
Klarna.
Brave New World
Haskell ~~and PureScript~~ in Production

Yet another talk about how you should be using X and how it solves all your problems*

*No, not really.

Me

• Research Fellow	EPFL	2016
• Lab Engineer	EPFL	2016-2017
• Senior Engineer	Klarna	2017-

Klarna.

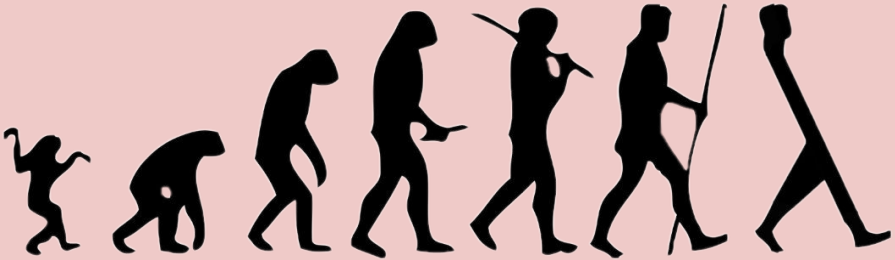
Write Haskell for Snoop Dogg


Get smoooth.


SMOOTH

How our teams went from zero to Monad*

*Or how we tried to go the full route with Scala.



But first, a survey!

- Immutable Data

- Immutable Data
- Lambdas

- Immutable Data
- Lambdas
- Pattern matching

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers
- Free Monads

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers
- Free Monads
- Extensible Effects

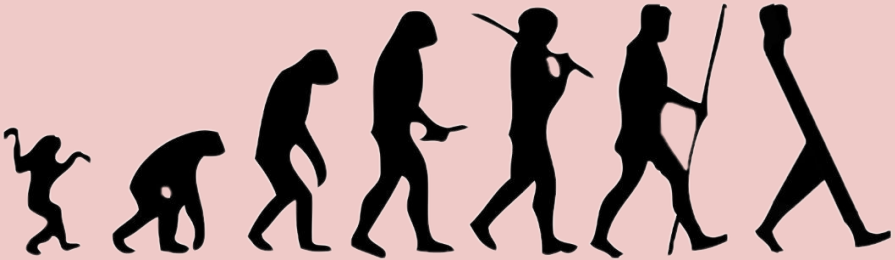
- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers
- Free Monads
- Extensible Effects
- Type Families

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers
- Free Monads
- Extensible Effects
- Type Families
- Functional Dependencies

- Immutable Data
- Lambdas
- Pattern matching
- Algebraic Data Types
- Type Classes
- Referential Transparency
- Type Constructors
- Higher Kinded Types
- Functors
- Monoids
- Applicatives
- Monads
- Foldable
- Comonads
- Rank-N types
- Traversable
- Bifunctors
- Monad Transformers
- Free Monads
- Extensible Effects
- Type Families
- Functional Dependencies
- Profunctor Optics

How our teams went from zero to Monad*

*Or how we tried to go the full route with Scala.



