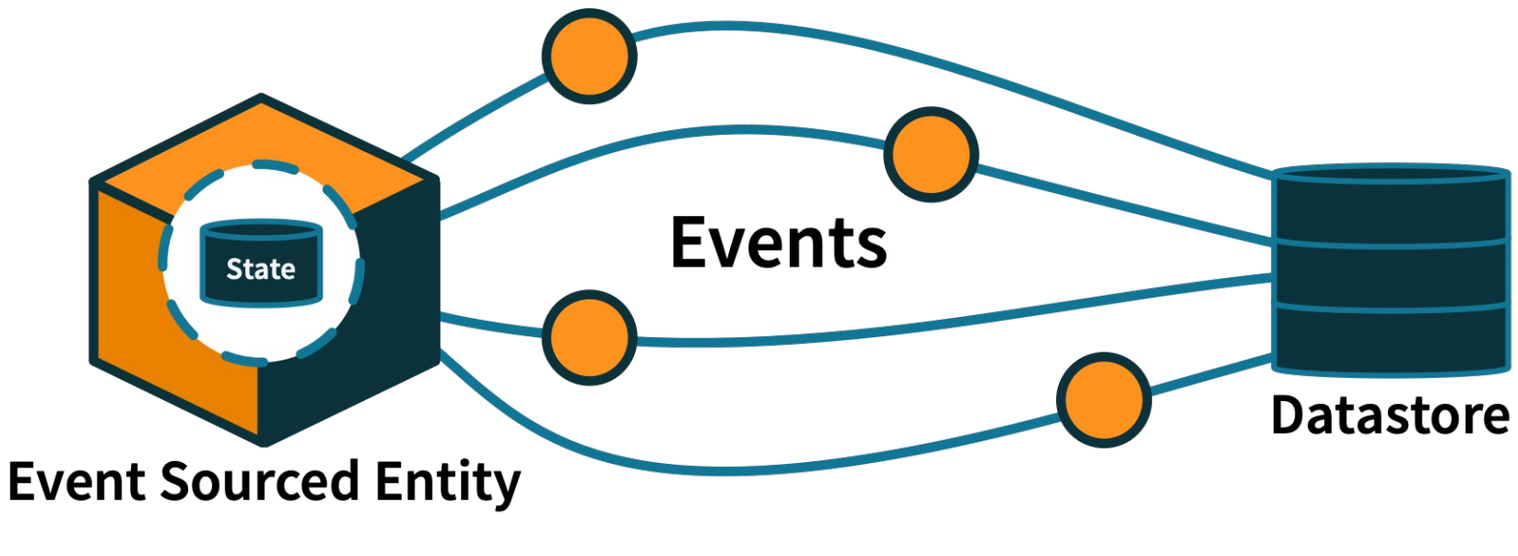
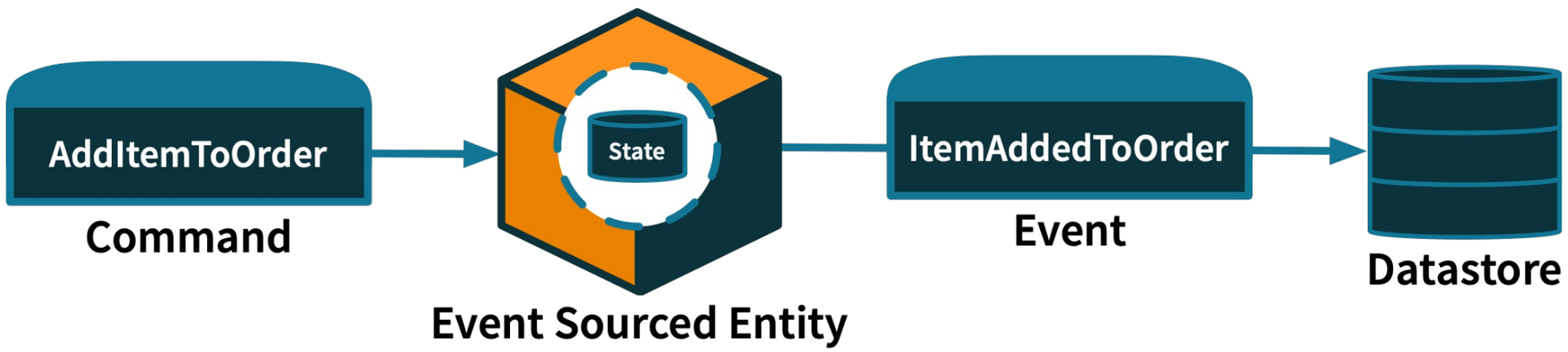
Event Sourcing

