cloudstate

cloudstate.io

serverless 2.0 with cloudstate

Sean Walsh | Field CTO and Cloud Evangelist @ Lightbend

> “We predict that serverless computing will grow to dominate the future of cloud computing.”

–Berkely CS Department

why serverless 2.0?

FaaS was a great start and paved the way, but it’s only the first step

FaaS != serverless

we need serverless to allow coarse-grained, general purpose applications

FaaS

good for:

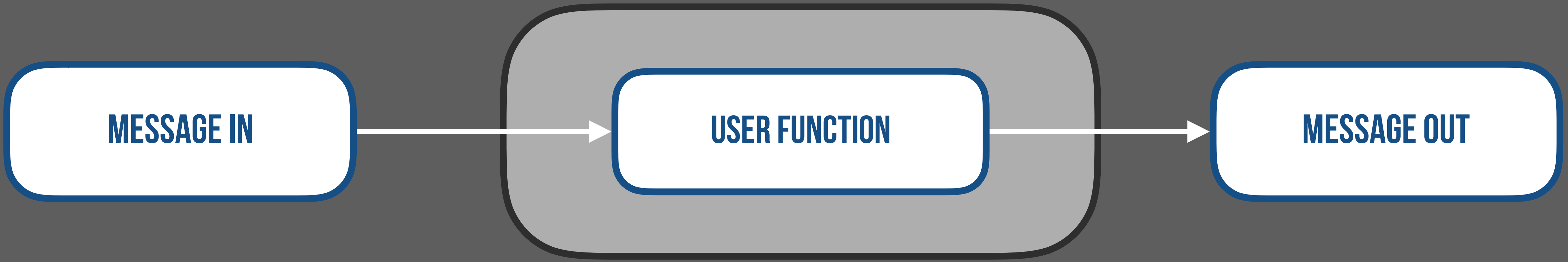
- embarrassingly parallel processes
- orchestration
- stateless web applications
- job scheduling and orchestration

bad at

- reasoning about as a holistic application
- guarantees around responsiveness and resilience
- general purpose applications

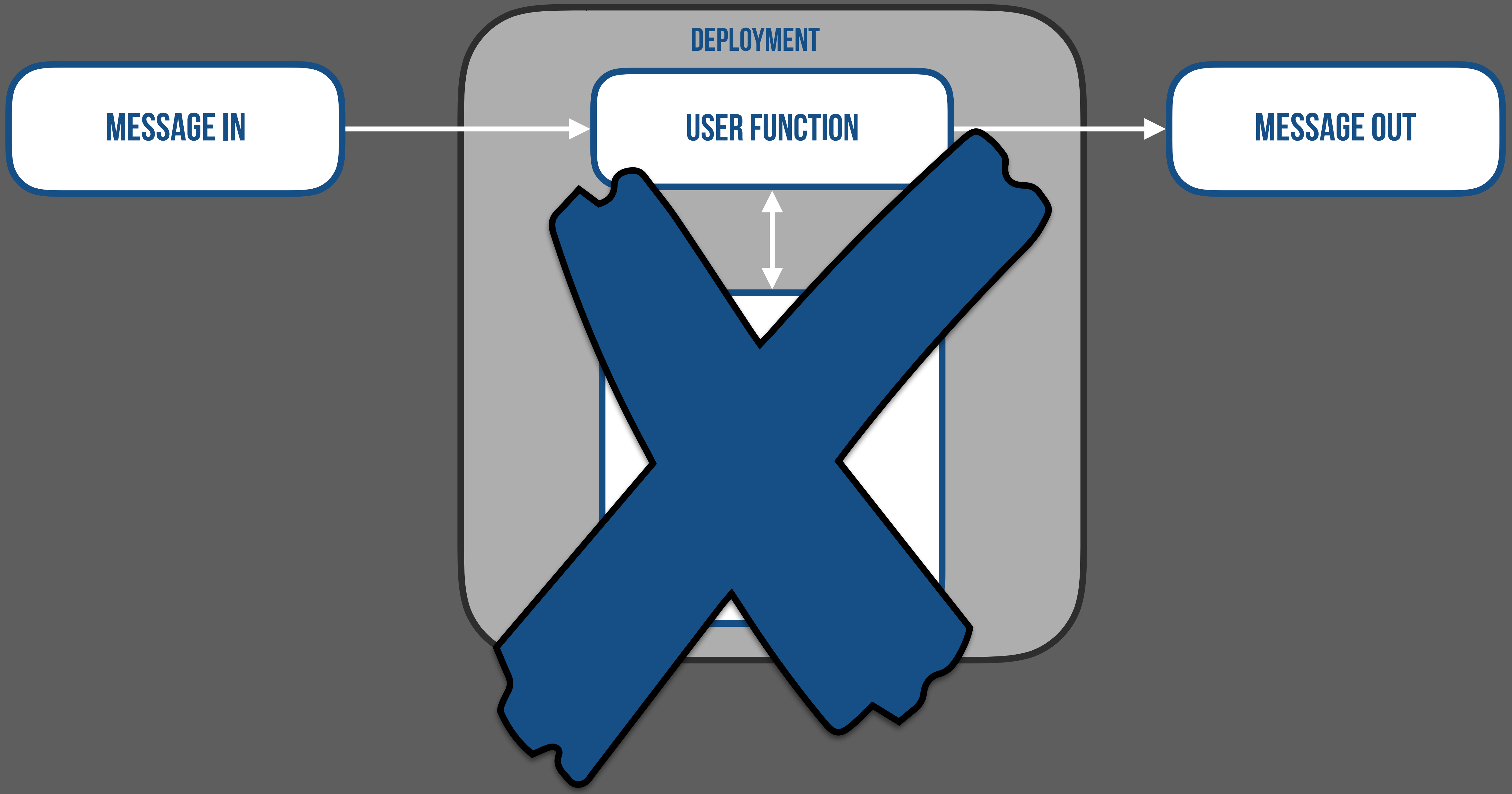
FaaS

abstracting over communication

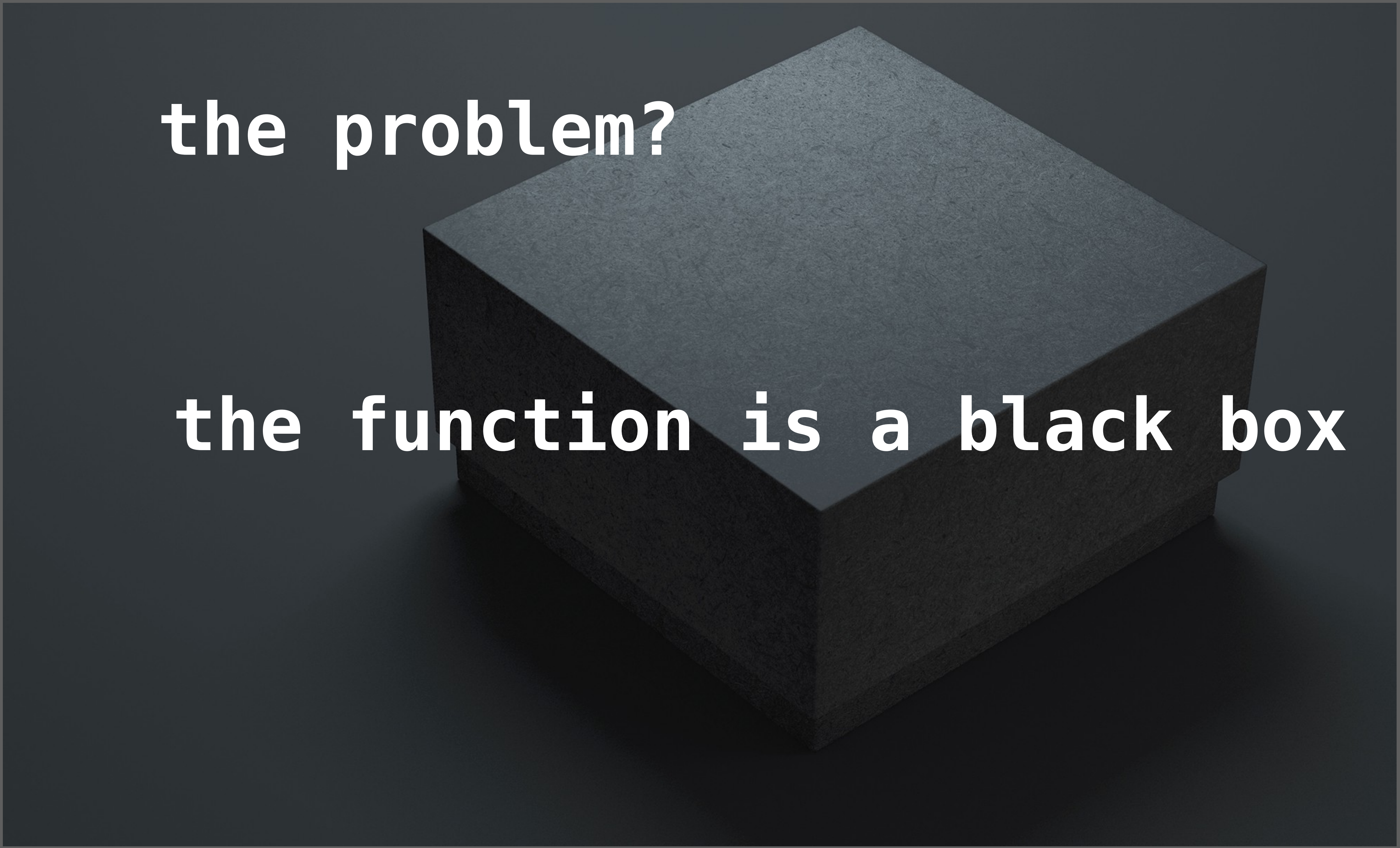


- works great as long as stateless or embarrassingly parallel
- operational concerns handled (GREAT)

FaaS



FaaS



the problem?

