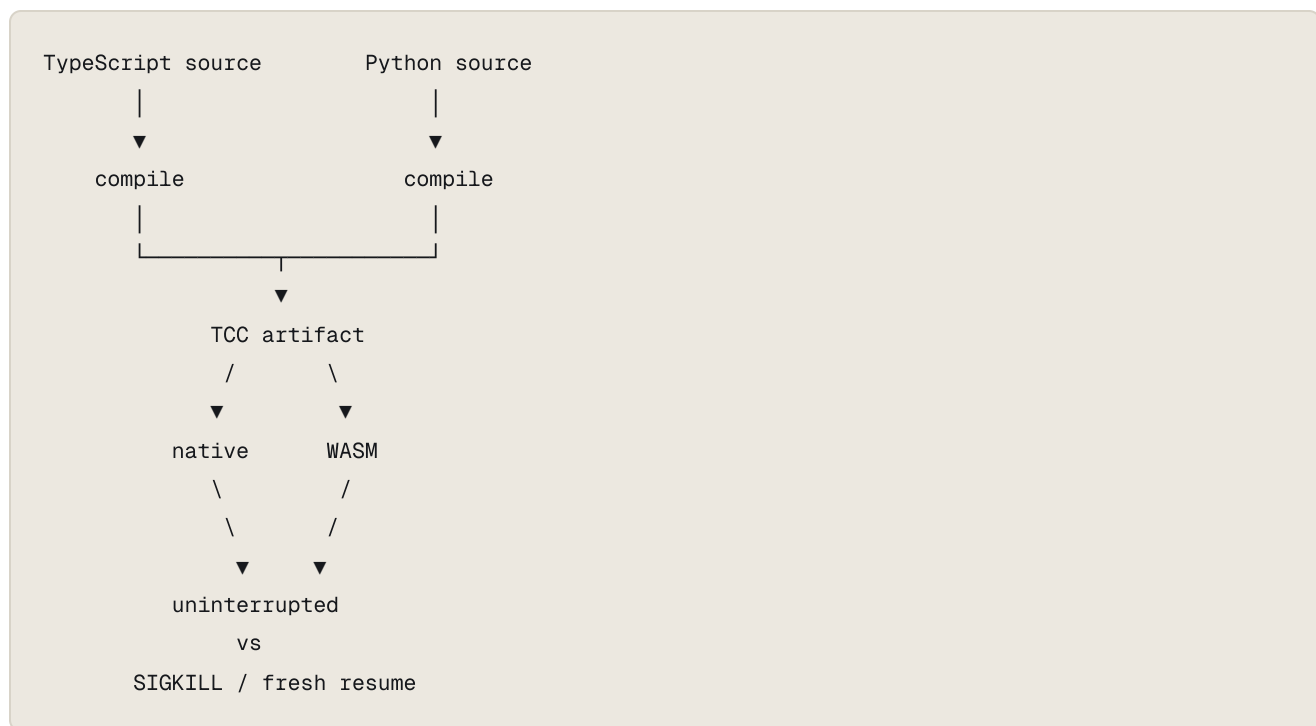
That is evidence that, for the grammars tested, crash recovery and resume from committed continuation checkpoints without prefix replay did not violate the invariants those grammars encode: completed work is not rerun, child identities remain stable, a

wait has one terminal winner, a join does not duplicate children, a cancelled execution does not resume ordinary work.

Compiler and runtime tests, plus crash injection at effect, wait, invoke, cancel, and join windows, sit underneath those campaigns. They are ordinary software tests, not a second empirical headline. They are not measurements of the portable engine.

Portable-engine conformance and crash validation

A frontend construct counts as supported only when its compilation and execution behavior agrees with the TCC semantics across the engine targets and recovery conditions claimed for that frontend.



Python-produced artifacts are executed natively and through the WebAssembly engine as a conformance target. The Python host embedding remains native. Native and WASM must agree on externally observable semantics. SQLite remains host-side and is not part of the core.

This is implementation validation, not a formal proof of correctness. It does not reuse the 50,000-case figure, and it does not produce new recovery-latency numbers.

18. What TCC changes

Under a history-replay architecture, **durable history is authoritative** and **current execution position is reconstructed**. Recovery's cost driver is, typically, how much history has been retained. Dynamic control state exists in the program, then disappears, then is rebuilt by running the program against the log.

Under TCC, **committed continuation state directly represents a resumable execution position**. History can still exist for audit. Recovery does not use prefix replay to rebuild the present. The cost driver is live continuation state plus whatever persistence work the common path performed to keep that state current.

