

Durable Execution: An End Run on the End-to-End Principle?

Johann Schleier-Smith
Temporal Technologies

Presented at: HPTS 2026

✕ @jssmith



HPTS 1997



Application Recovery: Advances Toward an Elusive Goal

David Lomet
Microsoft Research
Redmond, WA 98052
lomet@microsoft.com

Introduction

Persistent savepoints
"A *persistent savepoint*
is saved to stable storage.
If the system should
crash, it doesn't have to run
off." {BeNe97}

The value of persistent
programming compared to
"... the code is not only
context is now taken care of."
{GrRe93}

Traditionally, a savepoint
savepoint operation is

Phoenix: Making Applications Robust

Roger S. Barga
David B. Lomet
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
{barga, lomet}@microsoft.com

1. Introduction

At Microsoft Research, we have an ongoing project called *Phoenix*. The long term goal of the Phoenix project is to develop and apply recovery techniques that will enable application programs to survive system crashes and allow application programmers to deal effectively with application errors. The effort relies fundamentally on exploiting and extending conventional database recovery techniques. Our initial systems focus is to provide persistent database sessions to client applications across server failures, without the application itself needing to take measures for its recoverability. Our approach is based on (i) application message logging, in which the client application's interactions with the database server are captured and (ii) database session state materialized as database tables and logged on the server. To demonstrate the viability of our approach, we implemented a prototype system, named Phoenix/ODBC, that



Overview

- Durable execution from first principles
- Why is it taking off now?

Failure modes are pervasive in modern software

- Distributed systems are ubiquitous
- Networks, disks, processes, and services fail due to bugs, misconfiguration, overload, attacks, etc.
- Independent failure creates complex scenarios



Insidious problem has
been building for decades

Programs become riddled with error handling code

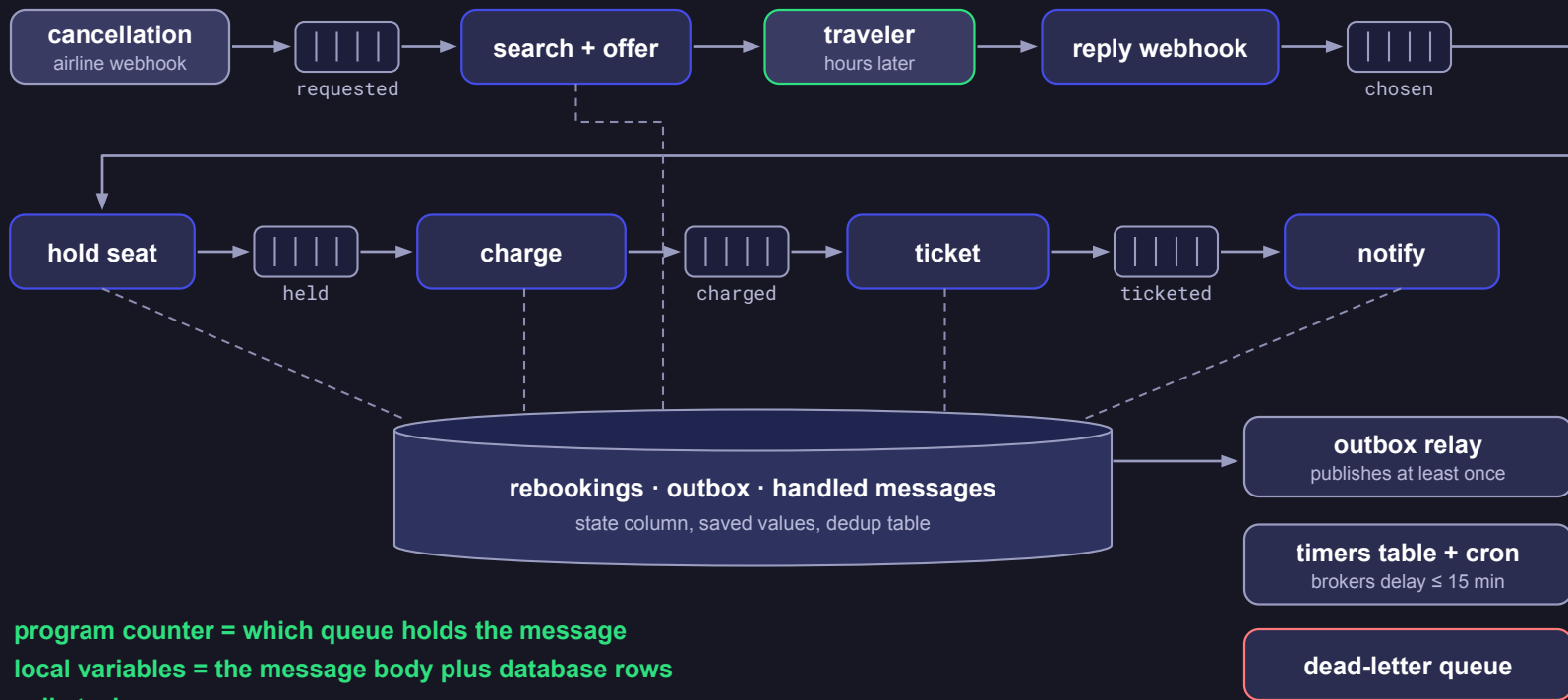
```
def rebook(trip):  
    for attempt in range(5):  
        options = airline.search(trip)  
        break  
    except Timeout:  
        sleep(backoff(attempt))  
    else:  
        raise GiveUp("search unavailable")  
    with circuit_breaker():  
        ranked = model.rank(options)  
    save_progress(trip, ranked)  
    send_choices(trip, ranked)  
    ...  
    # also need sweeper job for recovery
```



