metis

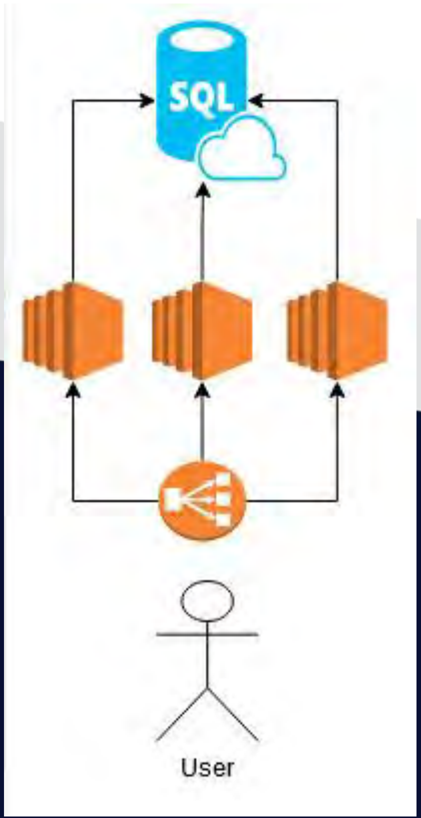
Effective Observability

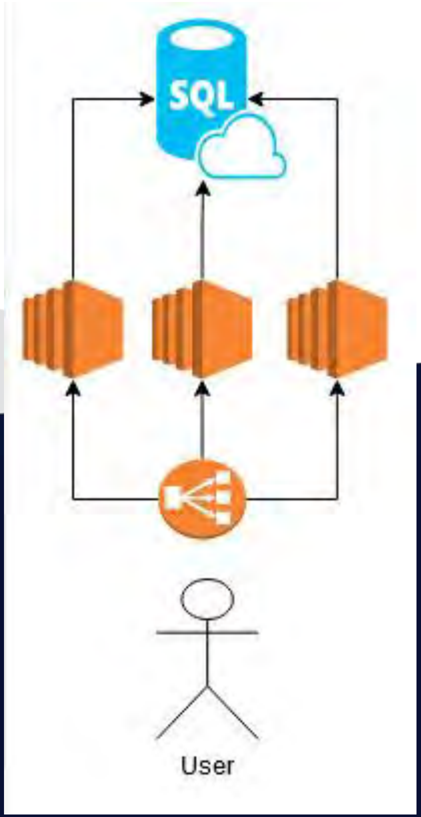
In PostgreSQL, MySQL, and other SQL databases

Adam Furmanek

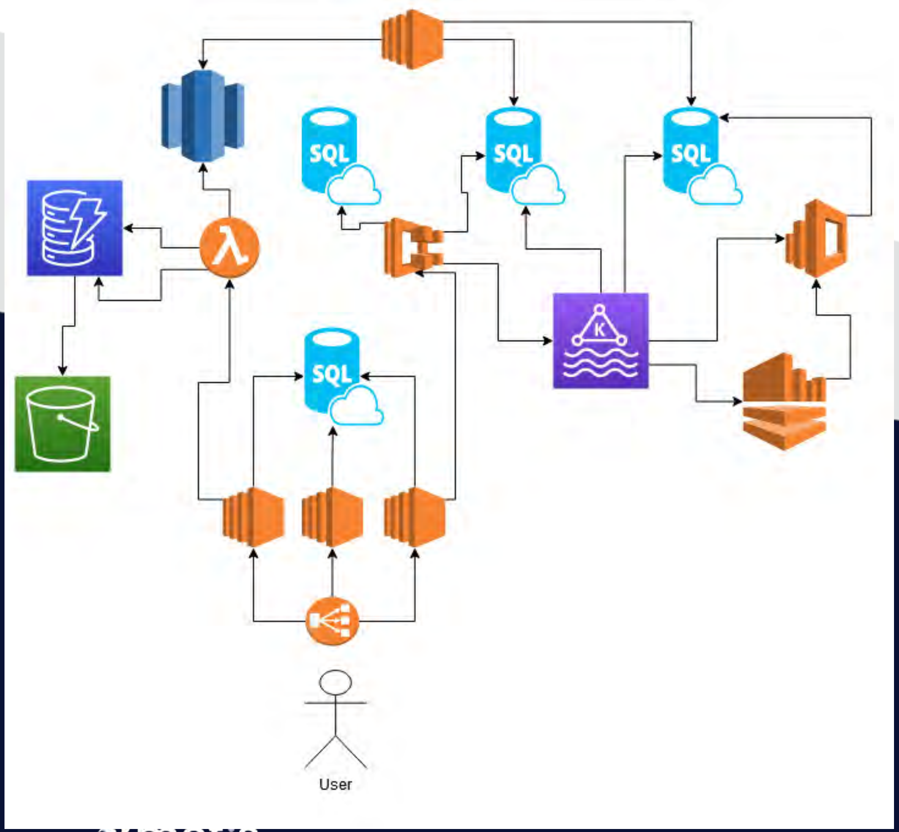
https://www.metisdata.io/

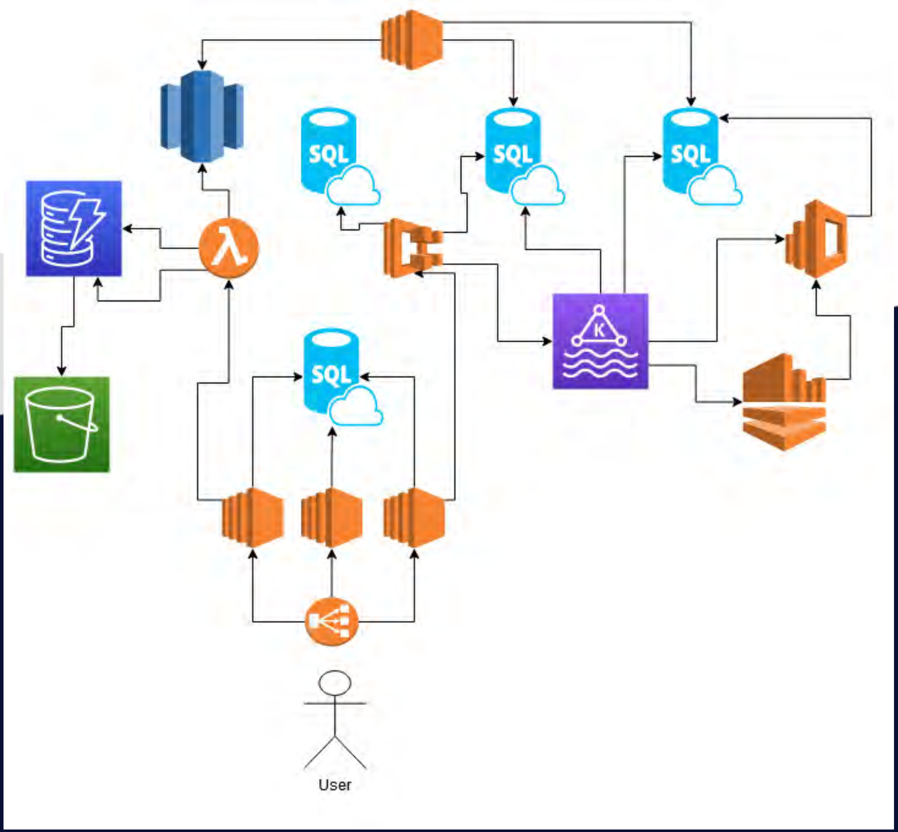
Where we used to be...



