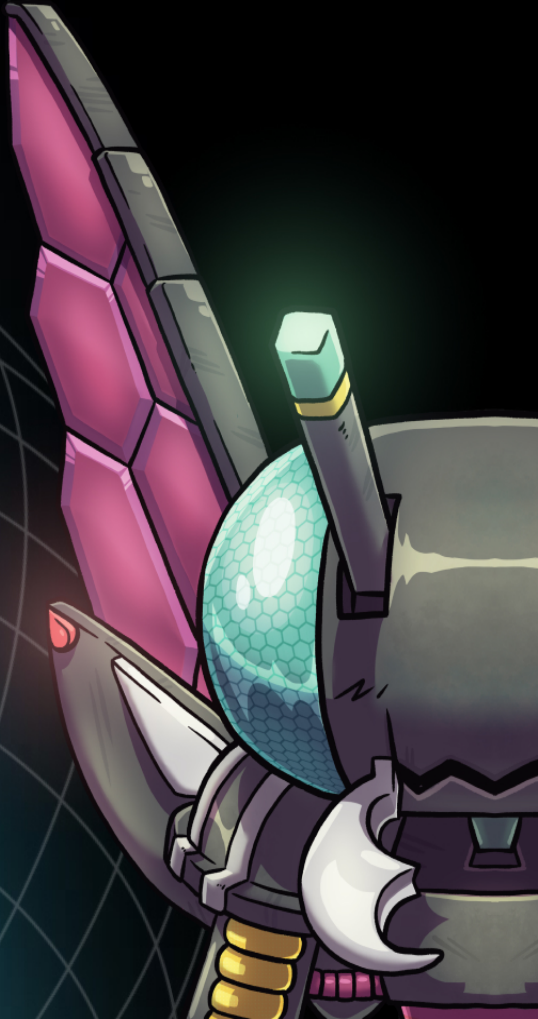
TigerBeetle

1000X

WORLD TOUR

13 CITIES | 6 DAYS



1000x

Cadê?

novos | inclusões | eventos | veja | ?

Busca | Informações

Consulta

Feliz Natal!

Compras
CDs, Flores, Livros

Ciência e Tecnologia
Institutos, Centros de Pesquisa

Educação
Cursos, Escolas, Universidades

Cultura
Museus, Música, Personalidades

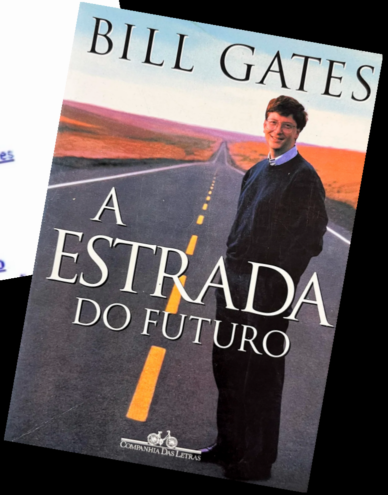
Finanças
Bancos, Bolsas, Seguros

Esportes
Automobilismo, Futebol

Indústria e Comércio

Governo

Dialing Progress



1000X

OLAP

Online Analytical Processing

1000X

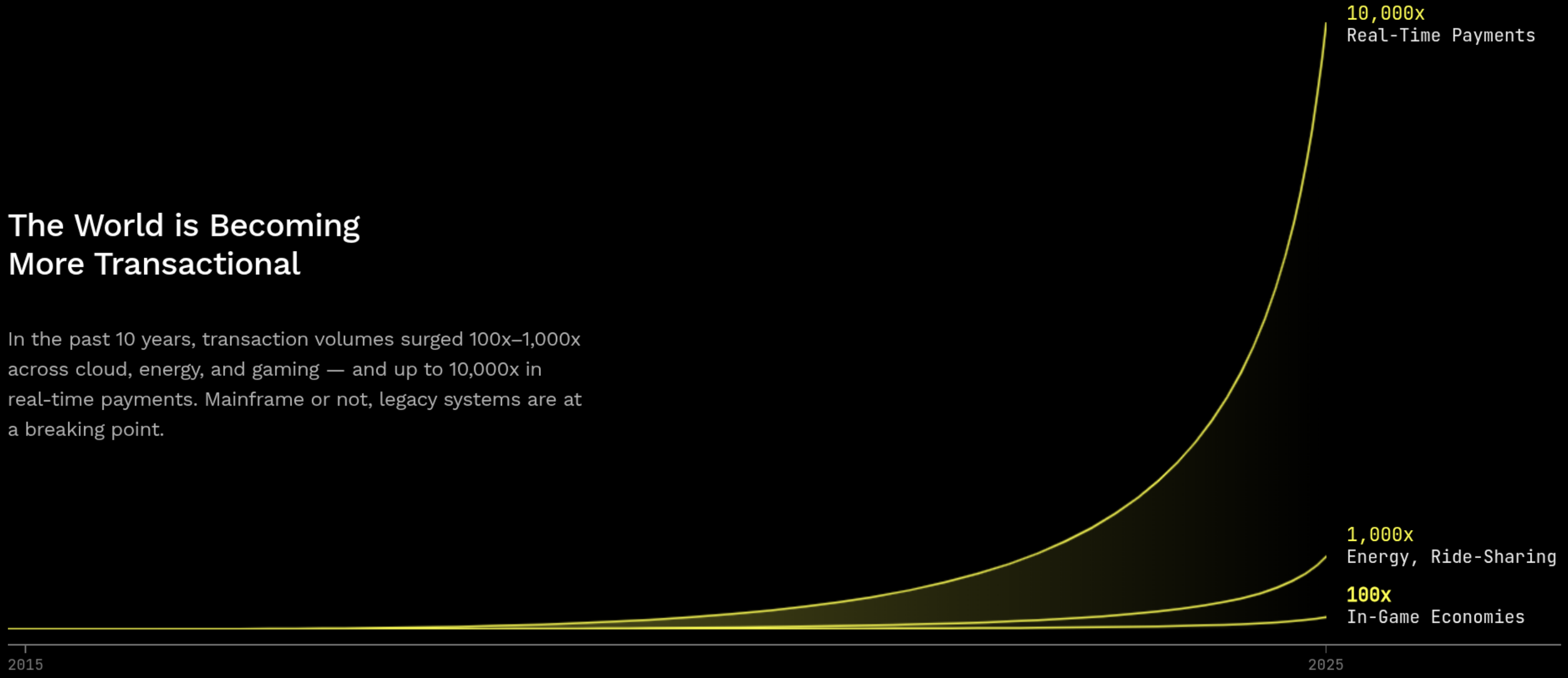
“A história não se repete, mas rima.”

Mark Twain

1000X

The World is Becoming More Transactional

In the past 10 years, transaction volumes surged 100x–1,000x across cloud, energy, and gaming — and up to 10,000x in real-time payments. Mainframe or not, legacy systems are at a breaking point.



1000X

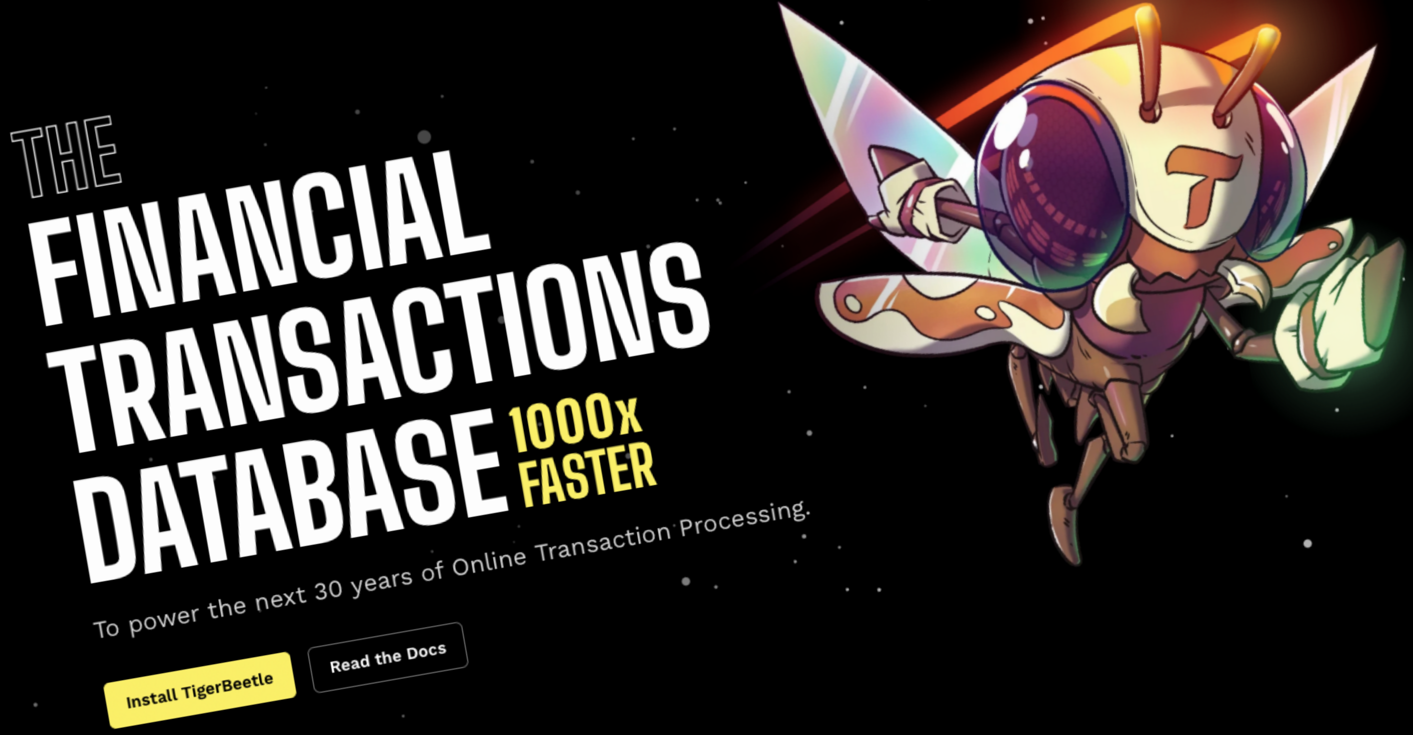
Transações estão em toda parte!

1000X

OLTP

Online Transaction Processing

1000X



THE FINANCIAL TRANSACTIONS DATABASE 1000x FASTER

To power the next 30 years of Online Transaction Processing.

1000X

THE FINANCIAL TRANSACTIONS DATABASE 1000x FASTER

To power the next 30 years of Online Transaction Processing.