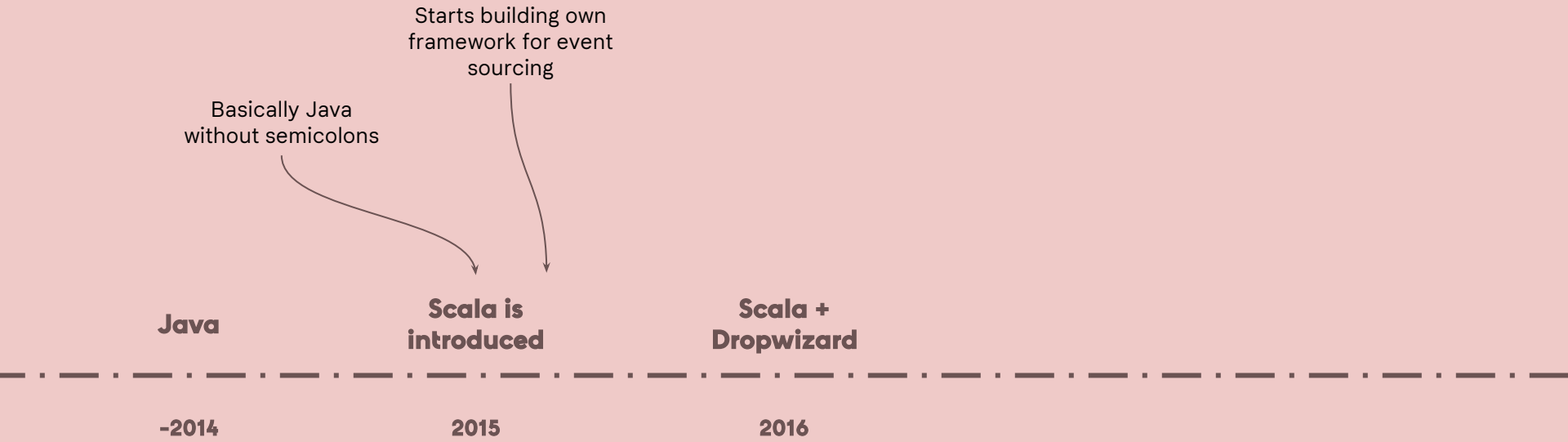
Java

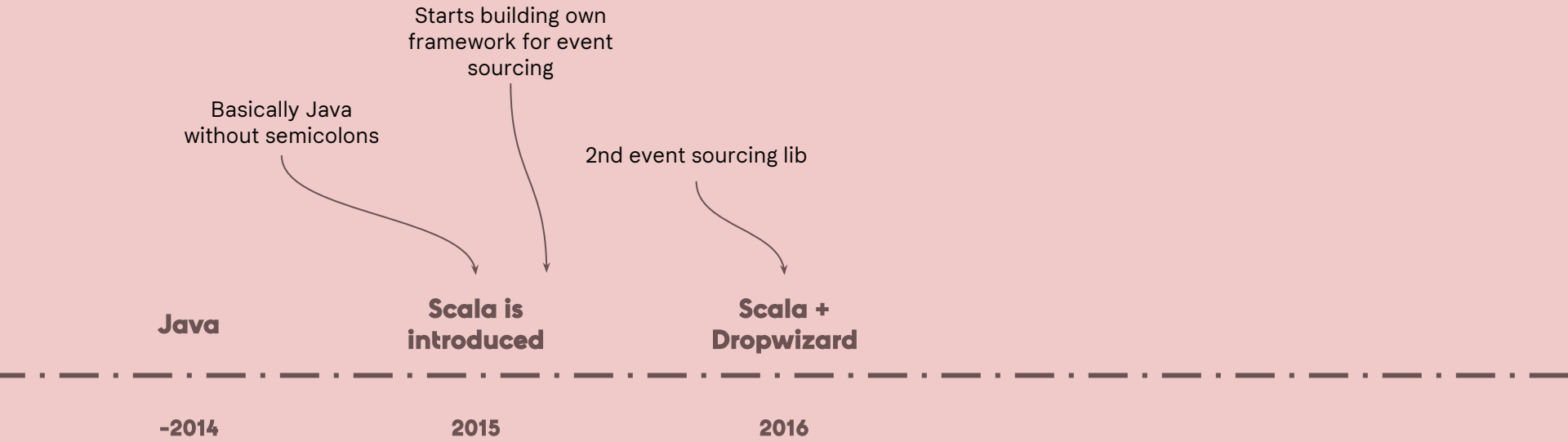
-2014

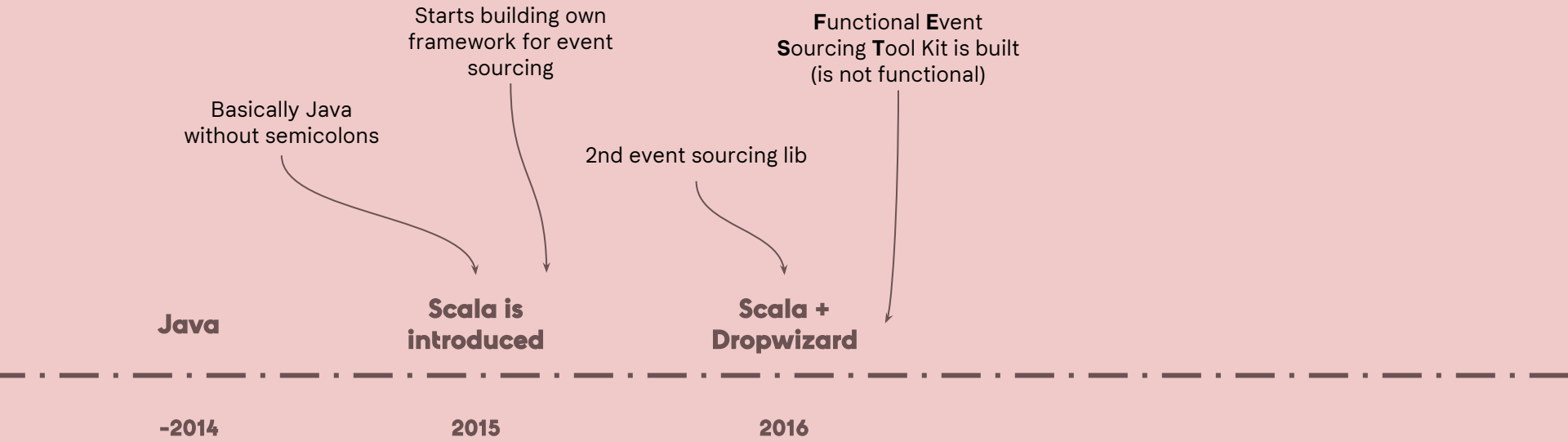

Java
Scala is introduced
-2014
2015

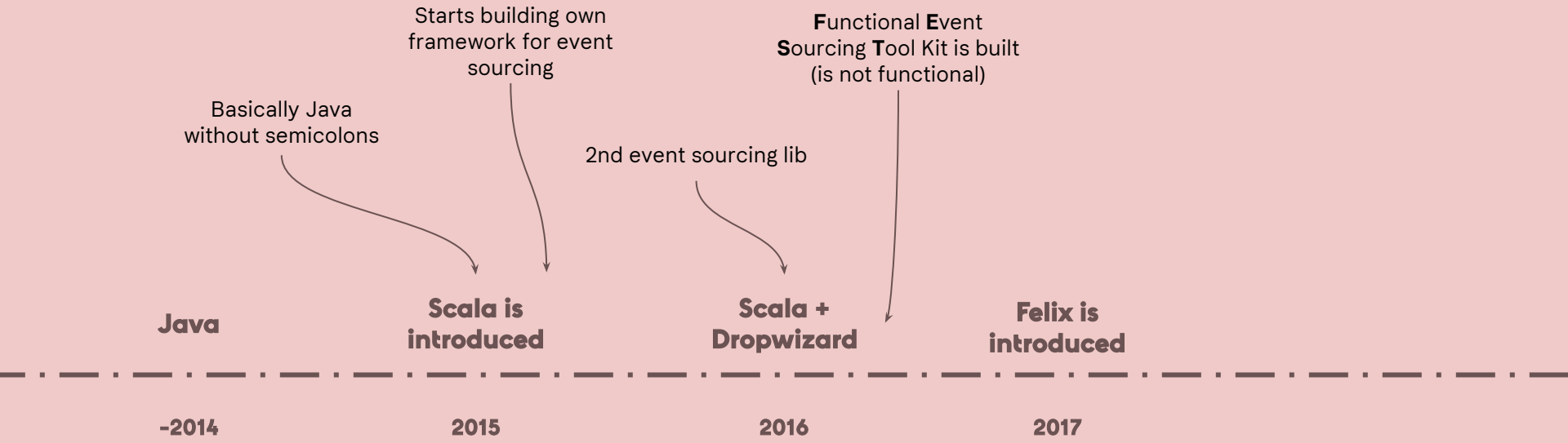

Basically Java
without semicolons
Java
Scala is
introduced
-2014
2015

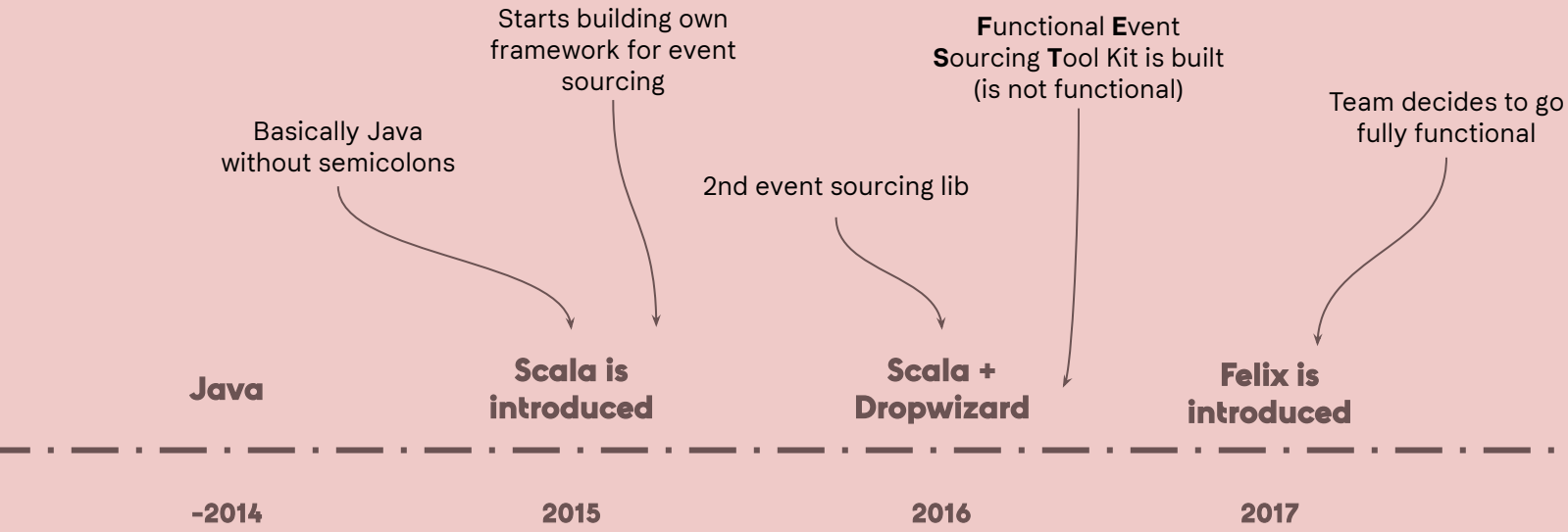

Starts building own framework for event sourcing
Basically Java without semicolons
Java
Scala is introduced
-2014
2015

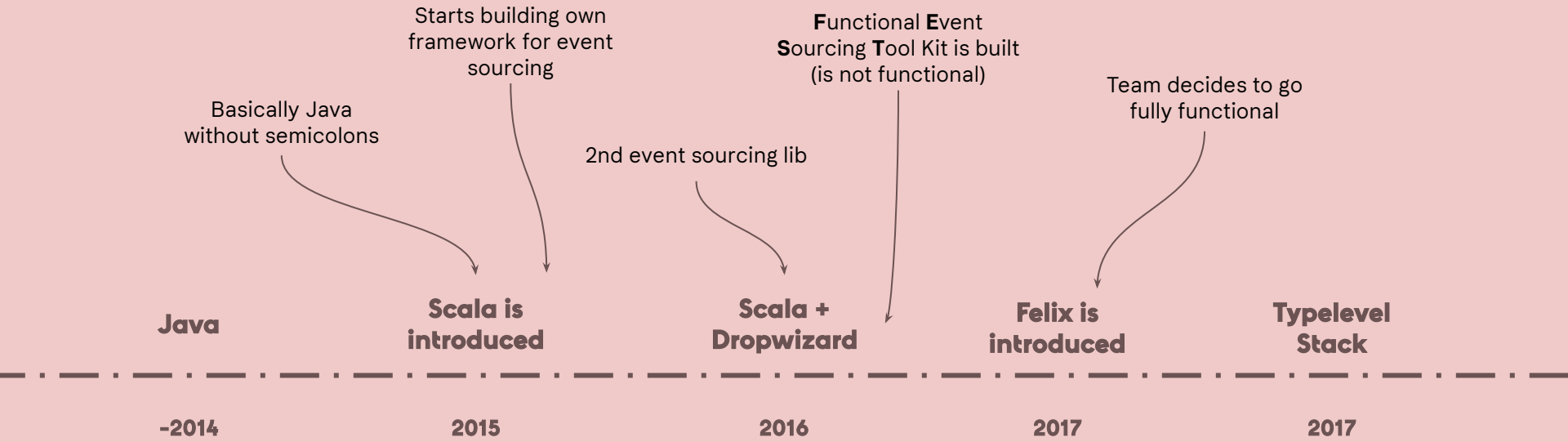

Starts building own framework for event sourcing
Basically Java without semicolons
Java
Scala is introduced
Scala + Dropwizard
-2014
2015
2016