Those are different engineering programs. Replay inherits a mature literature, operational tooling, and a simple mental model: the log is the workflow. Continuation restore inherits a different literature — checkpoint/restart, CPS, intent logging — and a different failure mode: large live state, version skew between continuation and code, compiler surface area.

TCC is not inherently better for every workload. Executions that finish quickly, retain little history, and rarely crash will not notice reconstruction cost. Continuation persistence has a common-path cost, measured in Section 16. Large live state has a recovery cost, measured in Section 15. A declared language subset has a complexity cost that every additional construct pays in the compiler and in the test grammar.

The architecture is a choice about which variable should govern recovery. This paper's evidence is that the choice is real: in the evaluated configuration, history length was not the primary variable governing TCC recovery, and live continuation state was.

19. Limitations

The current evaluation establishes properties of the TCC architecture and of two implementations of it, not production performance guarantees for Trigora Cloud.

Language subsets. The TypeScript and Python frontends each implement declared subsets of their respective source languages. Unsupported constructs fail at compile time. The existence of multiple frontends does not imply arbitrary TypeScript or Python support. The surface will move as conformance expands. This paper is not that matrix.

Serialization and live state. Anything that must survive a checkpoint must be representable as continuation state. Values that cannot be serialized cannot be live across a boundary. Large live state is recoverable and expensive, as Section 15 shows.

External effects. At-least-once in the started-but-not-completed window. Idempotency keys where callees honor them. No rewrite of unknown SDKs. No universal exactly-once.

Code evolution. Continuations resume against the artifact that produced them. Migrating in-flight executions across incompatible program changes is not solved by restore.

Compiler and runtime complexity. A declared subset is the price of an explicit state machine. That complexity is a limitation of the approach, not a temporary omission of documentation.

Benchmark environment. Single-machine, synthetic depth, $n=3$, Apple silicon, Temporal test environment plus one local Postgres-backed server. No multi-node Temporal, no production-scale TCC store, no representative agent workload as a timed benchmark. Absolute milliseconds will move under different hardware, stores, and RPC.

Implementation maturity. The original measurements come from the frozen research prototype. A production-oriented portable TCC engine now exists in Rust. It targets native and WASM execution. Declared TypeScript and Python subsets compile to that core. A durable Node/SQLite reference host demonstrates fresh-process recovery; a Python host embedding uses the native engine against the same logical protocol. Neither is the hosted control plane. Distributed production operation, multi-tenant hosted operation, broader language coverage, production performance characterization, and operational hardening remain separate work. Healthy-path overhead on the prototype is still visible.

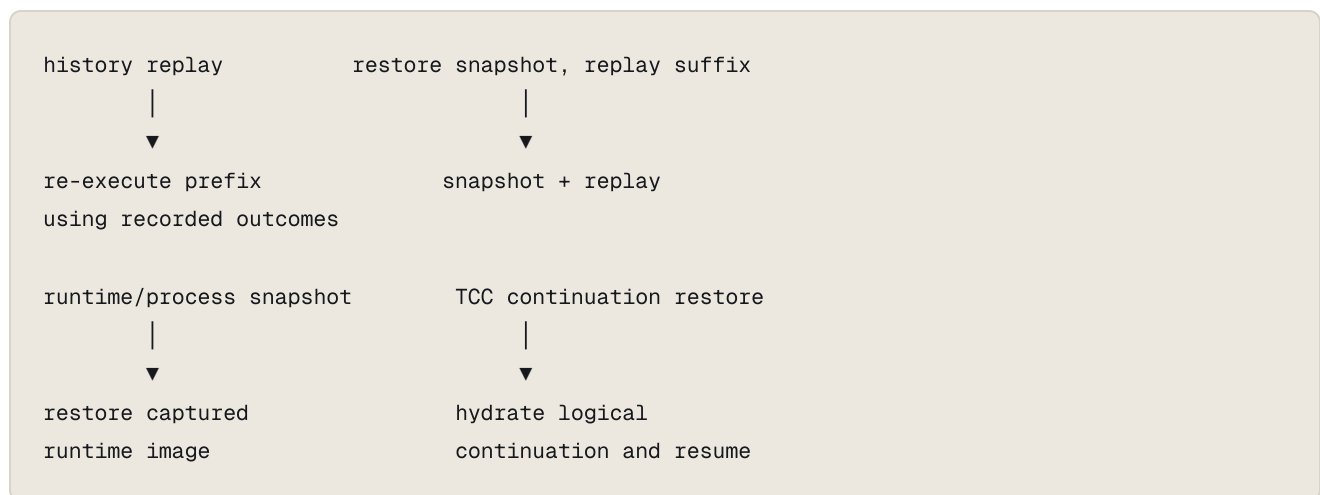
Absence of production-scale evaluation. Nothing in Sections 13–16 is a load test of Trigora Cloud, a multi-tenant tail-latency study, or a week-long agent trace. The argument that is meant to travel is the identity of the scaling variable, not the intercept.

