# Stateful Functions @ BOSS

Stephan Ewen

# Disclaimer

*This is an early-stage open source project*

*Most designs and choices here were validated via user workloads, not via rigorous academic evaluations and benchmarks.*

*Not that we would not welcome academic evaluation and rigor. Collaborations welcome *wink*.*

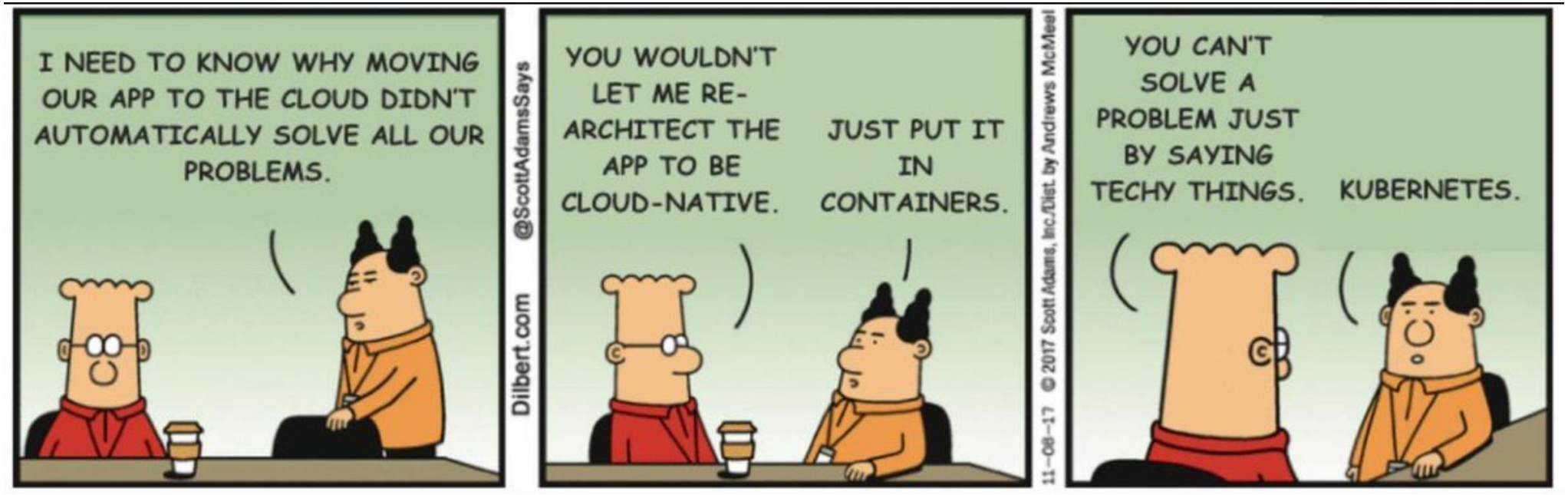



# Alternative Title

# An Excursion of a Stream Processor into the World of Applications and (Micro) Services



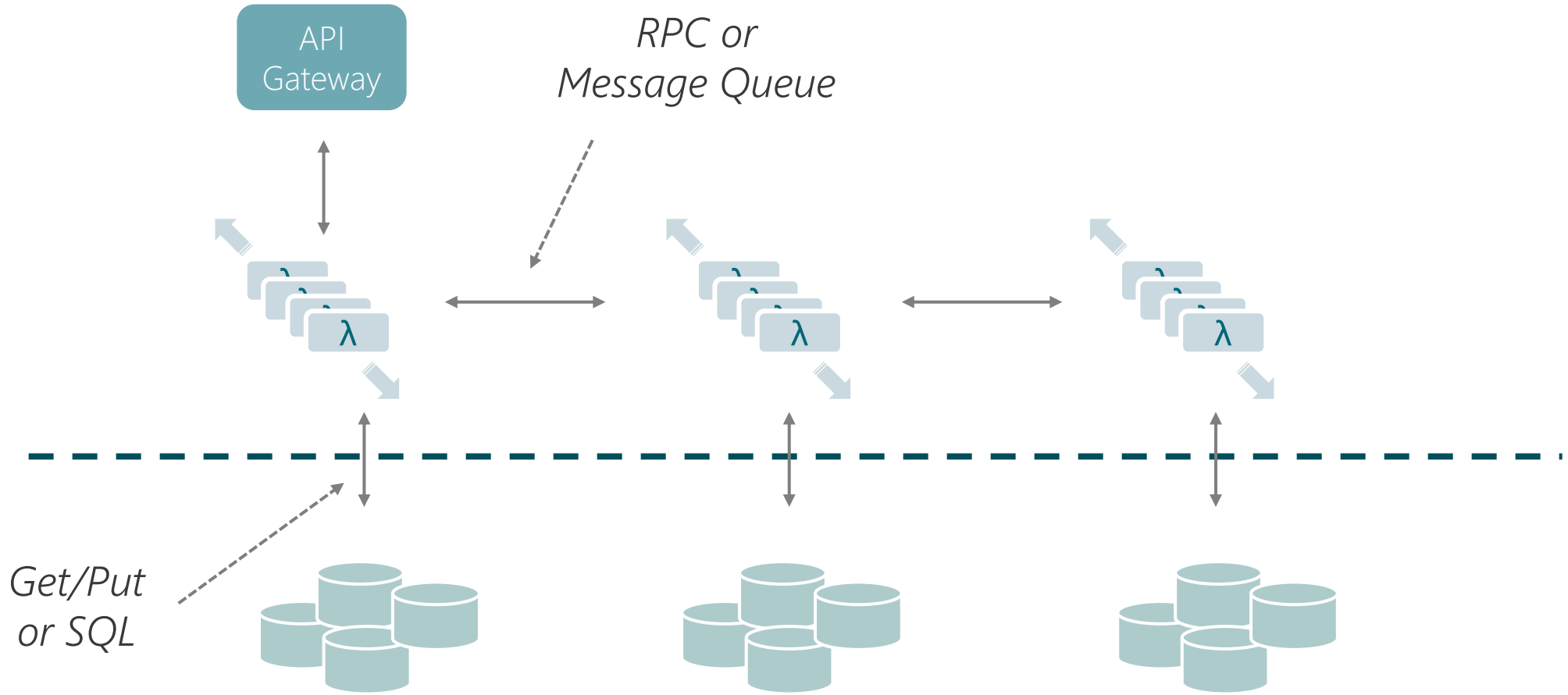

# Some (non scientific) Motivation



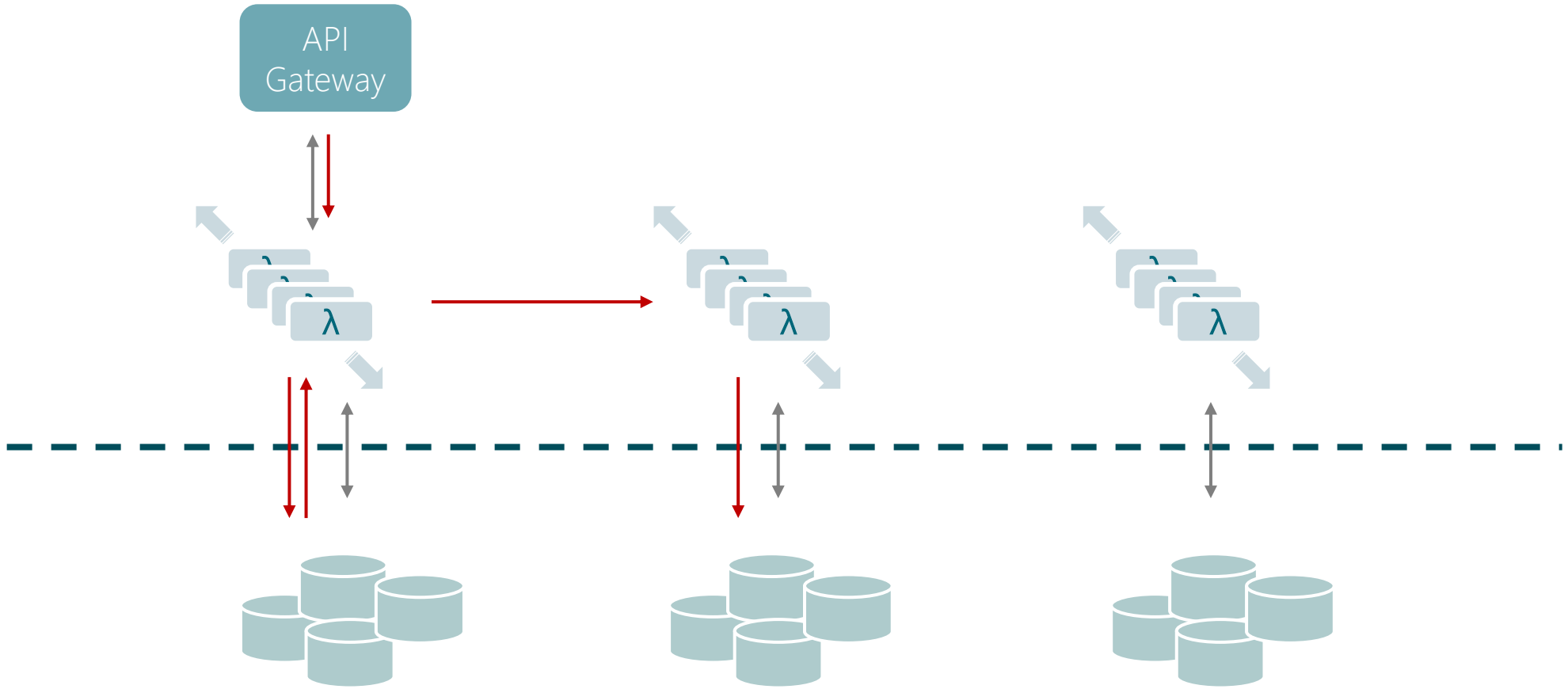
# Let’s look at some (micro) services interacting



