



Designing a multi-language live programming tool with Phoenix and GenStage

Andrea Amantini



lo_zampino



zampino

nextjournal



Designing a multi-language live programming tool with Phoenix and GenStage*

Andrea Amantini



lo_zampino



zampino

nextjournal

*and why you don't want to use it this way

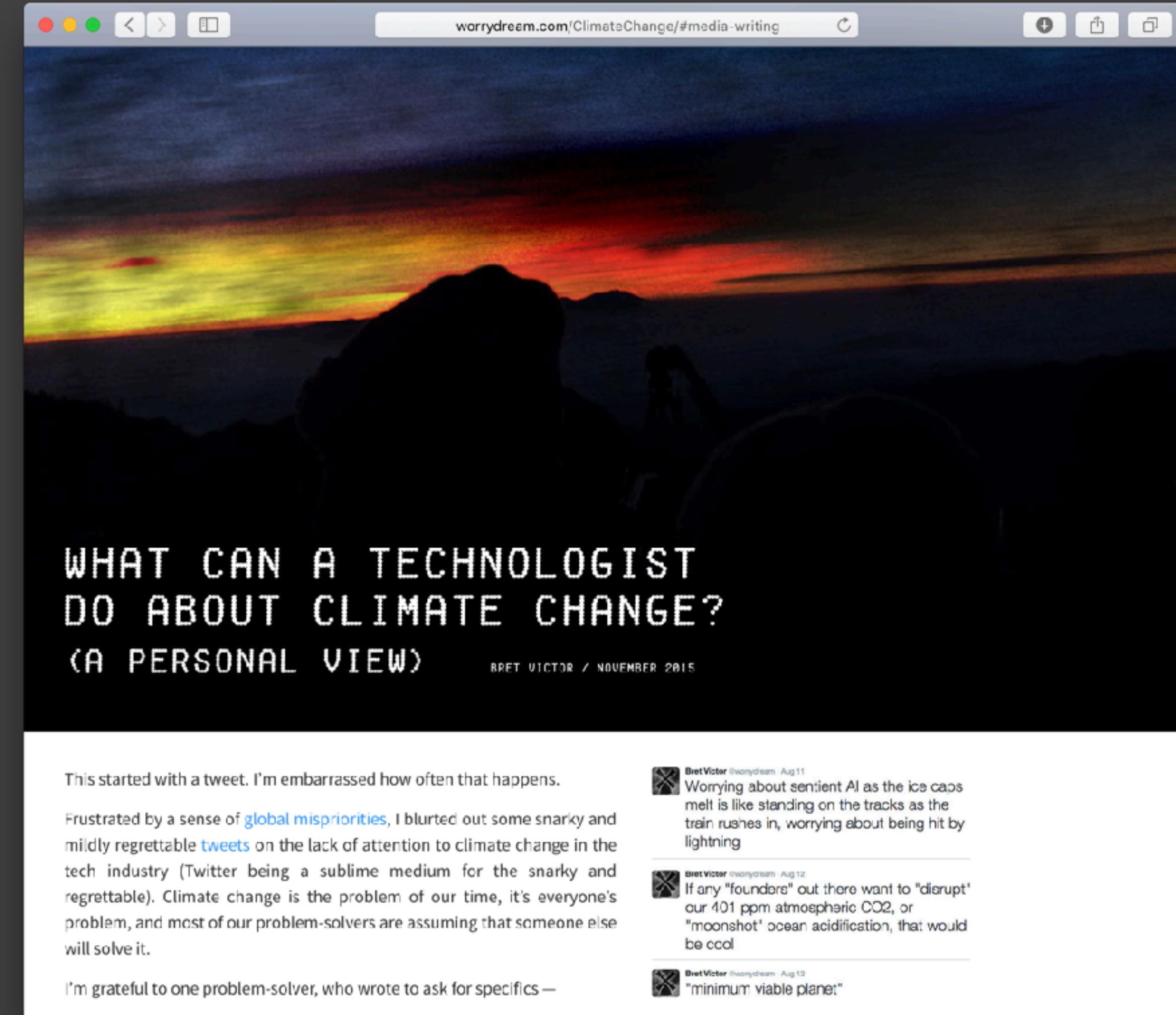
"Imagine an authoring tool designed for
arguing from evidence...

...literally *autocomplete*ing sourced facts
directly into the document...

...what if it were as easy to insert facts,
data, and models as it is to insert emoji
and cat photos?"

—Bret Victor

worrydream.com/ClimateChange



worrydream.com/ClimateChange/#media-writing

WHAT CAN A TECHNOLOGIST DO ABOUT CLIMATE CHANGE? (A PERSONAL VIEW)

BRET VICTOR / NOVEMBER 2015

This started with a tweet. I'm embarrassed how often that happens.