Transaction processing helps but has limitations

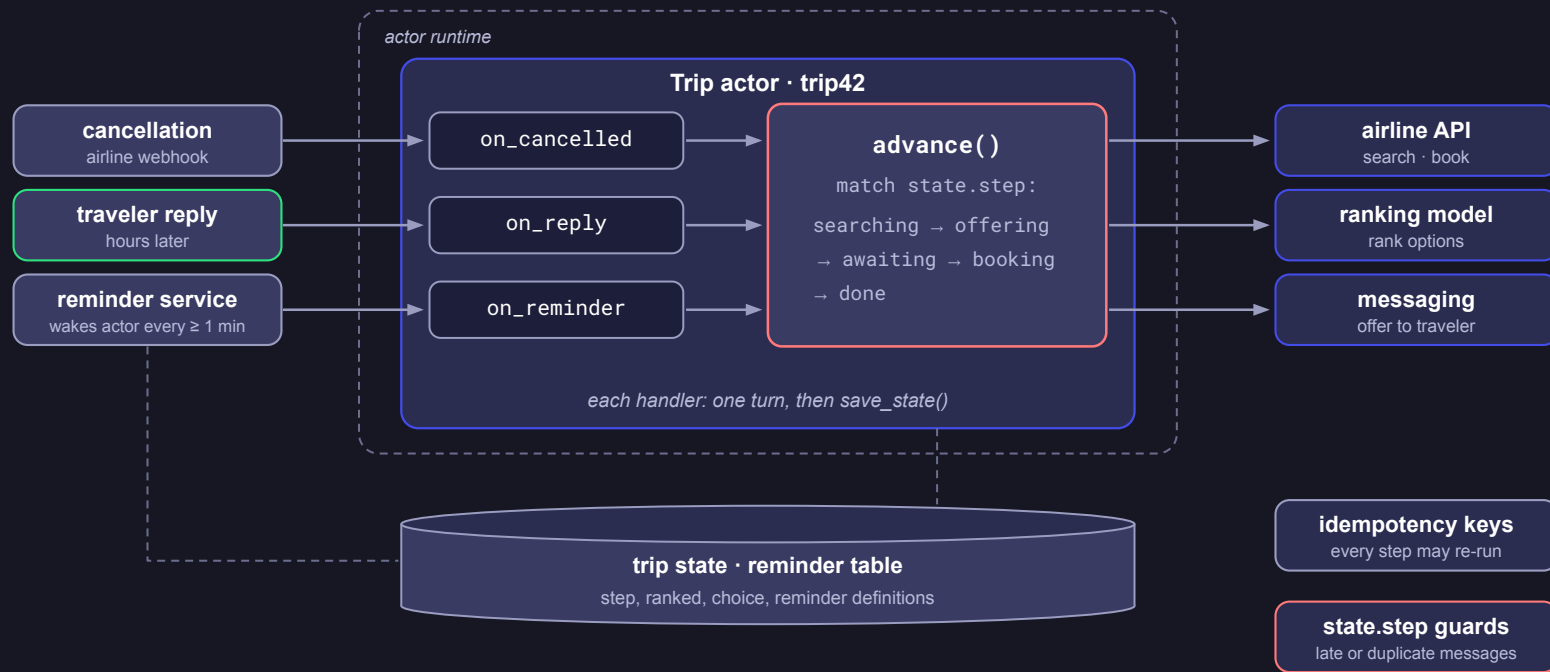
- Guarantees all-or-nothing but generally not exactly-once
- Need to operate across systems with different owners
- Two-phase commit creates unacceptable dependencies
- Web services are the dominant integration pattern
- Gaps remain between the systems

