Couchbase

Automating Database Testing in Your CI/CD Pipelines

Ben Greenberg, Senior Developer Advocate

@hummusonrails


THE DATABASE
STRIKES BACK...
@hummusonrails

The Rebel Alliance needs the

FORCE

Your database needs

TESTING

What we’re going to talk about

- Why test your database?
  What are the unique challenges in DB testing?
- Types of database tests.
- Automating with 
- A word on 

Why test your database?

Do. Or do not.

There is only try.

Testing your database ensures:

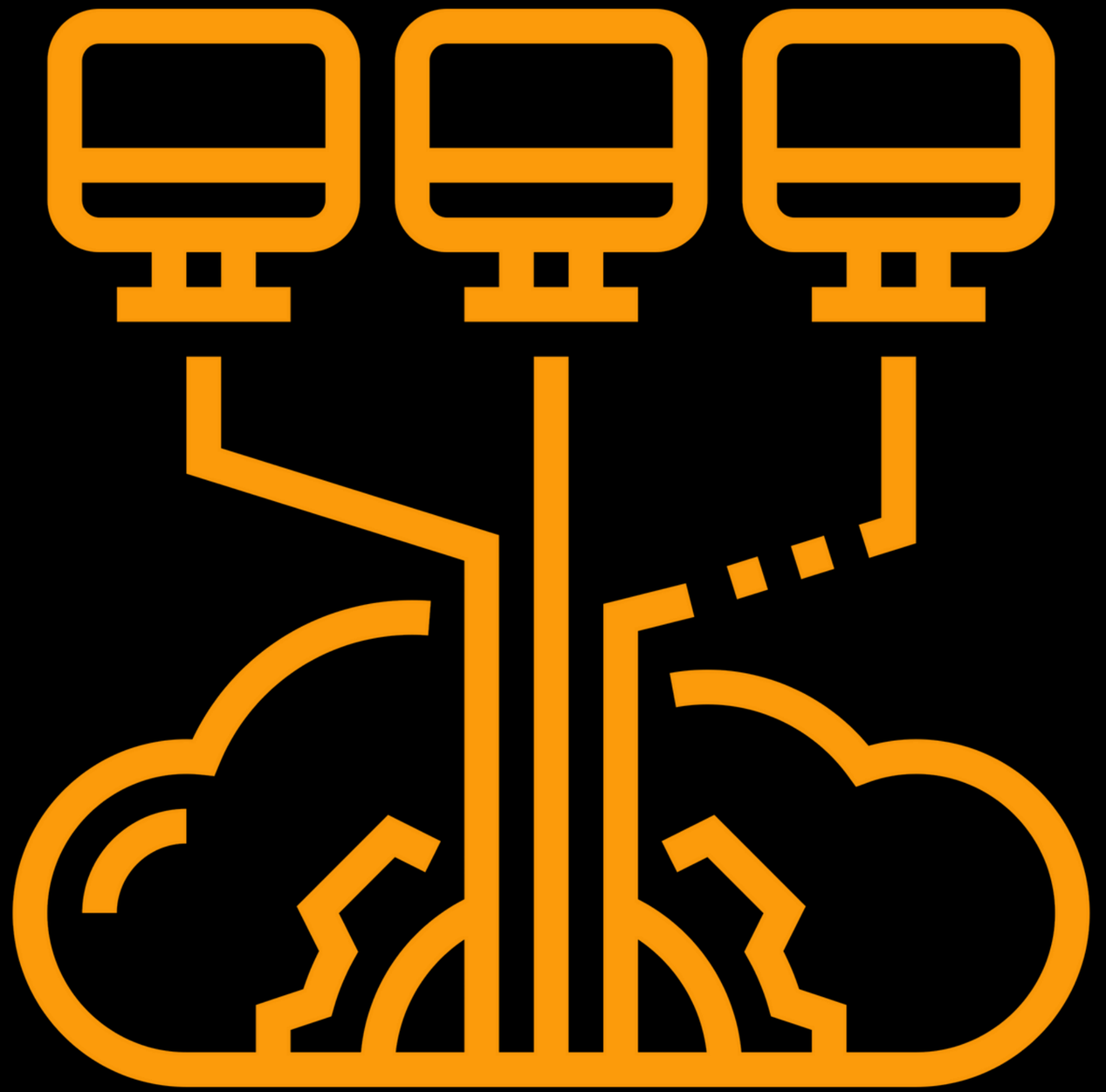
1. **Data integrity**
2. **Data reliability**

It’s a critical part of your software

development lifecycle


Challenges do
exist

Concurrency Issues

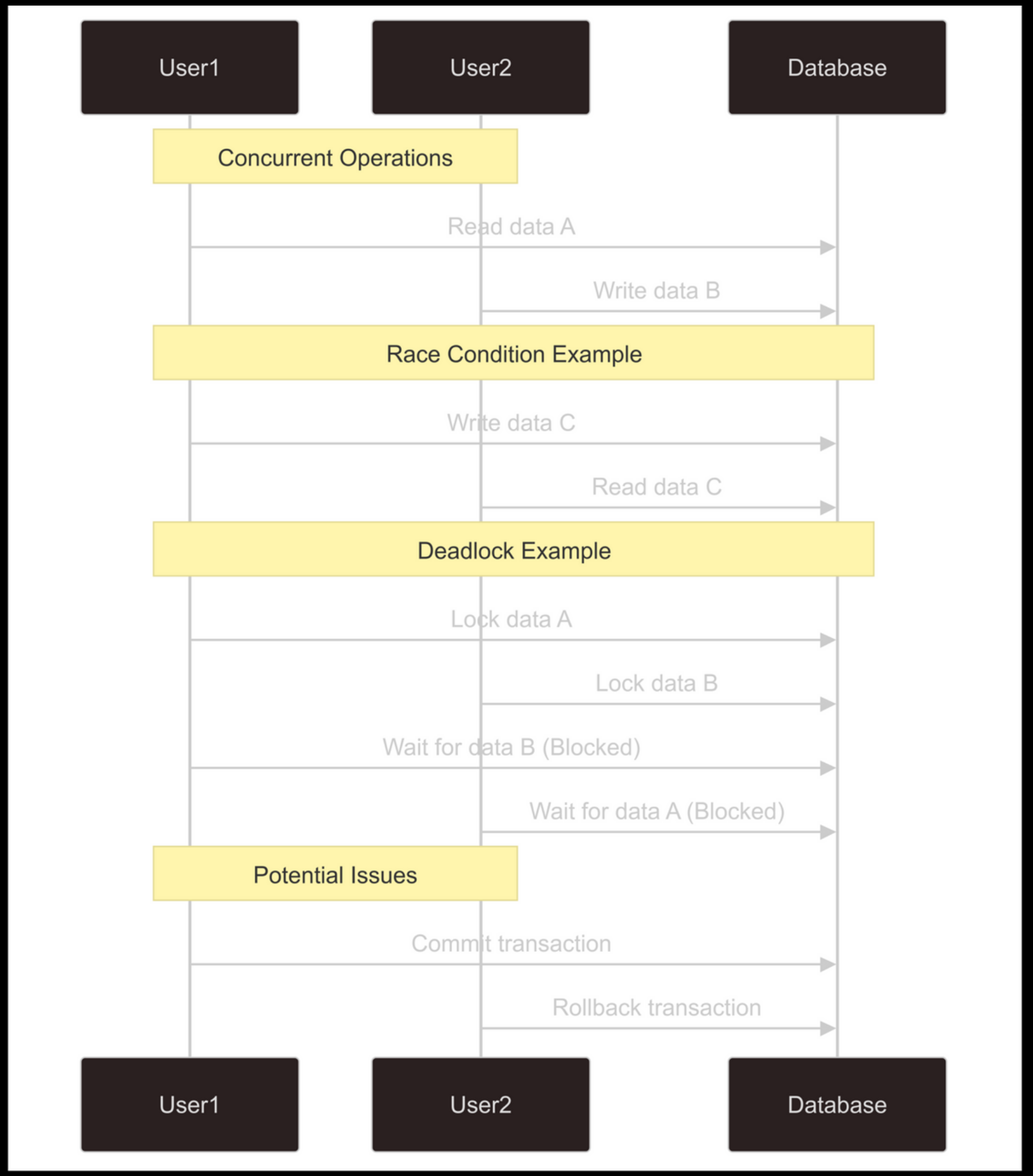


@hummusonrails

What happens when multiple database operations are executed simultaneously?

