TigerBeetle

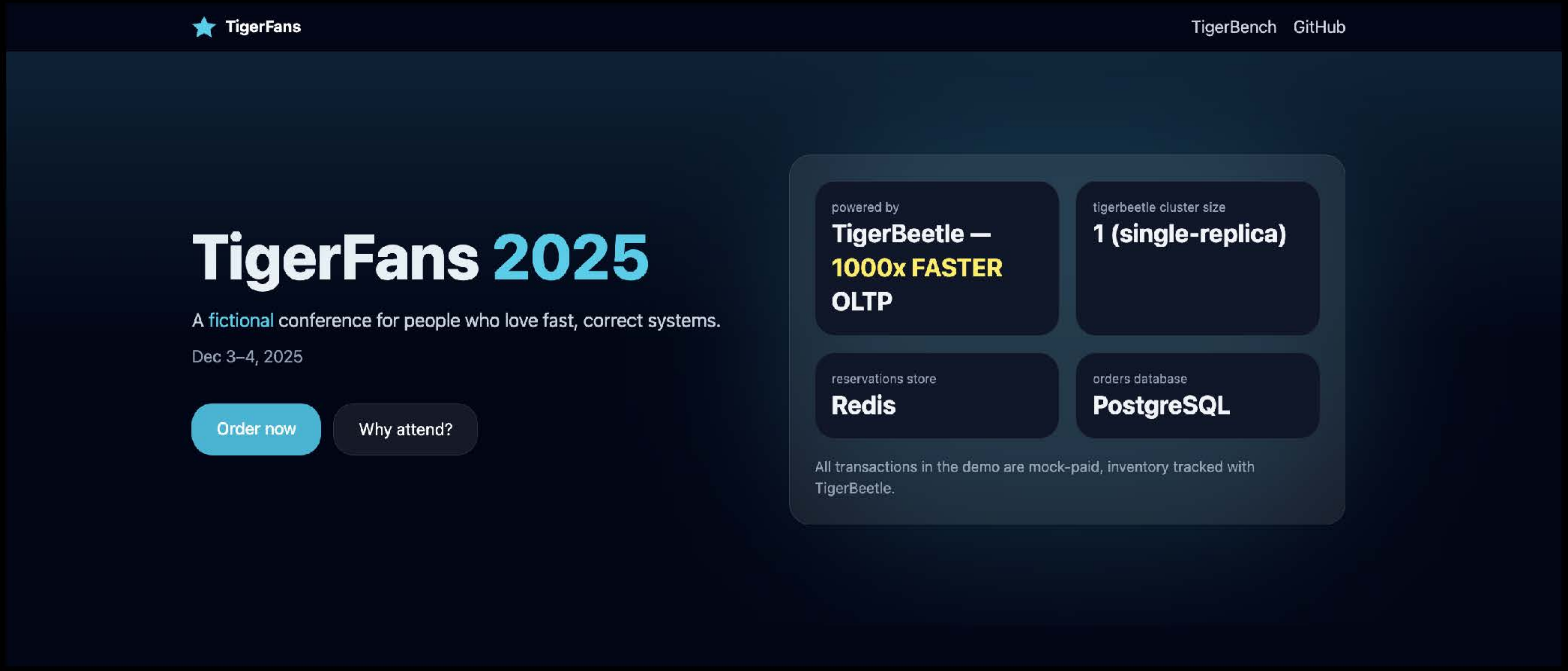
1000X

WORLD TOUR

13 CITIES | 6 DAYS

TigerFans: High-Performance Ticketing with TigerBeetle

From Double-Entry Accounting to 977 Tickets per Second



@renerocksai

2025-12-04

What Started TigerFans

It began with a tweet.

"How would you sell Oasis-scale tickets without overselling?"

Answer from Joran: Too easy: TigerBeetle.

I wanted to know:

- **HOW** do you actually model ticketing as **double-entry accounting**?
- What does the **real code** look like?
- How do you handle **timeouts, webhooks, idempotency?**

So I decided to **build the whole thing.**

Goal: Realistic, Not Toy

I didn't want a toy example.

I wanted a **realistic ticketing flow**

- Checkout with multiple ticket classes & goodies
- Pending reservations with timeouts
- Webhook-based payment confirmation
- Strict no overselling guarantee

Stack:

- **FastAPI** (Python, async)
- **TigerBeetle** (accounting)
- **SQLite** at first
- **MockPay** (fake payment provider)

Tickets as Financial Transactions

TigerBeetle gives us accounts, transfers, debits, credits.

We turn ticketing into an accounting problem:

- Every reservation is a **transfer**
- Every ticket class is a set of **accounts**
- **Debits = Credits** => system never goes out of balance

Double-entry accounting gives us:

- Built-in error detection
- Auditability for free
- And with TB: durability + performance in the same model

Three Accounts per Ticket Class

For each ticket class:

- **Operator:**
  Holds all inventory
- **Budget:**
  What we’re allowed to sell
- **Spent:**
  Tickets actually reserved/used

Flow:

1. Fund Budget from Operator
2. Move from Budget Spent on reservation
3. Post or void based on payment outcome

```
INIT FLOW: Fund the budget
==========================

+----------+              +----------+              +----------+
|          |  --credit--> |          |              |          |
| Operator |              |  Budget  |              |  Spent   |
|          |  <--debit--  |          |              |          |
+----------+              +----------+              +----------+
```

```
HOLDING FLOW: Limited Hold on the resource
==========================================

Tickets:
     +----------+    ---------credit--------->    +----------+
     |          |                                 |          |
     |  Budget  |    flags=PENDING, timeout=5min  |  Spent   |
     |          |                                 |          |
     +----------+    <---------debit-----------   +----------+
```

```
COMMIT FLOW: Post the pending transfers
=======================================

Tickets:
     +----------+    ---------credit--------->    +----------+
     |          |                                 |          |
     |  Budget  |    flags=POST_PENDING_TRANSFER  |  Spent   |
     |          |                                 |          |
     +----------+    <---------debit-----------   +----------+
```

Pending Transfers & Timeouts

Checkout = create pending transfer:

- Budget -> Spent
- 5-minute timeout
- DEBITS MUST NOT EXCEED CREDITS flag on accounts

Payment succeeds:

- Post the pending transfer => ticket is locked in

Payment fails / user disappears:

- Void the transfer => it just vanishes

No cron jobs. No cleanup races.