Event-driven architecture can work but is complicated



program counter = which queue holds the message
local variables = the message body plus database rows
call stack = gone

Actor systems may need application-level state machines



program counter = state.step

local variables = fields saved before each turn ends

call stack = gone; a reminder re-enters advance()

See: Bernstein et al. 2014

Wish to get reliability from a module

```
def rebook(trip):  
    for attempt in range(5):  
        options = airline.search(trip)  
        break  
    except Timeout:  
        sleep(backoff(attempt))  
    else:  
        raise GiveUp("search unavailable")  
    with circuit_breaker():  
        ranked = model.rank(options)  
    save_progress(trip, ranked)  
    send_choices(trip, ranked)  
    ...  
    choice = receive_choice(trip)  
    ...  
    book(trip, choice)  
    # also need sweeper job for recovery
```

```
@reliably  
def rebook(trip):  
    options = airline.search(trip)  
    ranked = model.rank(options)  
    choice = offer_choices(trip, ranked)  
    book(trip, choice)
```

How to keep the benefits of structured programming?

- Readable control flow
- Local reasoning
- Scoped error handling
- Easier changes
- Fewer bugs
- Less code

Callback Hell

What is “callback hell”?

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

```
fs.readdir(source, function (err, files) {  
  if (err) {  
    console.log('Error finding files: ' + err)  
  } else {  
    files.forEach(function (filename, fileIndex) {  
      console.log(filename)  
      gm(source + filename).size(function (err, values) {  
        if (err) {  
          console.log('Error identifying file size: ' + err)  
        } else {  
          console.log(filename + ' : ' + values)  
          aspect = (values.width / values.height)  
          widths.forEach(function (width, widthIndex) {  
            height = Math.round(width / aspect)  
            console.log(filename + ' : ' + height + ' : ' + width)  
          })  
        }  
      })  
    })  
  }  
})
```

<https://callbackhell.com/>

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** state-

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while B repeat A** or **repeat A until B**). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as “induction”

The end-to-end principle (from networking) tells us that recovery generally requires application knowledge

Saltzer, Reed, and Clark, 1984

- Careful file transfer: Not enough to compose reliable network communication and disk reads/writes—need to know we are transferring files
- Airline reservations: Making unconfirmed reservations durable is not vital and does not alter retry scenarios

