TigerBeetle Goes GenAI

Do LLMs write better code with Accounting Primitives?

@renerocksai

2025-12-04

Experiment Overview

Core Question:

Can Large Language Models (LLMs) write better code for specialized databases (TigerBeetle) than generic ones (PostgreSQL)?

Hypothesis:

TigerBeetle's high-level **accounting primitives** act as "**guardrails**," enabling LLMs to generate robust, correct, and performant code more easily than SQL's flexible schema.

Setup: Process Payments Challenge

Challenge Task:

Implement a standard financial workload:

- **Input:** 1,000 Payments
- **Constraints:** Idempotent, Double-Entry, Safe, Efficient
- **Targets:** TigerBeetle (Python) vs. PostgreSQL (psycopg2)

Variables:

1. **Model Capability:** GPT-4o (Mid-Tier) vs. GPT-5.1 (Reasoner)
2. **Concurrency:** Single-Threaded vs. Multi-Threaded

How It Works: The Feedback Loop

Iterative Code Generation Process:

1. **Prompt:** Send task description + API docs to LLM
2. **Generate:** LLM writes Python code
3. **Execute:** Run code against fresh database (1,000 payments)
4. **Validate:** Check correctness + measure performance

If validation fails:

- Error feedback sent back to LLM
- Database reset to initial state (clean slate for next attempt)
- LLM attempts to fix the code
- Repeat up to 10 iterations

Success: Code passes validation OR reaches iteration limit

What We Measured

Correctness:

- **Balance preservation** (total money conserved)
- **Double-entry bookkeeping** (debits = credits)
- **No negative balances**

Performance:

- **Single-threaded** throughput (payments/second)
- **Multi-threaded** throughput (4 threads)

Code Quality (5 dimensions):

- Correctness
- Performance
- Safety
- Concurrency
- Code Quality
- **Claude Sonnet 4.5 be the judge!**

Ease of Generation:

- **Iterations needed** to achieve success

Task Prompts

```
You are tasked with implementing a payment processing system.

TASK:
Implement the function process_payments() that processes a batch of payments by creating transfers in the database.

CRITICAL: You must use the database API to create transfers that will modify account balances. The database contains accounts with existing balances, and you must use the API to move money between accounts. Simply returning PaymentResult objects is NOT sufficient - you must actually execute database operations to transfer money.

INTERFACE SPECIFICATION:
{interface_spec}

API DOCUMENTATION:
{api_docs}

DATABASE CONNECTION:
{connection_info}

INITIAL STATE:
The database has been initialized with {num_accounts} accounts. Each account has:
- An ID (integer from 1 to {num_accounts})
- A balance stored in the database
- ledger=1 (all accounts use the same ledger)
- code=1 (all accounts use the same code)

TEST DATA:
You will receive a list of {num_payments} payments to process. Each payment specifies:
- id: Unique payment identifier (use this as the transfer ID for idempotency)
- sender_id: Account to debit (subtract from their balance in the database)
- recipient_id: Account to credit (add to their balance in the database)
- amount: Amount to transfer (in cents)

REQUIREMENTS:
- Use the database API to create transfers that modify account balances
- For TigerBeetle: Create Transfer objects and call client.create_transfers()
- For PostgreSQL: Execute SQL statements to update account balances
- Process all payments using double-entry bookkeeping (debit sender, credit recipient)
- Handle the batch as efficiently as possible (use batching where available)
- Operations must be idempotent (use payment.id as the transfer/transaction ID)
- Must be safe under concurrent execution
- Return PaymentResult for each payment indicating success/failure

VALIDATION:
After your code runs, the test harness will check the actual database balances to verify correctness. Your code MUST modify the database - returning success without creating transfers will cause validation to fail.

Please provide complete, runnable Python code that implements the process_payments() function.
Only provide the code, no additional explanation.

