Some ^ Application Development Challenges with Postgres

What makes an accidental DBA/architect?

- Team solely responsible for an application or service
- Limited external support for operations and infrastructure
- Being first to admit you know some SQL

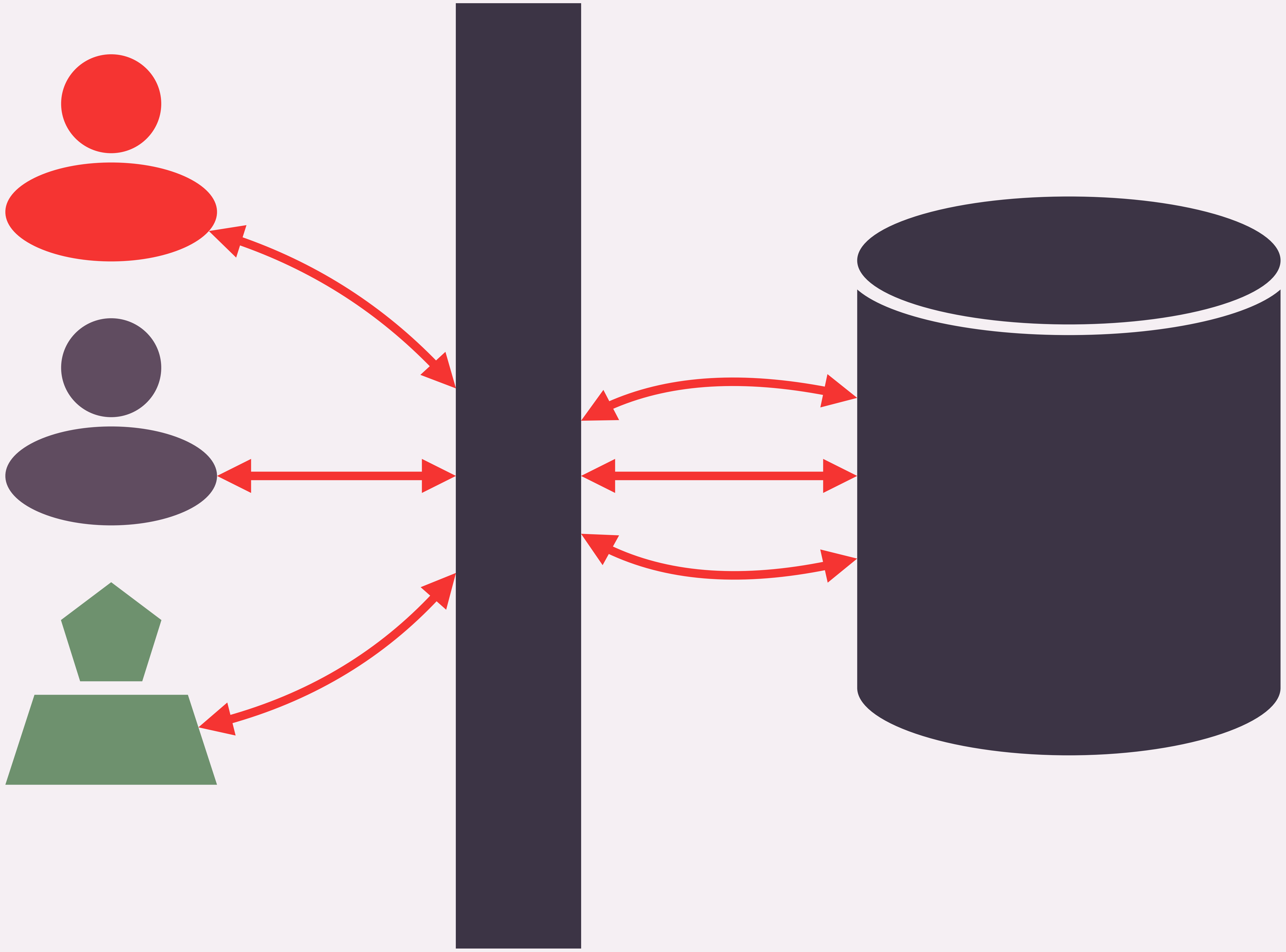
What makes an accidental DBA/architect?

- Team solely responsible for an application or service
- Limited external support for operations and infrastructure
- Being first to admit you know some SQL

Time constraints

What makes application development special?

Applications “hide” a database from users/systems



Web and local apps

REST, GraphQL, other APIs

Micro, macro, in-between services

Applications “hide” a database from users/systems

- Larger user base has more varied needs and goals
- Commitments are closer to “realtime” than “on time”
- Measurements & guarantees are holistic not specific
- System boundary is the application, not the database
- ….usually.

The vernacular: architecture without architects

The vernacular: architecture without architects

- Follows practical need over theoretical rigor

The vernacular: architecture without architects

- Follows practical need over theoretical rigor
- Builds to suit conditions on the ground

The vernacular: architecture without architects

- Follows practical need over theoretical rigor
- Builds to suit conditions on the ground
- Considers place within the whole environment

The vernacular: architecture without architects

- Follows practical need over theoretical rigor
- Builds to suit conditions on the ground
- Considers place within the whole environment
- Varies within well-known or traditional idioms

Let’s build an application!