*Lower-layer reliability can improve performance,
but cannot ensure correctness*

Durable execution from first principles

- The reliability module must control the “ends”
 - Envelope around the language runtime
 - Once system acknowledges a *start* command, clients know it will keep working until the code finishes
-

- Two ways to recover state
 - Reload saved state: annotate functions to be *durably memoized*
 - Recompute from input: annotate functions to be *recomputed*

Helpful symmetry: memoized results == input to recomputation

Temporal code excerpt

Activities do the I/O

retried on failure, with the same idempotency key

```
@activity.defn
async def hold_seat(req: HoldRequest) -> Hold:
    return services.airline.hold(
        req.option, req.traveler_id, req.idempotency_key
    )

@activity.defn
async def release_hold(hold_id: str) -> None:
    services.airline.release(hold_id)

...

@activity.defn
async def issue_ticket(req: TicketRequest) -> Ticket:
    ...
    try:
        return services.airline.ticket(
            req.hold_id, req.idempotency_key
        )
    except RuntimeError as e:
        raise ApplicationError(
            str(e), type="TicketingRejected", non_retryable=True
        )
```

Idempotency keys from the workflow ID

the same on every retry and replay

Compensation is application code

a declined charge releases the hold

Each attempt gets a timeout

failed attempts retry with backoff

Workflow calls activities directly

replayed from the event history after a crash

```
@workflow.defn
class RebookingWorkflow:
    ...

    @workflow.run
    async def run(self, inp: RebookingInput) -> RebookingResult:
        ...
        intent = workflow.info().workflow_id
        ...
        hold = await workflow.execute_activity(
            hold_seat,
            HoldRequest(chosen, trip.traveler_id, f"{intent}:hold"),
            start_to_close_timeout=TIMEOUT,
        )
        try:
            charge = await workflow.execute_activity(
                charge_fare_difference,
                ChargeRequest(
                    trip.card_token, hold.fare_difference_cents, f"{intent}:charge"
                ),
                start_to_close_timeout=TIMEOUT,
            )
        except ActivityError:
            await workflow.execute_activity(
                release_hold, hold.hold_id, start_to_close_timeout=TIMEOUT
            )
        ...
        try:
            ticket = await workflow.execute_activity(
                issue_ticket,
                TicketRequest(hold.hold_id, f"{intent}:ticket"),
                start_to_close_timeout=TIMEOUT,
            )
        ...
```



Temporal event history excerpt

68 events, decoded and abridged

```
1 WorkflowExecutionStarted
  input: {"trip_id":"TRIP-7", ...}
... 2-16 15 events
17 ActivityTaskScheduled      rank_options
18 ActivityTaskStarted        attempt: 1
19 ActivityTaskCompleted
  result: [{"option_id":"opt-2","flight":"AA 2201", ...}, ...]
... 20-43 24 events
44 ActivityTaskScheduled      hold_seat
45 ActivityTaskStarted        attempt: 1
46 ActivityTaskCompleted
  result: {"hold_id":"HLD0001", ...}
... 47-49 workflow task
50 ActivityTaskScheduled      charge_fare_difference
51 ActivityTaskStarted        attempt: 1
52 ActivityTaskCompleted
... 53-55 workflow task
56 ActivityTaskScheduled      issue_ticket
  input: {"idempotency_key":"rebook-TRIP-7-reply-lost-49bf47:ticket", ...}
57 ActivityTaskStarted        attempt: 3
  lastFailure: "connection reset after ticket 016-0000000002 committed"
58 ActivityTaskCompleted
  result: {"ticket_number":"016-0000000002", ...}
... 59-67 9 events
68 WorkflowExecutionCompleted
  result: {"outcome":"ticketed","ticket_number":"016-0000000002", ...}
```

