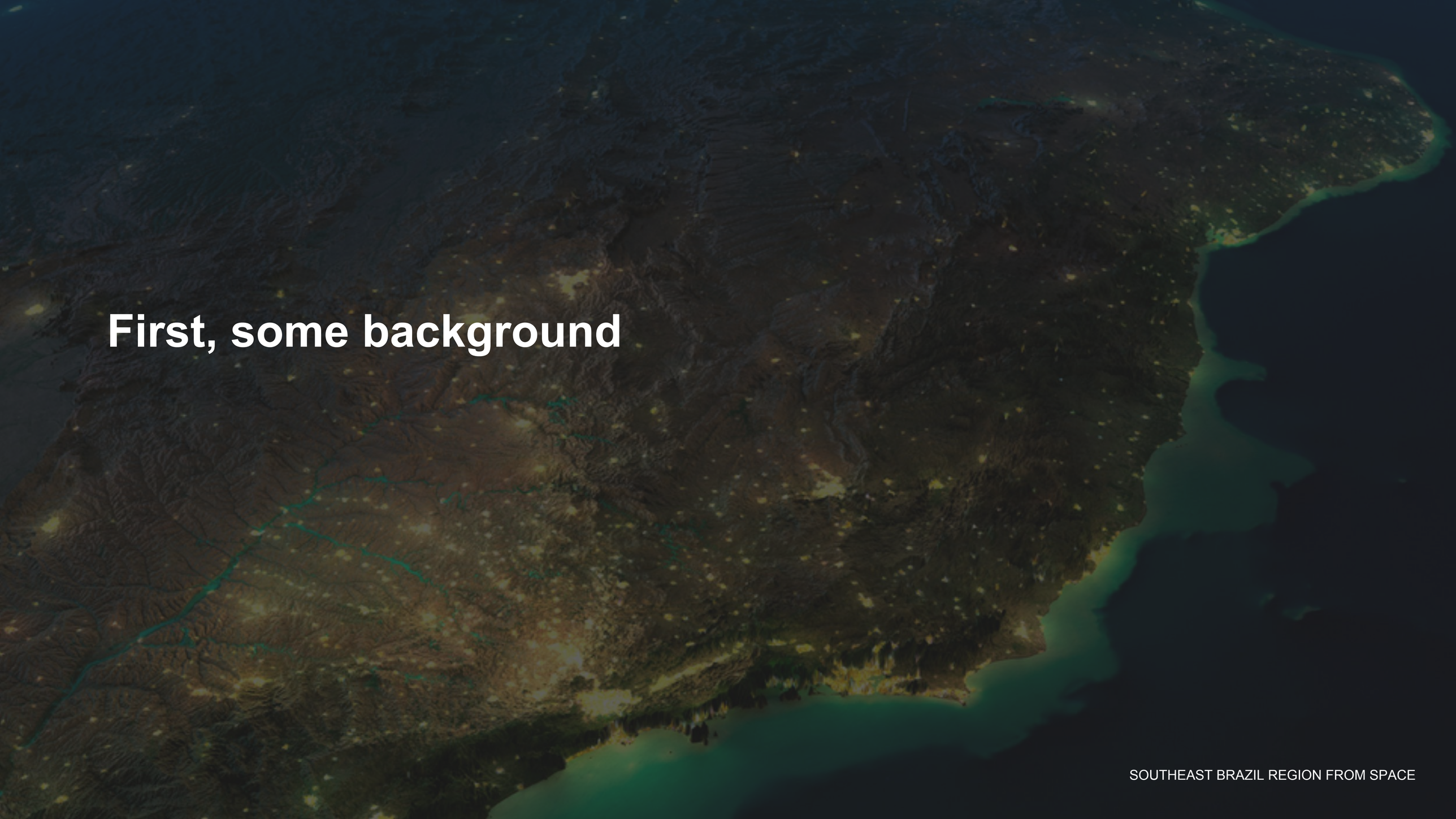
How we test at

Phillip Mates

First, some background



SOUTHEAST BRAZIL REGION FROM SPACE

What is Nubank?

- mobile-driven bank account and credit card services for ~4 million Brazilians
- Started in 2014 & we've been using clojure from the start
- 200 clojure microservices with 100+ engineers
- includes a small eng office in Berlin focused on data infrastructure

Why tests are important to us

Tests help with

- preventing mistakes in production
- cross-team mobility

Why tests are important to us

Tests help with

- preventing mistakes in production
- cross-team mobility

But tests aren’t a perfect solution:

- just like code, you can write unreadable tests
- there are maintenance costs
- buggy tests and buggy test frameworks

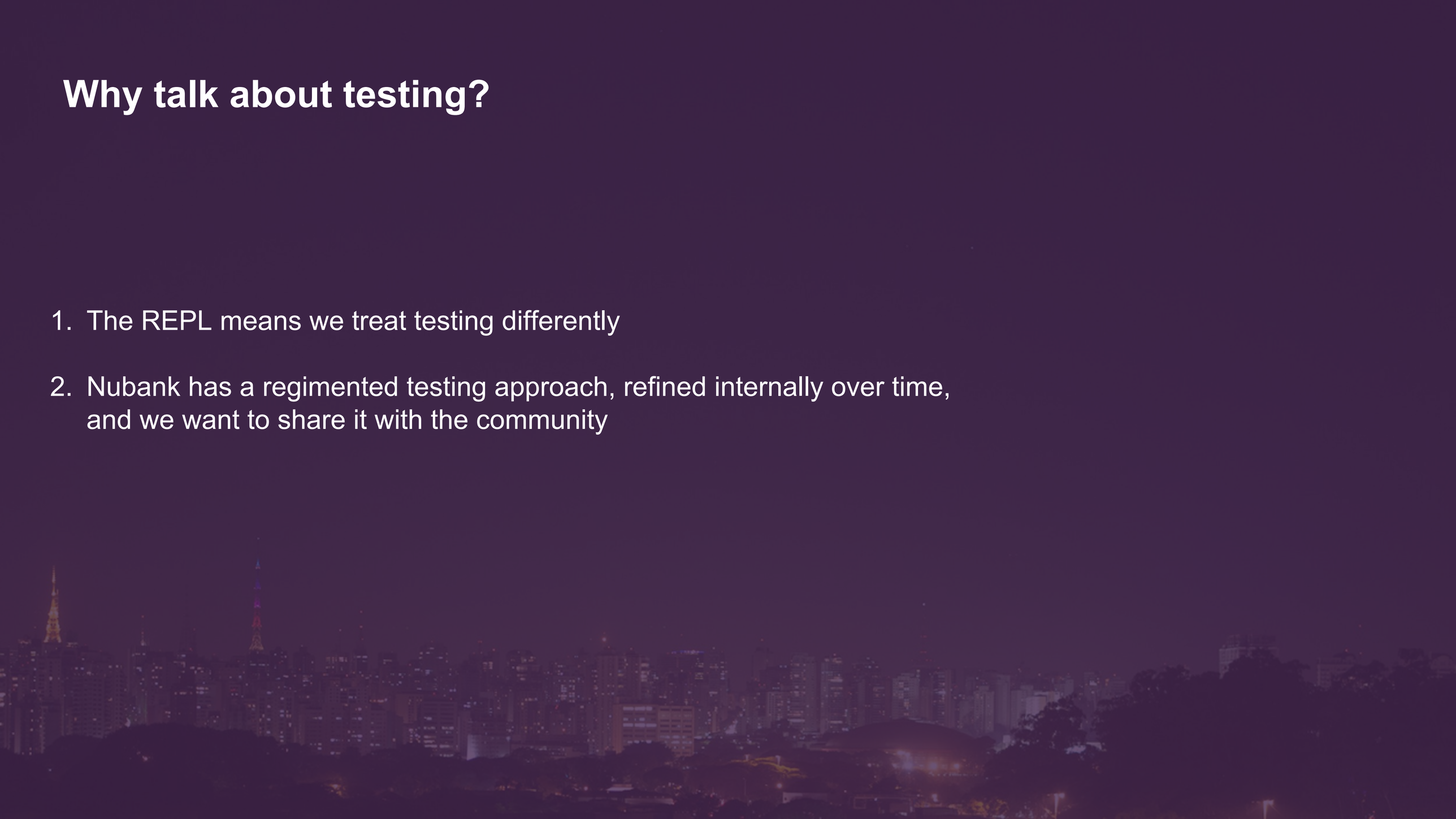
Why talk about testing?