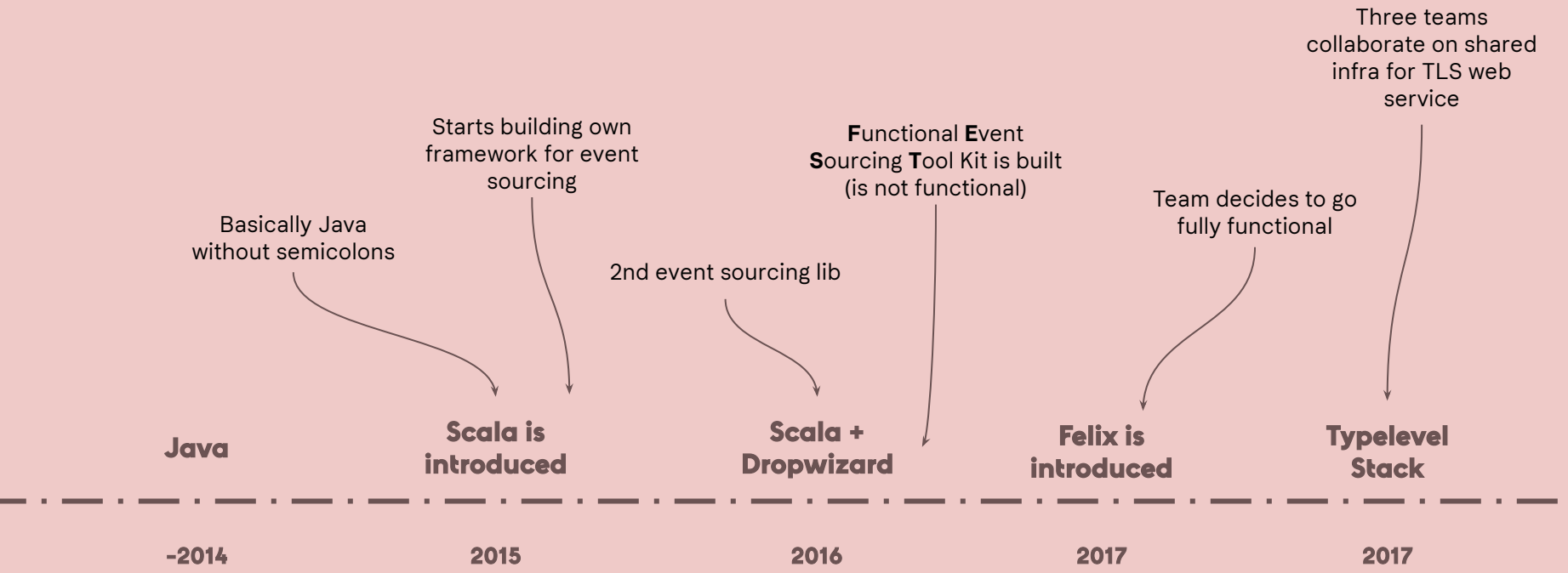

Starts building own framework for event sourcing
Basically Java without semicolons
2nd event sourcing lib
Java
Scala is introduced
Scala + Dropwizard
-2014
2015
2016

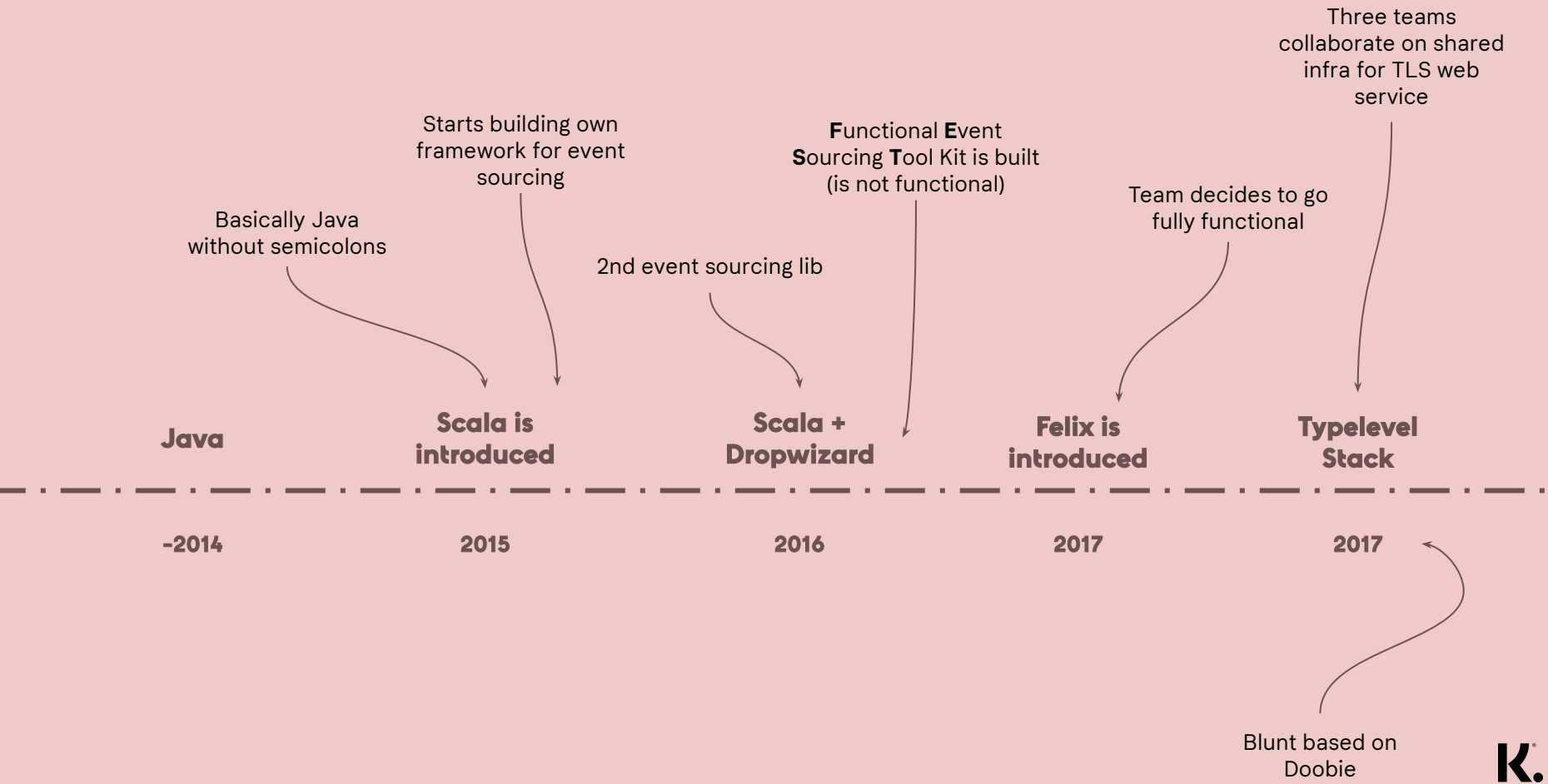

Starts building own framework for event sourcing
Functional Event Sourcing Tool Kit is built (is not functional)
Basically Java without semicolons
2nd event sourcing lib
Java
Scala is introduced
Scala + Dropwizard
-2014
2015
2016


Starts building own framework for event sourcing
Functional Event Sourcing Tool Kit is built (is not functional)
Basically Java without semicolons
2nd event sourcing lib
Java
Scala is introduced
Scala + Dropwizard
Felix is introduced
-2014
2015
2016
2017


Basically Java without semicolons
Starts building own framework for event sourcing
2nd event sourcing lib
Functional Event Sourcing Tool Kit is built (is not functional)
Team decides to go fully functional
Java
Scala is introduced
Scala + Dropwizard
Felix is introduced
-2014
2015
2016
2017


Basically Java without semicolons
Starts building own framework for event sourcing
2nd event sourcing lib
Functional Event Sourcing Tool Kit is built (is not functional)
Team decides to go fully functional
Java
Scala is introduced
Scala + Dropwizard
Felix is introduced
Typelevel Stack
-2014
2015
2016
2017
2017


Three teams collaborate on shared infra for TLS web service
Starts building own framework for event sourcing
Functional Event Sourcing Tool Kit is built (is not functional)
Team decides to go fully functional
Basically Java without semicolons
2nd event sourcing lib
Java
Scala is introduced
Scala + Dropwizard
Felix is introduced
Typelevel Stack
-2014
2015
2016
2017
2017


Three teams collaborate on shared infra for TLS web service
Starts building own framework for event sourcing
Functional Event Sourcing Tool Kit is built (is not functional)
Team decides to go fully functional
Basically Java without semicolons
2nd event sourcing lib
Java
Scala is introduced
Scala + Dropwizard
Felix is introduced
Typelevel Stack
-2014
2015
2016
2017
2017
Blunt based on Doobie

You had us at purely functional.

Yeah, I guess we're all pretty much convinced by now.

Disclaimer: I still  Scala

Brace yourselves for the bashing.