Challenges

- Deadlocks
- Race conditions
- Performance degradation



Deadlocks

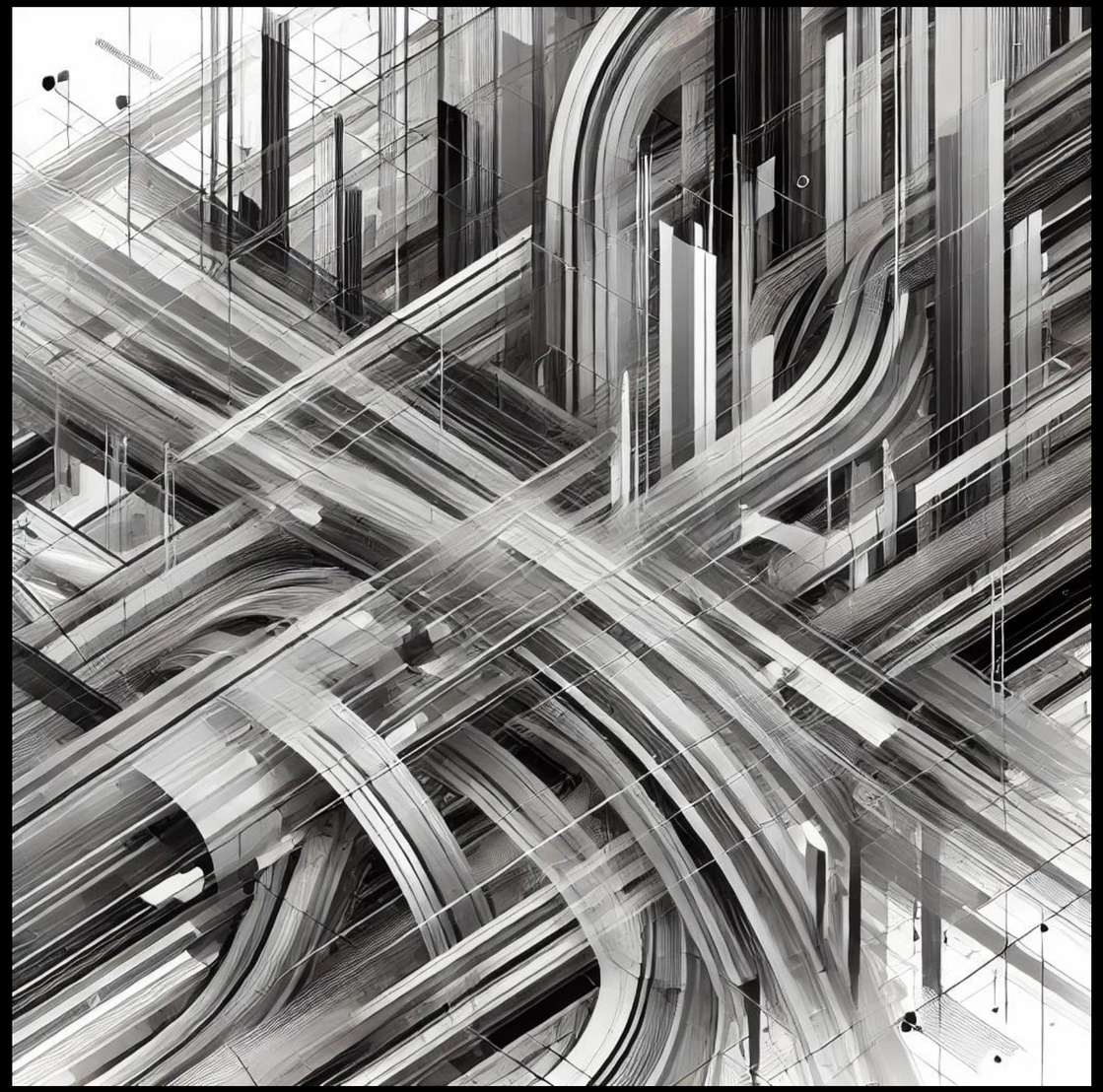
Imagine two transactions:

One locking a customer record and the other locking an order record, both waiting for the other to finish first.

Race Conditions

Imagine two operations:

Two users updating the same inventory count in a shopping site at the same time.



Performance Degradation

It’s Black Friday and the site is under heavy load.

Can it handle the influx of write operations?

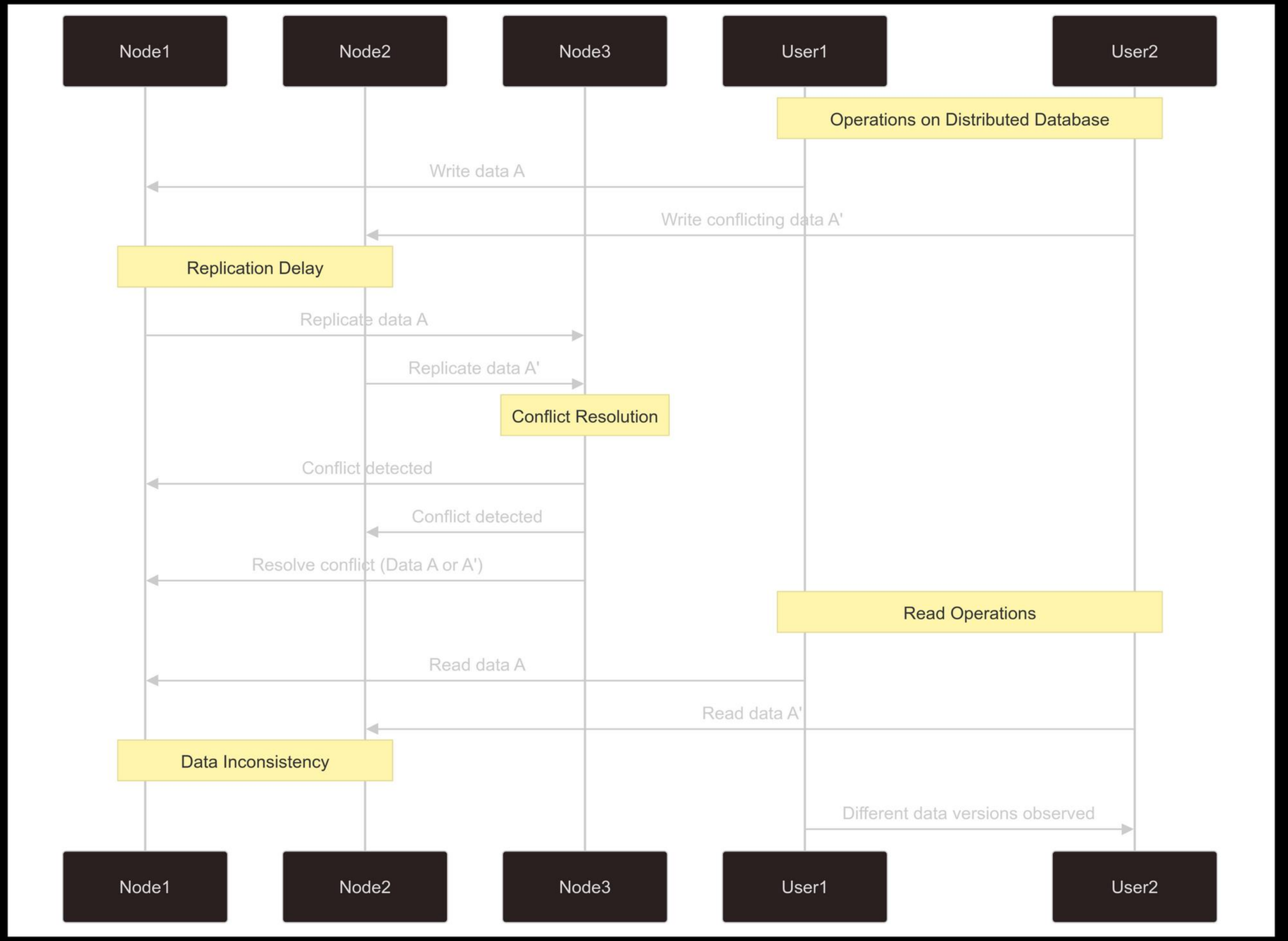
Data Consistency

@hummusonrails

Do all users see the same data being accessed at the same time?