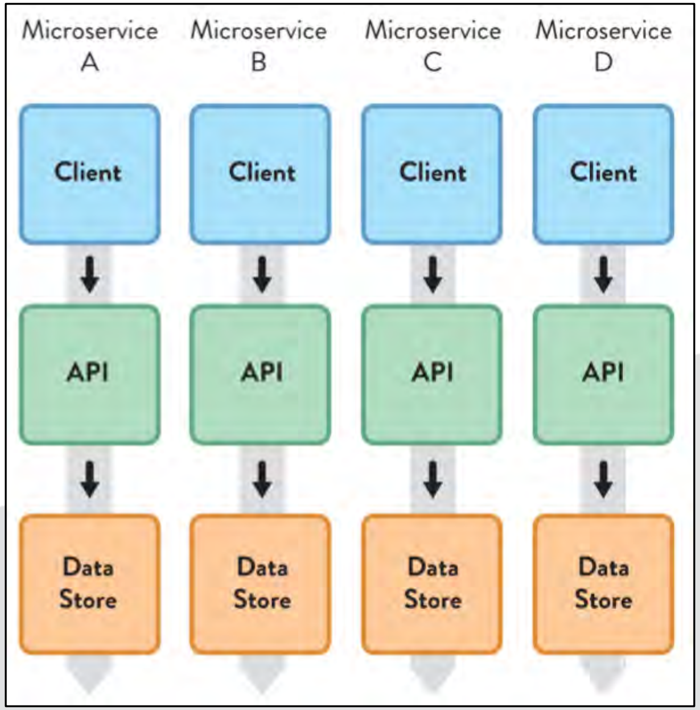


Where we are now

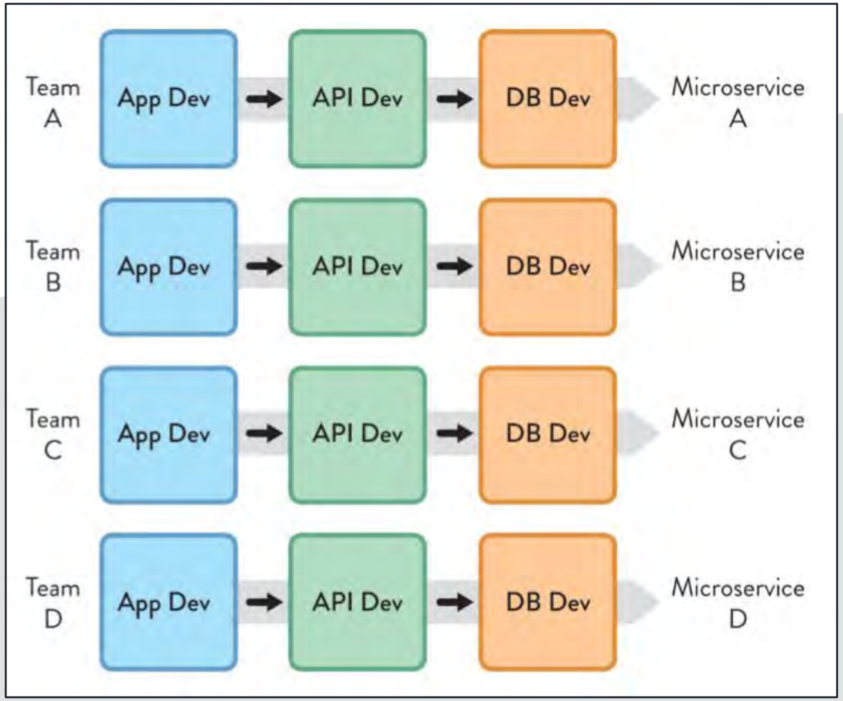




Conway’s Law

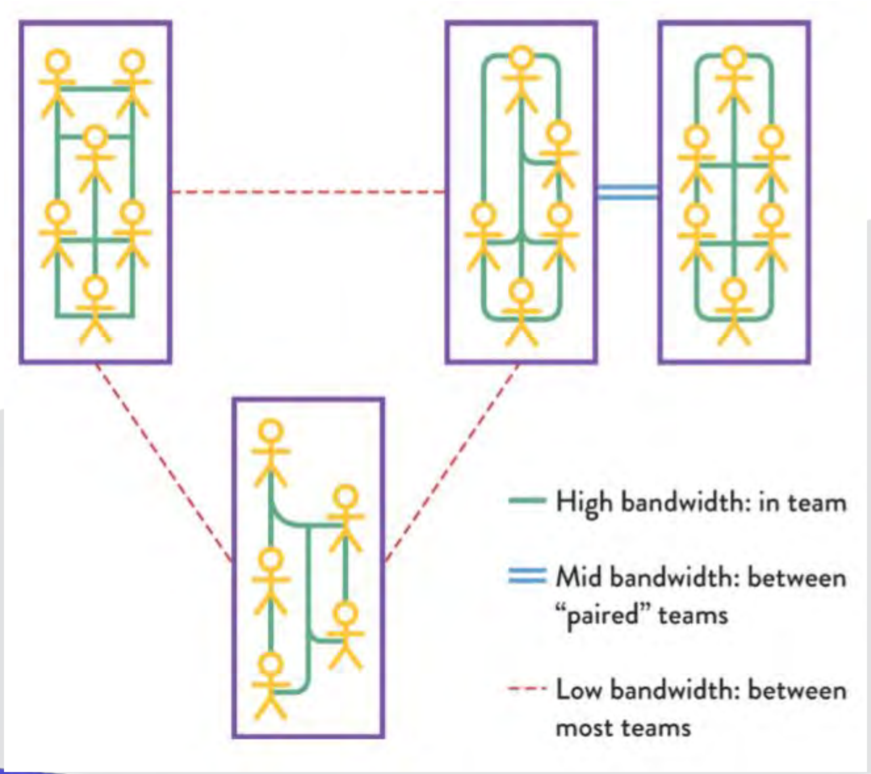


Microservice A
Microservice B
Microservice C
Microservice D
Client
Client
Client
Client
API
API
API
API
Data Store
Data Store
Data Store
Data Store



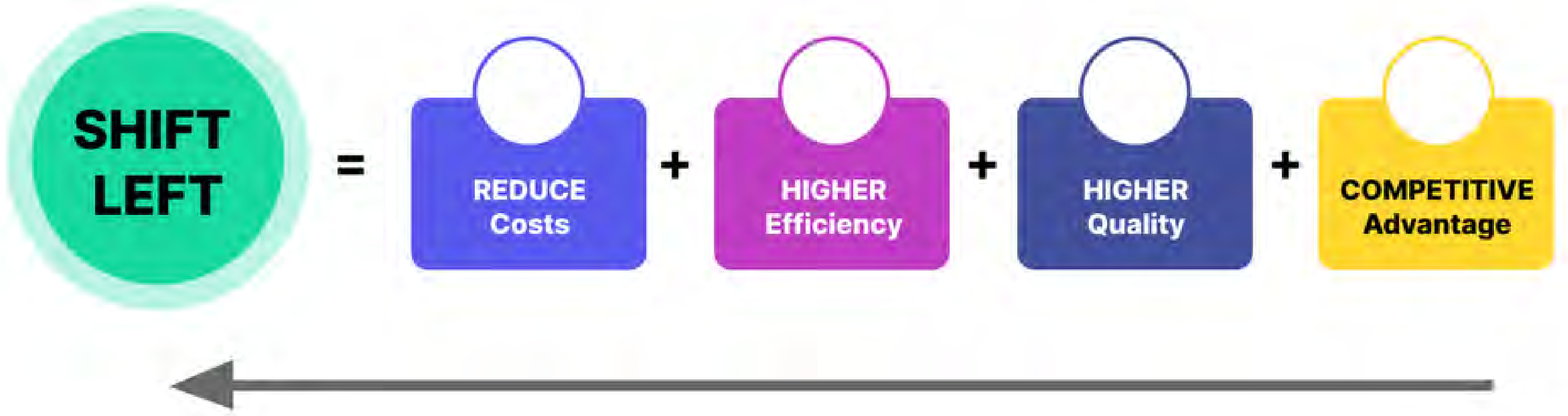
Team A
App Dev
API Dev
DB Dev
Microservice A
Team B
App Dev
API Dev
DB Dev
Microservice B
Team C
App Dev
API Dev
DB Dev
Microservice C
Team D
App Dev
API Dev
DB Dev
Microservice D