Correctness is enforced by TigerBeetle's invariants.

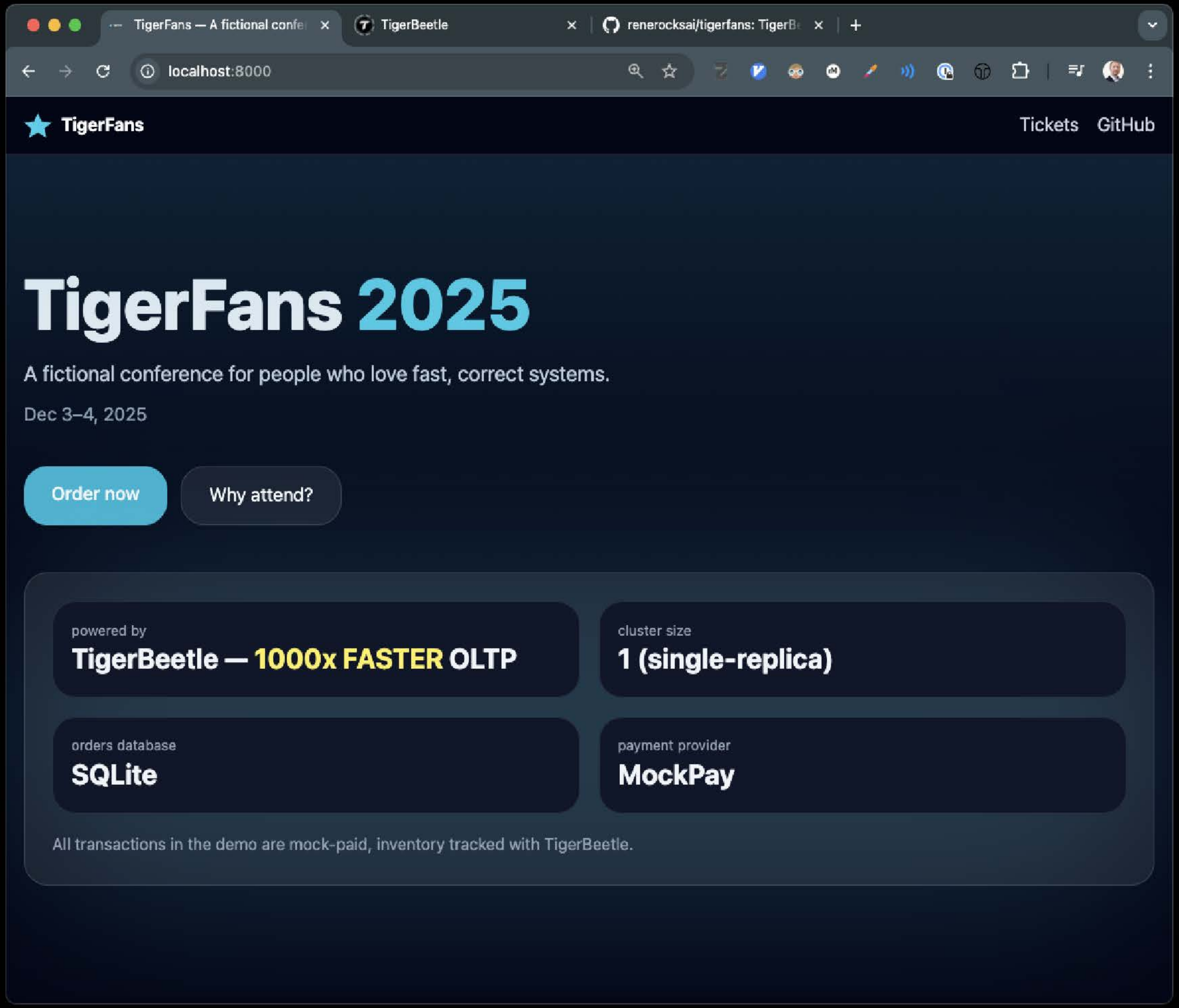
Initial Architecture: Simple & Clean

Version 1:

- FastAPI (1 worker)
- SQLite
- TigerBeetle (dev mode)
- MockPay

Everything worked:

- 2-phase checkout
- Timeouts
- Webhooks
- UI + Admin view



Baseline Performance

First measurement:

- ~115 tickets/sec

Oasis baseline:

- ~1.4M tickets / 6 hours
- ≈ 65 tickets/sec

So we're at about O(1.7 Oasis).

I send this to Joran…

Baseline Performance

First measurement:

- ~115 tickets/sec

Oasis baseline:

- ~1.4M tickets / 6 hours
- ≈ 65 tickets/sec

So we're at about O(1.7 Oasis).

I send this to Joran…

…he replies:

"I was surprised the TPS is so low.

It should be ~10k."