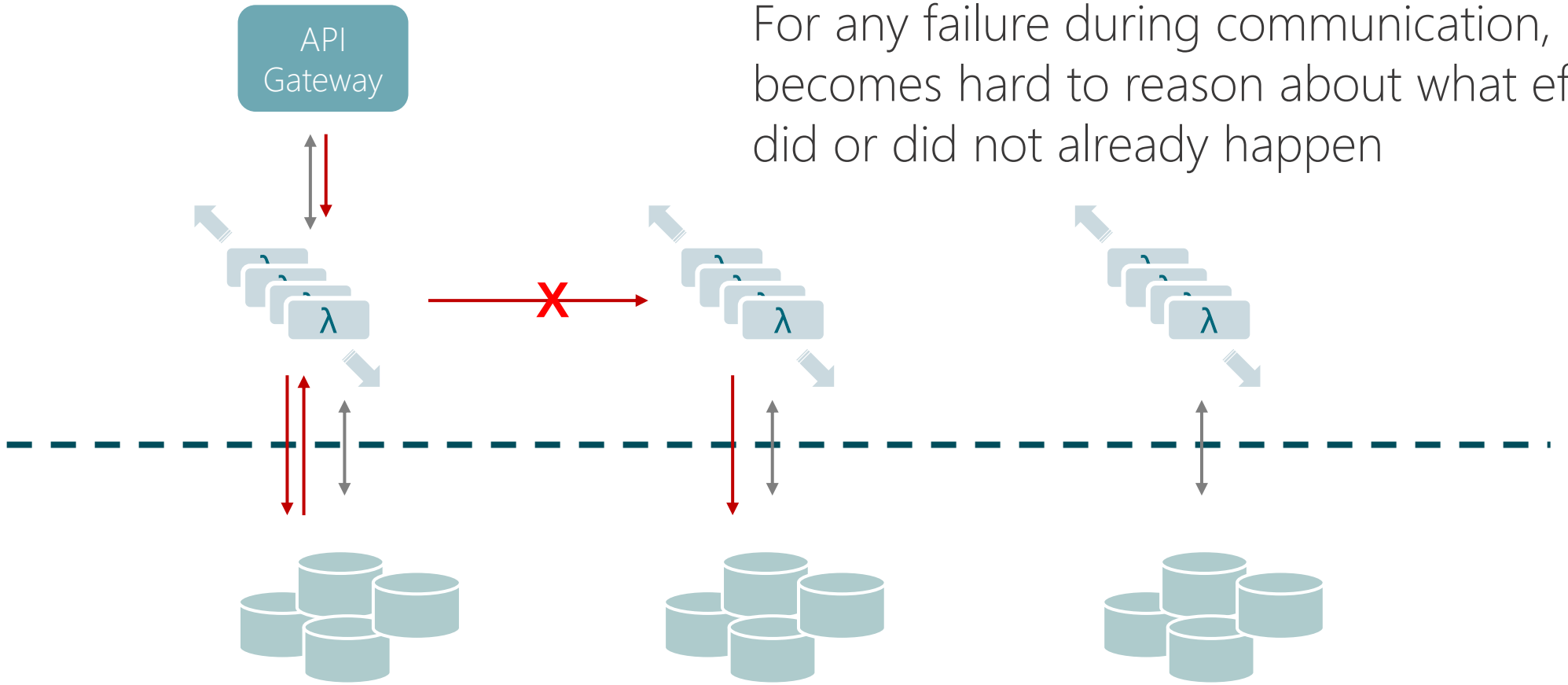

# Some interacting services



# Interaction is typically driven by the Application Layer



# Interaction is typically driven by the Application Layer



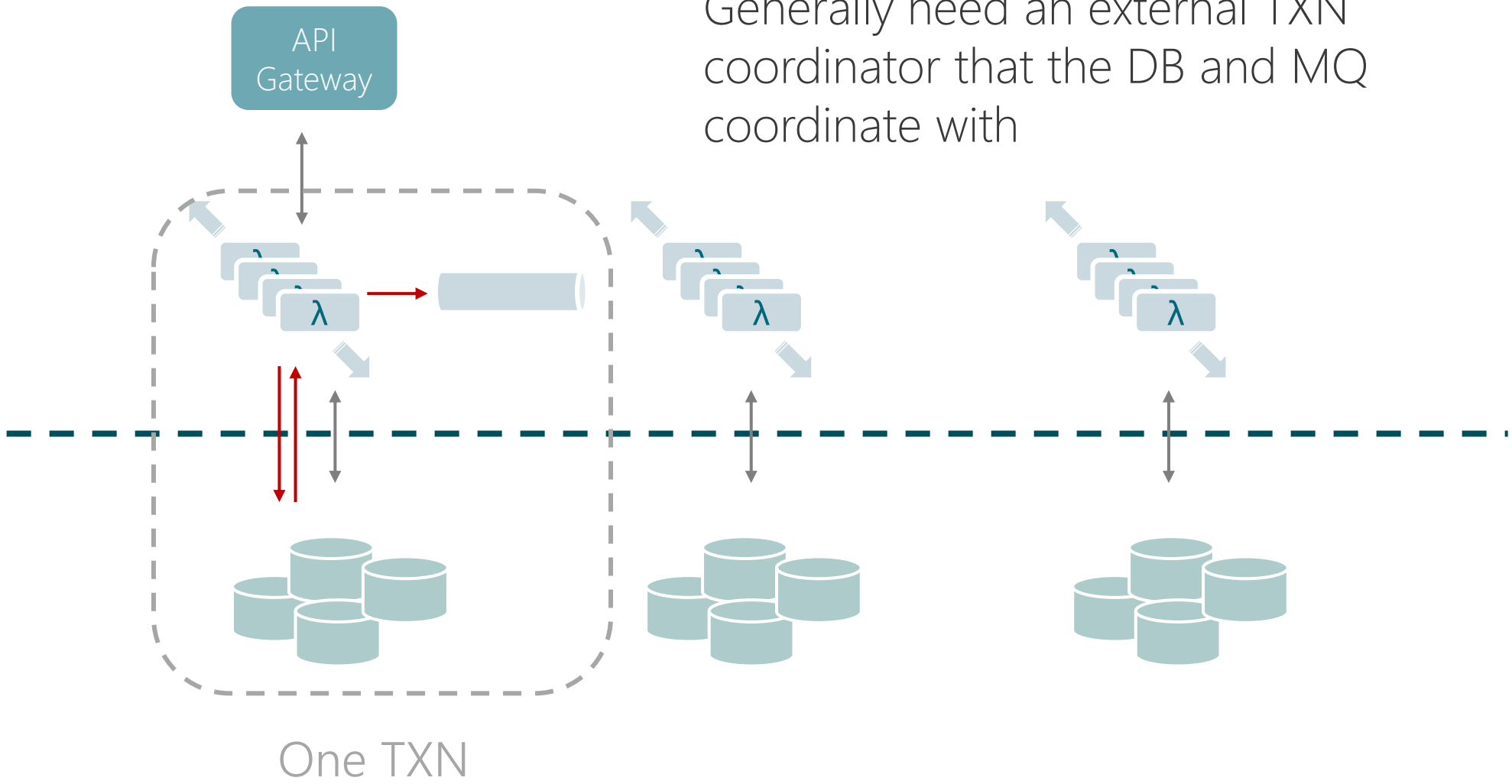

For any failure during communication, it becomes hard to reason about what effects did or did not already happen

# How to ensure consistency with failed communication

- "Two Generals Problem": Don't know if the message arrived or not
- But, we are not interested in whether the message was communicated between components exactly-once, but in whether some effects from the message were materialized consistently
- Lots of solutions for that
  - Distributed transactions
  - Do/undo workflows
  - Idempotent operations
  - …