Communication


High bandwidth: in team
Mid bandwidth: between "paired" teams
Low bandwidth: between most teams

metis

Shift Left



https://katalon.com/resources-center/blog/shift-left-testing-approach

Self Service

Dashboards

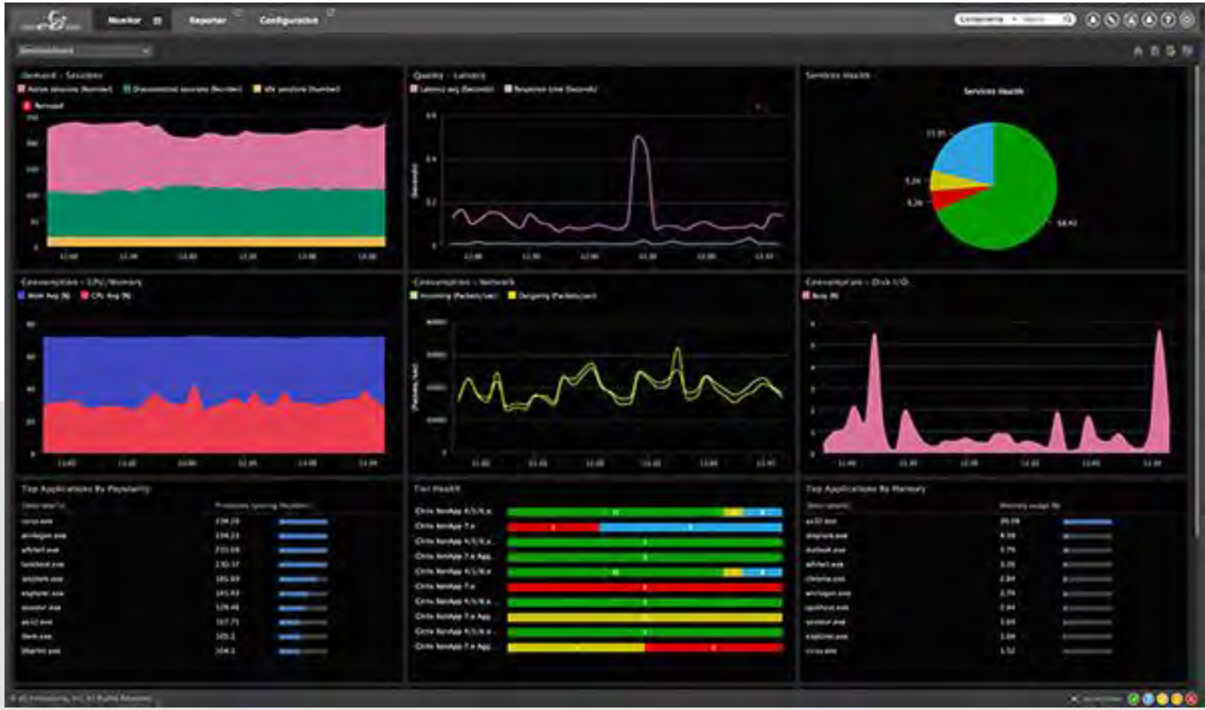
Lots of data.

Lack of information.

Lots of signals.

Lack of clarity.

No way to understand the big picture.



https://www.eginnovations.com/blog/monitoring-dashboards-it-performance/

What breaks in the current world?

Problems with databases

Slow queries.

Inaccurate statistics.

Incompatible changes in schema.

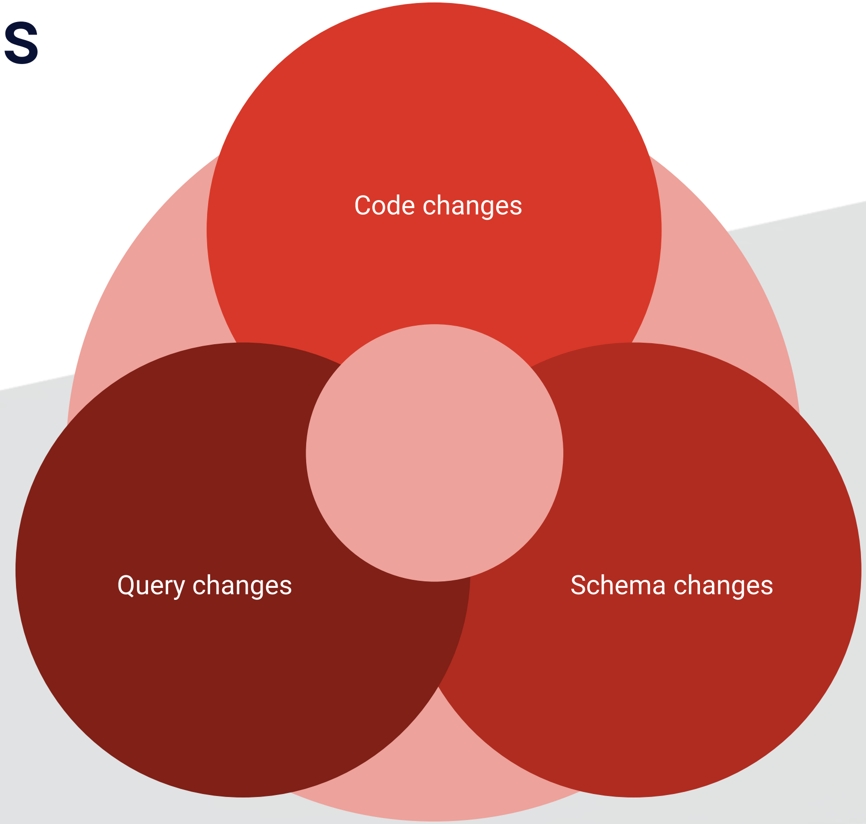
Bugs.

Missing indexes.

Data quality.

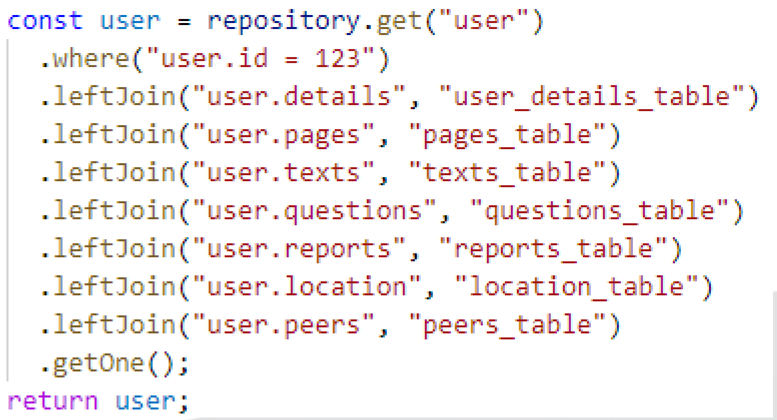
Configuration.

Locks.