Event Sourcing



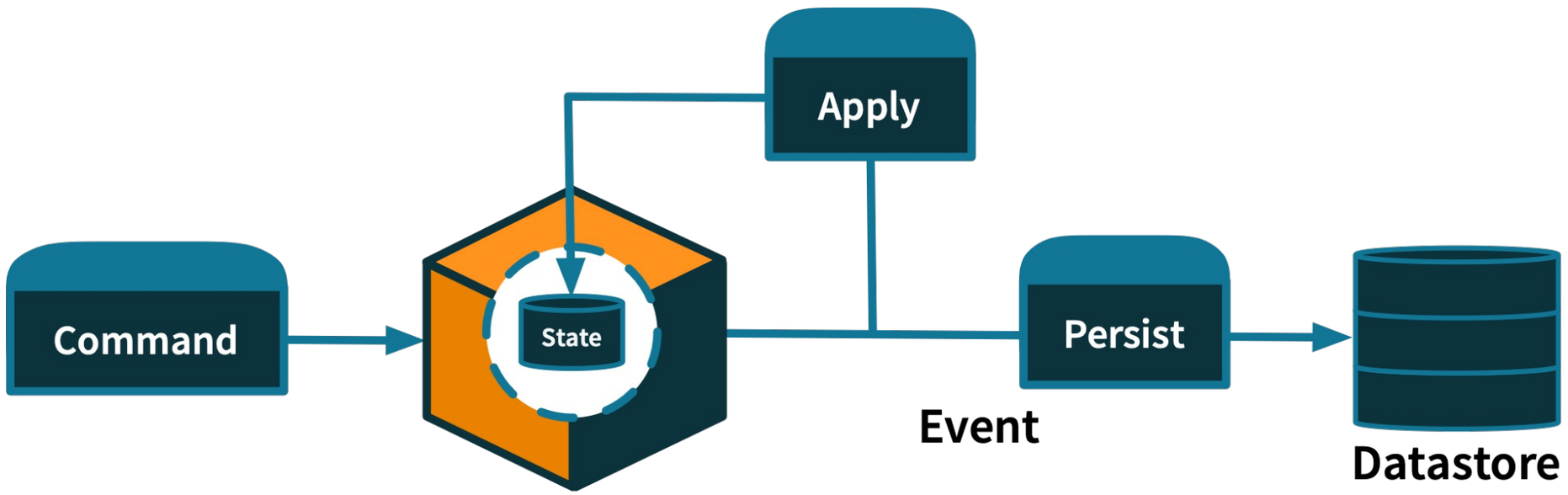
- *Event Sourcing* is a popular technique for highly *Elastic* and *Resilient* applications.
- Instead of persisting *State*, we persist *Events*.
- This abstraction allows us to limit our database complexity.
- The database becomes an implementation detail, handled by the platform.