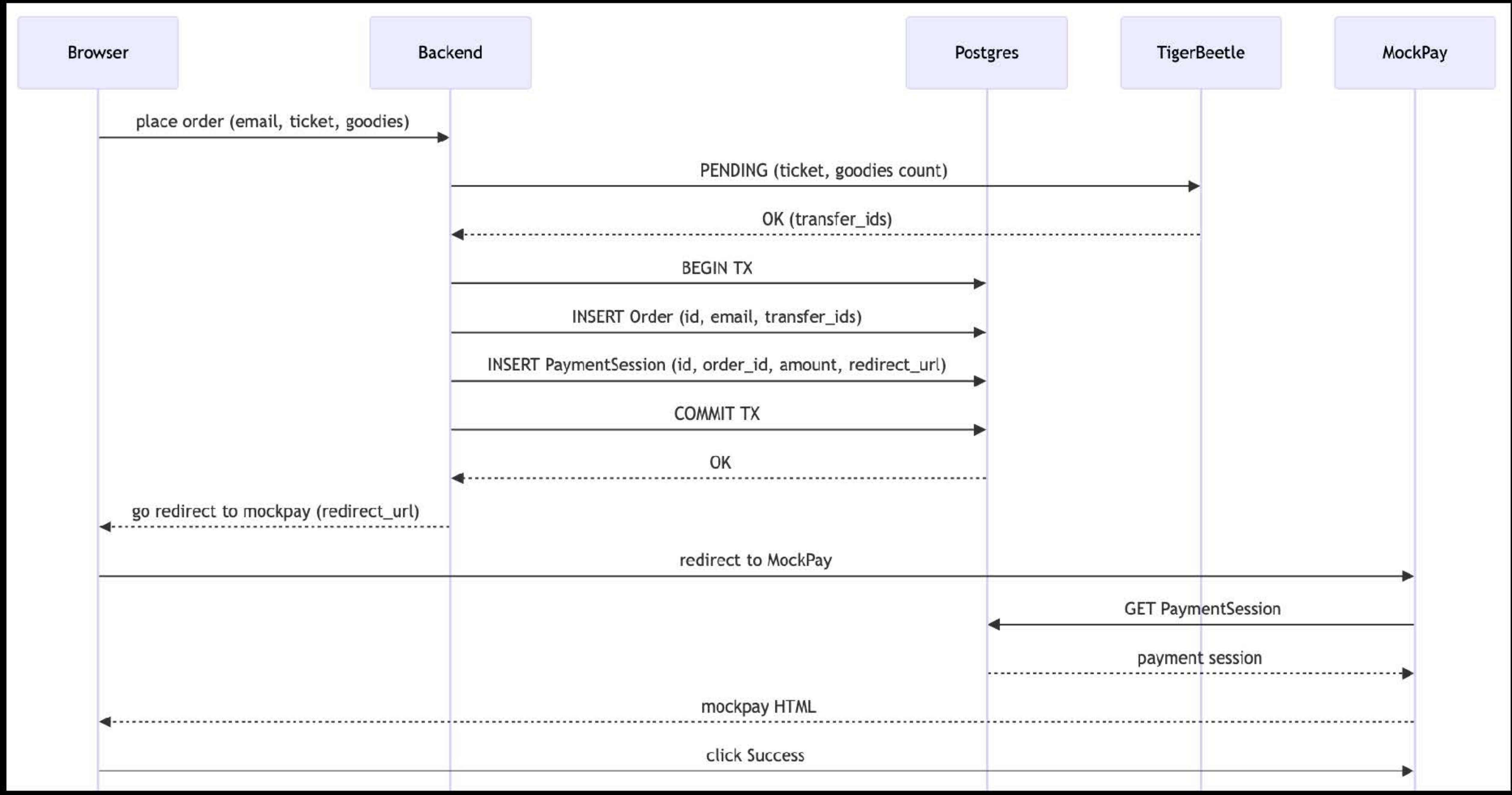
Where Is the Time Going?

Checkout flow:

1. Browser -> Server: *place order*
2. Server -> TB: PENDING transfers
3. Server -> DB: sessions + orders
4. Server -> Browser: redirect

Webhook flow:

1. MockPay -> Server
2. Server -> DB: idempotency checks
3. Server -> TB: POST transfers
4. Server -> DB: final order write



Bottleneck #1: PostgreSQL Everywhere

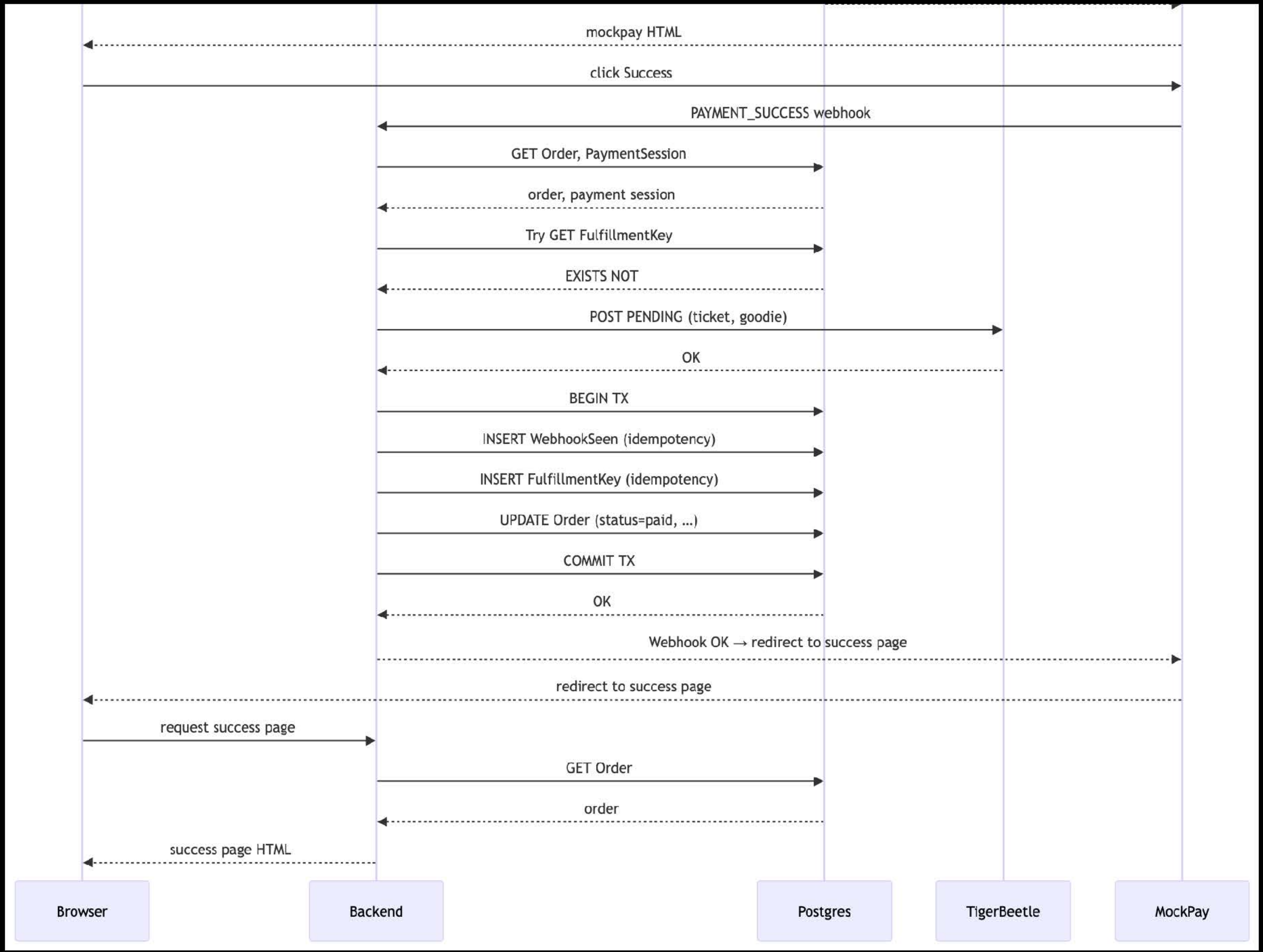
With PostgreSQL in the hot path:

- Reservations: ~150 ops/s
- Webhooks: ~130 ops/s

Every request:

- 2–4 round-trips to PG
- Multiple fsyncs
- Contention on the same tables

The database, not TigerBeetle, is the bottleneck.



Redis Experiment: All-In Memory

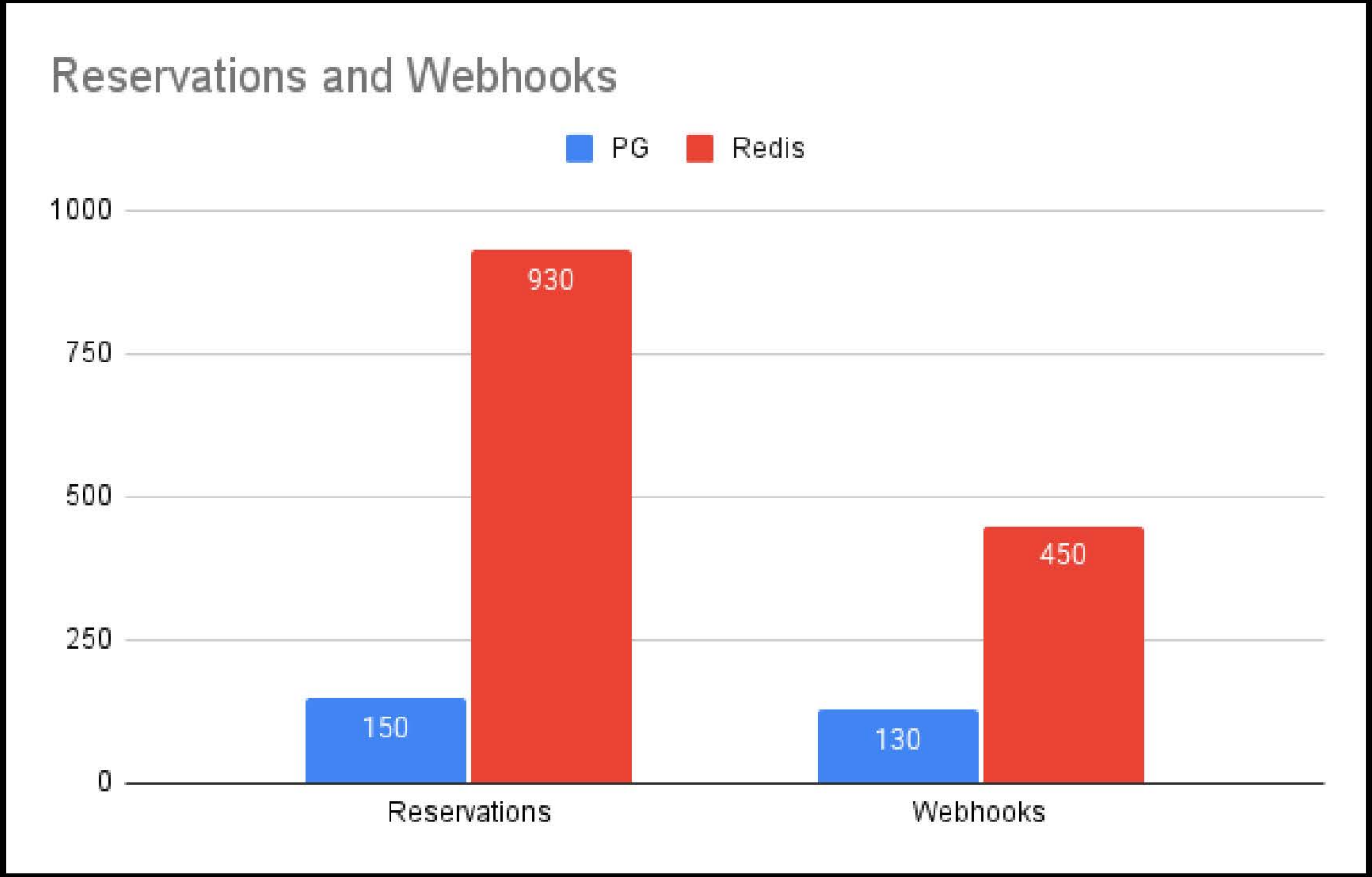
I swap PostgreSQL -> Redis for everything :

- Sessions
- Idempotency
- Orders

Results:

- Reservations: ~930 ops/s
- Webhooks: ~450 ops/s

Great… except Redis everysec can lose 1s of orders on crash.
That's not acceptable.