the function is a black box

state

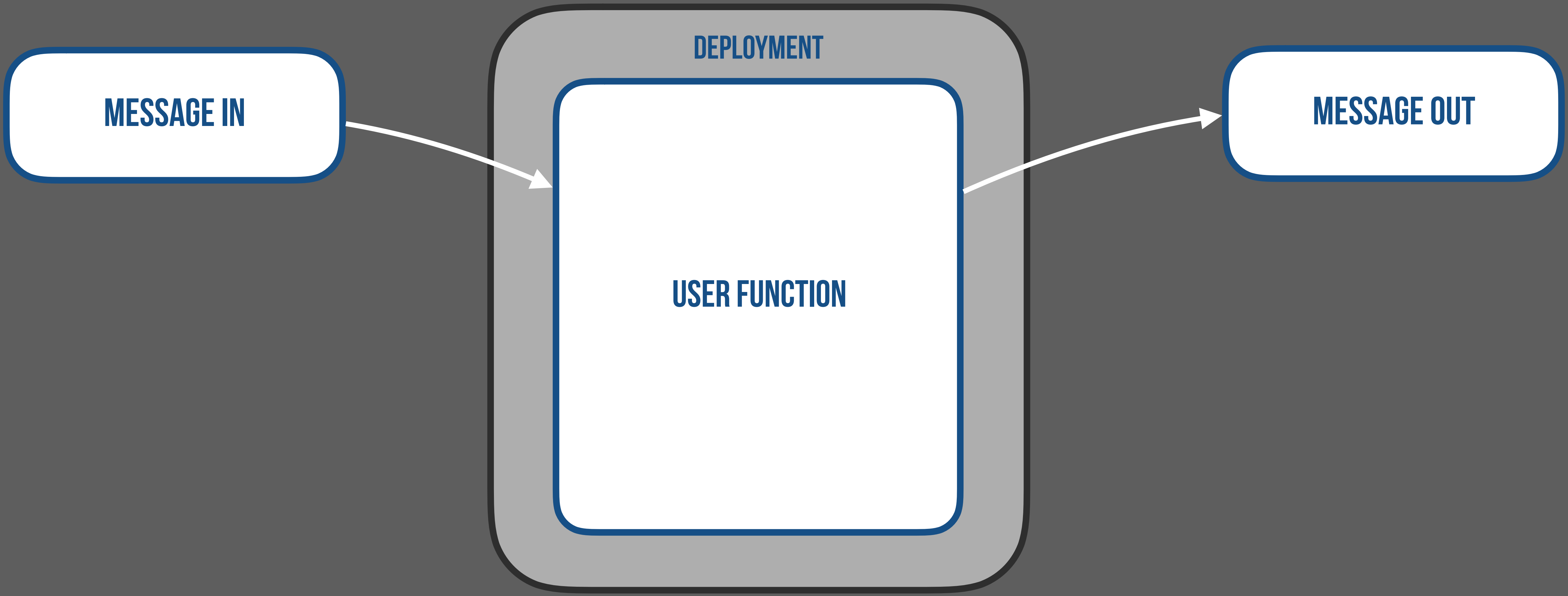
serverless 2.0

realtime database access must be removed to allow autonomy and reliability of the functions

(guarantees are not possible if we pass in the entire database to a function, or allowed unbridled reads)

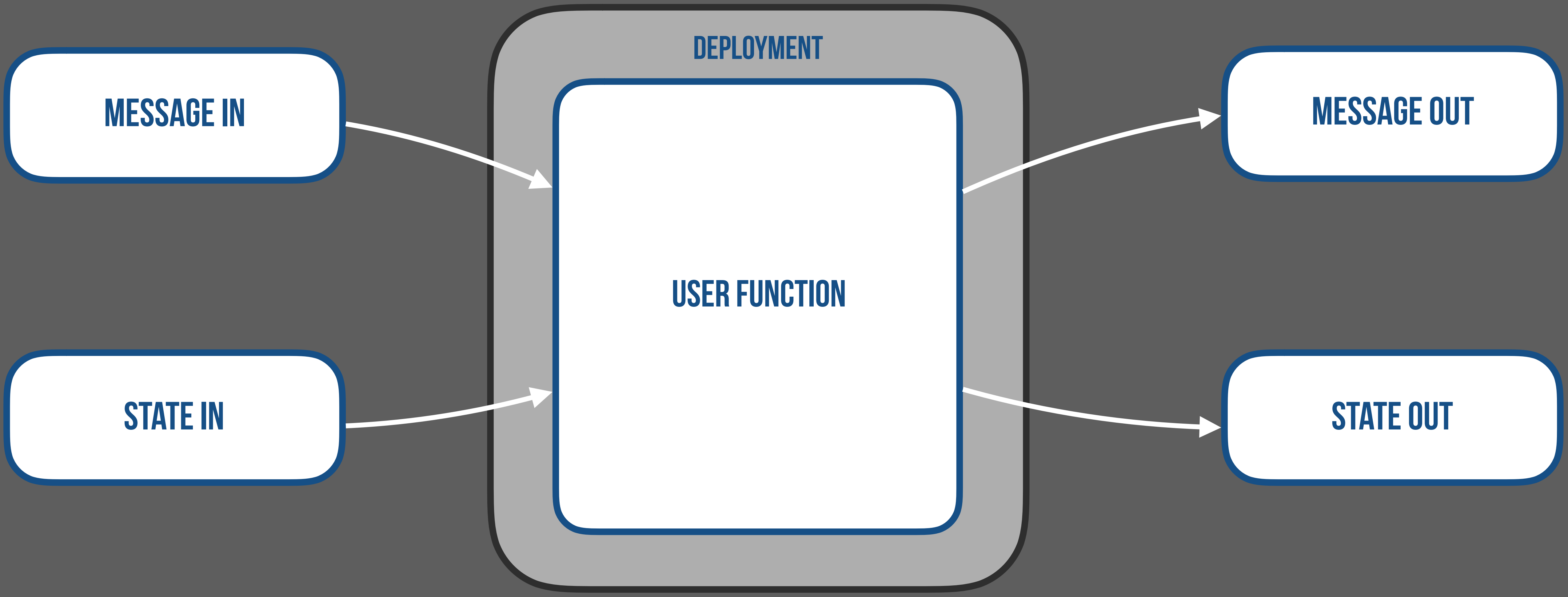
FaaS

`abstracting over communication`



stateful serverless

abstracting over state



enter…

what is cloudstate?

cloudstate is a distributed, clustered and stateful cloud runtime, providing a zero-ops experience, with polyglot client support

(essentially serverless 2.0)

cloudstate

CLOUDSTATE IS OPEN SOURCE,
UTILIZING BEST OF BREED
TECHNOLOGIES, HARNESSING ALL
THEIR POWER, WHILE REMOVING
ALL THEIR COMPLEXITY

cloudstate

don’t worry about:

- complexities of distributed systems
- managing state, databases, service meshes
- message routing, failover, recovery
- running and operationalizing applications

cloudstate

technical highlights:

- polyglot:python, java, spring, go, rust, javascript, .net, swift, scala and more…
- powerful state models: event sourcing, CQRS, key/value, CRUD, CRDTs
- polydb: SQL, NoSQL, NewSQL, in-memory
- leverages akka, gRPC, knative, GraalVM, running on kubernetes

“freedom is not so much the absence of restrictions as finding the right ones, the liberating restrictions.”