Events



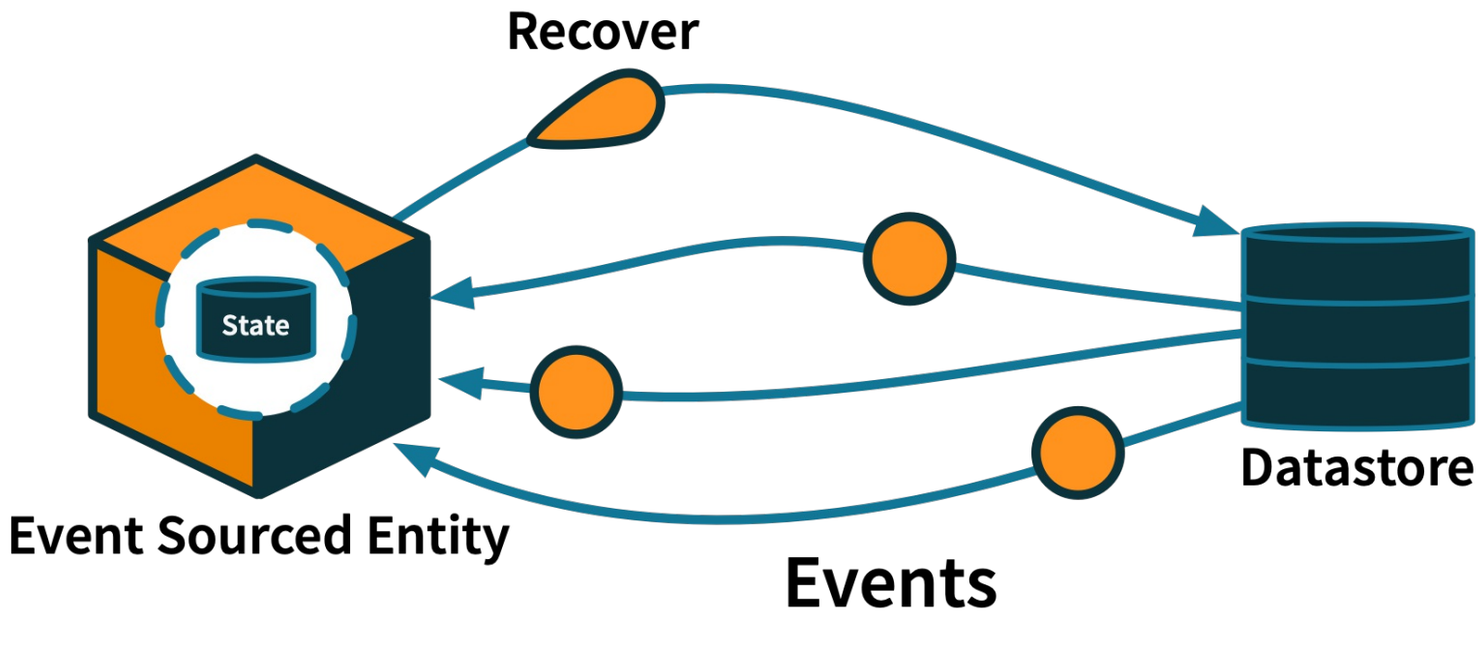
- So far, we have interacted with our *Entity* using *Commands*.
- *Commands* tell our *Entity* "what to do."
- *Events* record "what we did."
- I.E. *Commands* are the actions our system takes. *Events* are the result of those actions.
- Example: If the *Command* was AddItemToOrder the resulting *Event* might be ItemAddedToOrder.

Generating Events



- Each time a *Command* is issued to our *Entity*, one or more *Events* can be generated.
- These *Events* will be processed, resulting in a change to the in-memory *State* of our *Entity*.
- They are also persisted into the *Datastore*.
- What if our *Entity* is removed from memory and needs to be recovered?

Replaying Events



- Whenever the *State* of an *Entity* is loaded/recovered, *Akka Serverless* replays the *Events* for that *Entity*.
- As the *Events* are processed, we reapply the *State* changes recorded by the *Event*.
- If we apply those changes correctly, we should arrive at the proper *State*.
- But, what happens if our *Event* log grows too long?

Snapshots

- In some cases, it may take a long time to replay all the *Events*.
- To optimize recovery, we can also persist a *Snapshot*.
- A *Snapshot* persists the current *State* of the *Entity* in the *Datastore*.
- During recovery, we start from the most recent *Snapshot*, and apply *Events* that occur after.
- This reduces the number of *Events* we need to replay, and speeds up recovery.
- By default, Akka Serverless persists a *Snapshot* after every 100 *Events*.