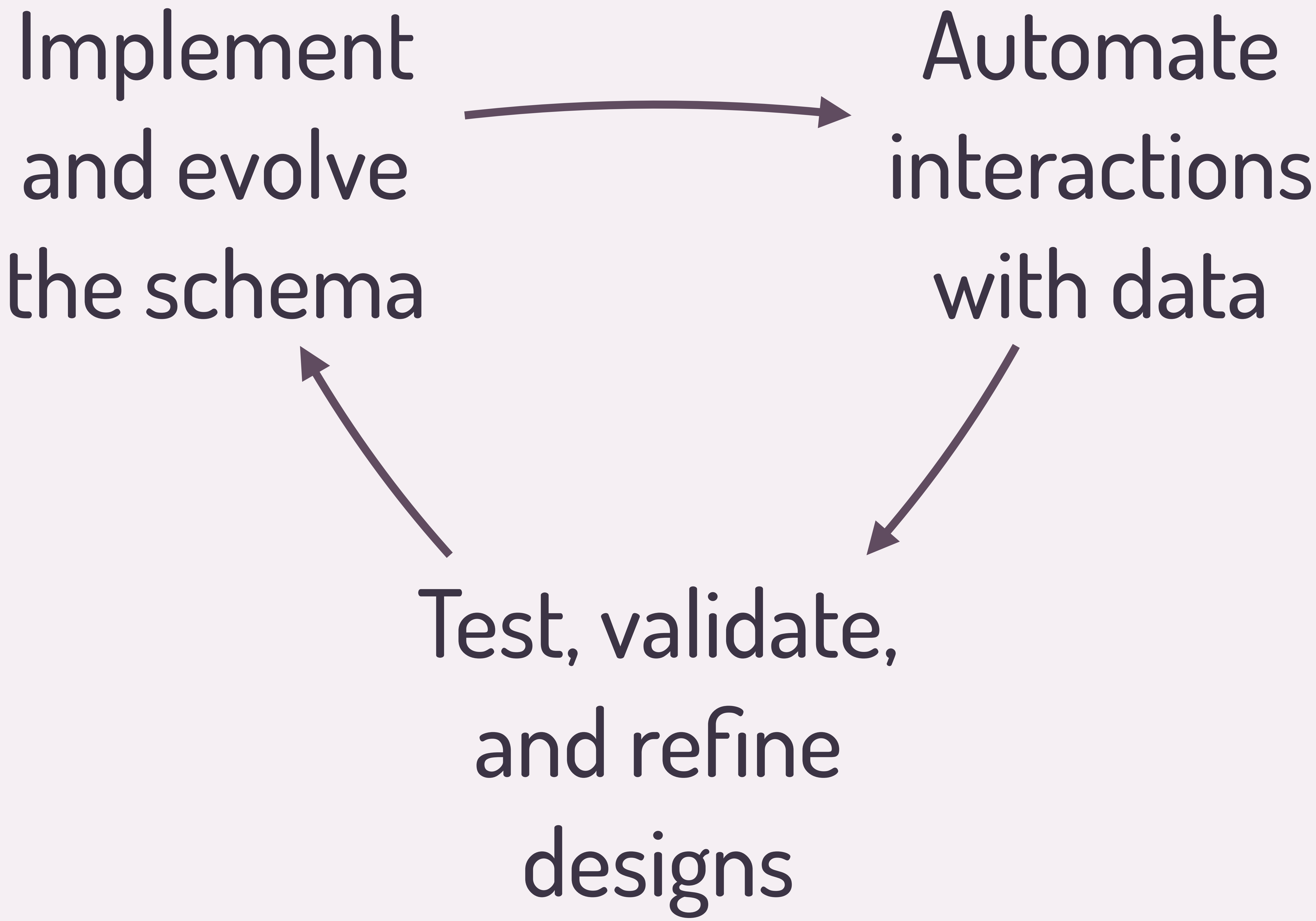
Shopping List

- Schema evolution tool
- Data access layer for application code
- That should be all we need, right?

Shopping List

- Schema evolution tool
- Data access layer for application code
- Connection pooling
- Monitoring/observability
- Backups

How do we interact with Postgres?



Implement and evolve the schema

Automate interactions with data

Test, validate, and refine designs

What goes into our data-architectural decisions?