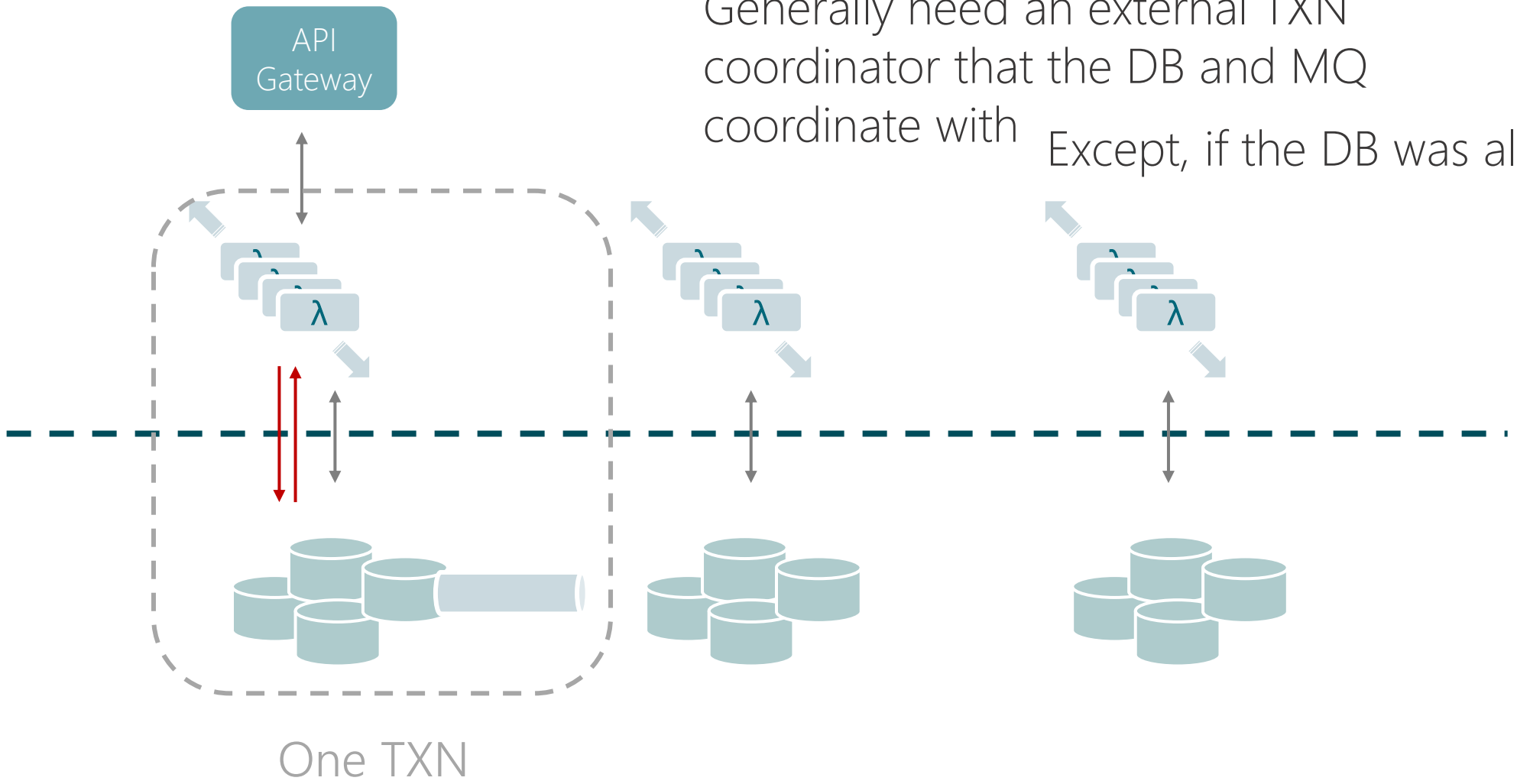



# Can we not run everything as a transaction?




Generally need an external TXN coordinator that the DB and MQ coordinate with

# Can we not run everything as a transaction?




Generally need an external TXN coordinator that the DB and MQ coordinate with

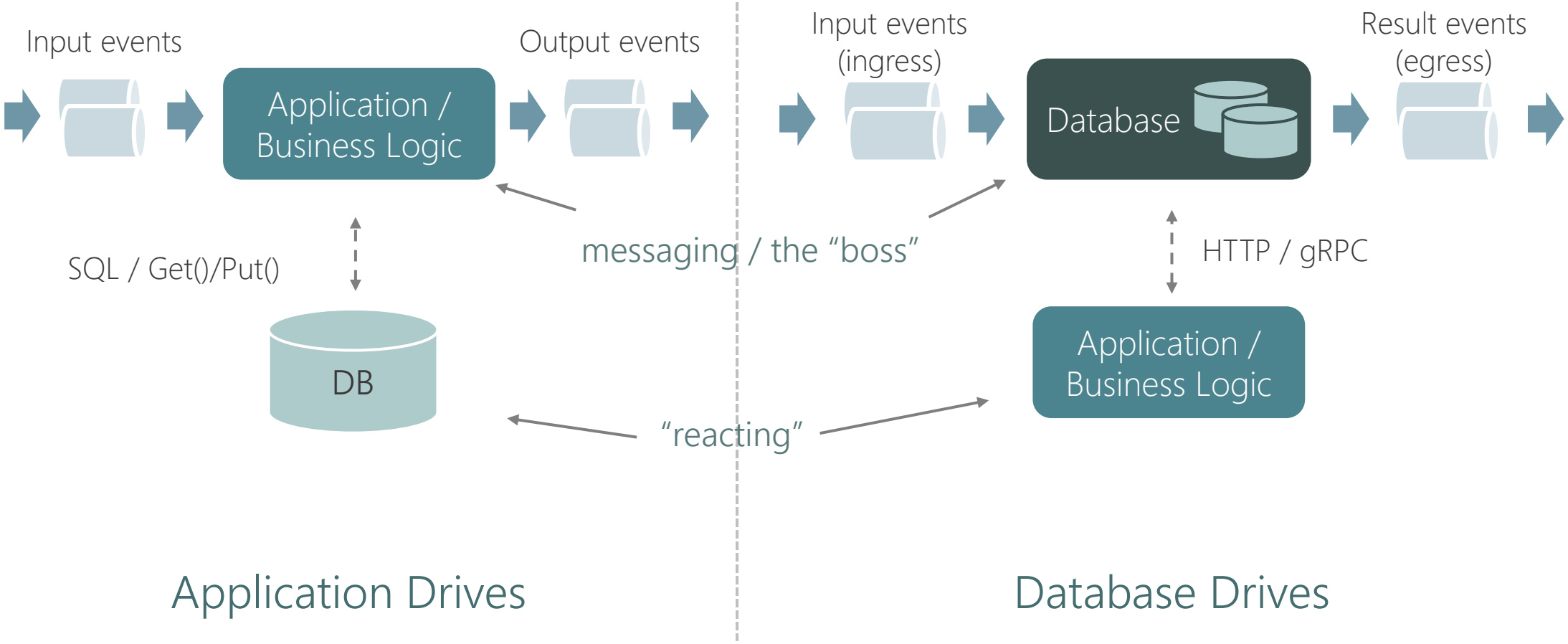
Except, if the DB was also a MQ

# Thought experiment:

# Let’s reverse the DB / App roles (acting / reacting)

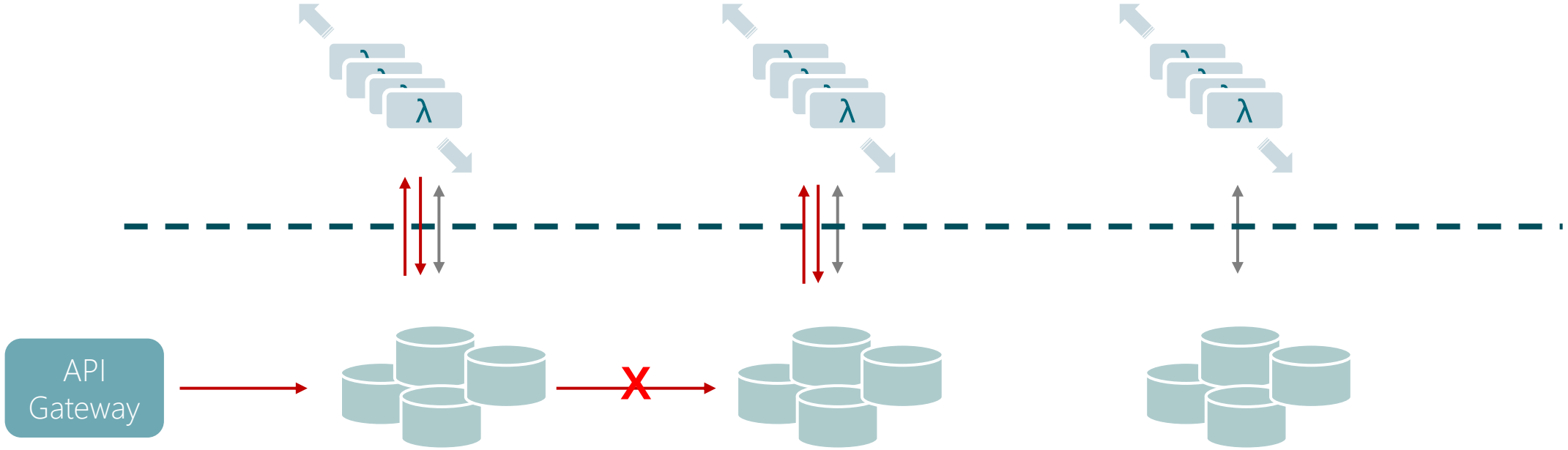



# Inverting the Roles of Application and Database



# The Same Layer handles State and Messaging

application logic are pure/stateless functions,
ergo idempotent by default




state updates and messaging are
easily made “atomic”

There are many solutions to solving integrated state and messaging.

In its core, it is a distributed transaction log problem.

From a more high-level perspective, it is what Stream Processors have worked on the last years.




# A Brief Excursion into Apache Flink

*Which takes the role similar to the Database here*

# Apache Flink: Analytics and Applications on Streaming Data

Streaming Analytics

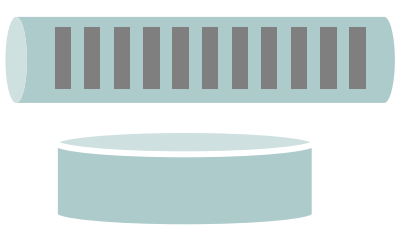
*SQL &*
*Dynamic Tables*

Stateful
Stream Processing

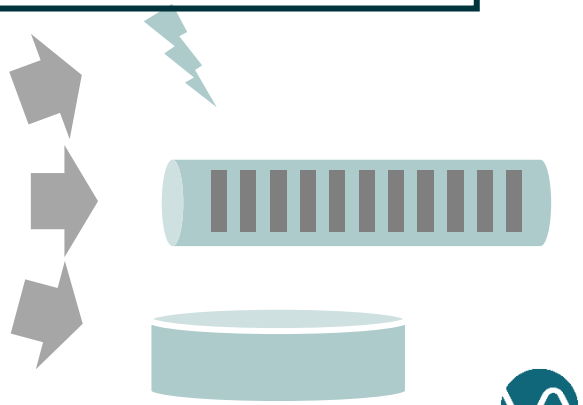
*Streams, State, Time*

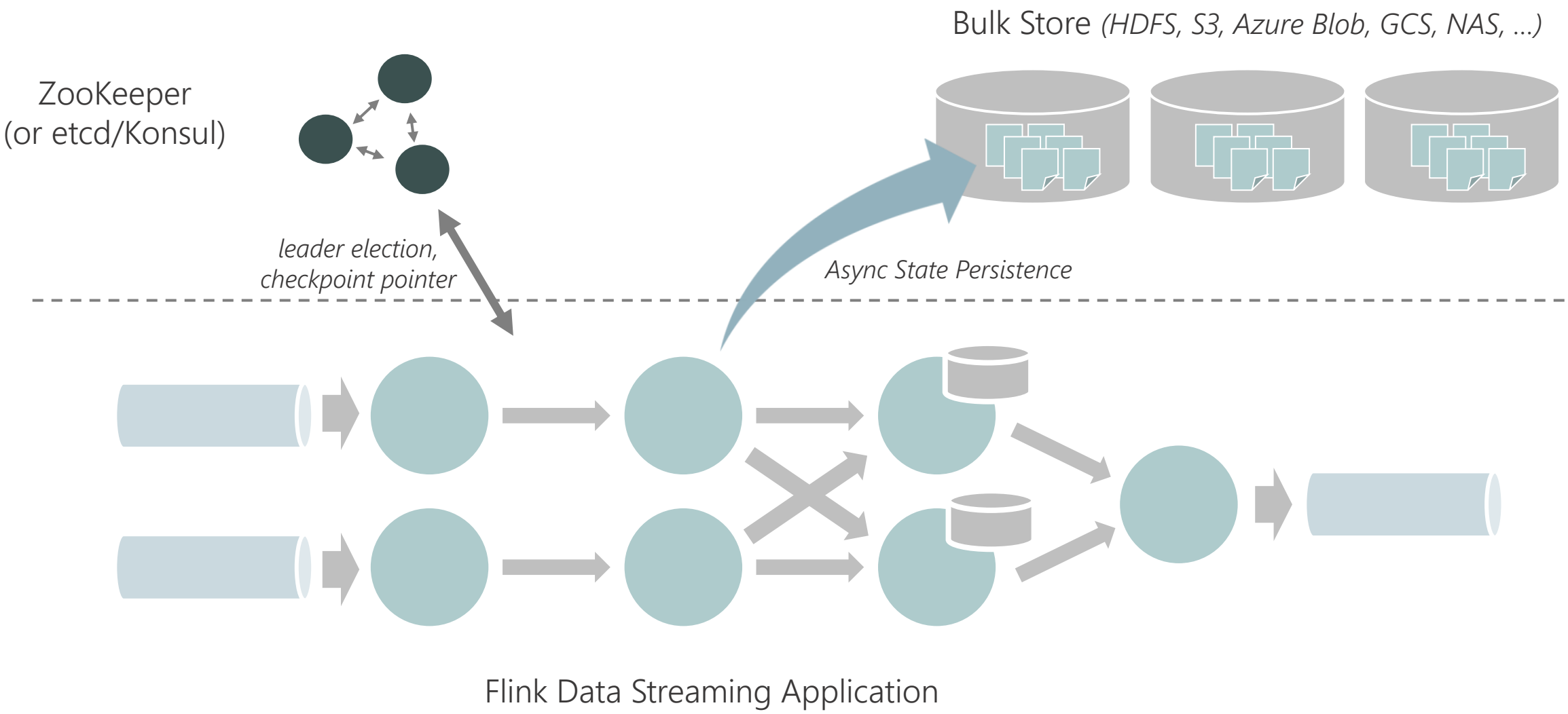
Event-driven
Applications

*Stateful Functions*



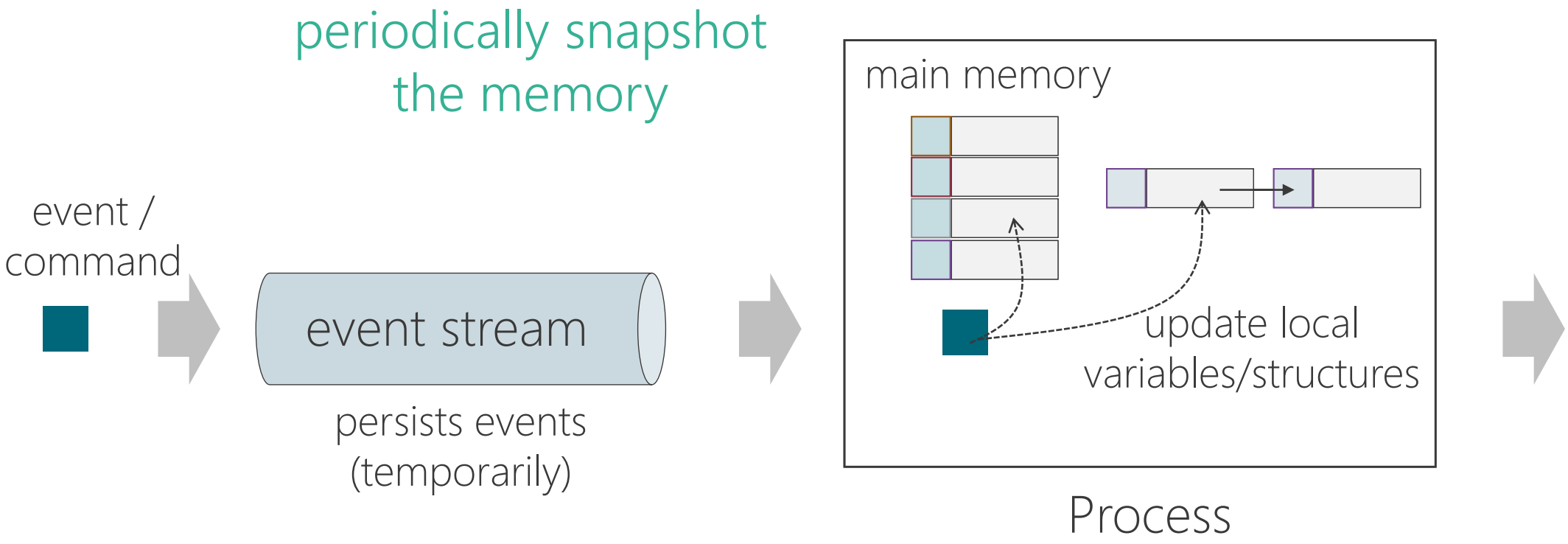
Flink Runtime
Stateful Computations over Data Streams