1. The REPL means we treat testing differently

Why talk about testing?

1. The REPL means we treat testing differently

2. Nubank has a regimented testing approach, refined internally over time, and we want to share it with the community



What we are testing

Microservices with a ports & adapters structure

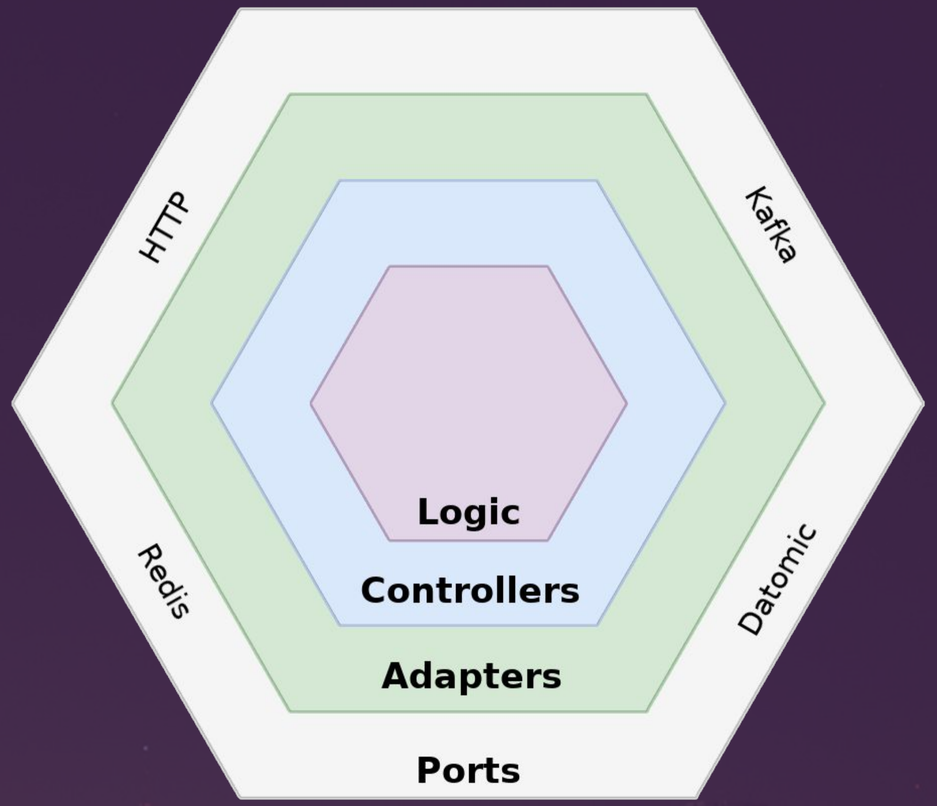
Logic: pure business logic

Controllers: glue that coordinates calls between ports and pure logic

Adapters: convert between external and internal data representations

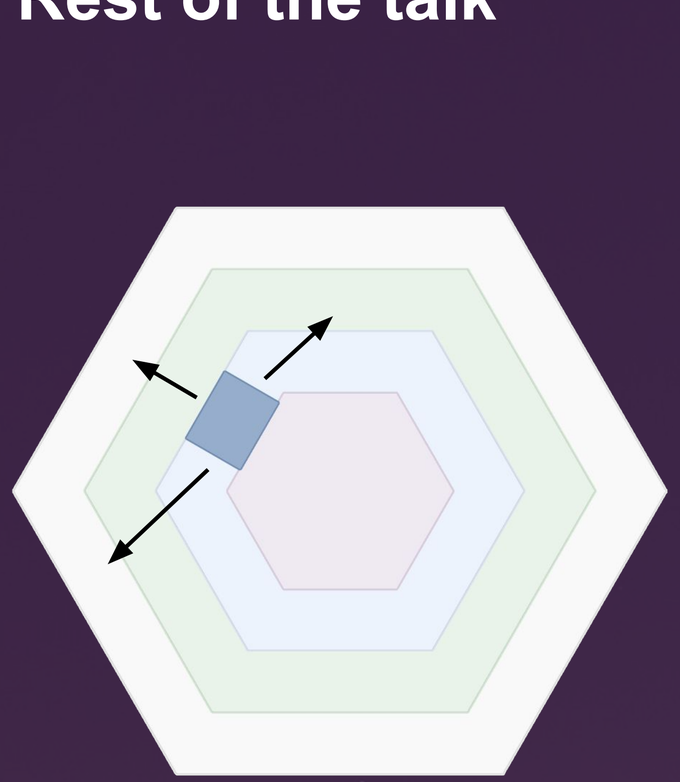
Ports: http, pubsub, database, other I/O