Frustrated by a sense of [global mispriorities](#), I blurted out some snarky and mildly regrettable [tweets](#) on the lack of attention to climate change in the tech industry (Twitter being a sublime medium for the snarky and regrettable). Climate change is the problem of our time, it's everyone's problem, and most of our problem-solvers are assuming that someone else will solve it.

I'm grateful to one problem-solver, who wrote to ask for specifics —

BretVictor worrydream Aug 11 Worrying about sentient AI as the ice caps melt is like standing on the tracks as the train rushes in, worrying about being hit by lightning

BretVictor worrydream Aug 12 If any "founders" out there want to "disrupt" our 401 ppm atmospheric CO₂, or "moonshot" ocean acidification, that would be cool

BretVictor worrydream Aug 12 "minimum viable planet"

nextjournal

localhost:4000/fo0/my-article/edit

nextjournal New article

Writing with Nextjournal

1. Data driven Models

We've uploaded a CSV file with climate data from the OpenNEX PlanetOS project.

OpenNEX-chicago-climate.csv

Let's graph climate change data from the OpenNEX PlanetOS project using Python Pandas. We've uploaded a CSV file - we parse it and prepare the data for plotting. To see the Python code, uncollapse the cell using the arrow icon to the left of the graph.

```
opennexus_climate
using DataFrames
table = readtable(OpenNEX-chicago-climate.csv)

yearProjection = function(column)
    map(d -> split(d, "-")[1], column)
end

table[:Temperature] = table[:Value] - 273.15
table[:Year] = yearProjection(table[:Date])
by_year = by(table, [:Scenario, :Year], t -> maximum(t[:Temperature]))
data = [scatter(;x=g[:Year], y=g[3], name=g[1], :Scenario, mode="lines") for g in g]
plotly_data = Base.typed_vcat(GenericTrace, data)

plot(plotly_data, Layout(layout))
```

Julia>

nothing
183360x7 Data
nothing

(anonymous fu
nothing
[-0.097753906
Any["2000", "2
382x3 DataFra
[{"y": [29.169
[{"y": [29.169
nothing
{"layout": {"y