–Timothy Keller

one very important constraint

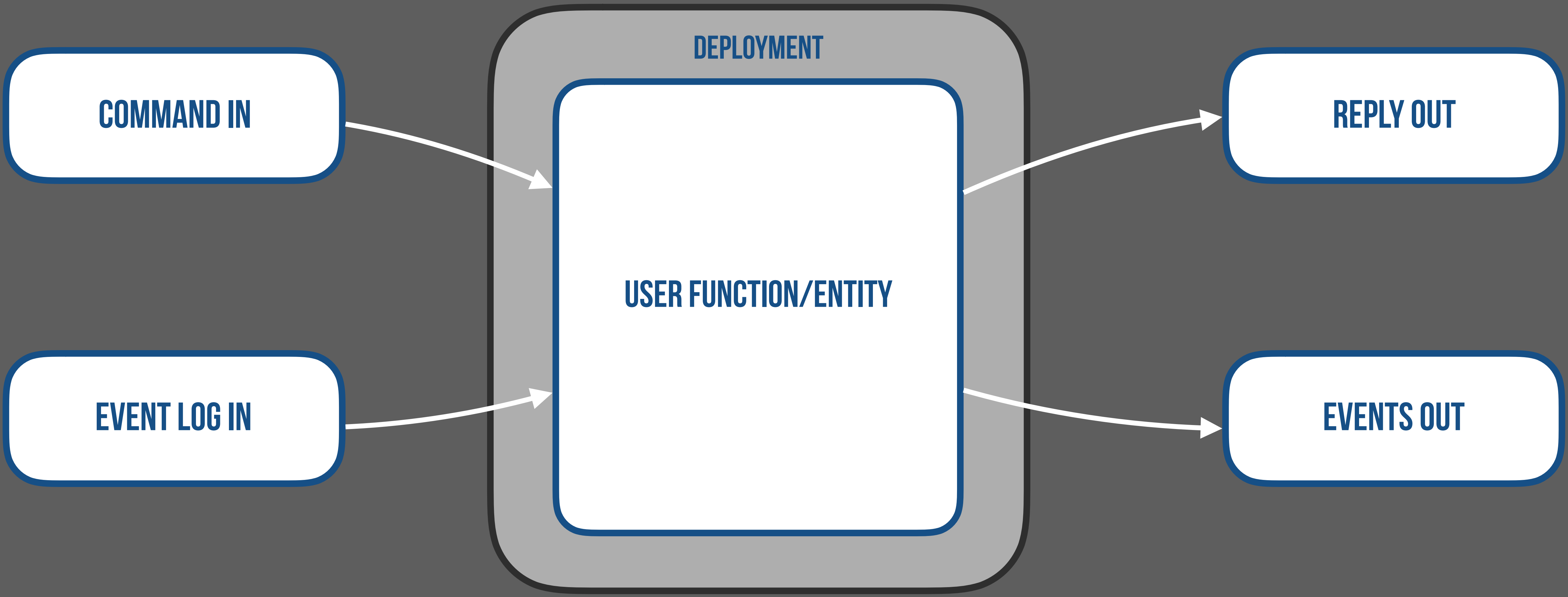
event

sourcing

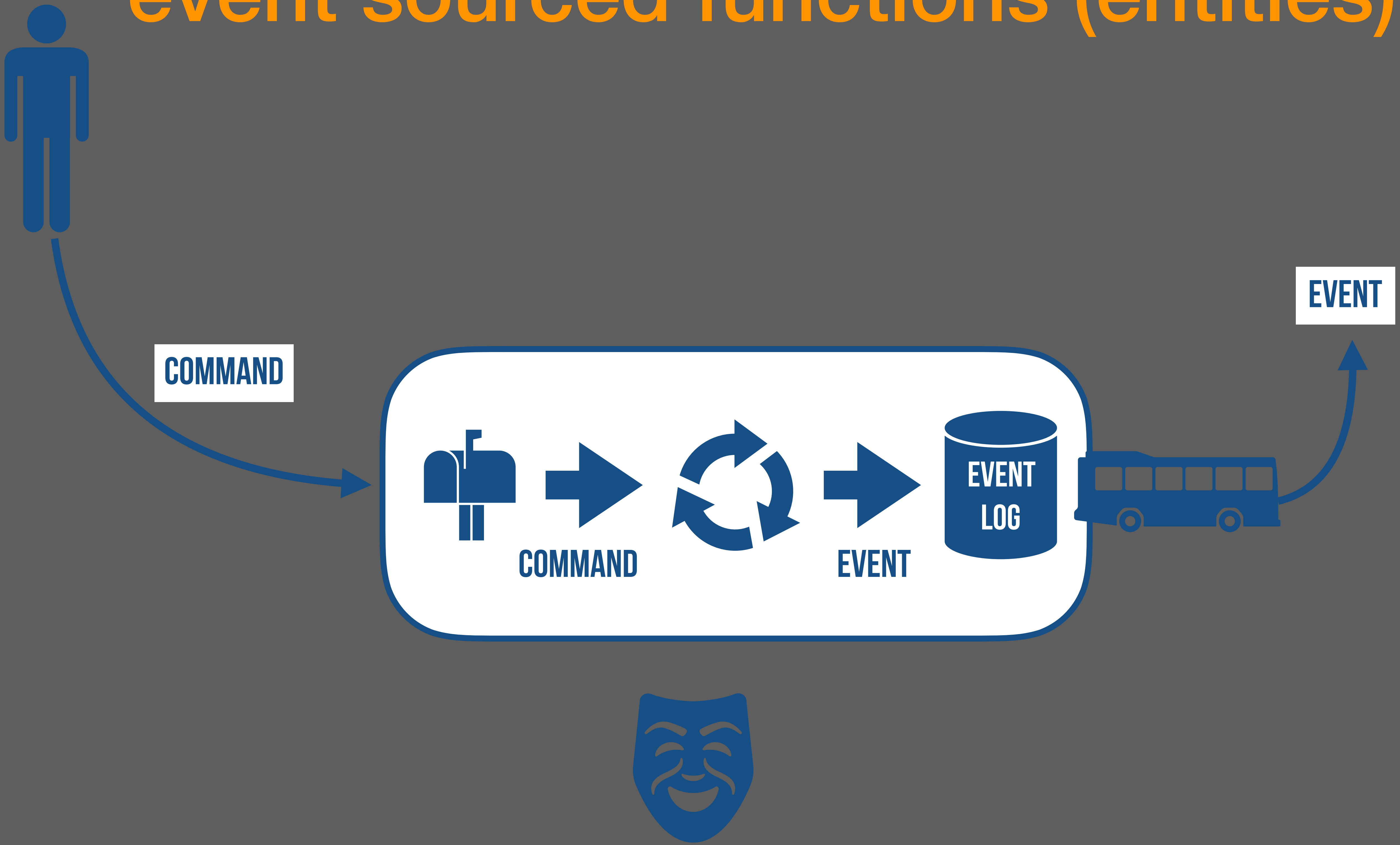
benefits of event sourcing

- single source of truth with full history
- allows for memory image (durable in-memory state)
- avoids object-relational mismatch
- allows subscription to state changes
- mechanical sympathy (single writer principle)

cloudstate: event sourcing

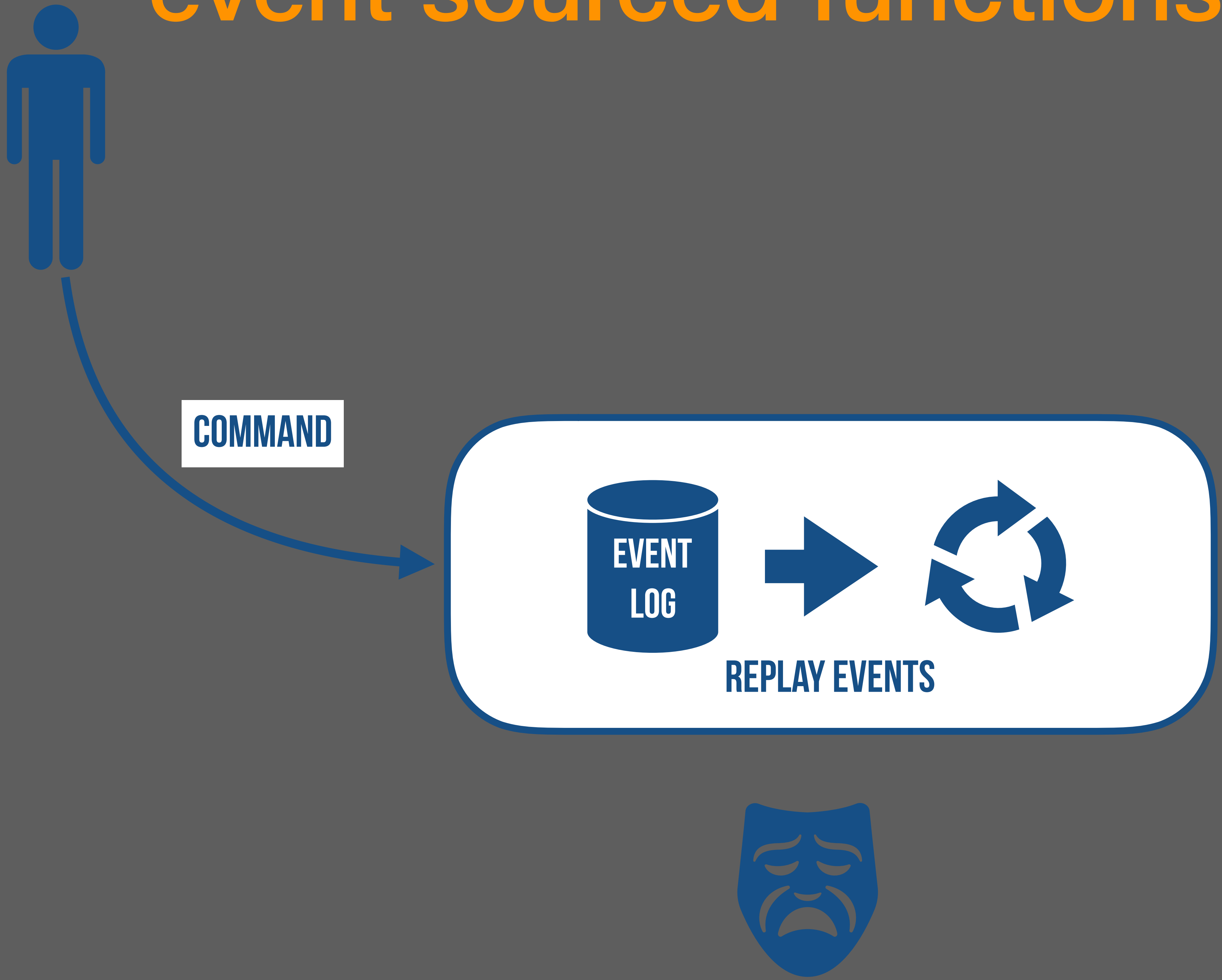


event sourced functions (entities)



