Onyx

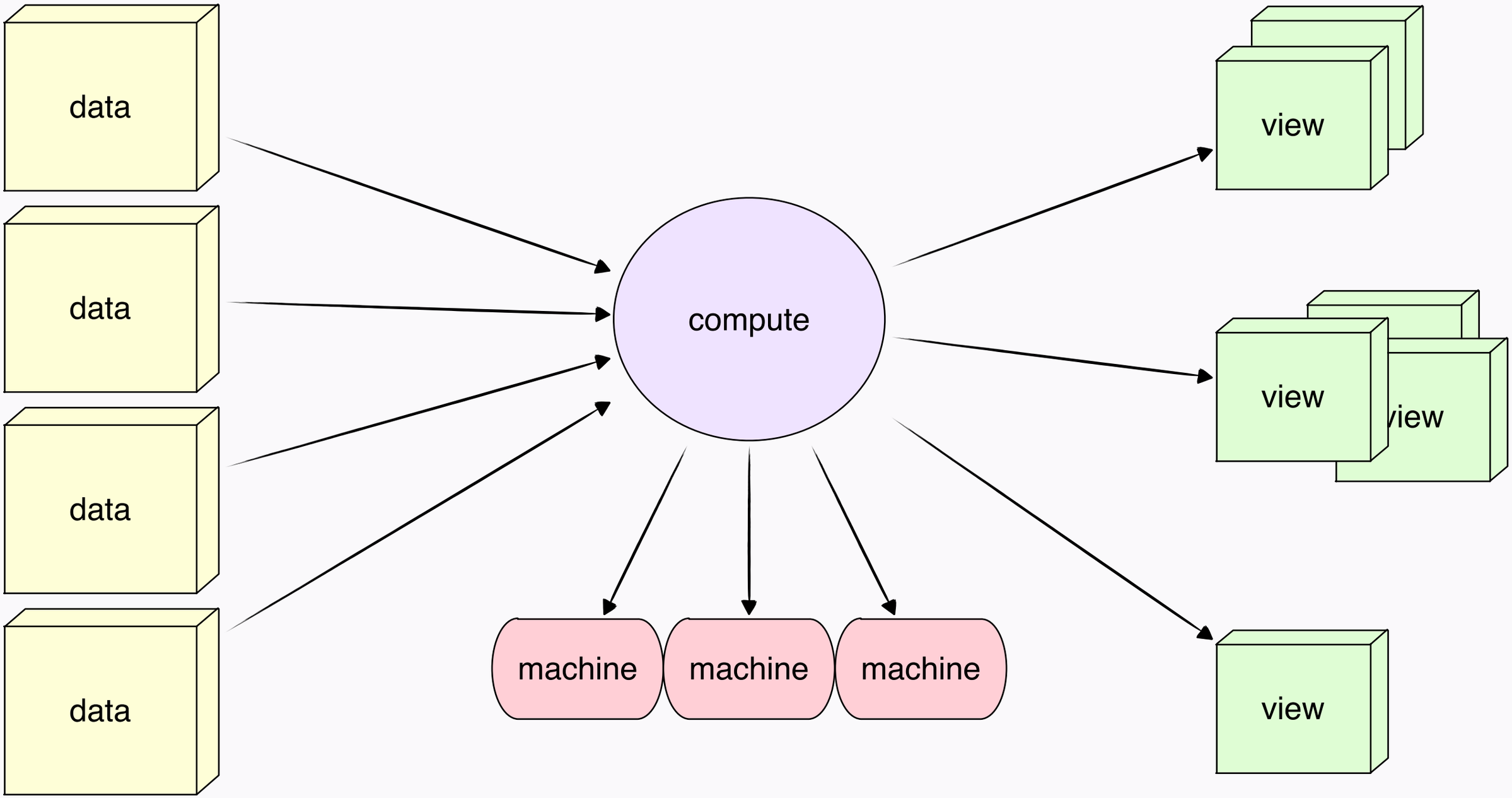
Distributed Computing for Clojure

@MichaelDrogalis / @OnyxPlatform

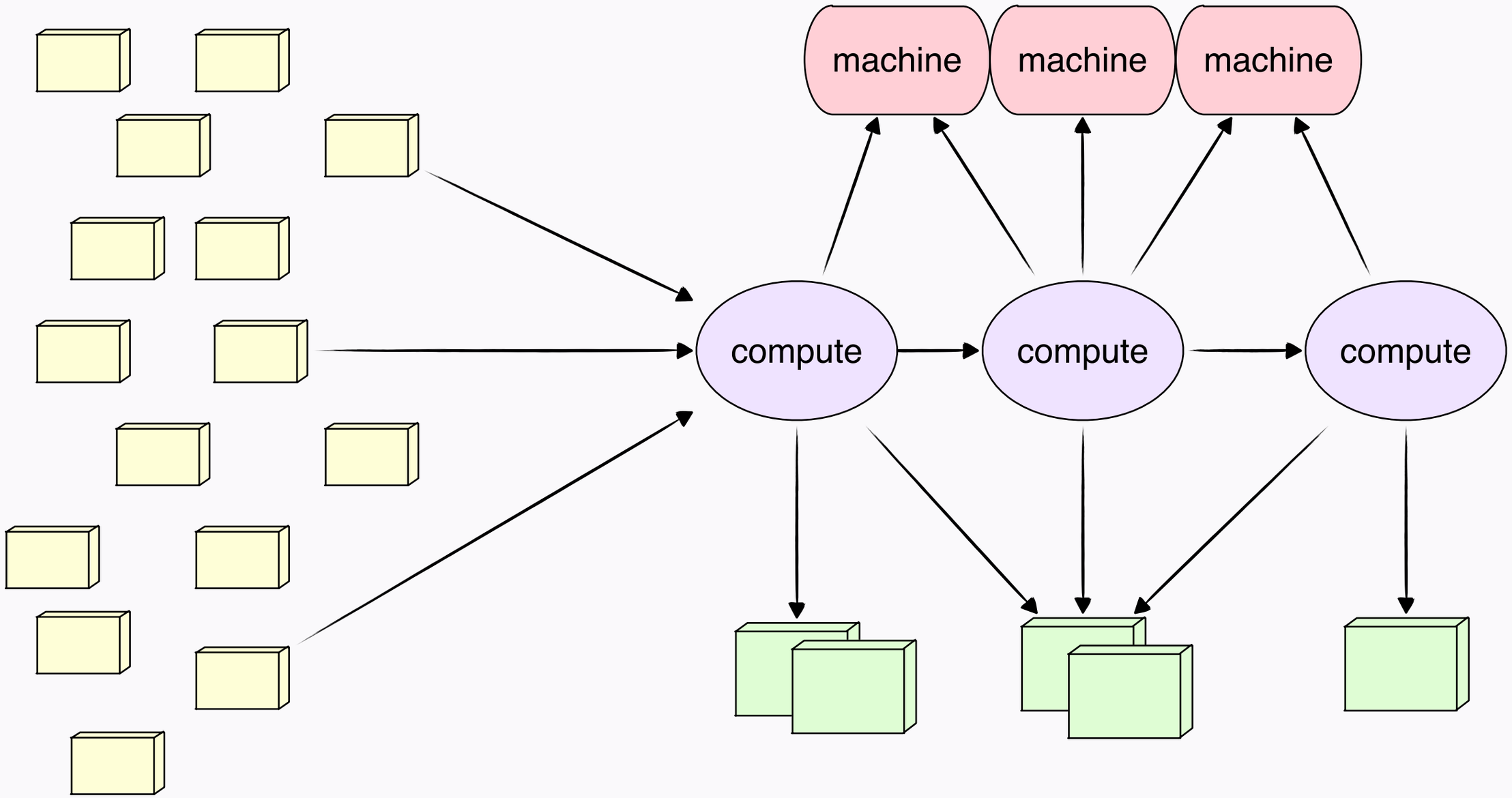
(hi)

- Creator of Onyx
- Developer at ViaSat
- Co-Founder of Distributed Masonry
- Years of commercial analytics development
- @MichaelDrogalis

Batch Workloads



Streaming Workloads



What is Onyx?

- Cloud-scale, distributed data processing platform
- Batch and streaming hybrid
- Masterless coordination
- 6,000 lines of pure Clojure
- Decouples behavioral set from specific execution***

Tigon

Esper

Samza

CEDR

Cascalog

Spark

SQLStream

Trill

Hadoop M/R

Flink

TelegraphCQ

Pulsar

GigaScope

MillWheel

Heron

Hive

Sonora

Pig

DataFlow

Storm

Aurora

Cascading

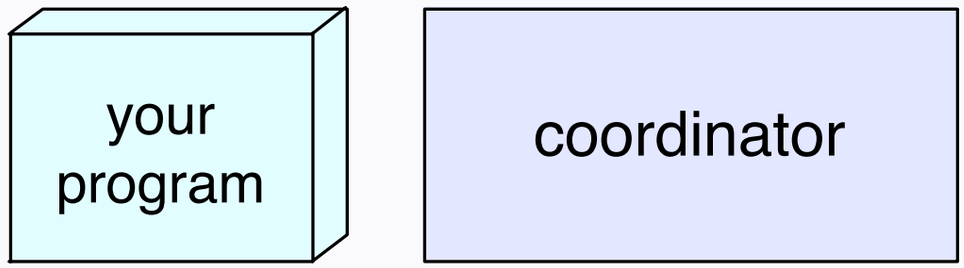
Niagara

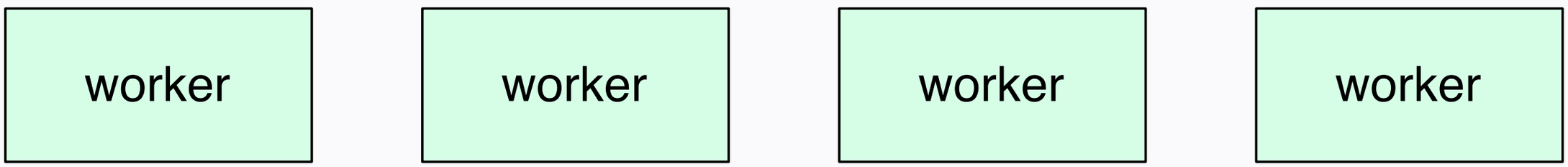
```clojure
(ns my.ns
  (:require [slingshot.slingshot :refer [try+]]
            [com.stuartsierra.component :as component]
            [backtype.storm.clojure :as storm]))

(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

coordinator

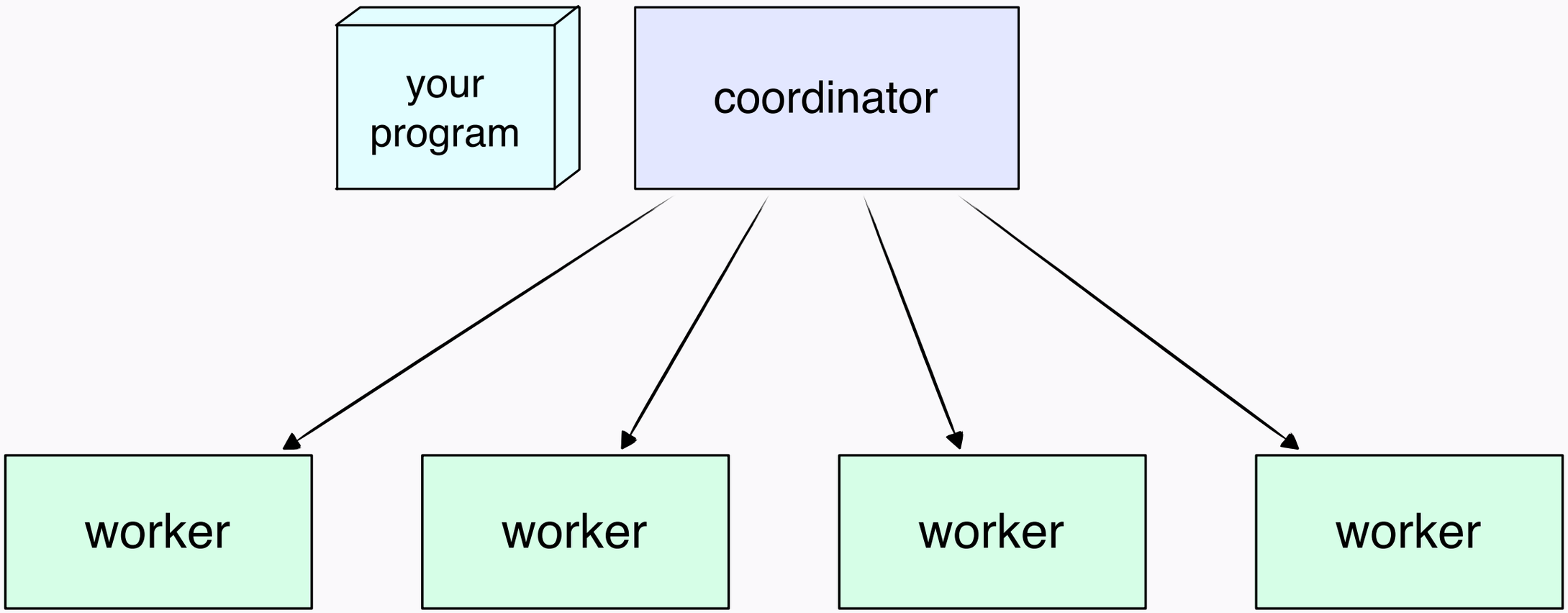
worker worker worker worker