1000X

A filosofia por trás do processo de desenvolvimento do TigerBeetle

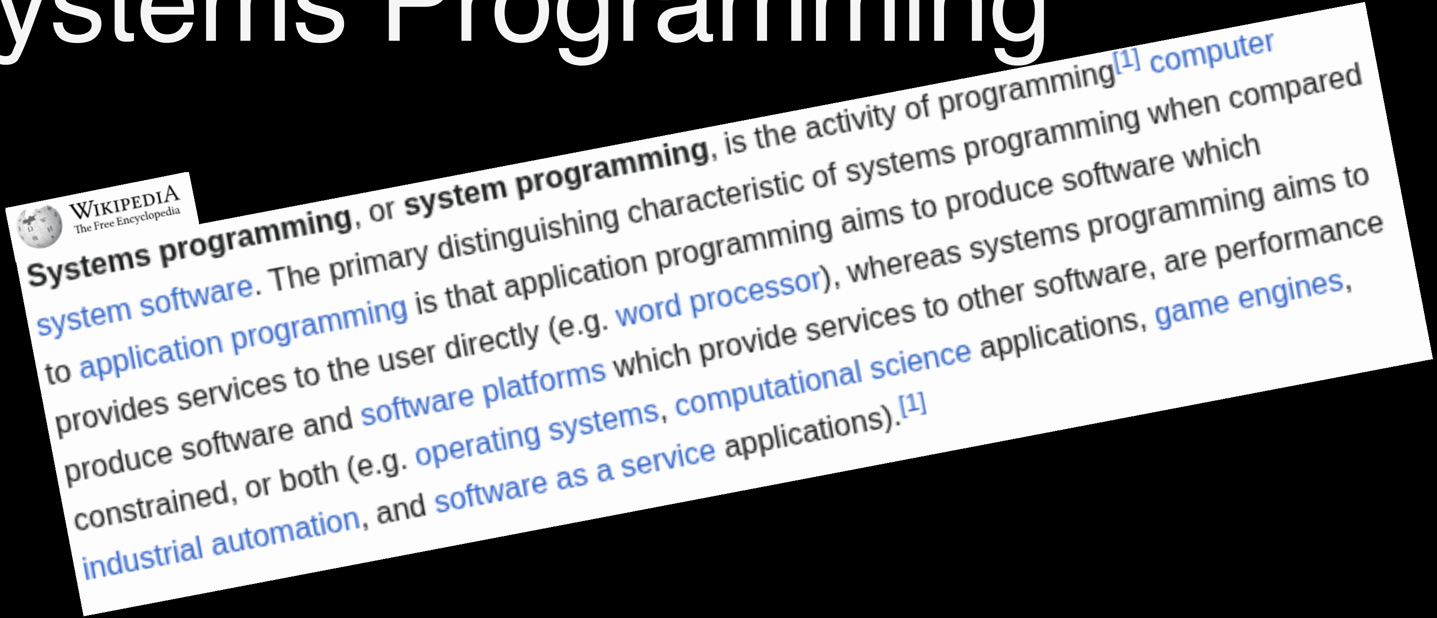
Rafael Batiati
Senior Engineer @ TigerBeetle

x.com/rbatiati
github.com/batiati

1000X

10^1 = Systems Programming

10^1 = Systems Programming



Systems programming, or **system programming**, is the activity of programming[1] computer system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both (e.g. operating systems, computational science applications, game engines, industrial automation, and software as a service applications).[1]

1000X

10^1 = Systems Programming

"*A way of modeling software development*"

Andrew Kelley
Criador da linguagem
de programação ZIG

https://youtu.be/Qncdi-Fg0-I

10^1 = Systems Programming

> *“The purpose of abstracting is not to be vague,
> but to create a new semantic level in which
> one can be absolutely precise.”*

Edsger W. Dijkstra

1000X

10^1 = Systems Programming

1000X

10^2 = Esforço *vs* Resultado

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

"Go slow to go fast"

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

1000X

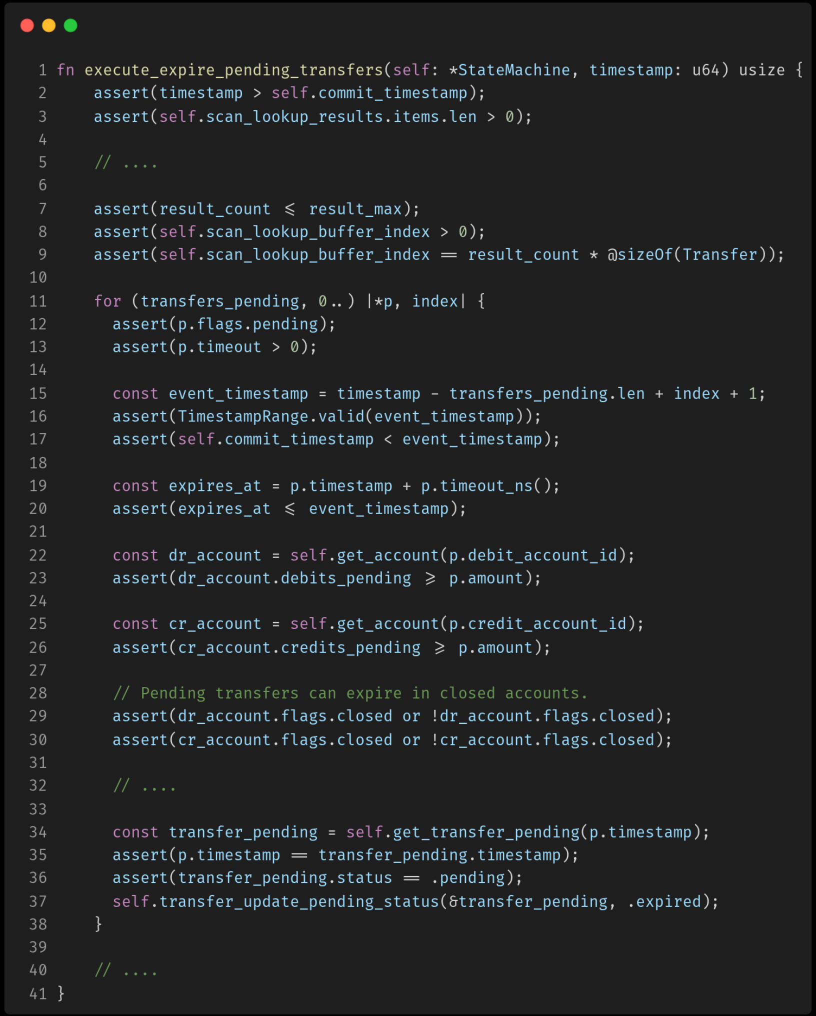
10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

100

$10^2 =$ ado

Tempo = **+ Testes + Incidentes**

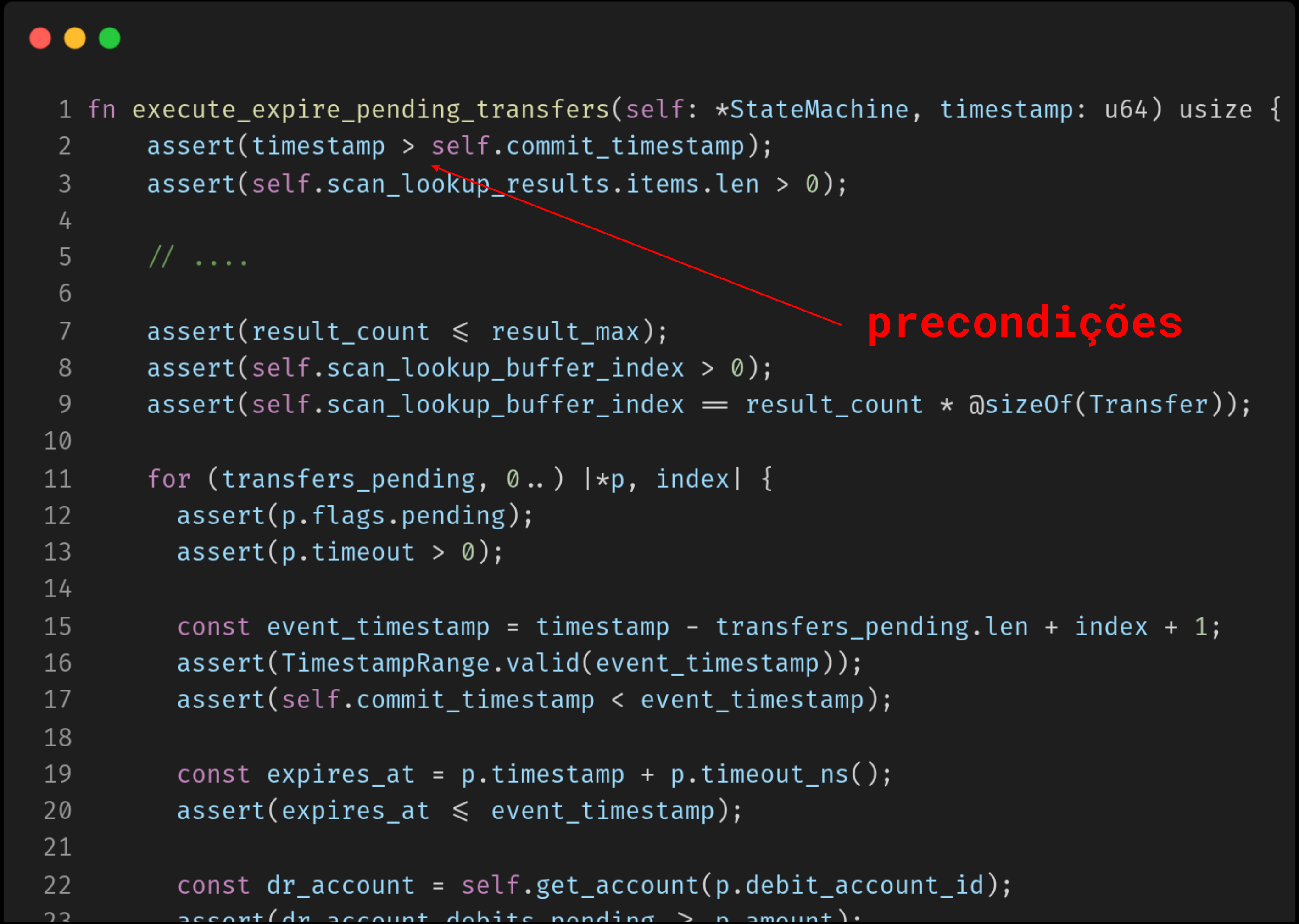


```
fn execute_expire_pending_transfers(self: *StateMachine, timestamp: u64) usize {
    assert(timestamp > self.commit_timestamp);
    assert(self.scan_lookup_results.items.len > 0);

    // ....

    assert(result_count <= result_max);
    assert(self.scan_lookup_buffer_index > 0);
    assert(self.scan_lookup_buffer_index == result_count * @sizeOf(Transfer));

    for (transfers_pending, 0..) |*p, index| {
      assert(p.flags.pending);
      assert(p.timeout > 0);

      const event_timestamp = timestamp - transfers_pending.len + index + 1;
      assert(TimestampRange.valid(event_timestamp));
      assert(self.commit_timestamp < event_timestamp);

      const expires_at = p.timestamp + p.timeout_ns();
      assert(expires_at <= event_timestamp);

      const dr_account = self.get_account(p.debit_account_id);
      assert(dr_account.debits_pending >= p.amount);

      const cr_account = self.get_account(p.credit_account_id);
      assert(cr_account.credits_pending >= p.amount);

      // Pending transfers can expire in closed accounts.
      assert(dr_account.flags.closed or !dr_account.flags.closed);
      assert(cr_account.flags.closed or !cr_account.flags.closed);

      // ....

      const transfer_pending = self.get_transfer_pending(p.timestamp);
      assert(p.timestamp == transfer_pending.timestamp);
      assert(transfer_pending.status == .pending);
      self.transfer_update_pending_status(&transfer_pending, .expired);
    }

    // ....
}
```