What the headline is not. Not constant-time recovery. Not zero-overhead execution. Not a speedup ratio. Not a proof of correctness. Not a claim that every history-replay system behaves like the measured baseline. Not a claim that WASM snapshots are the continuation. Not a claim that the SQLite host is Trigora Cloud.

20. Related work and alternative recovery architectures

This paper does not argue that nobody thought of continuations. TCC explores a particular architecture for making continuation restoration the basis of a practical durable execution system, and evaluates its recovery and economic properties.

Figure E. Four recovery strategies.



Architecture	Durable recovery representation	Recovery mechanism
History replay	Event/execution history	Re-execute prefix using recorded outcomes
Runtime/process snapshot	Runtime memory/state image	Restore captured runtime image

Architecture	Durable recovery representation	Recovery mechanism
Snapshot + replay	Periodic snapshot + subsequent history	Restore snapshot, replay suffix
TCC	Logical continuation state	Hydrate continuation and resume

Whole-runtime or object-graph snapshots can be expensive if they are taken after every transition, and they are difficult to evolve across code versions. That is a real historical objection, including in early durable-execution prototypes that dumped an asynchronous object graph by reflection. TCC does not take that path. It checkpoints compiler- and runtime-controlled logical execution state at durable boundaries. Artifact pinning solves recovery against the creating program. It does not eliminate the in-flight migration problem in Section 12.

Kappa

Kappa [4] is the closest systems ancestor and deserves to be read that way, not as a buried citation.

Kappa runs ordinary-looking parallel Python on unmodified serverless platforms. Lambda functions are time-bounded; Kappa's answer is **continuation-based checkpointing** in user mode. A long task checkpoints, the lambda times out or fails, and execution restores on a fresh lambda. Kappa also provides a concurrency API (spawn, futures, queues) and treats nondeterminism and side effects as first-class: checkpoints exist so execution does not diverge, and in-system side effects are not blindly re-executed across timeouts.

What Kappa demonstrated: continuation checkpointing is a viable way to give long-running, concurrent programs a life beyond a single short-lived worker, without modifying the underlying compute platform.

What this paper evaluates is a different engineering object. Kappa's problem is serverless time-bounds and elasticity. TCC's problem is **durable execution** in the

workflow/agent sense: explicit durable operations, days-long waits without a live worker, child executions, joins, cancellation, and recovery whose cost should not track retained application history. The measurements here are against a history-replay durable-execution baseline, including healthy-path persistence cost, which Kappa did not treat as the question.

The addition of a Python frontend does not change this distinction. Python is a source language for TCC's durable-execution artifact model; TCC does not inherit Kappa's serverless time-bound execution model or concurrency API.

The lineage is real. The claim is not “continuations, newly invented.” The claim is that continuation restoration can be the recovery primitive of a durable execution system, and that the resulting scaling behavior is measurable.

Intent logging: Olive and Beldi

Olive [2] gives snippets of code exactly-once semantics against cloud storage by logging intents and re-executing unfinished work. Beldi [3] extends that log-based approach to federated stateful serverless functions: invocations, transactions, garbage collection. TCC's effect identities and recorded outcomes sit in this family (Section 9). Durable effect logging is prior art. TCC does not replace it and does not claim to.

History-replay durable execution

Temporal [1], and the broader class of workflow engines that reconstruct in-flight executions by replaying retained history, are the successful industrial model this paper measures against. Cadence is the lineage Temporal comes from. Azure Durable Functions [13] persist orchestration history and, on resume, re-execute the orchestrator from the start while substituting recorded task results. Cloudflare Workflows [14] persist completed step results on the Workers platform and resume using that retained step state. Those are published recovery models, not timed baselines. Neither is claimed to be identical to Temporal. The measured instance in this paper is Temporal's TypeScript SDK in the configuration of Section 13.

Restate [5] and DBOS [7] explore adjacent designs and were not timed here. Restate records handler progress and completed operations in a durable journal; recovery continues by replaying those recorded steps. That is journal-based durable execution, not the continuation-checkpoint recovery studied here. DBOS, building on the Apiary line of work [6], checkpoints workflow inputs and step outputs in a DBMS and can commit application database work together with the durability record. On recovery it re-enters the workflow and substitutes recorded step outputs. Both are interesting points on the same larger map — evidence that “durable execution” is no longer one product category with one recovery algorithm — and neither is claimed as worse.