```
storm jar my-jar.jar my.ns
```

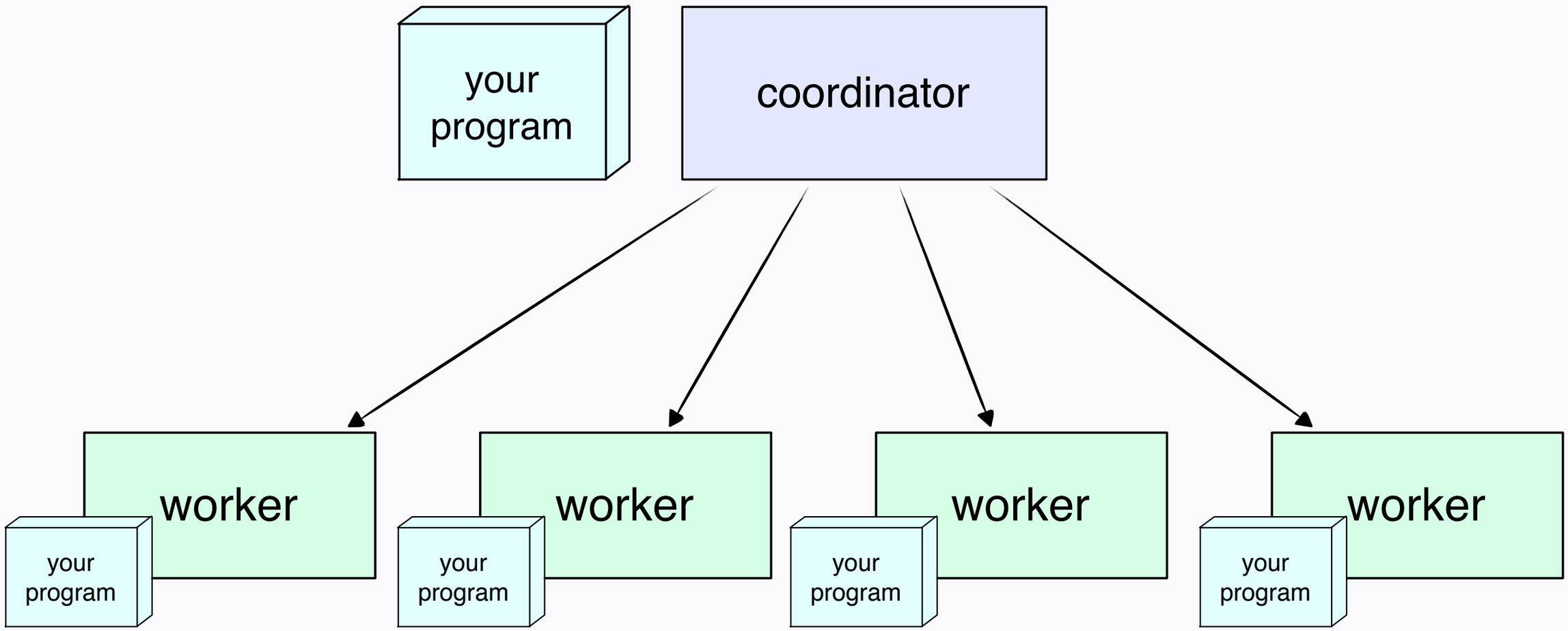





```
storm jar my-jar.jar my.ns
```




```
storm jar my-jar.jar my.ns
```




```
storm jar my-jar.jar my.ns
```

collection of behaviors

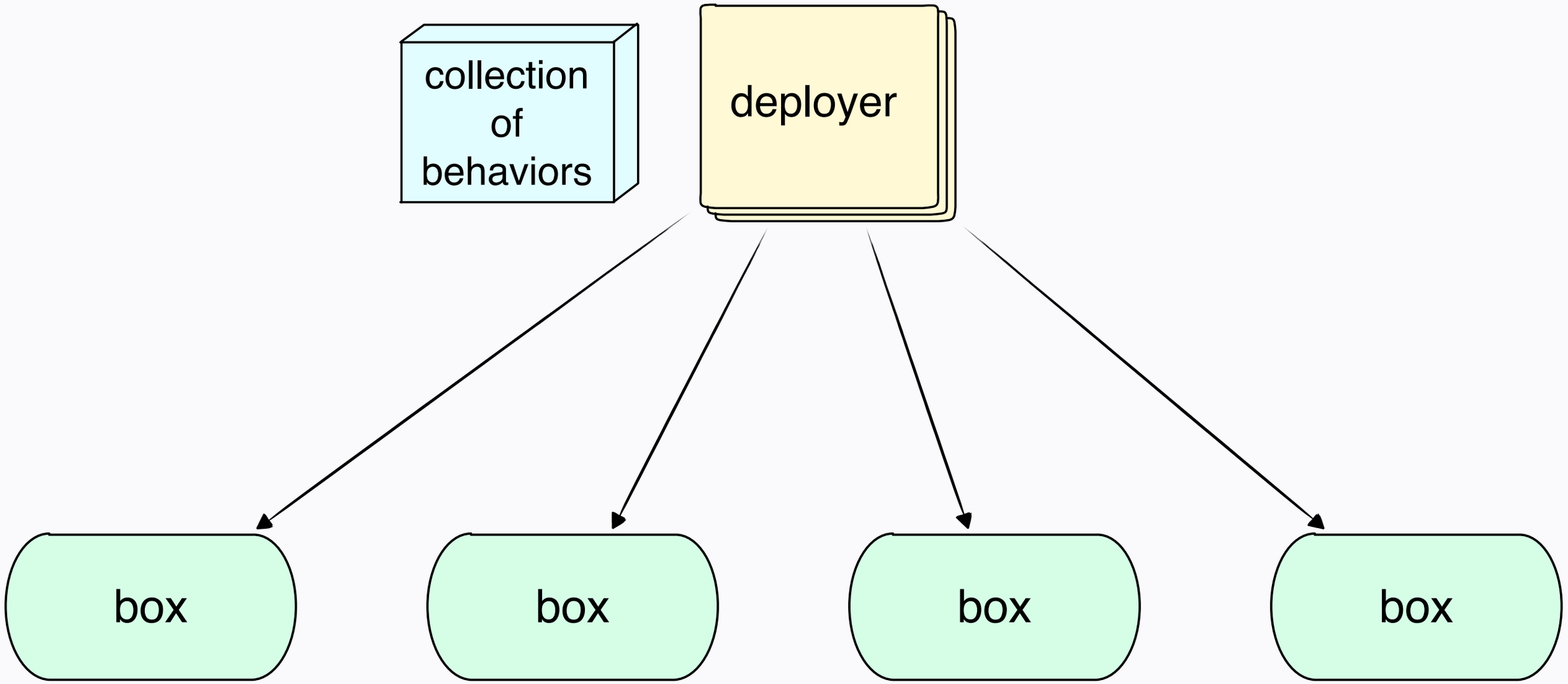
deployer

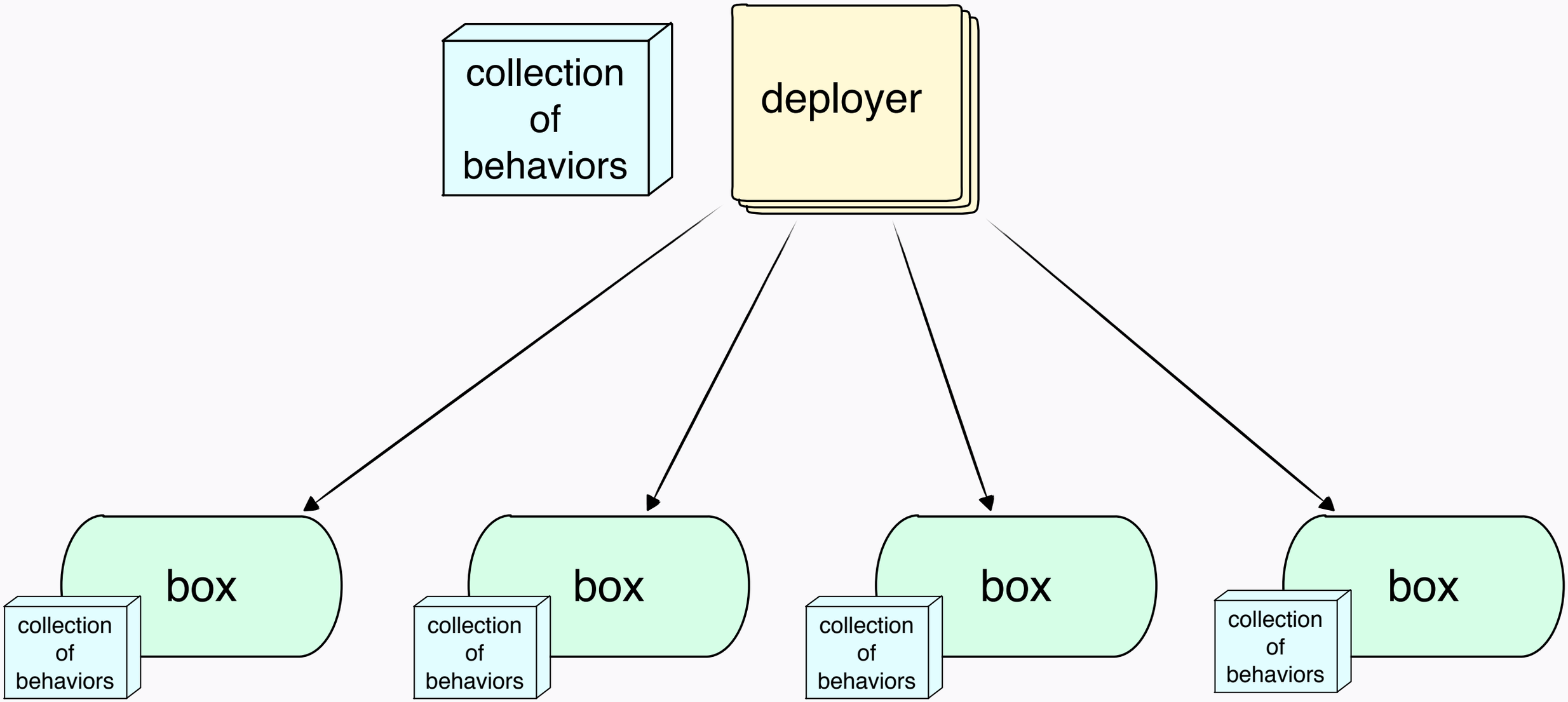
box

box

box

box


collection
of
behaviors
deployer
box
box
box
box


collection
of
behaviors
deployer
box
collection
of
behaviors
box
collection
of
behaviors
box
collection
of
behaviors
box
collection
of
behaviors

collection of behaviors

deployer

box

collection of behaviors

box

collection of behaviors

box

collection of behaviors

box

collection of behaviors

execution spec

collection of behaviors

deployer

box

collection of behaviors

box

collection of behaviors

box

collection of behaviors