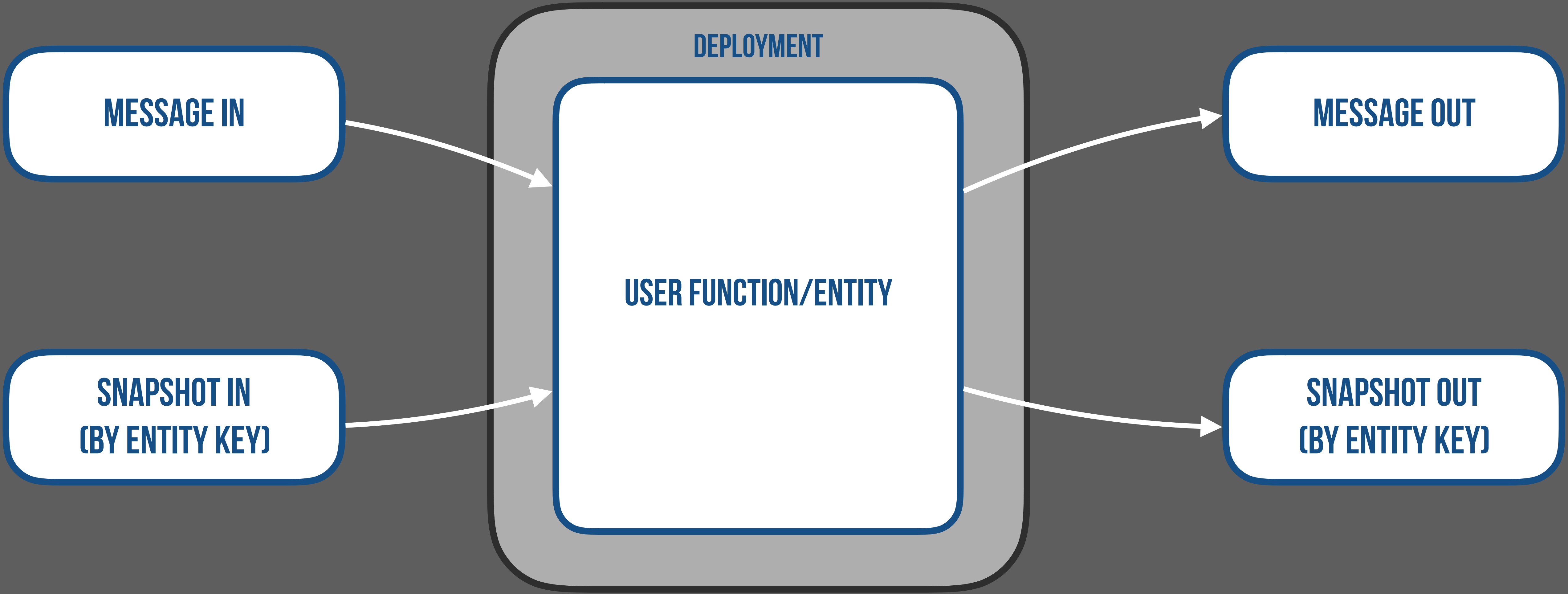
HAPPY PATH

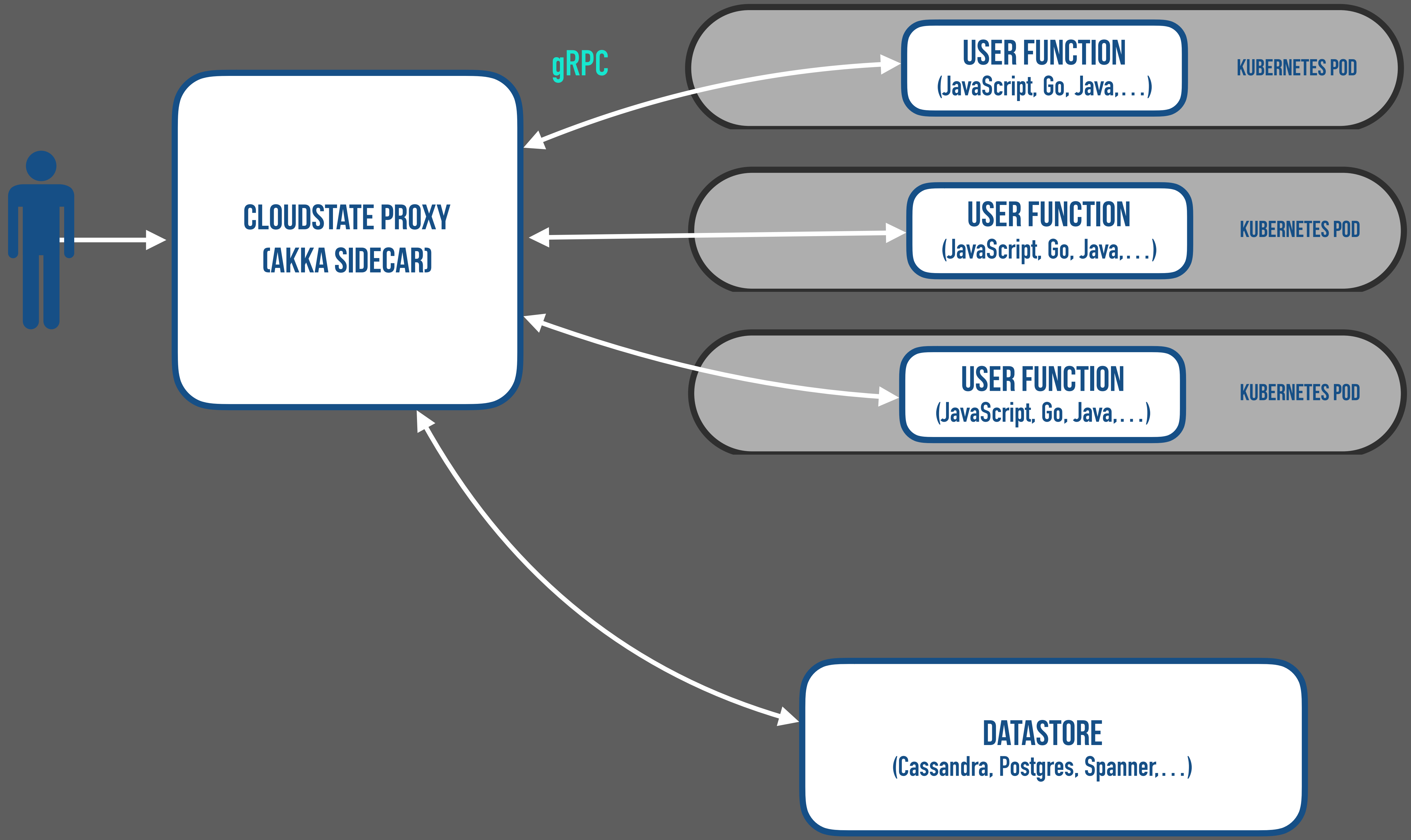
event sourced functions (entities)



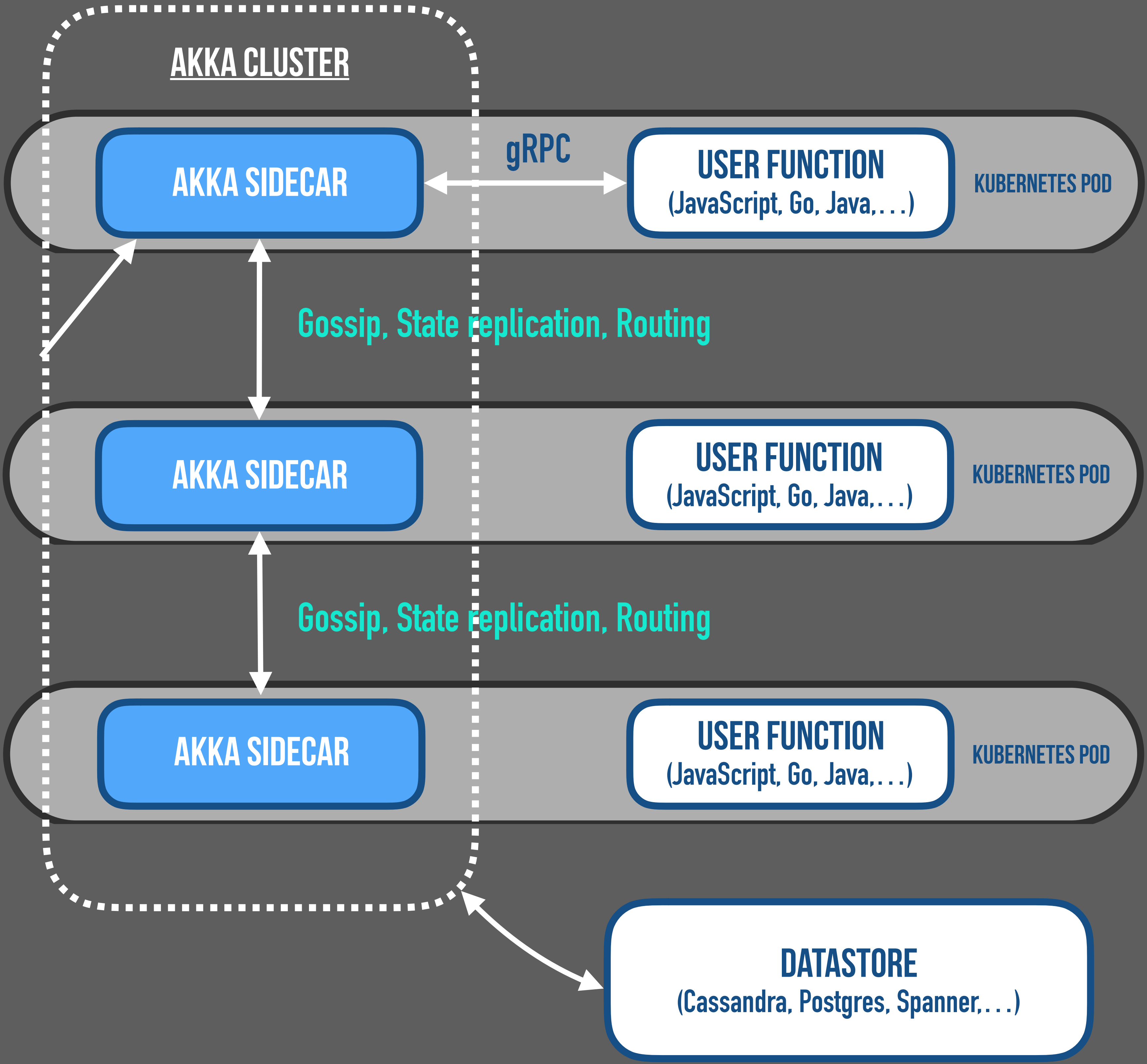
SAD PATH, RECOVER FROM FAILURE

(and yeah you can still do CRUD)




gRPC
CLOUDSTATE PROXY
(AKKA SIDECAR)
USER FUNCTION
(JavaScript, Go, Java,. . .)
KUBERNETES POD
USER FUNCTION
(JavaScript, Go, Java,. . .)
KUBERNETES POD
USER FUNCTION
(JavaScript, Go, Java,. . .)
KUBERNETES POD
DATASTORE
(Cassandra, Postgres, Spanner,. . .)