Best Practices for Event Sourcing

Revealing Intent

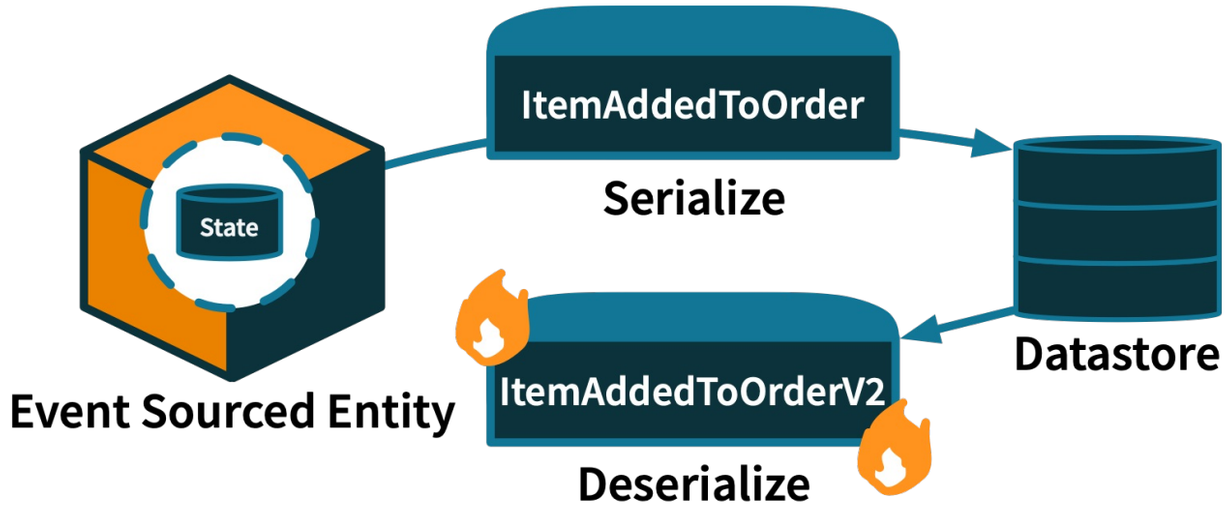
Vague	Intent Revealing
OrderCreated	OrderOpened
OrderUpdated	ItemAddedToOrder
OrderDeleted	OrderCancelled

- A common mistake is creating *Events* that don't reveal their intent.
- Events such as OrderUpdated aren't specific.
  - This reduces our ability to use *Events* for debugging.
- They often contain a full copy of the entire *State* (basically a Snapshot).
  - This can hinder performance as the entire *State* may be large.
- Intent revealing *Events* increase the value of *Event Sourcing*.
- Best Practice: Be specific, and use language from the domain.

Snapshots are an Optimization

- It may be tempting to create *Snapshots* often (perhaps every *Event*).
- Usually *Snapshots* are larger than *Events* (perhaps significantly).
- Creating too many *Snapshots* will increase your storage, network usage, and more.
- Your application may perform better with fewer *Snapshots*.
- Best Practice: Use *Snapshots* to optimze load times, but use them sparingly. Only adjust the snapshot interval if replaying entity events is taking too long.

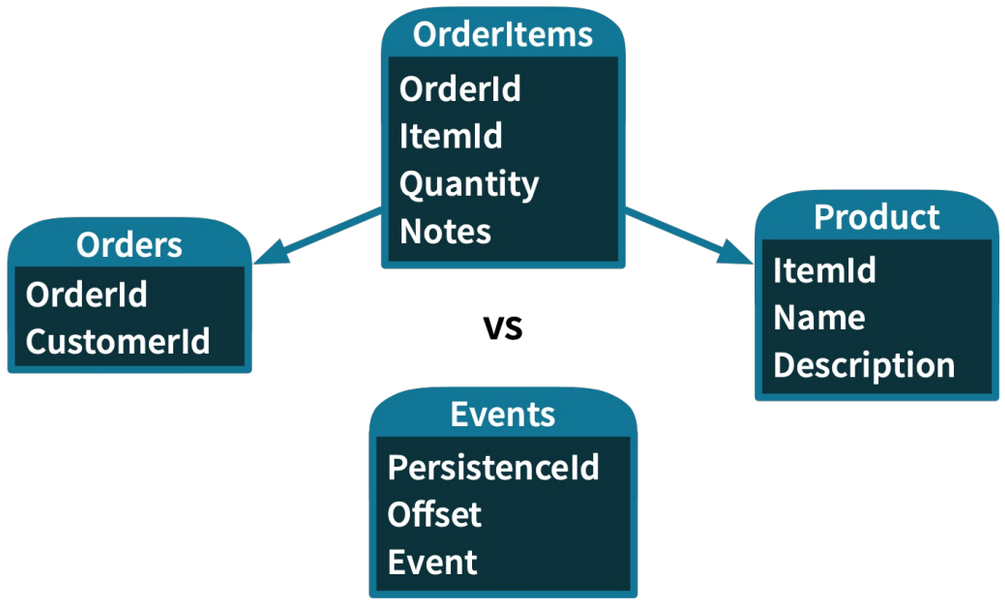
Schema Evolution



- *Events* are persisted using the Protobuf *Any* type.
- The name of the *Event* is persisted alongside the serialized data.
- If you later rename the *Event*, you will no longer be able to read it.
- Therefore, you must be careful about how you evolve your *Events*.
- Best Practice: Ensure your *Event* names don't change.
- You may wish to separate your *API* objects from your *Domain Events* so they can evolve independently.