box

collection of behaviors

execution spec

collection of behaviors

deployer

execution spec

box

collection of behaviors

execution spec

box

collection of behaviors

execution spec

box

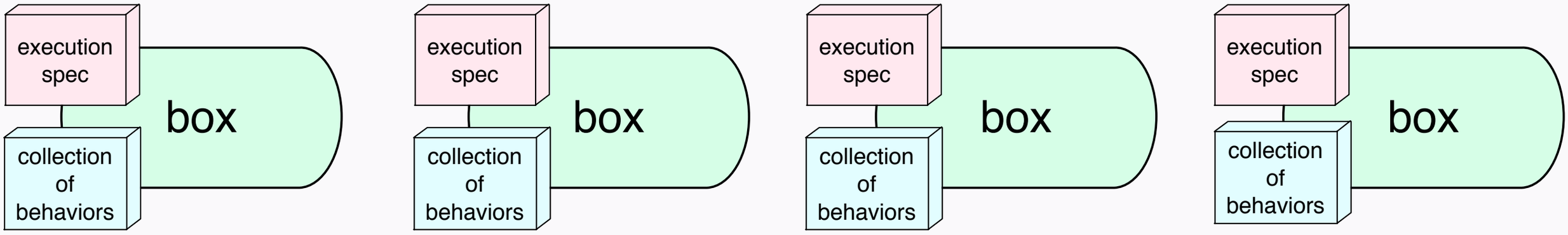
collection of behaviors

execution spec

box

collection of behaviors

execution spec


execution spec
box
collection of behaviors
execution spec
box
collection of behaviors
execution spec
box
collection of behaviors
execution spec
box
collection of behaviors

DYNAMISM

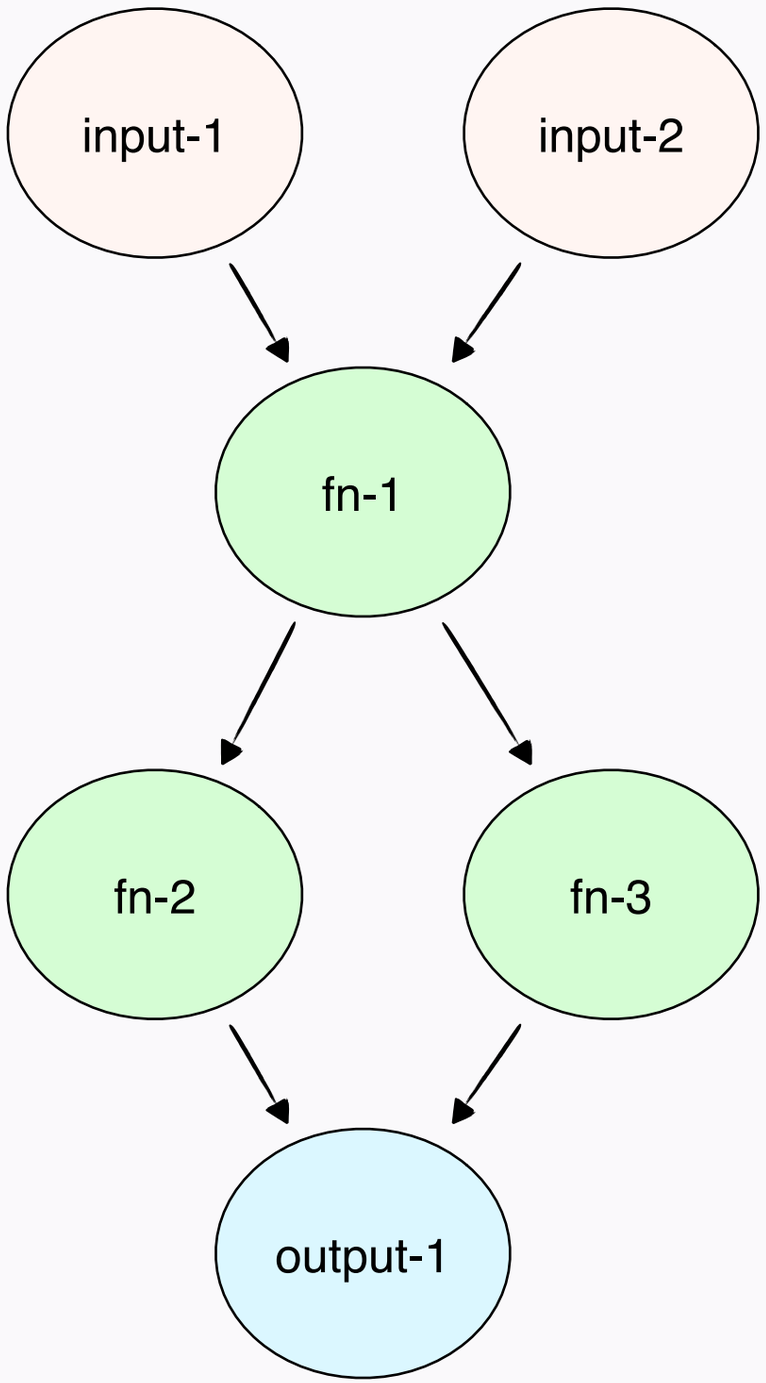
NECESSITATES

DISTANCE

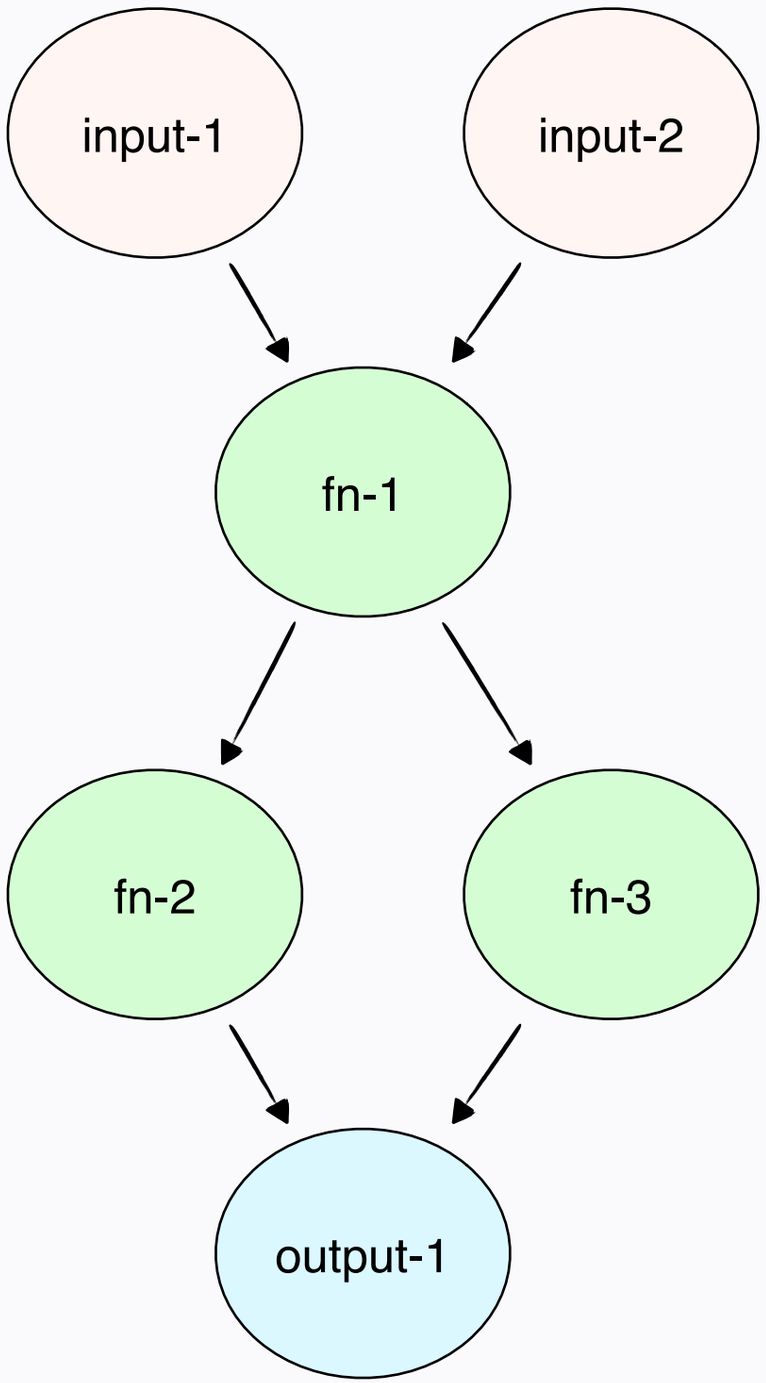
Onyx's API

- Structure
- Functional context
- Side Effects
- Flow
- Process

API: Structure

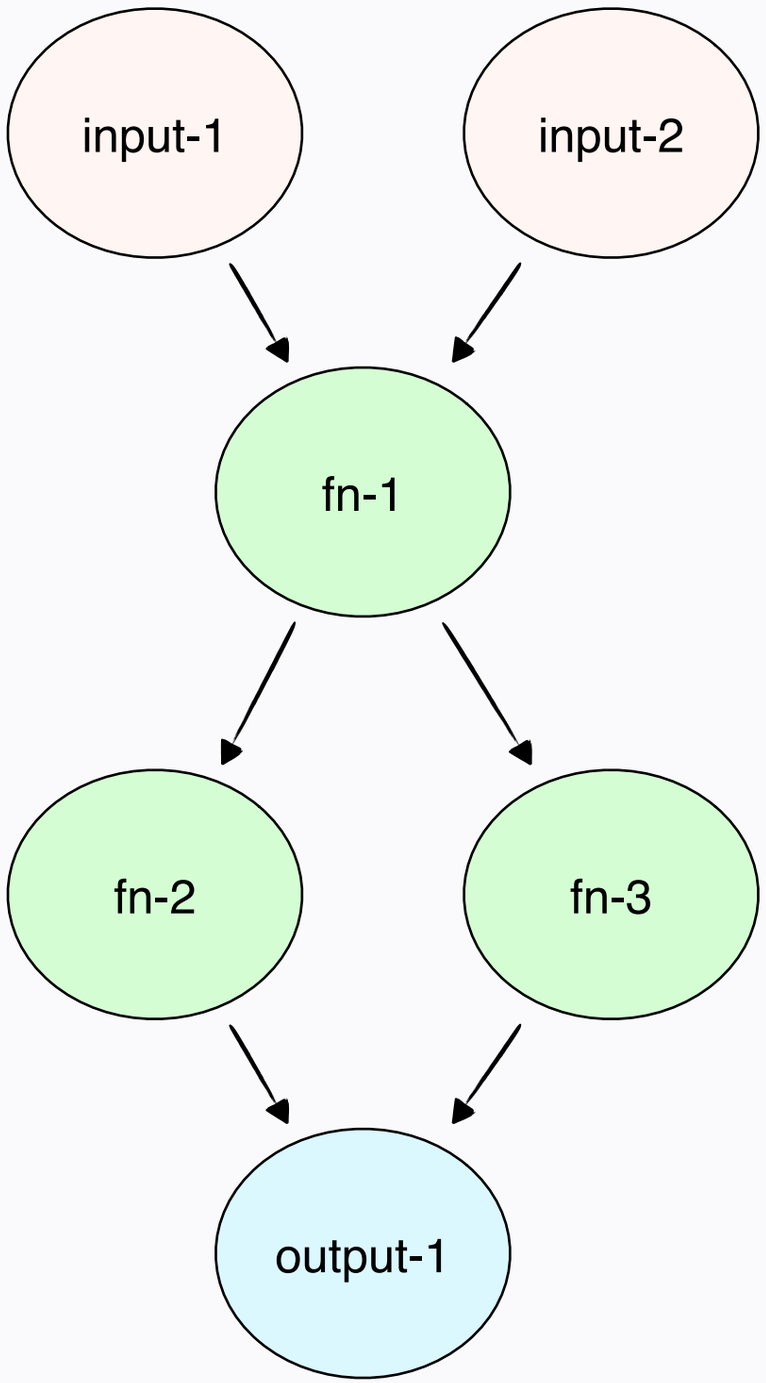


API: Structure

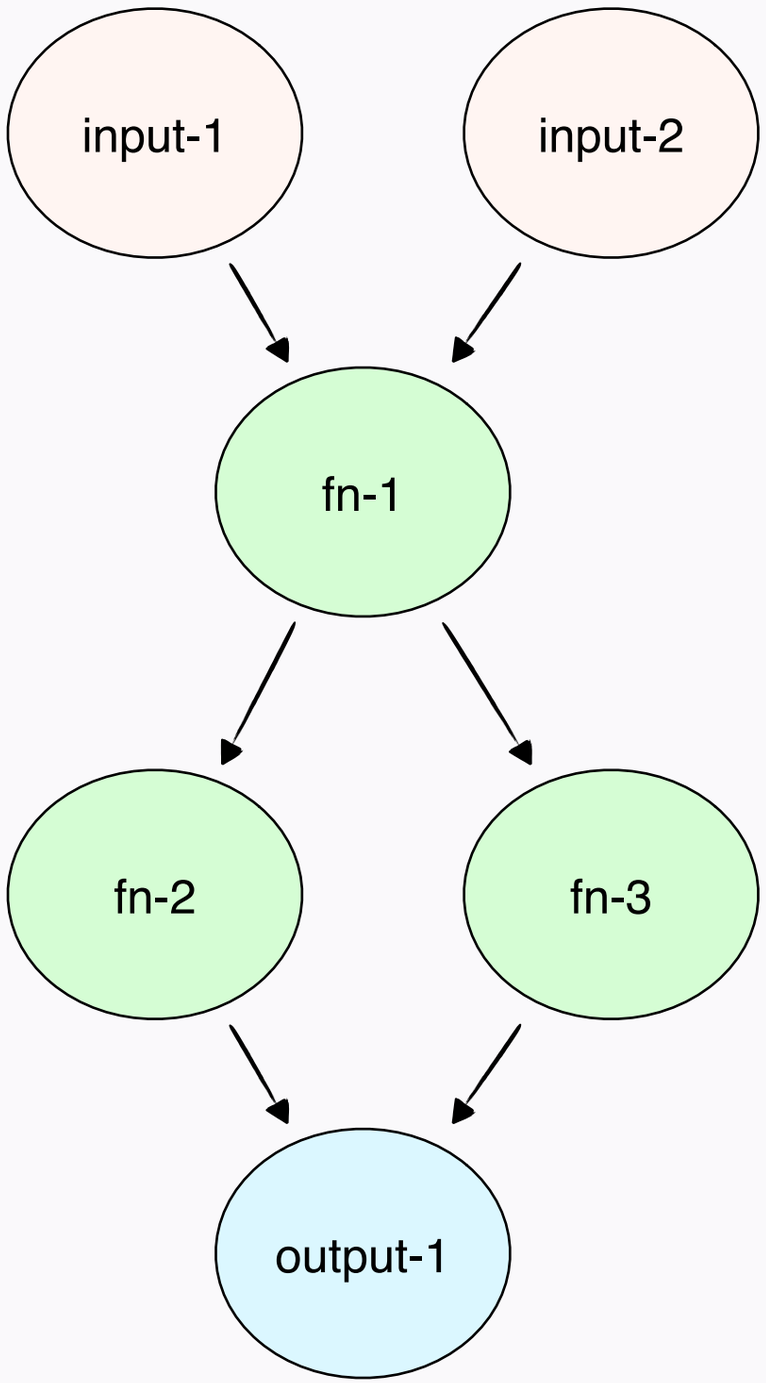