```

initial prompt

```
Your code produced the following error:

ERROR:
{error_message}

STDOUT:
{stdout}

STDERR:
{stderr}

{validation_errors}
CRITICAL: The validation above checks the ACTUAL DATABASE BALANCES after your code executes.

DEBUGGING TIPS:

{debugging_tips}

Please fix the code and provide a corrected version.
Only provide the code, no additional explanation.
```

error feedback prompt

Finding #1: Reliability Gap

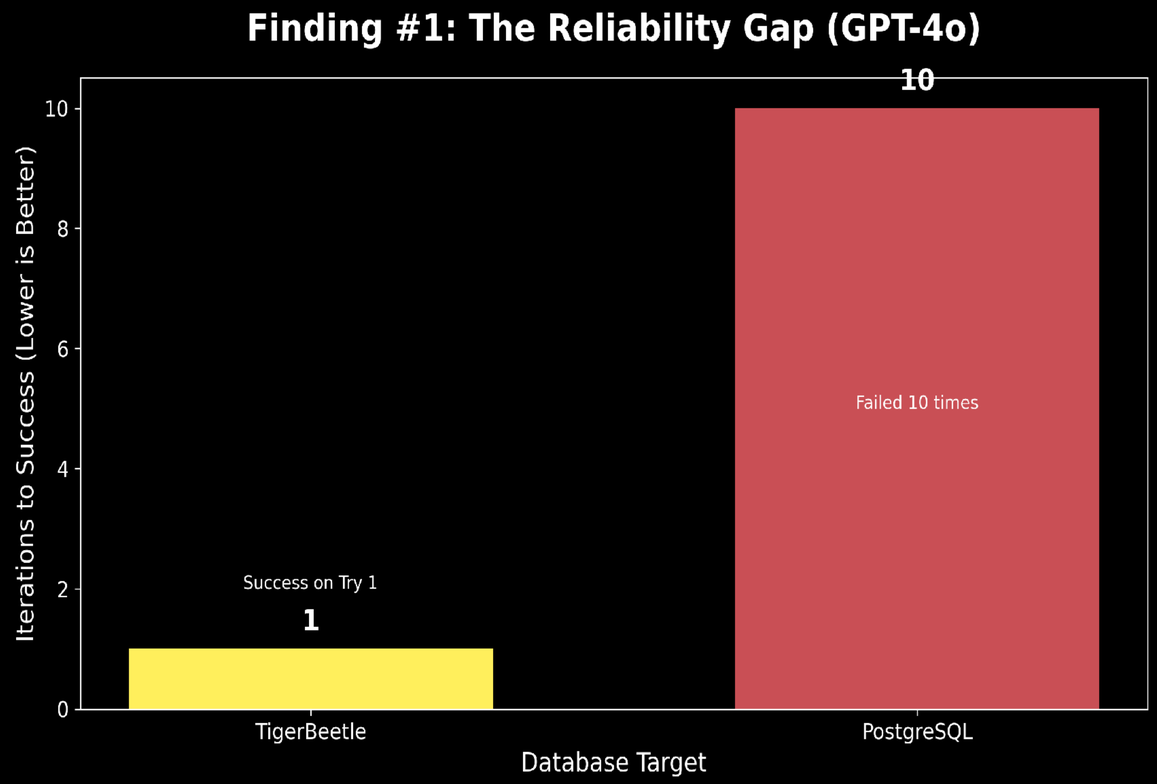
TigerBeetle: "Easy Mode"

GPT-4o Results:

- TigerBeetle: Success first try!
- PostgreSQL: Failed 10 times

GPT-5.1 Results:

- Both succeeded on Iteration 1



Why GPT-4o Failed PostgreSQL

Model choked on architectural complexity:

- **Schema Design:** Failed to create tables correctly (Schema relation errors)
- **Transaction Management:** Struggled to coordinate BEGIN/COMMIT with DDL

TigerBeetle Difference:

- No schema to design. No transactions to manage.
- Just filling out a struct.