cloudstate architecture


AKKA CLUSTER
AKKA SIDECAR
gRPC
USER FUNCTION
(JavaScript, Go, Java,...)
KUBERNETES POD
Gossip, State replication, Routing
AKKA SIDECAR
USER FUNCTION
(JavaScript, Go, Java,...)
KUBERNETES POD
Gossip, State replication, Routing
AKKA SIDECAR
USER FUNCTION
(JavaScript, Go, Java,...)
KUBERNETES POD
DATASTORE
(Cassandra, Postgres, Spanner,...)

as a managed service

- Pay as you go:
  - on–demand instance creation, passivation and failover
  - autoscaling–up and down

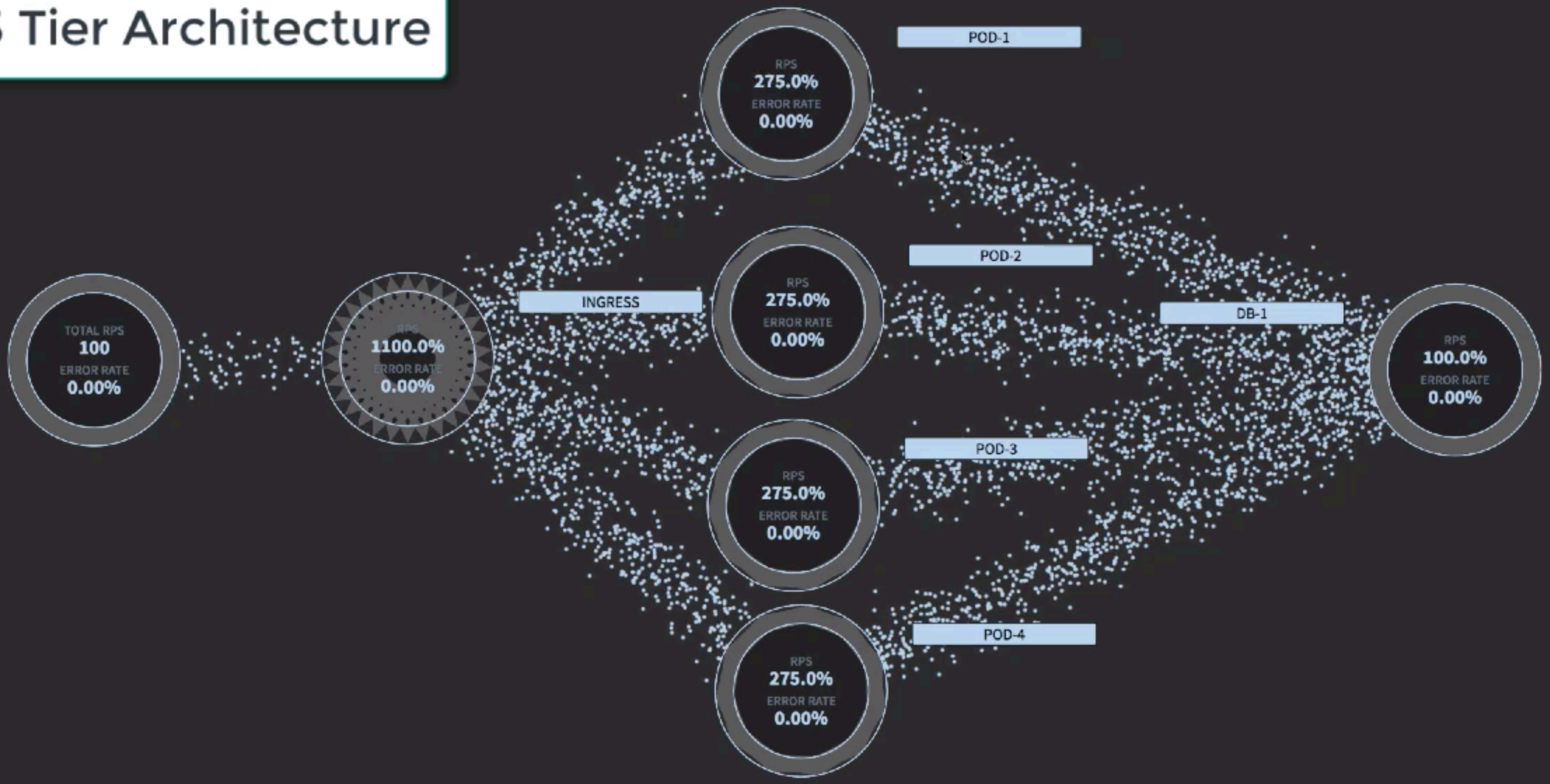
- ZeroOps:
  - automated message routing
  - automated state management
  - Automated deployment, provisioning, upgrades

multitenancy

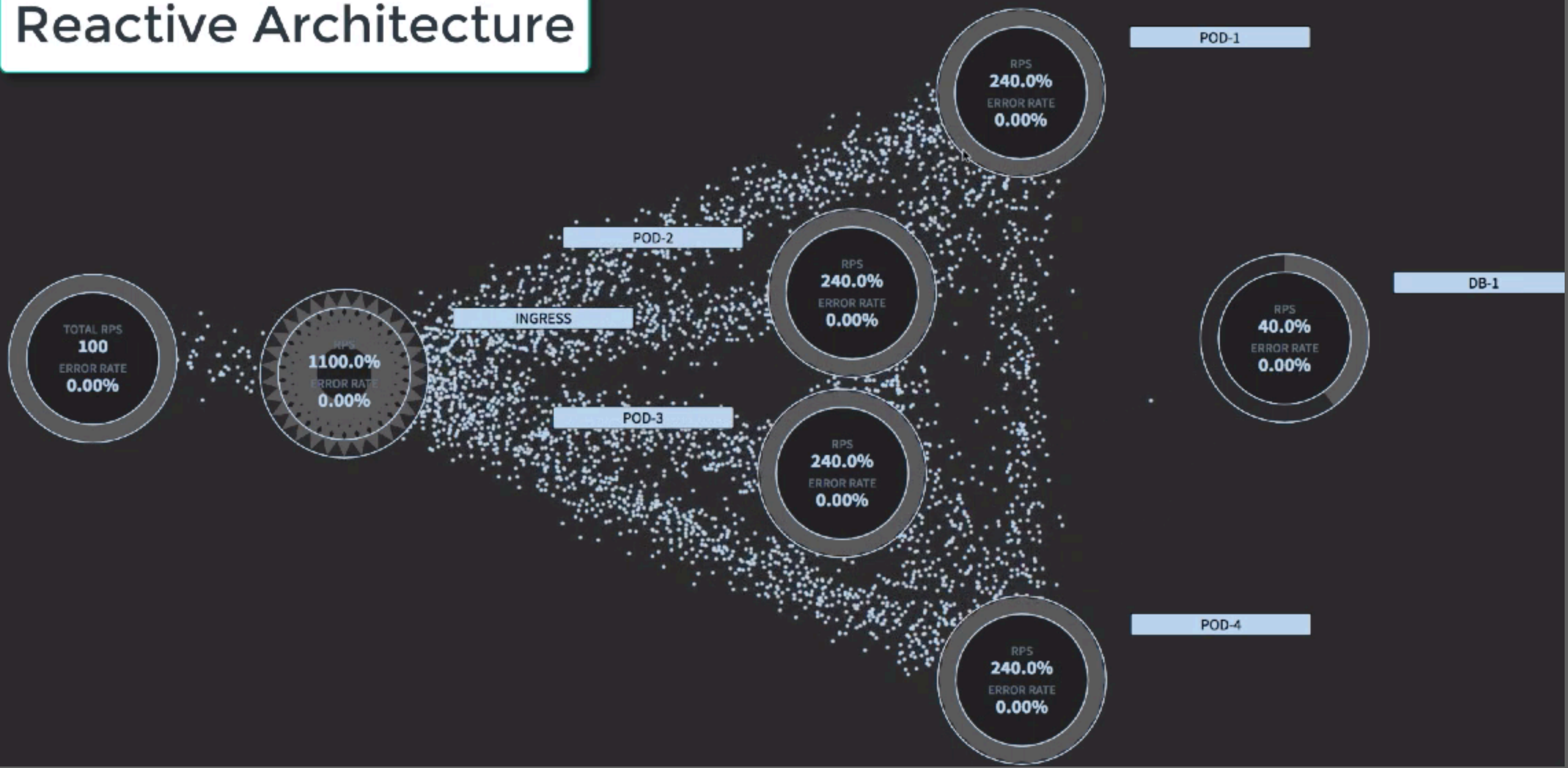
- FaaS:
  - inadequate bulkheading: neighbor’s function can hog resources

- cloudstate:
  - multitenancy from the ground up via pods
  - complete bulkingheading: even at the data level
  - complete security due to clear separations