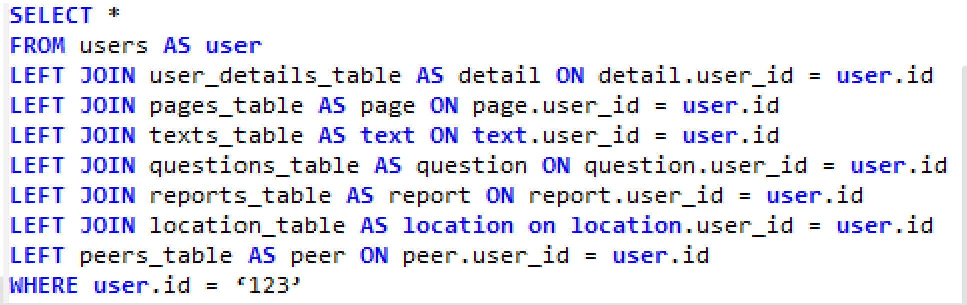


metis

Slow queries



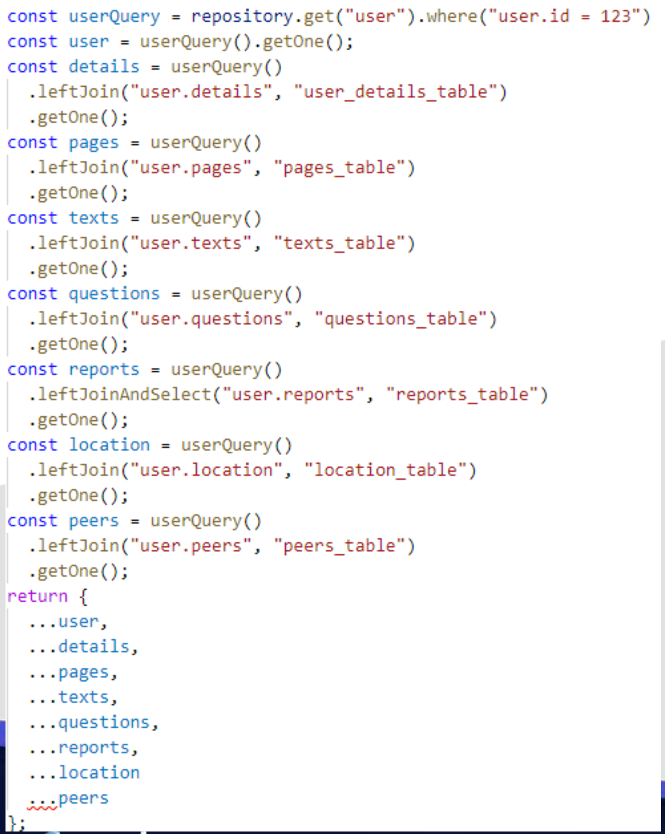
```
const user = repository.get("user")
  .where("user.id = 123")
  .leftJoin("user.details", "user_details_table")
  .leftJoin("user.pages", "pages_table")
  .leftJoin("user.texts", "texts_table")
  .leftJoin("user.questions", "questions_table")
  .leftJoin("user.reports", "reports_table")
  .leftJoin("user.location", "location_table")
  .leftJoin("user.peers", "peers_table")
  .getOne();
return user;
```



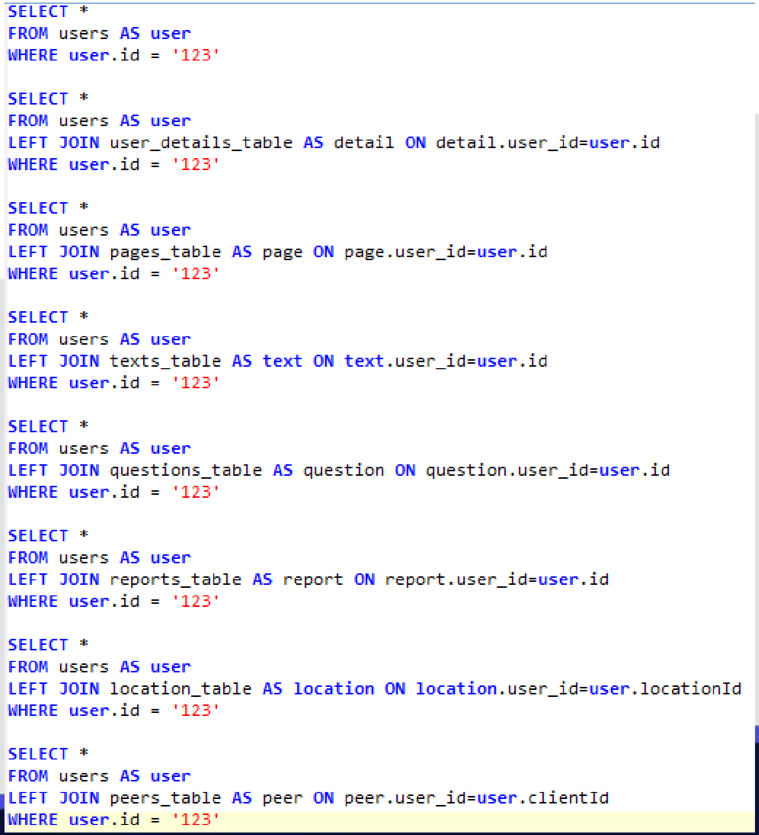
```
SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id = user.id
LEFT JOIN pages_table AS page ON page.user_id = user.id
LEFT JOIN texts_table AS text ON text.user_id = user.id
LEFT JOIN questions_table AS question ON question.user_id = user.id
LEFT JOIN reports_table AS report ON report.user_id = user.id
LEFT JOIN location_table AS location on location.user_id = user.id
LEFT peers_table AS peer ON peer.user_id = user.id
WHERE user.id = ‘123’
```

This reads ~300k rows and runs for 25 seconds.

Slow queries



```
const userQuery = repository.get("user").where("user.id = 123")
const user = userQuery().getOne();
const details = userQuery()
  .leftJoin("user.details", "user_details_table")
  .getOne();
const pages = userQuery()
  .leftJoin("user.pages", "pages_table")
  .getOne();
const texts = userQuery()
  .leftJoin("user.texts", "texts_table")
  .getOne();
const questions = userQuery()
  .leftJoin("user.questions", "questions_table")
  .getOne();
const reports = userQuery()
  .leftJoinAndSelect("user.reports", "reports_table")
  .getOne();
const location = userQuery()
  .leftJoin("user.location", "location_table")
  .getOne();
const peers = userQuery()
  .leftJoin("user.peers", "peers_table")
  .getOne();
return {
  ...user,
  ...details,
  ...pages,
  ...texts,
  ...questions,
  ...reports,
  ...location
  ...peers
};
```



```
SELECT *
FROM users AS user
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id=user.id
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN pages_table AS page ON page.user_id=user.id
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN texts_table AS text ON text.user_id=user.id
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN questions_table AS question ON question.user_id=user.id
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN reports_table AS report ON report.user_id=user.id
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN location_table AS location ON location.user_id=user.locationId
WHERE user.id = '123'

SELECT *
FROM users AS user
LEFT JOIN peers_table AS peer ON peer.user_id=user.clientId
WHERE user.id = '123'
```

Slow queries

```
-- 7925812 rows
SELECT COUNT(*)
FROM boarding_passes
```

```
-- 13 seconds
WITH cte_performance AS (
        SELECT *, MD5(MD5(ticket_no)) AS double_hash
        FROM boarding_passes
)
SELECT COUNT(*)
FROM cte_performance AS C1
JOIN cte_performance AS C2 ON C2.ticket_no = C1.ticket_no AND C2.flight_id = C1.flight_id AND C2.boarding_no = C1.boarding_no
JOIN cte_performance AS C3 ON C3.ticket_no = C1.ticket_no AND C3.flight_id = C1.flight_id AND C3.boarding_no = C1.boarding_no
WHERE
        C1.double_hash = '525ac610982920ef37b34aa56a45cd06'
        AND C2.double_hash = '525ac610982920ef37b34aa56a45cd06'
        AND C3.double_hash = '525ac610982920ef37b34aa56a45cd06'
```

```
-- 8 seconds
SELECT COUNT(*)
FROM boarding_passes AS C1
JOIN boarding_passes AS C2 ON C2.ticket_no = C1.ticket_no AND C2.flight_id = C1.flight_id AND C2.boarding_no = C1.boarding_no
JOIN boarding_passes AS C3 ON C3.ticket_no = C1.ticket_no AND C3.flight_id = C1.flight_id AND C3.boarding_no = C1.boarding_no
WHERE
        MD5(MD5(C1.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'
        AND MD5(MD5(C2.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'
        AND MD5(MD5(C3.ticket_no)) = '525ac610982920ef37b34aa56a45cd06'
```

metis

Incompatible changes in schema

Adding a column

- May cause issues when we use SELECT *
- May cause table reorganization because of lack of space (and outage in result)

Dropping a column

- Nearly never safe

Altering the column type

- May change the representation, this depends on the ORM and the driver
- May require some extensions installed to the database engine
- May cause table reorganization

Missing Indexes

May cause scanning whole table instead of getting rows directly.

May cause using inefficient JOIN strategy (nested loop instead of hash join or merge join).