Finding #2: Safety Gap

Even when models succeeded, **code quality differed.**

PostgreSQL Code (GPT-4o)

- Correctness Score: 9/10
- **Critical Flaw:** Missing Idempotency
- *Why? Model focused on SQL syntax/locking, "forgot" safety.*

TigerBeetle Code (GPT-4o)

- Correctness Score: 10/10
- **Safety:** Idempotency Enforced
- *Why? API requires transfer.id. Database forced the safety.*

TigerBeetle Example

TigerBeetle

Run 1:

- ✓ Success in 1 iterations
- ✓ Validation passed
- Performance: 0.74s (1350 payments/s)
- Batching: loop_with_batches
- **Code Quality: 9.6/10**

Code Review Details:

Dimension	Score
Correctness	10/10
Performance	10/10
Safety	9/10
Concurrency	10/10
Code Quality	9/10

Correctness:
- Strengths:
  - Correctly implements double-entry bookkeeping by creating Transfer objects with debit_account_id and credit_account_id
  - Properly handles idempotent retries by treating EXISTS errors as success, which is correct per API documentation
  - All required Transfer fields are correctly populated with appropriate values
  - Correctly maps error indices back to original payments in batches
  - Edge case of empty payment list is handled implicitly (no operations performed)

Performance:
- Strengths:
  - Excellent use of batching with the maximum batch size of 8189 transfers per request
  - Minimizes database round-trips by processing as many transfers as possible in each batch
  - Efficient batch processing loop that respects TigerBeetle's limits
  - No unnecessary lookups or operations - creates all transfers directly
  - Single client connection properly reused across all batches

Safety:
- Issues:
  - Connection string parsing is lenient but could fail silently on malformed input (uses defaults without warning)
  - No validation that payment.id, sender_id, recipient_id, and amount are within valid u128 ranges (though int() cast may handle this)
- Strengths:
  - Proper use of context manager (with statement) ensures client cleanup
  - Comprehensive error handling - all transfer errors are captured and reported
  - Correctly distinguishes between idempotent retries (EXISTS) and actual failures
  - Uses error_map to safely handle sparse error responses
  - All required Transfer fields explicitly set to valid values (no implicit defaults that could cause issues)

Concurrency:
- Strengths:
  - Correctly leverages TigerBeetle's idempotency guarantees - payment.id ensures each transfer is processed exactly once even with retries
  - API documentation confirms clients are thread-safe, so this function is safe for concurrent calls
  - No race conditions: idempotency is handled at the database level via unique transfer IDs
  - Proper handling of EXISTS error means concurrent or retry attempts are safely managed
  - Each function call creates its own client connection, avoiding shared mutable state

Code Quality:
- Issues:
  - Connection string parsing logic is somewhat ad-hoc and could be extracted to a helper function for reusability and testing
  - Silent fallback on parsing errors (e.g., invalid cluster_id) could make debugging harder - consider logging or raising warnings
- Strengths:
  - Clean, readable structure with clear separation of concerns
  - Excellent use of NamedTuples for type safety and documentation
  - Well-commented code explaining key decisions (idempotency handling, batch processing)
  - Consistent naming conventions throughout
  - Good use of type hints making the API clear
  - Constant BATCH_SIZE is well-named and documented
  - The batching loop is clear and easy to understand
  - Result construction is straightforward and unambiguous