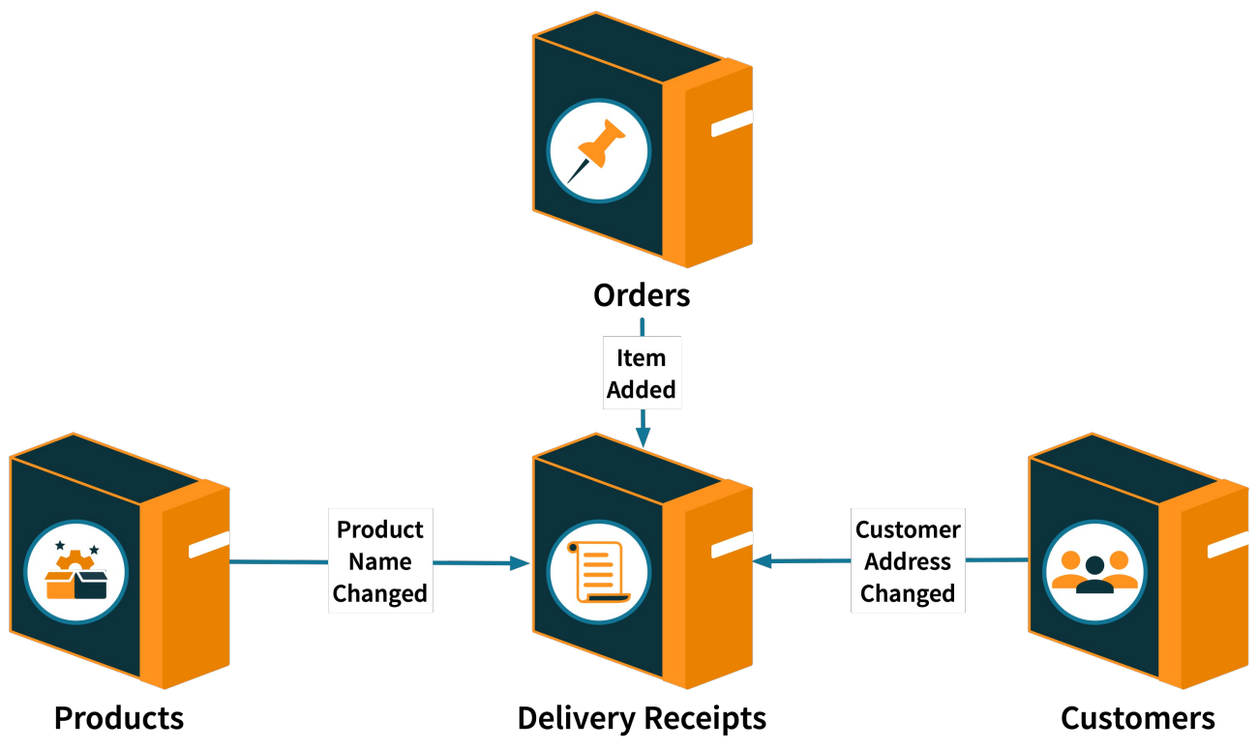
Benefits of Event Sourcing

Limited Database Complexity



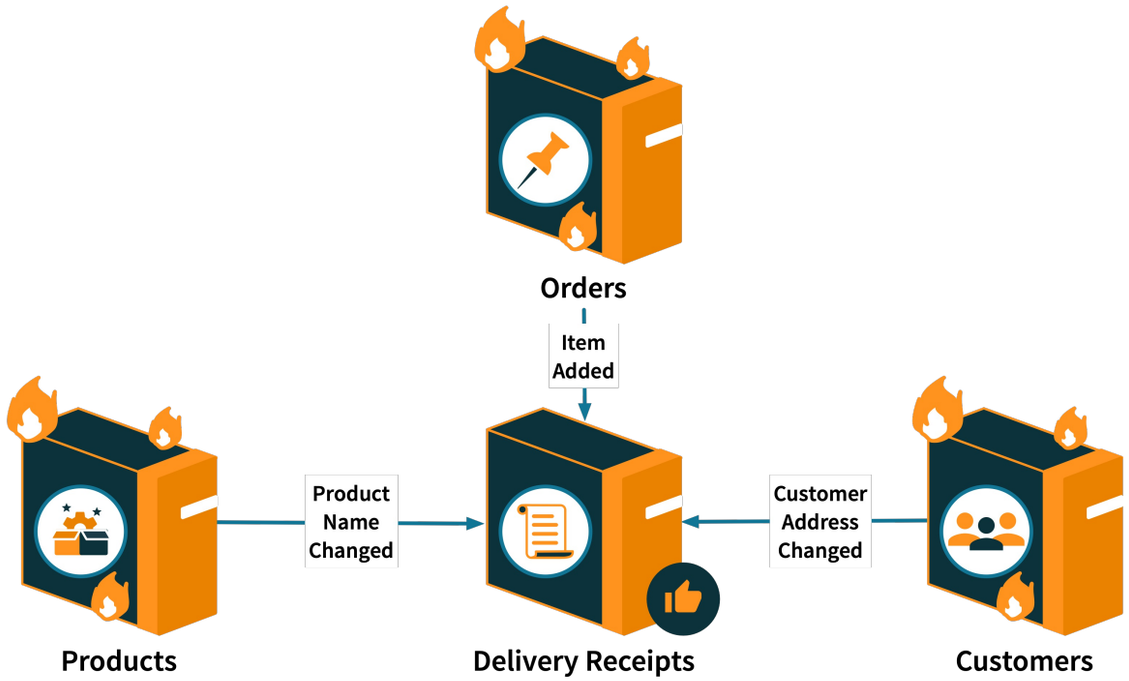
- With *Event Sourcing*, our database access is limited to just a few operations:
  - Persist or Recover *Events*
  - Persist or Recover *Snapshots*
- Simplicity allows us to abstract away most database concerns.
- The result is a clean, database-agnostic, persistence API.