Pros

- Java and JVM
- Type Safety
  - Explicit errors
  - Newtypes
- Edge validation
- Strong community around TLS

Cons

- Java and JVM
- Hard to teach
  - Syntax & Extensions
  - Unopinionated
- Unclear threading model
- Type Inference and FP in Scala
- Mix paradigms

Pros

- Mature Ecosystem
- Type Safety
  - Explicit errors
  - Newtypes
- Edge validation
- Strong community around TLS

Cons

- Java and JVM
- Hard to teach
  - Syntax & Extensions
  - Unopinionated
- Unclear threading model
- Type Inference and FP in Scala
- Mix paradigms

Felix Mulder
@FelixMulder

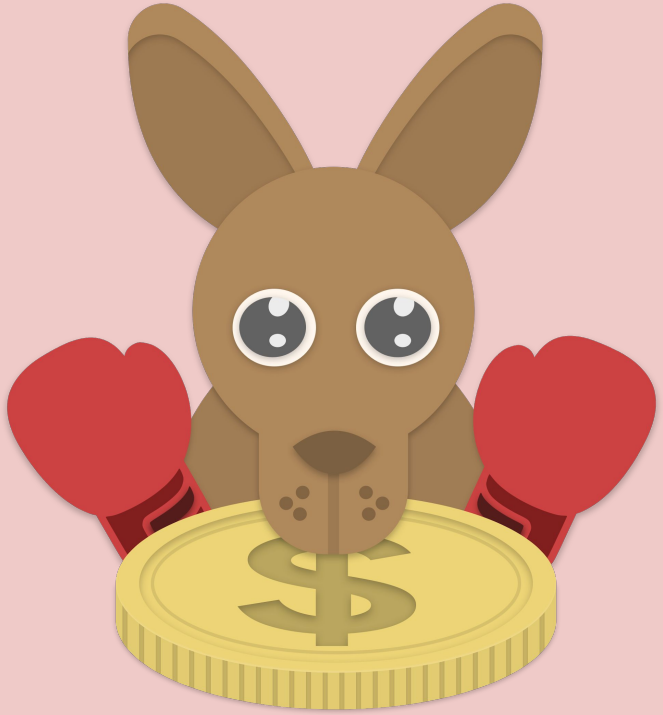
Unpopular opinion: I don't think purely functional programming is idiomatic to Scala the way it's currently done. The techniques pioneered in Haskell should not be used verbatim in Scala 🤷

2:48 PM - 30 Apr 2019

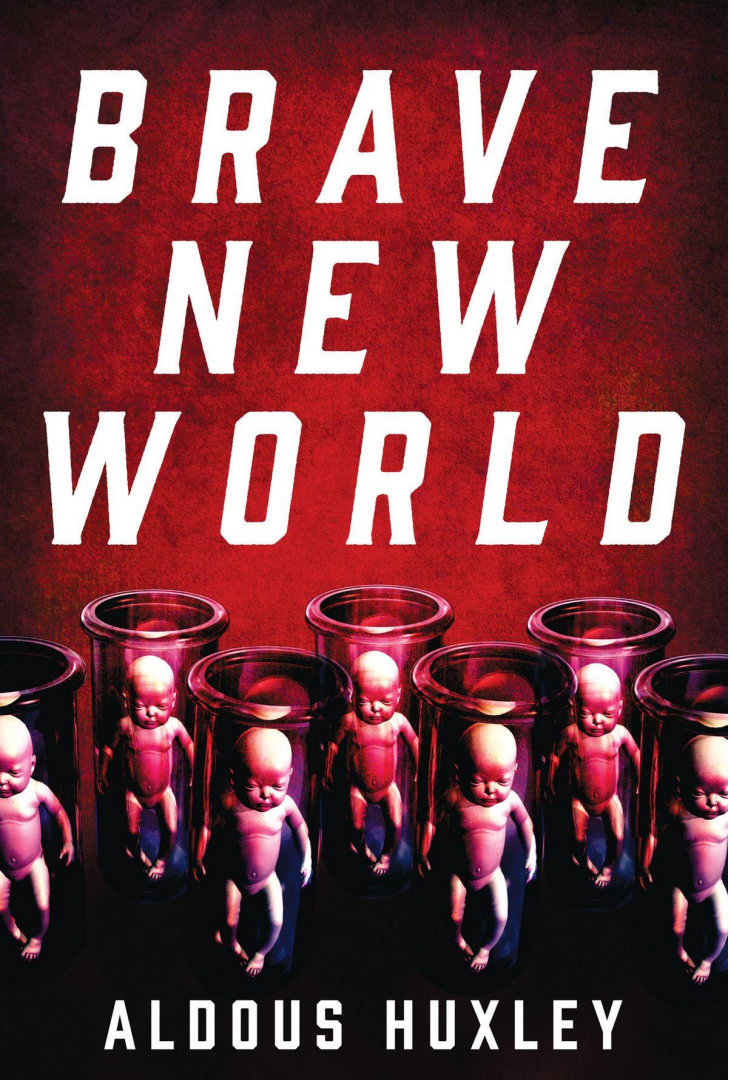
Disclaimer:
I still  Scala

Sorry for all the bashing.

“Klarna is going to be the AWS of banking”



BUCKAROOS

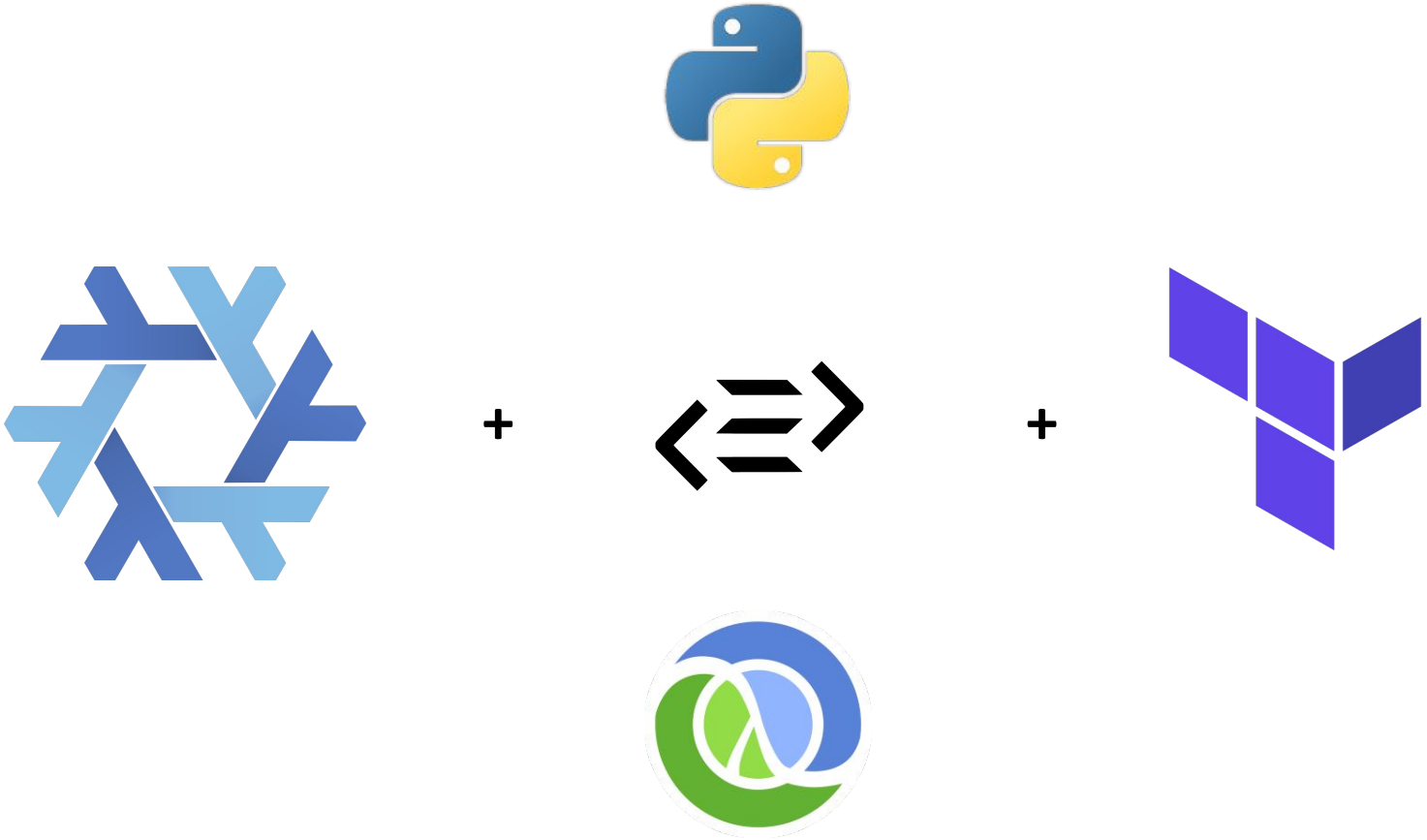


We know what we as a team liked about the TLS in Scala.

We know what we didn’t like.

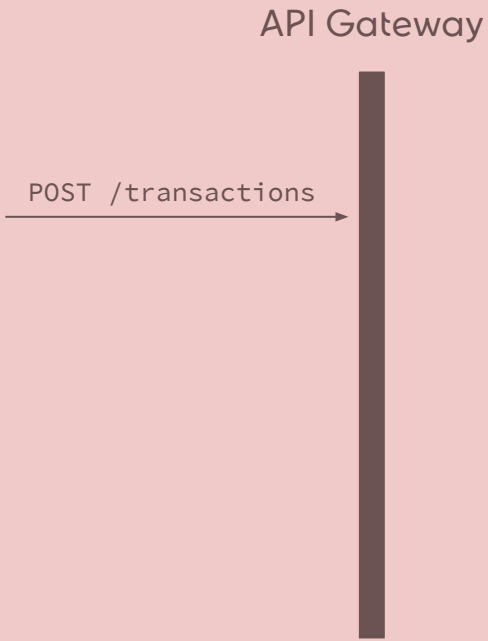
A pure FP language seems to be the next step in that evolution.