“Components” abstraction is used to organize ports

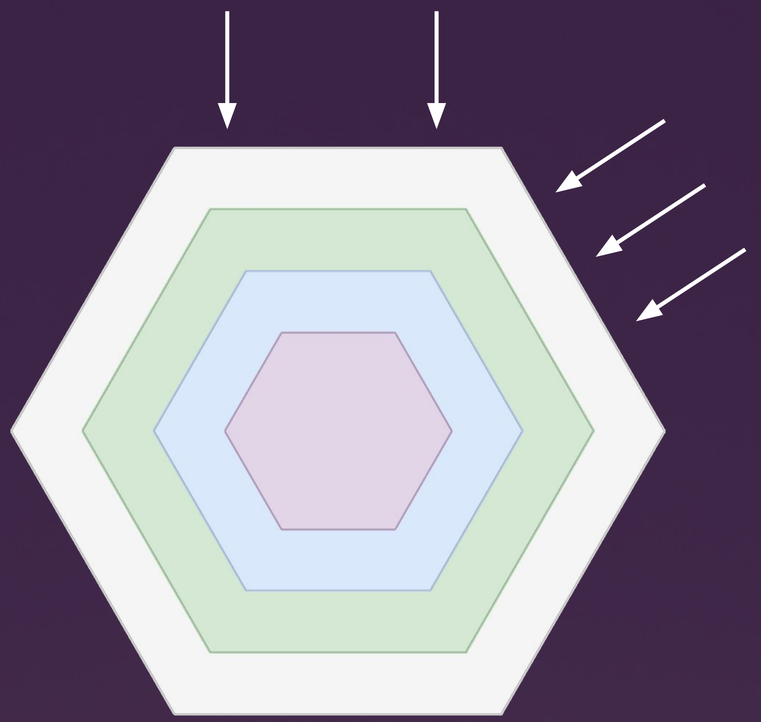


service

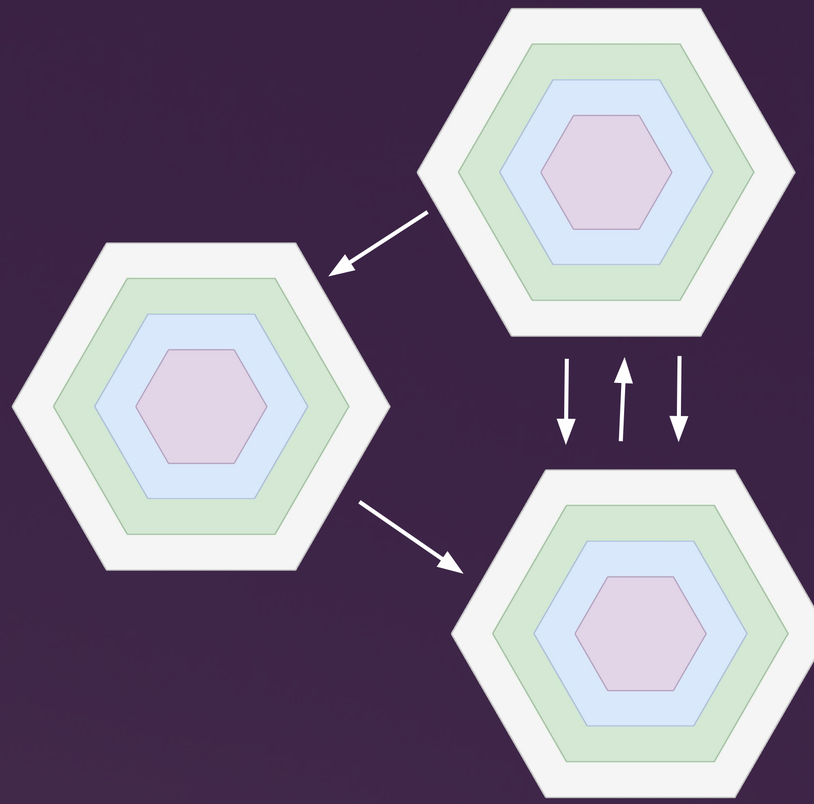
Rest of the talk



unit

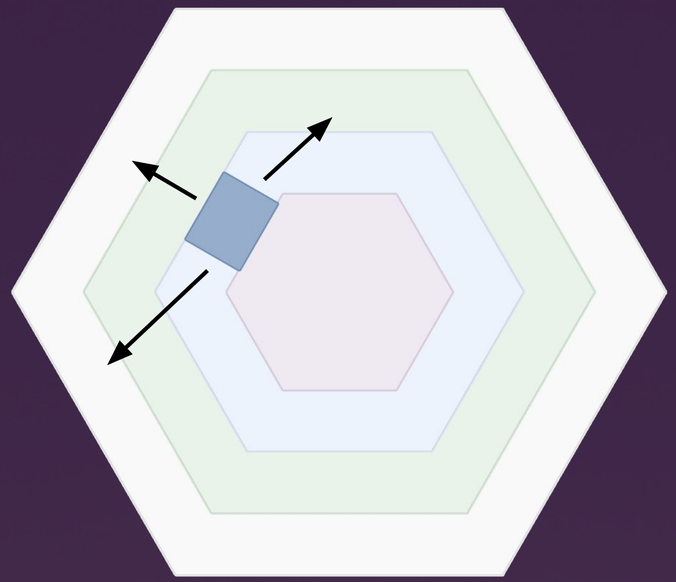


integration



end-to-end

unit tests



Anatomy of a unit test in Midje

```
1 (ns basic-microservice-example.logic)
2
3 (defn new-account [customer-id customer-name]
4   {:id          (java.util.UUID/randomUUID)
5    :name        customer-name
6    :tags        ["savings-beta" "verified"]
7    :customer-id customer-id})
```

Anatomy of a unit test in Midje

```clojure
(ns basic-microservice-example.logic)

(defn new-account [customer-id customer-name]
  {:id          (java.util.UUID/randomUUID)
   :name        customer-name
   :tags        ["savings-beta" "verified"]
   :customer-id customer-id})
```

```clojure
(ns basic-microservice-example.logic-test ...)

(def customer-id (java.util.UUID/randomUUID))

(fact "New account generation"
  (new-account customer-id "Tim Maia") => (just {:id          uuid?
                                                 :name        "Tim Maia"
                                                 :tags        sequential?
                                                 :customer-id customer-id}))