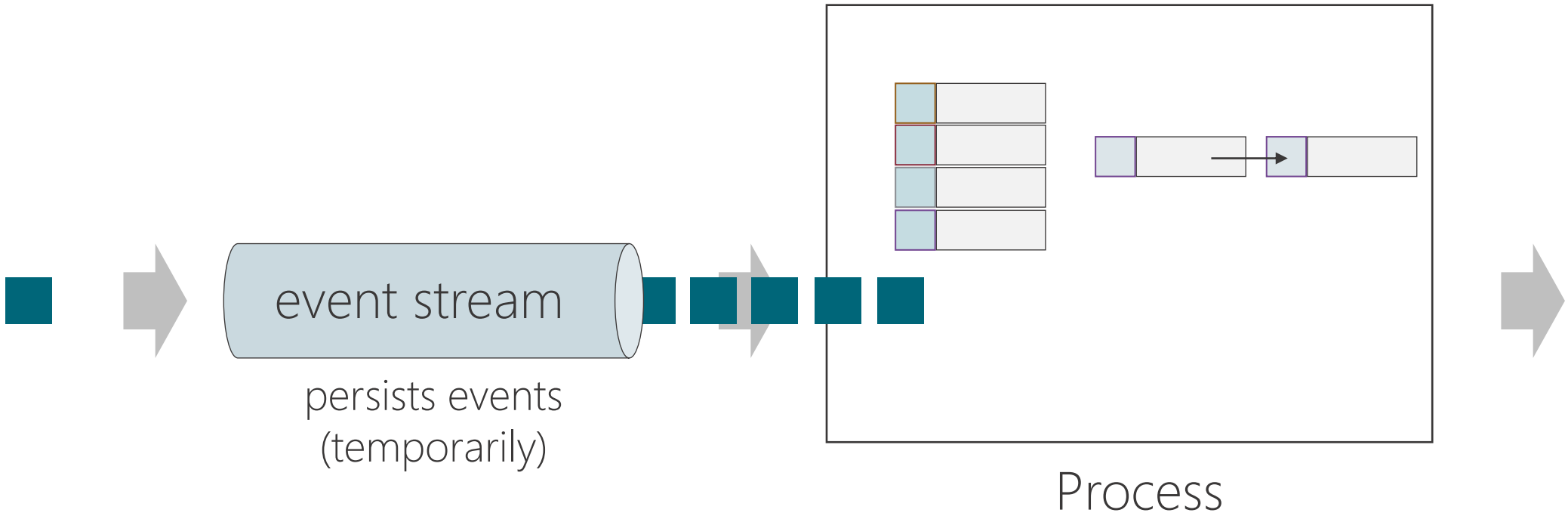

*Data keeps flowing directly between processes. Persistence is an "asynchronous background task".*

# Fault Tolerant State: Event Sourcing + Memory Image



# Fault Tolerant State: Event Sourcing + Memory Image

Recovery: Restore snapshot and replay events since snapshot



# Distributed Snapshots (Async Barrier Snapshots)

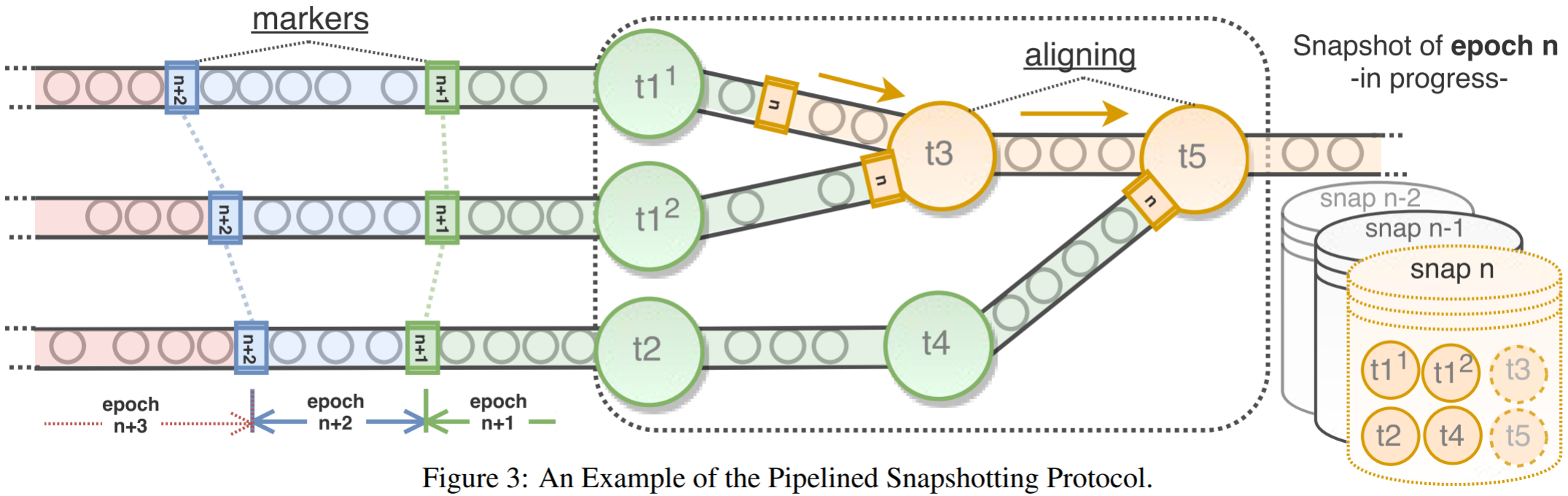



Figure 3: An Example of the Pipelined Snapshotting Protocol.

Paris Carbone et al.
"State Management in Apache Flink - Consistent Stateful Distributed Stream Processing."
PVLDB Vol. 10, No. 12, 2017

# Distributed Snapshots (Async Barrier Snapshots)

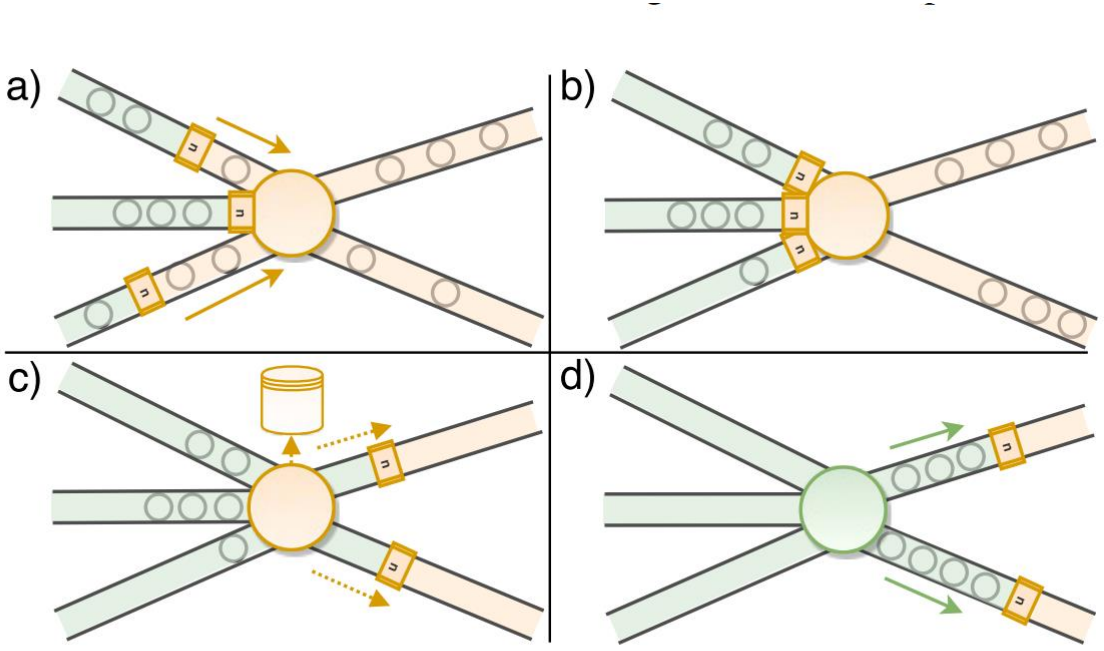



Figure 4: Alignment and Snapshotting Highlights.

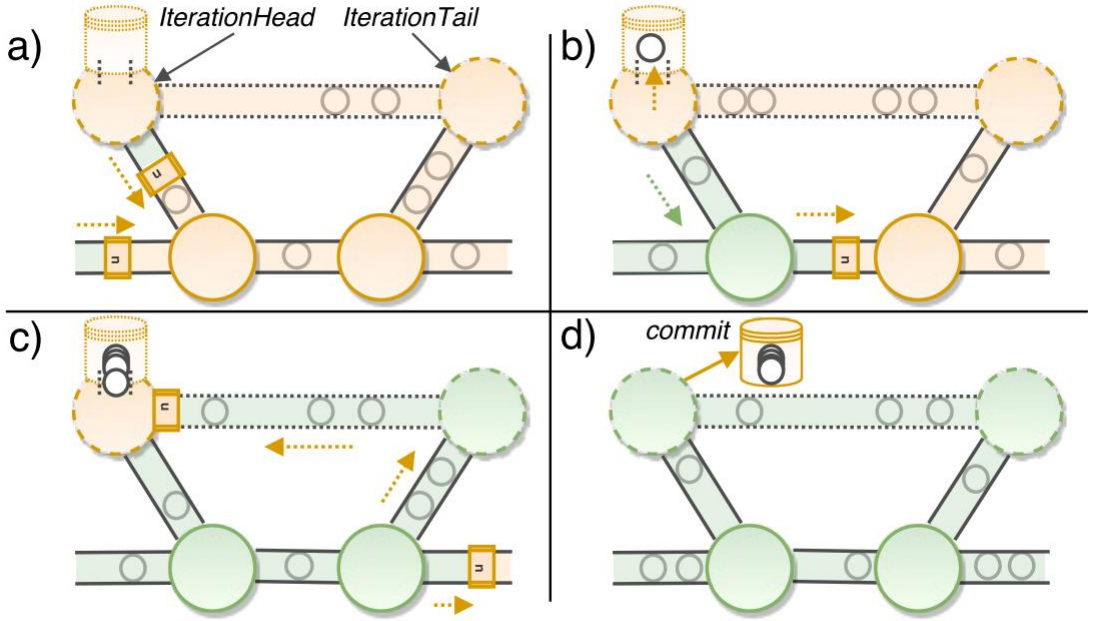



Figure 5: Cycle Snapshotting Highlights.