```
val textFile = spark.textFile("hdfs://...")
val errors = textFile.filter(line => line.contains("ERROR"))
// Count all the errors
errors.count()
// Count errors mentioning MySQL
errors.filter(line => line.contains("MySQL")).count()
// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()
```

API: Structure

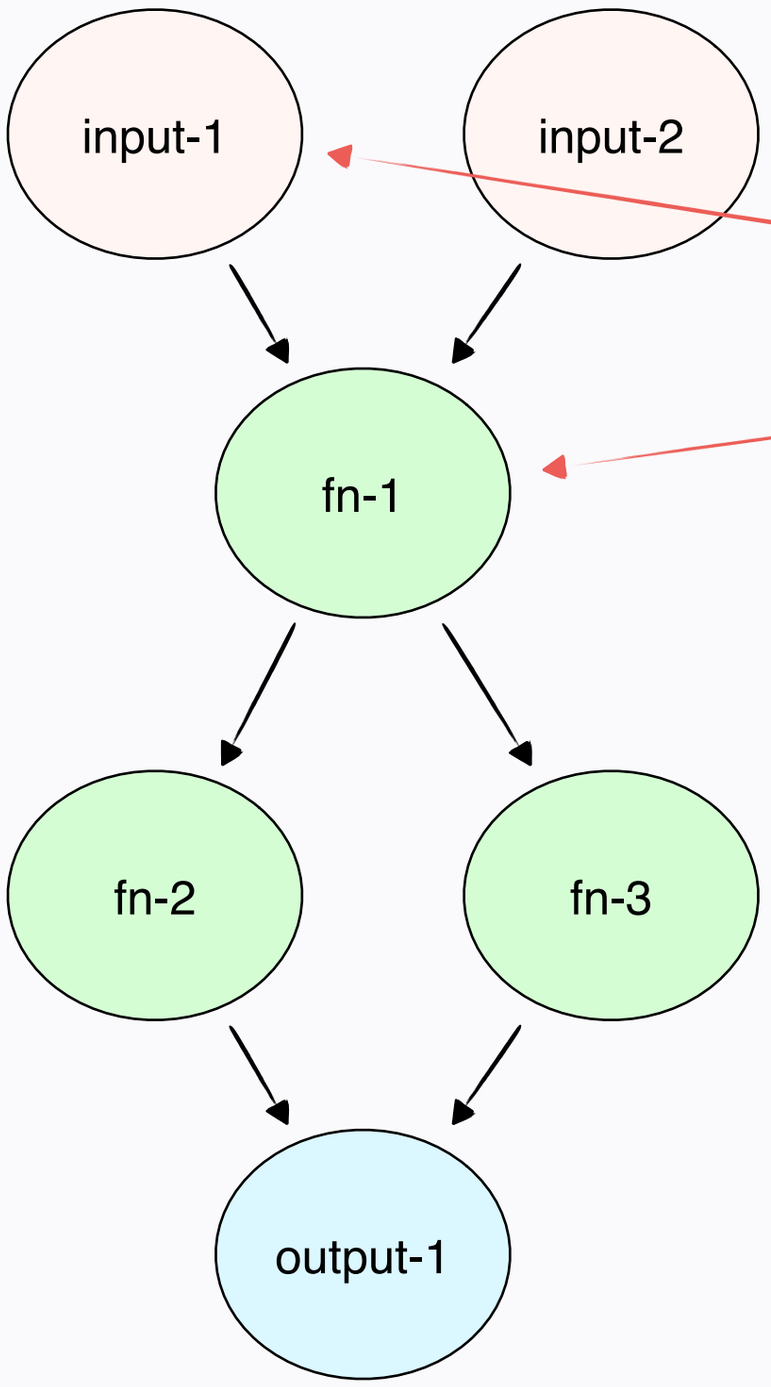


API: Structure




```
[[:input-1 :fn-1]
 [:input-2 :fn-1]
 [:fn-1 :fn-2]
 [:fn-1 :fn-3]
 [:fn-2 :output-1]
 [:fn-3 :output-1]]
```

API: Structure




```
[[:input-1 :fn-1]
 [:input-2 :fn-1]
 [:fn-1 :fn-2]
 [:fn-1 :fn-3]
 [:fn-2 :output-1]
 [:fn-3 :output-1]]
```

API: Functional Context

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Functional Context

```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

```clojure
(defn process-tweet [conn prefix {:keys [id tweet]}]
  (db/write-tweet conn (str prefix "-" id))
  {:id id :success? true})
```

```clojure
(defn process-tweet [conn prefix {:keys [id tweet]}]
  (db/write-tweet conn (str prefix "-" id))
  {:id id :success? true})
```

```clojure
{:onyx/name :process-tweet
 :onyx/fn :my.ns/process-tweet
 :tweet/prefix "tweet-preprocessor"
 :onyx/params [:tweet/prefix]
 :onyx/type :function
 :onyx/batch-size batch-size
 :onyx/doc "Writes the tweet ID to the database"}
```

```clojure
(defn process-tweet [conn prefix {:keys [id tweet]}]
  (db/write-tweet conn (str prefix "-" id))
  {:id id :success? true})
```

```clojure
{:onyx/name :process-tweet
 :onyx/fn :my.ns/process-tweet
 :tweet/prefix "tweet-preprocessor"
 :onyx/params [:tweet/prefix]
 :onyx/type :function
 :onyx/batch-size batch-size
 :onyx/doc "Writes the tweet ID to the database"}
```

```
[{:onyx/name :read-log
  :onyx/plugin :onyx.plugin.seq/input
  :onyx/fn :my.ns/unwrap-elements
  :seq/elements-per-segment 1
  :onyx/type :input
  :onyx/medium :seq
  :onyx/batch-size 10
  :onyx/max-peers 1
  :onyx/doc "Reads log data from line-seq"}

 {:onyx/name :write-parse-failures
  :onyx/plugin :my.ns/output
  :onyx/type :output
  :onyx/medium :null
  :onyx/batch-size 10
  :onyx/doc "Writes parse failures for future re-consumption"}

{:onyx/name :write-to-datomic
 :onyx/plugin :onyx.plugin.datomic/write-datoms
 :onyx/type :output
 :onyx/medium :datomic
 :onyx/fn :my.ns/entry->transaction
 :datomic/uri db-uri
 :datomic/partition :db.part/my-part
 :onyx/batch-size batch-size
 :onyx/doc "Transacts segments to storage"}]
```

structure **functional context** side effects flow process

```
[{:onyx/name :read-log
  :onyx/plugin :onyx.plugin.seq/input
  :onyx/fn :my.ns/unwrap-elements
  :seq/elements-per-segment 1
  :onyx/type :input
  :onyx/medium :seq
  :onyx/batch-size 10
  :onyx/max-peers 1
  :onyx/doc "Reads log data from line-seq"}

 {:onyx/name :write-parse-failures
  :onyx/plugin :my.ns/output
  :onyx/type :output
  :onyx/medium :null
  :onyx/batch-size 10
  :onyx/doc "Writes parse failures for future re-consumption"}

 {:onyx/name :write-to-datomic
  :onyx/plugin :onyx.plugin.datomic/write-datoms
  :onyx/type :output
  :onyx/medium :datomic
  :onyx/fn :my.ns/entry->transaction
  :datomic/uri db-uri
  :datomic/partition :db.part/my-part
  :onyx/batch-size batch-size
  :onyx/doc "Transacts segments to storage"}]
```

API: Side Effects

API: Side Effects

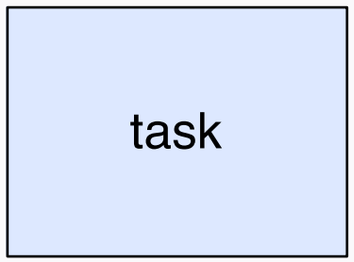
```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

API: Side Effects

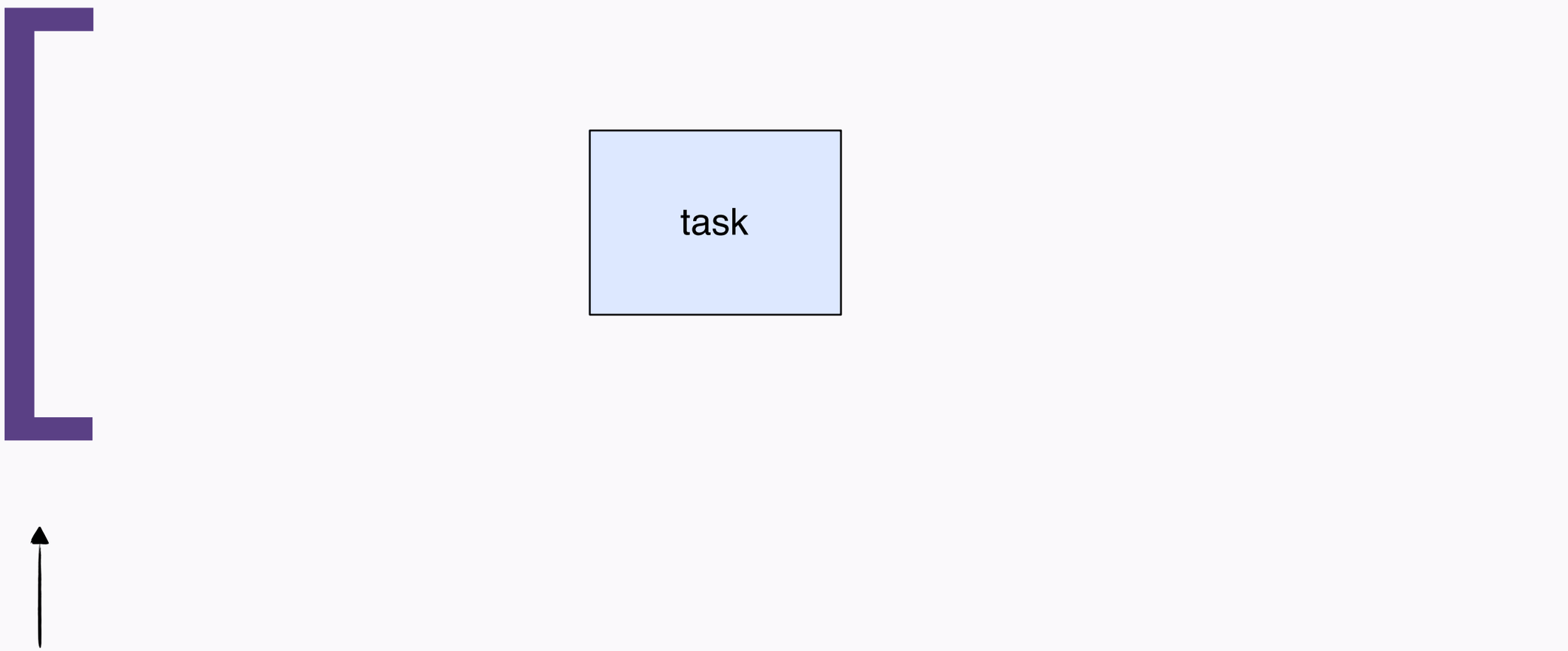
```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```