```

Anatomy of a unit test in Midje

```
1  (ns basic-microservice-example.controller)
2
3
4
5
6
7
8  (defn create-account! [customer-id storage http]
9    (let [customer (get-customer customer-id http)
10         account  (logic/new-account customer-id (:customer-name customer))]
11     (db.saving-account/add-account! account storage)
12     account))
```

Anatomy of a unit test in Midje

```
1  (ns basic-microservice-example.controller)
2
3  (defn get-customer [customer-id http]
4    (let [customer-url  (str "http://customer-service.com/customer/" customer-id)]
5      (:body (http-client/req! http {:url    customer-url
6                                     :method :get}))))
7
8  (defn create-account! [customer-id storage http]
9    (let [customer (get-customer customer-id http)
10         account  (logic/new-account customer-id (:customer-name customer))]
11     (db.saving-account/add-account! account storage)
12     account))
```

Anatomy of a unit test in Midje

```
(ns basic-microservice-example.controller)

(defn get-customer [customer-id http]
  (let [customer-url  (str "http://customer-service.com/customer/" customer-id)]
    (:body (http-client/req! http {:url    customer-url
                                   :method :get}))))

(defn create-account! [customer-id storage http]
  (let [customer (get-customer customer-id http)
        account  (logic/new-account customer-id (:customer-name customer))]
    (db.saving-account/add-account! account storage)
    account))
```

```
(ns basic-microservice-example.controller-test)

(def customer-id (UUID/randomUUID))

(fact "Sketching account creation"
  (create-account! customer-id ..storage.. ..http..) => (just {:id          uuid?
                                                               :name        "Tom Zé"
                                                               :tags        (just ["verified" "savings-beta"]
                                                                                  :in-any-order)
                                                               :customer-id customer-id})
```

Anatomy of a unit test in Midje

```
1  (ns basic-microservice-example.controller)
2
3  (defn get-customer [customer-id http]
4    (let [customer-url  (str "http://customer-service.com/customer/" customer-id)]
5      (:body (http-client/req! http {:url    customer-url
6                                     :method :get}))))
7
8  (defn create-account! [customer-id storage http]
9    (let [customer (get-customer customer-id http)
10         account  (logic/new-account customer-id (:customer-name customer))]
11     (db.saving-account/add-account! account storage)
12     account))
```

```
1  (ns basic-microservice-example.controller-test)
2
3  (def customer-id (UUID/randomUUID))
4
5  (fact "Sketching account creation"
6    (create-account! customer-id ..storage.. ..http..) => (just {:id          uuid?
7                                                                 :name        "Tom Zé"
8                                                                 :tags        (just ["verified" "savings-beta"]
9                                                                                    :in-any-order)
10                                                                :customer-id customer-id})
11   (provided
12     (get-customer customer-id ..http..) => {:customer-name "Tom Zé"}
13     (db/add-account! (contains {:name "Tom Zé"}) ..storage..) => irrelevant))
14
```

unit tests: take-aways

Pros

- Allows quick iteration with stubbing functionality
- Light-weight and ideal for testing pure logic

Cons

- High-touch: due to stubs and lots of entry points
- Can get cluttered

Take-away:

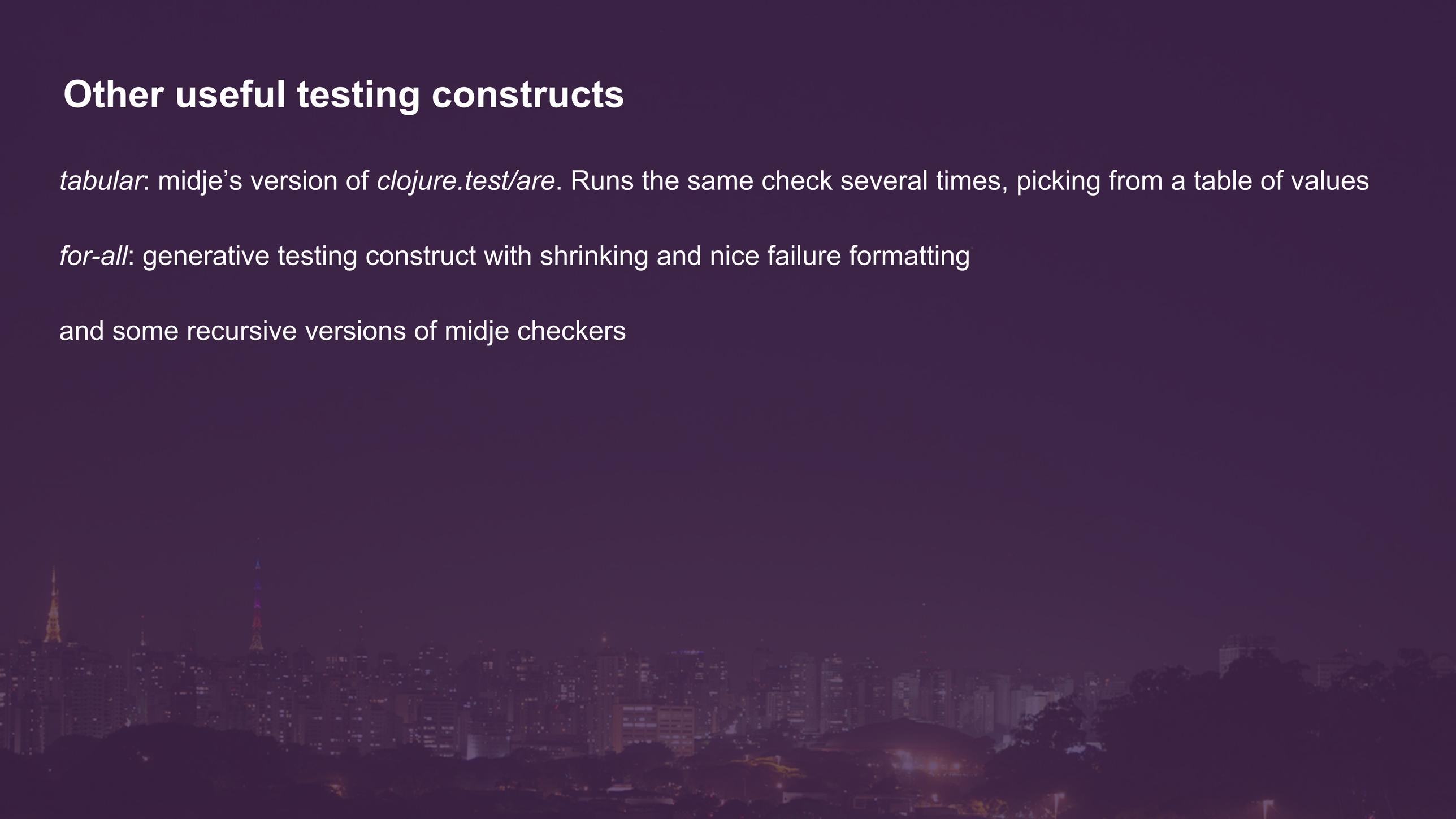
should be used to test core logic and guide one through incremental code dev