No in-flight data captured

Feedback in-flight data captured,
cf. Chandy-Lamport Algorithm

# How big can you go? - Alibaba: Double 11 / Singles Day

incl. sub-second updates to the GMV dashboard

Real-time Data Applications

Search | Rec. | Ads | BI | Security

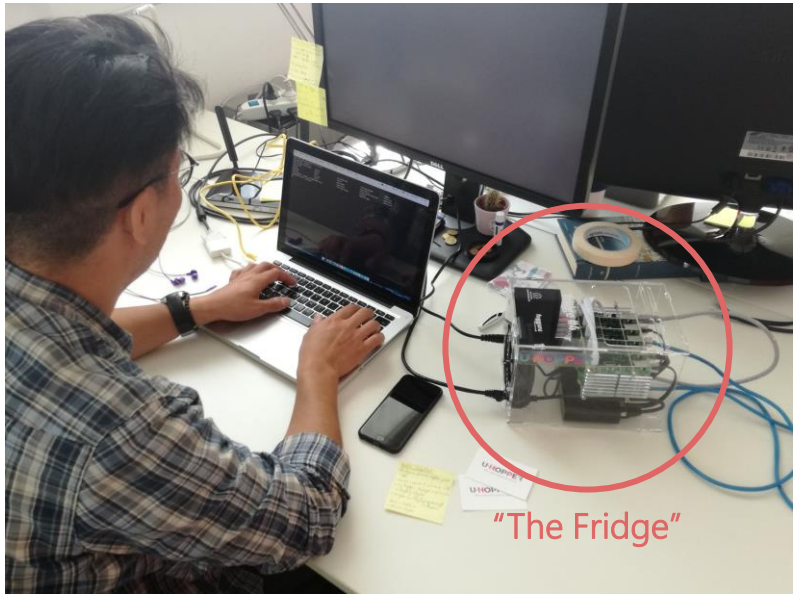
| Infrastructure | Data Size | Throughput (Peak) |
|---|---|---|
| >5K nodes<br>>500K CPU cores | 985PB | 2.5B events/sec |
| **State Size (Biggest)** | **Latency** | |
| 100TB | Sub-sec | |


Learn more: Optimizations in Blink Runtime for Global Shopping Festival at Alibaba

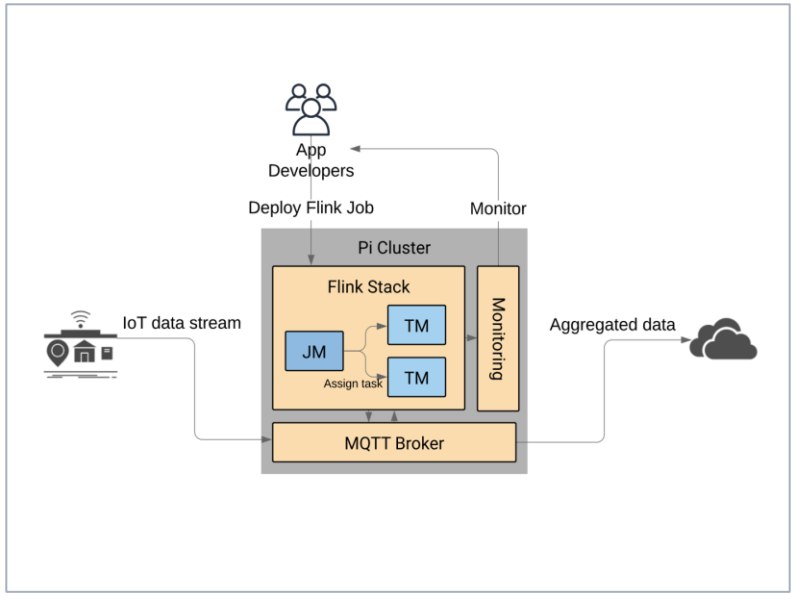
# How small can you go? - U-Hopper FogGuru

FogGuru is a platform for developing and deploying fog applications in resource-constrained devices.

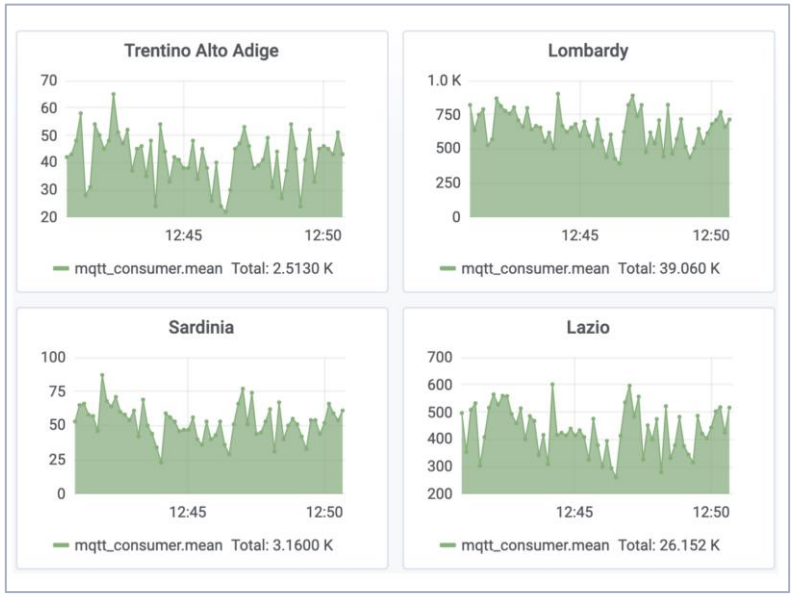
Cluster of 5 Raspberry Pi 3b+




Docker Swarm + Flink + Mosquitto




Data volume: 800 events/sec







Learn more: [FogGuru: a Fog Computing Platform Based on Apache Flink]

# Enter Stateful Functions

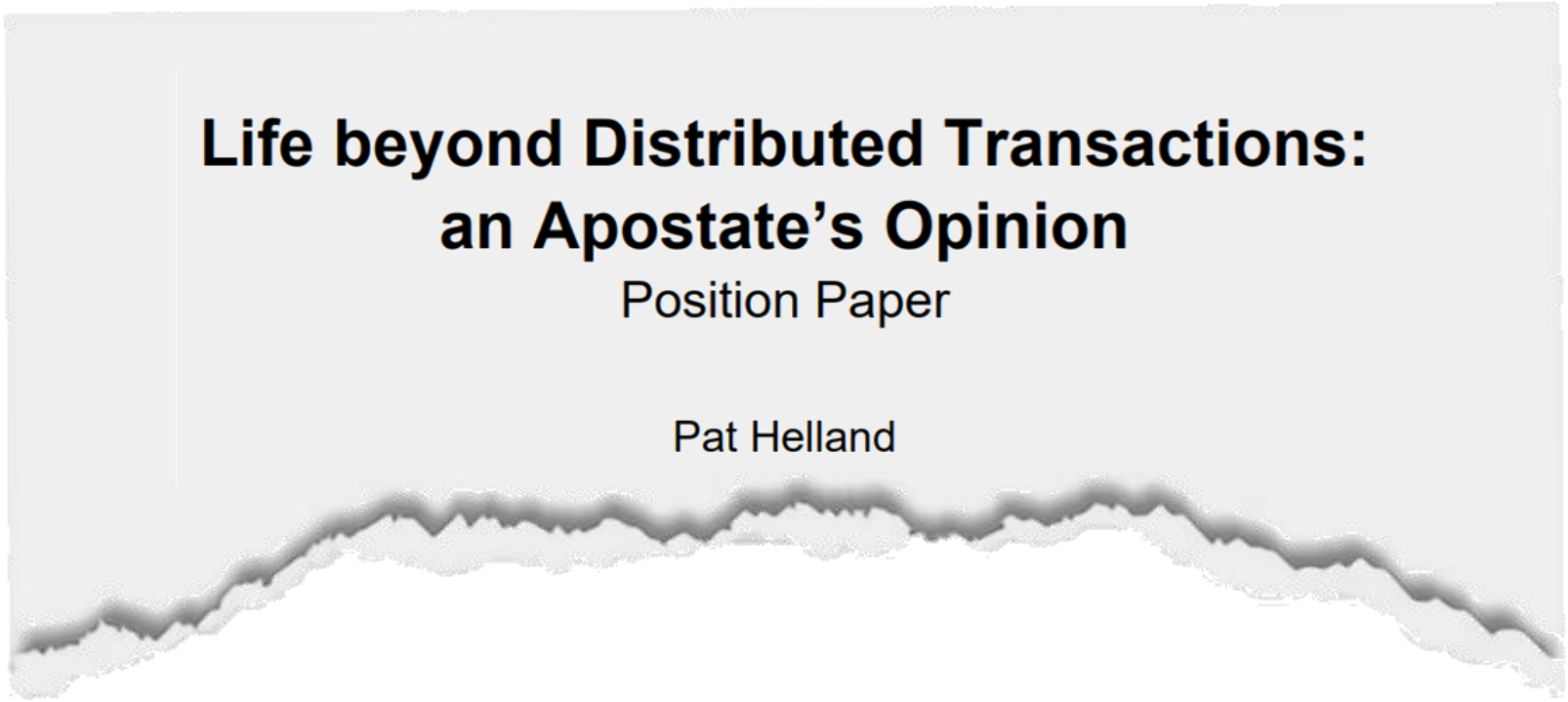
*Polyglot Event-Driven Functions for Distributed Stateful Applications*




# What is Stateful Functions?

An API that simplifies building distributed stateful applications …

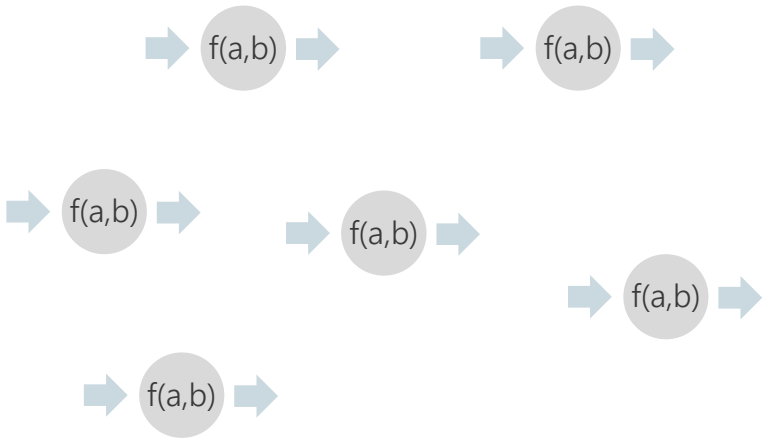



➔ Stateful Event-driven Entities / Actors

# Stateful Functions - API

An API that simplifies building distributed stateful applications …




## Building block: Functions

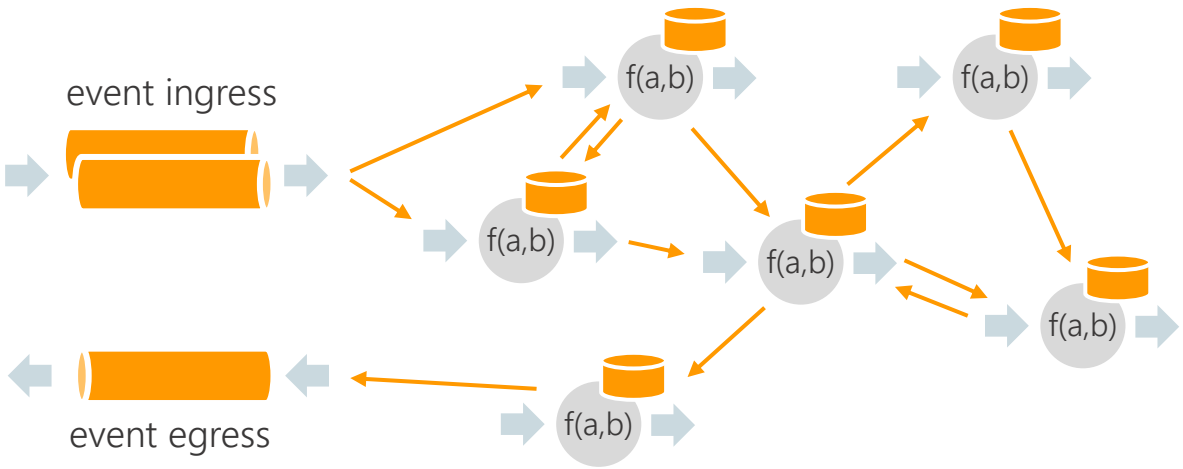
- Small piece of logic that represents entities
- Invokable through messages
- Inactive functions don't consume resources