- Requirements from user research or otherwise
- Intuition about **transient** representations
- Fear, or worse, fearlessness

Schema evolution, part I

- Extensions can save work — if we know about them
- Simple role permissions, usually
- Postgres' modeling flexibility is a two-edged sword
- Data access tools may be less capable

Data access and manipulation

- Application developers go to great lengths to avoid SQL
- Need to run queries with dynamic criteria & select lists
- Want to avoid SQL injection risk
- Want to minimize boilerplate connection/cursor juggling

The Origin of Data Access Layers

DAL evolution: the beginning

- SQL statements in client code
- Hand-built dynamic SQL
- No connection/cursor management affordances
- Result extraction from cursor, ResultSet, etc
- No inherent organization

DAL evolution: object/relational mappers

- Hibernate, ActiveRecord
- Managed connections & cursors
- Results marshaled into classes recapitulating data model
- Impedance mismatch

DAL evolution: data mappers and query builders

- MyBatis, MassiveJS
- SQL statements prewritten and/or generated
- Managed connections & cursors
- Results marshaling

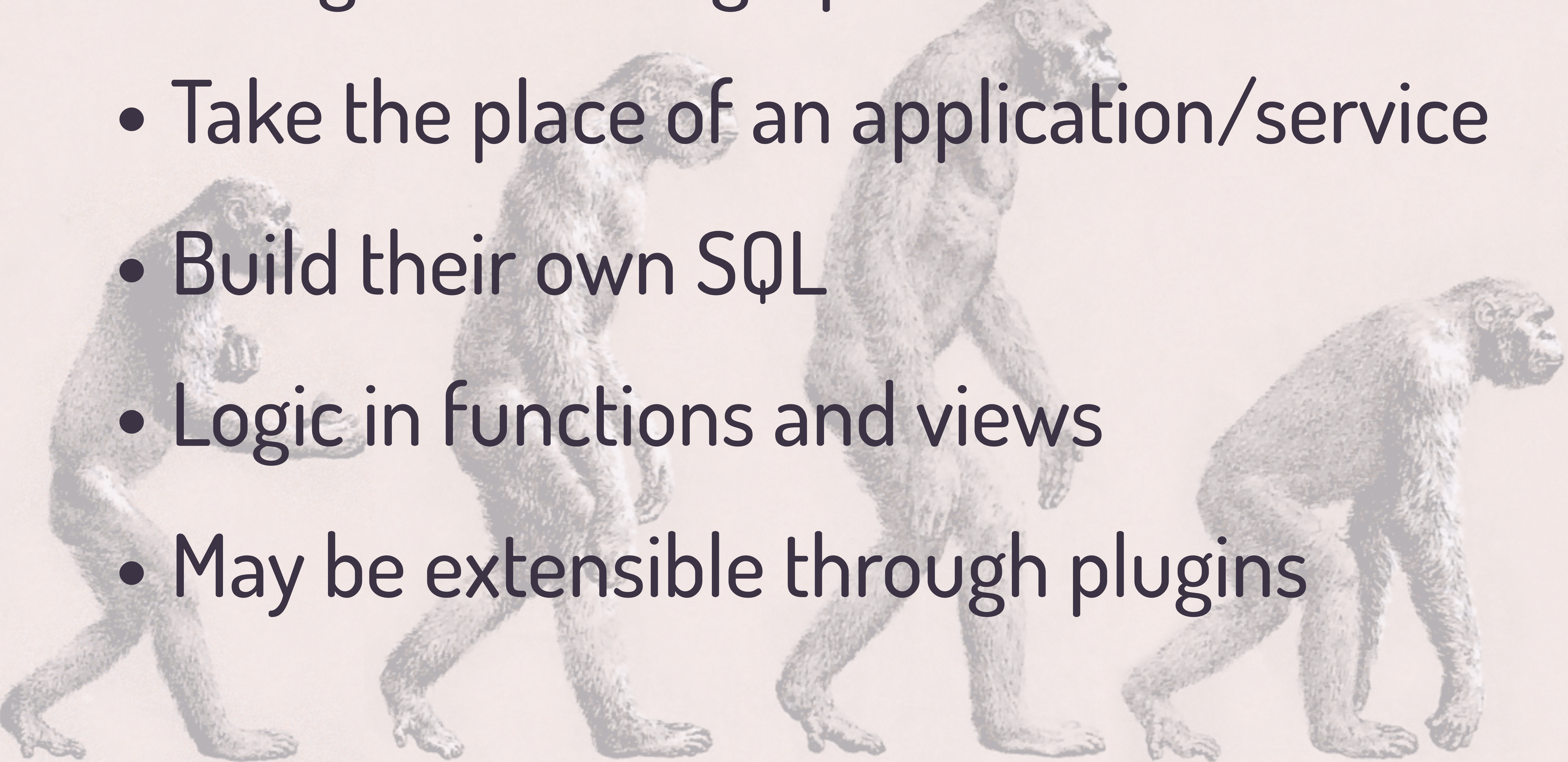
- Knex.js, SQLAlchemy Core, jOOQ, penkala, monstrous
- Build SQL with relational-algebraic functions
- Managed connections & cursors
- Results marshaling

