TECHNICAL REPORT V0.1

Transparent Continuation Checkpointing

Recovering long-running programs from committed execution state

Omar Abdelrahman · Trigora · September 2026

CONTENTS

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.

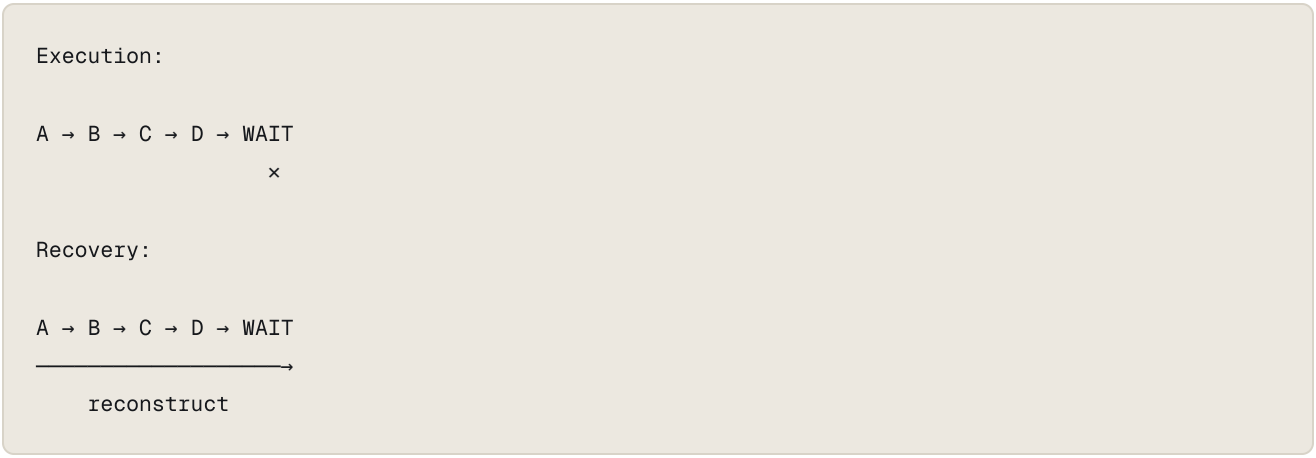
In controlled experiments at fixed live state, TCC recovery remained approximately 0.6–0.9 ms as execution-history depth increased from 10 to 1,000, while the evaluated history-replay reconstruction baseline increased from approximately 61 ms to 1.7 s. No semantic failures were observed across 50,000 generated cases within the supported language subset.

These 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.



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.

The tradeoff is equally structural. Reconstruction work can grow with retained execution history. Every recovery that rebuilds the present from the prefix must walk enough of that prefix to re-establish control position, local state, and outstanding waits. For short workflows the cost is rarely the point. For executions that accumulate thousands of durable steps — tool calls, retries, child completions, timer firings — history is no longer a small transcript. It is the thing recovery has to read.

TCC takes the other side of that tradeoff. At a durable boundary it persists a **continuation**: enough resumable state to continue, rather than enough history to reconstruct.

```
Execution:

A → B → C → D → [ checkpoint ]
                      ×

Recovery:

                 [ restore ]
                     ↓
                     D → ...
```

The rest of this paper defines that architecture, states what recovery from a committed continuation checkpoint does and does not mean, and reports measurements of the resulting scaling behavior.

2. Motivation: long-running dynamic software

The workloads that make the tradeoff urgent are long-running and *dynamic*. They are not a fixed DAG of steps known when the execution starts.

Agents are a useful motivating case. An agent may run for hours or days. It calls tools whose results are not known in advance. It waits for a human. It starts subagents. It branches on model output. It talks to nondeterministic external systems. Each of those 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.

```
Source program
      ↓
TCC execution
      ↓
durable boundary
      ↓
continuation checkpoint
      ↓

      crash

      ↓
restore continuation
      ↓
resume
```

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 mechanism needed to explain the result. The internal representation, serialization layout, and storage keys are implementation, and they are not specified here.

TCC is **transparent** relative to a declared language subset: the developer writes ordinary control flow and explicit durable operations, rather than assembling a workflow graph by hand. It is not transparent with respect to arbitrary host-language programs. Unsupported constructs are errors. Transparency is a property of the supported surface, not a claim that every TypeScript program is a durable execution.

Languages are how developers express programs. An execution representation is how a runtime understands them. TCC is how that representation is executed durably. This paper discusses the programming model and the recovery property. It does not specify an intermediate language.

4. Programming and execution model

A Trigora program is a long-running function whose progress can be persisted and resumed. The language frontend evaluated here is TypeScript. The example below is representative, not a hidden second API:

```
export async function researchAgent(input) {
  const sources = await effect("search", () =>
    search(input.query)
  );

  for (const source of sources) {
    await analyze(source);
  }

  const approval = await waitForEvent("approved");

  return effect("publish", () =>
    publish(approval)
  );
}
```

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 model they include:

- `effect` — durable external work;
- `waitForEvent` — suspend without holding compute;
- `invoke` — a durable child execution;
- structured `Promise.all` of child invokes — a join of independently durable children.