Index

- Created automatically with a primary key
- May be created on demand
- May store one or more columns
- Stores data in an order, so it’s easy to do binary search

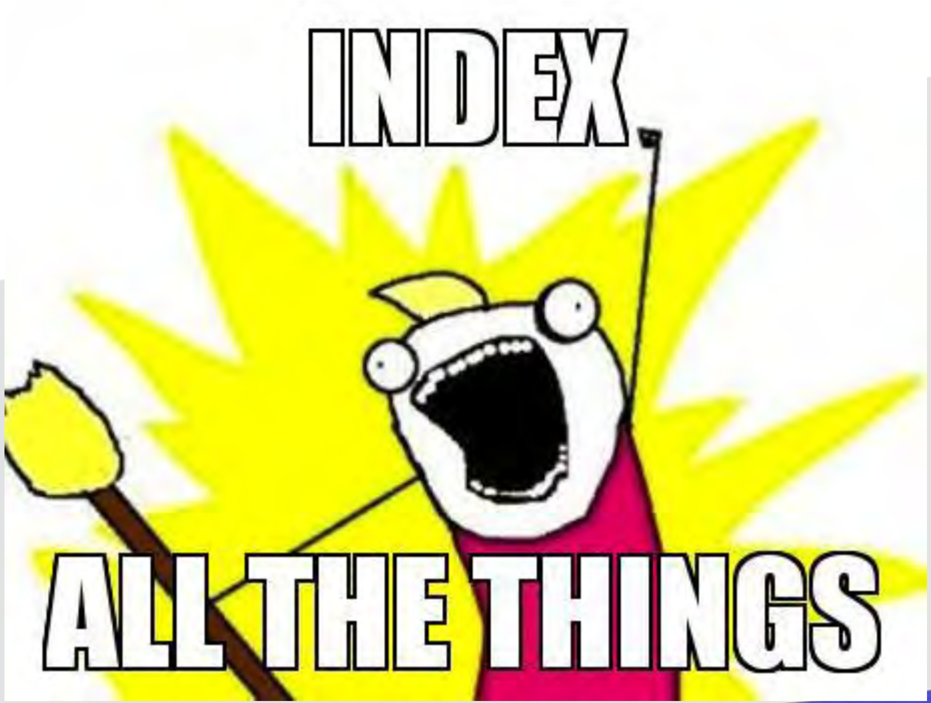
Some index types

- B-Tree
- Hash index
- GIS-based (for geolocation)
- GIN (inverted indexes)

Too many indexes

Indexes are not free

- They store data in a specific order that needs to be maintained over time
- They need to copy the data on the side to build additional dictionaries
- Updating one row may cause an update in multiple indexes
- **Do not index blindly!** Evaluate if the performance increases



metis

ORM challenges - n+1 selects

Problem:

```
aircrafts = aircrafts.load();
for(aircraft in aircrafts) {
  seatsCount = aircraft.seats.size;
}
```

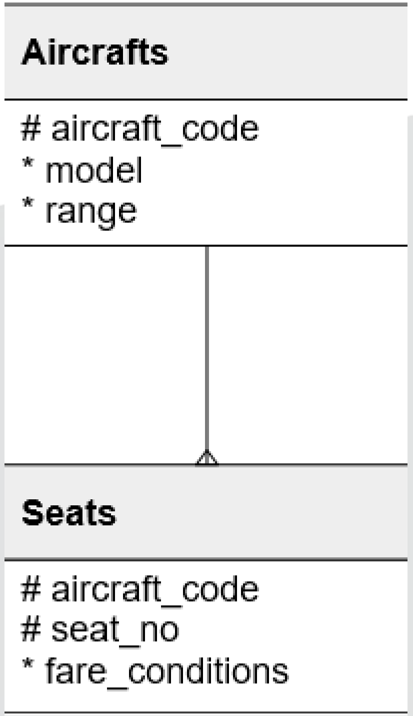
This generates:

```
SELECT * FROM aircrafts;

SELECT * FROM seats WHERE aircraft_code = 1
SELECT * FROM seats WHERE aircraft_code = 2
SELECT * FROM seats WHERE aircraft_code = 3
...
```

However, this could be done in one query:

```
SELECT * FROM aircrafts
LEFT JOIN seats ON seats.aircraft_code = aircrafts.aircraft_code
```



metis

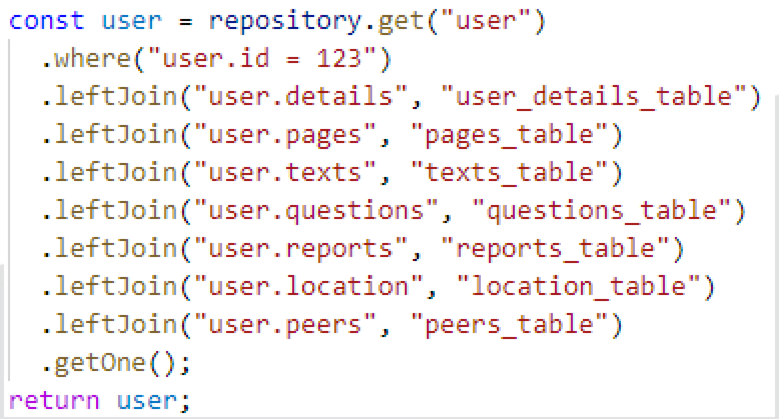
ORM challenges – joins

Normalization leads to multiple joins that may be slow.

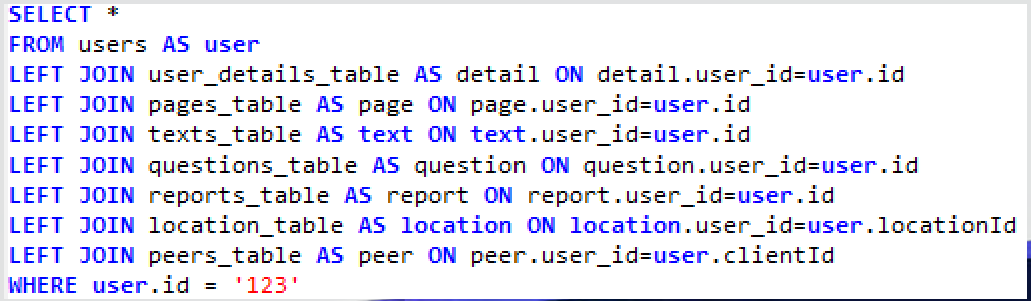
We may need to decompose these queries manually.

We may need to rework our domain model.

We may need to change bounded contexts.



```
const user = repository.get("user")
  .where("user.id = 123")
  .leftJoin("user.details", "user_details_table")
  .leftJoin("user.pages", "pages_table")
  .leftJoin("user.texts", "texts_table")
  .leftJoin("user.questions", "questions_table")
  .leftJoin("user.reports", "reports_table")
  .leftJoin("user.location", "location_table")
  .leftJoin("user.peers", "peers_table")
  .getOne();
return user;
```



```
SELECT *
FROM users AS user
LEFT JOIN user_details_table AS detail ON detail.user_id=user.id
LEFT JOIN pages_table AS page ON page.user_id=user.id
LEFT JOIN texts_table AS text ON text.user_id=user.id
LEFT JOIN questions_table AS question ON question.user_id=user.id
LEFT JOIN reports_table AS report ON report.user_id=user.id
LEFT JOIN location_table AS location ON location.user_id=user.locationId
LEFT JOIN peers_table AS peer ON peer.user_id=user.clientId
WHERE user.id = '123'
```

ORM challenges – lack of visibility

Transaction isolation level

- Each transaction has a level (SERIALIZABLE, READ COMMITTED, etc.)
- What's the default?
- Can you change it?

Transaction scope

- When is transaction started? When does it end?
- Do you have nested transactions?

Commit/rollback

- Who controls how things are committed and rolled back?
- What happens in case of errors?

Caching

- Is the data cached?
- Does it work with parallel connections?
- What about sticky sessions/

Pooling

- Do you have a connection pool?
- Will it scale well?
- How often do you recycle the connection?

Query hints

- How do you make sure indexes are used?
- How do you configure join strategy?

ORM challenges - migrations

How do you define your migrations

- SQL files with CREATE TABLE...
- Code first with ORM model
- Or maybe you already have the database?