Challenges

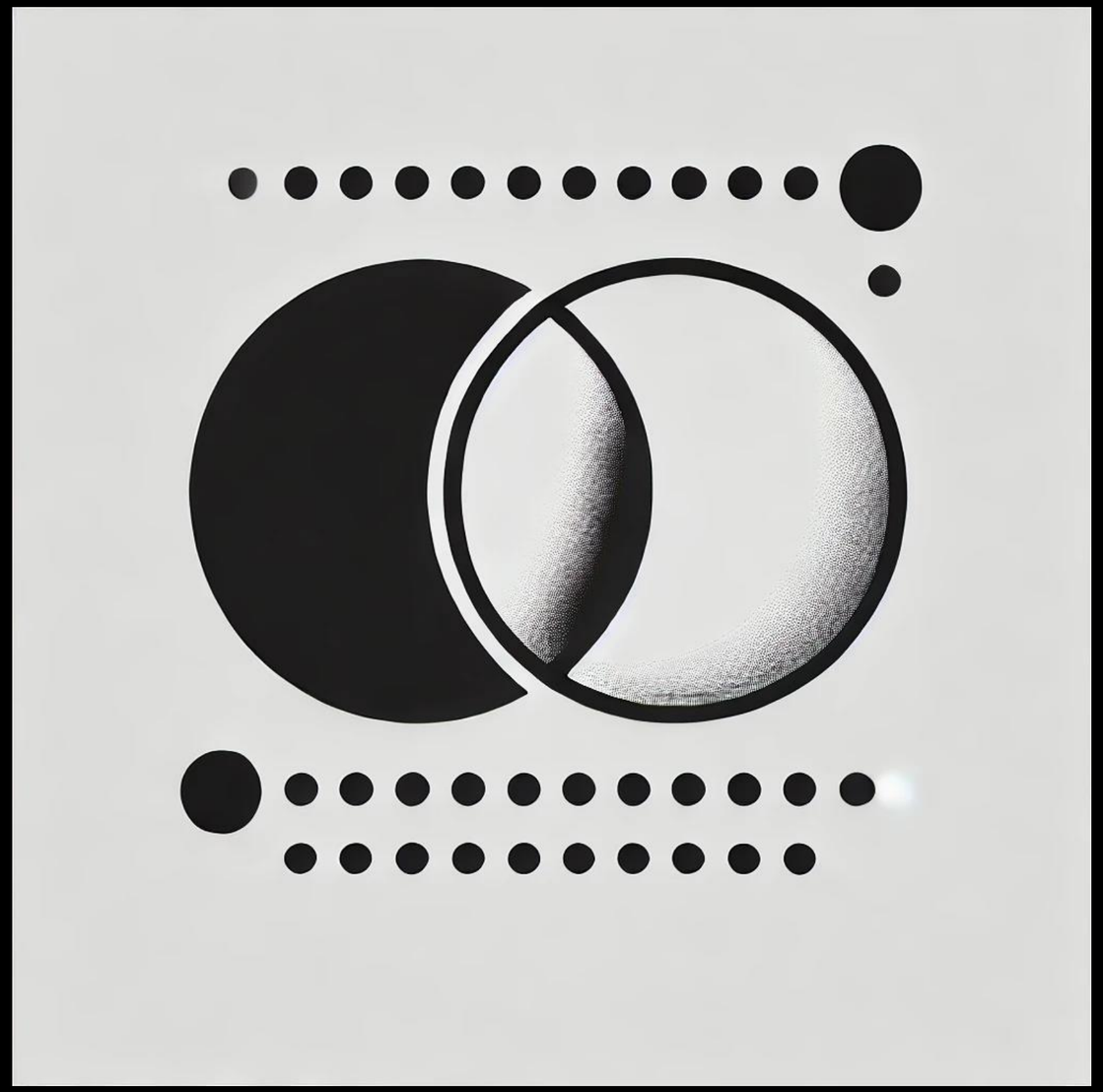
- Read operations
- Write operations
- Replication delays



Replication Delays

User updates their profile.
Data is being replicated
across nodes.

Will User B see User A’s updated profile or the old one?

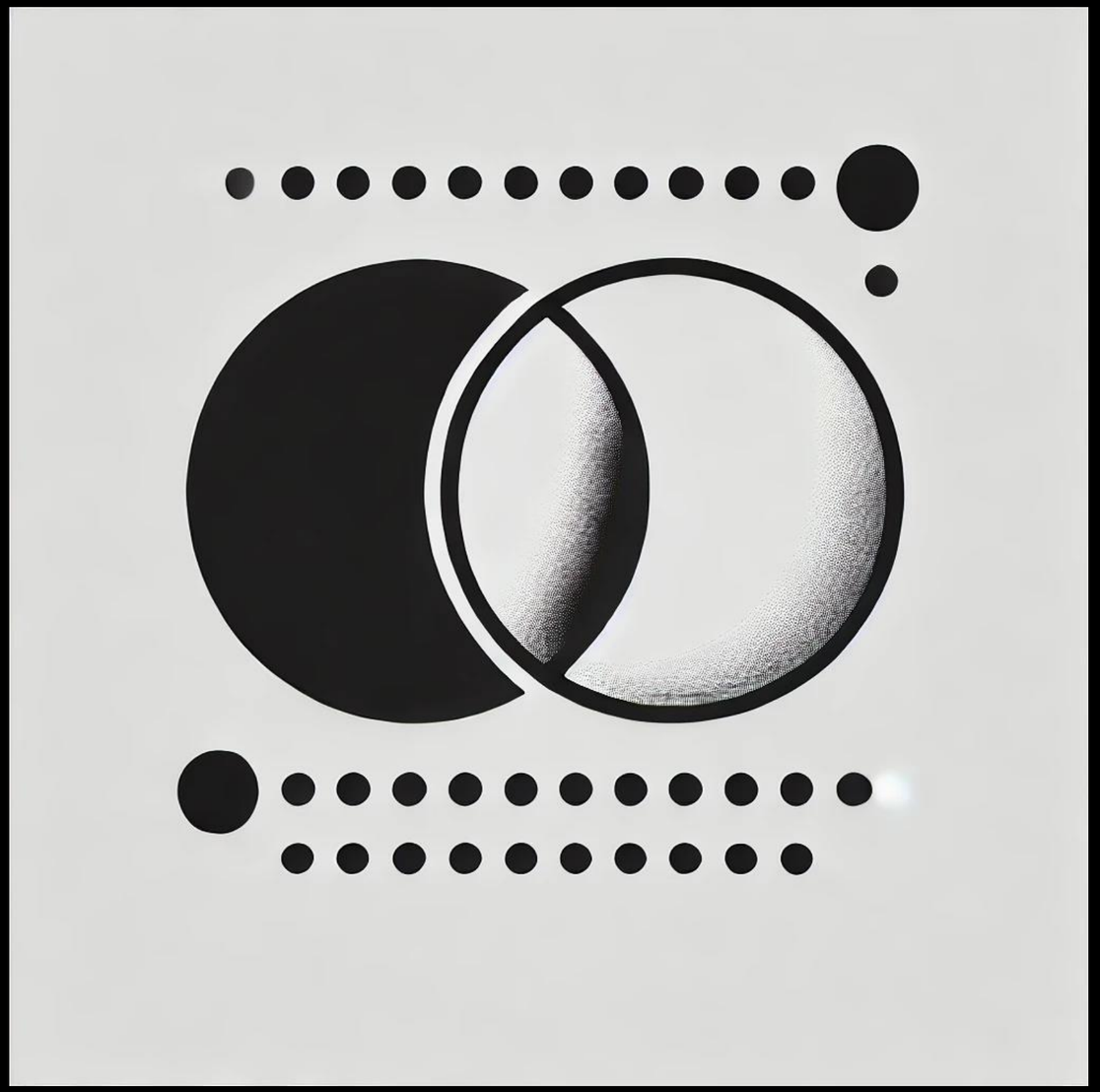


Read/Write Anomalies

Bank account balance shows an old value during a transaction.

Conflict when multiple writes target the same data.

Inconsistent data is read due to ongoing writes.




How do we overcome these challenges?

Types of database tests

Unit Tests

Individual Components

Focus on the different components of the database

Example:

Testing document *upsert* functionality

```
const { expect } = require('chai');
const { CouchbaseContainer } = require('testcontainers');
const couchbase = require('couchbase');

describe('Couchbase Upsert Operation', () => {
  let container, cluster, bucket, collection;

  before(async () => {
    container = await new CouchbaseContainer().start();
    cluster = await couchbase.connect(container.getConnectionString(),
{      username: container.getUsername(),
      password: container.getPassword()
    });
    bucket = cluster.bucket(process.env.COUCHBASE_BUCKET);
    collection = bucket.defaultCollection();
  });

  after(async () => {
    await cluster.disconnect();
    await container.stop();
  });

  it('should upsert a document', async () => {
    const docId = 'user::123';
    const doc = { name: 'John Doe', email: 'john.doe@example.com' };

    await collection.upsert(docId, doc);
    const result = await collection.get(docId);

    expect(result.content).to.deep.equal(doc);
  });
});
```

Integration Tests

Working Together

Ensuring different parts of the database work together

Example:

Testing document flow

```
const { expect } = require('chai');
const { CouchbaseContainer } = require('testcontainers');
const couchbase = require('couchbase');

describe('Couchbase Integration Test', function () {
  this.timeout(60000);

  let container, cluster, bucket, collection;

  before(async () => {
    container = await new CouchbaseContainer().start();
    cluster = await couchbase.connect(container.getConnectionString(), {
      username: container.getUsername(),
      password: container.getPassword()
    });
    bucket = cluster.bucket('default');
    collection = bucket.defaultCollection();
    await collection.upsert('test-doc', { name: 'Integration Test', type: 'example' });
  });

  after(async () => {
    await cluster.disconnect();
    await container.stop();
  });

  it('should retrieve and query the document', async () => {
    const doc = await collection.get('test-doc');
    expect(doc.content).to.deep.equal({ name: 'Integration Test', type: 'example' });

    const queryResult = await cluster.query(`SELECT name, type FROM \`default\` WHERE type =
'example'`);
    expect(queryResult.rows).to.deep.include({ name: 'Integration Test', type: 'example' });
  });
});
```

End-to-end Tests

Is it all working?