100

$10^2 =$

Tempo = **es**



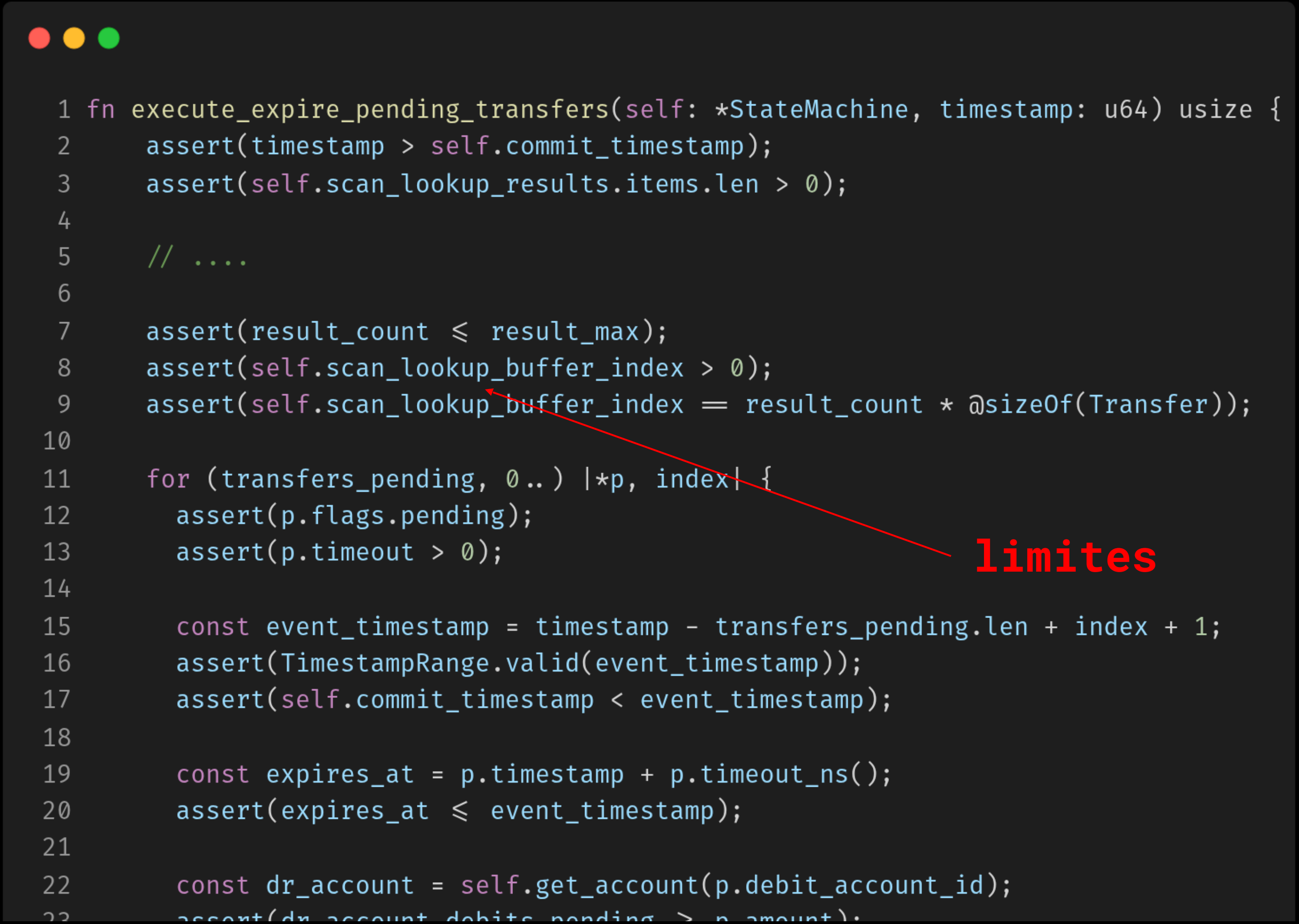
```
fn execute_expire_pending_transfers(self: *StateMachine, timestamp: u64) usize {
    assert(timestamp > self.commit_timestamp);
    assert(self.scan_lookup_results.items.len > 0);

    // ....

    assert(result_count <= result_max);
    assert(self.scan_lookup_buffer_index > 0);
    assert(self.scan_lookup_buffer_index == result_count * @sizeOf(Transfer));

    for (transfers_pending, 0..) |*p, index| {
      assert(p.flags.pending);
      assert(p.timeout > 0);

      const event_timestamp = timestamp - transfers_pending.len + index + 1;
      assert(TimestampRange.valid(event_timestamp));
      assert(self.commit_timestamp < event_timestamp);

      const expires_at = p.timestamp + p.timeout_ns();
      assert(expires_at <= event_timestamp);

      const dr_account = self.get_account(p.debit_account_id);
      assert(dr_account.debits_pending >= p.amount);
```

100

$10^2 =$

Tempo =

es



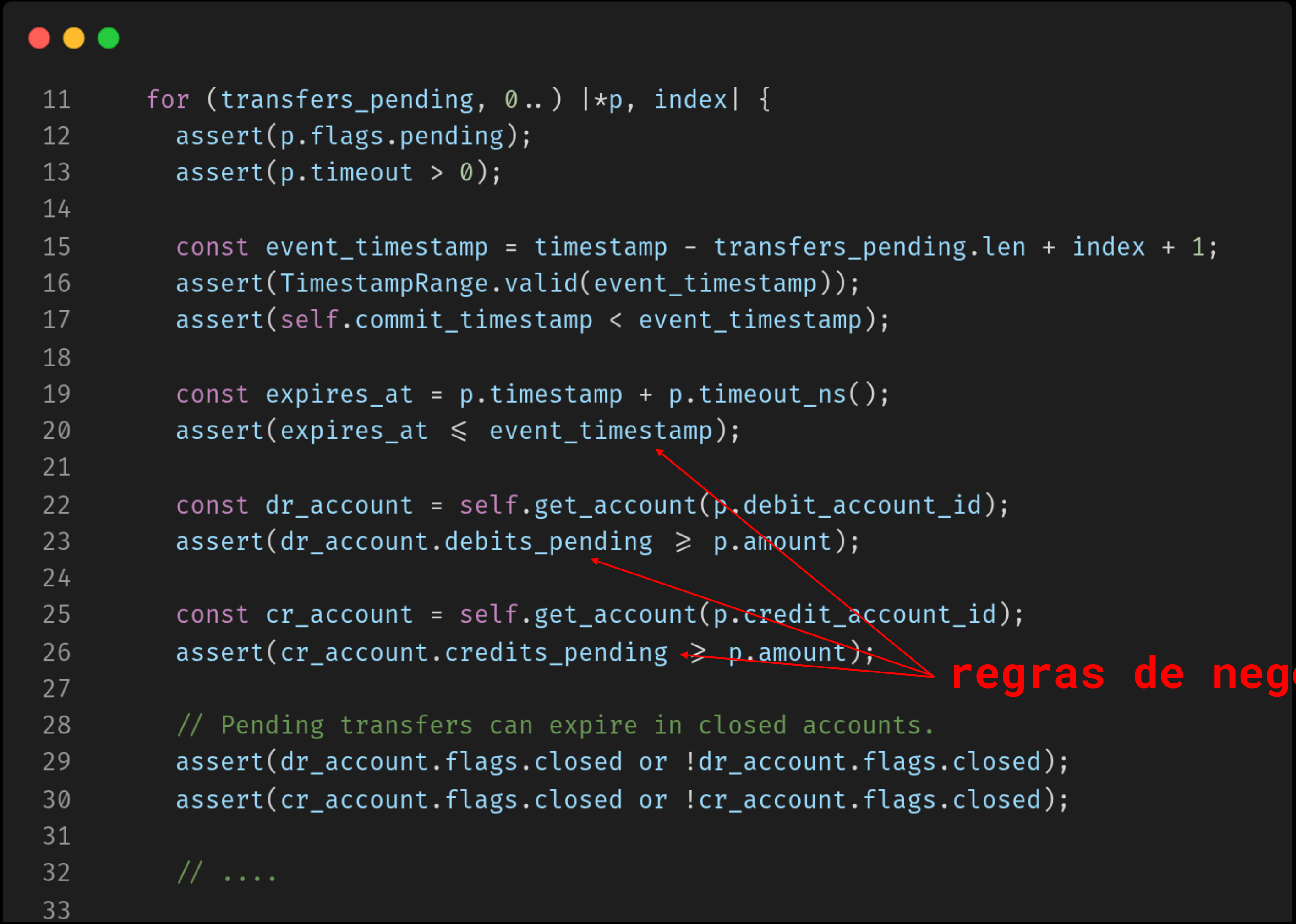
```
fn execute_expire_pending_transfers(self: *StateMachine, timestamp: u64) usize {
    assert(timestamp > self.commit_timestamp);
    assert(self.scan_lookup_results.items.len > 0);

    // ....

    assert(result_count <= result_max);
    assert(self.scan_lookup_buffer_index > 0);
    assert(self.scan_lookup_buffer_index == result_count * @sizeOf(Transfer));

    for (transfers_pending, 0..) |*p, index| {
      assert(p.flags.pending);
      assert(p.timeout > 0);

      const event_timestamp = timestamp - transfers_pending.len + index + 1;
      assert(TimestampRange.valid(event_timestamp));
      assert(self.commit_timestamp < event_timestamp);

      const expires_at = p.timestamp + p.timeout_ns();
      assert(expires_at <= event_timestamp);

      const dr_account = self.get_account(p.debit_account_id);
      assert(dr_account.debits_pending >= p.amount);
```

100

$10^2 =$

Tempo =

es



```
for (transfers_pending, 0..) |*p, index| {
  assert(p.flags.pending);
  assert(p.timeout > 0);

  const event_timestamp = timestamp - transfers_pending.len + index + 1;
  assert(TimestampRange.valid(event_timestamp));
  assert(self.commit_timestamp < event_timestamp);

  const expires_at = p.timestamp + p.timeout_ns();
  assert(expires_at <= event_timestamp);

  const dr_account = self.get_account(p.debit_account_id);
  assert(dr_account.debits_pending >= p.amount);

  const cr_account = self.get_account(p.credit_account_id);
  assert(cr_account.credits_pending >= p.amount);

  // Pending transfers can expire in closed accounts.
  assert(dr_account.flags.closed or !dr_account.flags.closed);
  assert(cr_account.flags.closed or !cr_account.flags.closed);

  // ....

```

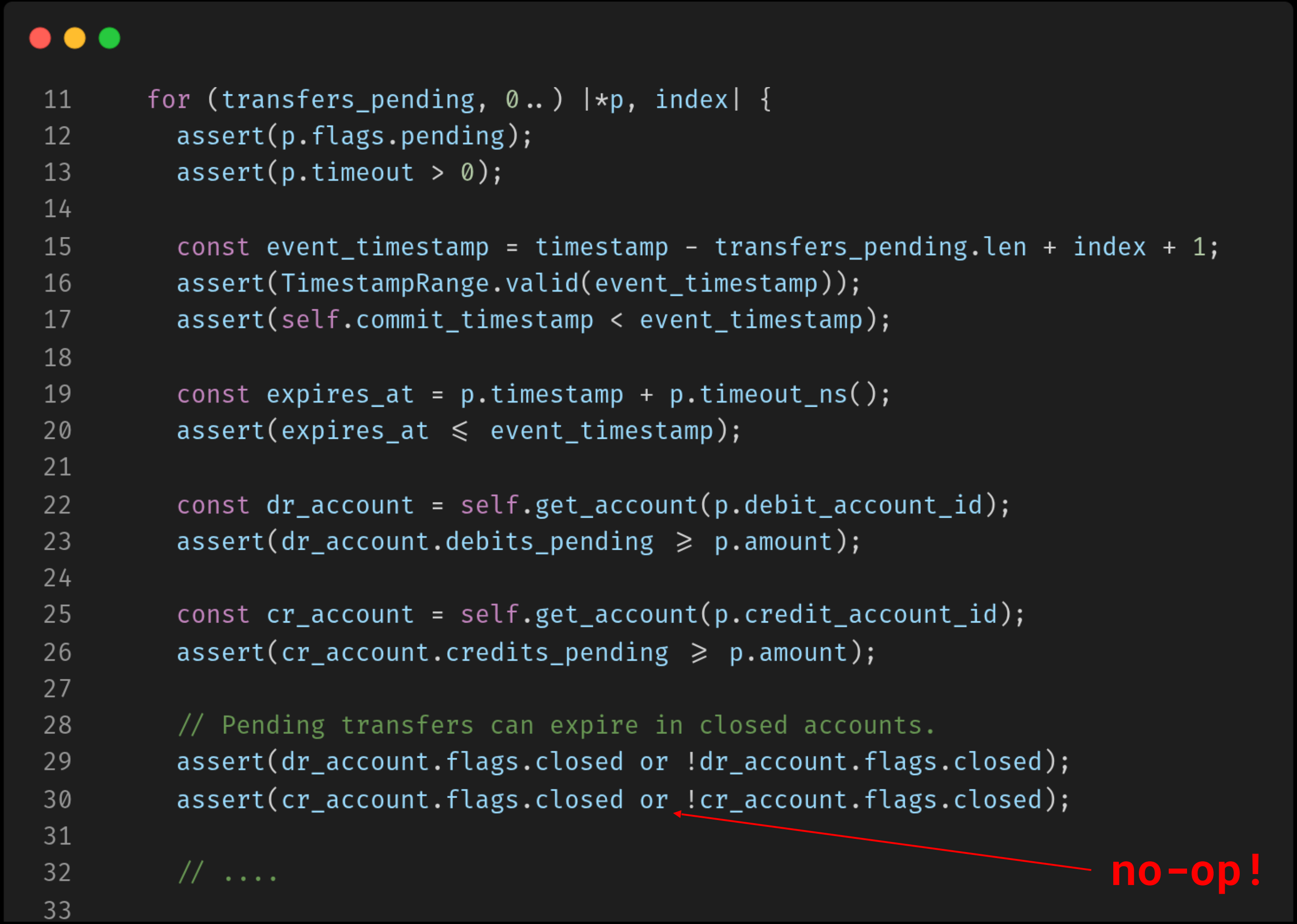
regras de negócio

100

10^2 =

Tempo =

es



```
for (transfers_pending, 0..) |*p, index| {
  assert(p.flags.pending);
  assert(p.timeout > 0);

  const event_timestamp = timestamp - transfers_pending.len + index + 1;
  assert(TimestampRange.valid(event_timestamp));
  assert(self.commit_timestamp < event_timestamp);

  const expires_at = p.timestamp + p.timeout_ns();
  assert(expires_at ≤ event_timestamp);

  const dr_account = self.get_account(p.debit_account_id);
  assert(dr_account.debits_pending ≥ p.amount);

  const cr_account = self.get_account(p.credit_account_id);
  assert(cr_account.credits_pending ≥ p.amount);

  // Pending transfers can expire in closed accounts.
  assert(dr_account.flags.closed or !dr_account.flags.closed);
  assert(cr_account.flags.closed or !cr_account.flags.closed);

  // ....