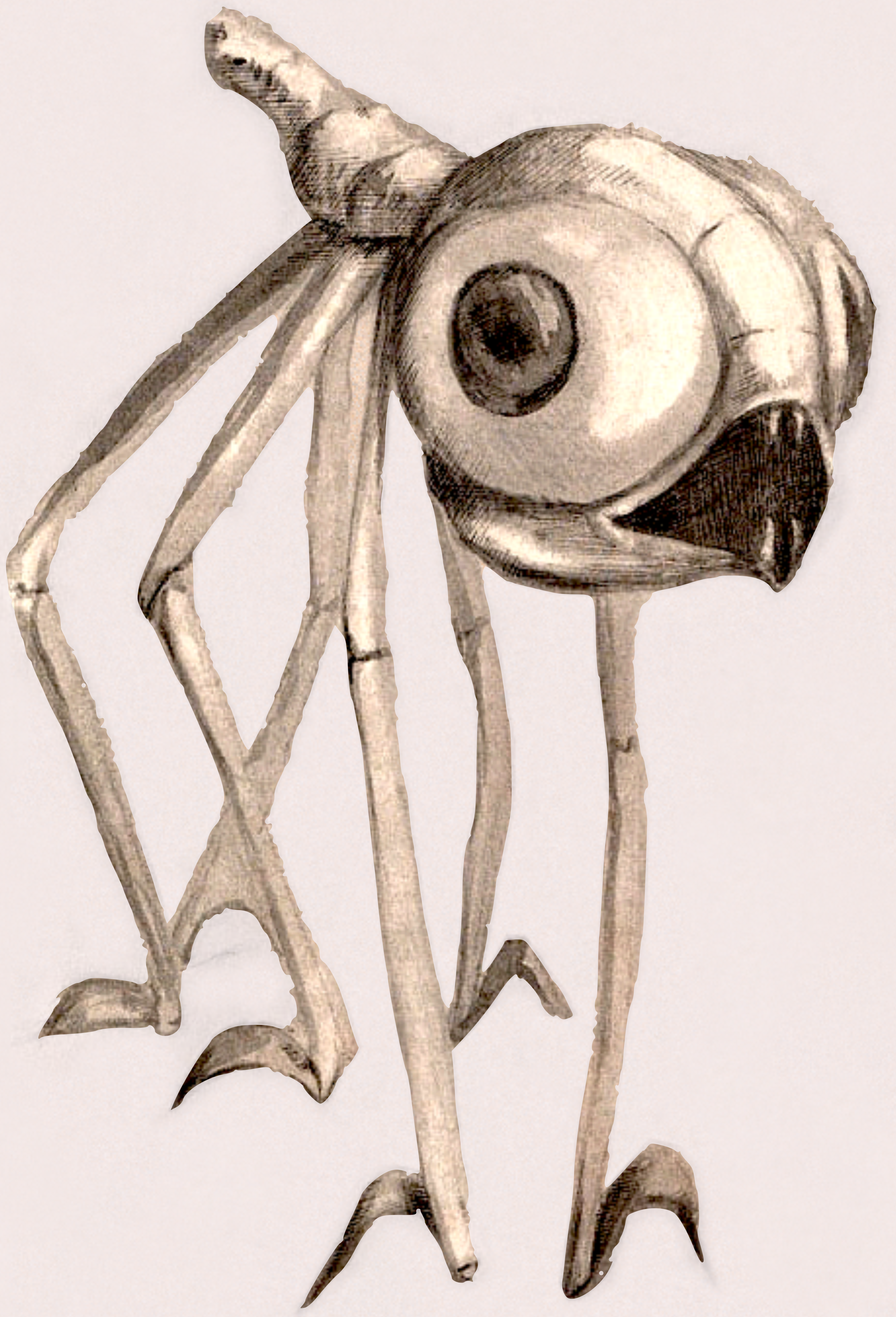
DAL evolution: query runners

- pg-promise, slonik, aiosql
- SQL statements in client code, hand-built dynamic SQL
- Managed cursors
- Results marshaling
- SQL organization, sometimes

DAL evolution: introspecting API generators

- PostgREST, Postgraphile, Hasura
- Take the place of an application/service
- Build their own SQL
- Logic in functions and views
- May be extensible through plugins





DAL evolution: what's current?