Ranking output, recorded

replay returns it; the ranking model is not called again

Hold and charge, recorded

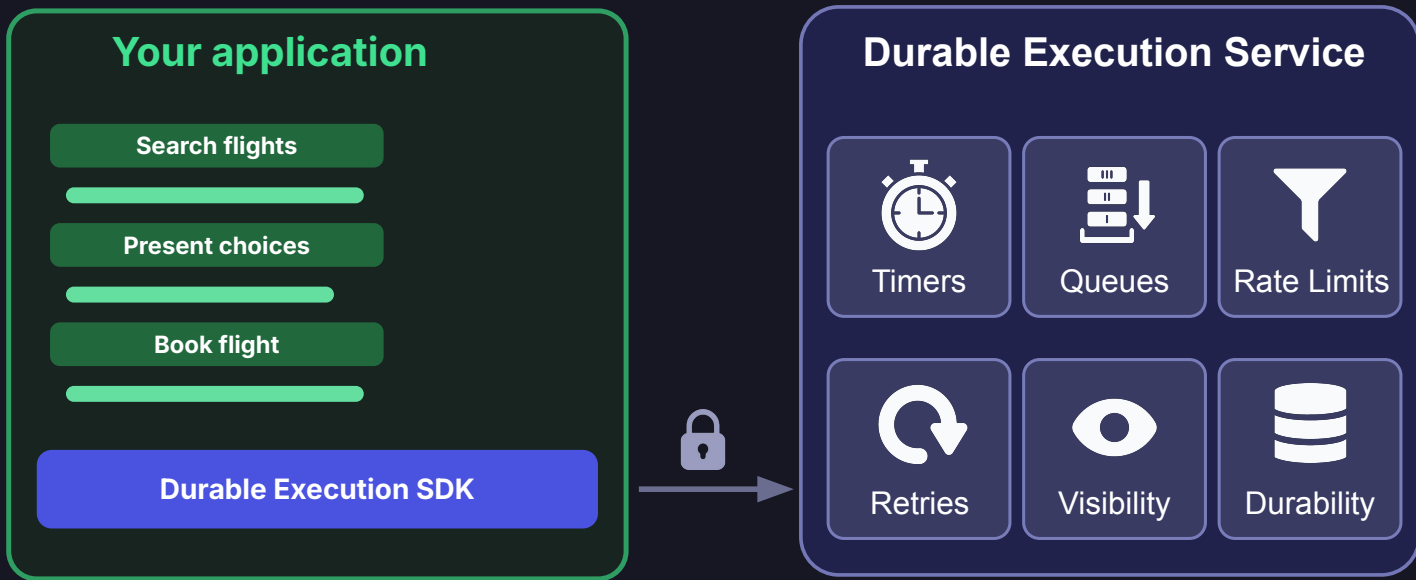
after a crash, replay returns these results