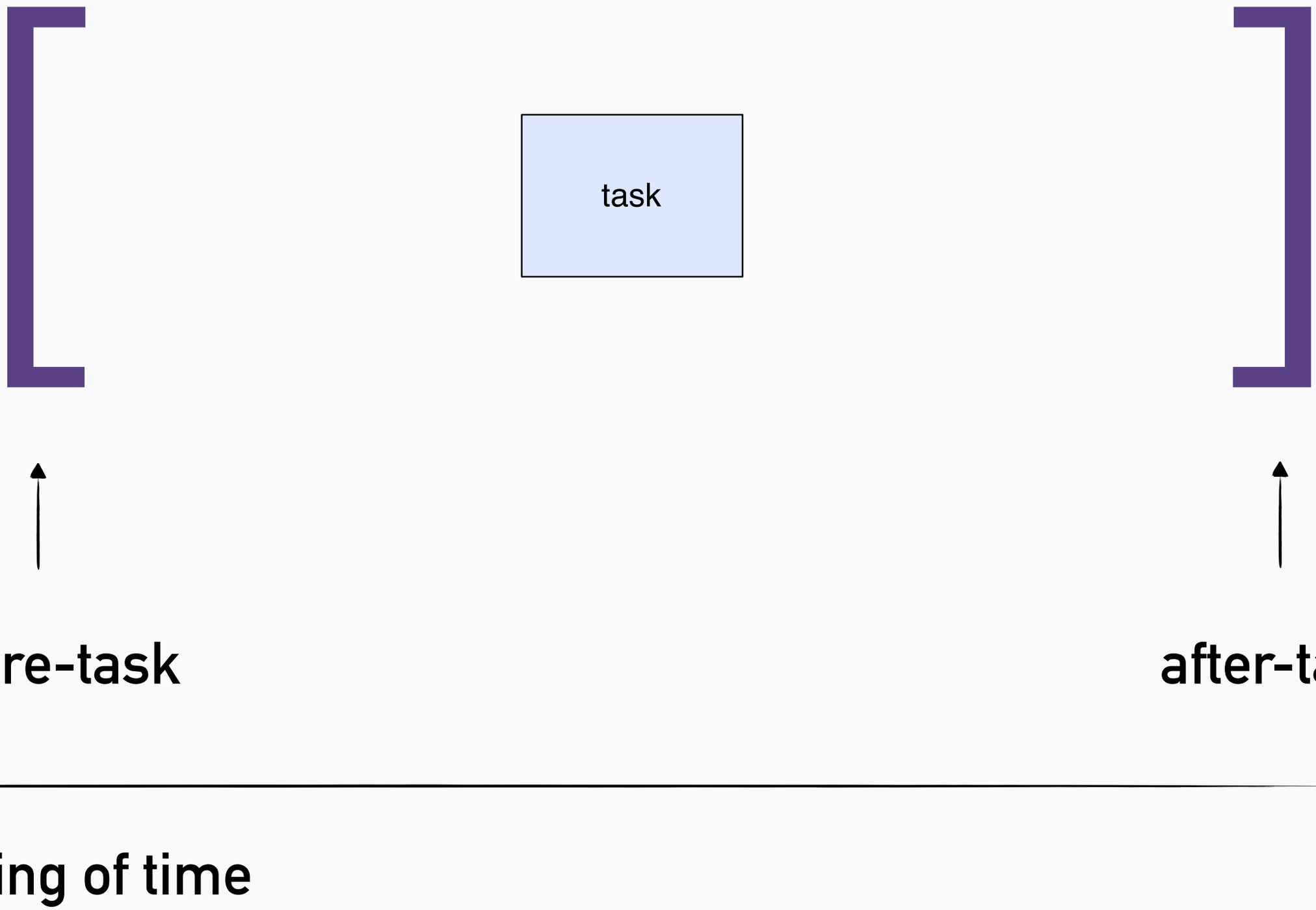
API: Side Effects

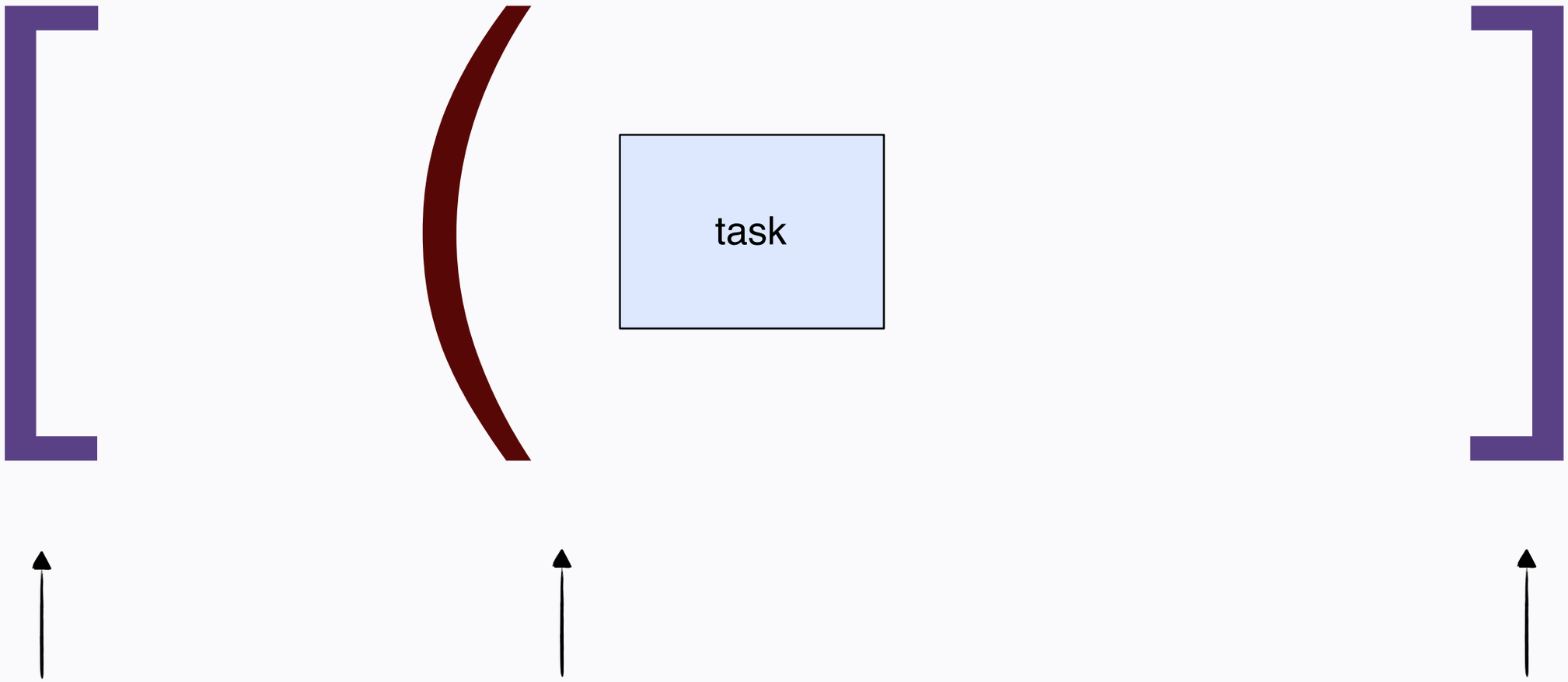
```
(storm/defbolt
  process-tweet ["user" "tweet"] {:prepare true}
  [conf context collector]
  (let [conn (-> (boot-db-connection) :db-subsystem :conn)]
    (storm/bolt
     (storm/execute
      [tuple]
      (try+
       (let [id (.getValueByField tuple "id")]
         (log-message (str "Processing " id))
         (db/write-tweet conn id)
         (storm/ack! collector tuple))
       (catch Throwable e
         (log-error e)
         (storm/fail! collector tuple)))))))
```