Overall Assessment:
This is excellent production-quality code that demonstrates deep understanding of TigerBeetle's API. The implementation correctly uses batching for optimal performance (10/10 on performance), properly handles idempotency for safe retries (10/10 on concurrency), and correctly implements double-entry bookkeeping (10/10 on correctness). The code properly treats EXISTS errors as successful idempotent retries, which is the correct interpretation per the API documentation. Error handling is comprehensive and appropriate. The only minor weaknesses are in connection string parsing (which could be more robust) and lack of input validation for numeric ranges, but these are minor issues that don't affect the core payment processing logic. Overall, this code is well-suited for the stated requirements of efficiently processing 1000 payments with idempotent operations that are safe under concurrency.

```python
from typing import List, NamedTuple, Optional
import tigerbeetle as tb

class Payment(NamedTuple):
    """A payment to process"""
    id: int  # Unique payment ID (for idempotency)
    sender_id: int  # Account to debit
    recipient_id: int  # Account to credit
    amount: int  # Amount in cents

class PaymentResult(NamedTuple):
    """Result of processing a payment"""
    payment_id: int
    success: bool
    error: Optional[str] = None

def process_payments(
    payments: List[Payment],
    connection_info: str
) -> List[PaymentResult]:
    cluster_id, addresses = connection_info.split(",")

    with tb.ClientSync(
        cluster_id=int(cluster_id.split('=')[1]),
        replica_addresses=addresses.split('=')[1]
    ) as client:

        transfers = [
            tb.Transfer(
                id=payment.id,
                debit_account_id=payment.sender_id,
                credit_account_id=payment.recipient_id,
                amount=payment.amount,
                ledger=1,
                code=1,
                flags=0,
                pending_id=0,
                user_data_128=0,
                user_data_64=0,
                user_data_32=0,
                timeout=0,
                timestamp=0,
            )
            for payment in payments
        ]

        transfer_errors = client.create_transfers(transfers)

        # Map errors by index
        error_map = {error.index: error.result for error in transfer_errors}

        results = []
        for i, payment in enumerate(payments):
            if i in error_map:
                if error_map[i] == tb.CreateTransferResult.EXISTS:
                    results.append(PaymentResult(payment.id, True, None))
                else:
                    results.append(PaymentResult(payment.id, False, error_map[i].name))
            else:
                results.append(PaymentResult(payment.id, True, None))

        return results
```

PostgreSQL Example

PostgreSQL

Run 1:

- ✓ Success in 2 iterations
- ✓ Validation passed
- Performance: 1.62s (618 payments/s)
- Batching: unknown
- **Code Quality: 7.2/10**

Code Review Details:

Dimension	Score
Correctness	9/10
Performance	3/10
Safety	8/10
Concurrency	9/10
Code Quality	7/10

Correctness:
- Issues:
  - Missing idempotency check: The requirements specify 'idempotent operations', but there's no mechanism to prevent duplicate processing if the same payment.id is submitted multiple times. Should use 'INSERT INTO processed_payments ... ON CONFLICT DO NOTHING' or similar to track processed payment IDs.
- Strengths:
  - Double-entry bookkeeping correctly implemented: debits sender, credits recipient for same amount
  - Proper validation: checks sender exists and has sufficient funds before proceeding
  - Proper validation: checks recipient exists before transferring
  - Correct use of FOR UPDATE to lock rows during balance checks
  - Returns detailed results for each payment with success/failure status

Performance:
- Issues:
  - No batching whatsoever: processes 1000 payments one-by-one with individual transactions, resulting in 2000+ database round-trips (2+ queries per payment)
  - Each payment requires 4 separate queries (2 SELECTs + 2 UPDATEs) when this could be drastically reduced
  - With 1000 payments, this means ~4000 query executions and 1000 separate commit operations
  - Missing opportunity to batch updates: could group payments by accounts and update balances in fewer operations
  - Does not use executemany(), execute_values(), or any batch processing techniques mentioned in API docs
- Strengths:
  - Uses parameterized queries correctly to prevent SQL injection
  - Fetches only necessary columns (balance) rather than SELECT *

Safety:
- Issues:
  - Explicit BEGIN inside transaction context manager: The 'with conn' context manager already handles transactions, so explicit 'BEGIN' is redundant and could cause confusion
  - Mixed transaction control: uses both cur.execute('ROLLBACK') and conn.rollback(), which is inconsistent (though functionally equivalent)
- Strengths:
  - Proper exception handling with try-except blocks
  - Rollback on errors to maintain consistency
  - Uses context managers for automatic connection cleanup
  - Returns error messages in results rather than silently failing
  - Validates business logic (sufficient funds) before attempting updates

Concurrency:
- Issues:
  - Without idempotency tracking, concurrent processes could execute the same payment.id multiple times (though each individual execution would be safe due to FOR UPDATE locking)
- Strengths:
  - Correctly uses SELECT ... FOR UPDATE to acquire row locks on both sender and recipient accounts
  - Locks are held for the duration of the transaction, preventing concurrent modifications to the same accounts
  - Transaction isolation ensures atomic updates: either both accounts are updated or neither
  - Row-level locking allows different payments affecting different accounts to process concurrently
  - Proper transaction boundaries prevent partial updates