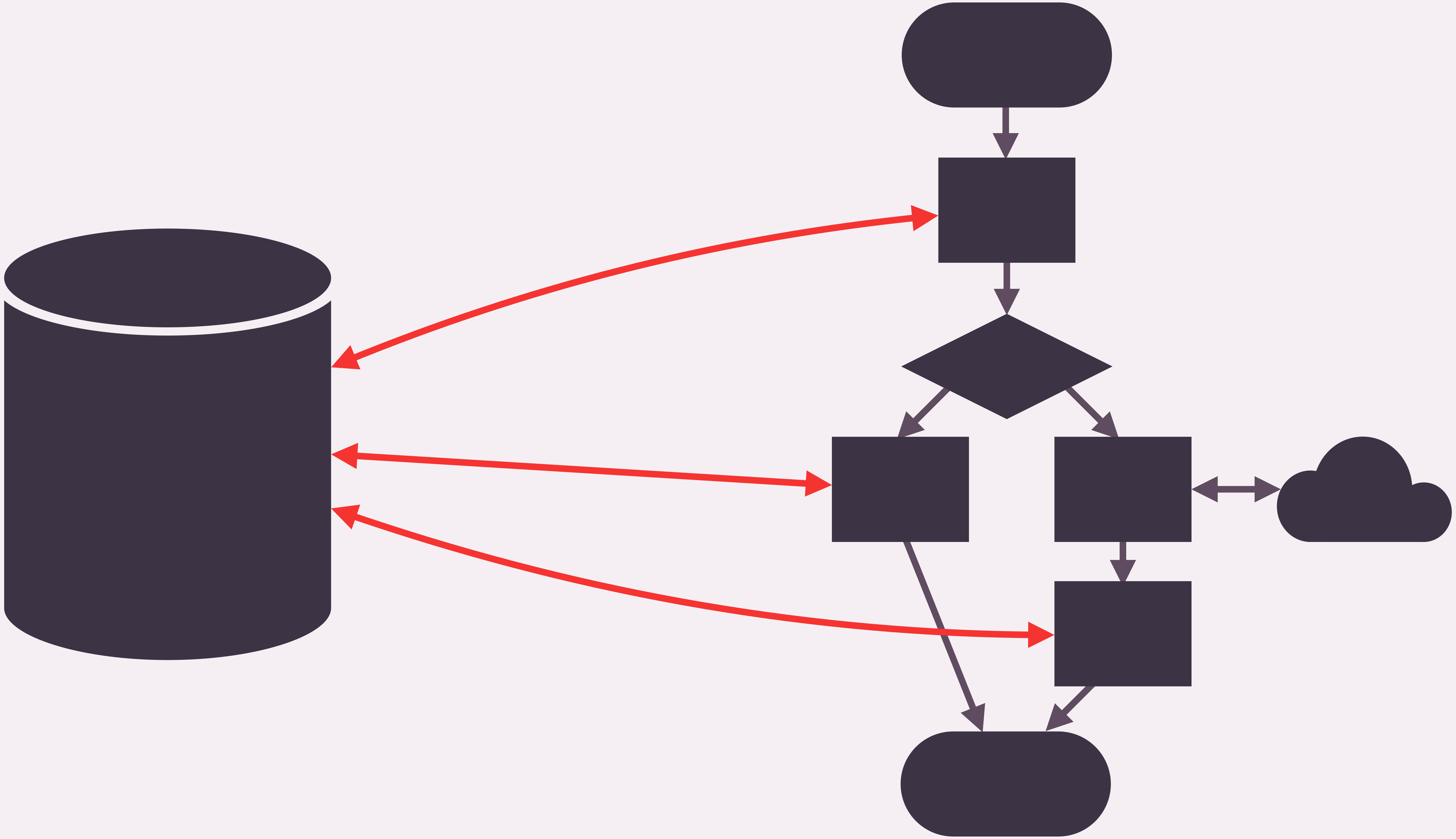
- Query runners are a strict improvement on yoloSQL
- O/RM problems are well understood
- Beyond that, It Depends

Let’s do some testing!