Effects run application code against the outside world. Their outcomes, once committed, are reused on recovery rather than re-executed. Section 6 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.

Structured concurrency. A join of child executions is a barrier. The parent resumes when all branches succeed, or earlier if a branch fails or is cancelled (fail-fast, with remaining siblings requested to cancel). Completed branches are not rerun.

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

The current TypeScript frontend supports common sequential control flow, loops, exceptions, waits, child executions, cancellation, and a restricted form of structured concurrency. Unsupported constructs fail at compile time rather than silently changing semantics. The exact support matrix is documentation, not architecture; it will move as the frontend expands. See Language support.

5. 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 6 exists.

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** — the 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.

6. 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.

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.

7. Waiting, children, and structured concurrency

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 `waitForEvent` (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.

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.

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

Join. A structured join of child executions is a barrier. 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.

Those are the semantics recovery has to preserve, and they are what the generated tests in Section 12 check.

8. Evaluation methodology

The measurements below are from a frozen local research prototype. They are intended to make a scaling claim interpretable, not to reproduce the engine.

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 11) 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.

9. Recovery versus execution history

Figure 1 is the central result.

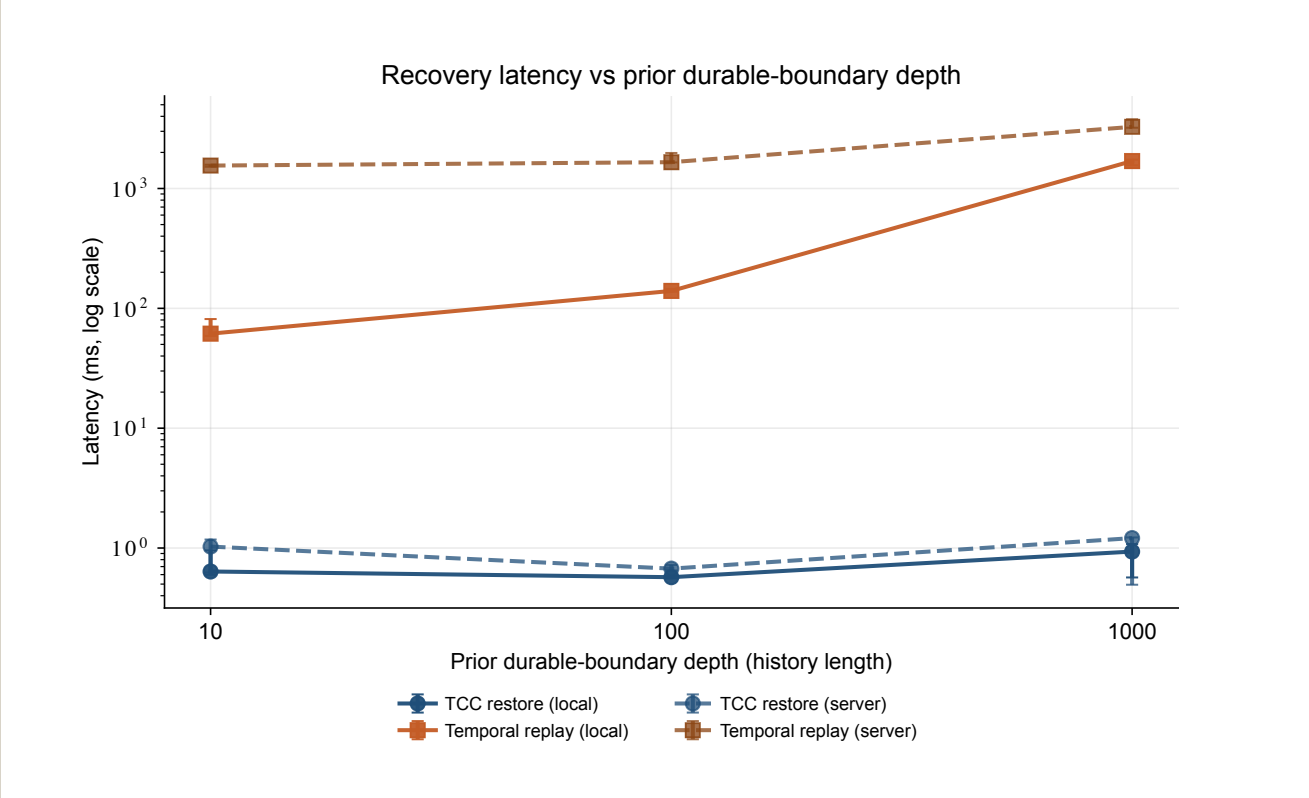


Figure 1. Recovery latency versus prior durable-boundary depth. TCC: time from loading a committed continuation until the verified suspended state. Temporal: time from worker readiness until replay reaches the equivalent state (worker creation excluded). Live continuation ≈ 4 KB. Solid: in-process Temporal 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.

10. 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.

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.

11. 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 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.