```

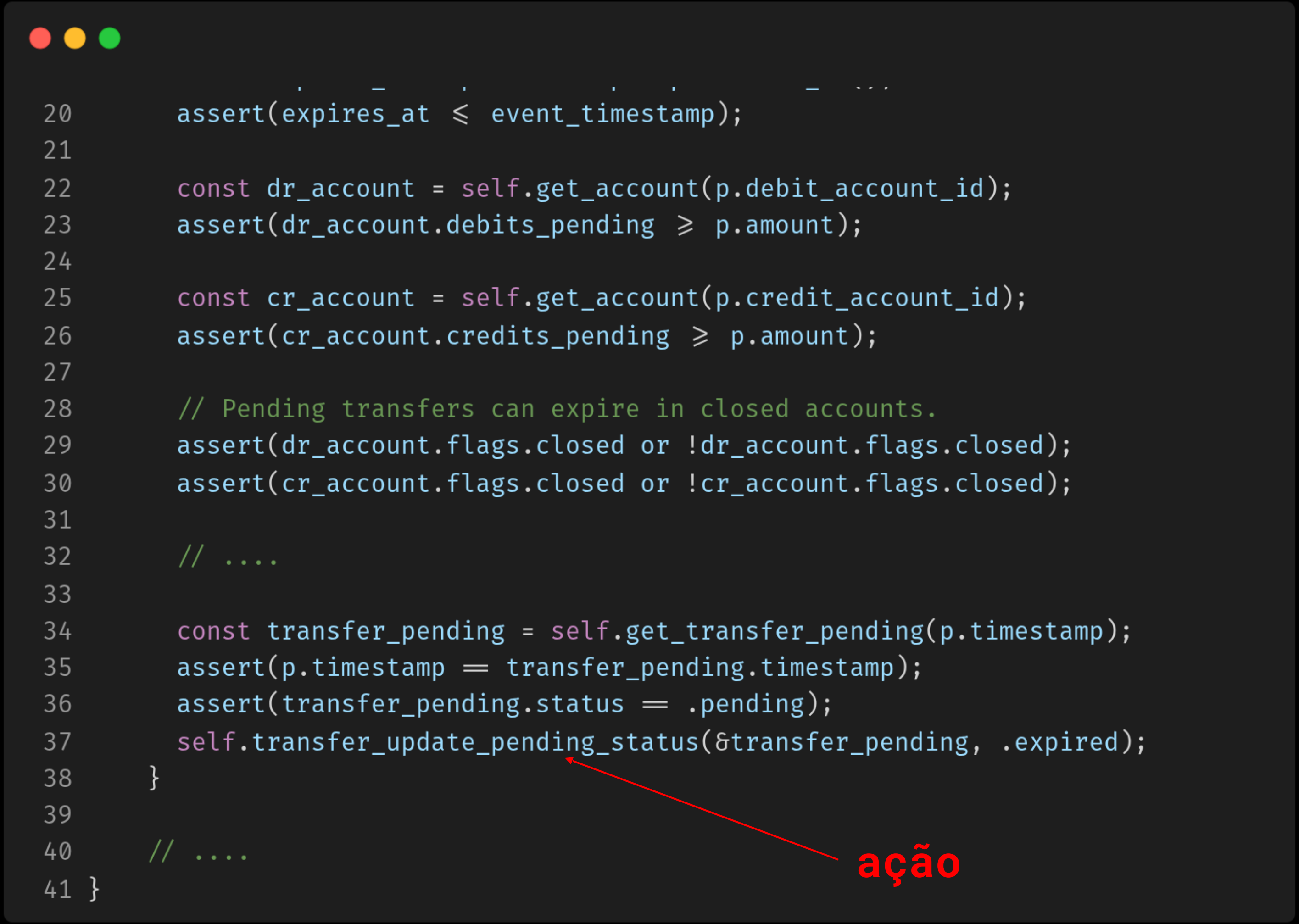
no-op!

100

$10^2 =$

Tempo =

es



```
      assert(expires_at <= event_timestamp);

      const dr_account = self.get_account(p.debit_account_id);
      assert(dr_account.debits_pending >= p.amount);

      const cr_account = self.get_account(p.credit_account_id);
      assert(cr_account.credits_pending >= p.amount);

      // Pending transfers can expire in closed accounts.
      assert(dr_account.flags.closed or !dr_account.flags.closed);
      assert(cr_account.flags.closed or !cr_account.flags.closed);

      // ....

      const transfer_pending = self.get_transfer_pending(p.timestamp);
      assert(p.timestamp == transfer_pending.timestamp);
      assert(transfer_pending.status == .pending);
      self.transfer_update_pending_status(&transfer_pending, .expired);
    }

    // ....
}
```

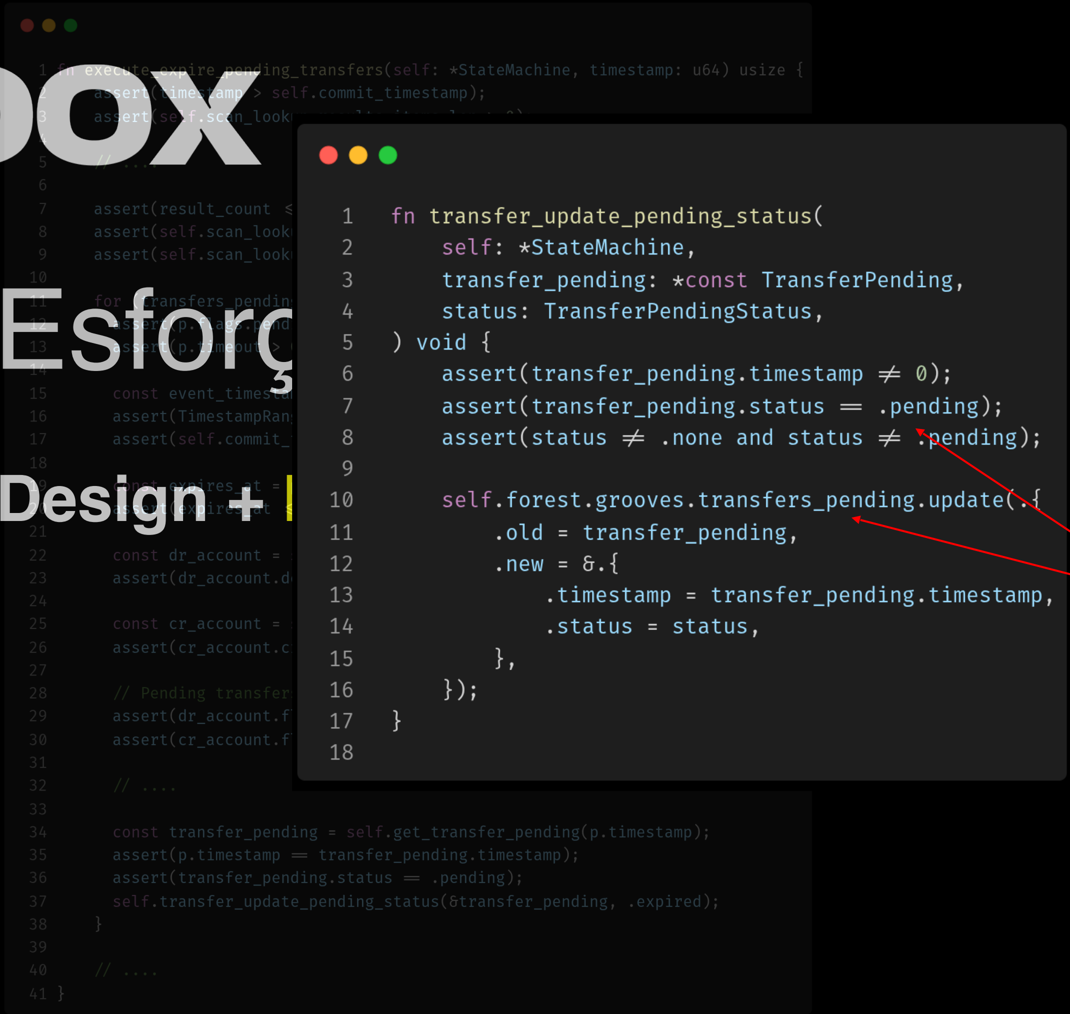
ação

1000x

10^2 = Esforç

Tempo = Design +

+ Incidentes



```
fn execute_expire_pending_transfers(self: *StateMachine, timestamp: u64) usize {
    assert(timestamp > self.commit_timestamp);
    assert(self.scan_look

    // ....

    assert(result_count ≤
    assert(self.scan_look
    assert(self.scan_look

    for (transfers_pendin
        assert(p.flags.pend
        assert(p.timeout >

        const event_timesta
        assert(TimestampRan
        assert(self.commit_

        const expires_at =
        assert(expires_at

        const dr_account =
        assert(dr_account.d

        const cr_account =
        assert(cr_account.c

        // Pending transfer
        assert(dr_account.f
        assert(cr_account.f

        // ....

        const transfer_pending = self.get_transfer_pending(p.timestamp);
        assert(p.timestamp == transfer_pending.timestamp);
        assert(transfer_pending.status == .pending);
        self.transfer_update_pending_status(&transfer_pending, .expired);
    }

    // ....
}
```

```
fn transfer_update_pending_status(
    self: *StateMachine,
    transfer_pending: *const TransferPending,
    status: TransferPendingStatus,
) void {
    assert(transfer_pending.timestamp != 0);
    assert(transfer_pending.status == .pending);
    assert(status != .none and status != .pending);

    self.forest.grooves.transfers_pending.update(.{
        .old = transfer_pending,
        .new = &.{
            .timestamp = transfer_pending.timestamp,
            .status = status,
        },
    });
}
```

tudo outra vez!!!