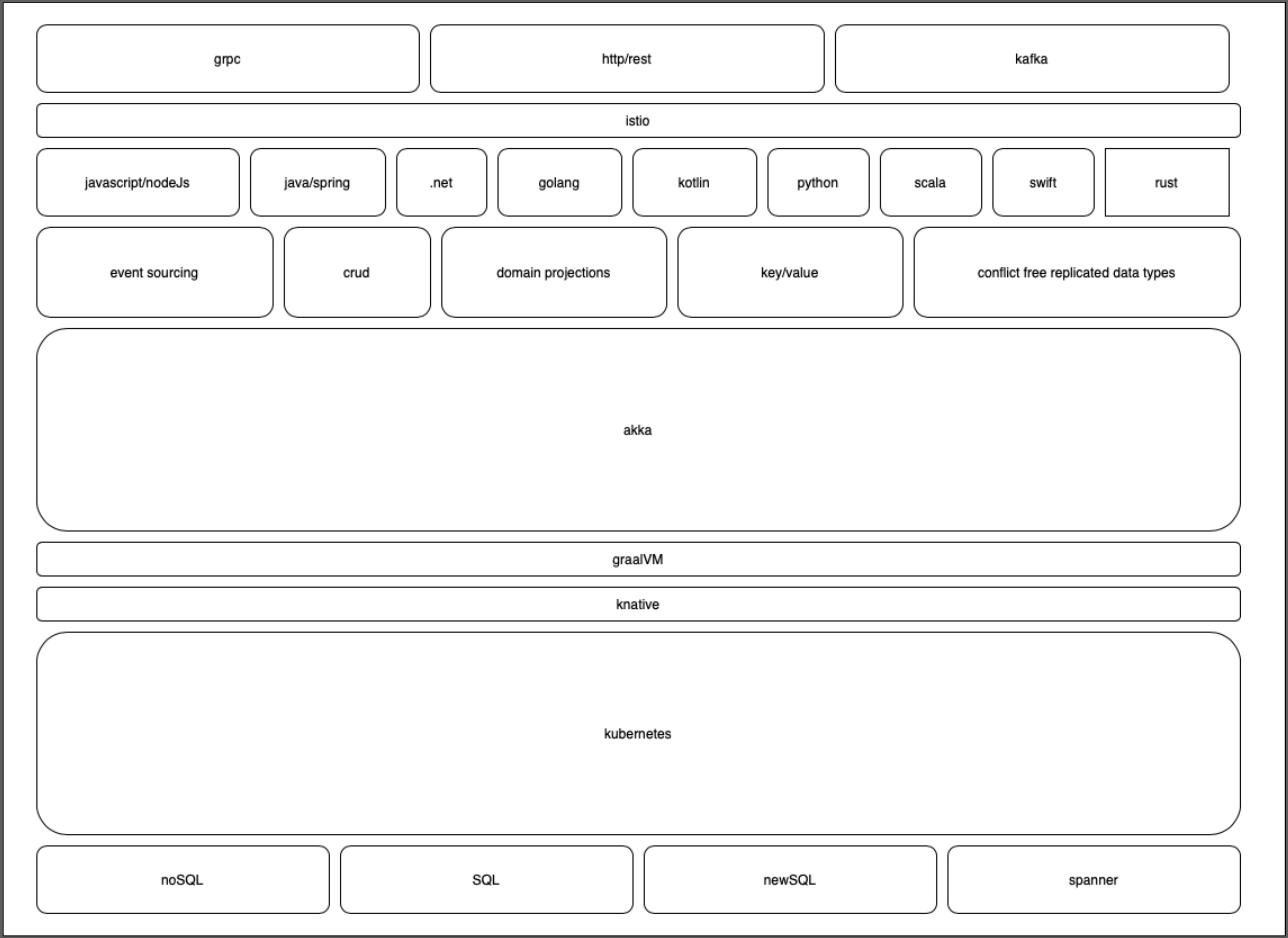
3 Tier Architecture



Reactive Architecture



cloudstate architecture



grpc | http/rest | kafka

istio

javascript/nodeJs | java/spring | .net | golang | kotlin | python | scala | swift | rust

event sourcing | crud | domain projections | key/value | conflict free replicated data types