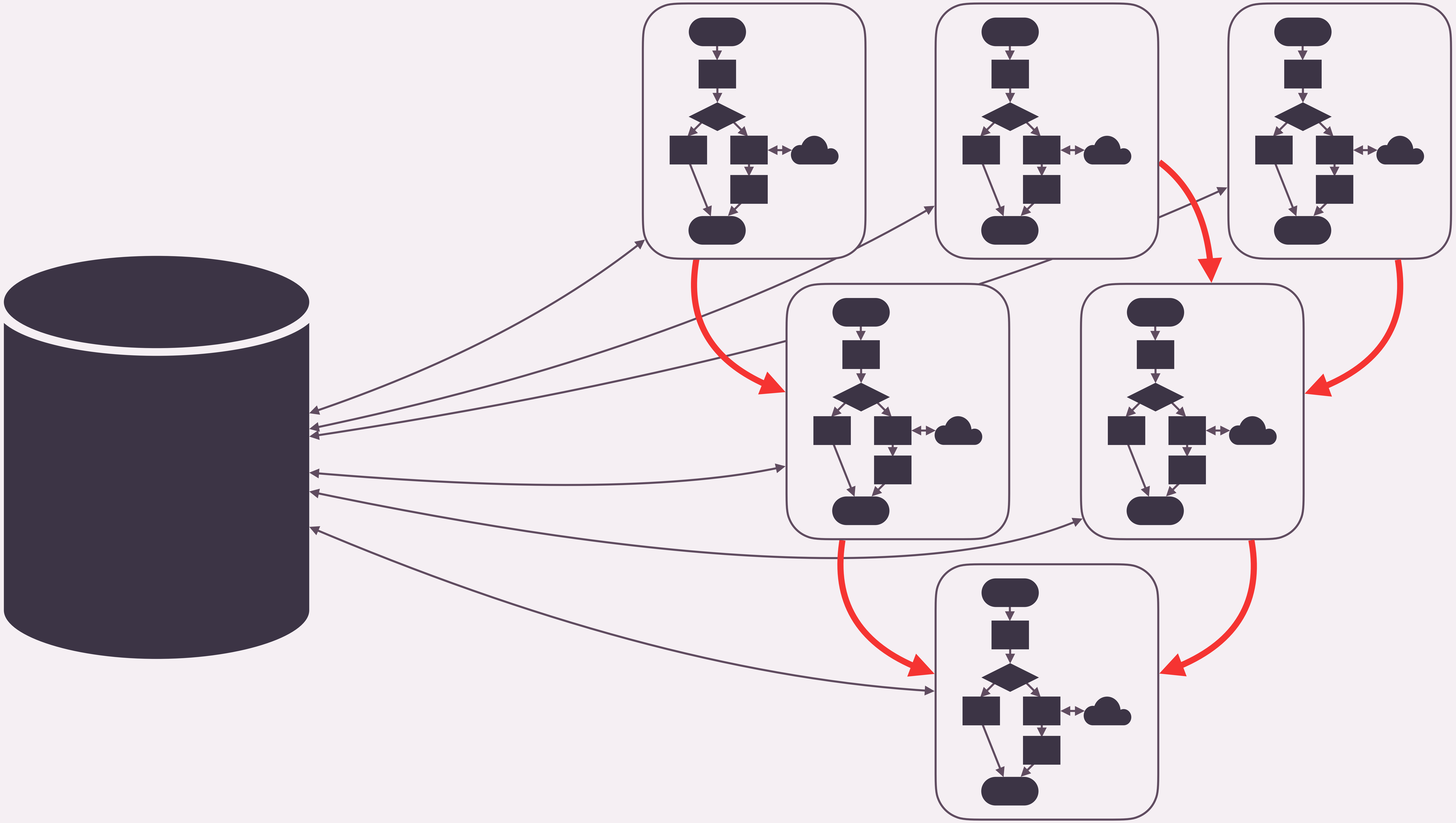
Testing and transactions

- Transactions avoid side effects — when available
- Nontransactional tests must clean up or tolerate pollution
- Parallel tests can lock or violate each other’s constraints

Testing flows and feedback loops



Testing flows and feedback loops



Testing and data prerequisites

Rely on data from earlier tests

Maintain complete testing datasets

Pick one!