OpenNex Climate

Temperatures

Year

historical
rcp26
rcp45
rcp60
rcp85



I Title



Runner

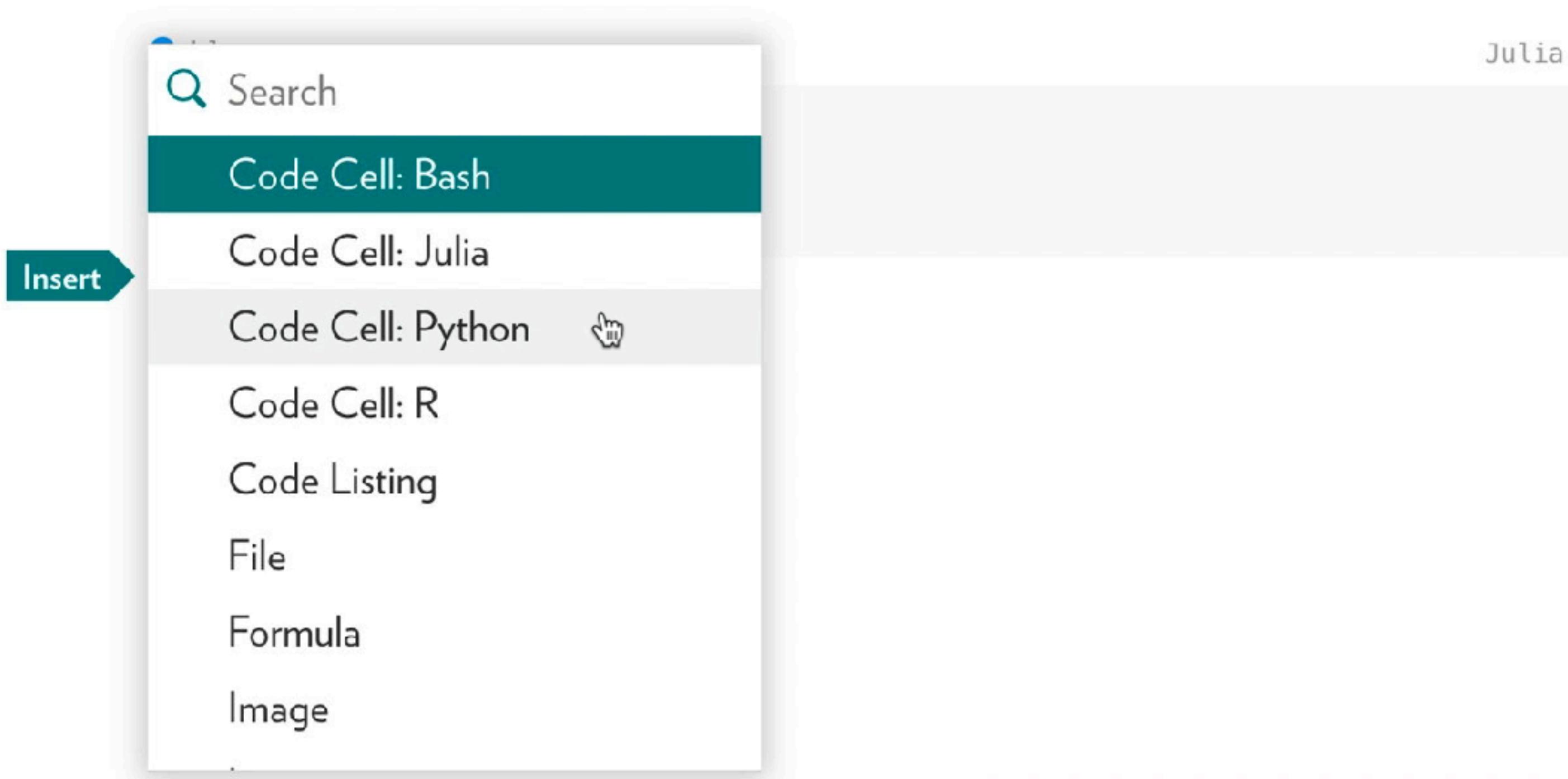


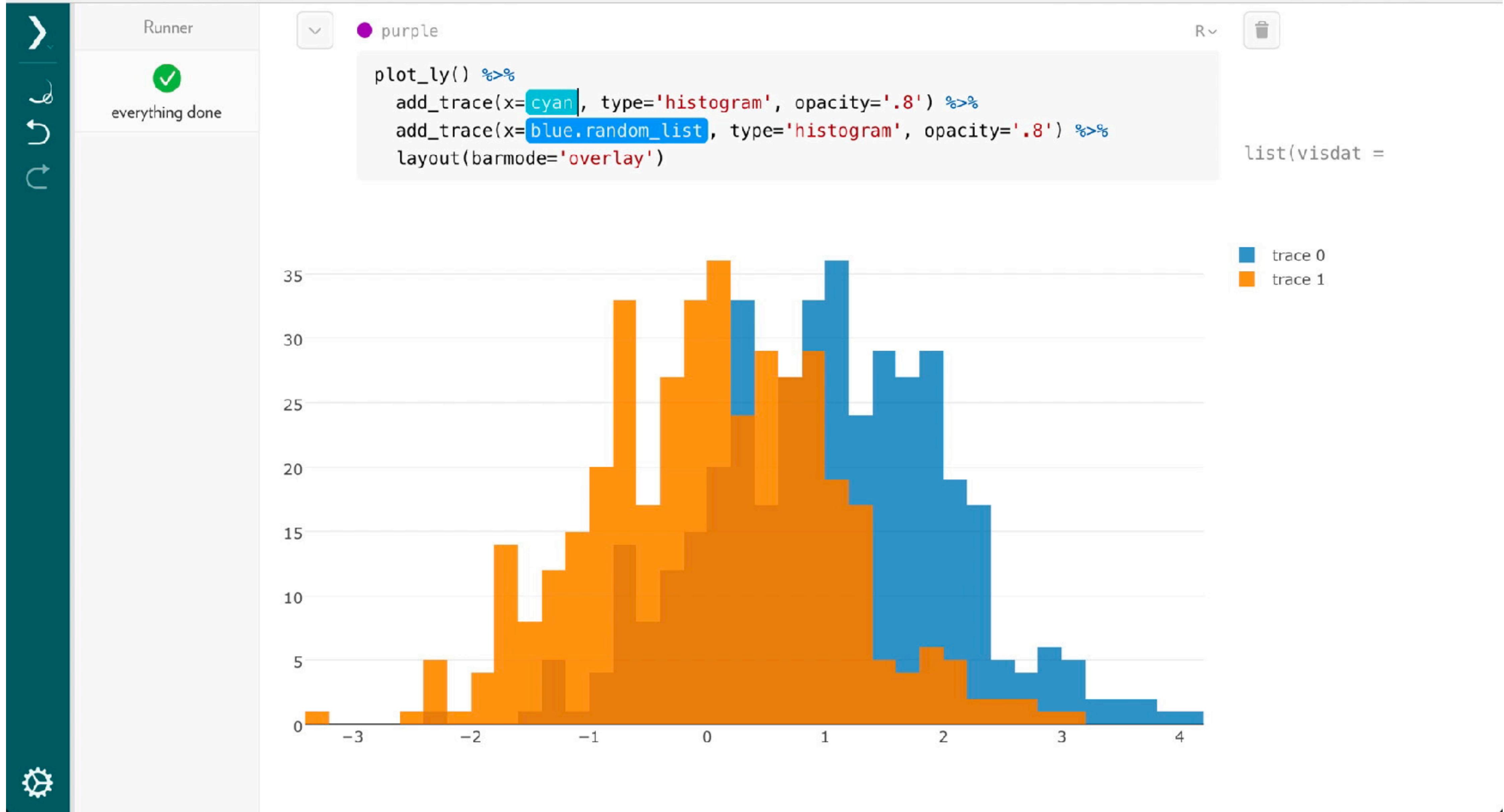
everything done

Welcome to nextjournal

A Nextjournal article is composed of cells. This one is a normal paragraph.

Hovering between sections, we can add new cells. Code cells can run Python, R or Julia... live!







Runner ready

functions

Cell has changes

Alanine Dipeptide Analysis

1. Introduction

We are going to analyze NVE simulations of terminally blocked *Alanine* a little 22-atom compound often used the simplest molecular system to be studied and analyzed. In some sense this is the hydrogen atom of the biomolecular simulation community. For the view of a biologist this is the simplest peptide you could build (although it does not exist in nature). It is comprised of one aminoacid *Alanine* with some initial and final atoms. In reality proteins are build out of chains of the (20 natural occurring) aminoacids.

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, and K. A. Dill, “*Long-time protein folding dynamics from short-time molecular dynamics simulations*,” Multiscale Model. Sim., vol. 5, no. 4, pp. 1214–1226, 2006.

2. File imports

the file uploaded contains the so called replica-exchange trajectories



The screenshot shows a configuration interface for a Jupyter Notebook runner. On the left, there are navigation icons: a green arrow pointing right, a green play button, a green circular arrow, and a gear icon with a hand cursor. Above the gear icon is a status message: "Runner ready". Below the gear icon is a "functions" section with a "Cell has changes" indicator.