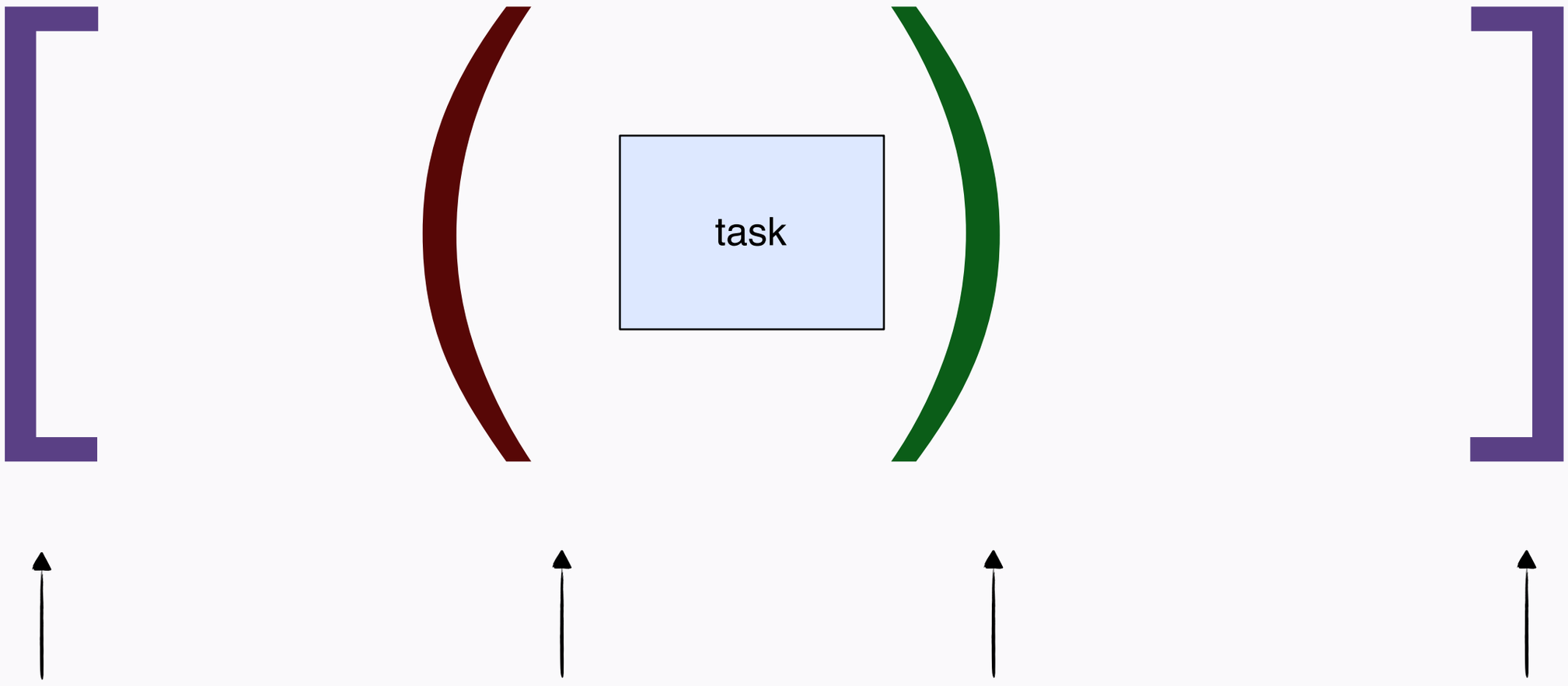
beginning of time


task
before-task
beginning of time

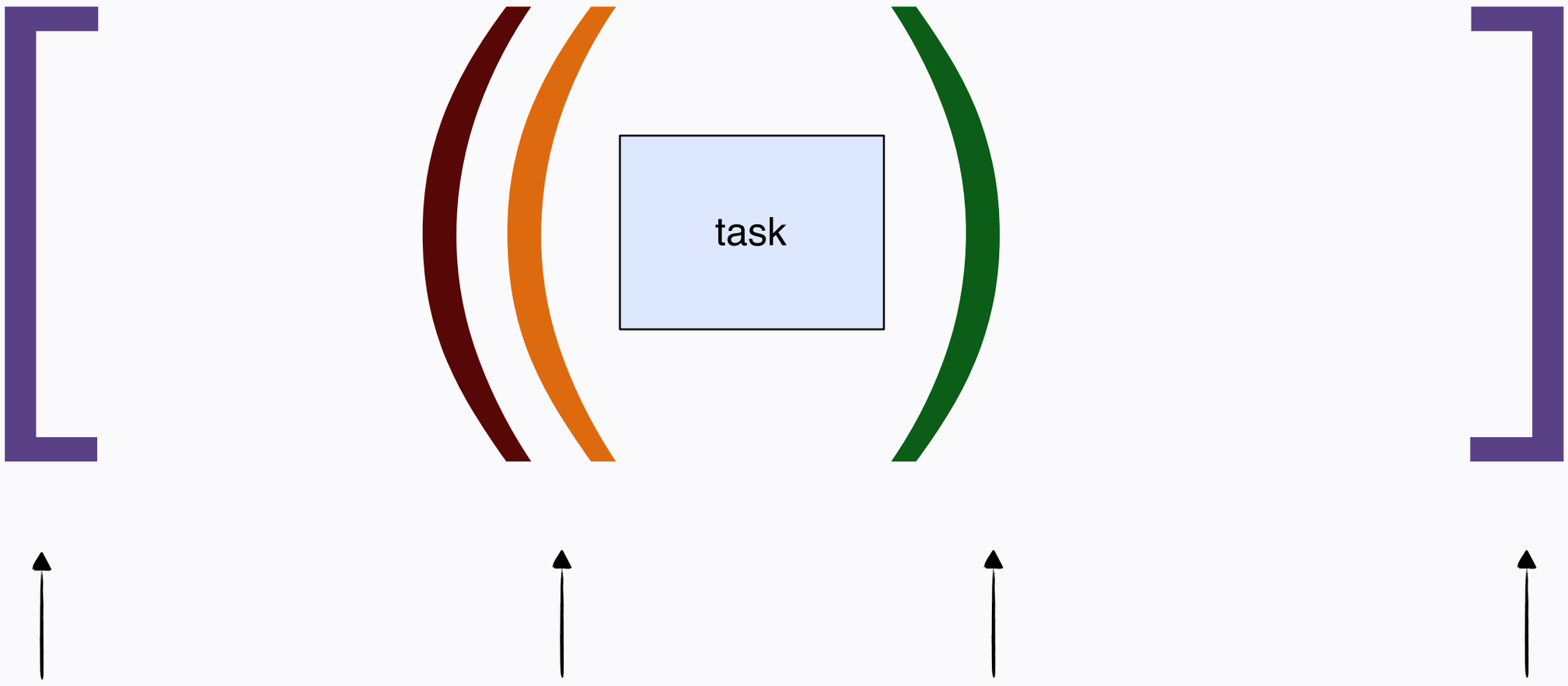

task
before-task
after-task
beginning of time



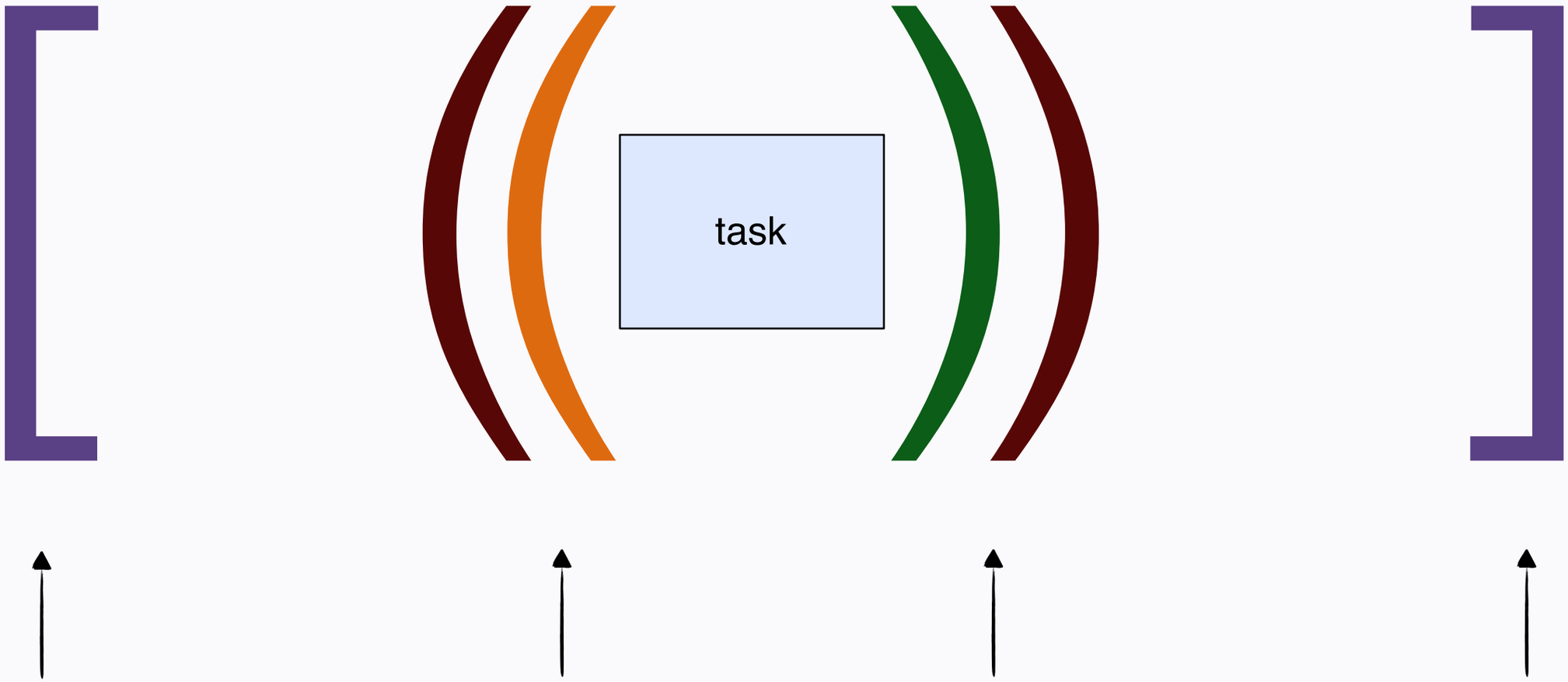
beginning of time

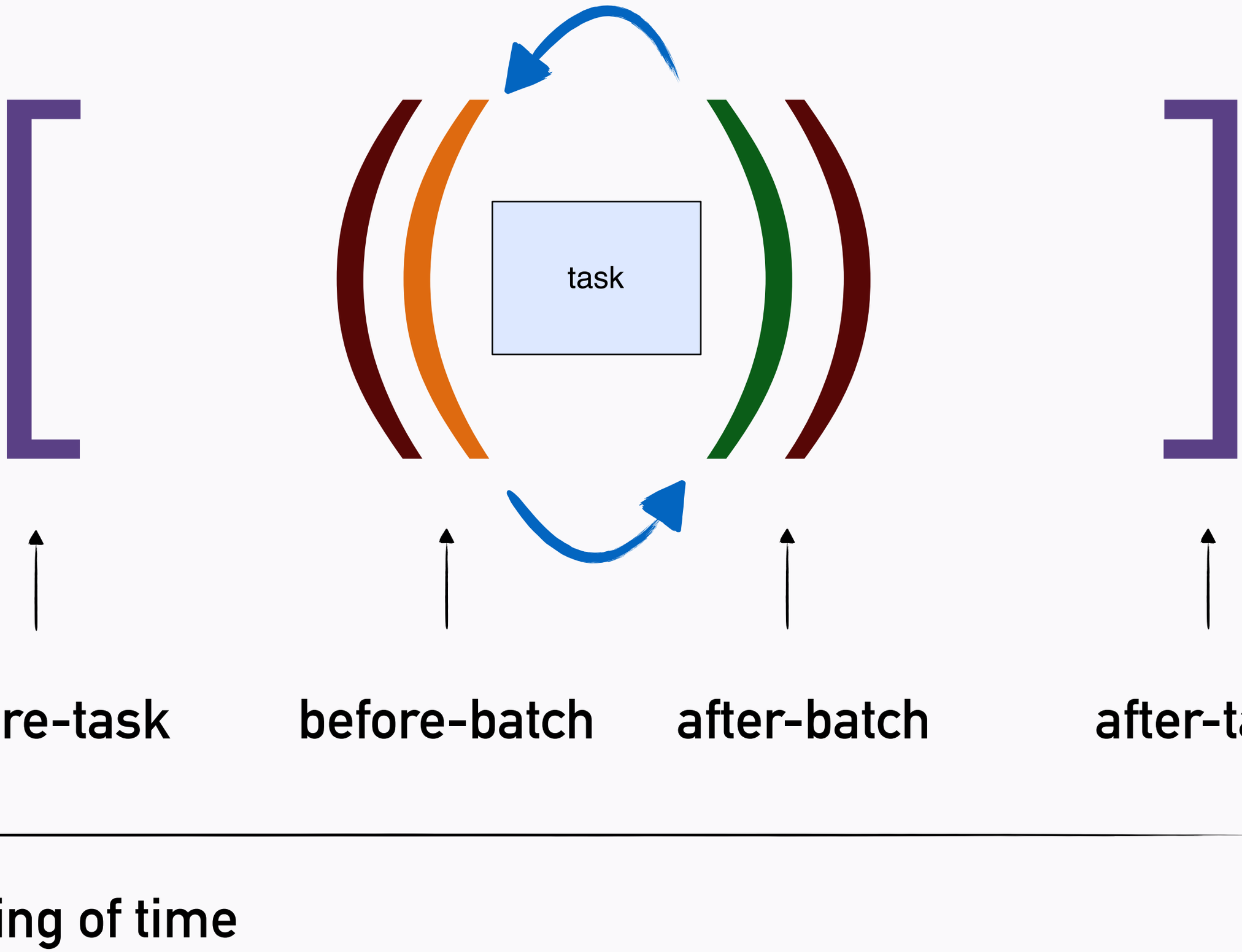


beginning of time

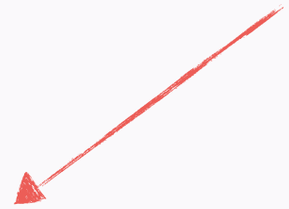


beginning of time


task
before-task
before-batch
after-batch
after-task
beginning of time


task
before-task
before-batch
after-batch
after-task
beginning of time

```
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))

(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```



```
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```clojure
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```clojure
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))

(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```clojure
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(def lifecycles
  [{:lifecycle/task :my-task
    :lifecycle/calls :my.ns/inject-hooks
    :lifecycle/doc "Side-effects for :my-task"}])
```

```clojure
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```clojure
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(def lifecycles
  [{:lifecycle/task :my-task
    :lifecycle/calls :my.ns/inject-hooks
    :lifecycle/doc "Side-effects for :my-task"}])
```

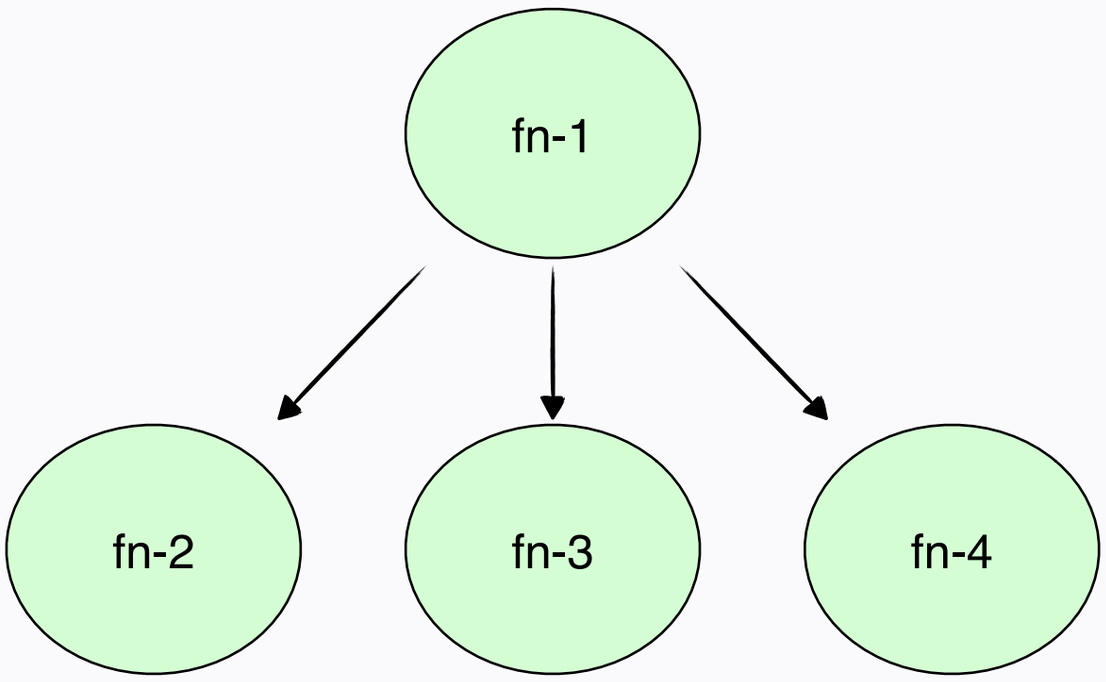
```clojure
(defn log-batch [event lifecycle]
  (doseq [m (:onyx.core/results event)]
    (info "Logging segment: " m))
  {})

(defn inject-conn [event lifecycle]
  (let [task-map (:onyx.core/task-map event)
        conn (d/connect (:datomic/uri task-map))]
    {:datomic/conn conn}))
```

```clojure
(def inject-hooks
  {:lifecycle/before-task-start inject-conn
   :lifecycle/after-batch log-batch})
```