## Multi-language Support

- Can be implemented in any programming language

# Stateful Functions - API

An API that simplifies building distributed stateful applications …




## Dynamic messaging

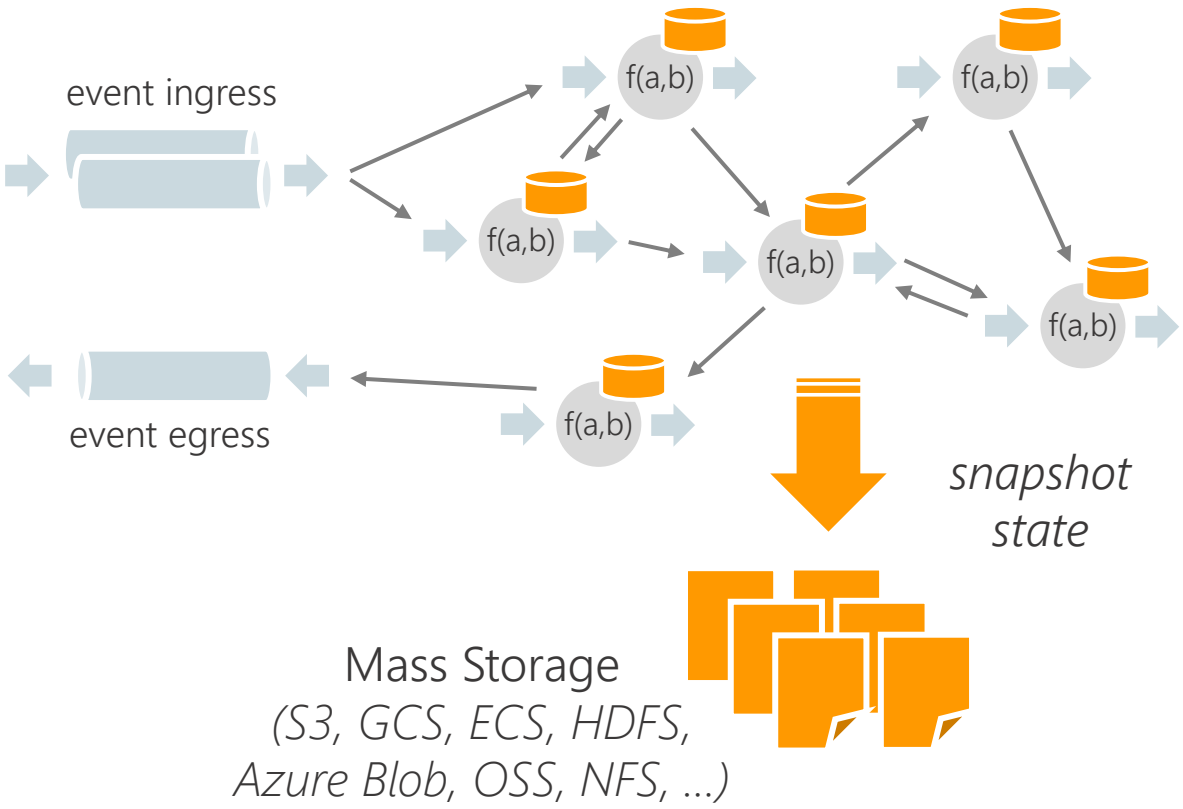
- Arbitrary communication between functions
- Functions message each other by logical addresses - no service discovery needed

## Consistent state

- Functions keep local state that is persistent and integrated with messaging
- Out-of-box exactly-once state access / updates & messaging

# Stateful Functions - API

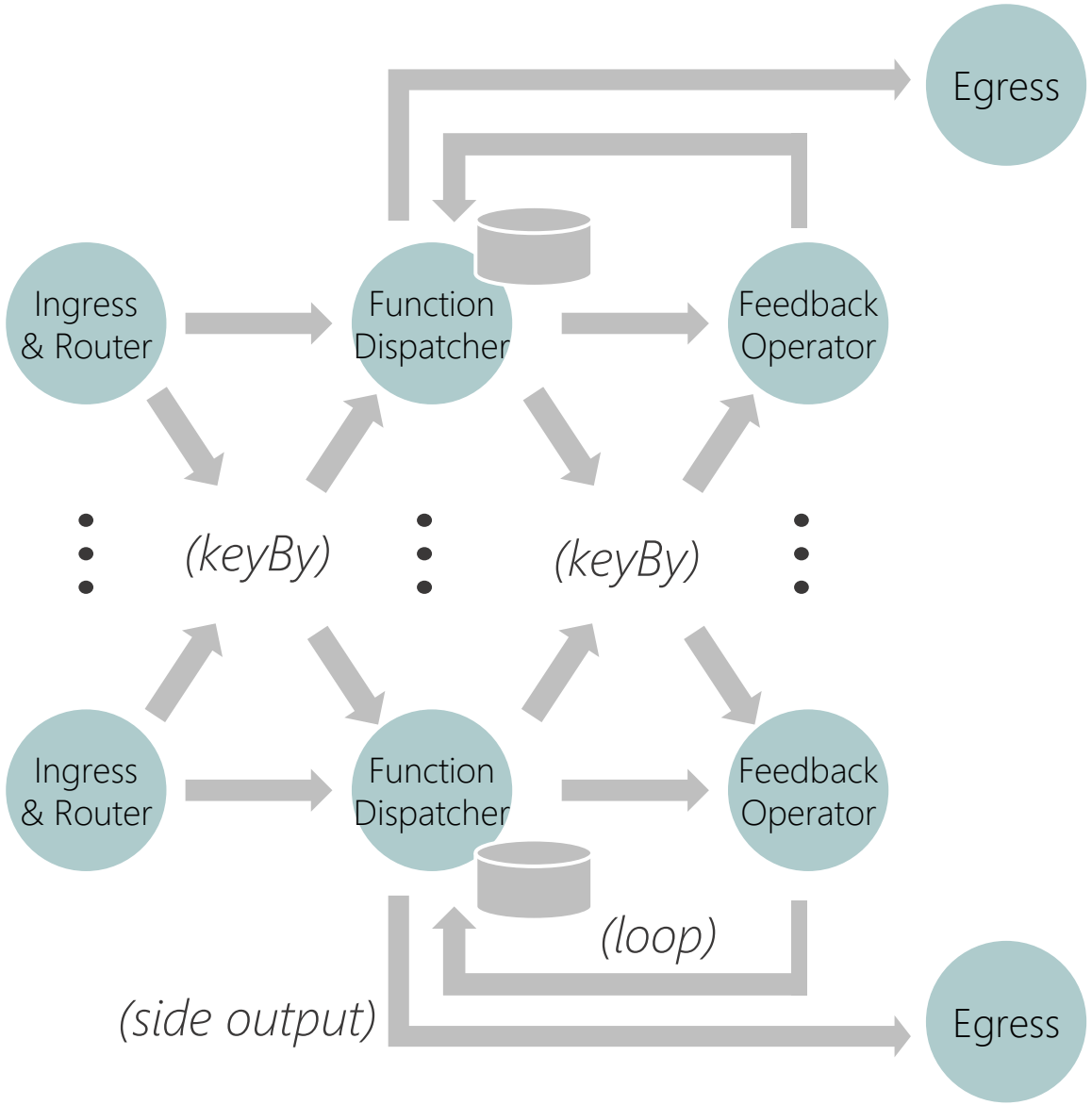
… with a runtime build for serverless architectures.



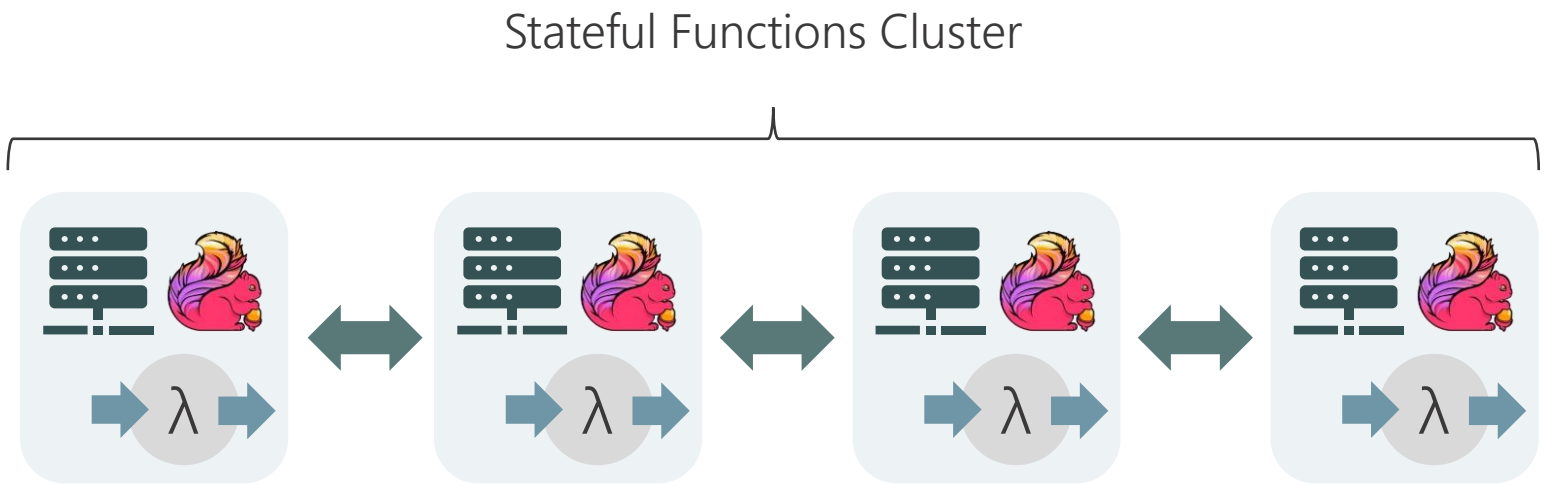

## Snapshots, no Database

- Uses Flink's distributed snapshots model for state durability and fault tolerance
- Requires only a simple blob storage tier to store state snapshots