Code Quality:
- Issues:
  - Poor separation of concerns: validation, locking, and updates all mixed in one large loop
  - Repetitive code: balance fetching pattern is repeated for sender and recipient
  - Magic numbers in error messages: could use constants or enums for error types
  - The explicit BEGIN/COMMIT/ROLLBACK calls are unnecessary given the context manager usage
  - Could benefit from helper functions like 'lock_and_get_balance()' to reduce duplication
- Strengths:
  - Well-typed with NamedTuple classes for Payment and PaymentResult
  - Clear function signature with type hints
  - Good use of context managers for resource management
  - Descriptive variable names (sender_row, recipient_id, etc.)
  - Returns structured results that are easy to process

Overall Assessment:
This code correctly implements the core payment processing logic with proper double-entry bookkeeping and good concurrency safety through row-level locking. However, it has a critical performance problem: it processes 1000 payments sequentially with individual transactions, resulting in thousands of database round-trips. The requirements explicitly state 'efficient batching', which is completely absent here. For production use with 1000 payments, this would be unacceptably slow. The code also lacks idempotency tracking despite this being a stated requirement. To meet the requirements, this needs significant refactoring to batch operations (group payments by affected accounts, use bulk updates, reduce transaction count). The safety and correctness are solid, but the performance issues are severe enough to make this unsuitable for the stated use case.

```python
from typing import List, NamedTuple, Optional
import psycopg2

class Payment(NamedTuple):
    """A payment to process"""
    id: int  # Unique payment ID (for idempotency)
    sender_id: int  # Account to debit
    recipient_id: int  # Account to credit
    amount: int  # Amount in cents

class PaymentResult(NamedTuple):
    """Result of processing a payment"""
    payment_id: int
    success: bool
    error: Optional[str] = None

def process_payments(
        payments: List[Payment],
        connection_info: str
    ) -> List[PaymentResult]:

    with psycopg2.connect(connection_info) as conn:
        with conn.cursor() as cur:
            result = []

            for payment in payments:
                try:
                    # Begin transaction
                    cur.execute("BEGIN")

                    # Fetch sender's balance
                    cur.execute(
                        "SELECT balance FROM accounts WHERE id = %s FOR UPDATE",
                        (payment.sender_id,)
                    )
                    sender_row = cur.fetchone()

                    if not sender_row or sender_row[0] < payment.amount:
                        # Insufficient funds or sender does not exist
                        result.append(PaymentResult(payment_id=payment.id, success=False, error="Insufficient funds or sender does not exist"))
                        cur.execute("ROLLBACK")
                        continue

                    # Fetch recipient's account to ensure it exists
                    cur.execute(
                        "SELECT balance FROM accounts WHERE id = %s FOR UPDATE",
                        (payment.recipient_id,)
                    )

                    if not cur.fetchone():
                        # Recipient does not exist
                        result.append(PaymentResult(payment_id=payment.id, success=False, error="Recipient does not exist"))
                        cur.execute("ROLLBACK")
                        continue

                    # Update sender's balance
                    cur.execute(
                        "UPDATE accounts SET balance = balance - %s WHERE id = %s",
                        (payment.amount, payment.sender_id)
                    )

                    # Update recipient's balance
                    cur.execute(
                        "UPDATE accounts SET balance = balance + %s WHERE id = %s",
                        (payment.amount, payment.recipient_id)
                    )

                    # Commit transaction
                    conn.commit()

                    result.append(PaymentResult(payment_id=payment.id, success=True))

                except Exception as e:
                    conn.rollback()  # Ensure rollback on exception
                    result.append(PaymentResult(payment_id=payment.id, success=False, error=str(e)))

    return result
```