- Developers focus on the domain, rather than the database.

Replicating State



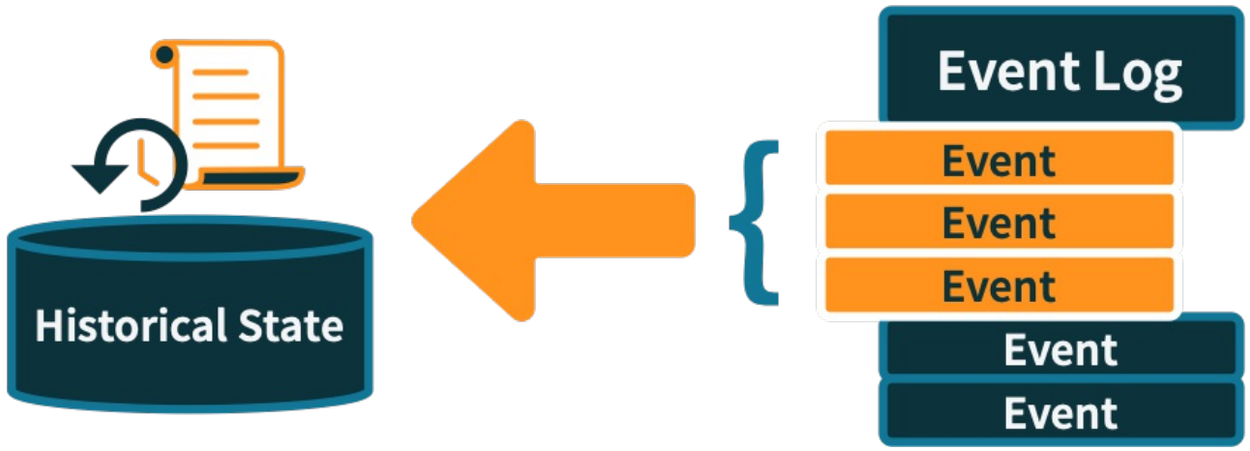
- *Event Sourcing* can be used to reliably replicate *State* to other services or views.
- The *Event* log can be replayed and sent to other parts of the system.
- These other parts of the system can rebuild a copy of the *State* optimized for their own use.
- They can have their own view of the data, picking and choosing *Events* that are relevant.