Other useful testing constructs

tabular: midje's version of *clojure.test/are*. Runs the same check several times, picking from a table of values

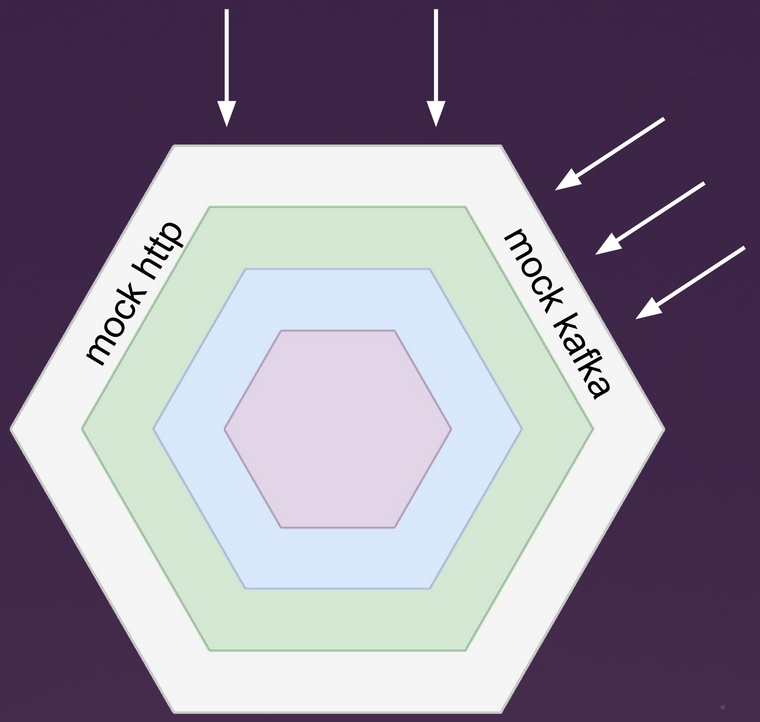
for-all: generative testing construct with shrinking and nice failure formatting

and some recursive versions of midje checkers



Demo: checking nested data with matcher-combinators

Integration



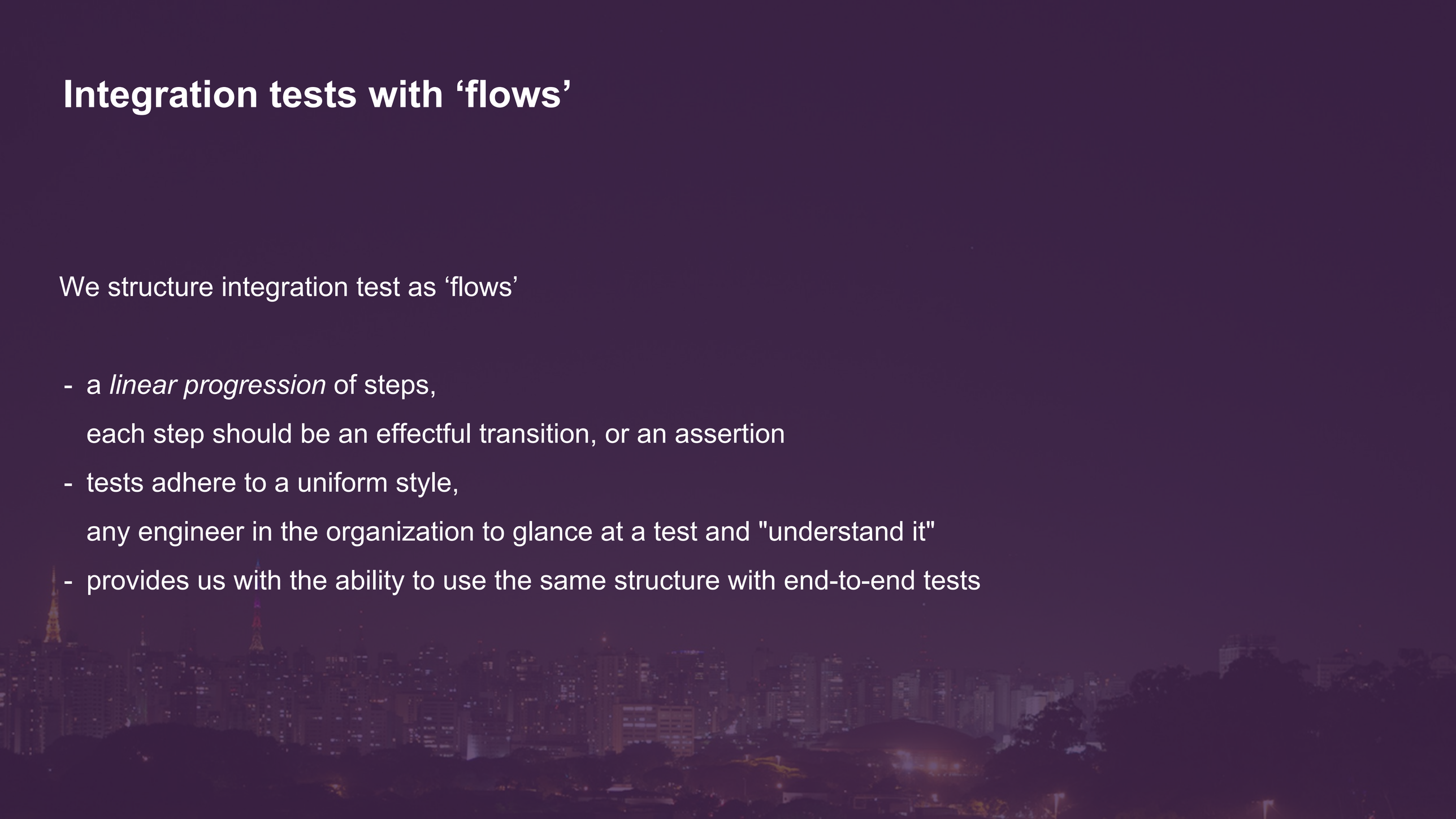
Integration tests

- Cover an entire flow of business logic
- Granularity of a single service. The service code remains unmocked, but mocks for http, pubsub, and other ports are needed
- Trigger code paths via the service’s boundaries, making assertions on outgoing messages