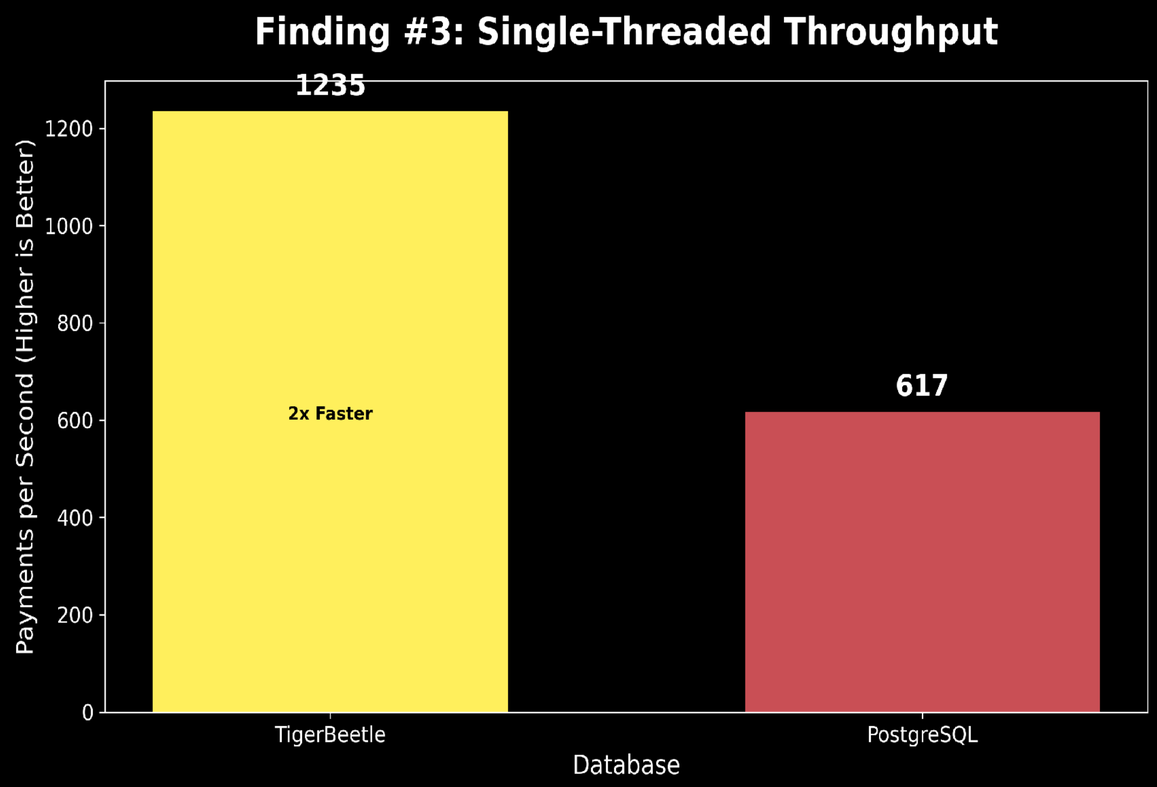
Finding #3: Single-Threaded Speed

Hypothesis Validated:
TigerBeetle 2x Faster

TigerBeetle: 1,235 pay/s
- Strategy: **Batching**

PostgreSQL: 617 pay/s
- Strategy: **Seq. Transactions**

Hypothesis looks perfect.