How do you track which things were executed

- Keep another table with history
- Make sure changes are idempotent
- You run them manually

How do you roll back

- Up + Down methods

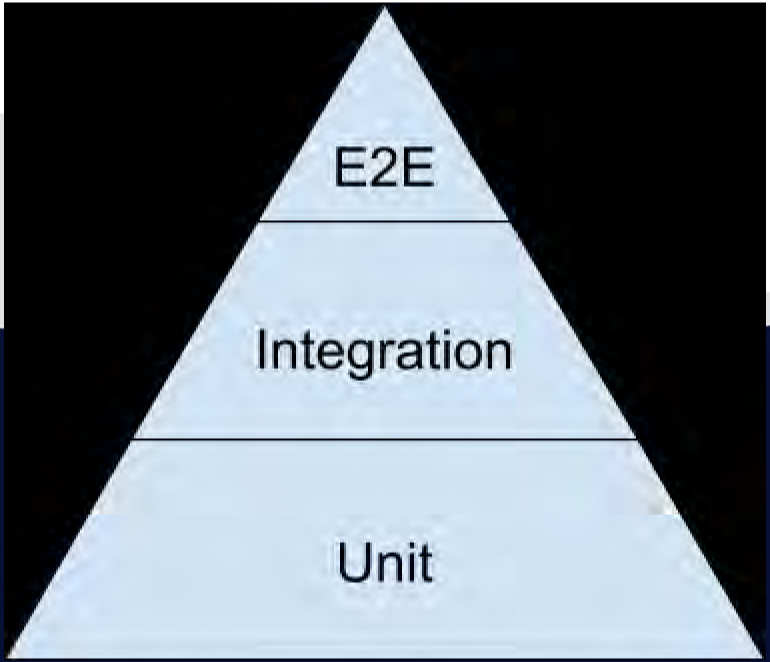
What if there are multiple heterogeneous applications?

What if your ORM creates tables automatically?

How do you deal with migrations in unit tests?

How do you fix errors which you spot later on?

Tests - do they work?



Load testing?

Cost

- Load test takes hours to complete (think caching, tiered compilation, etc.)

Data distribution and cardinality

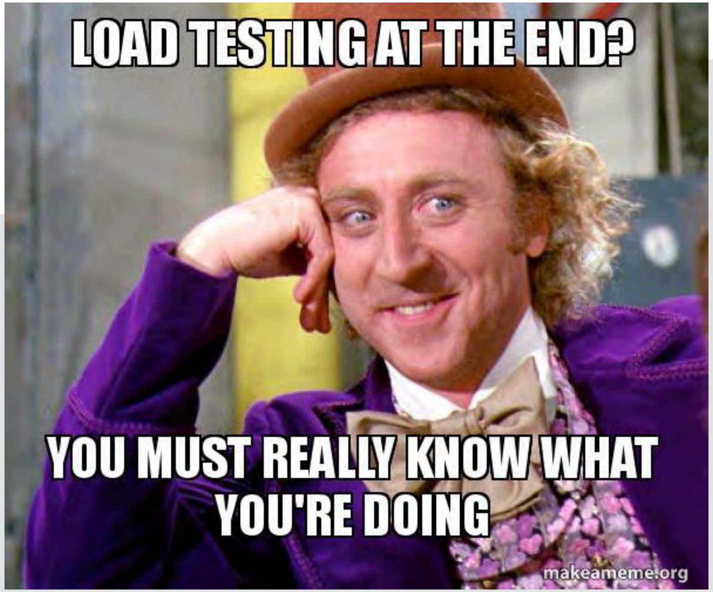
- You can’t test your EU stack with the data from the USA
- What about smaller countries?

Hardware and environment

- GPUs are expensive and not very available
- Edge computing? Custom hardware?
- Do you pay for it 24/7?

Data anonymity

- What about SSN? How do you anonymize it in pre-production?



Nonoptimal Configuration

The Right Solutions

- Trigram or JSONB indexes
- Vector Database vs pgvector
- Impedance mismatch

The Solutions Done Right

- Maintenance windows
- Vacuuming
- Defragmentation

Load handling

- Scalability, hardware, peak hours

Differences between regions?

Multi-tenancy?

Old versions or cheaper editions?

Can we change the database?

Solution - Database Guardrails

metis

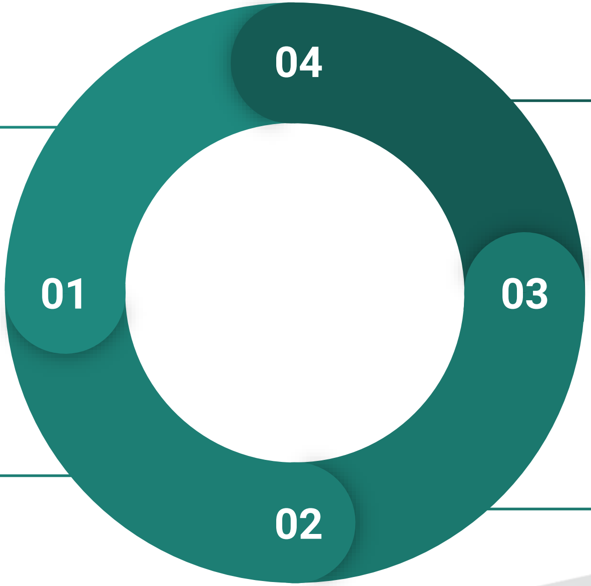
Know the context to find the root cause

Telemetry

Ability to collect data - logs, metrics, traces.

Visibility

Seeing “what” inside the system.



Observability

Deep dive into technical details for root cause analysis.

Application Performance Management

High-level end-to-end system health.

Monitoring and Observability

Monitoring

- Alerts about errors.
- Often swamps with raw data, metrics, charts, graphs.
- Often application-agnostic, focuses on infrastructure.
- Rarely connects the dots between various systems.

Observability

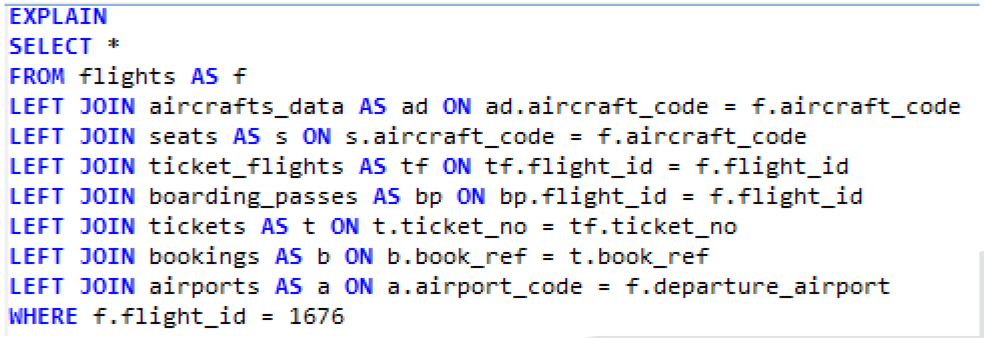
- Shows root causes of the errors.
- Provides semantic understanding of what is happening.
- Understands the characteristics of the application.
- Makes the interconnection clear and visible.

What to observe?

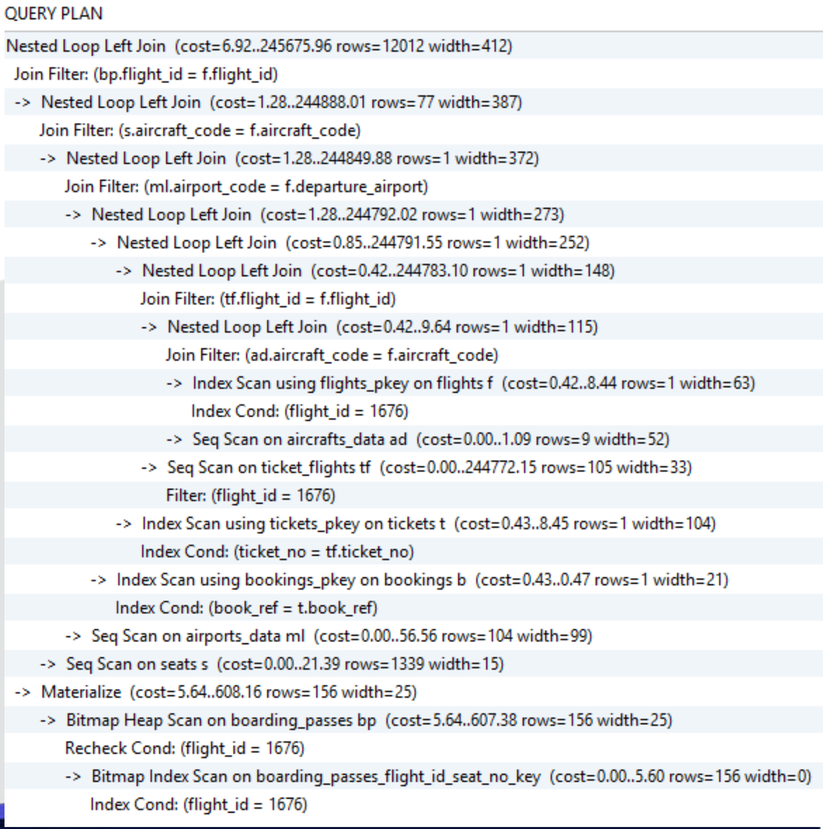
Executing the query

Parser	Query is parsed into an Abstract Syntax Tree (AST). This allows to manipulate the query mechanically.
Rewriter	Query is rewritten to a standard form. This makes processing the query easier.
Planner	A plan is prepared. It contains details of how to read data, how to join tables, how to filter rows, etc.
Executor	Finally, the query is physically executed.