Integration tests with ‘flows’

We structure integration test as ‘flows’

- a *linear progression* of steps,
  each step should be an effectful transition, or an assertion
- tests adhere to a uniform style,
  any engineer in the organization to glance at a test and "understand it"
- provides us with the ability to use the same structure with end-to-end tests

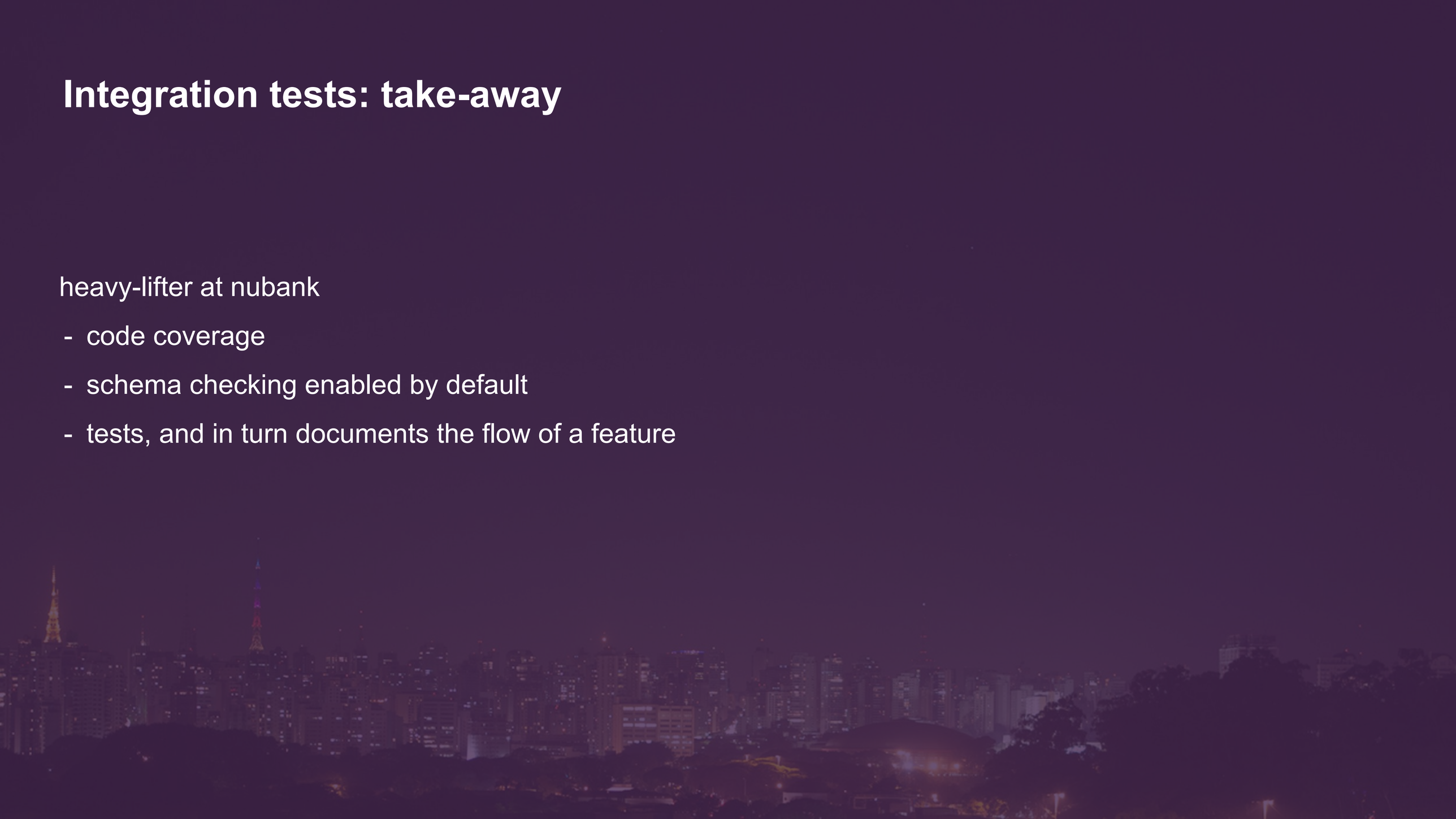


Demo: flow

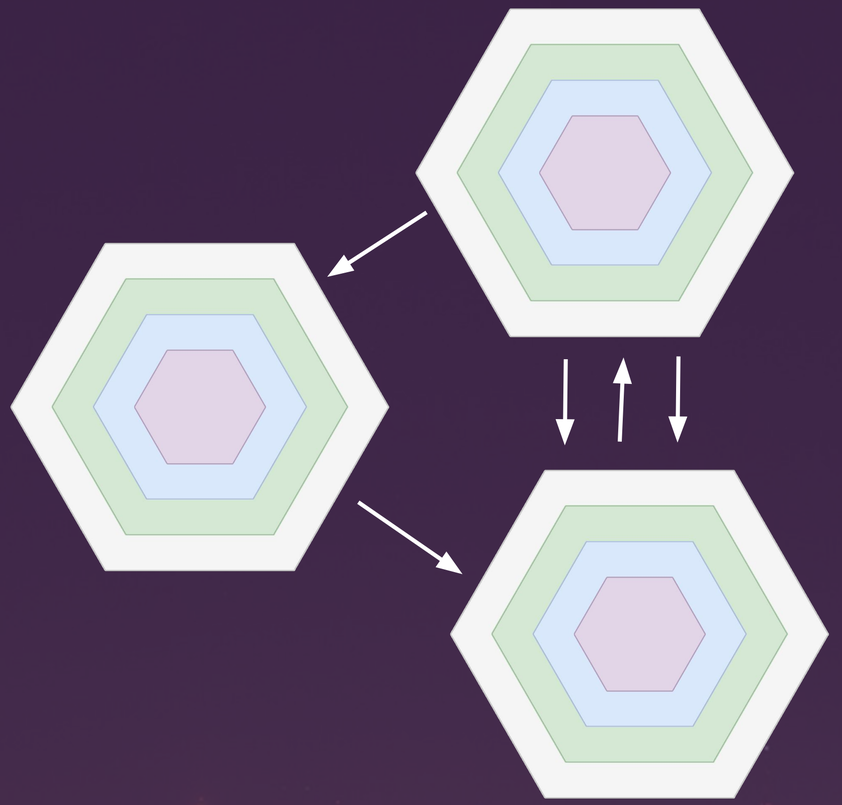
Integration tests: take-away

heavy-lifter at nubank

- code coverage
- schema checking enabled by default
- tests, and in turn documents the flow of a feature