akka

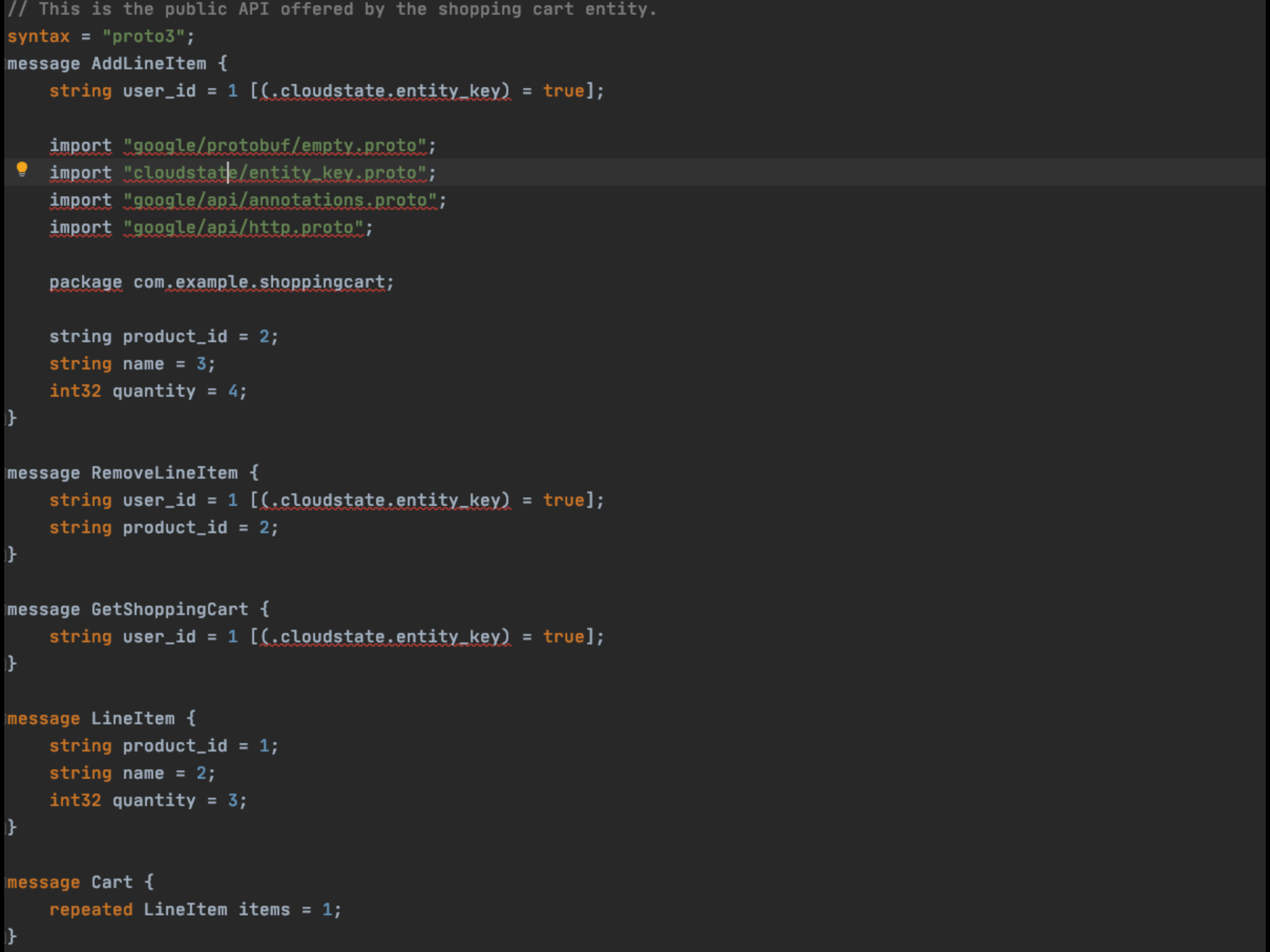
graalVM

knative

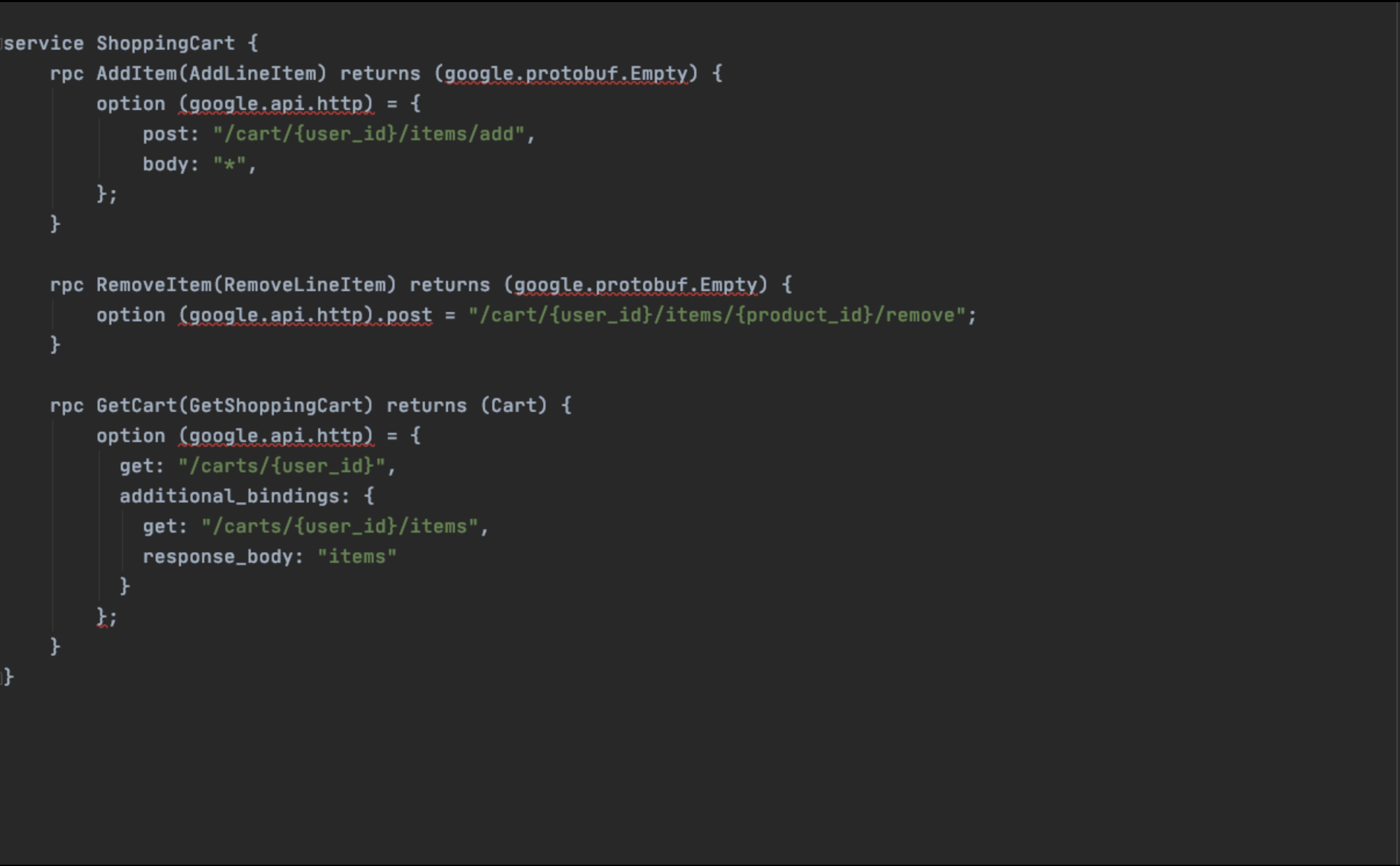
kubernetes

noSQL | SQL | newSQL | spanner

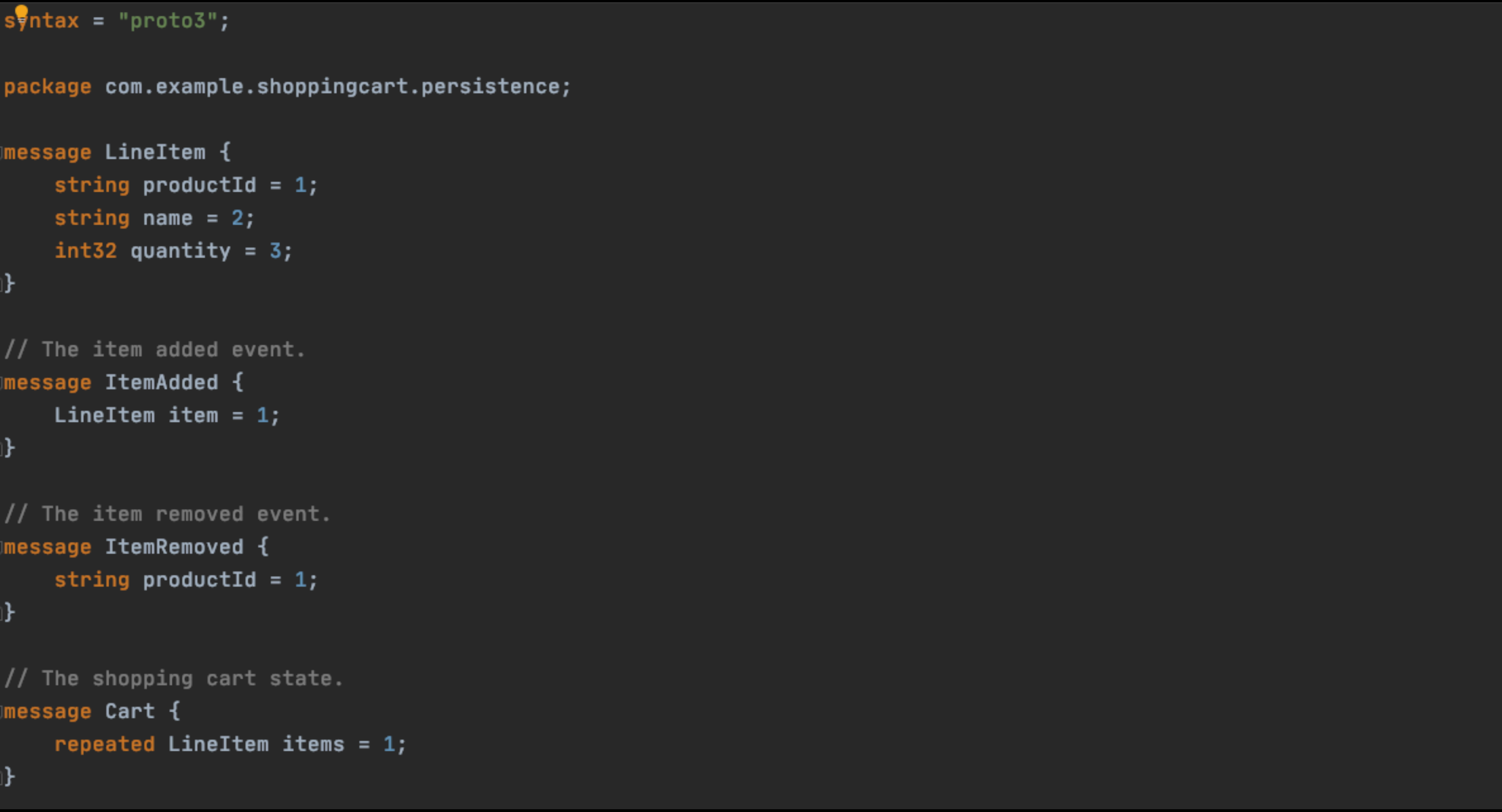
LET’S LOOK AT SOME CODE!



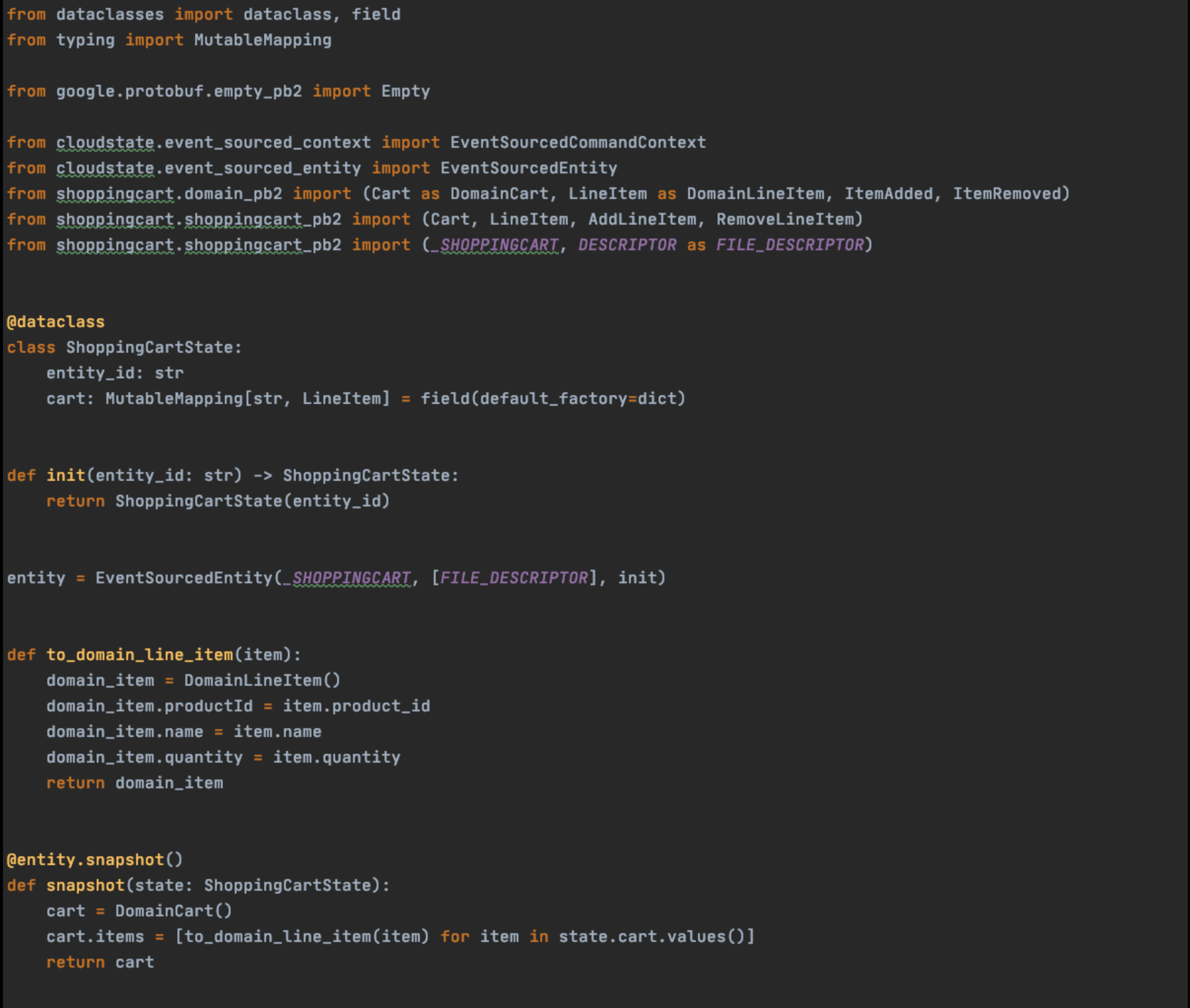
```
// This is the public API offered by the shopping cart entity.
syntax = "proto3";
message AddLineItem {
    string user_id = 1 [(.cloudstate.entity_key) = true];

    import "google/protobuf/empty.proto";
    import "cloudstate/entity_key.proto";
    import "google/api/annotations.proto";
    import "google/api/http.proto";

    package com.example.shoppingcart;

    string product_id = 2;
    string name = 3;
    int32 quantity = 4;
}

message RemoveLineItem {
    string user_id = 1 [(.cloudstate.entity_key) = true];
    string product_id = 2;
}

message GetShoppingCart {
    string user_id = 1 [(.cloudstate.entity_key) = true];
}

message LineItem {
    string product_id = 1;
    string name = 2;
    int32 quantity = 3;
}

message Cart {
    repeated LineItem items = 1;
}
```