Bash

Docker Image: `nextjournal/bash:tag` Preamble: none ▾

Julia

Docker Image: `nextjournal/julia:tag` Preamble: none ▾

Python

Docker Image: `nextjournal/python-pyemma:35c8d0a` Preamble: ● preamble ▾

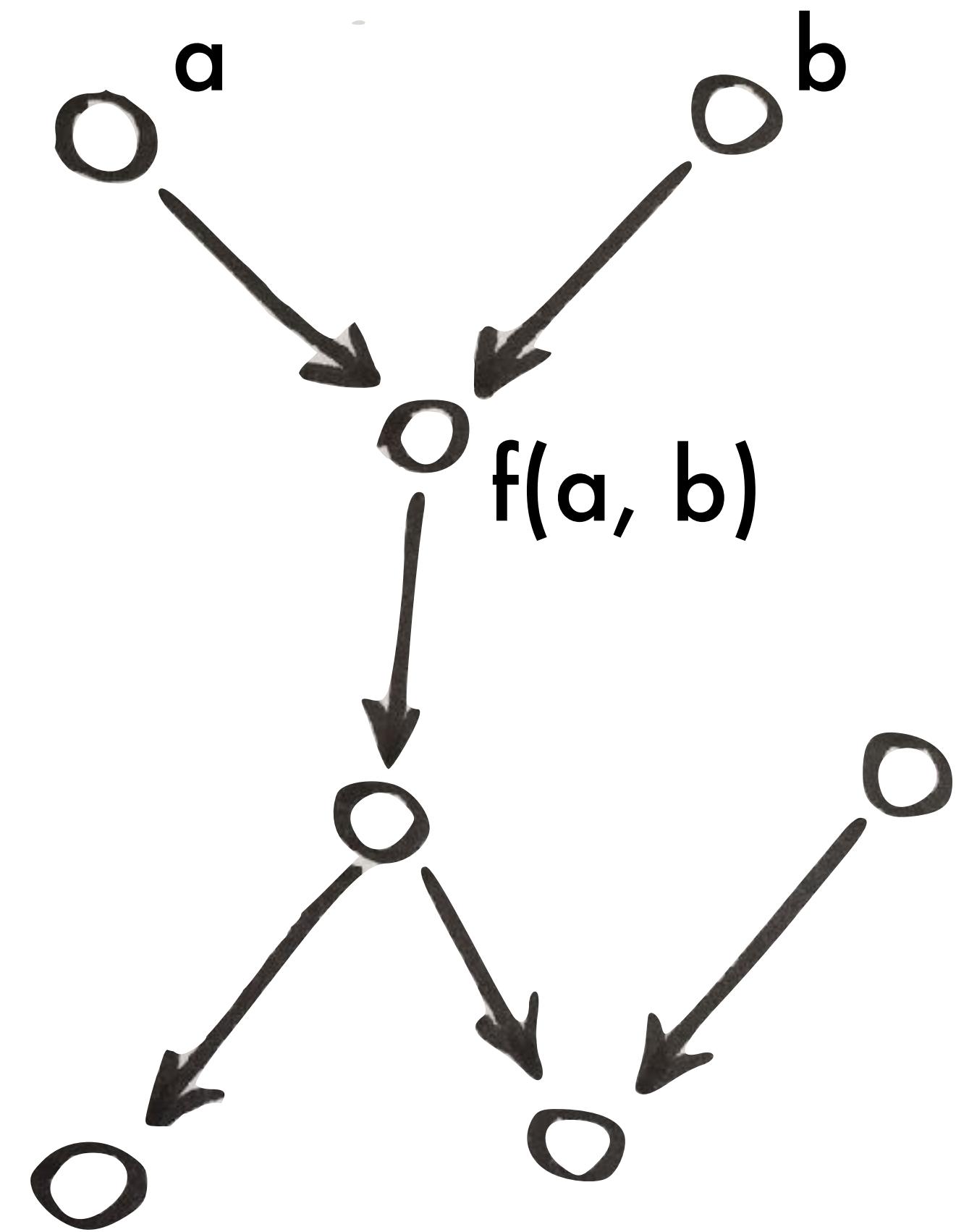
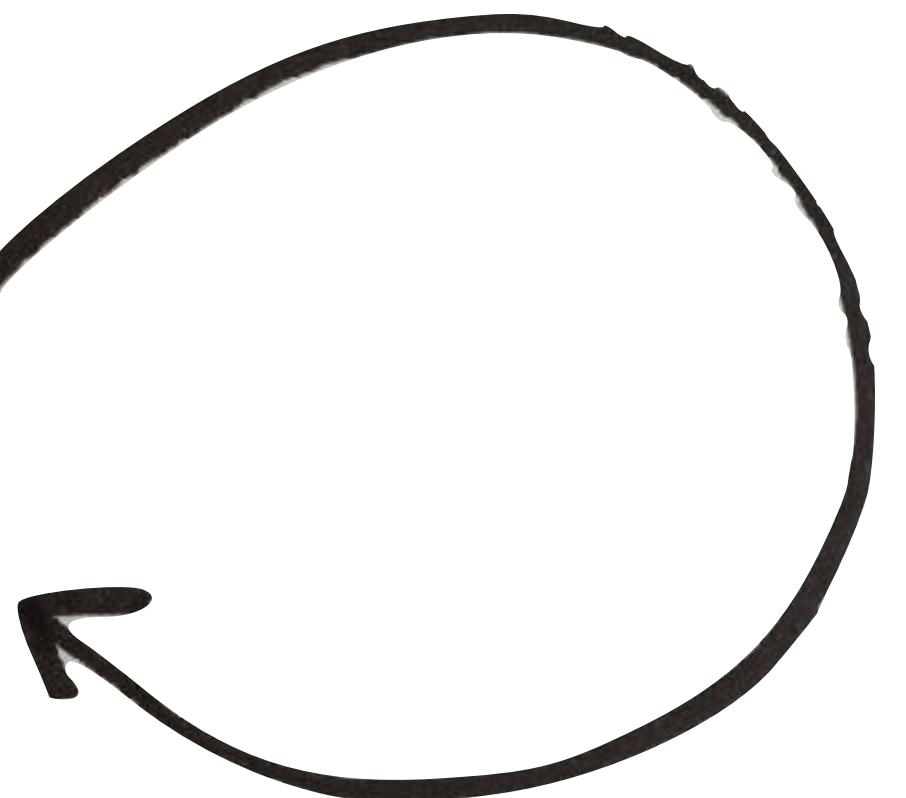
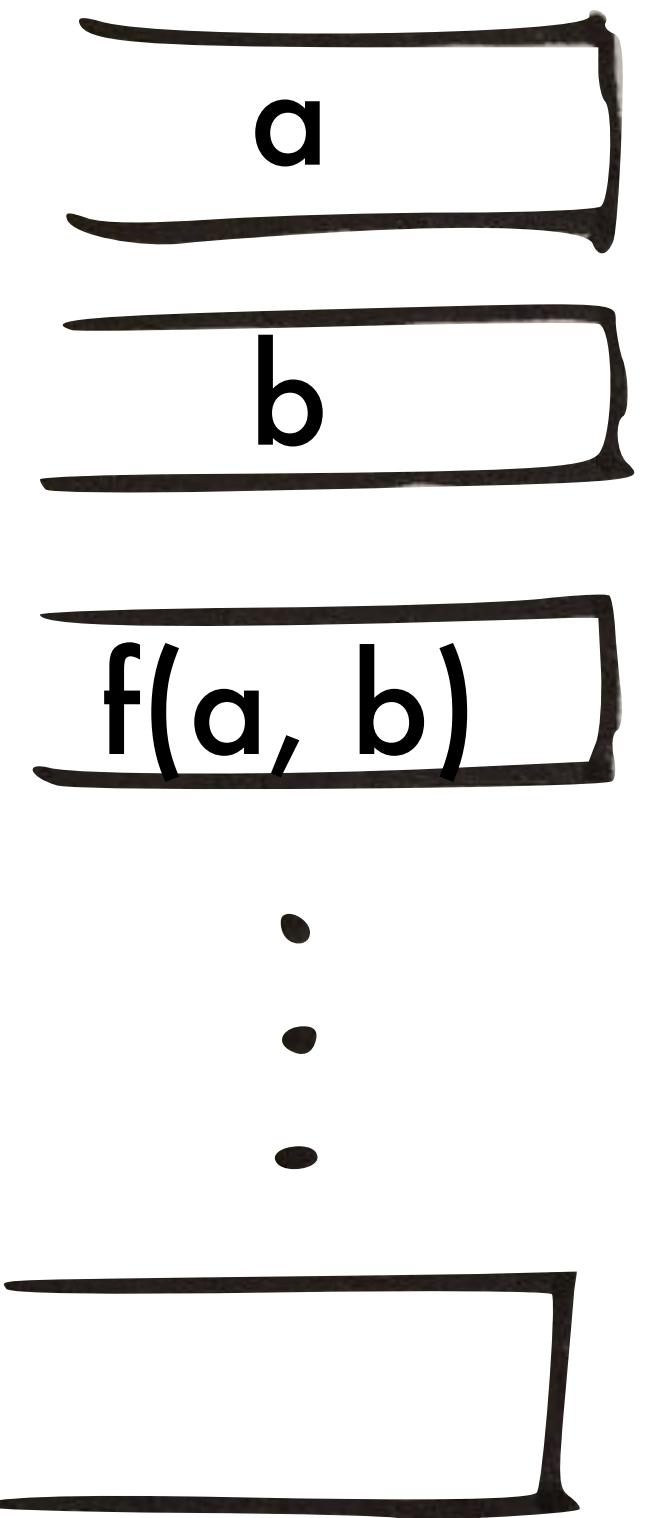
R