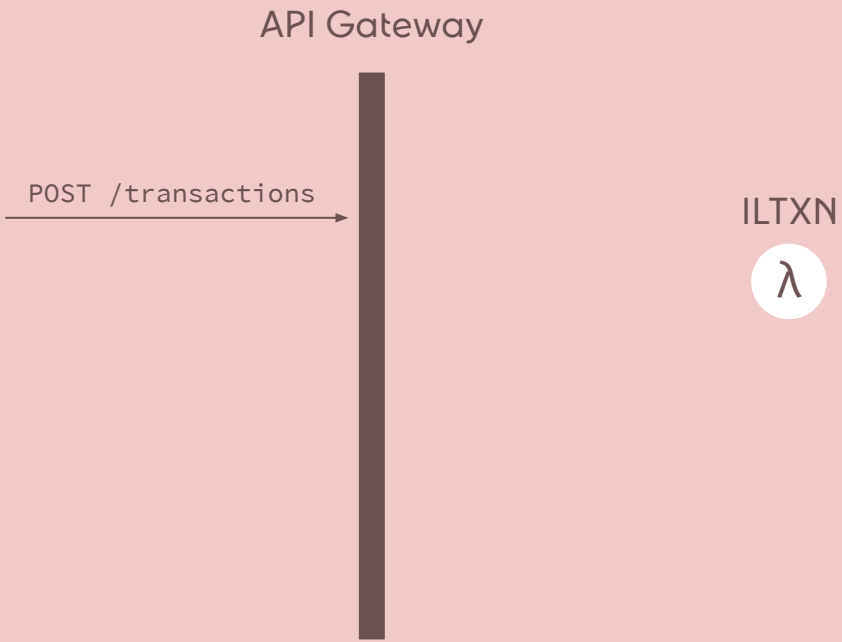
Purely Functional and Serverless

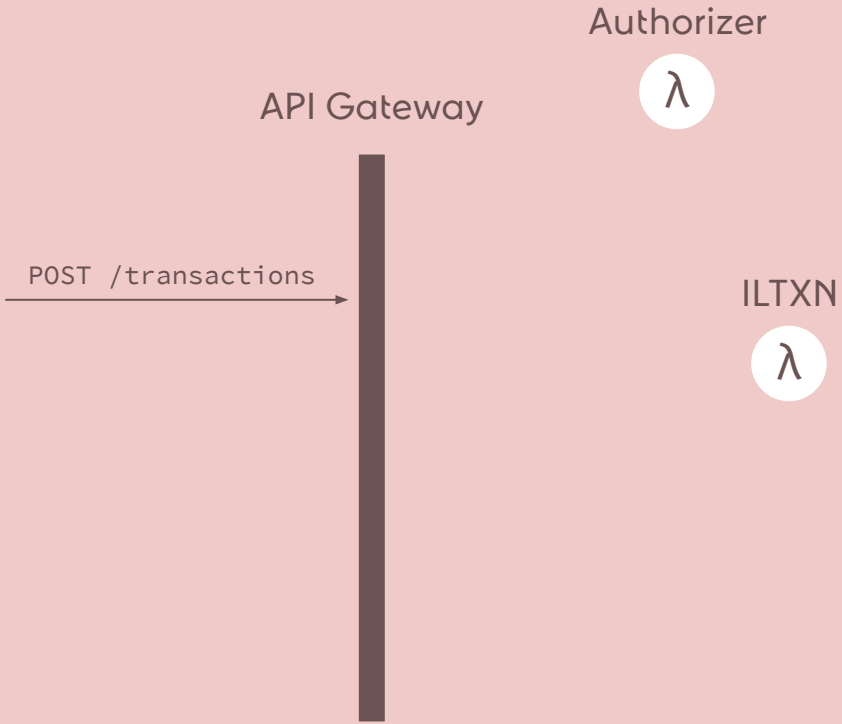


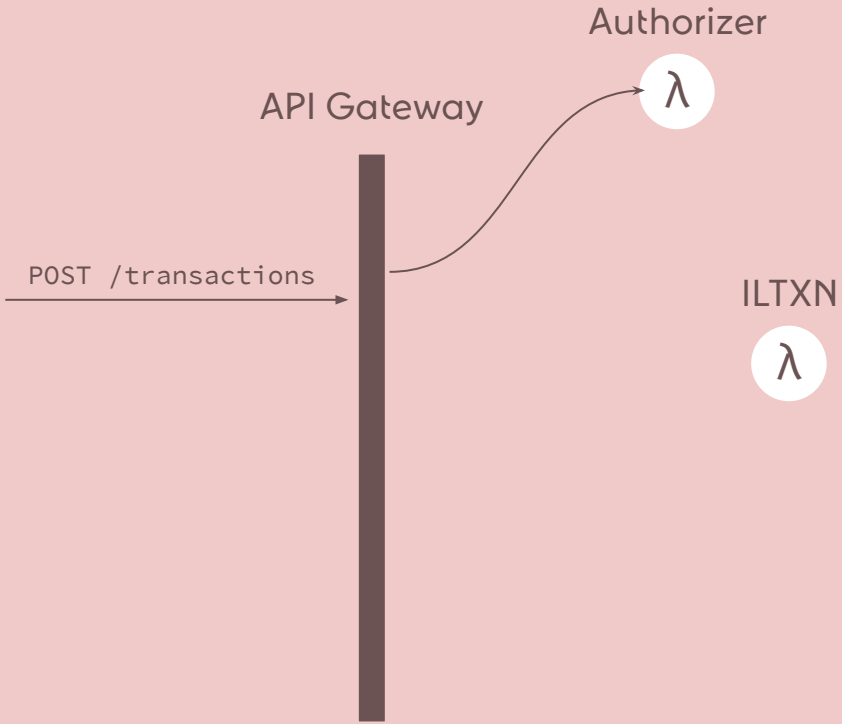
Serverless Architecture

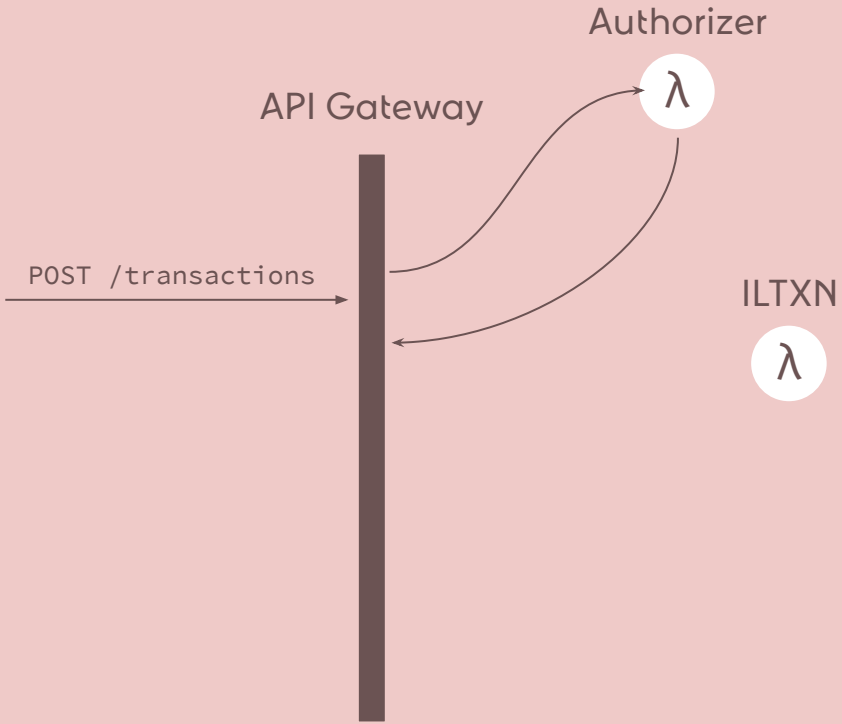
API Gateway

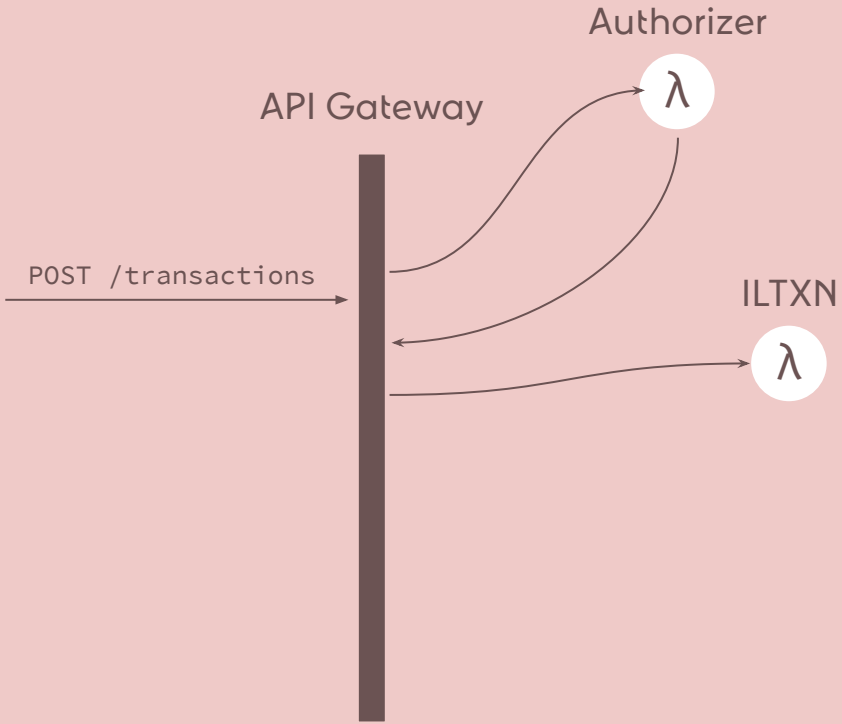

API Gateway
POST /transactions

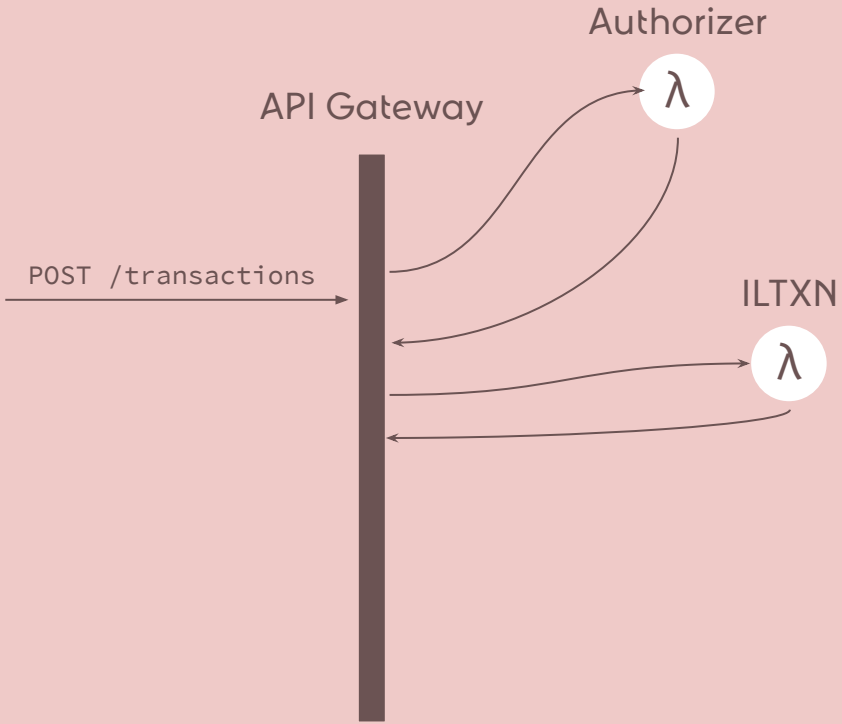

API Gateway
POST /transactions
ILTXN
λ

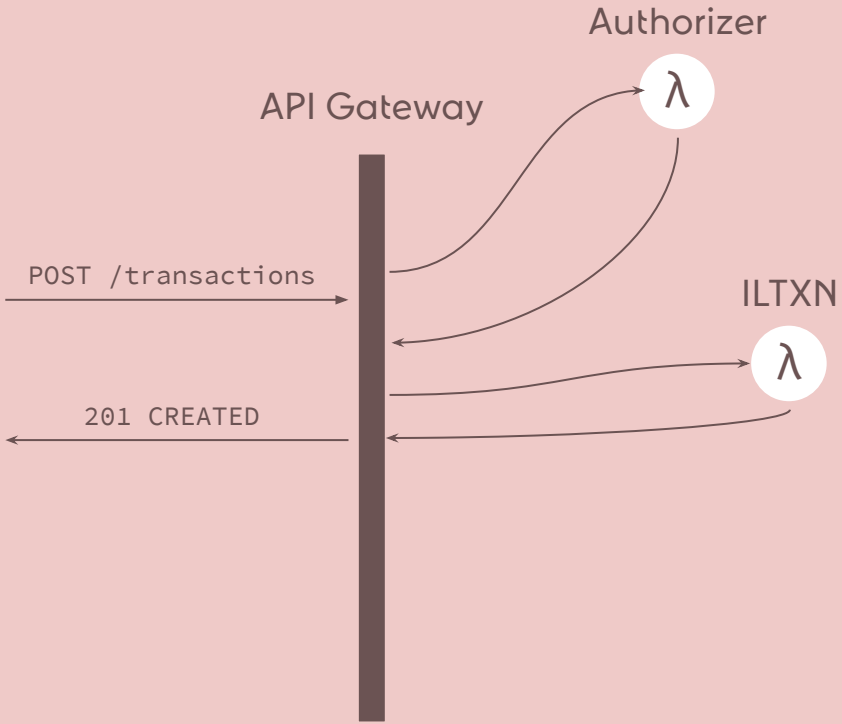

Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ

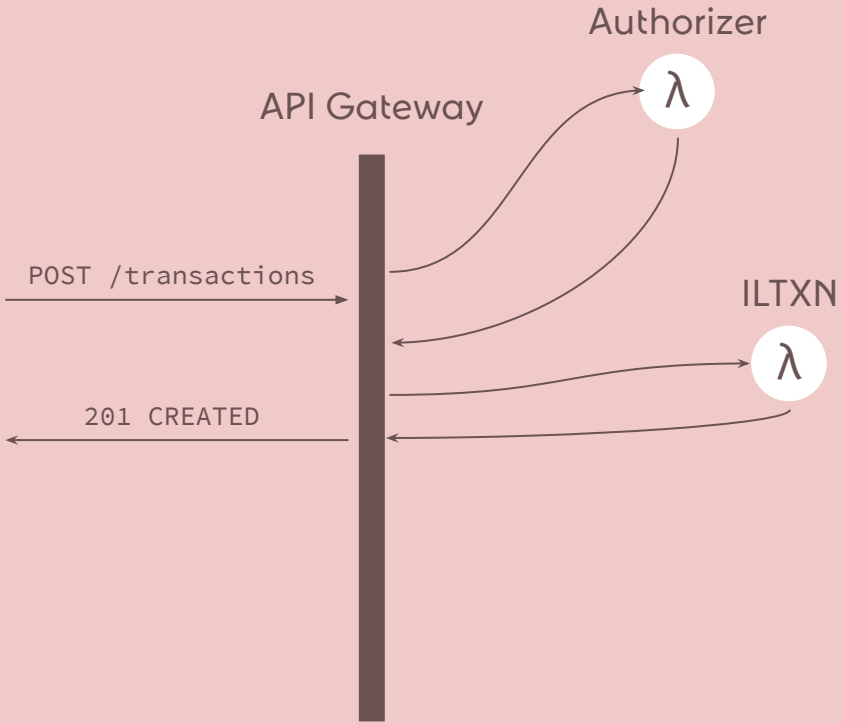

Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ


Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ

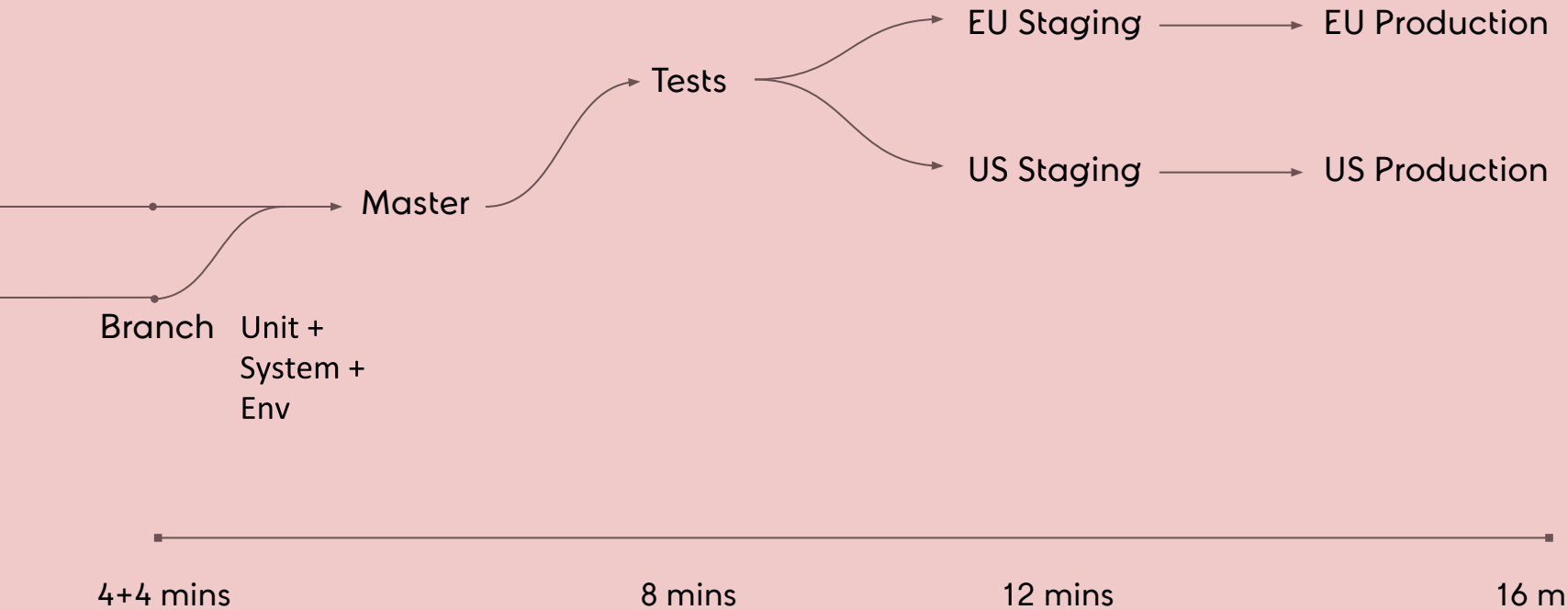

Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ


Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ


Authorizer
λ
API Gateway
POST /transactions
ILTXN
λ
201 CREATED



- Fast to deploy
- Immutable artefacts
- Simple interface
- API GW Proxy is the way to go


EU Staging
EU Production
Tests
US Staging
US Production
Master
Branch
Unit +
System +
Env
4+4 mins
8 mins
12 mins
16 mins

How to FP

Tagless Final?
MTL?
Free Monads?
Handler Pattern?