Making sure the whole application works **from the user's perspective.**

Example:

Testing an ecommerce platform

```
const { expect } = require('chai');
const { CouchbaseContainer } = require('testcontainers');
const couchbase = require('couchbase');

describe('eCommerce Platform End-to-End Test', function () {
  this.timeout(60000);

  let container, cluster, collection;

  before(async () => {
    container = await new CouchbaseContainer().start();
    cluster = await couchbase.connect(container.getConnectionString(), {
      username: container.getUsername(),
      password: container.getPassword()
    });
    collection = cluster.bucket('default').defaultCollection();
  });

  after(async () => {
    await cluster.disconnect();
    await container.stop();
  });

  it('should create and retrieve customer account', async () => {
    const customerId = 'customer::123';
    const customerData = { name: 'Jane Doe', email: 'jane.doe@example.com', address: '123 Main St' };

    await collection.upsert(customerId, customerData);
    const result = await collection.get(customerId);

    expect(result.content).to.deep.equal(customerData);
    expect(renderCustomerData(result.content)).to.include('Jane
Doe').and.include('jane.doe@example.com').and.include('123 Main St');
  });

  function renderCustomerData(data) {
    return `Customer: ${data.name}, Email: ${data.email}, Address: ${data.address}`;
  }
});
```

Load Tests

Real world load

Identifying performance bottlenecks.

Example:

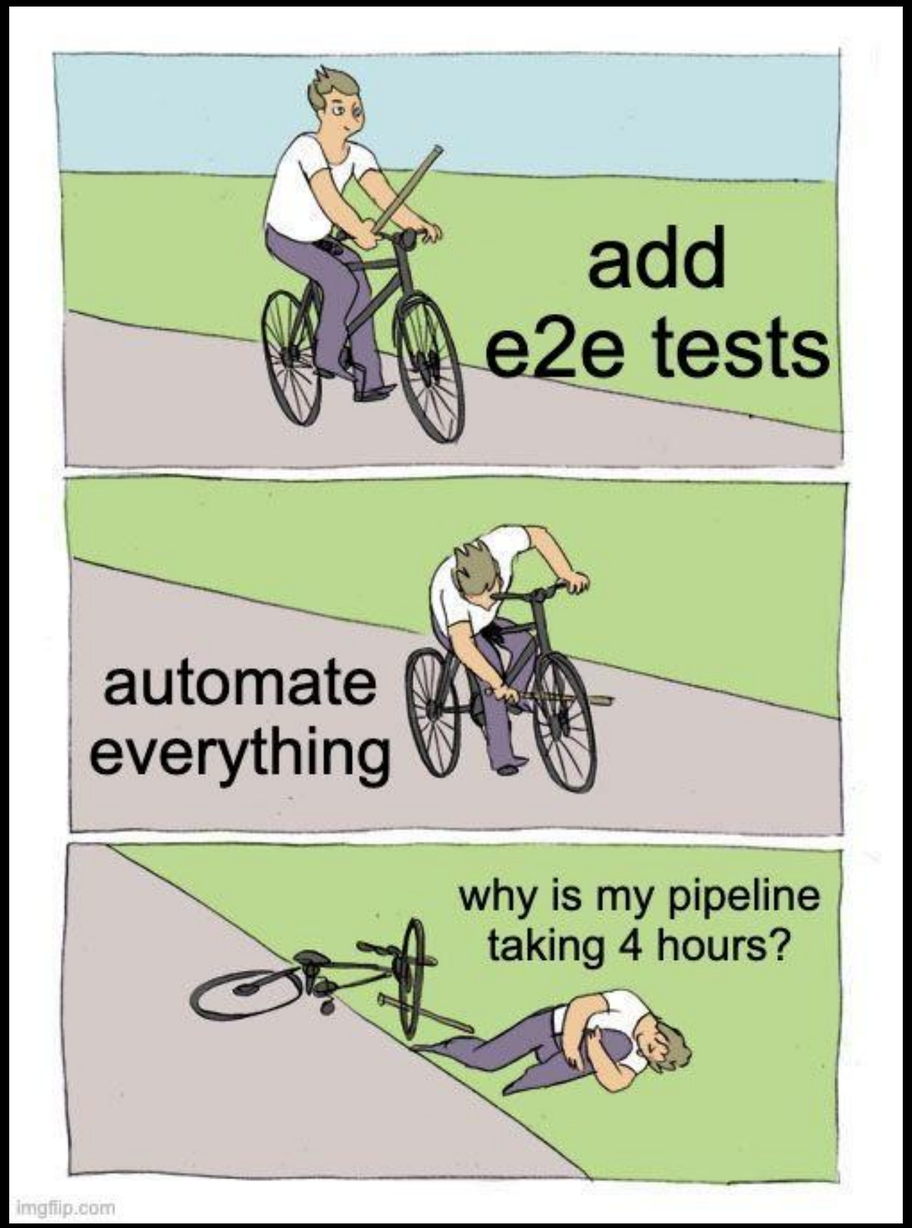
Testing multiple transactions simultaneously

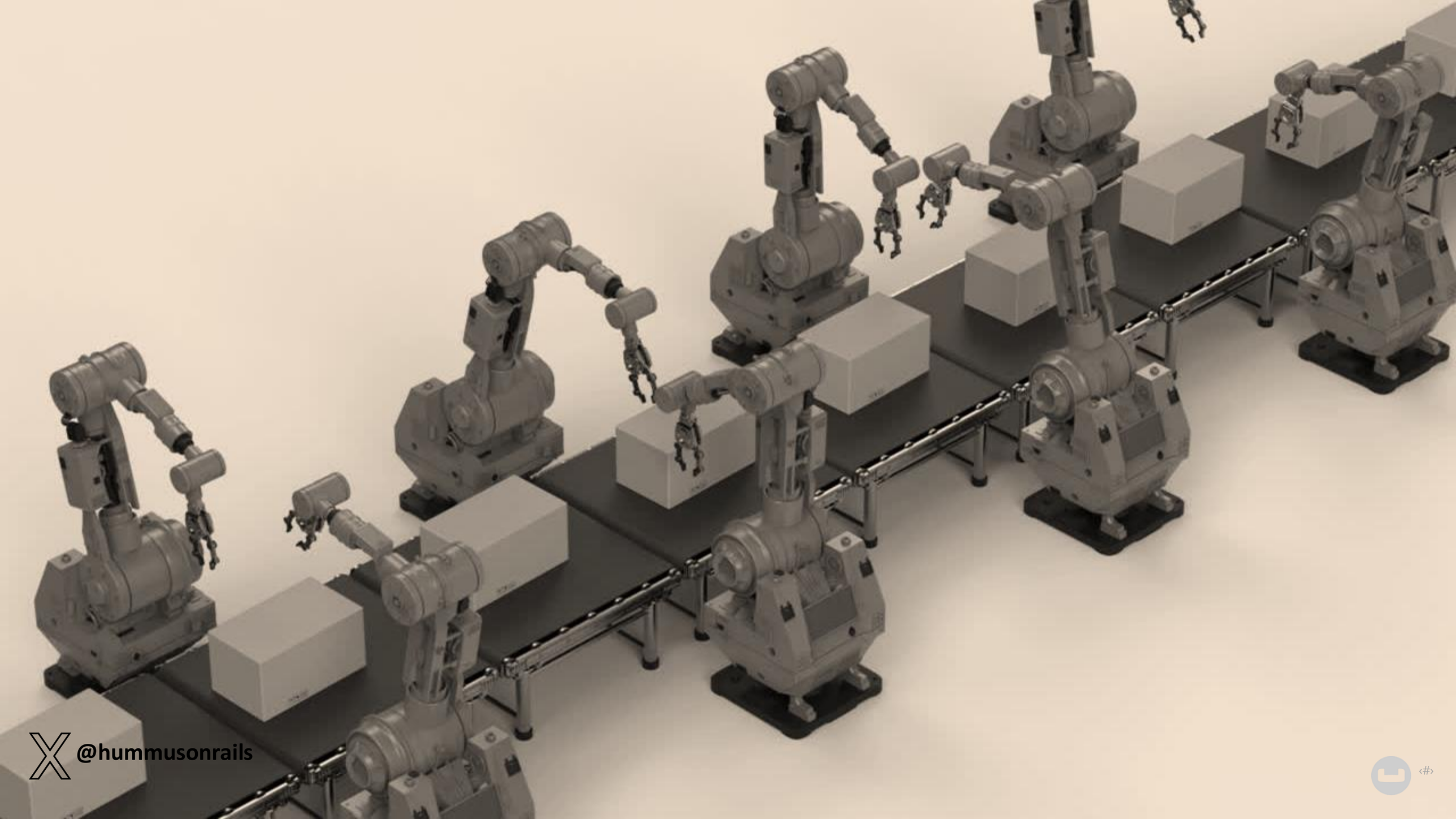
```
const { expect } = require('chai');
const artillery = require('artillery');
const couchbase = require('couchbase');

describe('Couchbase Load Testing', function () {
  this.timeout(30000);

  let cluster, collection;

  before(async () => {
    cluster = await couchbase.connect(process.env.COUCHBASE_CONNECTION_STRING, {
      username: process.env.COUCHBASE_USERNAME,
      password: process.env.COUCHBASE_PASSWORD
    });
    collection = cluster.bucket(process.env.COUCHBASE_BUCKET).defaultCollection();
  });

  after(async () => {
    await cluster.disconnect();
  });

  it('should handle load', async () => {
    const script = {
      config: { target: 'http://localhost:3000', phases: [{ duration: 10, arrivalRate: 5 }] },
      scenarios: [{ flow: [{ function: 'upsertDocument' }] }]
    };

    artillery.run(script, {
      functions: {
        upsertDocument: async function (context, events, done) {
          await collection.upsert(`user::${Math.random()}`, { name: 'John Doe', email: 'john.doe@example.com'
});        done();
        }
      }
    }, (err, result) => {
      expect(err).to.be.null;
      expect(result.failures).to.be.empty;
    });
  });
});
```

Automating Database Testing

Why do it manually?