Anatomy of an SQL query



```sql
EXPLAIN
SELECT *
FROM flights AS f
LEFT JOIN aircrafts_data AS ad ON ad.aircraft_code = f.aircraft_code
LEFT JOIN seats AS s ON s.aircraft_code = f.aircraft_code
LEFT JOIN ticket_flights AS tf ON tf.flight_id = f.flight_id
LEFT JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
LEFT JOIN tickets AS t ON t.ticket_no = tf.ticket_no
LEFT JOIN bookings AS b ON b.book_ref = t.book_ref
LEFT JOIN airports AS a ON a.airport_code = f.departure_airport
WHERE f.flight_id = 1676
```



```
QUERY PLAN
Nested Loop Left Join  (cost=6.92..245675.96 rows=12012 width=412)
  Join Filter: (bp.flight_id = f.flight_id)
  ->  Nested Loop Left Join  (cost=1.28..244888.01 rows=77 width=387)
        Join Filter: (s.aircraft_code = f.aircraft_code)
        ->  Nested Loop Left Join  (cost=1.28..244849.88 rows=1 width=372)
              Join Filter: (ml.airport_code = f.departure_airport)
              ->  Nested Loop Left Join  (cost=1.28..244792.02 rows=1 width=273)
                    ->  Nested Loop Left Join  (cost=0.85..244791.55 rows=1 width=252)
                          ->  Nested Loop Left Join  (cost=0.42..244783.10 rows=1 width=148)
                                Join Filter: (tf.flight_id = f.flight_id)
                                ->  Nested Loop Left Join  (cost=0.42..9.64 rows=1 width=115)
                                      Join Filter: (ad.aircraft_code = f.aircraft_code)
                                      ->  Index Scan using flights_pkey on flights f  (cost=0.42..8.44 rows=1 width=63)
                                            Index Cond: (flight_id = 1676)
                                      ->  Seq Scan on aircrafts_data ad  (cost=0.00..1.09 rows=9 width=52)
                                ->  Seq Scan on ticket_flights tf  (cost=0.00..244772.15 rows=105 width=33)
                                      Filter: (flight_id = 1676)
                          ->  Index Scan using tickets_pkey on tickets t  (cost=0.43..8.45 rows=1 width=104)
                                Index Cond: (ticket_no = tf.ticket_no)
                    ->  Index Scan using bookings_pkey on bookings b  (cost=0.43..0.47 rows=1 width=21)
                          Index Cond: (book_ref = t.book_ref)
              ->  Seq Scan on airports_data ml  (cost=0.00..56.56 rows=104 width=99)
        ->  Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
  ->  Materialize  (cost=5.64..608.16 rows=156 width=25)
        ->  Bitmap Heap Scan on boarding_passes bp  (cost=5.64..607.38 rows=156 width=25)
              Recheck Cond: (flight_id = 1676)
              ->  Bitmap Index Scan on boarding_passes_flight_id_seat_no_key  (cost=0.00..5.60 rows=156 width=0)
                    Index Cond: (flight_id = 1676)
```

metis

Anatomy of an SQL query

Each plan consists of **nodes**.

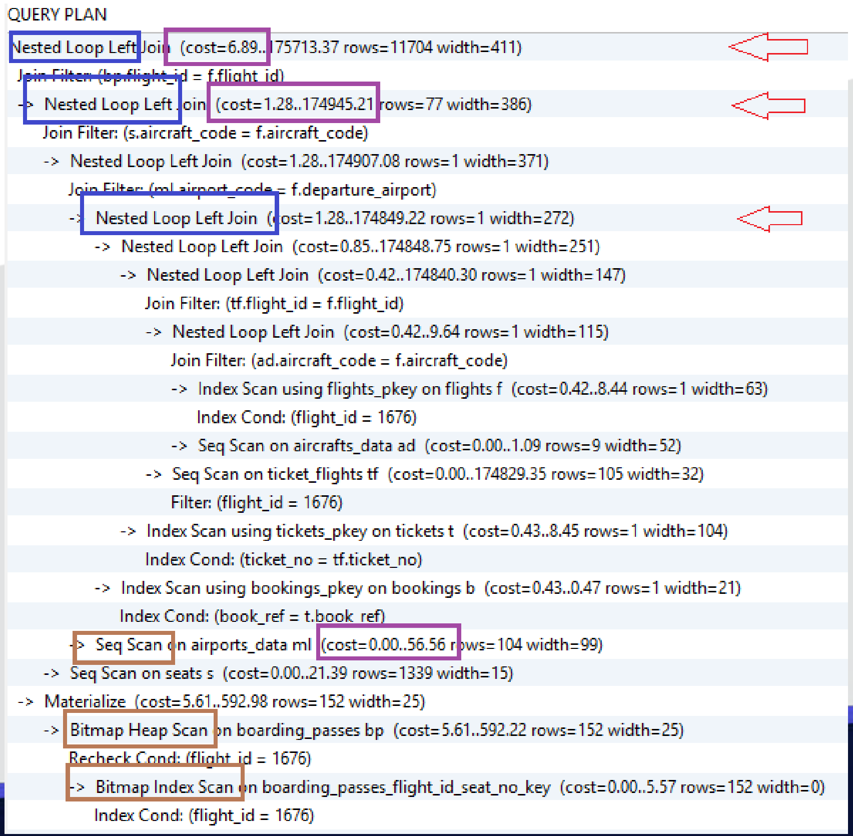
Nodes have **costs** associated with them.

- Cost is an arbitrary measure of “how hard it is to get the whole dataset”

Most important parts are:

- **Scans** - Sequential Scan Index Scan, index-Only scan
- **Joins** - Nested Loop, Hash, Merge
- Others: Limit, Materialize, Sort

https://www.pgmustard.com/docs/explain



```
QUERY PLAN
Nested Loop Left Join  (cost=6.89..175713.37 rows=11704 width=411)
  Join Filter: (bp.flight_id = f.flight_id)
  ->  Nested Loop Left Join  (cost=1.28..174945.21 rows=77 width=386)
        Join Filter: (s.aircraft_code = f.aircraft_code)
        ->  Nested Loop Left Join  (cost=1.28..174907.08 rows=1 width=371)
              Join Filter: (ml.airport_code = f.departure_airport)
              ->  Nested Loop Left Join  (cost=1.28..174849.22 rows=1 width=272)
                    ->  Nested Loop Left Join  (cost=0.85..174848.75 rows=1 width=251)
                          ->  Nested Loop Left Join  (cost=0.42..174840.30 rows=1 width=147)
                                Join Filter: (tf.flight_id = f.flight_id)
                                ->  Nested Loop Left Join  (cost=0.42..9.64 rows=1 width=115)
                                      Join Filter: (ad.aircraft_code = f.aircraft_code)
                                      ->  Index Scan using flights_pkey on flights f  (cost=0.42..8.44 rows=1 width=63)
                                            Index Cond: (flight_id = 1676)
                                      ->  Seq Scan on aircrafts_data ad  (cost=0.00..1.09 rows=9 width=52)
                                ->  Seq Scan on ticket_flights tf  (cost=0.00..174829.35 rows=105 width=32)
                                      Filter: (flight_id = 1676)
                          ->  Index Scan using tickets_pkey on tickets t  (cost=0.43..8.45 rows=1 width=104)
                                Index Cond: (ticket_no = tf.ticket_no)
                    ->  Index Scan using bookings_pkey on bookings b  (cost=0.43..0.47 rows=1 width=21)
                          Index Cond: (book_ref = t.book_ref)
              ->  Seq Scan on airports_data ml  (cost=0.00..56.56 rows=104 width=99)
        ->  Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
  ->  Materialize  (cost=5.61..592.98 rows=152 width=25)
        ->  Bitmap Heap Scan on boarding_passes bp  (cost=5.61..592.22 rows=152 width=25)
              Recheck Cond: (flight_id = 1676)
              ->  Bitmap Index Scan on boarding_passes_flight_id_seat_no_key  (cost=0.00..5.57 rows=152 width=0)
                    Index Cond: (flight_id = 1676)
```

How to observe?