All diets work*

They all reduce caloric intake by limiting or removing one of the primary sources of energy, be it carbohydrates or fats.

Novelty Budget

- Handler Pattern
- Readers
- Rowtype Polymorphic Functions
- Tagless Final

Handler Pattern

```
module Kafka (send) where

newtype Persist a = Persist
  { persist :: a -> Aff Unit }

persistPayment :: Persist Payment
persistPayment = …

persistMock :: forall a. Persist a
persistMock = Persist { persist: const $ pure unit }
```

Readers

```
persistAndCallback ::
     forall r m
   . Has (Persist Payment) r
  => Has Callback r
  => MonadAff m
  => MonadRead r m
  => Payment
  -> m Unit
peristAndCallback = do
  getter asks >>= persist
  getter asks >>= callback
```

Rowtype Polymorphism

```
type Persistable a =
  (persist :: a -> Aff Unit)

doPersist :: Record (Persistable Payment) -> Payment -> Aff Unit
doPersist r = r.persist

doPersistAndCallback ::
     forall r
   . Record (Persistable Payment | Callbackable | r)
  -> Payment
  -> Aff Unit
doPersistAndCallback r p =
  r.persist p *> r.callback Success
```

Tagless Final

```
class Monad m => Persist m where
  persist :: Payment -> m Unit

class Monad m => Callback m where
  callback :: m Unit

persistAndCallback ::
  forall m. Persist m => Callback m => Payment -> m Unit
persistAndCallback =
  persist >>> const callback
```

Tagless Final

```
instance ( Has DynamoDB r
         , MonadAff m
         ) => Persist (ReaderT r m) where
  persist p = asks getter >>= persist' p

persist' :: MonadAff m => Payment -> DynamoDB -> m Unit
persist' = …


runReaderT (persistAndCallback payment)
  {dynamo: myDynamo, callback: myCallback}
```

So why did we switch to Haskell?

Novelty Budget

```purescript
-- Kafka.purs
module Kafka (send) where

foreign import _send ::
    forall k v
  . KafkaProducer -> k -> v -> Effect (Promise Unit)
```

```purescript
-- Kafka.purs
module Kafka (send) where

foreign import _send ::
    forall k v
  . KafkaProducer -> k -> v -> Effect (Promise Unit)
```

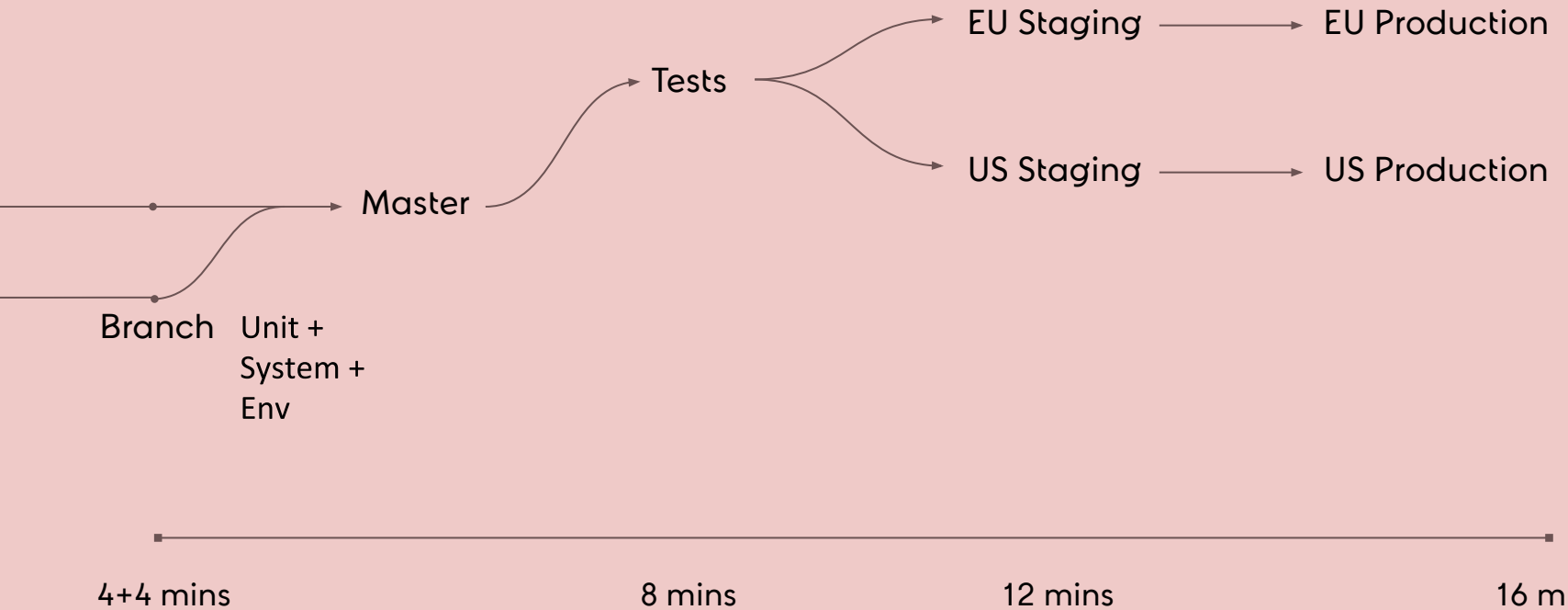
```javascript
// Kafka.js
exports._send = function (kafkaProducer) {
  return function (k) {
    return function (v) {
      return function () {
        return kafkaProducer.send(..., k, v);
      };
    };
  };
};
```