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

+

Revisão

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

Federico Lorenzi

https://youtu.be/P9nLS2reUOo

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic **S**imulation **T**esting

1000X

10^2 = Esforço *vs* Resultado

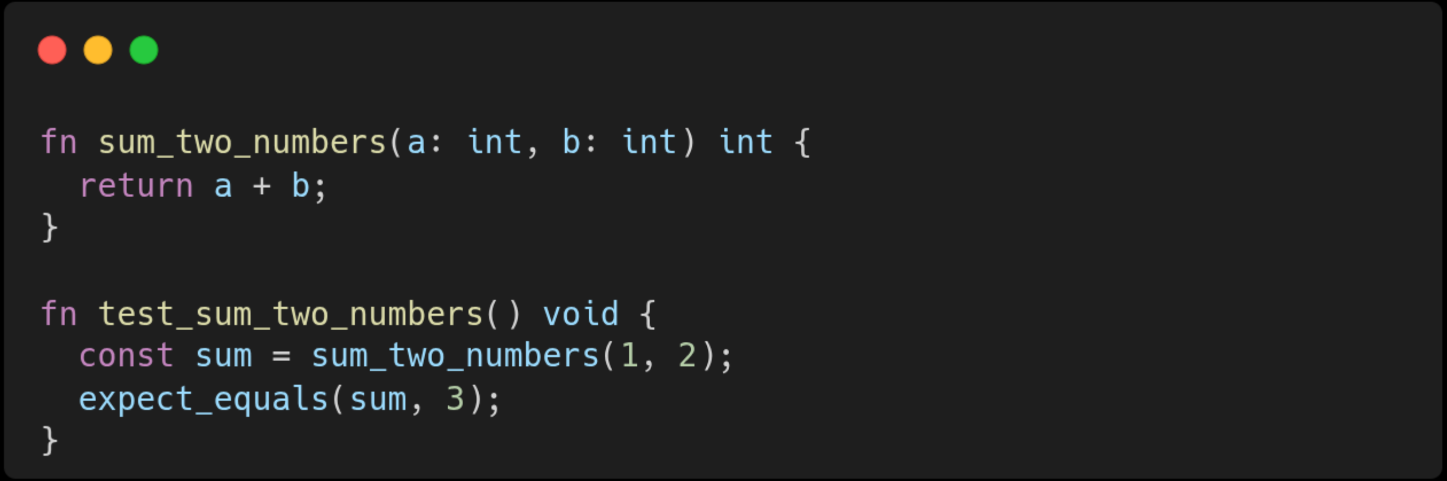
Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic **S**imulation **T**esting

100

$10^2 =$ I

Tempo = I



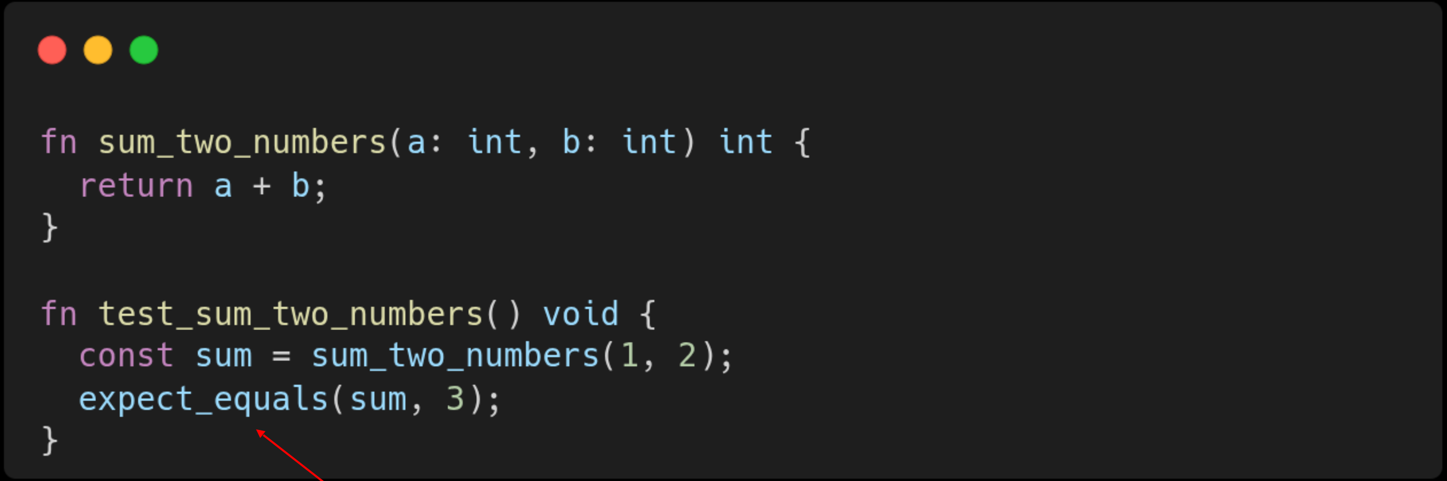
```
fn sum_two_numbers(a: int, b: int) int {
  return a + b;
}

fn test_sum_two_numbers() void {
  const sum = sum_two_numbers(1, 2);
  expect_equals(sum, 3);
}
```

Deterministic Simulation Testing

100

10^2 = I

Tempo = I



```
fn sum_two_numbers(a: int, b: int) int {
  return a + b;
}

fn test_sum_two_numbers() void {
  const sum = sum_two_numbers(1, 2);
  expect_equals(sum, 3);
}
```

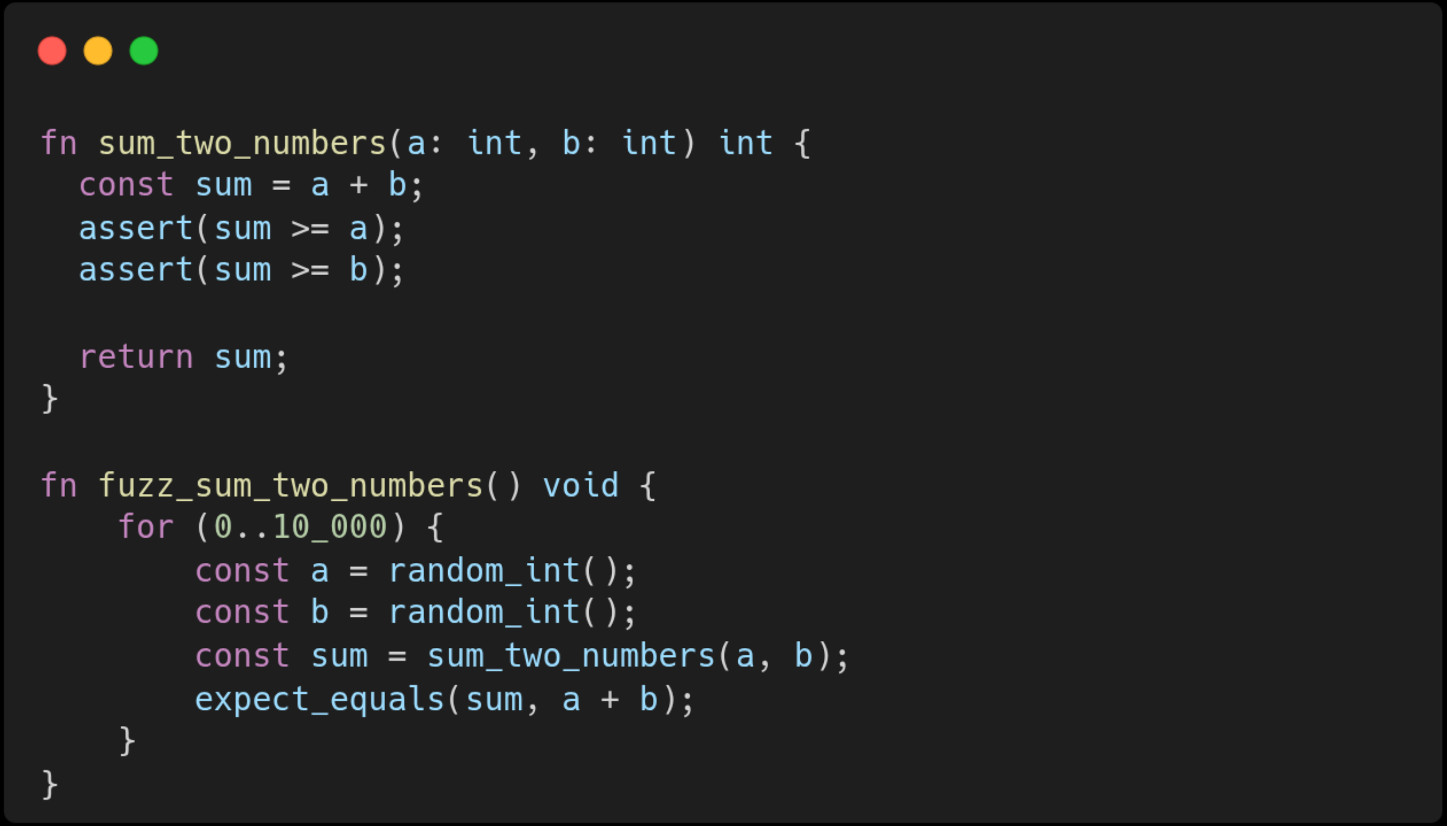
Deterministic Simulation Testing

comportamento
esperado

100

$10^2 =$

Tempo =

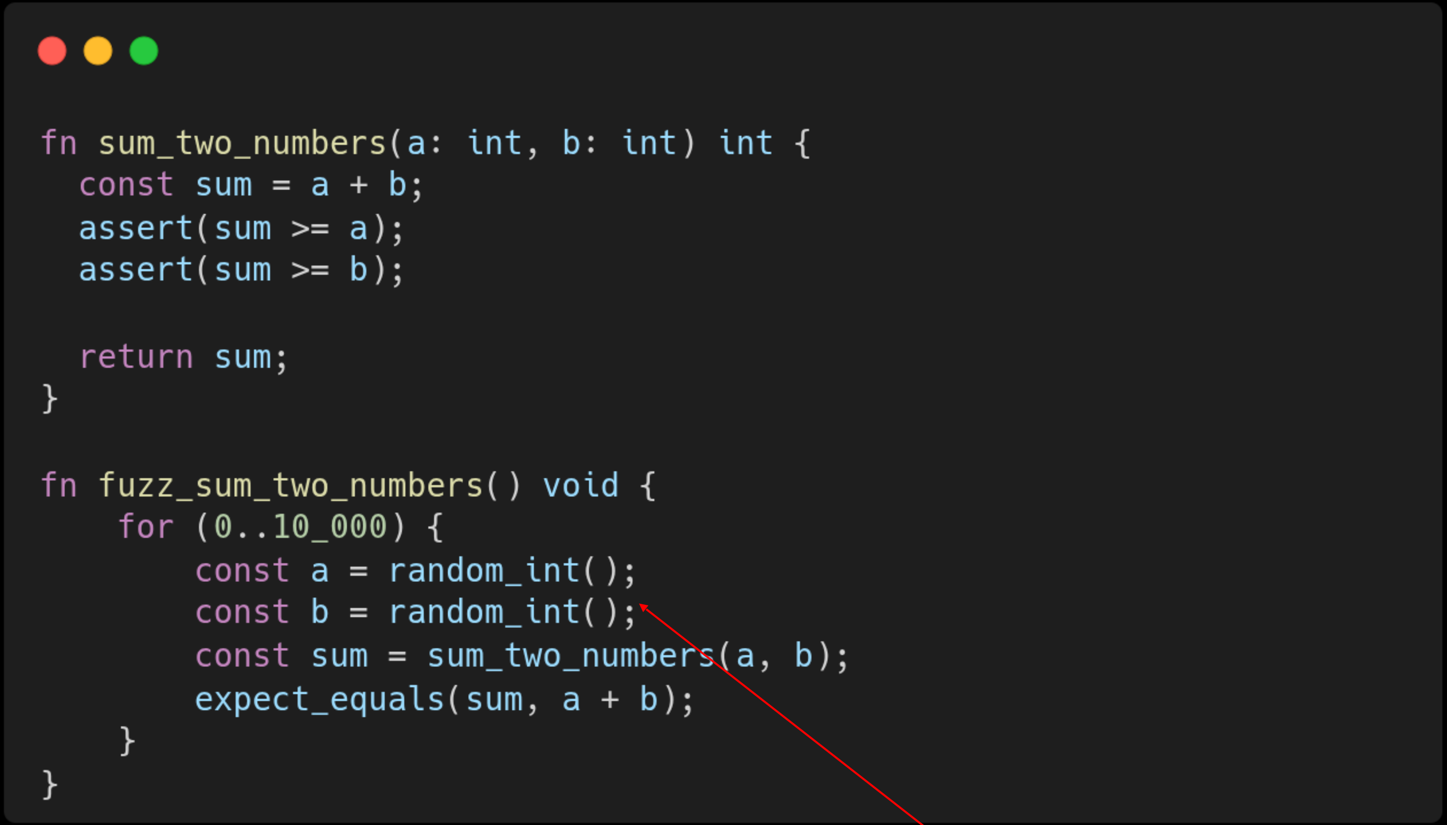


```
fn sum_two_numbers(a: int, b: int) int {
  const sum = a + b;
  assert(sum >= a);
  assert(sum >= b);

  return sum;
}

fn fuzz_sum_two_numbers() void {
    for (0..10_000) {
        const a = random_int();
        const b = random_int();
        const sum = sum_two_numbers(a, b);
        expect_equals(sum, a + b);
    }
}
```