Ticket committed, reply lost

attempt 3 reuses the same key; one ticket is issued



Durable execution separates code for application concerns from system concerns

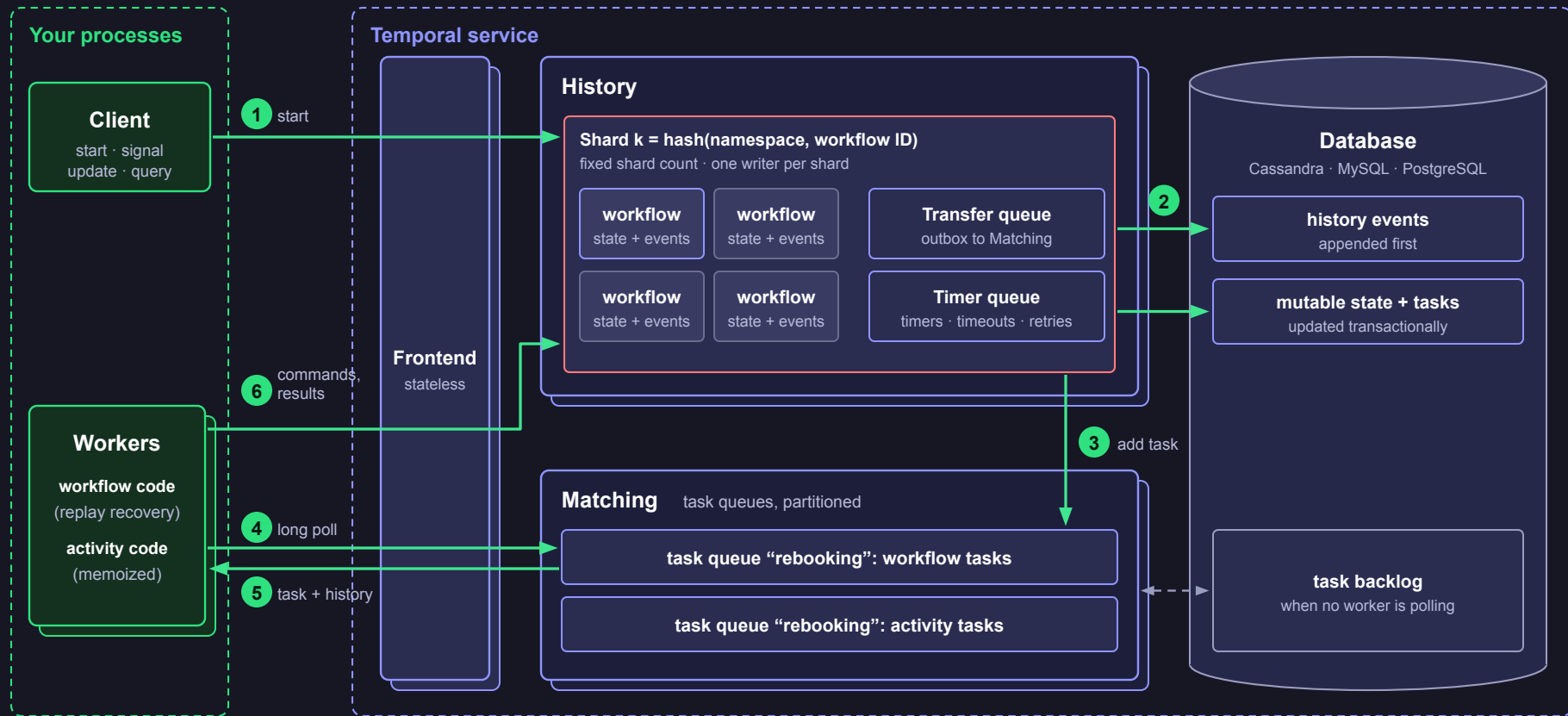


Business Logic



System Logic

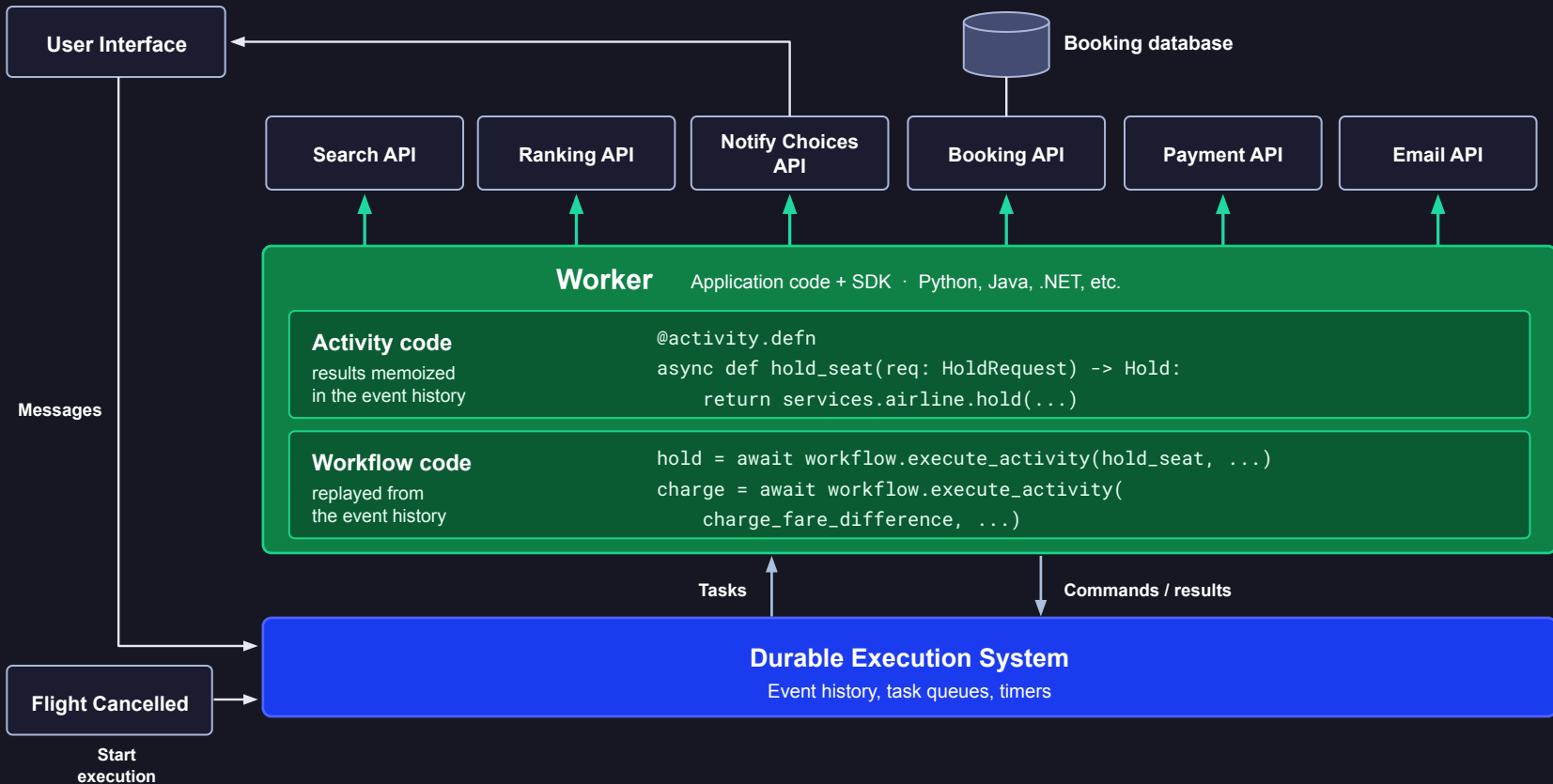
Inside the Temporal service



Additional details of Temporal

- SDKs integrate tightly with the language and core libraries
- Need to ensure deterministic replay for
 - Timestamps
 - Random numbers
 - Concurrent programs
- Send messages to running programs for flexible interaction patterns
- Provides flow control to avoid retry storms, ensure priority and fairness
- Long-running programs require support for upgrades while executing
- May need to truncate replay history to prevent it from becoming too long
- Open protocol for durable execution across administrative domains (Nexus)

Flight rebooking with durable execution



Overview

- Durable execution from first principles
- Why is it taking off now?