When you can automate it!





@hummusonrails

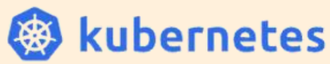

Testcontainers

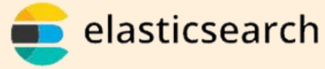
Unit tests with real dependencies

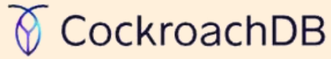
Testcontainers is an open source framework for providing throwaway, lightweight instances of databases, message brokers, web browsers, or just about anything that can run in a Docker container.

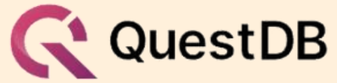
Spin Up Realistic Test Environments for CI/CD Workflows

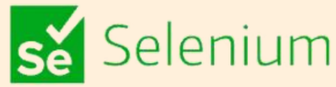
Support for





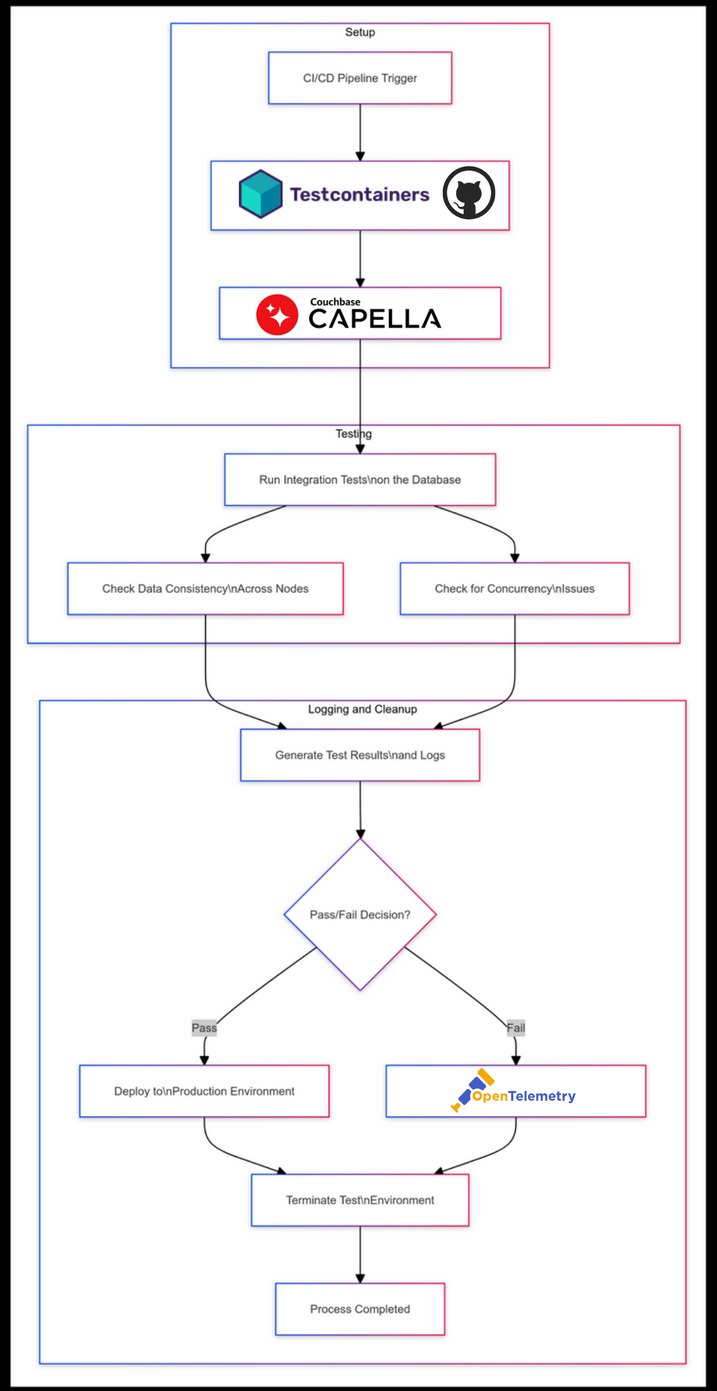




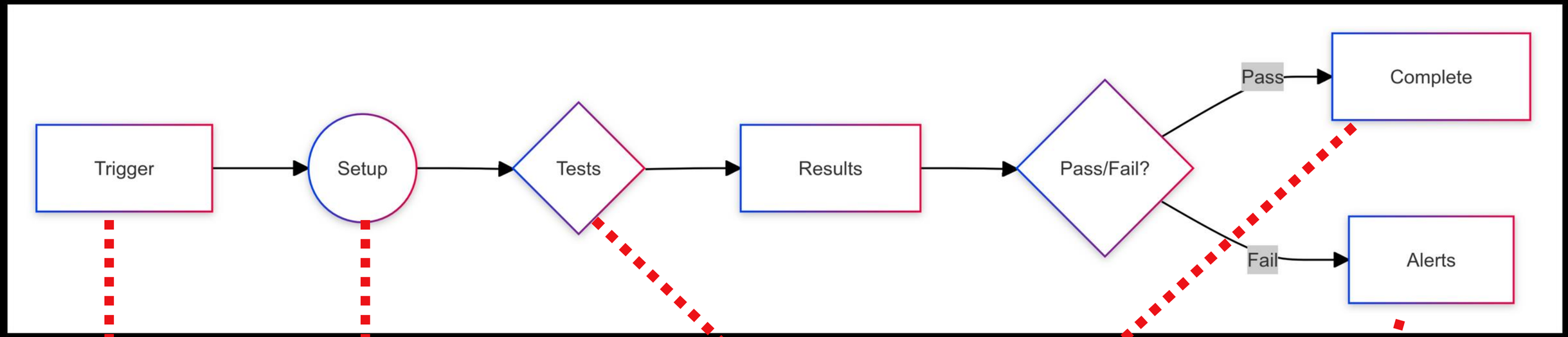



@hummusonrails

the PROCESS



1. CI/CD Trigger
2. Spin Up Environment
3. Initialize DB Connection
4. Run Tests
5. Generate Results
6. Pass/Fail Decision
7. Deploy to Production
8. Send Alerts/Logs
9. Terminate Env
10. Complete Process



- New PR
- New Commit

Testcontainers

Couchbase CAPELLA

- Unit
- Integration
- End-to-end
- Load

- Logging and analytics

OpenTelemetry

Grafana

Prometheus

- Deploy to:

aws

Google Cloud

Azure

netlify

Vercel

@hummusonrails

Unit Test

with  Testcontainers

Include the correct dependency container

Initialize connection to database in BEFORE block

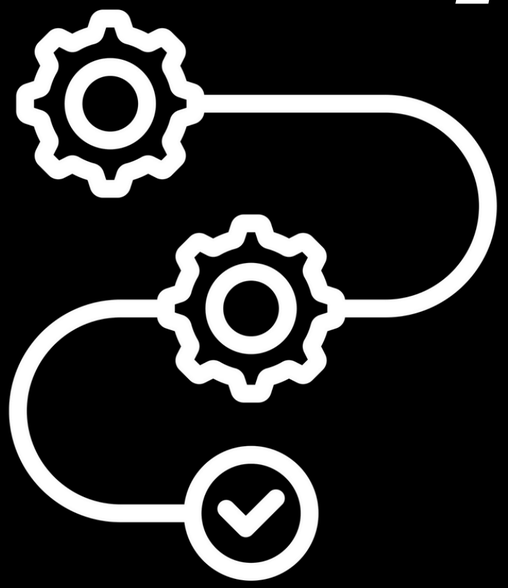
Close connection to database in AFTER block

Run the actual test

```
const { expect } = require('chai');
const { CouchbaseContainer } = require('testcontainers');
const couchbase = require('couchbase');

describe('Couchbase Upsert Operation', () => {
  let container, cluster, bucket, collection;

  before(async () => {
    container = await new CouchbaseContainer().start();
    cluster = await couchbase.connect(container.getConnectionString(),
{     username: container.getUsername(),
      password: container.getPassword()
    });
    bucket = cluster.bucket(process.env.COUCHBASE_BUCKET);
    collection = bucket.defaultCollection();
  });

  after(async () => {
    await cluster.disconnect();
    await container.stop();
  });

  it('should upsert a document', async () => {
    const docId = 'user::123';
    const doc = { name: 'John Doe', email: 'john.doe@example.com' };

    await collection.upsert(docId, doc);
    const result = await collection.get(docId);

    expect(result.content).to.deep.equal(doc);
  });
});
```