100

$10^2 =$

Tempo =



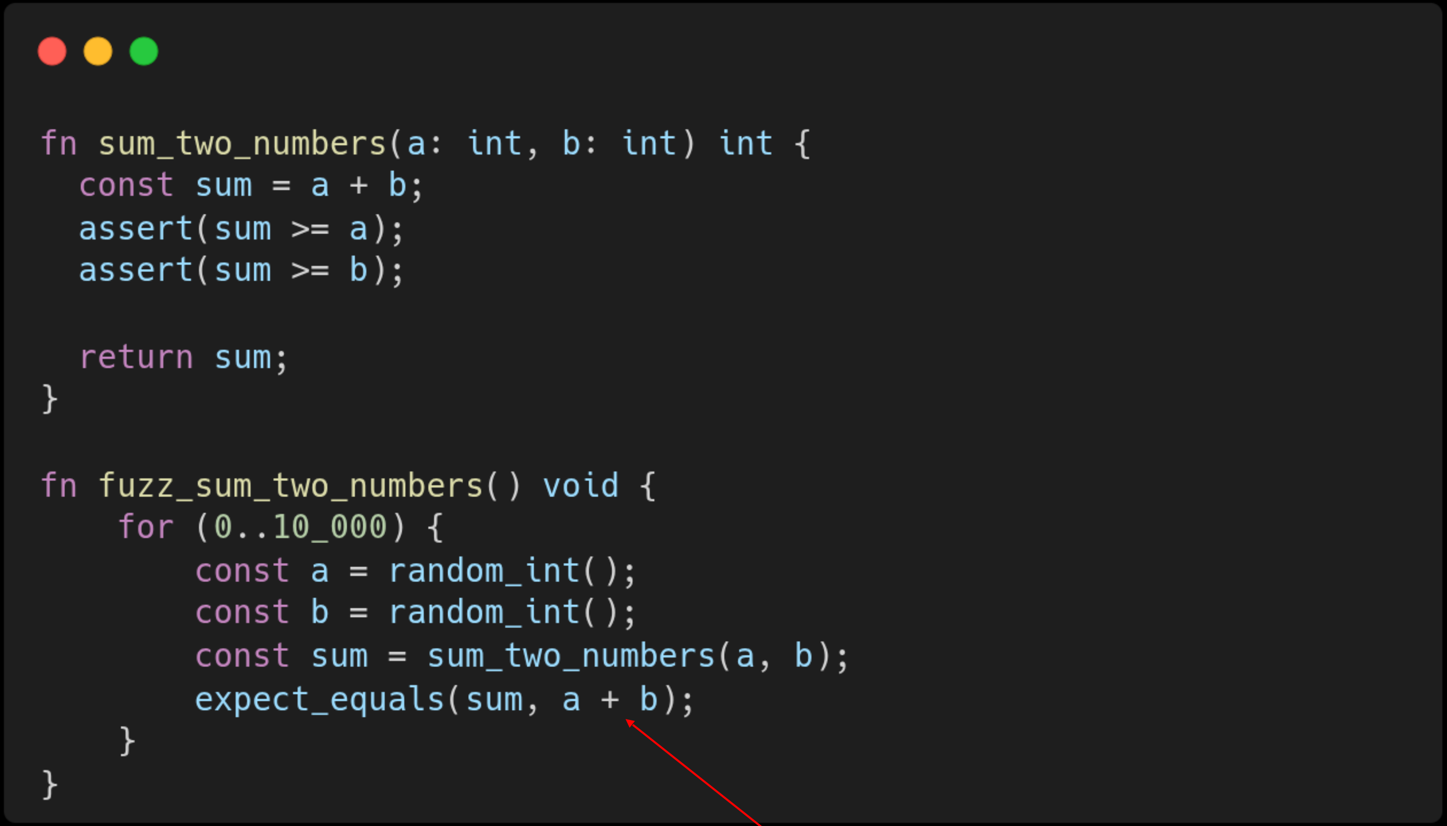
```
fn sum_two_numbers(a: int, b: int) int {
  const sum = a + b;
  assert(sum >= a);
  assert(sum >= b);

  return sum;
}

fn fuzz_sum_two_numbers() void {
    for (0..10_000) {
        const a = random_int();
        const b = random_int();
        const sum = sum_two_numbers(a, b);
        expect_equals(sum, a + b);
    }
}
```

inputs
randômicos

100

$10^2 =$

Tempo =



```
fn sum_two_numbers(a: int, b: int) int {
  const sum = a + b;
  assert(sum >= a);
  assert(sum >= b);

  return sum;
}

fn fuzz_sum_two_numbers() void {
    for (0..10_000) {
        const a = random_int();
        const b = random_int();
        const sum = sum_two_numbers(a, b);
        expect_equals(sum, a + b);
    }
}
```

mímica da
regra de negócio

100

10^2 =

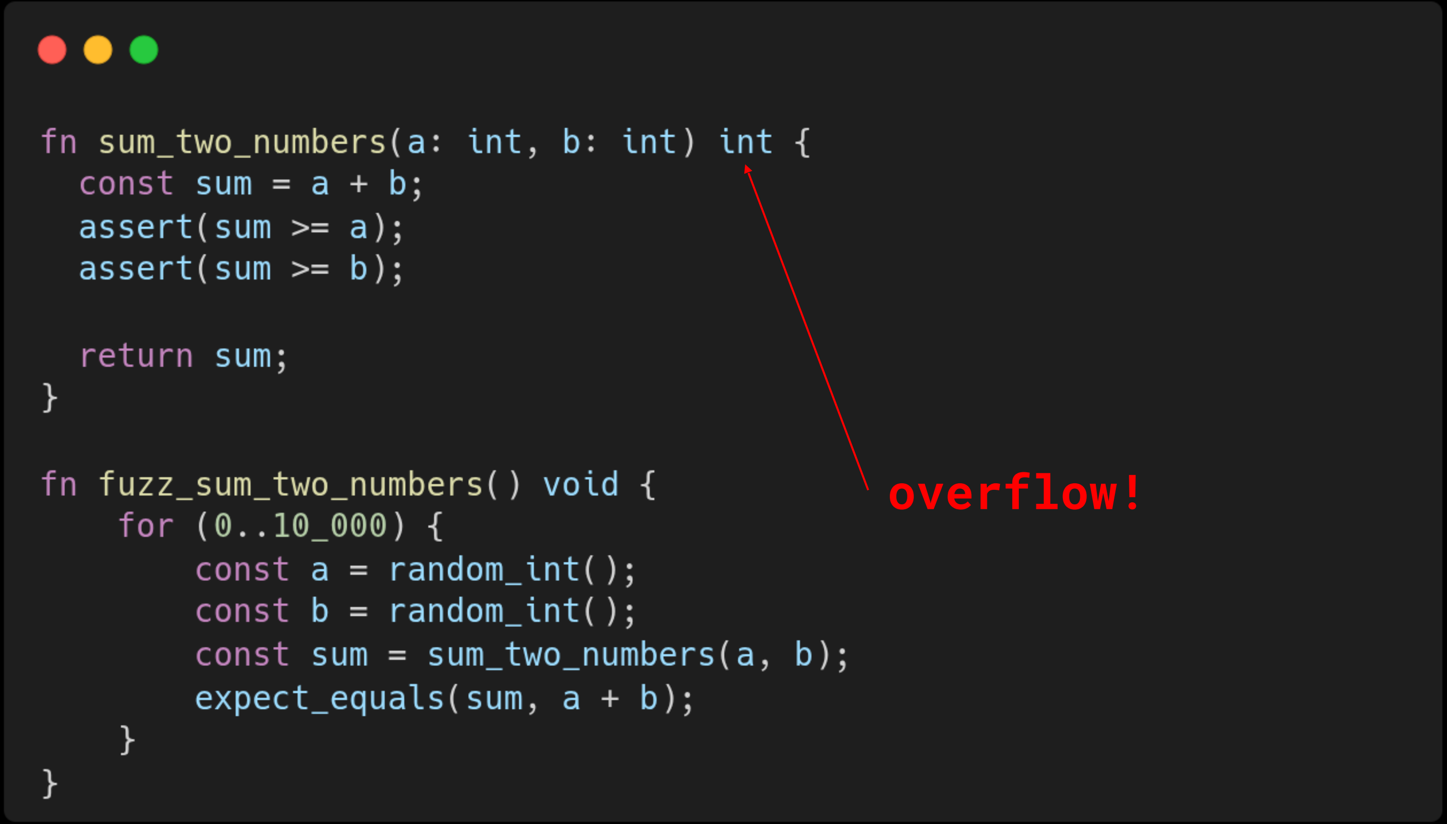
Tempo

```
fn sum_tw
```

100

$10^2 =$

Tempo =



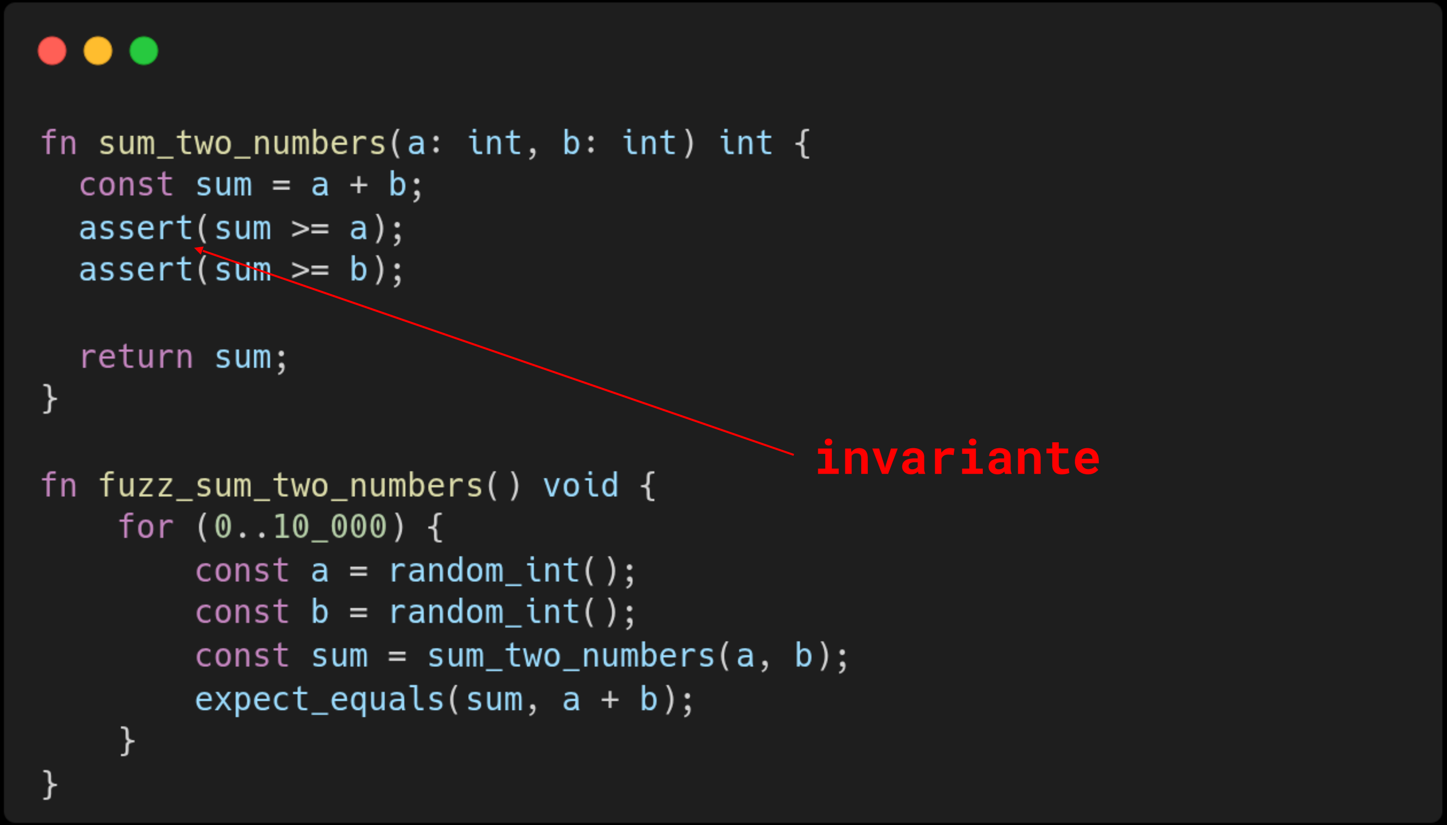
```
fn sum_two_numbers(a: int, b: int) int {
  const sum = a + b;
  assert(sum >= a);
  assert(sum >= b);

  return sum;
}

fn fuzz_sum_two_numbers() void {
    for (0..10_000) {
        const a = random_int();
        const b = random_int();
        const sum = sum_two_numbers(a, b);
        expect_equals(sum, a + b);
    }
}
```

overflow!