Rafael’s Hot/Cold Path Insight

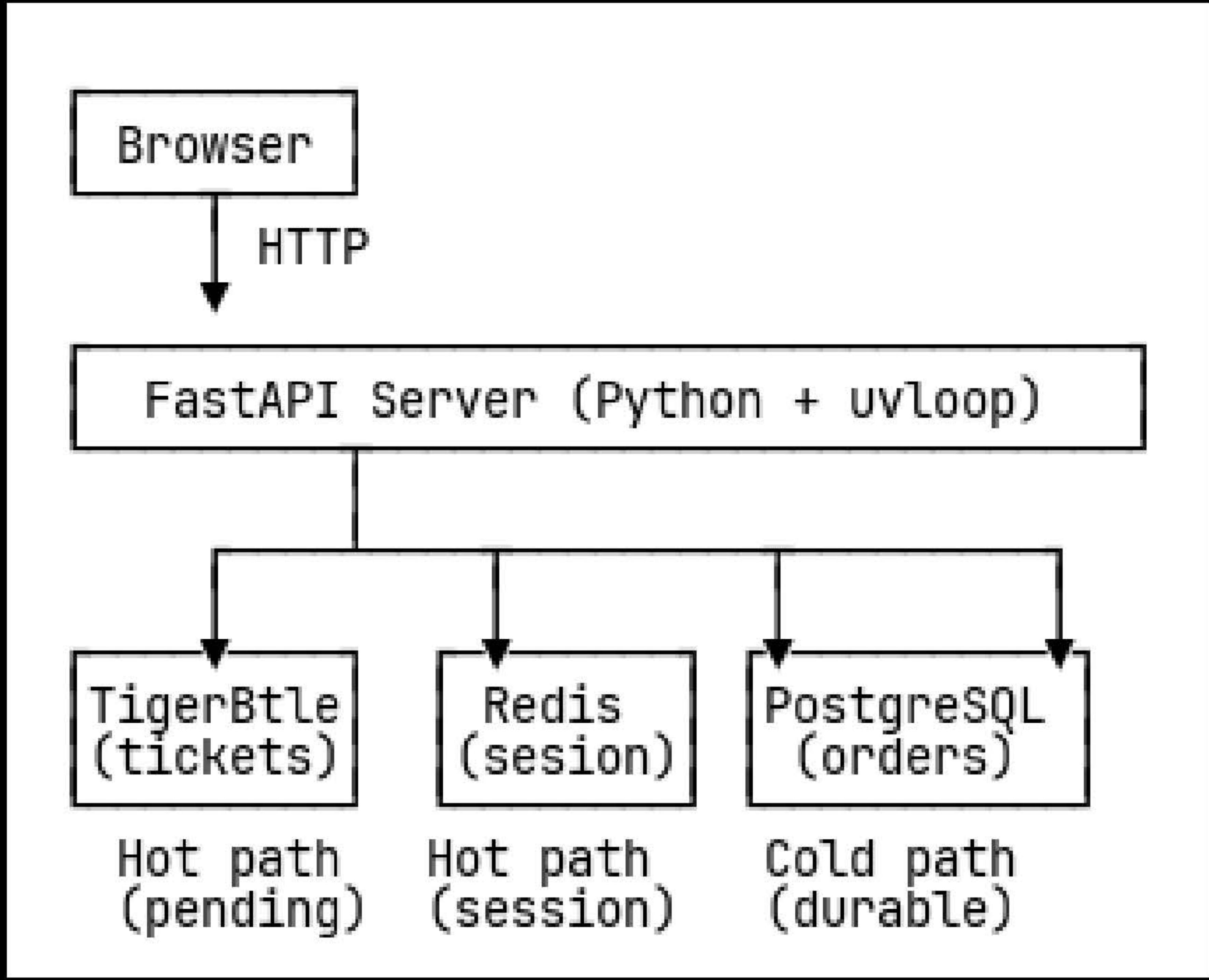
Rafael’s suggestion:

- Use Redis for hot data:
  - Payment sessions
  - Idempotency keys

- Use PostgreSQL only for cold data
  - Completed orders

TigerBeetle:

- Always handles ticket accounting
- Always fully durable



Three Configuration Levels

Level 1 – PG Only

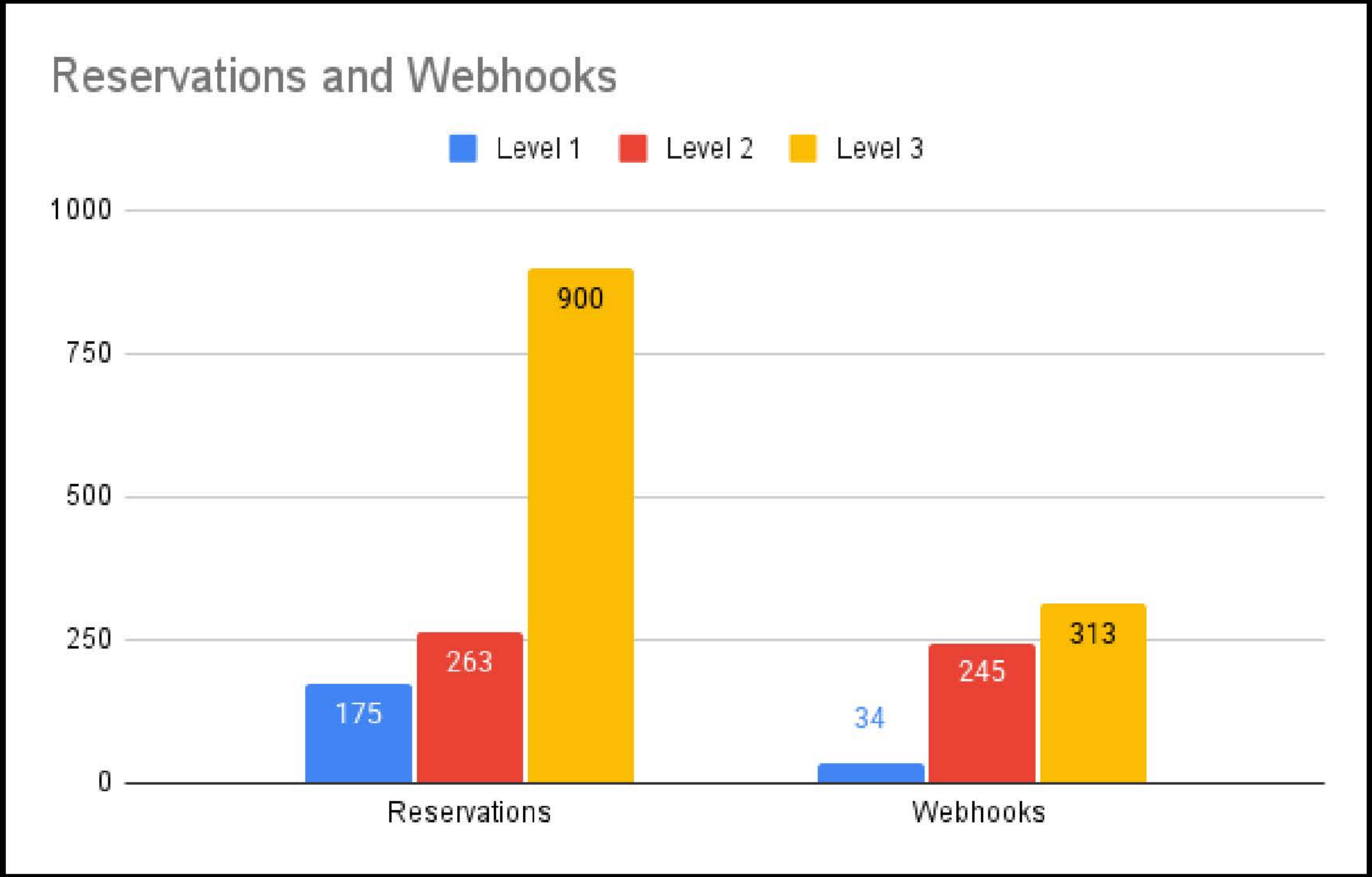
- PG for sessions, accounting, orders

Level 2 – PG + Redis

- Redis: sessions
- PG: accounting + orders

Level 3 – TB + Redis

- Redis: sessions
- TigerBeetle: accounting
- PG: orders



Interface Impedance: Requests vs Batches

TigerBeetle loves **big batches**:

- Up to **8,190 transfers** per call
- One network round-trip amortized

FastAPI loves **per-request awaits**:

- Each request calls **create_transfers()** once
- No chance to "wait a bit and batch"

Instrumentation showed:

- **Batch size ≈ 1**
- We were flying a 747 with 1 passenger per flight.

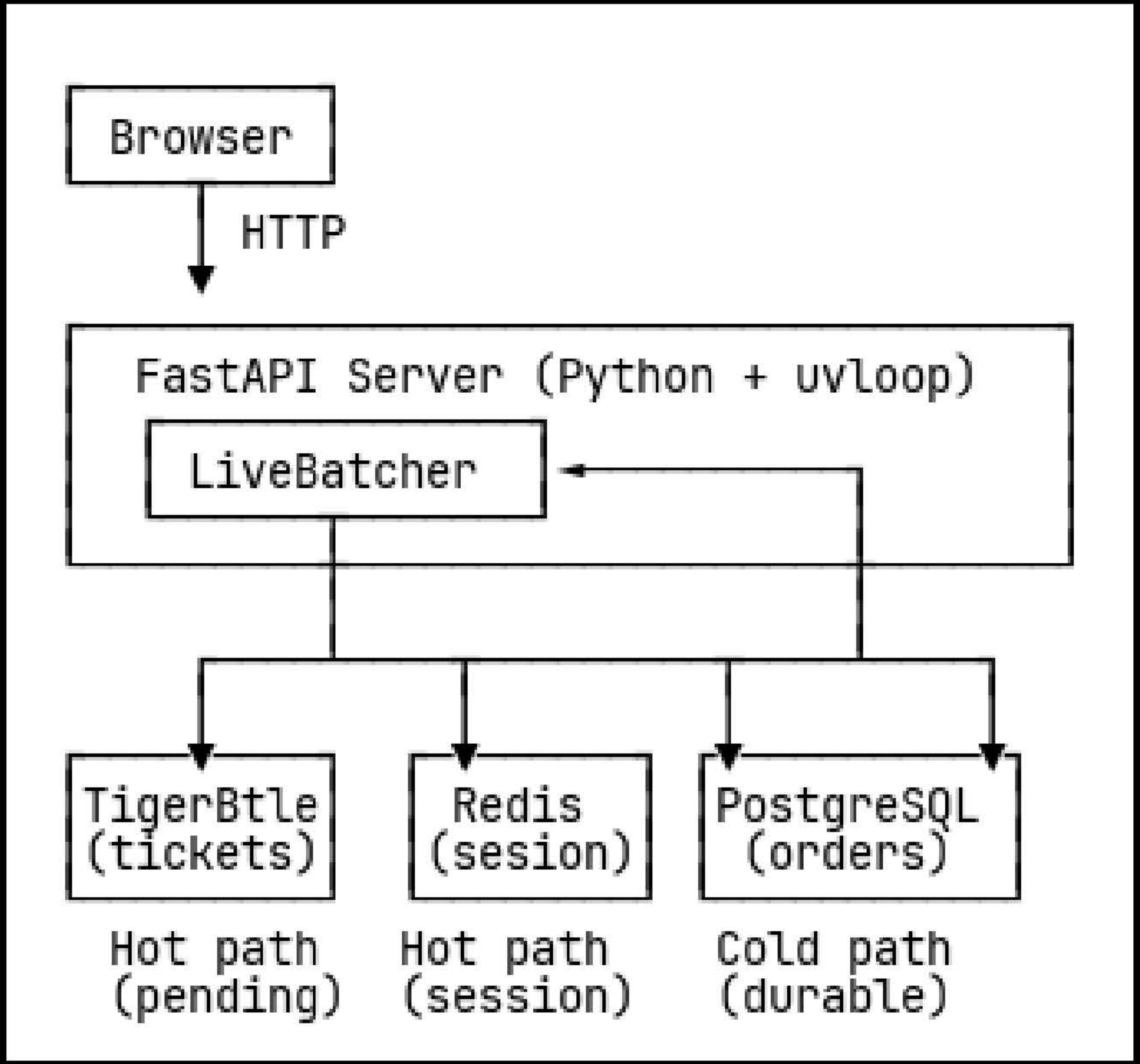
LiveBatcher: Continuous Auto-Batching

LiveBatcher:

- Sits between app and TB client
- Collects concurrent requests
- Packs them into batches while previous batch is in flight
- Maps TB errors back to each caller

Behavior:

- Under load: batch sizes 5–6 transfers
- Low load: small batches, low latency



Batching Results

With hot/cold path + LiveBatcher:

- Reservations: **977 ops/s**
- Average TB **batch size: 5–6**

Compared to baseline:

- PG-only: ~150 ops/s
- TB+Redis (before batching): ~900 ops/s

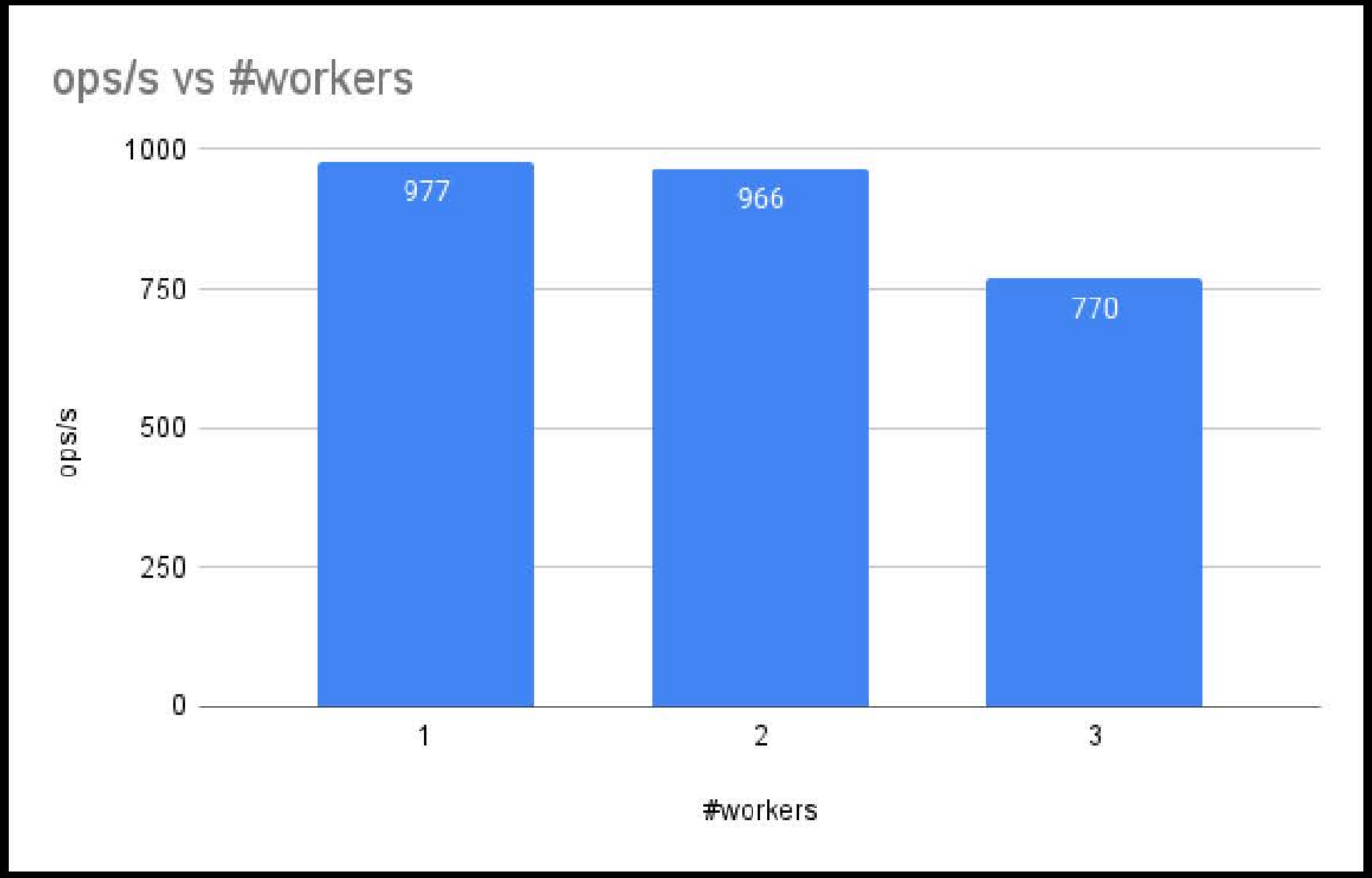
LiveBatcher gave us the final 8% — the earlier wins came from **architecture**, not micro-optimizations.