@hummusonrails

GitHub Action Setup



GitHub Action Workflow

Define the trigger

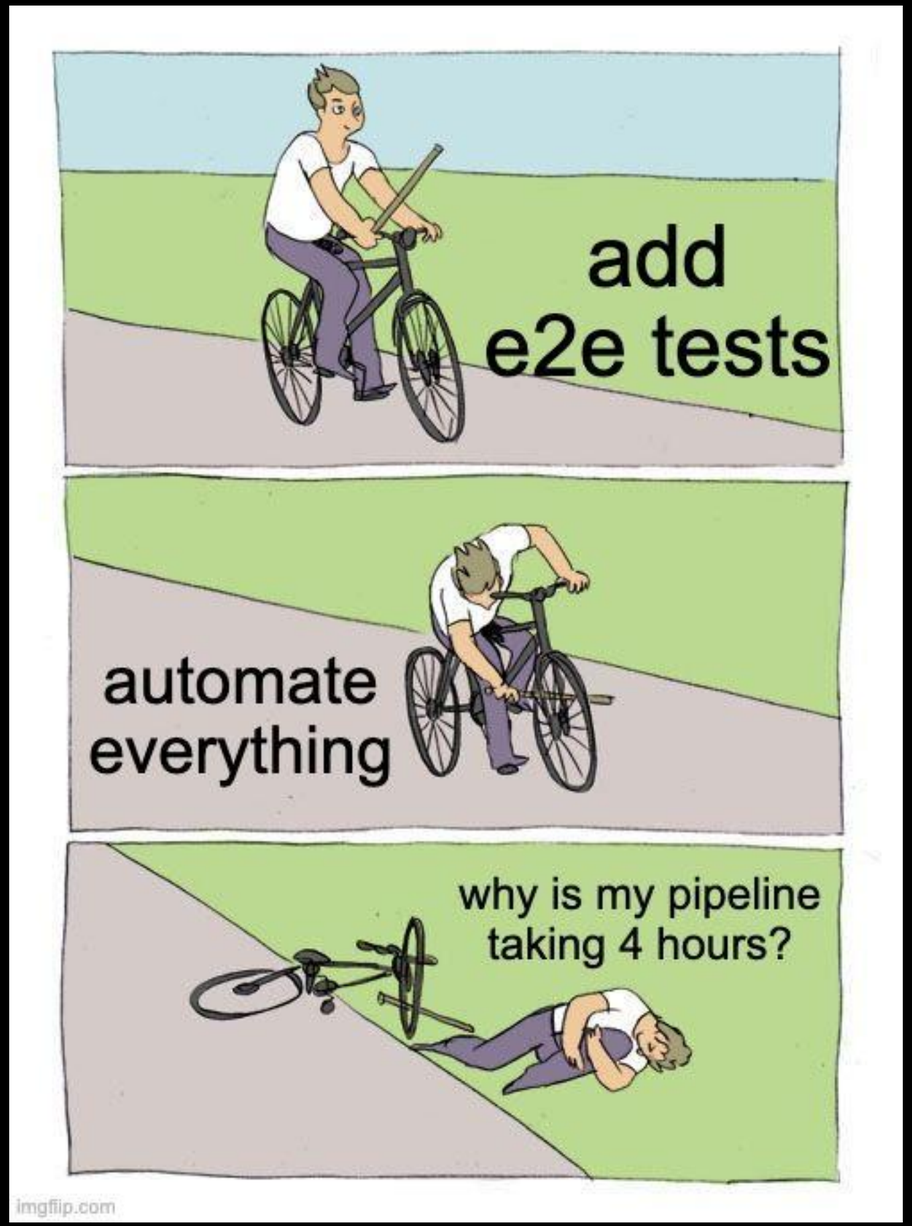
Set up environment:
- **Node.js**
- **Docker**
- **Install dependencies**

Define Actions secrets variables

Run the test

```yaml
name: CI

on:
  pull_request:
  push:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '20'

      - name: Set up Docker
        run: |
          sudo apt-get update
          sudo apt-get install -y docker-compose

      - name: Install dependencies
        run: npm install

      - name: Run tests
        env:
          COUCHBASE_CONNECTION_STRING: ${{ secrets.COUCHBASE_CONNECTION_STRING
}}        COUCHBASE_BUCKET: ${{ secrets.COUCHBASE_BUCKET }}
          COUCHBASE_USER: ${{ secrets.COUCHBASE_USER }}
          COUCHBASE_PASSWORD: ${{ secrets.COUCHBASE_PASSWORD }}
        run: npm test
```

remember THIS?



GitHub Action Workflow

with Testcontainers *Cloud*

Offloading container execution from your test environment

Instead of initializing a Docker container directly inside your GitHub Actions environment

```yaml
name: CI

on:
    pull request:
    push:
        branches: [ main ]

jobs:
    test:
        runs-on: ubuntu-latest

        steps:
            - name: Checkout code
              uses: actions/checkout@4

            - name: Setup up Node.js
              uses: actions/setup-node@v2
              with:
                node-version: '22'

            - name: Setup TestContainers Cloud Client
              uses: atomicjar/testcontainers-cloud-setup-action@main
              with:
                token: ${{ secrets.TC_CLOUD_TOKEN }}

            - name: Run tests
              # Rest of the workflow
```


.github/
workflows/
unit-tests.yml
integration-tests.yml
database-tests.yml
e2e-tests.yml
security-checks.yml
deployment.yml