100

10^2 = I

Tempo = [



```
fn sum_two_numbers(a: int, b: int) int {
  const sum = a + b;
  assert(sum >= a);
  assert(sum >= b);

  return sum;
}

fn fuzz_sum_two_numbers() void {
    for (0..10_000) {
        const a = random_int();
        const b = random_int();
        const sum = sum_two_numbers(a, b);
        expect_equals(sum, a + b);
    }
}
```

invariante

1000X

10^2 = Esforço *vs* Resultado

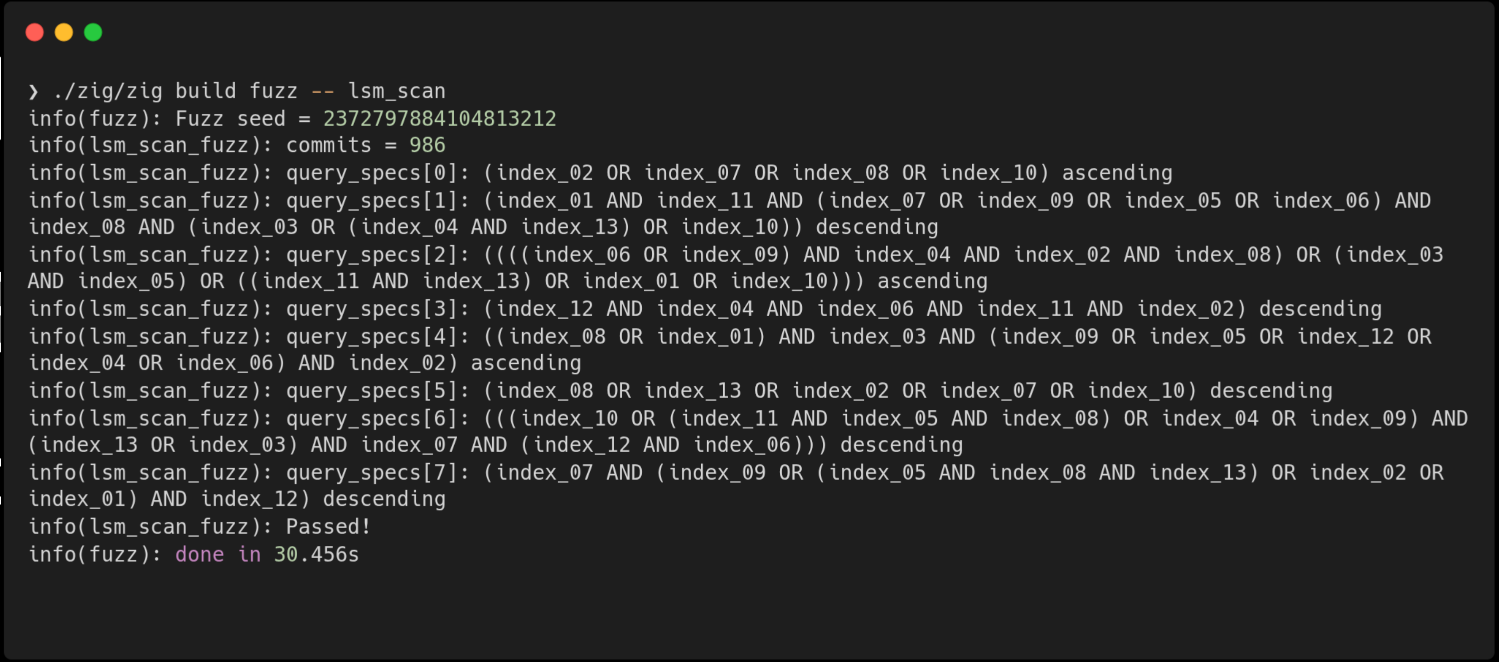
Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic Simulation Testing

100

$10^2 =$

Tempo =



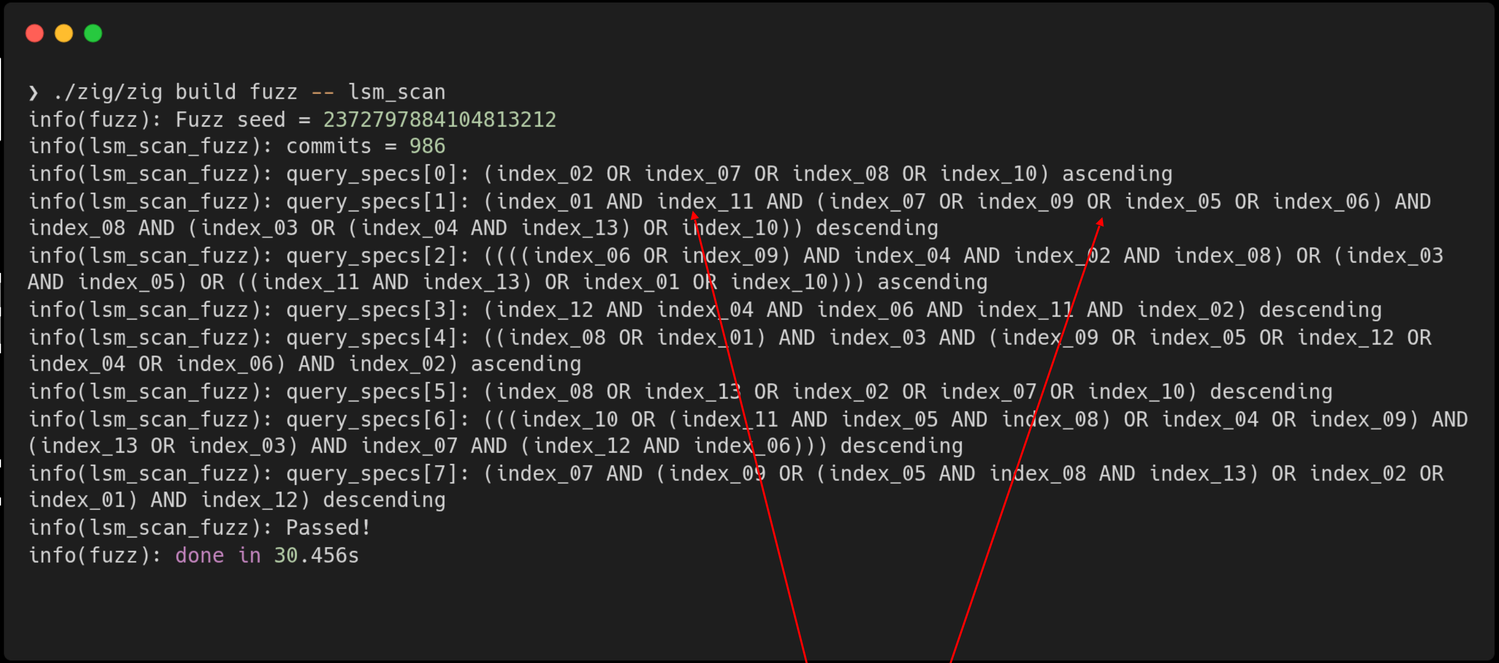
```
❯ ./zig/zig build fuzz -- lsm_scan
info(fuzz): Fuzz seed = 2372797884104813212
info(lsm_scan_fuzz): commits = 986
info(lsm_scan_fuzz): query_specs[0]: (index_02 OR index_07 OR index_08 OR index_10) ascending
info(lsm_scan_fuzz): query_specs[1]: (index_01 AND index_11 AND (index_07 OR index_09 OR index_05 OR index_06) AND
index_08 AND (index_03 OR (index_04 AND index_13) OR index_10)) descending
info(lsm_scan_fuzz): query_specs[2]: ((((index_06 OR index_09) AND index_04 AND index_02 AND index_08) OR (index_03
AND index_05) OR ((index_11 AND index_13) OR index_01 OR index_10))) ascending
info(lsm_scan_fuzz): query_specs[3]: (index_12 AND index_04 AND index_06 AND index_11 AND index_02) descending
info(lsm_scan_fuzz): query_specs[4]: ((index_08 OR index_01) AND index_03 AND (index_09 OR index_05 OR index_12 OR
index_04 OR index_06) AND index_02) ascending
info(lsm_scan_fuzz): query_specs[5]: (index_08 OR index_13 OR index_02 OR index_07 OR index_10) descending
info(lsm_scan_fuzz): query_specs[6]: (((index_10 OR (index_11 AND index_05 AND index_08) OR index_04 OR index_09) AND
(index_13 OR index_03) AND index_07 AND (index_12 AND index_06))) descending
info(lsm_scan_fuzz): query_specs[7]: (index_07 AND (index_09 OR (index_05 AND index_08 AND index_13) OR index_02 OR
index_01) AND index_12) descending
info(lsm_scan_fuzz): Passed!
info(fuzz): done in 30.456s
```

Deterministic Simulation Testing

100

10^2 = I

Tempo = D



```
❯ ./zig/zig build fuzz -- lsm_scan
info(fuzz): Fuzz seed = 2372797884104813212
info(lsm_scan_fuzz): commits = 986
info(lsm_scan_fuzz): query_specs[0]: (index_02 OR index_07 OR index_08 OR index_10) ascending
info(lsm_scan_fuzz): query_specs[1]: (index_01 AND index_11 AND (index_07 OR index_09 OR index_05 OR index_06) AND
index_08 AND (index_03 OR (index_04 AND index_13) OR index_10)) descending
info(lsm_scan_fuzz): query_specs[2]: ((((index_06 OR index_09) AND index_04 AND index_02 AND index_08) OR (index_03
AND index_05) OR ((index_11 AND index_13) OR index_01 OR index_10))) ascending
info(lsm_scan_fuzz): query_specs[3]: (index_12 AND index_04 AND index_06 AND index_11 AND index_02) descending
info(lsm_scan_fuzz): query_specs[4]: ((index_08 OR index_01) AND index_03 AND (index_09 OR index_05 OR index_12 OR
index_04 OR index_06) AND index_02) ascending
info(lsm_scan_fuzz): query_specs[5]: (index_08 OR index_13 OR index_02 OR index_07 OR index_10) descending
info(lsm_scan_fuzz): query_specs[6]: (((index_10 OR (index_11 AND index_05 AND index_08) OR index_04 OR index_09) AND
(index_13 OR index_03) AND index_07 AND (index_12 AND index_06))) descending
info(lsm_scan_fuzz): query_specs[7]: (index_07 AND (index_09 OR (index_05 AND index_08 AND index_13) OR index_02 OR
index_01) AND index_12) descending
info(lsm_scan_fuzz): Passed!
info(fuzz): done in 30.456s
```