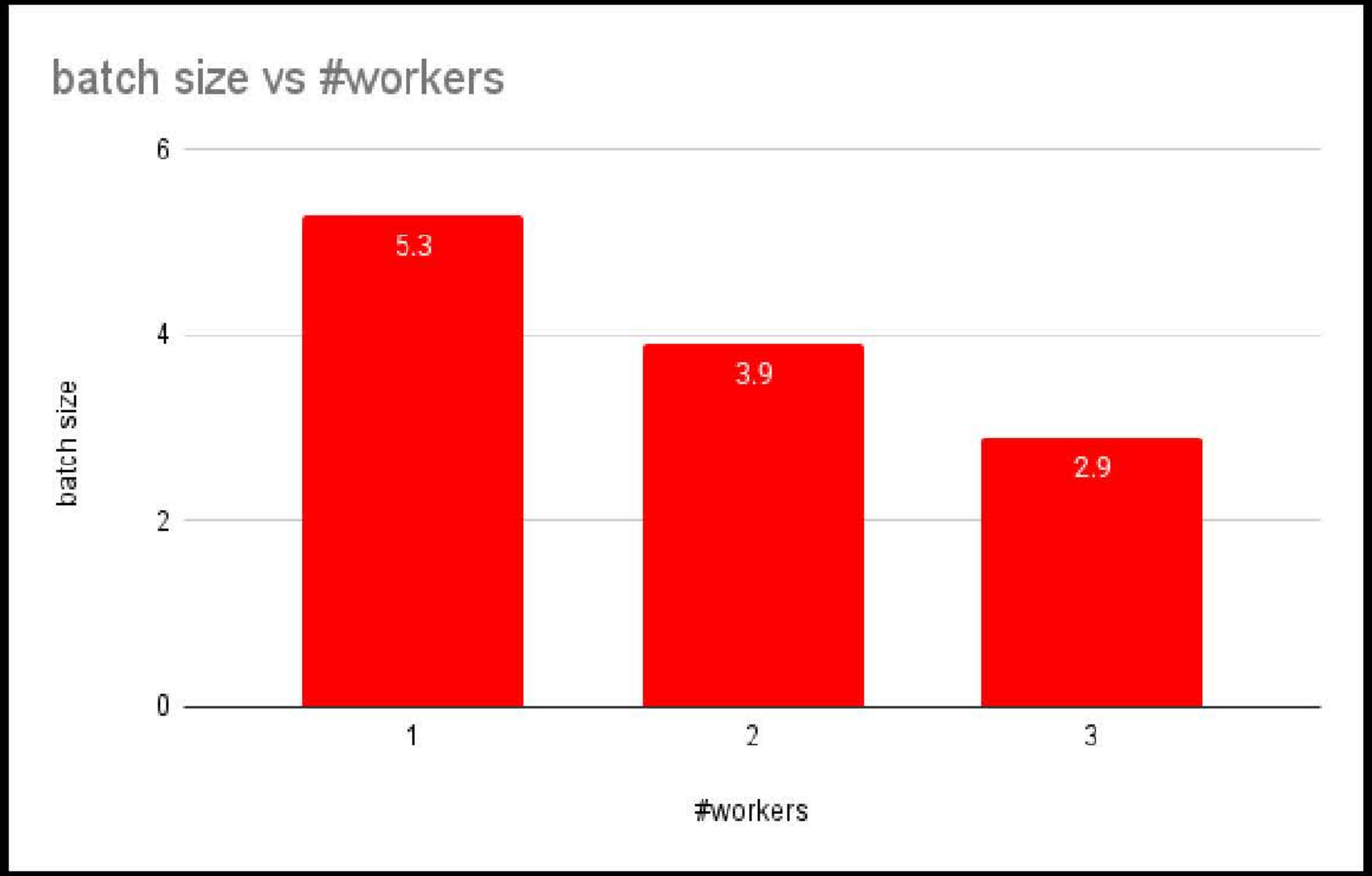
The Single-Worker Paradox

1 worker
- 977 ops/s
- Avg batch size: 5.3

2 workers
- 966 ops/s
- Avg batch size: 3.9

3 workers
- 770 ops/s
- Avg batch size: 2.9





More workers -> smaller batches -> lower throughput.

Final Performance Snapshot

End result (Python, 1 worker):

- **977 reservations / second**
- **≈ 15× Oasis baseline**
- **TB batch size: 5–6 transfers**

The real limit now isn't TigerBeetle — it's **Python's event loop overhead.**

Amdahl's Law: Why ~1k TPS Makes Sense

In checkout:

- We sped up **accounting** (PG TB) by ≈3×
- Moved sessions to Redis
- But every request still spends:
  - ~5ms in Python/FastAPI
  - Routing, JSON, business logic

Amdahl's Law says:

- That serial 5ms chunk becomes a hard ceiling
- Even with "infinite-speed" TB, Python caps us

What Amdahl Suggests Next

If we keep the same **architecture**:

- Same double-entry model
- Same hot/cold split
- Same LiveBatcher idea

...but move to:

- Go, Zig, ...

Then:

- **Python's 5ms** -> maybe **0.1 – 0.5ms**
- Batch sizes can grow
- Amdahl's Law says:
- **10–50× more throughput** is realistic

TigerBeetle Ticket Challenge (TTC)

CHALLENGE:

- Same patterns: <-- or do it your way :-)
  - Double-entry resource modeling
  - Hot/cold path
  - Auto-batching
  - Single-worker friendly design
- Any language, any stack <-- … any platform? Raspberry Pi?
- Benchmark it, share results <-- please :-)

Resources:

- tigerfans.io (demo + TigerBench)
- github.com/renerocksai/tigerfans

DEMO DEMO DEMO

1. Ticket Booking

- Start a checkout
- See pending reservation appear

2. MockPay

- Complete payment
- Webhook posts TB transfers

3. Admin Live View

- Orders, reservations, timeouts

4. TigerBench

- Compare PG-only vs PG+Redis vs TB+Redis
- Ops vs. Transactions

TigerFans: High-Performance Ticketing with TigerBeetle

From Double-Entry Accounting to 977 Tickets per Second

THANK YOU for your attention!

[Backup] Amdahl's Law: Component Breakdown

Baseline checkout (Level 1 – PG Only):

- Total time ≈ 15.4 ms
- PG accounting: 14.03 ms (~91%)
- PG sessions: 1.36 ms (~9%)

Optimized checkout (Level 3 – TB + Redis):

- Total time ≈ 5.63 ms
- TB accounting: 4.58 ms
- Redis sessions: 1.04 ms

Amdahl's Law:

Speedup_overall = 1 / ((1 - P) + P / S)

Here:

- Accounting P ≈ 0.91, S ≈ 3.0
- Predicted speedup ≈ 2.73×
- Measured component speedup ≈ 2.74×

[Backup] Amdahl & the Batching Ceiling

For batching, Amdahl applies again:

- TigerBeetle batch time ≈ 3 ms
- Python per-request serial work ≈ 3 ms
- Routing, parsing, validation
- Business logic
- Preparing TB transfers

During one TB batch (~3 ms):

- Python can push ≈ 1 new request to the batcher

So batch size naturally stabilizes around:

- 2–6 transfers depending on concurrency load

In Go/Zig:

- Serial overhead ≈ 0.1 ms
- Same architecture => batches of 30–50+