```clojure
(def lifecycles
  [{:lifecycle/task :my-task
    :lifecycle/calls :my.ns/inject-hooks
    :lifecycle/doc "Side-effects for :my-task"}])
```

API: Flow

API: Flow



API: Flow

API: Flow

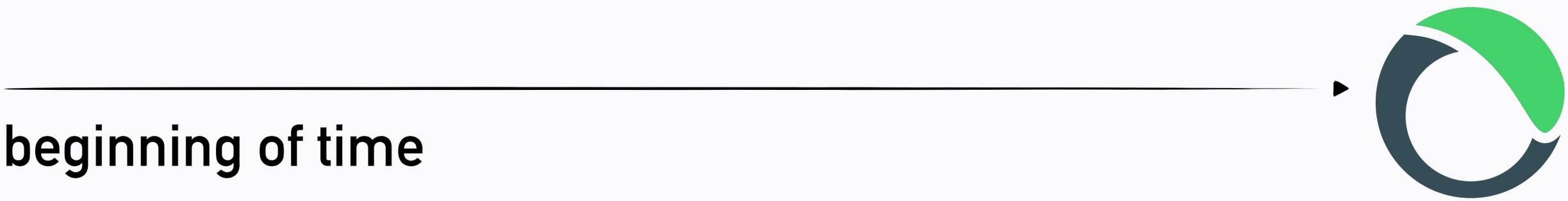
```
[{:flow/from :fn-1
  :flow/to :all
  :flow/short-circuit? true
  :flow/predicate :my.ns/pred-test-all?}

 {:flow/from :fn-1
  :flow/to [:fn-2 :fn-4]
  :flow/predicate :my.ns/pred-test-split?}

{:flow/from :all
 :flow/to [:fn-3]
 :flow/short-circuit? true
 :flow/predicate :my.ns/pred-test-ingress?}]
```

API: Flow

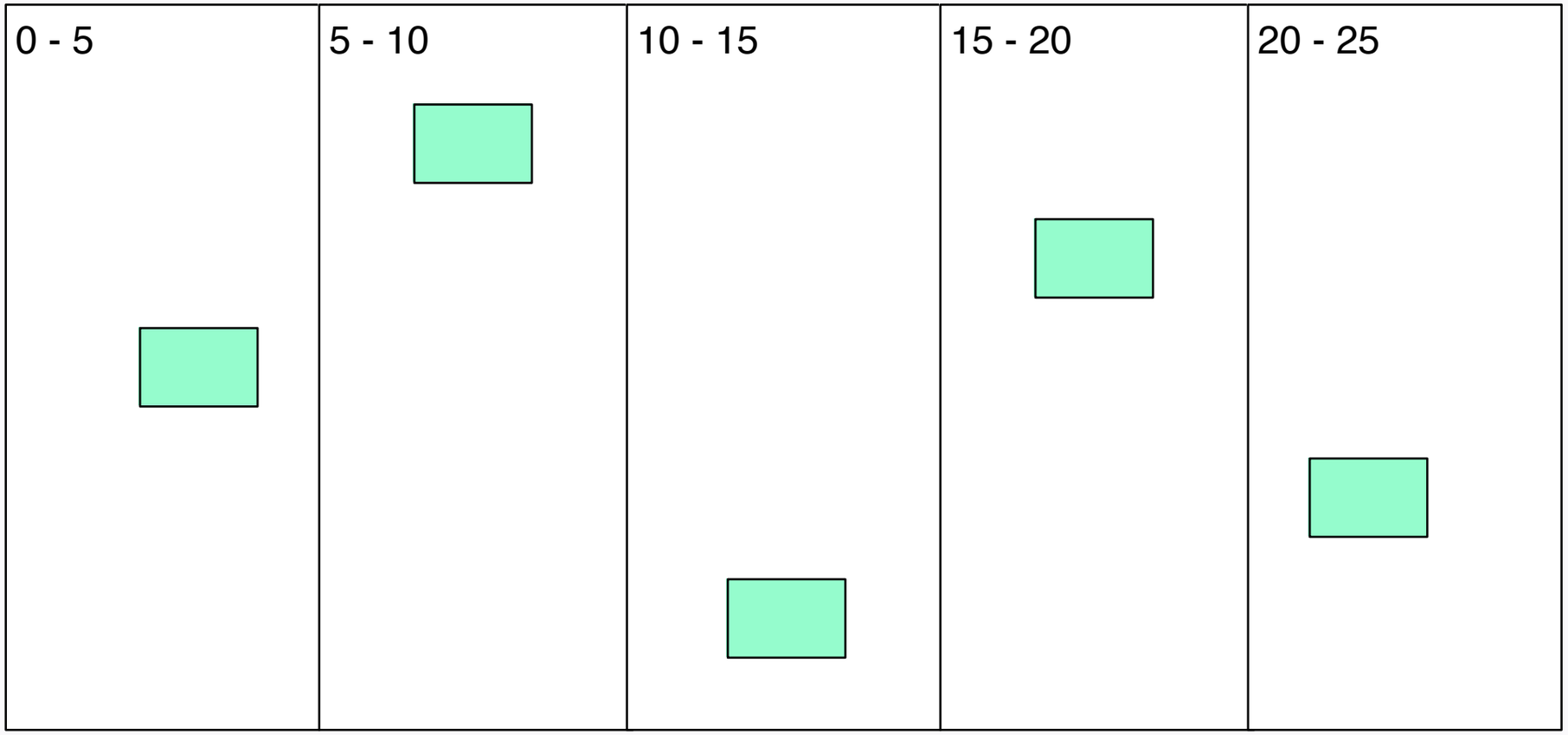
```
[{:flow/from :fn-1
  :flow/to :all
  :flow/short-circuit? true
  :flow/predicate :my.ns/pred-test-all?}

 {:flow/from :fn-1
  :flow/to [:fn-2 :fn-4]
  :flow/predicate :my.ns/pred-test-split?}

 {:flow/from :all
  :flow/to [:fn-3]
  :flow/short-circuit? true
  :flow/predicate :my.ns/pred-test-ingress?}]
```

```
(defn pred-test-all? [event old new all-new]
  (>= (:some-attribute new) (:some-attribute old)))
```

API: Process

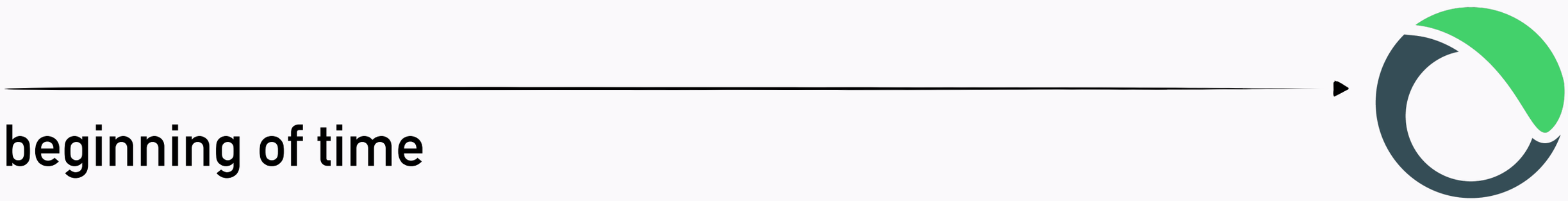


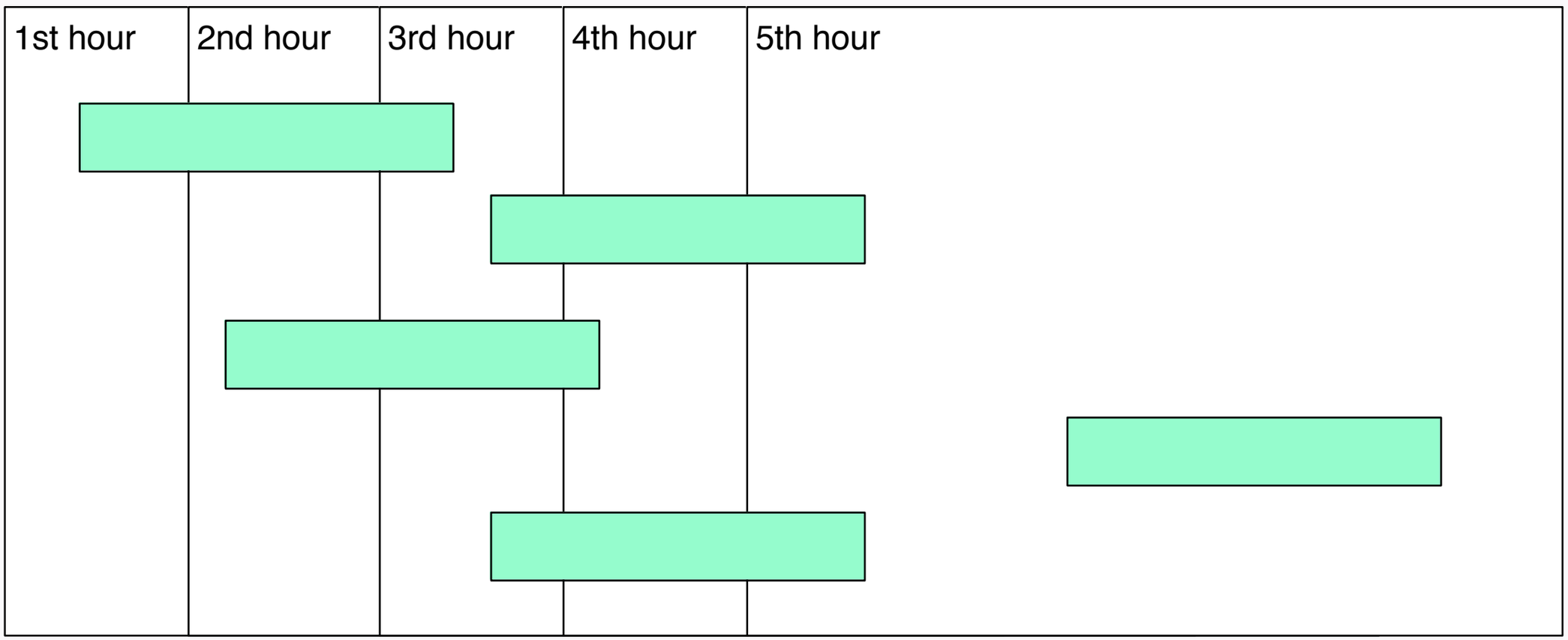
API: Process



beginning of time

structure functional context side effects flow **process**





beginning of time

structure functional context side effects flow **process**

```clojure
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

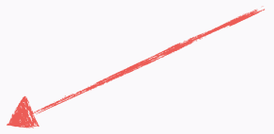
```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])
```



```
(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```



```
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```clojure
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

```clojure
(def windows
  [{:window/id :collect-segments
    :window/task :process-users
    :window/type :sliding
    :window/aggregation [:onyx.windowing.aggregation/min :age]
    :window/window-key :event-time
    :window/range [1 :hour]
    :window/slide [30 :minutes]}])


(def triggers
  [{:trigger/window-id :collect-segments
    :trigger/refinement :discarding
    :trigger/on :segment
    :trigger/threshold [500 :elements]
    :trigger/sync :my.ns/sync-to-dynamo}])
```

Let's talk

Kafka

core.async

Datomic

Durable Queue

SQL

arbitrary lazy seqs

Redis

Robert Stuttaford
@RobStuttaford

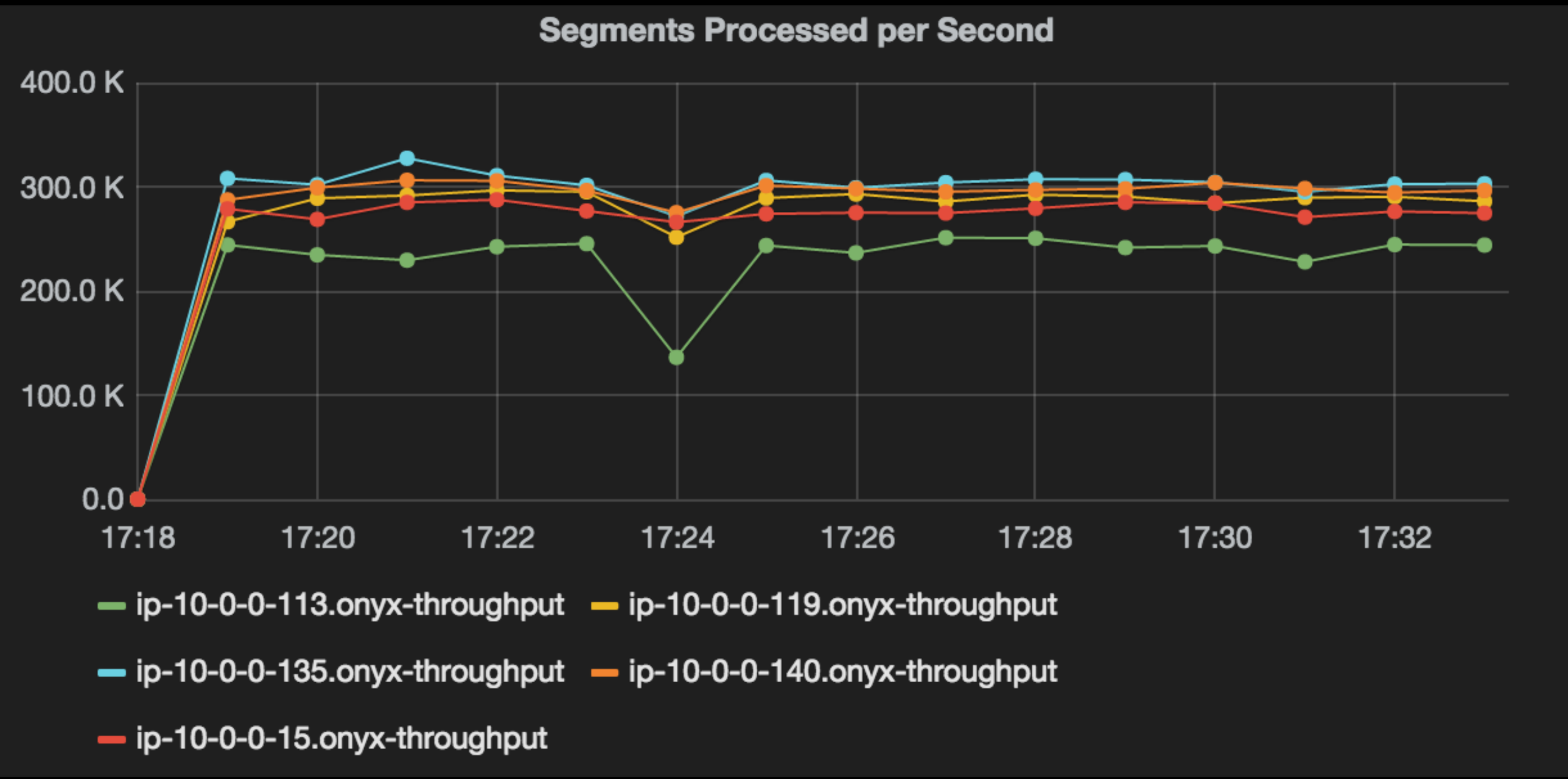
Follow

Our new system uses a dream stack: #Clojure, #ClojureScript, #Datomic and #Onyx. Datomic does the Remembering, and Onyx the Thinking.