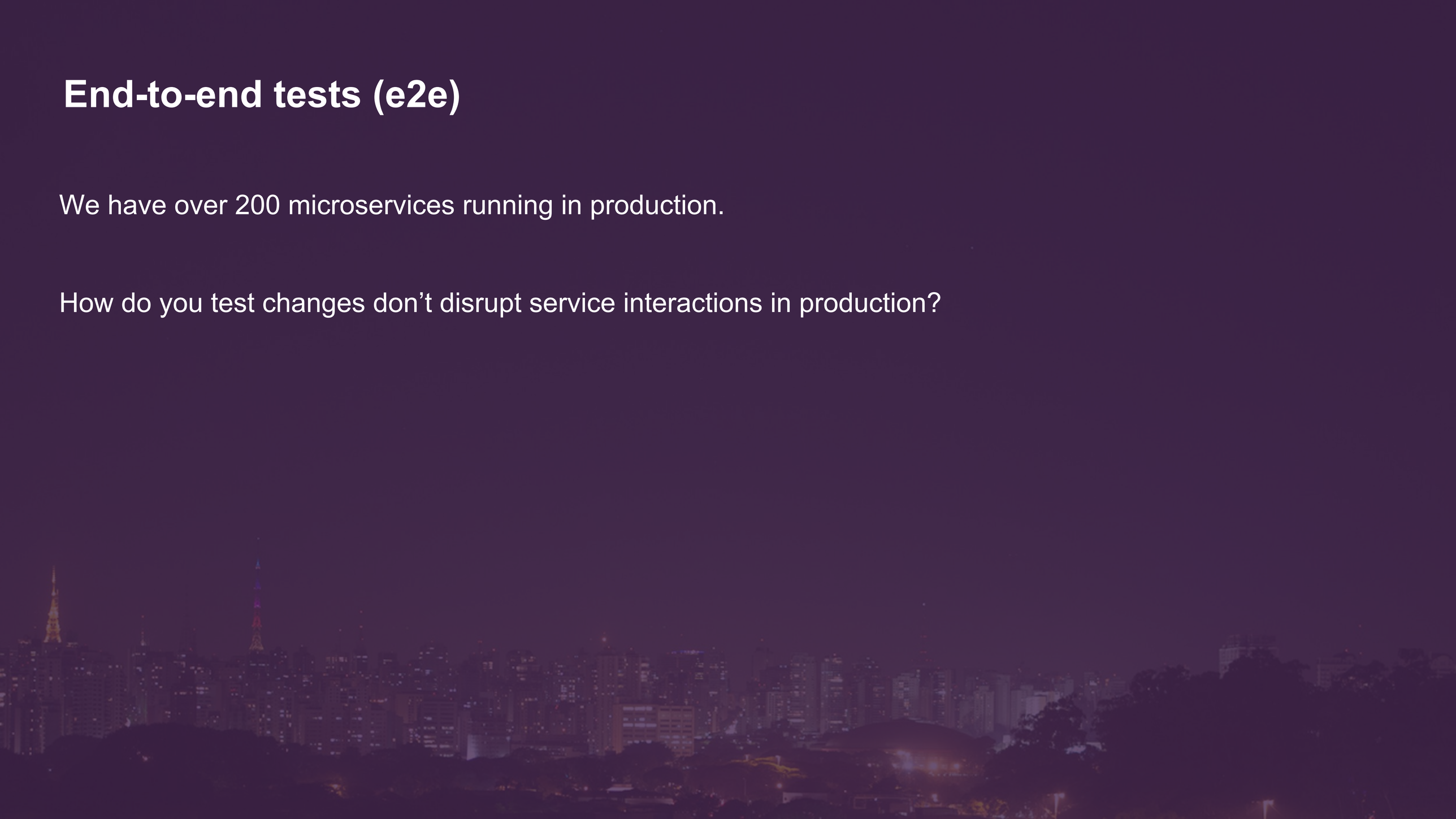
End-to-end tests (e2e)



End-to-end tests (e2e)

We have over 200 microservices running in production.

How do you test changes don’t disrupt service interactions in production?



End-to-end tests (e2e)

We have over 200 microservices running in production.

How do you test changes don’t disrupt service interactions in production?

Make a copy of our production stack and simulate actions via the flow abstraction.

- Gave us confidence in the early years
- Last check before changes are deployed; takes 15 minutes
- Test only happy path
- Tests can be flaky
- Means it is a huge bottleneck