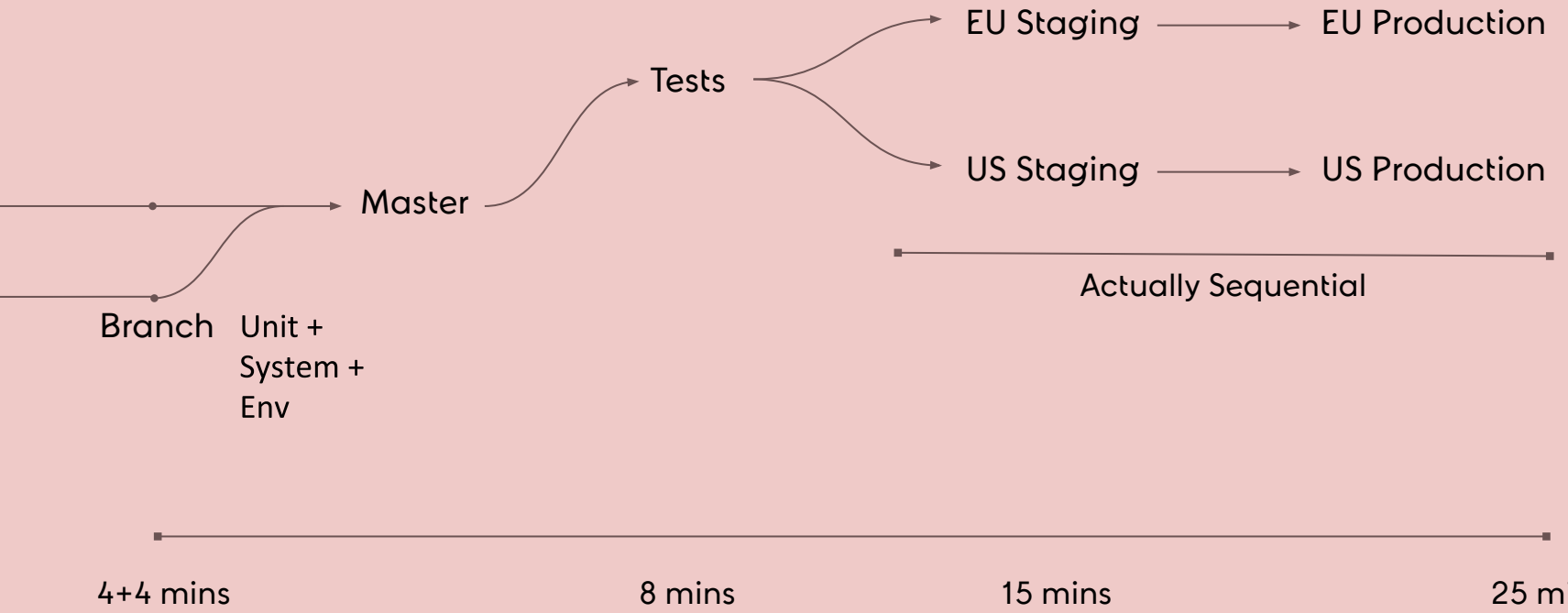
```purescript
-- Kafka.purs
module Kafka (send) where

foreign import _send ::
    forall k v
  . KafkaProducer -> k -> v -> Effect (Promise Unit)

send ::
    forall k v
  . KafkaProducer -> k -> v -> Aff Unit
send producer key value =
  toAffE $ _send producer key value
```

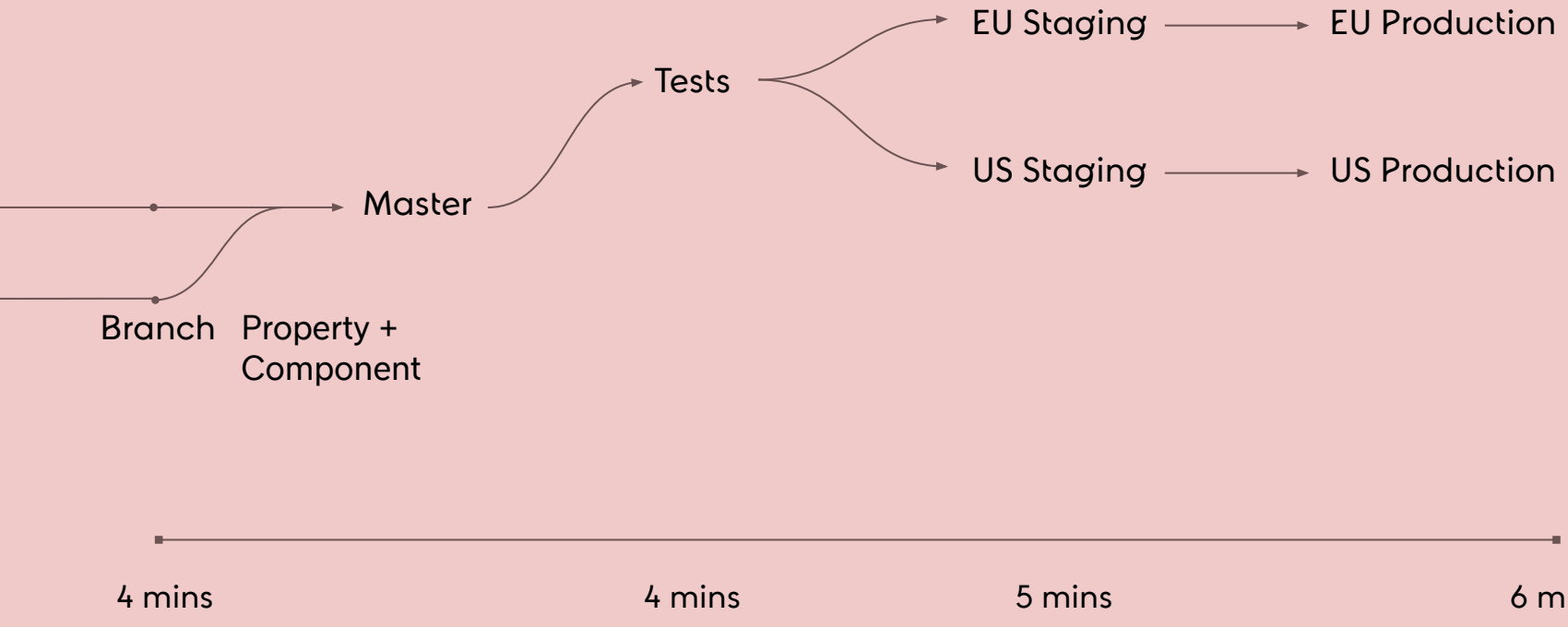
Haskell is just that much more mature

Serverless considered harmful*

*Not really.


EU Staging
EU Production
Tests
US Staging
US Production
Master
Branch
Unit +
System +
Env
4+4 mins
8 mins
12 mins
16 mins


EU Staging
EU Production
Tests
US Staging
US Production
Master
Actually Sequential
Branch
Unit +
System +
Env
4+4 mins
8 mins
15 mins
25 mins


EU Staging
EU Production
Tests
US Staging
US Production
Master
Branch
Property +
Component
4 mins
4 mins
5 mins
6 mins

```haskell
-- Handler.hs
module RequestHandler
  ( handleRequest_
  , handleRequest
  ) where

type EventHandler = KafkaConsumer -> IO ()

handleRequest_ :: KafkaProducer -> HttpClient -> EventHandler
handleRequest_ = …

handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest = …
```

```haskell
handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest = …

class MonadPersist m where
  persistTransaction :: Transaction -> m (Persisted Transaction)

class MonadRespond m where
  respond :: Response -> m Response

class MonadLog m where
  log :: LogLine -> m ()
```

```haskell
-- persistTransaction :: Transaction -> m (Persisted Transaction)
-- respond :: Response -> m Response
-- log :: LogLine -> m ()

handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest = undefined
```

```haskell
-- persistTransaction :: Transaction -> m (Persisted Transaction)
-- respond :: Response -> m Response
-- log :: LogLine -> m ()

handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest (Request Create transaction) =
  undefined
```

```haskell
-- persistTransaction :: Transaction -> m (Persisted Transaction)
-- respond :: Response -> m Response
-- log :: LogLine -> m ()

handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest (Request Create transaction) = do
  persistTransaction transaction >>=
  _andThen
```

```haskell
-- persistTransaction :: Transaction -> m (Persisted Transaction)
-- respond :: Response -> m Response
-- log :: LogLine -> m ()

handleRequest ::
     MonadPersist m
  => MonadRespond m
  => MonadLog m
  => Request -> m Response
handleRequest (Request Create transaction) = do
  persistTransaction transaction >>=
  respond . bifoldMap Error Created
```

```haskell
class MonadPersist m where
  persistTransaction :: Transaction -> m (Persisted Transaction)
```

```haskell
class MonadRespond m where
  respond :: Response -> m Response
```

```haskell
class MonadLog m where
  log :: LogLine -> m ()
```

```haskell
class MonadPersist m where
  persistTransaction :: Transaction -> m (Persisted Transaction)

instance ( Has DB.Connection r
         , MonadLog m
         , MonadIO m
         ) => MonadPersist (ReaderT r m) where
  persistTransaction transaction = do
    -- Read the DB connection from 'r'
    database <- asks getter

    -- Log some info using 'MonadLog'
    log $ "going to persist transaction: " <> show transaction

    -- Try to insert the transaction
    insertRes <- try @SqlError (DB.persist database transaction)

    -- Log the result
    log insertRes

    -- Wrap in return ADT
    pure . either PersistenceError Persisted $ insertRes
```

```haskell
newtype TestT r m a = TestT { runTestT :: ReaderT r m a }
  deriving (Functor, Applicative, Monad, MonadReader r)


type Test r = TestT r Identity


instance Monad m => MonadPersist (TestT r m) where
  persistTransaction = pure . Persisted
```

A Brave New World

Great & not so great things about Haskell.

Cons

- Records
- Extensions
- Cabal vs Stack vs Nix
- Gotchas when using Kafka

Pros

- Refactor without fear
- Very performant, no cold starts
- Great ecosystem
- Training is *shockingly* easier


Klarna.
What are you optimizing for?

Klarna.