8:27 AM - 24 Aug 2015

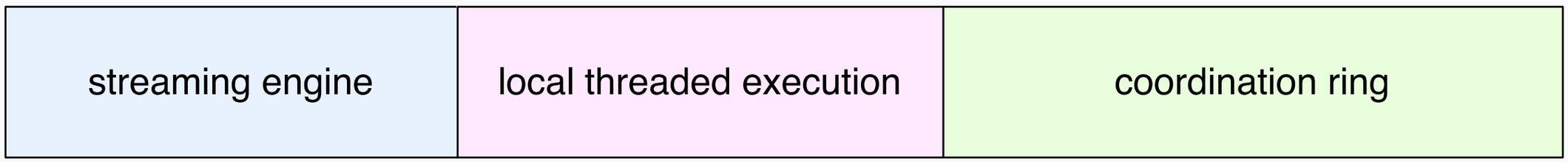
9 29

Got Perf?

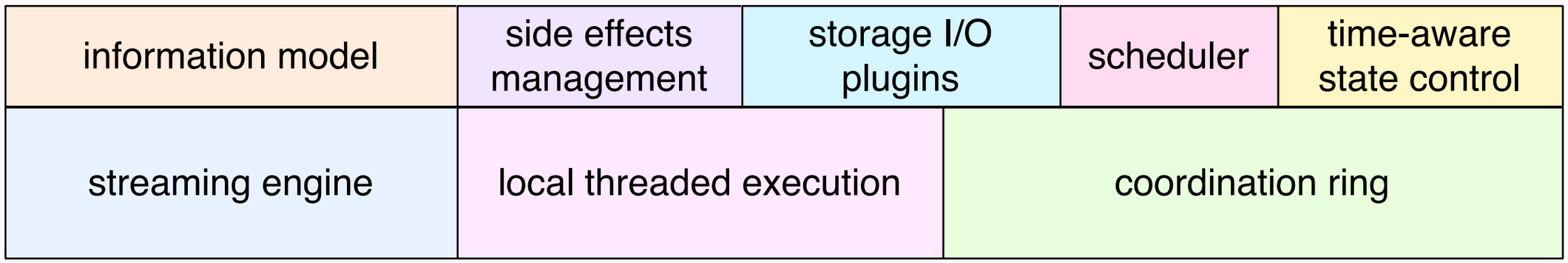

Segments Processed per Second
400.0 K
300.0 K
200.0 K
100.0 K
0.0
17:18
17:20
17:22
17:24
17:26
17:28
17:30
17:32
ip-10-0-0-113.onyx-throughput
ip-10-0-0-119.onyx-throughput
ip-10-0-0-135.onyx-throughput
ip-10-0-0-140.onyx-throughput
ip-10-0-0-15.onyx-throughput

The Ecosystem

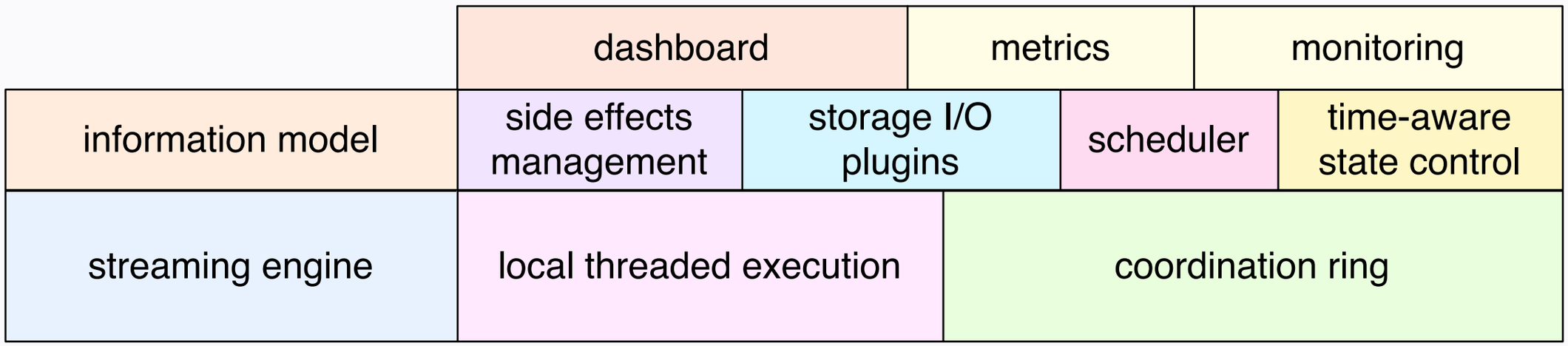
The Ecosystem



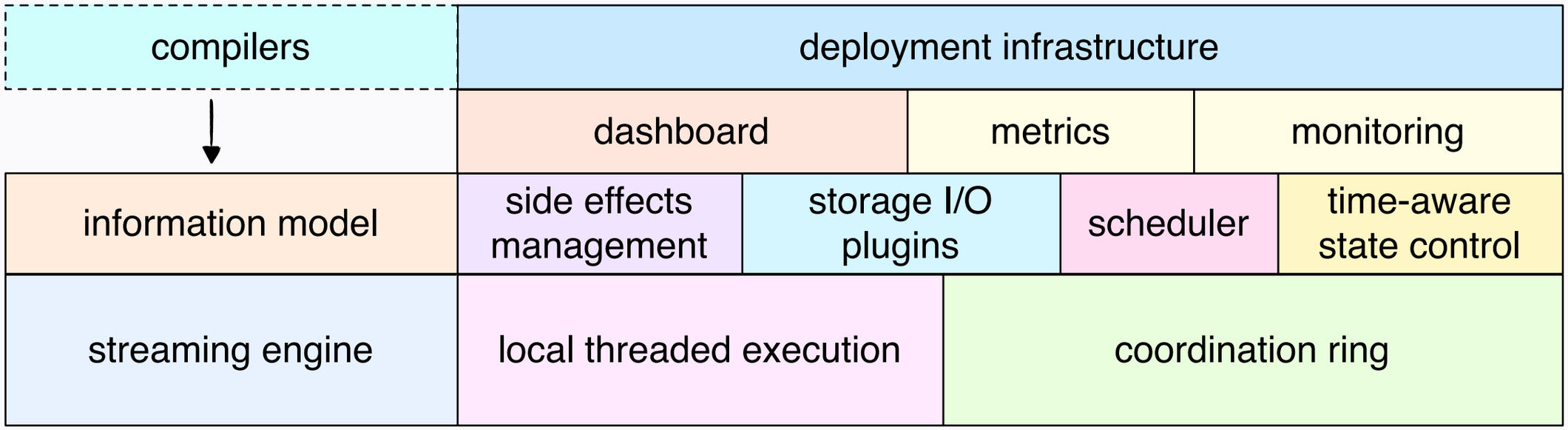
The Ecosystem



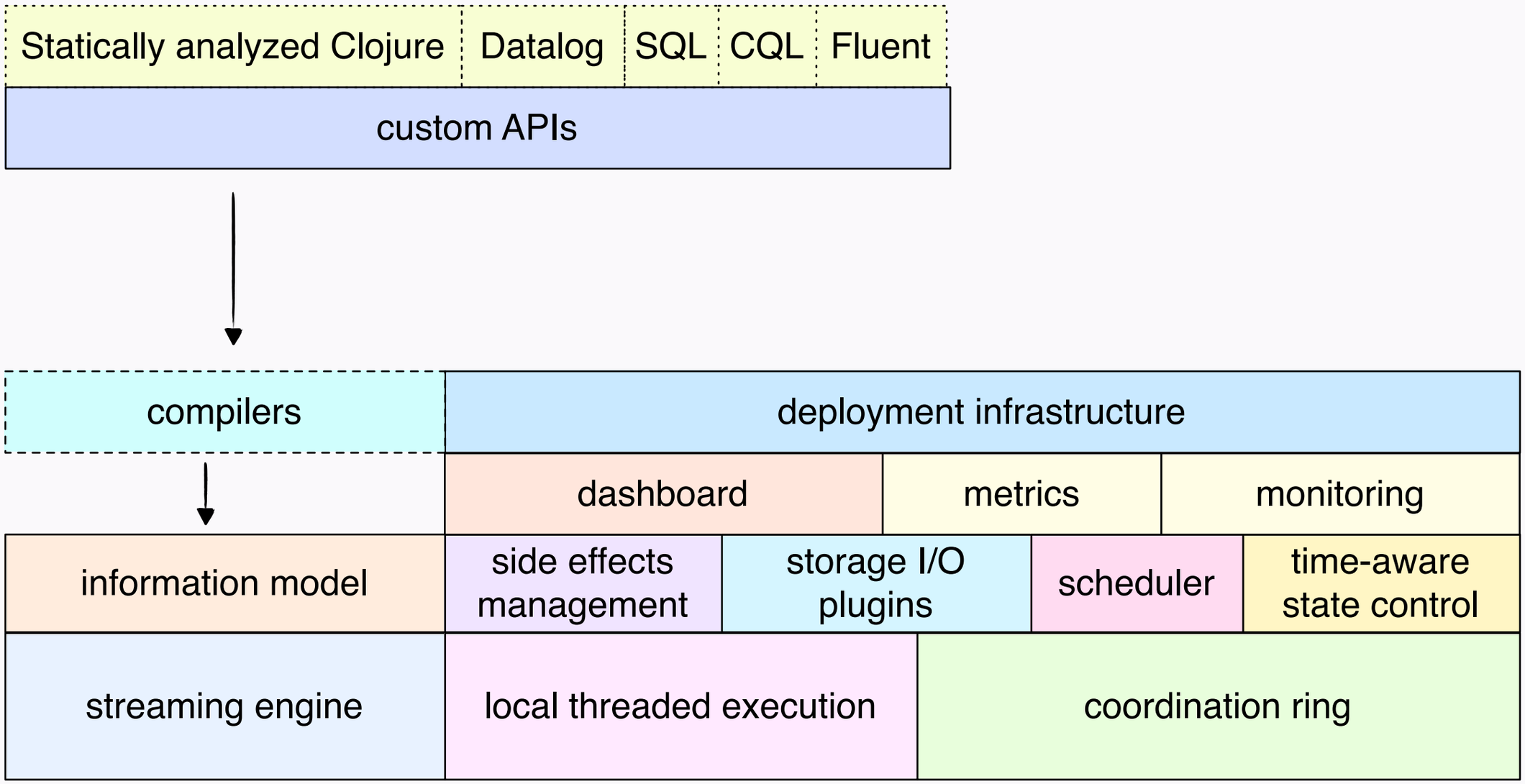
The Ecosystem



The Ecosystem



The Ecosystem



Production Grade

cognician

thinktopic

LITTLE BIRD

Thank you!!

Lucas Bradstreet (@ghaz)

Learn

- Website: http://www.onyxplatform.org/
- GitHub: https://github.com/onyx-platform/onyx
- Gitter: https://gitter.im/onyx-platform/onyx
- Slack: https://clojurians.slack.com/messages/onyx
- Training: https://github.com/onyx-platform/learn-onyx
- `lein new onyx-app hello-world`

Questions?

@MichaelDrogalis / @OnyxPlatform