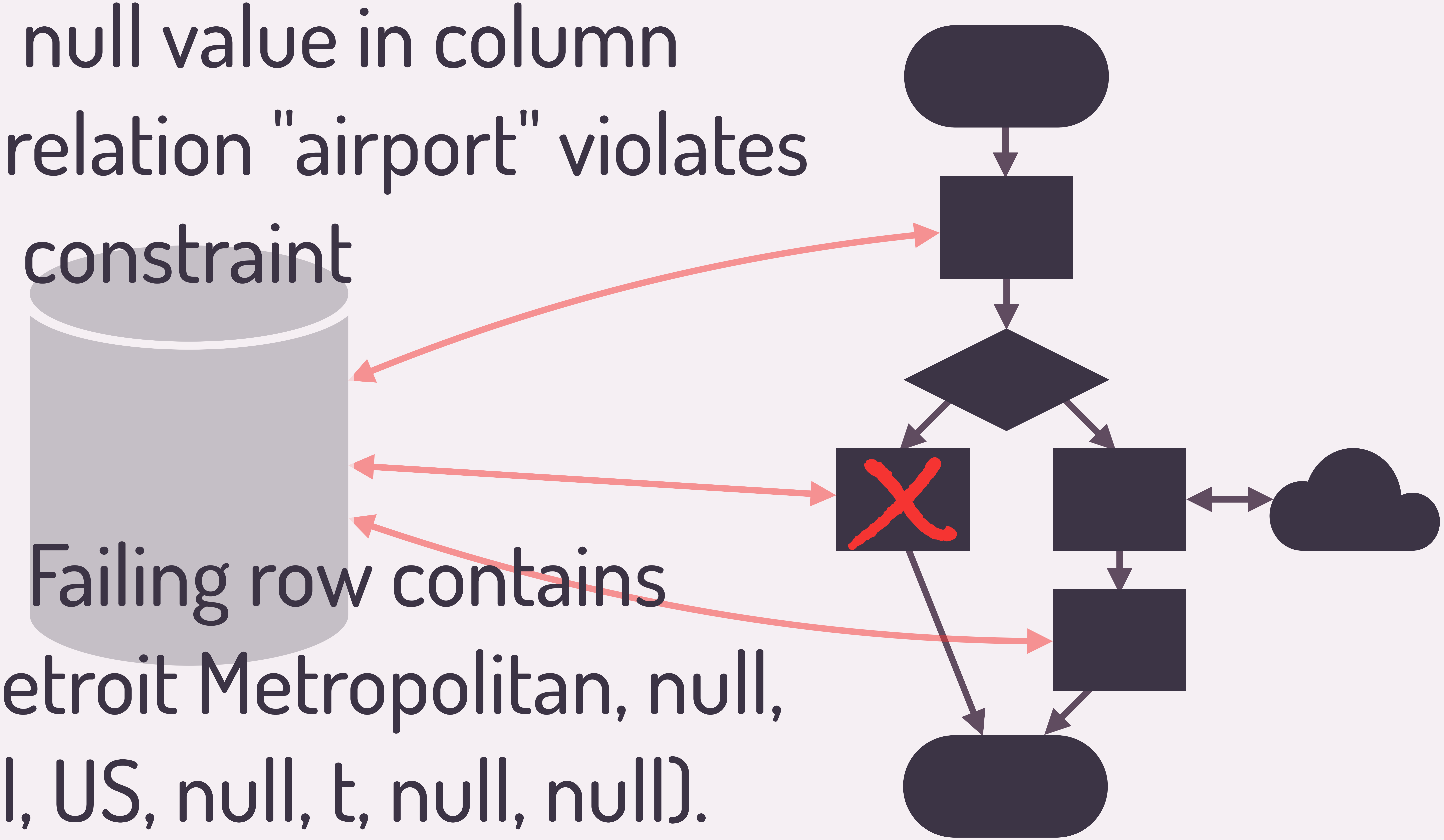
Bespoke test setup code

Orchestrate mini-fixtures

Let’s debug some problems!

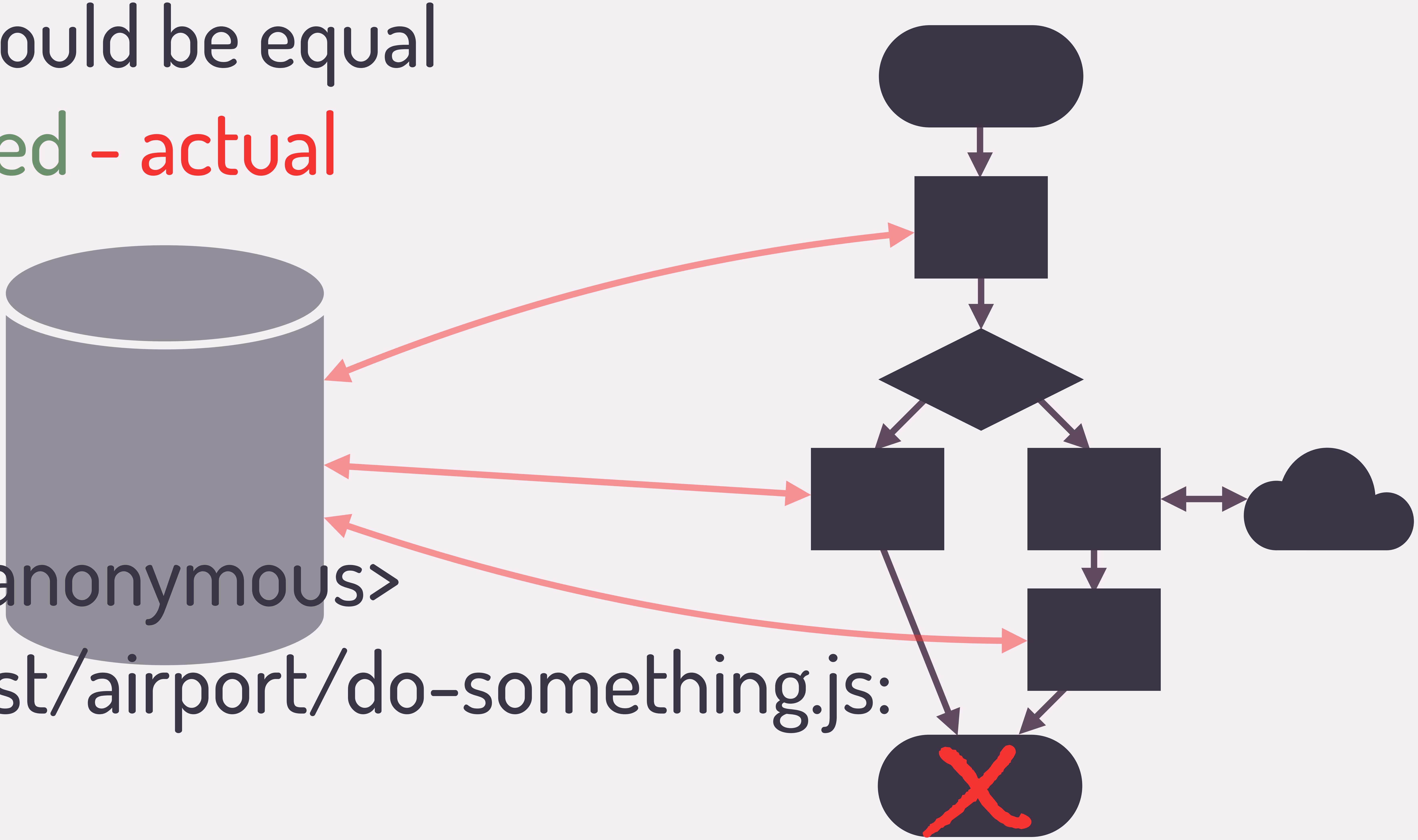
The best case

```
ERROR:  null value in column
"city" of relation "airport" violates
not-null constraint

DETAIL:  Failing row contains
(DTW, Detroit Metropolitan, null,
null, null, US, null, t, null, null).
```