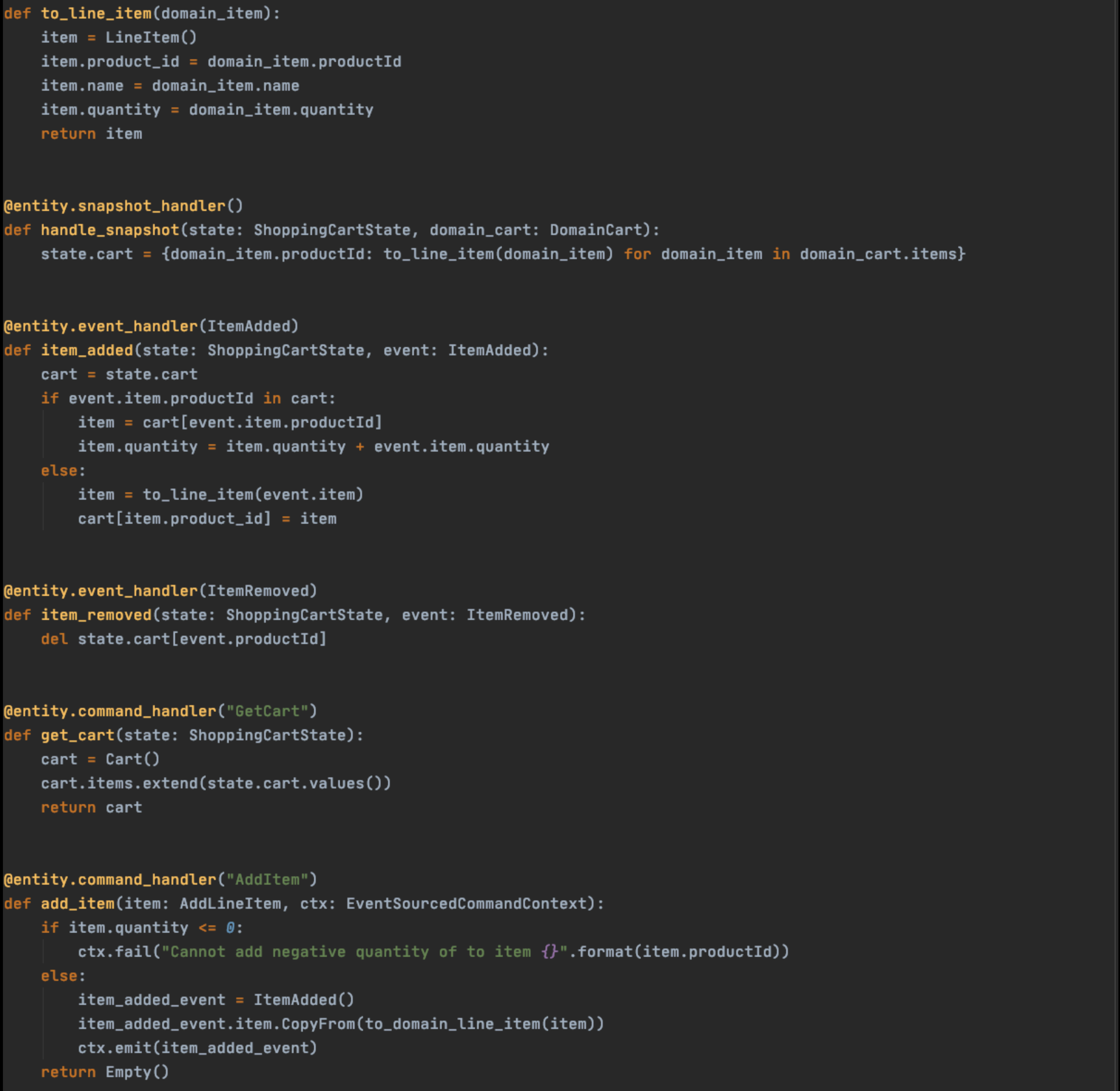
```
service ShoppingCart {
    rpc AddItem(AddLineItem) returns (google.protobuf.Empty) {
        option (google.api.http) = {
            post: "/cart/{user_id}/items/add",
            body: "*",
        };
    }

    rpc RemoveItem(RemoveLineItem) returns (google.protobuf.Empty) {
        option (google.api.http).post = "/cart/{user_id}/items/{product_id}/remove";
    }

    rpc GetCart(GetShoppingCart) returns (Cart) {
        option (google.api.http) = {
          get: "/carts/{user_id}",
          additional_bindings: {
            get: "/carts/{user_id}/items",
            response_body: "items"
          }
        };
    }
}
```



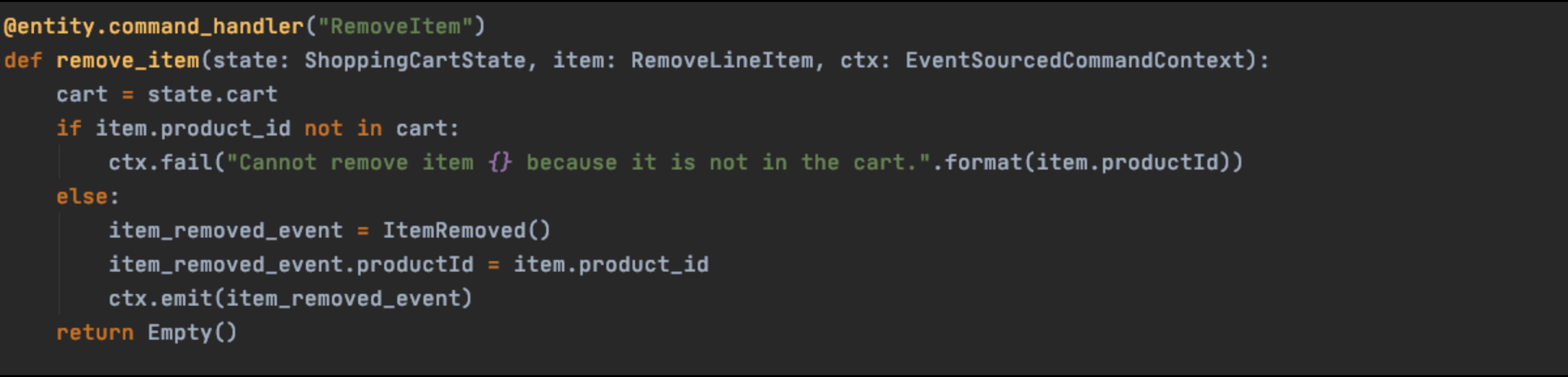
```
syntax = "proto3";

package com.example.shoppingcart.persistence;

message LineItem {
    string productId = 1;
    string name = 2;
    int32 quantity = 3;
}

// The item added event.
message ItemAdded {
    LineItem item = 1;
}

// The item removed event.
message ItemRemoved {
    string productId = 1;
}

// The shopping cart state.
message Cart {
    repeated LineItem items = 1;
}
```



```python
from dataclasses import dataclass, field
from typing import MutableMapping

from google.protobuf.empty_pb2 import Empty

from cloudstate.event_sourced_context import EventSourcedCommandContext
from cloudstate.event_sourced_entity import EventSourcedEntity
from shoppingcart.domain_pb2 import (Cart as DomainCart, LineItem as DomainLineItem, ItemAdded, ItemRemoved)
from shoppingcart.shoppingcart_pb2 import (Cart, LineItem, AddLineItem, RemoveLineItem)
from shoppingcart.shoppingcart_pb2 import (_SHOPPINGCART, DESCRIPTOR as FILE_DESCRIPTOR)


@dataclass
class ShoppingCartState:
    entity_id: str
    cart: MutableMapping[str, LineItem] = field(default_factory=dict)


def init(entity_id: str) -> ShoppingCartState:
    return ShoppingCartState(entity_id)


entity = EventSourcedEntity(_SHOPPINGCART, [FILE_DESCRIPTOR], init)


def to_domain_line_item(item):
    domain_item = DomainLineItem()
    domain_item.productId = item.product_id
    domain_item.name = item.name
    domain_item.quantity = item.quantity
    return domain_item


@entity.snapshot()
def snapshot(state: ShoppingCartState):
    cart = DomainCart()
    cart.items = [to_domain_line_item(item) for item in state.cart.values()]
    return cart
```



```python
def to_line_item(domain_item):
    item = LineItem()
    item.product_id = domain_item.productId
    item.name = domain_item.name
    item.quantity = domain_item.quantity
    return item


@entity.snapshot_handler()
def handle_snapshot(state: ShoppingCartState, domain_cart: DomainCart):
    state.cart = {domain_item.productId: to_line_item(domain_item) for domain_item in domain_cart.items}


@entity.event_handler(ItemAdded)
def item_added(state: ShoppingCartState, event: ItemAdded):
    cart = state.cart
    if event.item.productId in cart:
        item = cart[event.item.productId]
        item.quantity = item.quantity + event.item.quantity
    else:
        item = to_line_item(event.item)
        cart[item.product_id] = item


@entity.event_handler(ItemRemoved)
def item_removed(state: ShoppingCartState, event: ItemRemoved):
    del state.cart[event.productId]


@entity.command_handler("GetCart")
def get_cart(state: ShoppingCartState):
    cart = Cart()
    cart.items.extend(state.cart.values())
    return cart


@entity.command_handler("AddItem")
def add_item(item: AddLineItem, ctx: EventSourcedCommandContext):
    if item.quantity <= 0:
        ctx.fail("Cannot add negative quantity of to item {}".format(item.productId))
    else:
        item_added_event = ItemAdded()
        item_added_event.item.CopyFrom(to_domain_line_item(item))
        ctx.emit(item_added_event)
    return Empty()
```



```python
@entity.command_handler("RemoveItem")
def remove_item(state: ShoppingCartState, item: RemoveLineItem, ctx: EventSourcedCommandContext):
    cart = state.cart
    if item.product_id not in cart:
        ctx.fail("Cannot remove item {} because it is not in the cart.".format(item.productId))
    else:
        item_removed_event = ItemRemoved()
        item_removed_event.productId = item.product_id
        ctx.emit(item_removed_event)
    return Empty()
```

ON BEHALF OF THE CLOUDSTATE.IO TEAM, THANKS!

the full sample can be found here:

https://github.com/cloudstateio/python-support