Test Summary

All tests passed

42 tests passed

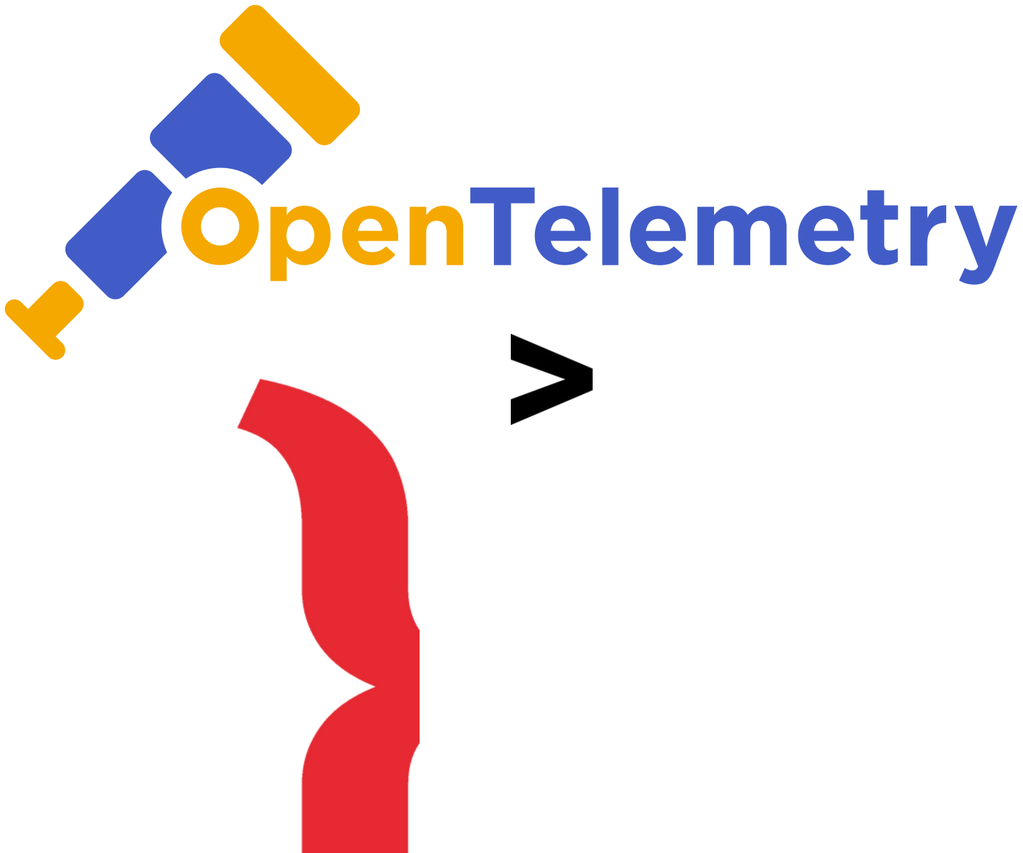
All tests failed

42 tests failed

Some tests failed

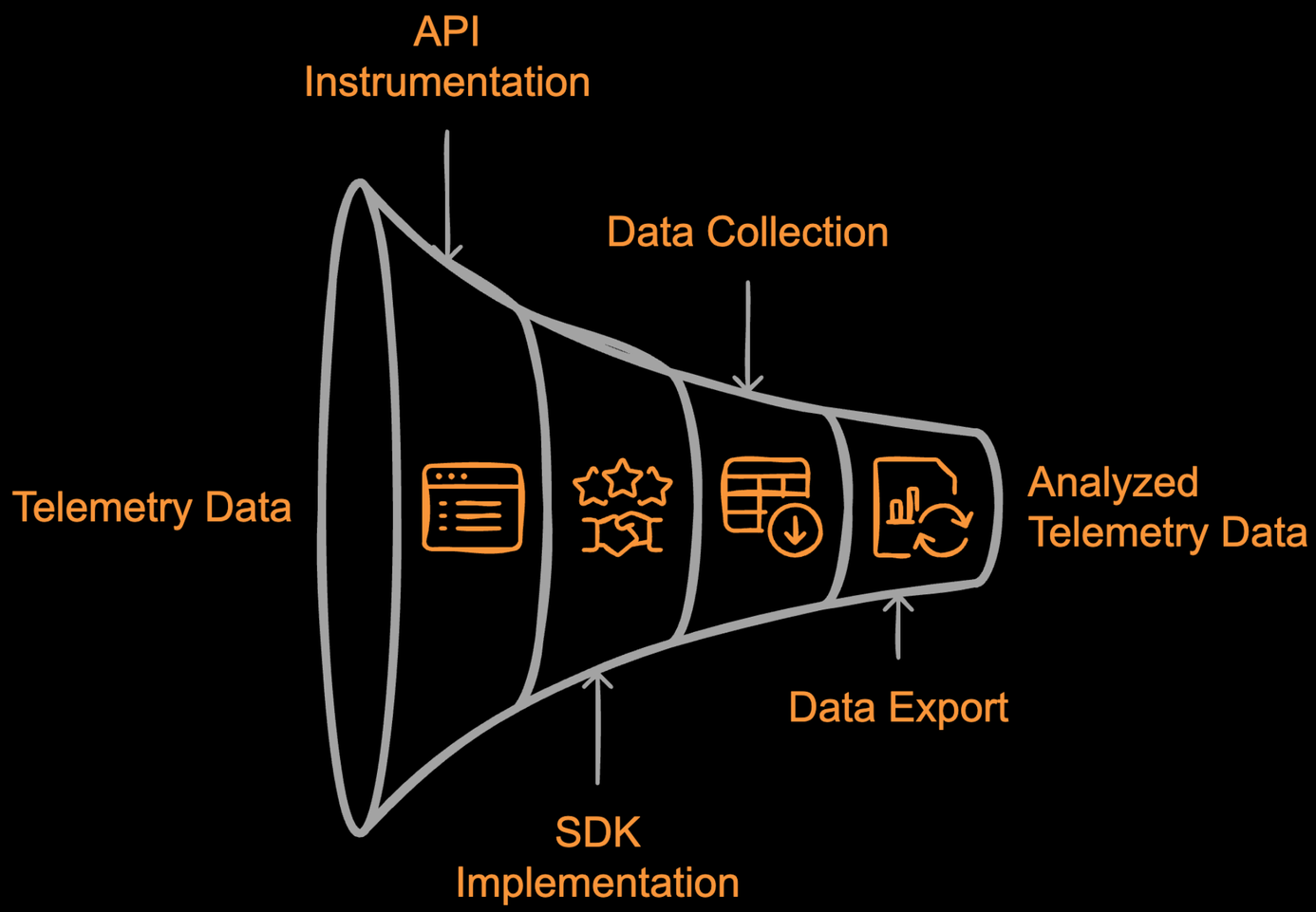
42 tests passed, 8 tests failed, 18 tests skipped

A Note On



>

”OpenTelemetry provides a collection of APIs, SDKs, and tools to standardize telemetry data (metrics, logs, and traces) collection and export ”



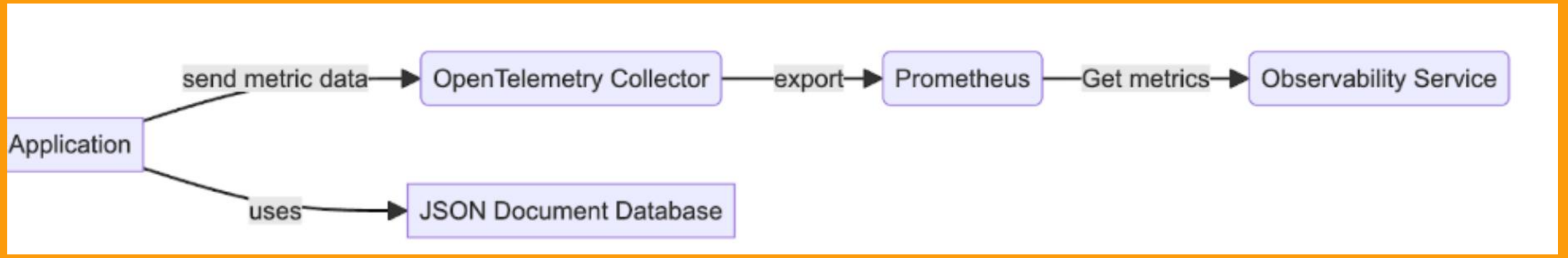
Database Logs and Metrics Improve...

Mean time to detection

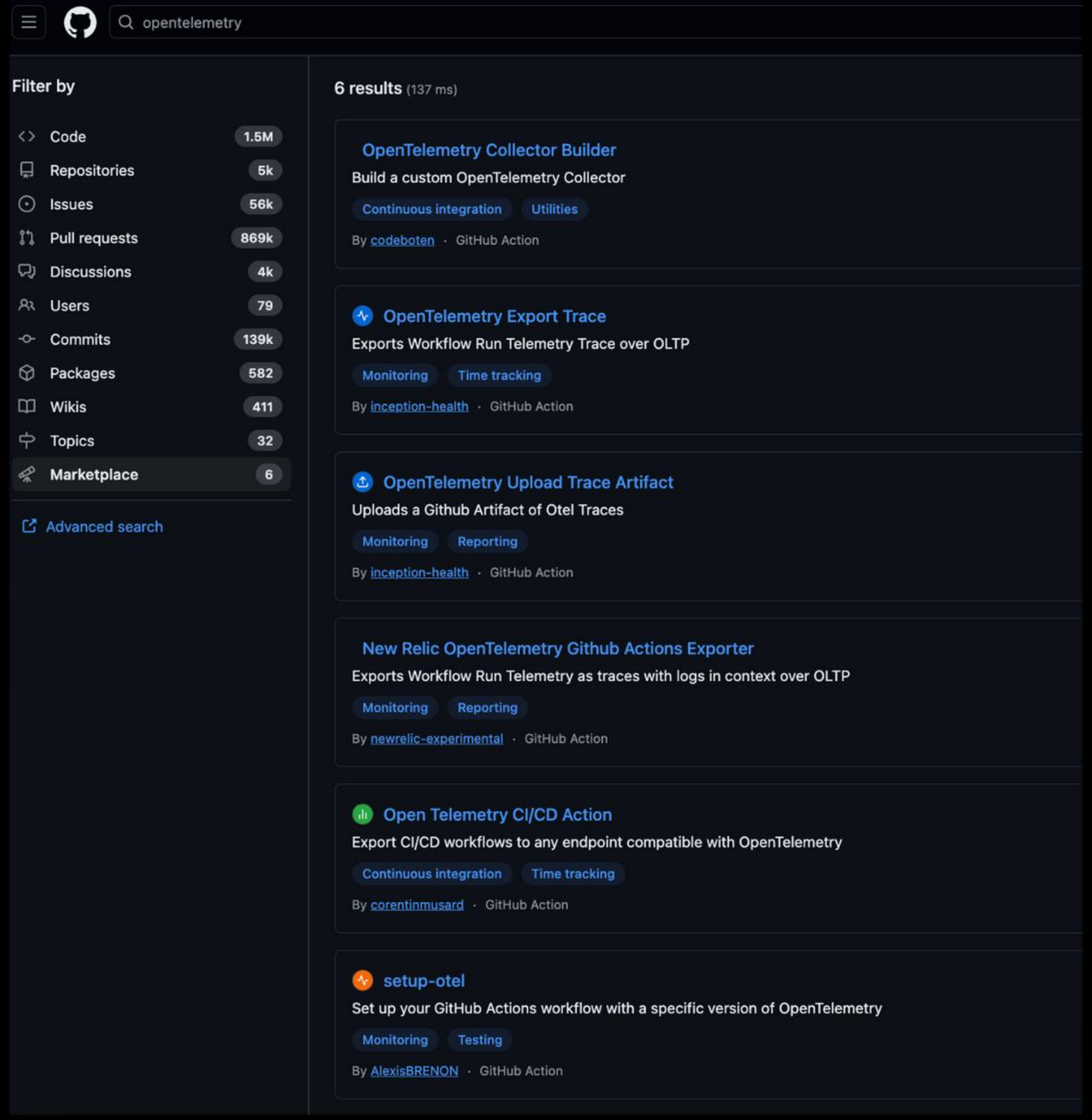
Root cause analysis

Mean time to resolution

Visualizing the Workflow



You don’t have to build it alone…



OTel Workflow

Example:

Using an OTel Exporter Marketplace Action with Database Test Metrics and Traces

```yaml
name: OpenTelemetry Export Metrics

on:
  workflow_run:
    workflows: ["Database Tests"]
    types: [completed]

jobs:
  otel-export-metrics:
    name: OpenTelemetry Export Metrics
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '20'

      - name: Set up Docker
        run: |
          sudo apt-get update
          sudo apt-get install -y docker-compose

      - name: Export Workflow Metrics and Traces
        uses: inception-health/otel-export-trace-action@latest
        with:
          otlpEndpoint: grpc://your-otel-endpoint:443/
          otlpHeaders: ${{ secrets.OTLP_HEADERS }}
          githubToken: ${{ secrets.GITHUB_TOKEN }}
          runId: ${{ github.event.workflow_run.id }}
```

CI/CD Telemetry Data Flow

Actionable Feedback Loop

Insights lead to CI/CD pipeline optimizations

Grafana

Grafana queries Prometheus for dashboard creation

Prometheus

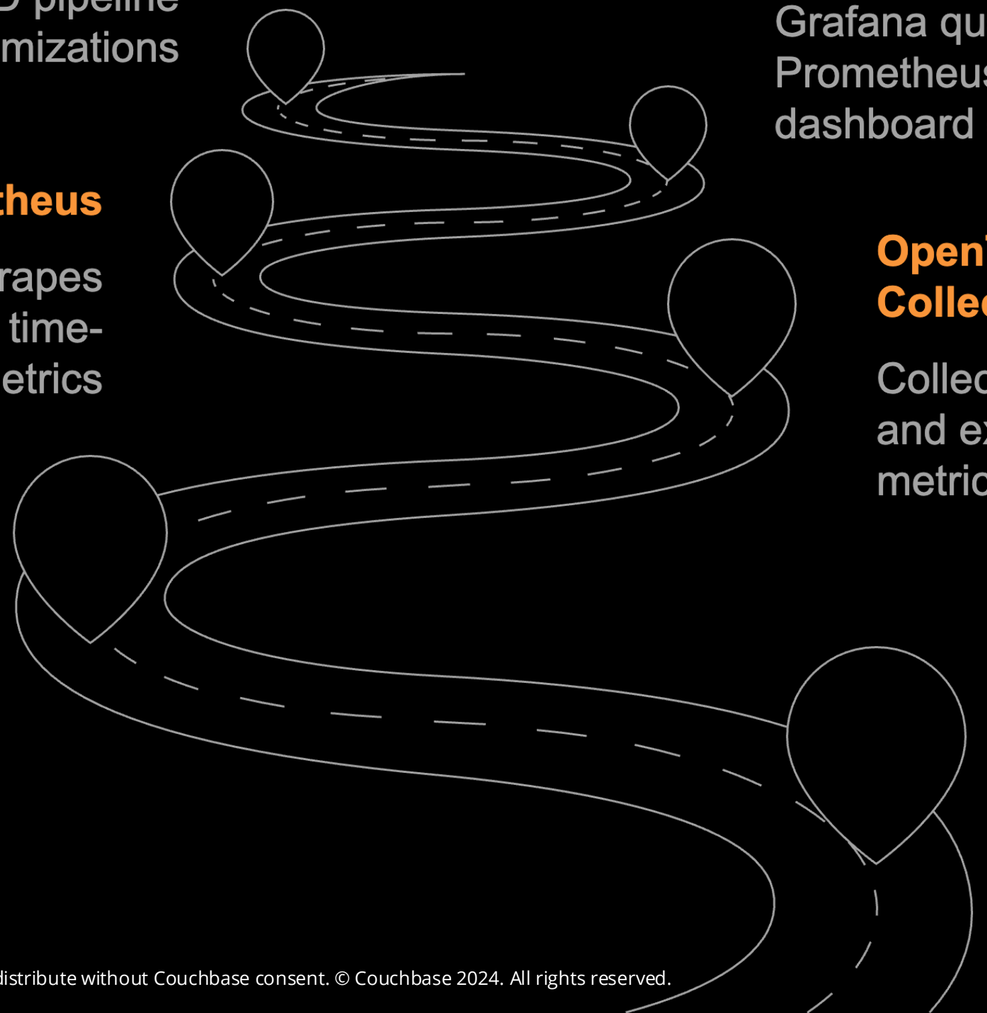
Prometheus scrapes and stores time-series metrics

OpenTelemetry Collector

Collector processes and exports data as metrics

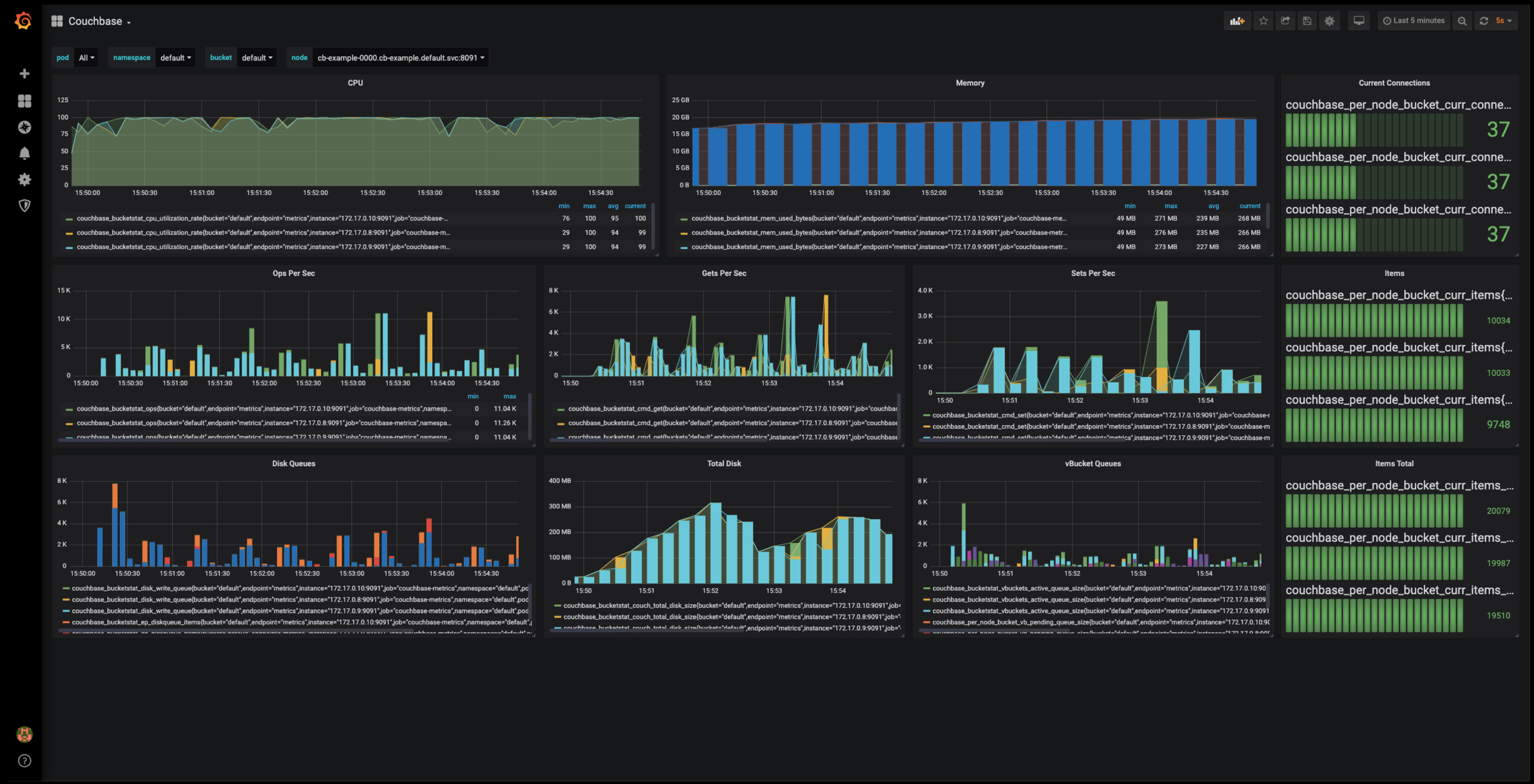
Telemetry Data Flow

Telemetry data flows to the OTel Collector



GitHub Actions CI/CD Workflow

CI/CD jobs emit telemetry data during tasks



@hummusonrails

Want to learn more?

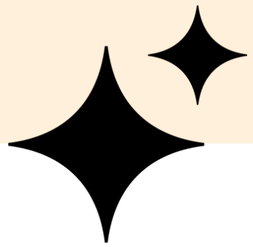
DB Testing with GitHub Actions

Testcontainers in GitHub Actions

JUST AS THE REBEL ALLIANCE TRIUMPHED WITH THE FORCE...

YOUR DATABASE CAN ACHIEVE GREATNESS
WITH THE POWER OF THOROUGH TESTING
AND VIGILANT OBSERVABILITY.

MAY YOUR CI/CD PIPELINE BE EVER IN YOUR FAVOR!



Thank you!

ben.greenberg@couchbase.com

@hummusonrails

Couchbase