Isolation and Autonomy



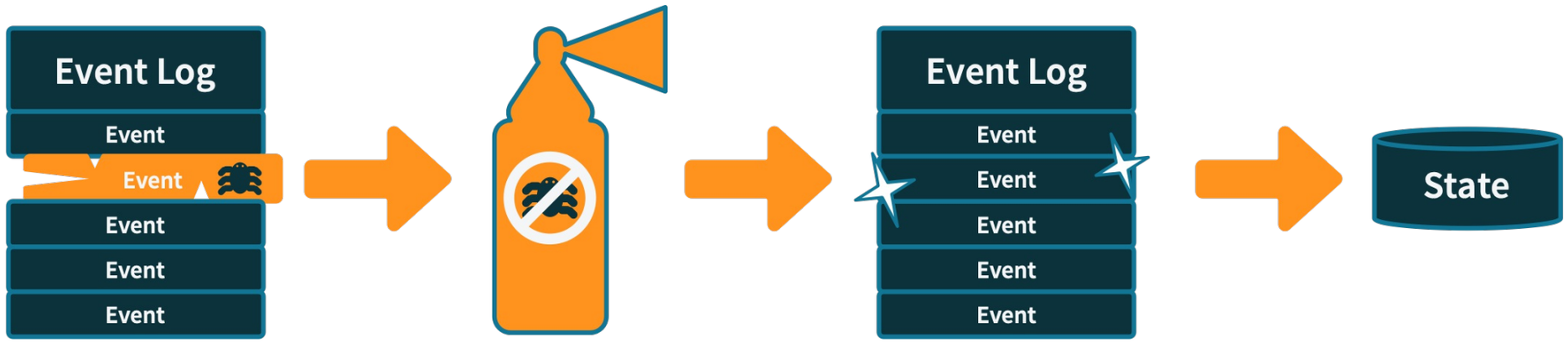
- Replicating our *State* to different parts of the system improves *Isolation* and *Autonomy*.
- Services can have their own copies of the data they require.
- They don't need to perform cross service queries to access that data.
- They can operate independently, even if other parts of the system fail.
- They are also more performant, since the data can be readily available in an optimized format.

Temporal Querying



- Rather than asking for all *Events*, we can instead ask for *Events* up to a certain date and time.
- This allows us to build historical views of our *State*.
- This can be useful for auditing, debugging, and even disaster recovery.

Fixing Incorrect State



- Despite our best efforts, errors can creep into our code.
- When this happens, our *State* can become corrupted.
- With *Event Sourcing*, we don't record the *State*.
- We record the *Events* that lead to the *State*.
- In many cases, we can fix the bug, replay the *Events*, and correct the corrupted *State*.

Implementing Event Sourcing

Implementing an Event as a Protobuf message

- .proto

```
package lightbend.example;
//Schema
message MyEvent {
    string some_property = 1;
}
```

- .js

```
const MyEventType = entity.lookupType('lightbend.example.MyEvent');
const myEvent = MyEventType.create({
  someProperty: 'some value',
});
```

- *Events* are implemented as *Profobuf* messages.
- In JavaScript, the *Event* is constructed using the Protobuf *message type*.

Emitting Events

```
entity.myCommandHandler = function (command, state, context) {
  ...
  const myEvent = MyEventType.create({
    someProperty: 'some value',
  });

  context.emit(myEvent);
  ...
  return response;
}
```

- Recall: a *Command Handler* is implemented as a function with three parameters: command, state, context.
- context.emit is how we register the *Event*. It will:
  - *Emit* the *Event* to be processed by an *Event Handler*.
  - *Persist* the *Event* in the *Datastore*.
- We can use the command and state objects to create the *Event*.

Implementing an Event Handler

```
entity.myEventHandler = function (event, state) {
  var newState = {
    ... //update state
  }
  return newState;
}
```

- An *Event Handler* is implemented as a function with two parameters:
  - event: The event to be handled, created previously in the command handler.
  - state: The state of the Entity at the time the *Event Handler* was invoked.
- The *State* is updated based on domain logic and properties of the *Event*.
- The updated *State* is returned by the *Event Handler*.

Defining a Behavior

```
entity.setBehavior(order => {
  return {
    commandHandlers: {
      MyCommand: entity.myCommandHandler,
    },
    eventHandlers: {
      MyEvent: entity.myEventHandler,
    },
  };
});
```

- Recall: A *Behavior* is defined with a group of *Command Handlers*.
- In addition to *Command Handlers* we can also add *Event Handlers*.
- *Event Handlers* associate an *Event* type with a *Handler* function.

Event Handlers vs Command Handlers

```
package lightbend.example;

message MyEvent { ... }

service MyService {
    rpc MyCommand(...) returns (...) {...}
}
```

- Command Handlers
  - Use the method name defined in the Protobuf.
    - Eg: MyCommand: entity.myCommandHandler
- Event Handlers
  - Use the name of the type of the *Event*.
    - Eg: MyEvent: entity.myEventHandler