# Running a Streaming Dataflow as an "Interpreter"



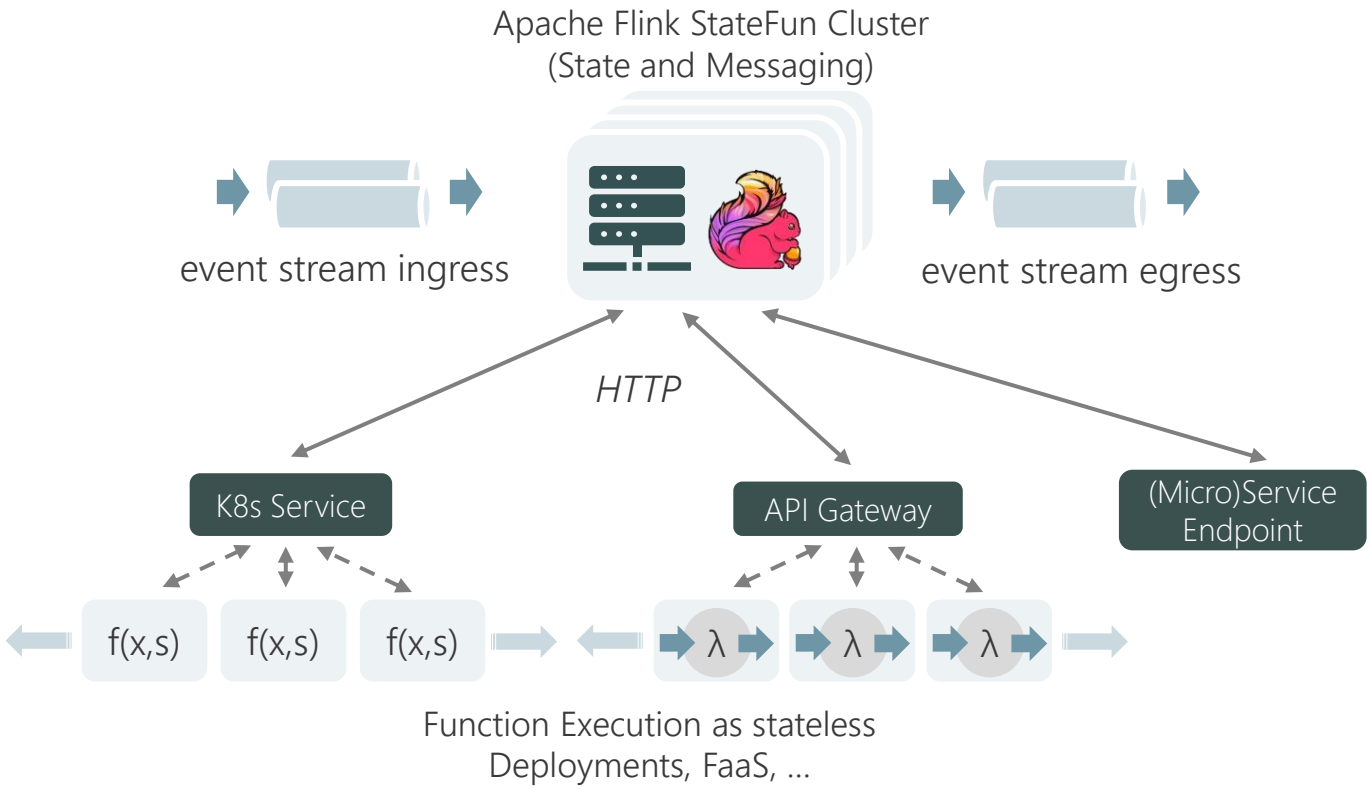
# Embedded Functions – Fast, but painfully stateful




*TaskManager &*
*StateFun Library*
*(JVM process / container)*

# Stateful Functions – Remote Functions

… with a runtime build for **serverless architectures.**



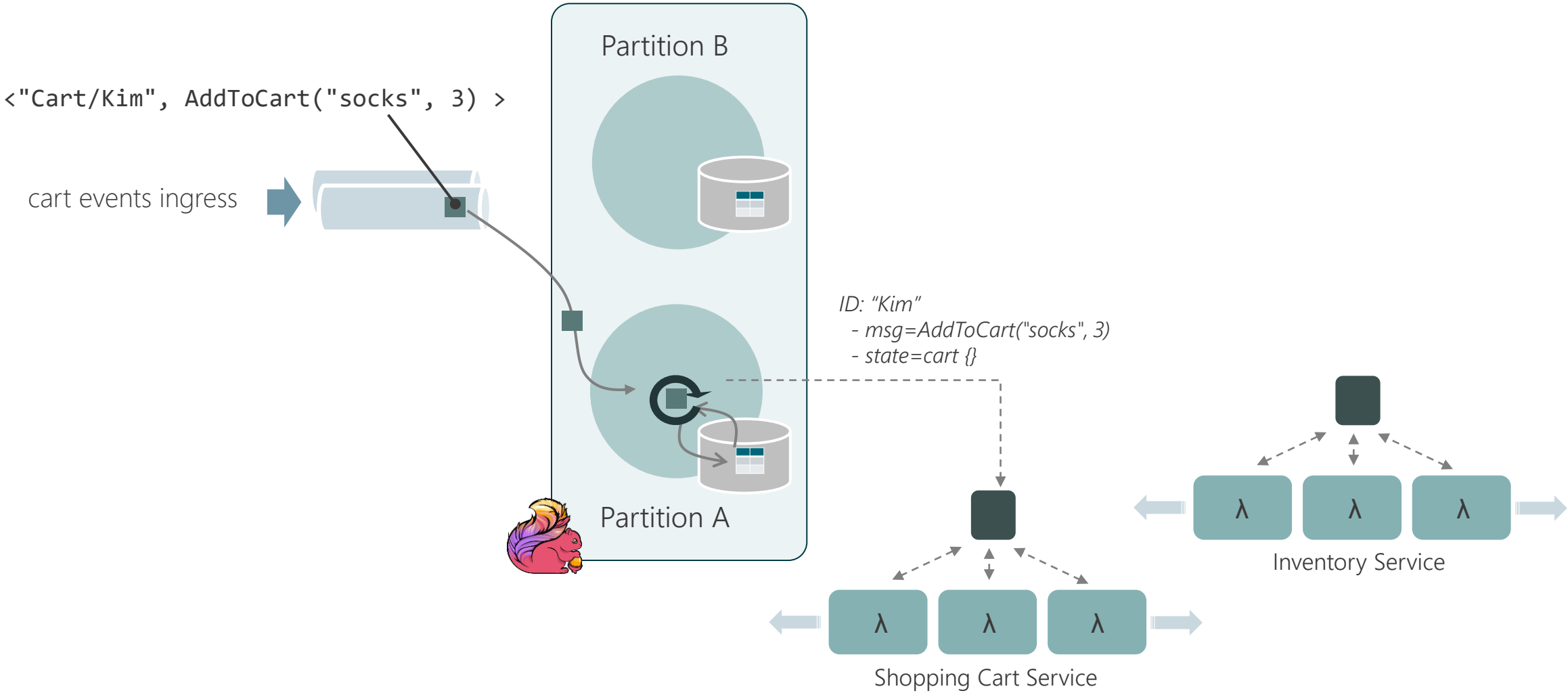

## Cloud Native

- Can be combined with capabilities of modern orchestration platforms (Kubernetes, FaaS platforms, …)

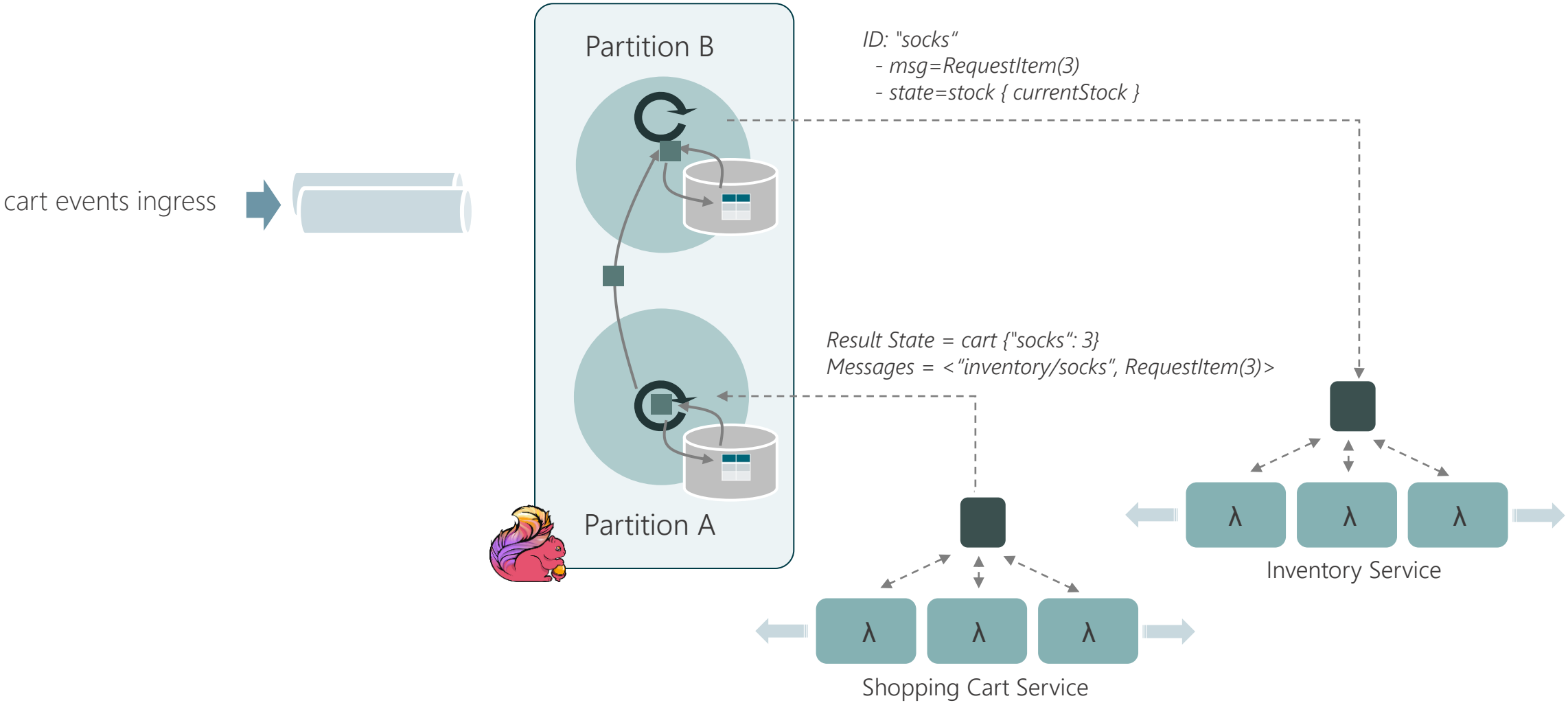
## "Stateless" Operation

- State access / updates is part of the invocations / responses
- Function deployments have benefits of stateless processes - rapid scalability, scale-to-zero, zero-downtime upgrades