Durable execution growing with many applications

- Agents: model calls, tools, world interactions, long-lived agents
- Infrastructure: cloud operations
- Commerce: rideshare, food orders, travel bookings
- Media pipelines: video transcoding
- Finance: money movement
- *Many others*

Central ideas developed over decades

RECOVERY AND REPLAY

1983 · Borg *et al.*

Auragen

Saved input messages

1985 · Strom & Yemini

Transparent recovery

Piecewise determinism

TRANSACTIONS AND WORKFLOWS

1987 · Garcia-Molina & Salem

Sagas

Steps and compensation

1990 · Bernstein *et al.*

Recoverable queues

Durable requests

APPLICATION RECOVERY

1997 · Lomet

Phoenix at HPTS

Replay application code

2002–04 · Barga *et al.*

Interaction contracts

Recovery across components

Industrial lineage of durable execution

AMAZON · UBER · TEMPORAL

2004–06

Amazon SQS

Durable messaging

2012

AWS SWF + Flow

History and replay

2017

Uber Cadence

Open-source engine

2019

Temporal

Fork of Cadence

MICROSOFT

2007 F# · 2012 C#

async/await

Sequential async code

2013

Durable Task Framework

SWF ideas in C#

2017–18

Durable Functions

Serverless orchestrations

MICROSOFT RESEARCH

1997–2004

Phoenix

Recovery by replay

2018

Ambrosia

Virtual resiliency



Work by Temporal's founders: Maxim Fateev and Samar Abbas

Why durable execution is taking off now

- Supports ordinary languages, programming style, and software development processes
 - Embraces the heterogeneous estate
 - Broad language support
 - Cloud agnostic
- Various backing stores
- Various deployment models: open source, managed services
- Multiple vendors
- AI applications
 - GPUs are expensive and often oversubscribed
 - Agents are long-running and interact with the world



Durable Execution: A new paradigm from old ideas

- Complementary to durable state
- Fills the gaps between independent systems
- Ergonomic breakthrough via language integration

Johann Schleier-Smith

✕ @jssmith

References & Related Reading

- Adya et al. 2002** Cooperative Task Management without Manual Stack Management. USENIX ATC.
- Barga & Lomet 1999** Phoenix: Making Applications Robust. HPTS; SIGMOD demo.
- Barga et al. 2002** Recovery Guarantees for General Multi-Tier Applications. ICDE.
- Barga et al. 2004** Recovery Guarantees for Internet Applications. ACM TOIT 4(3).
- Bernstein et al. 1990** Implementing Recoverable Requests Using Queues. SIGMOD.
- Bernstein et al. 2014** Orleans: Distributed Virtual Actors for Programmability and Scalability. MSR-TR-2014-41.
- Borg et al. 1983** A Message System Supporting Fault Tolerance. SOSP.
- Burckhardt et al. 2021** Durable Functions: Semantics for Stateful Serverless. PACMPL 5(OOPSLA).
- Dijkstra 1968** Go To Statement Considered Harmful. CACM 11(3).
- Elnozahy et al. 2002** A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys 34(3).
- Garcia-Molina & Salem 1987** Sagas. SIGMOD.
- Goldstein et al. 2020** A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. PVLDB 13(5).
- Gray 1985** Why Do Computers Stop and What Can Be Done About It? Tandem TR 85.7.
- Gunawi et al. 2016** Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. SoCC.
- Helland 2007** Life beyond Distributed Transactions: an Apostate's Opinion. CIDR.
- Laigner et al. 2021** Data Management in Microservices: State of the Practice, Challenges, and Research Directions. PVLDB 14(13).
- Li et al. 2026** Distributed Speculative Execution for Resilient Cloud Applications. OSDI.
- Lomet 1997** Application Recovery: Advances Toward an Elusive Goal. HPTS.
- Qi et al. 2023** Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. SOSP.
- Saltzer et al. 1984** End-to-End Arguments in System Design. ACM TOCS 2(4).
- Stoica et al. 2024** If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems. SOSP.
- Strom & Yemini 1985** Optimistic Recovery in Distributed Systems. ACM TOCS 3(3).
- Waldo et al. 1994** A Note on Distributed Computing. Sun Microsystems Laboratories SMLI TR-94-29.
- Wang 2021** Designing a Workflow Engine from First Principles. temporal.io/blog/workflow-engine-principles.
- Yuan et al. 2014** Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. OSDI.