Deterministic Simulation Testing

condições válidas
aleatórias

1000X

10^2 = Esforço *vs* Resultado

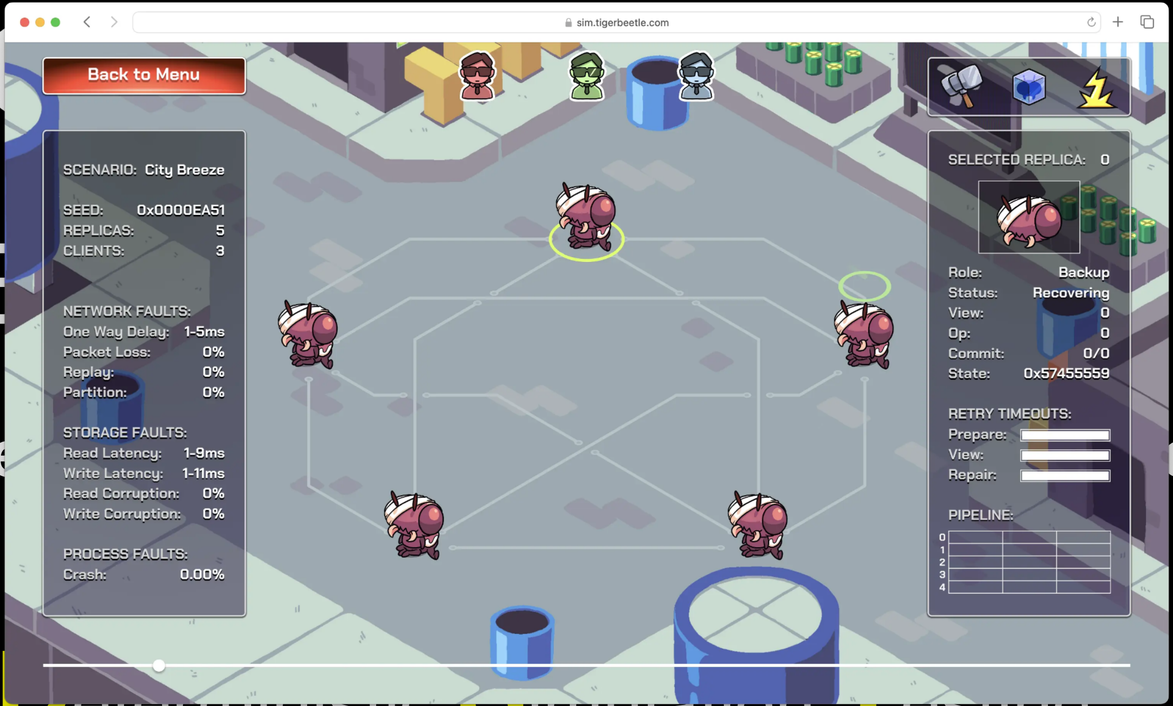
Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic **S**imulation **T**esting

100

10^2 = E

Tempo = De cidentes



SCENARIO: City Breeze

SEED:	0x0000EA51
REPLICAS:	5
CLIENTS:	3

NETWORK FAULTS:

One Way Delay:	1-5ms
Packet Loss:	0%
Replay:	0%
Partition:	0%

STORAGE FAULTS:

Read Latency:	1-9ms
Write Latency:	1-11ms
Read Corruption:	0%
Write Corruption:	0%

PROCESS FAULTS:

Crash:	0.00%

SELECTED REPLICA: 0

Role:	Backup
Status:	Recovering
View:	0
Op:	0
Commit:	0/0
State:	0x57455559

RETRY TIMEOUTS:

Prepare:

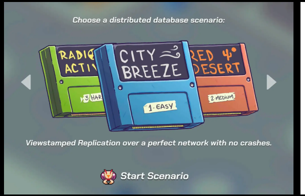
View:

Repair:

PIPELINE:

Deterministic Simulation Testing

https://sim.tigerbeetle.com



Viewstamped Replication over a perfect network with no crashes.

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic **S**imulation **T**esting

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

Deterministic **S**imulation **T**esting

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes

1000x

$10^2 =$

Tempo =

Incidentes

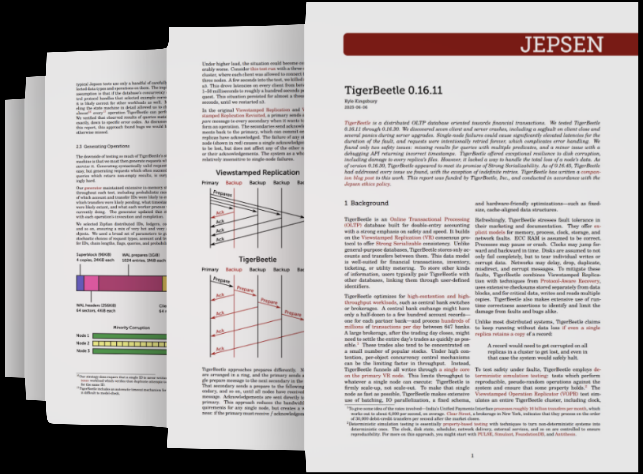
Viewstamped Replication Made Famous

A $20k Consensus Challenge

1000X

10^2 = Esforço *vs* Resultado

Tempo = Design + Desenvolvimento + Testes + Incidentes



JEPSEN

TigerBeetle 0.16.11

Kyle Kingsbury
2025-06-06

TigerBeetle is a distributed OLTP database oriented towards financial transactions. We tested TigerBeetle 0.16.11 through 0.16.30. We discovered seven client and server crashes, including a segfault on client close and several panics during server upgrades. Single-node failures could cause significantly elevated latencies for the duration of the fault, and requests were intentionally retried forever, which complicates error handling. We found only two safety issues: missing results for queries with multiple predicates, and a minor issue with a debugging API returning incorrect timestamps. TigerBeetle offered exceptional resilience to disk corruption, including damage to every replica's files. However, it lacked a way to handle the total loss of a node's data. As of version 0.16.30, TigerBeetle appeared to meet its promise of Strong Serializability. As of 0.16.45, TigerBeetle had addressed every issue we found, with the exception of indefinite retries. TigerBeetle has written a companion blog post to this work. This report was funded by TigerBeetle, Inc., and conducted in accordance with the Jepsen ethics policy.

1 Background

TigerBeetle is an Online Transactional Processing (OLTP) database built for double-entry accounting with a strong emphasis on safety and speed. It builds on the Viewstamped Replication (VR) consensus protocol to offer Strong Serializable consistency. Unlike general-purpose databases, TigerBeetle stores only accounts and transfers between them. This data model is well-suited for financial transactions, inventory, ticketing, or utility metering. To store other kinds of information, users typically pair TigerBeetle with other databases, linking them through user-defined identifiers.

TigerBeetle optimizes for high-contention and high-throughput workloads, such as central bank switches or brokerages. A central bank exchange might have only a half-dozen to a few hundred account records—one for each partner bank—and process hundreds of millions of transactions per day between 647 banks. A large brokerage, after the trading day closes, might need to settle the entire day's trades as quickly as possible.[1] These trades also tend to be concentrated on a small number of popular stocks. Under high contention, per-object concurrency control mechanisms can be the limiting factor in throughput. Instead, TigerBeetle funnels all writes through a single core on the primary VR node. This limits throughput to whatever a single node can execute: TigerBeetle is firmly scale-up, not scale-out. To make that single node as fast as possible, TigerBeetle makes extensive use of batching, IO parallelization, a fixed schema, and hardware-friendly optimizations—such as fixed-size, cache-aligned data structures.

Refreshingly, TigerBeetle stresses fault tolerance in their marketing and documentation. They offer explicit models for memory, process, clock, storage, and network faults. ECC RAM is assumed to be correct. Processes may pause or crash. Clocks may jump forward and backward in time. Disks are assumed to not only fail completely, but to tear individual writes or corrupt data. Networks may delay, drop, duplicate, misdirect, and corrupt messages. To mitigate these faults, TigerBeetle combines Viewstamped Replication with techniques from Protocol-Aware Recovery, uses extensive checksums stored separately from data blocks, and for critical data, writes and reads multiple copies. TigerBeetle also makes extensive use of runtime correctness assertions to identify and limit the damage from faults and bugs alike.

Unlike most distributed systems, TigerBeetle claims to keep running without data loss if even a single replica retains a copy of a record:

> A record would need to get corrupted on all replicas in a cluster to get lost, and even in that case the system would safely halt.

To test safety under faults, TigerBeetle employs deterministic simulation testing: tests which perform reproducible, pseudo-random operations against the system and ensure that some property holds.[2] The Viewstamped Operation Replicator (VOPR) test simulates an entire TigerBeetle cluster, including clock,

[1] To give some idea of the rates involved—India's Unified Payments Interface processes roughly 16 billion transfers per month, which works out to about 6,000 per second, on average. Clear Street, a brokerage in New York, indicates that they process on the order of 30,000 debit-credit transfers per second after the market closes.

[2] Deterministic simulation testing is essentially property-based testing with techniques to turn non-deterministic systems into deterministic ones. The clock, disk state, scheduler, network delivery, external services, and so on are controlled to ensure reproducibility. For more on this approach, you might start with PULSE, Simulant, FoundationDB, and Antithesis.

1000x

10^2

Tempo **s + Incidentes**

1000x

$10^2 =$ I

Tempo = D

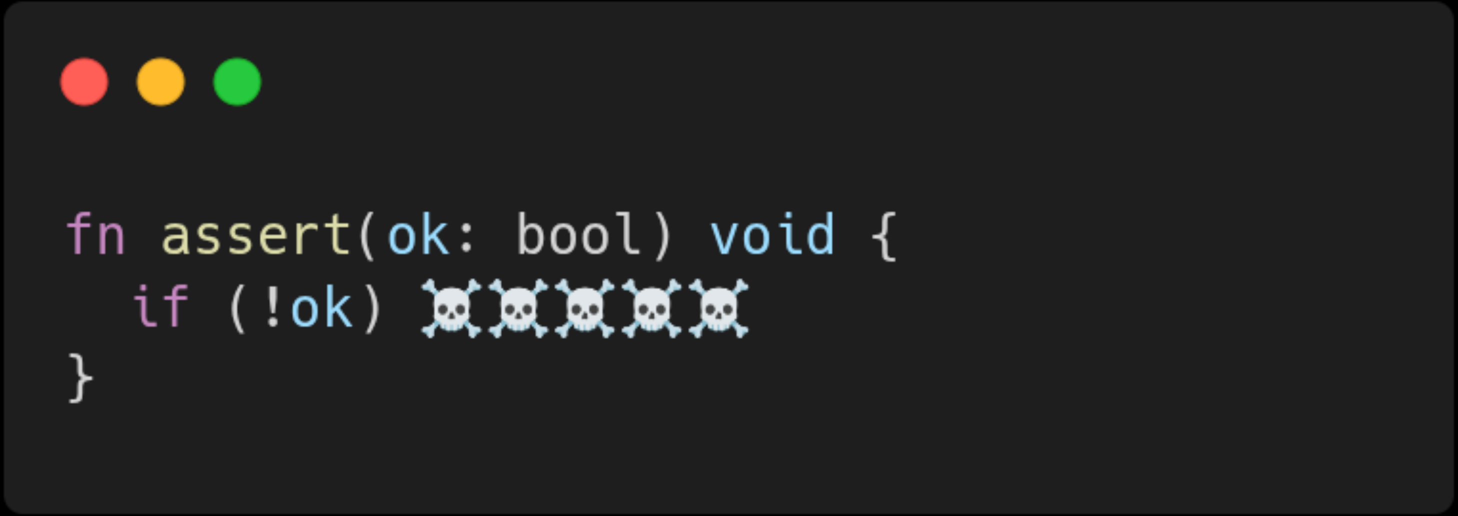
WE COME FROM THE LAND OF THE ICE AND SNOW..
FROM THE MIDNIGHT SUN WHERE THE HOT SPRINGS FLOW..
JOY

Incidentes

1000X

$10^2 =$

Tempo = **s**



```
fn assert(ok: bool) void {
  if (!ok) ☠️☠️☠️☠️☠️
}
```

1000X

10^3 = Build Trust and Have Fun

“Cultivar confiança e se divertir”

1000X

Muito obrigado!

> Perguntas?
> Comentários!

"BTHF"