Continuations, CPS, and checkpoint/restart

Capturing control as data is older than any of these systems. Continuation-passing style [8] is the linguistic form. Process checkpoint/restart [9] captures OS-level state so a job can move or survive. Incremental and copy-on-write checkpointing [10] exist because full snapshots are expensive — the same pressure TCC feels from live-state size, resolved here at the execution-state layer rather than at the process image.

TCC checkpoints declared logical execution state at language-level durable boundaries. That is not a POSIX process image, not a heap or object-graph snapshot, not a WASM linear-memory snapshot, and not a CPS transform the developer writes by hand.

Event sourcing

Event sourcing [11] stores the log of what happened and derives current state by folding it. Replay-based workflows are that idea applied to control position. Snapshot-plus-replay hybrids restore a periodic snapshot and fold the suffix. TCC keeps history as an audit artifact and keeps recovery off the fold. The two uses of a log should not be collapsed: one is how you know what happened; the other is how you know where to continue.

Structured concurrency

The join and cancellation semantics in Section 10 are in the spirit of structured concurrency [12]: children do not outlive the join that owns them in an unstructured way, and cancel is a durable transition down the tree. TCC’s contribution is not a new cancellation calculus. It is that those structures survive restore without being rebuilt from a parent history prefix.

21. Conclusion

TCC demonstrates that durable recovery need not require reconstruction of execution position through accumulated prefix replay. TypeScript and Python frontends compile declared source-language subsets into a common versioned TCC artifact model. A portable Rust core executes those artifacts natively and through WebAssembly, committing logical continuation state at durable boundaries. A fresh runtime loads the artifact that produced a continuation, hydrates its saved execution state, and continues without replaying the completed prefix.

In the evaluated research-prototype configuration, that dependence is visible: at fixed ~4 KB live state, TCC recovery stayed in a 0.6–0.9 ms band as history depth grew from 10 to 1,000, while history-replay reconstruction grew from tens of milliseconds to about 1.7 s. Recovery rose with live-state size. The prototype pays a measurable healthy-path cost for the privilege — about 10% lower wall throughput against a matched history-prefix persistence baseline at the tested concurrency.

The portable implementation provides an additional architectural result: the same TCC continuation and execution model can be targeted by multiple source-language frontends and executed through multiple engine embeddings without making process or WebAssembly memory the recovery representation. TypeScript and Python frontends target the same execution substrate; the Rust core can be embedded natively or through WebAssembly; and fresh-process crash tests validate continuation recovery against durable host state. A durable SQLite reference host validates recovery under injected process death, while conformance testing requires supported constructs to agree across compilation, native execution, WebAssembly execution, and

SIGKILL/resume. Work remains in expanding language coverage, productionization, distributed operation, and performance characterization of the portable engine.

We are applying this architecture to Trigora, a durable execution platform for long-running agents and dynamic software.

Acknowledgments

Measurements use a frozen research prototype and Temporal's publicly available SDK and server. Errors of interpretation are the author's.

References

Temporal Technologies. Temporal documentation: Workflows, Workers, and event history. <https://docs.temporal.io> (accessed 2026). Temporal TypeScript SDK 1.23; Temporal Server 1.31.

S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *OSDI*, 2016. (Olive.)

H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *OSDI*, 2020. (Beldi.)

W. Zhang, V. Fang, A. Panda, and S. Shenker. Kappa: A programming framework for serverless computing. In *SoCC*, 2020.

Restate. Restate documentation. <https://docs.restate.dev> (accessed 2026).

P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, L. Krishnamurthy, V. Le, C. Cano, J. Li, Y. Zhang, G. Candea, and M. Zaharia. Apiary: A DBMS-integrated transactional function-as-a-service framework. *CIDR*, 2023.

DBOS, Inc. DBOS documentation. <https://docs.dbos.dev> (accessed 2026).

G. J. Sussman and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. MIT AI Memo 349, 1975. See also D. P. Friedman and M. Wand, *Essentials of Programming Languages*.

J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IPDPS*, 2009. See also CRIU, <https://criu.org>.

J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. University of Tennessee Technical Report CS-95-302, 1995.

M. Fowler. Event sourcing. <https://martinfowler.com/eaDev/EventSourcing.html>, 2005.

N. J. Smith. Notes on structured concurrency, or: Go statement considered harmful. <https://vorp.us.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/>, 2018.

Microsoft. Azure Durable Functions documentation. <https://learn.microsoft.com/azure/azure-functions/durable/> (accessed 2026).

Cloudflare. Cloudflare Workflows documentation. <https://developers.cloudflare.com/workflows/> (accessed 2026).