Observability

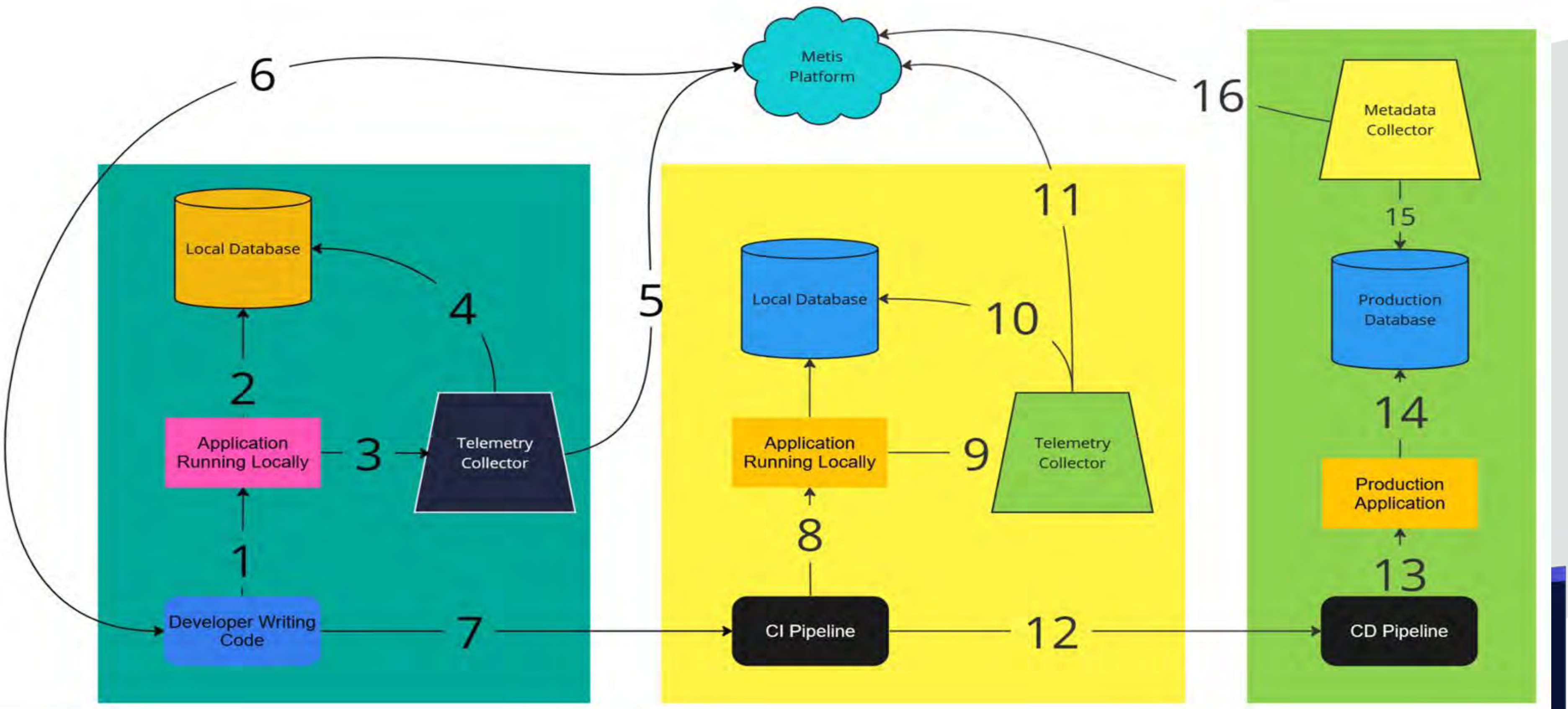
We need:

- Logs
- Traces
- Metrics

We face multiple challenges:

- Heterogeneous applications
- Correlations
- Extensibility

Observability



Database Guardrails

- ✓ Prevent bad code from reaching production

  Understand what’s happening inside the application

  Apply stats from production

- ✓ Monitor the system end-to-end

  Understand the application characteristics

  Turn raw data into actual knowledge

- ✓ Troubleshoot automatically by connecting the dots

  Focus on the root cause, not on the manifestation

  Work across stages

metis

Be proactive and push to the left!

Waiting for tickets from customers is expensive.

Load tests are slow, too late, and too expensive.

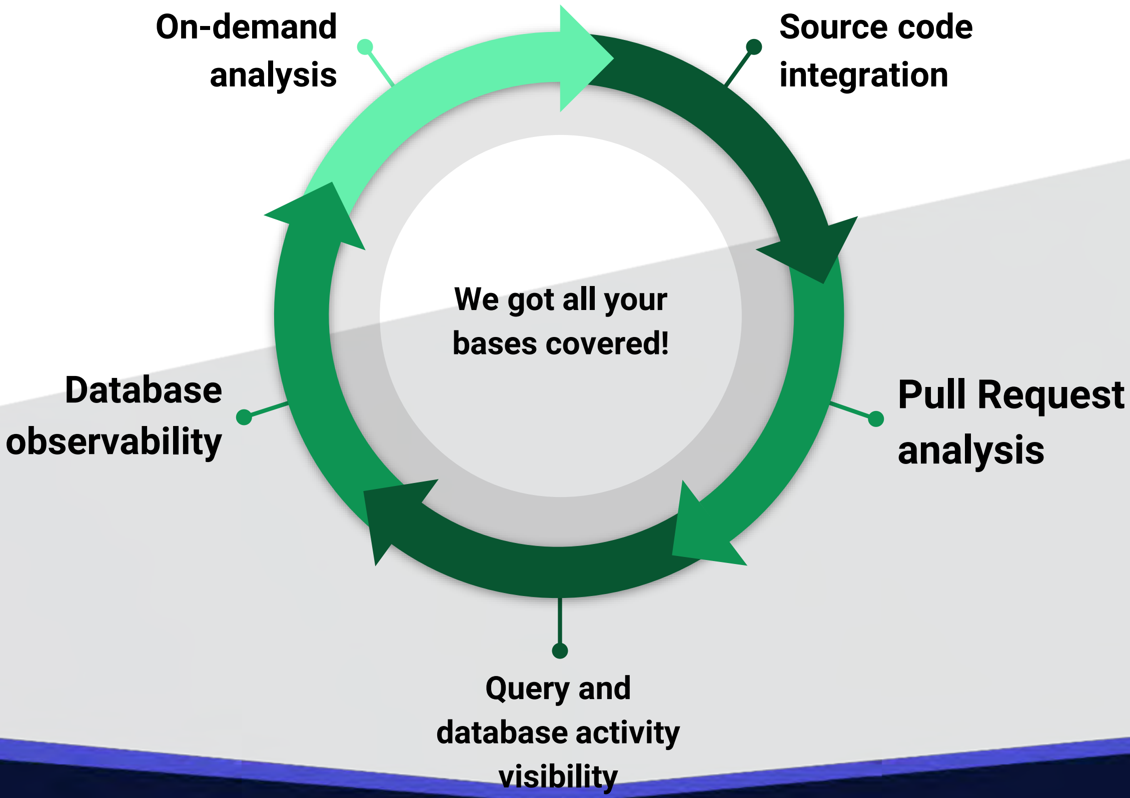
Issues need to be identified early and automatically.

metis

metis

Never go blind again!

Metis


On-demand analysis
Source code integration
We got all your bases covered!
Database observability
Pull Request analysis
Query and database activity visibility

Summary

✓ Database may break

- Bugs
- ORM quirks
- Database inefficiency

✓ You need to be proactive

- Load tests are too late
- Constant monitoring is needed

✓ Metis covers all of that

- App integration
- Pull requests
- Observability
- Safety

metis

Q&A

metis

Thank you!

https://www.metisdata.io/