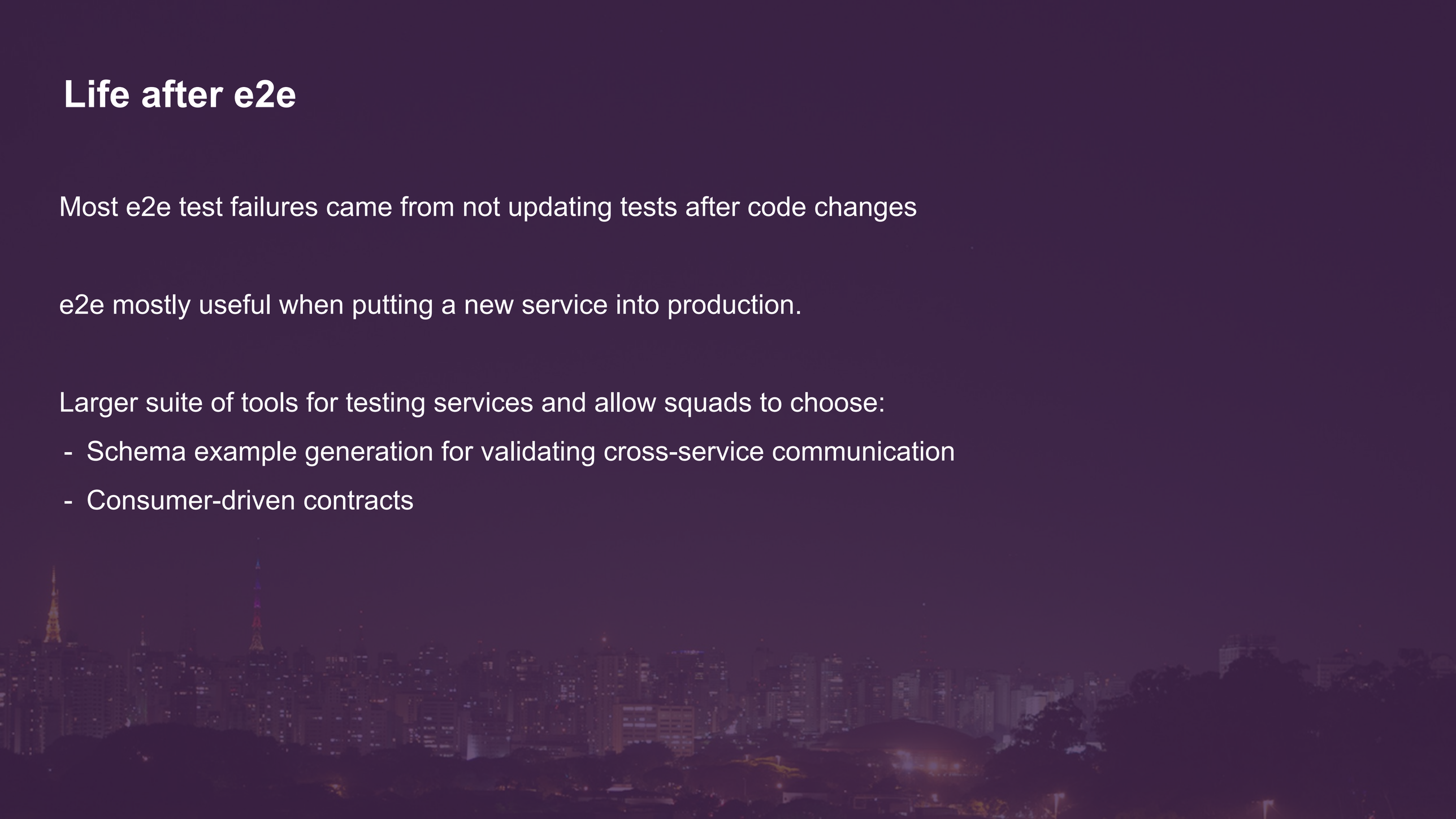
Life after e2e

Most e2e test failures came from not updating tests after code changes

e2e mostly useful when putting a new service into production.

Larger suite of tools for testing services and allow squads to choose:

- Schema example generation for validating cross-service communication
- Consumer-driven contracts

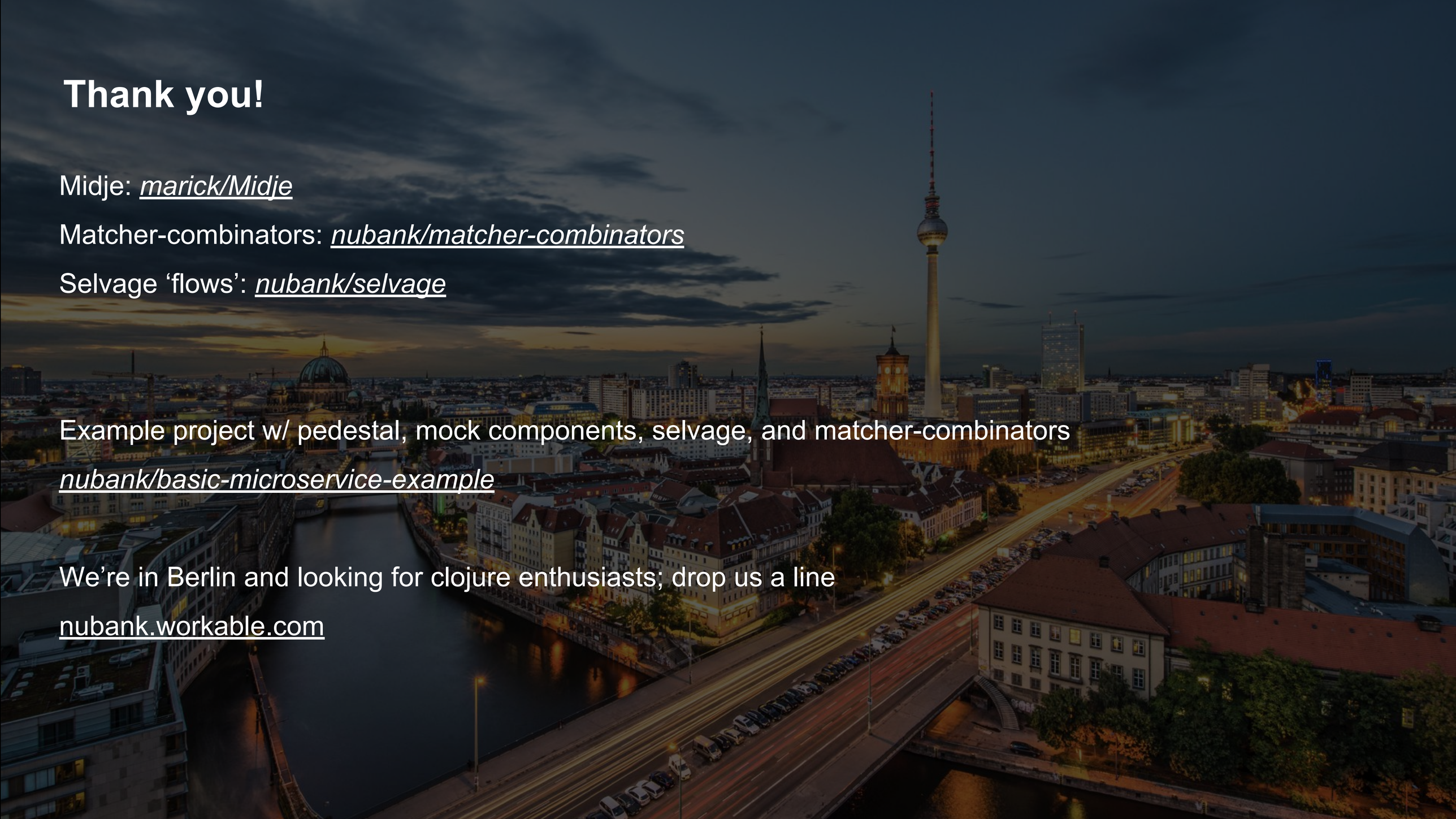


Wrap up

Unit tests are good for pure logic and iterating on code

Integration tests are the work-horse at Nubank

Cross-service testing is hard: moving from e2e to a suite of lightweight validation tools



Thank you!

Midje: *marick/Midje*

Matcher-combinators: *nubank/matcher-combinators*

Selvage ‘flows’: *nubank/selvage*

Example project w/ pedestal, mock components, selvage, and matcher-combinators

nubank/basic-microservice-example

We’re in Berlin and looking for clojure enthusiasts; drop us a line

nubank.workable.com