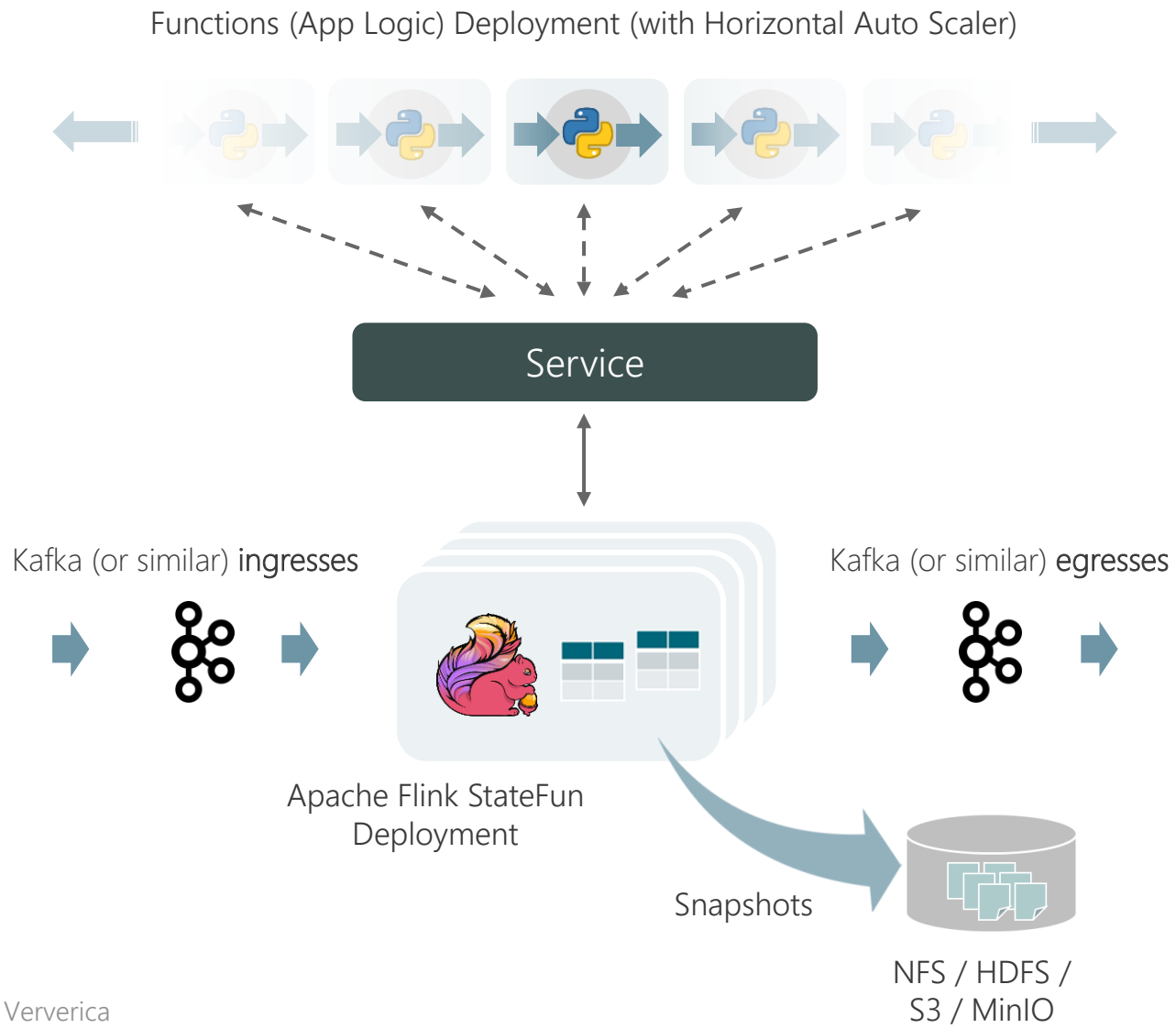
# State and Messages



# State and Messages

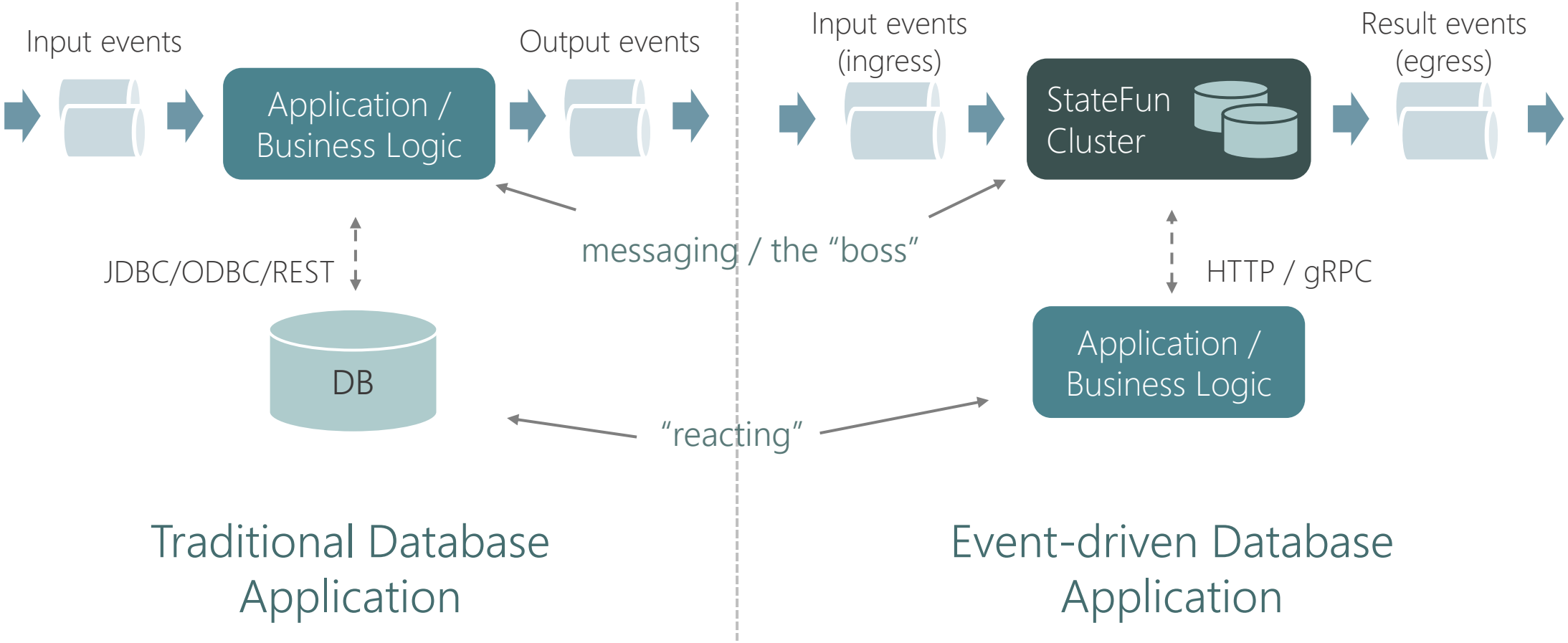


# Putting it all together: A Deployment on Kubernetes




- Deployment for Flink StateFun Cluster (stateful part)
- One or more deployments for the actual functions.
- Some Log or MQ for event ingress and egress.
- Some file system (or object store) for durability

# Separating Compute and State, but putting Messaging differently



# Wrapping it up




# Takeaways

- There is a lot of similarity between Stream Processing and Event-driven Applications. A loopy Stream Processor can be an interpreter for distributed event-driven applications.
- Having state and messaging tightly integrated makes a lot of things easy when building distributed applications.
- Integrating Compute with State is both the blessing and the curse of Stream Processing
- When separating Compute from State, let Messaging go where the State goes.
- Curious if we will see is a separate class of „event-driven databases“ in the future.




# Thank you for listening!

If you are interested in collaborating, please reach out to us.

@StephanEwen – sewen@apache.org

@ApacheFlink

@StateFun_IO

https://flink.apache.org/



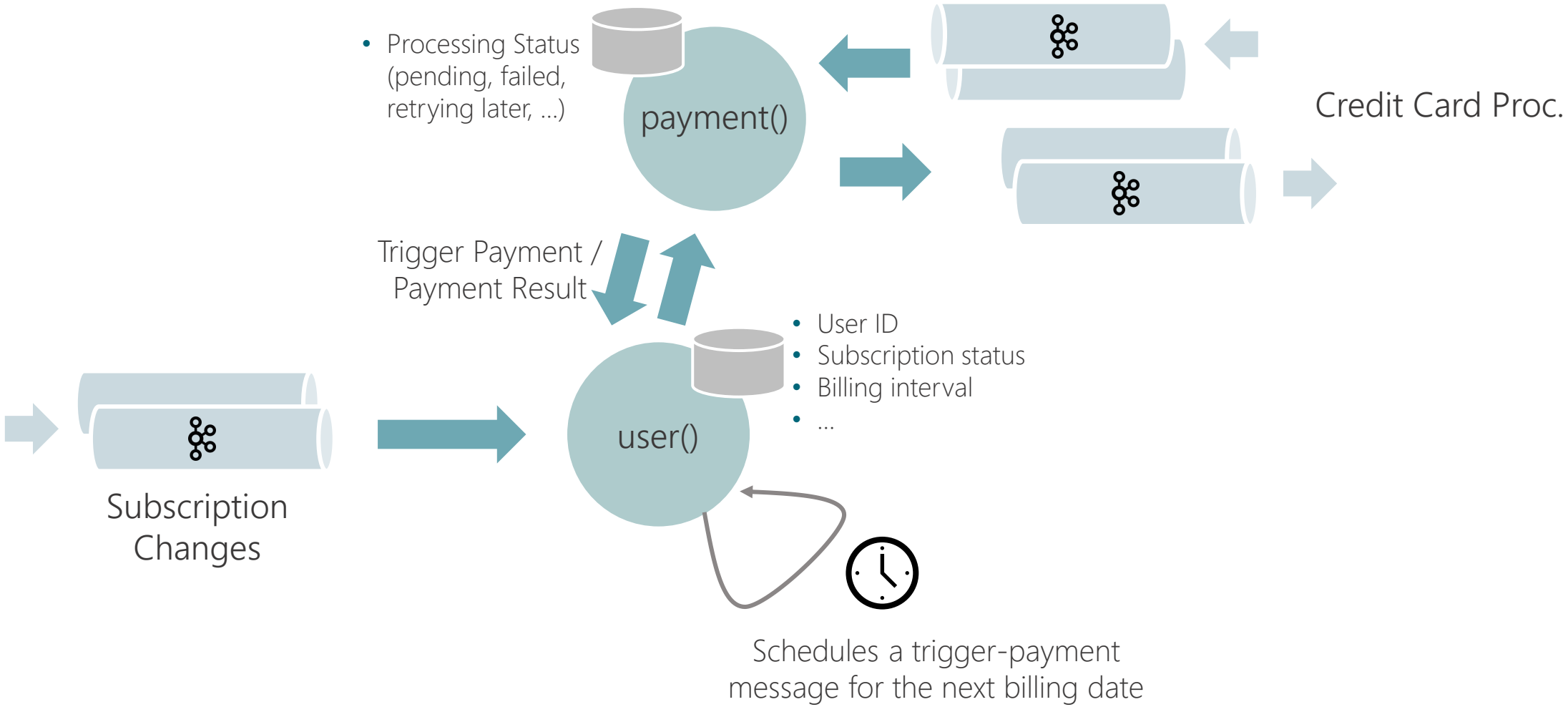

# Backup

# An Example and Demo

# A Billing Application




# Billing Application



# Code Samples

```python
@functions.bind("demo/subscription")
def model(context, message):
    if message.Is(NewSubscription.DESCRIPTOR):
        handle_new_subscription(context, message)
    elif message.Is(TimerMessage.DESCRIPTOR):
        handle_timer(context, message)
    elif message.Is(PaymentResult.DESCRIPTOR):
        handle_payment_result(context, message)
    else:
        raise ValueError('unknown message type ' + message)


def handle_new_subscription(context, message):
    sub = NewSubscription()
    message.Unpack(sub)

    # remember the subscription details in state
    state = SubscriptionState()
    state.user_id = sub.user_id
    state.next_payment_due = pay_date(sub.pay_interval)

    context.state('subscription').pack(state)

    # send to ourselves a reminder to start a payment process.
    schedule(context, state.next_payment_due.seconds_left)
```

```python
def handle_timer(context, message):
    """it's time to pay - send out a payment request."""
    state = context.state('subscription').unpack(SubscriptionState)
    payment = Payment()
    payment.user_id = state.user_id

    context.pack_and_send("demo/payment", context.address.identity, payment)


def schedule(context, seconds_remaining):
    # send to ourselves a reminder to start a payment process.
    context.pack_and_send_after(timedelta(seconds=seconds_remaining),
                                context.address.typename(),
                                context.address.identity,
                                TimerMessage())


def handle_new_payment(context, message):
    payment = Payment()
    message.Unpack(payment)

    state = PaymentState()
    state.payment = payment
    state.payment_id = new_UUID()
    state.success = False

    context.state('payment').pack(state)

    req = payment_request(payment, state.payment_id)

    # send out the payment request
    out_record = kafka_egress_record(topic="payment_requests", value=req)
    context.pack_and_send_egress("demo/kafka", out_record)
```