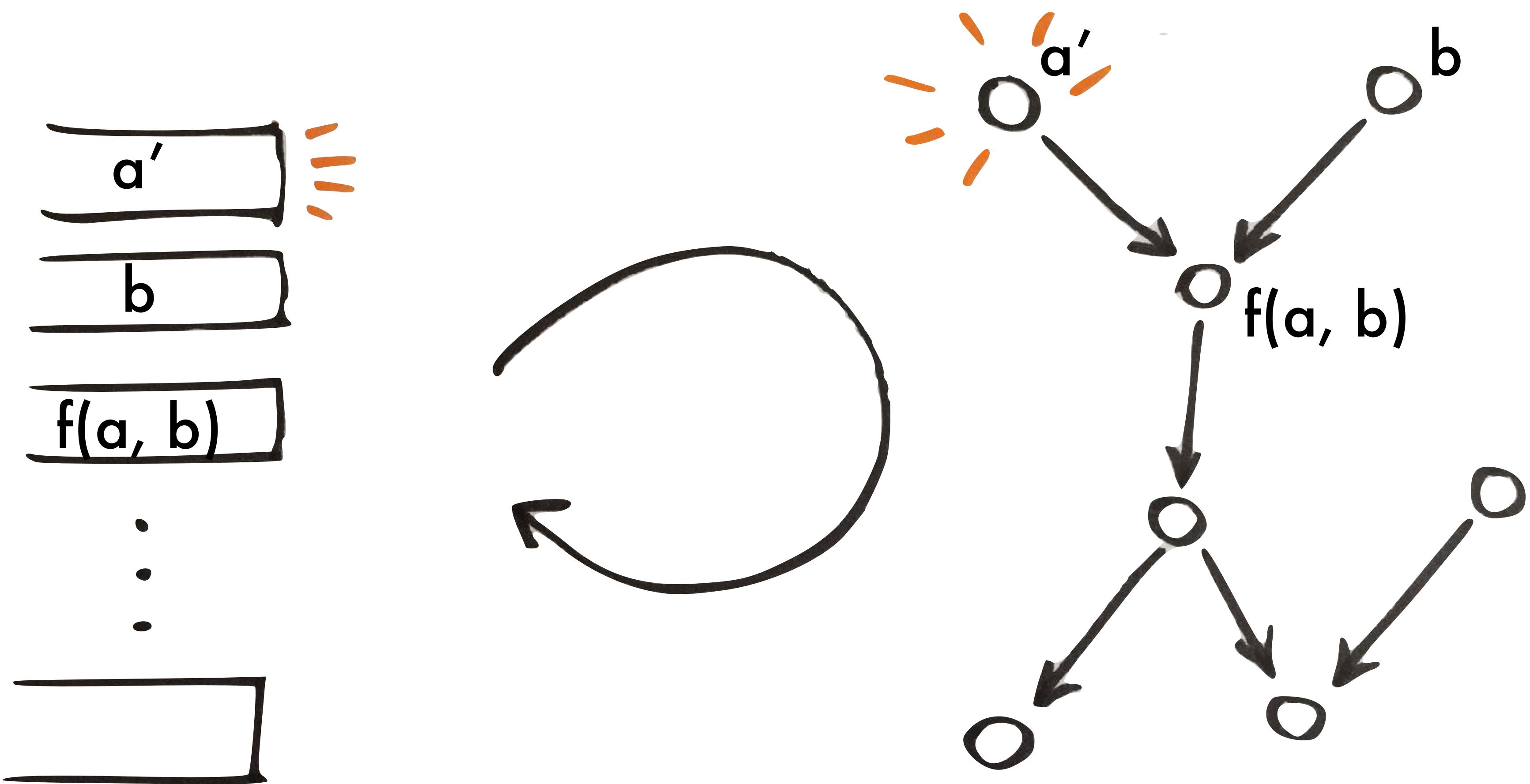
Docker Image: `nextjournal/r:tag` Preamble: none ▾