The worst case

```
Error: should be equal
+ expected - actual
-2
+1

at Test.<anonymous>
(file://test/airport/do-something.js:
287:5)
```



Following database execution flows

- Reproducing problems involves experimentation
- Single, file-based logging facility
- Functions are a logging boundary
- No profiler or session-activity collector

Following database execution flows

- pldebugger and friends
- Set up conditions locally
- Set breakpoints
- Construct function call or DML to trigger execution

Gallant uses the typewriter very carefully.

Following database execution flows

- Spray RAISE WARNING into everything plausible
- Reprise problem system behavior and watch

Goofus bangs on the typewriter and breaks it.

But is it fast?

- Performance is good until it isn’t
- EXPLAIN tells you what but not why
- Statistics are arcane

But is it fast?

- Shipping is the only way to find out what works
- Experimentation in production required
- Targeted band-aid fixes aren’t usually possible

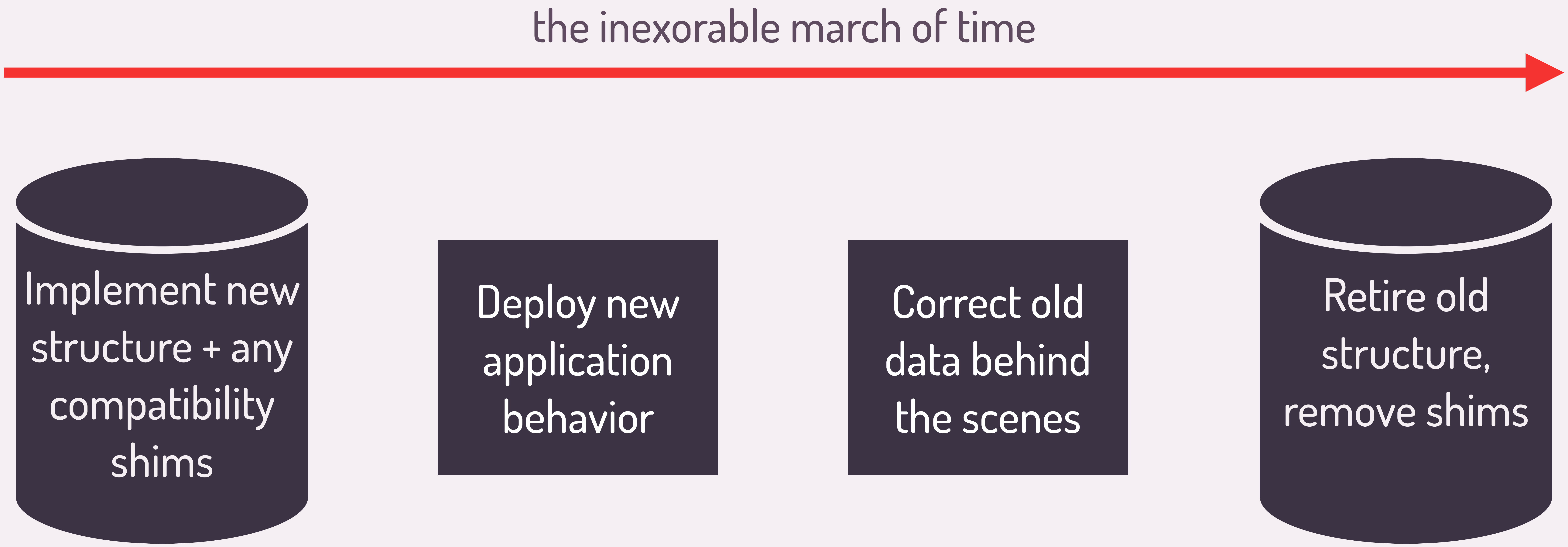
Let’s evolve our schema!

Schema evolution, part II: guarantees

- Atomicity: transactional DDL
- Idempotence: CREATE OR REPLACE, where available
- Performance: concurrent builds and IF (NOT) EXISTS

Wait, what does ACCESS EXCLUSIVE mean?

Schema evolution, part II: execution



the inexorable march of time

Implement new structure + any compatibility shims

Deploy new application behavior

Correct old data behind the scenes

Retire old structure, remove shims

Let’s recap!

All happy user bases are alike; each unhappy user base is unhappy in its own way

— Leo Tolstoy, probably

Different Expectations

- Schema evolution at the speed of requirements changes
- System legibility at par with application code
- Gentle scale/performance curve

Different Interfaces

- Expanded system boundary
- Data access needs not well-served by SQL
- Higher levels of abstraction and automation

Fin