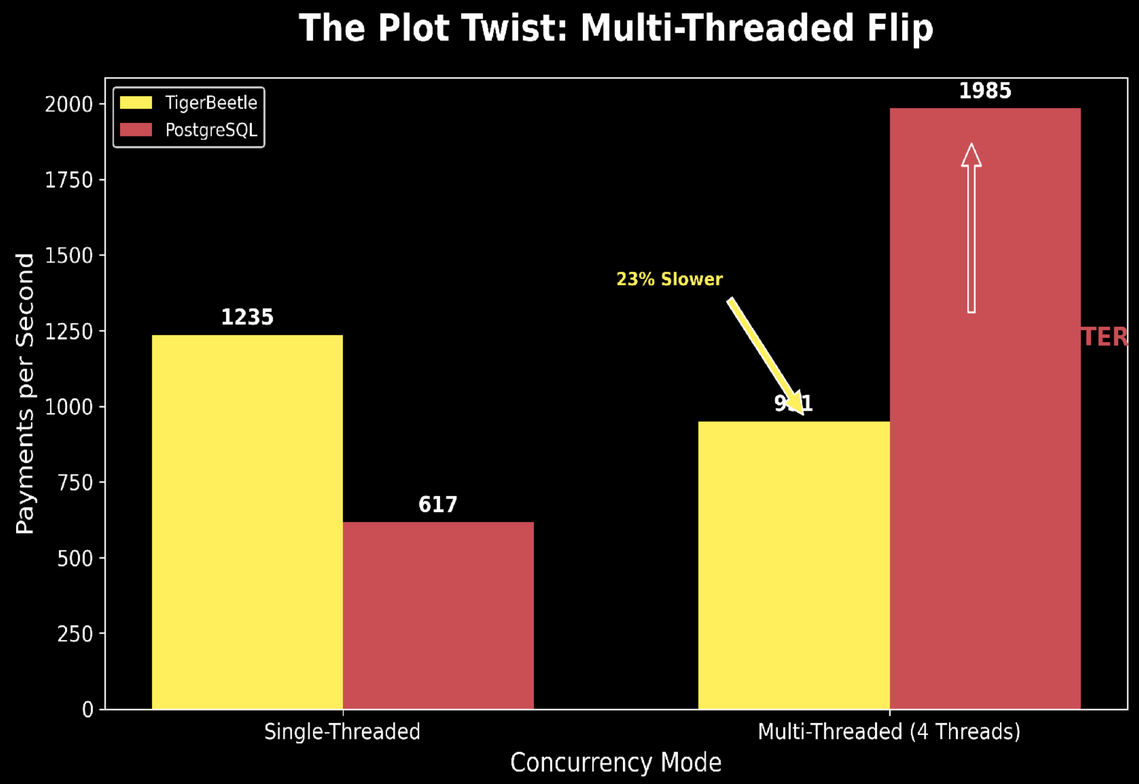
Plot Twist: Concurrency Test

We turned on Multi-Threading.
Results flipped:

TigerBeetle:
951 pay/s (-23%)

PostgreSQL:
1,985 pay/s (3.2x)

New Winner: Postgres



Analysis: TigerBeetle Slowdown

"Client Reuse" Trap

LLM code was syntactically correct but architecturally naive.

1. **Mistake:** Initialized New Client for every batch/thread
2. **Consequence:** Fragmented Batches
3. **Bottleneck:** Server flooded with tiny requests

Insight: API didn't prevent misuse of client lifecycle.

Accidental Genius: Sequential Isolation

"Bad" single-threaded code became "perfect" concurrent code.

1. **Pattern:** 1,000 isolated transactions
2. **Parallelism:** Postgres uses Row-Level Locking
3. **Result:** Perfect linear scaling

Irony: Model didn't *try* to be parallel; it tried to be transactional.
Postgres did the heavy lifting.

Does TigerBeetle Make GenAI Better?

Yes, but with a catch.

1. Guardrail Effect (Reliability)

- TigerBeetle is the **safest target**
- Eliminates architectural hallucinations
- Enforces safety guarantees
- **Allows weaker models to perform like SOTA**

2. Concurrency Trade-off (Performance)

- **TigerBeetle:** Optimizes for **Throughput**
- **PostgreSQL:** Optimizes for **Isolation**
- ***Must teach LLMs Resource Management to scale TB***

Human Implications

LLM as "Canary in the Coal Mine"

1. Cognitive RAM Lesson:
Removing "database design" turns **Junior Devs** into safe engineers.

2. Pit of Success:
Correctness must be **syntactic**. If code runs without safety, API is broken.

3. New Bottleneck:
As APIs abstract **logic**, humans must master **lifecycle**.

Final Verdict

TigerBeetle doesn't just help AI write better code.

It forces **humans** to think in:

- **Batches** (Performance)
- **Invariants** (Safety)

...while automating the "hard stuff" (Concurrency).

Final Verdict

TigerBeetle doesn't just help AI write better code.

It forces **humans** to think in:

- **Batches** (Performance)
- **Invariants** (Safety)

...while automating the "hard stuff" (Concurrency).

The best API for an AI is simply the best API for a Human.

TigerBeetle Goes GenAI

Do LLMs write better code with Accounting Primitives?

THANK YOU for your attention!