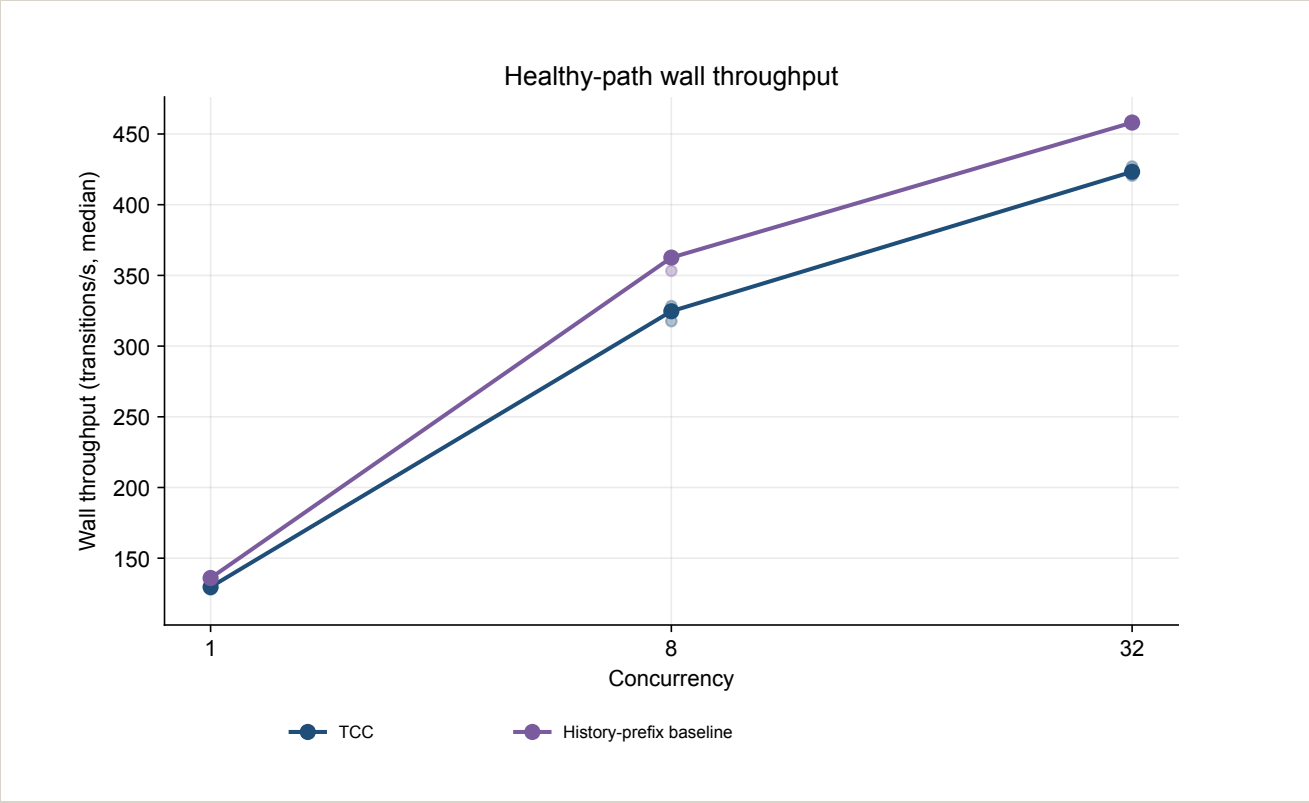


Figure 2. Healthy-path wall throughput: TCC continuation persistence versus a matched history-prefix persistence baseline. Throughput is committed transitions per wall-clock second. Concurrency 1, 8, 32; n=3.

At concurrency 8, the current 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.

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.

12. Semantic validation

A recovery architecture that never runs the supported language is not an execution system. The prototype includes a compiler and runtime for the declared TypeScript 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.

13. 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 11. Large live state has a recovery cost, measured in Section 10. 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.

14. Limitations

The current evaluation establishes properties of the TCC prototype, not production performance guarantees for Trigora Cloud.

Language subset. The evaluation covers only the current TypeScript frontend. Unsupported constructs fail at compile time. The current surface is documented separately and will change; 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 10 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.

Implementation maturity. The prototype is a research engine with crash injection and generated tests. It is not the hosted control plane. Healthy-path overhead is still visible. Distributed operation — multiple machines, shared durable storage, coordination of waits and children across failures of the control plane itself — is future work relative to the measurements here.

Absence of production-scale evaluation. Nothing in Sections 8–12 is a load test of Trigora Cloud, a multi-tenant tail-latency study, or a week-long agent trace. Absolute milliseconds will move under different hardware, stores, and RPC. 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.

15. Related work

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.

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 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 6). 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 8.

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 a declared execution state at language-level durable boundaries, not a POSIX process, 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. 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 7 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.

16. Conclusion

TCC demonstrates that durable recovery need not require reconstruction of execution position through accumulated prefix replay. By persisting resumable continuation state at durable boundaries, recovery can instead depend primarily on the state required to continue the current execution.

In the evaluated 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 prototype achieves this at that cost, and substantial work remains in language coverage, productionization, and distributed operation.

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/eaaDev/EventSourcing.html, 2005.

N. J. Smith. Notes on structured concurrency, or: Go statement considered harmful. https://vorpus.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).