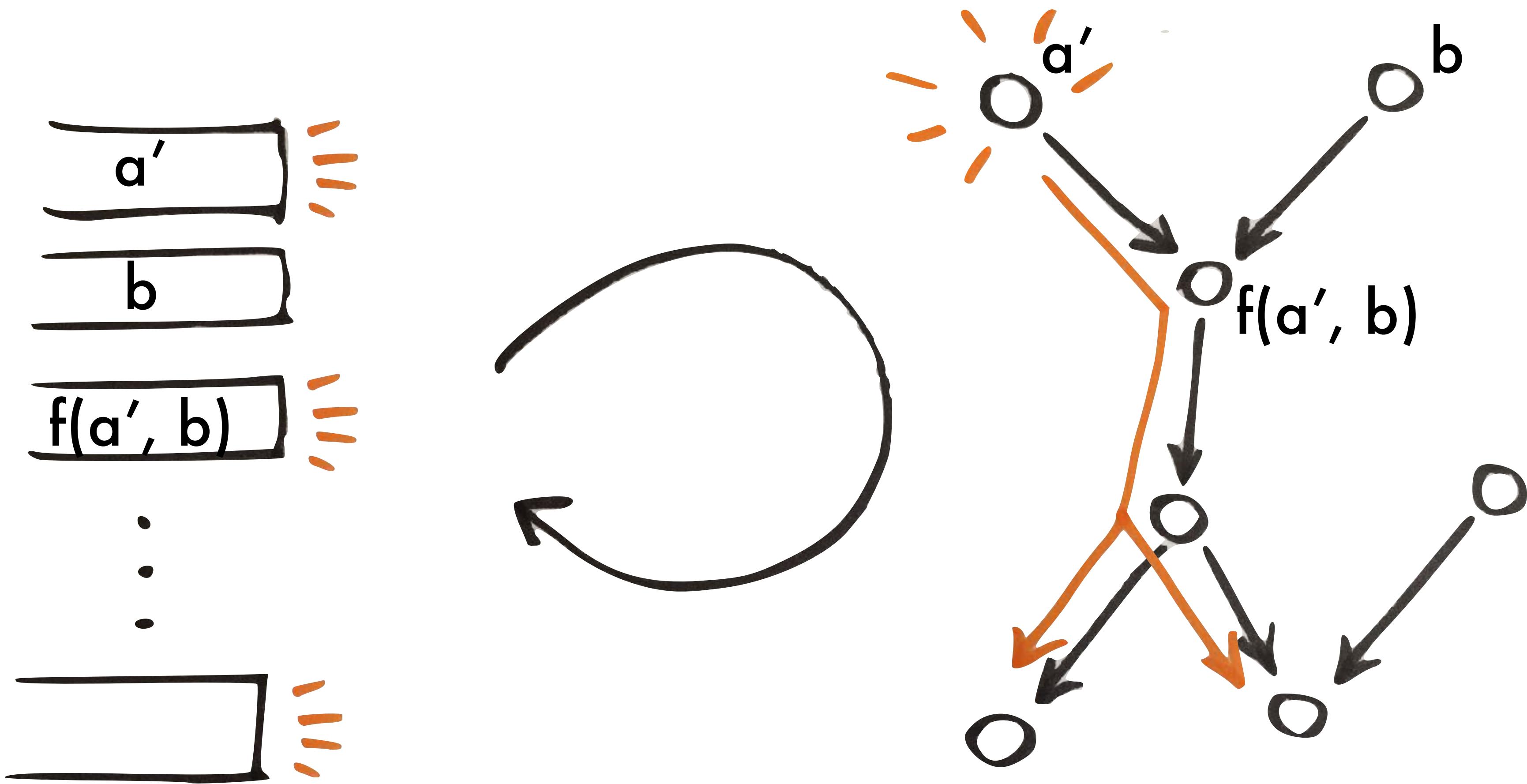
Done

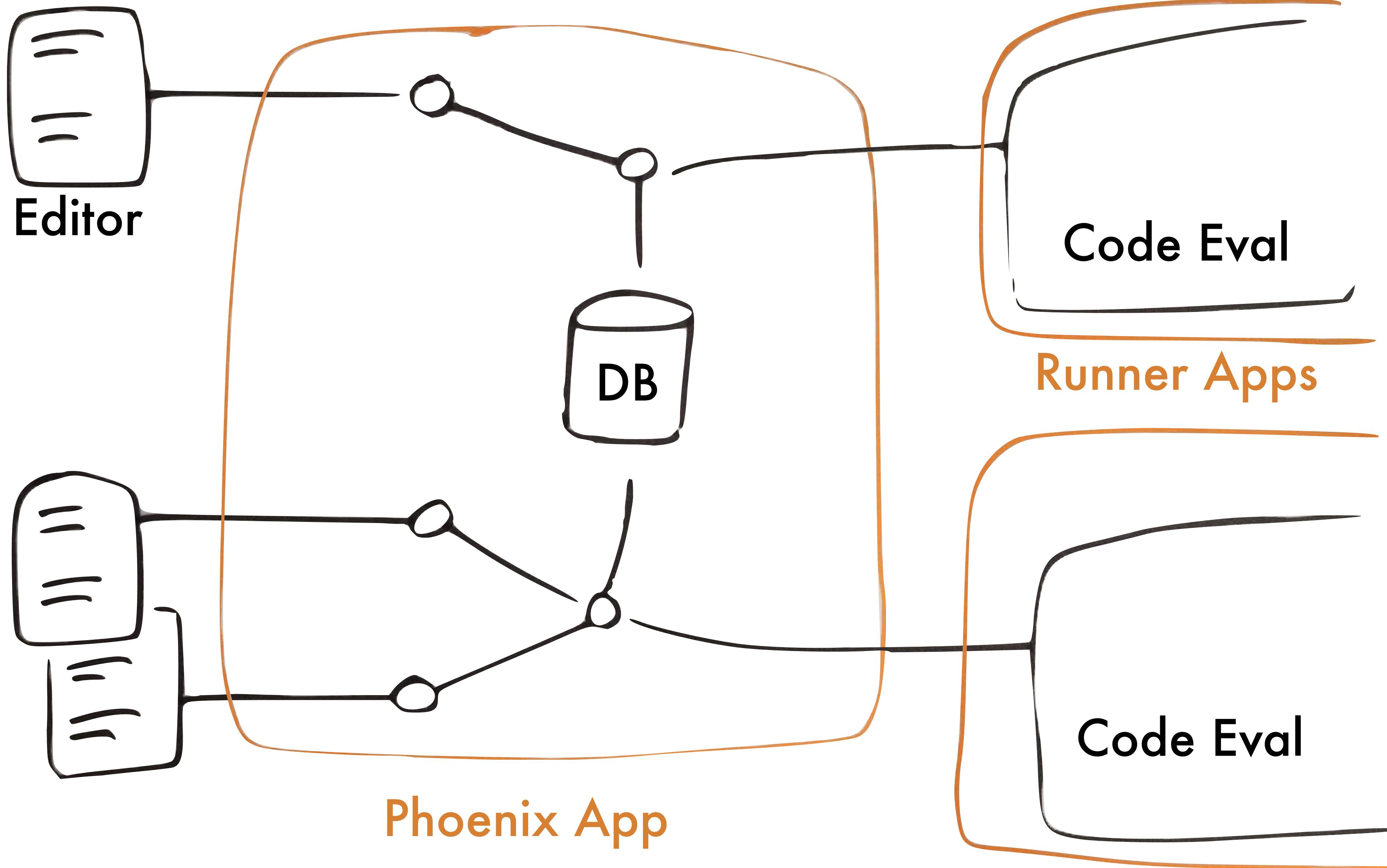
1. Introduction

We are going to analyze NVE simulations of terminally blocked *Alanine* a little 22-atom compound often used the simplest molecular system to be studied and analyzed. In some sense this is the hydrogen atom of the biomolecular simulation

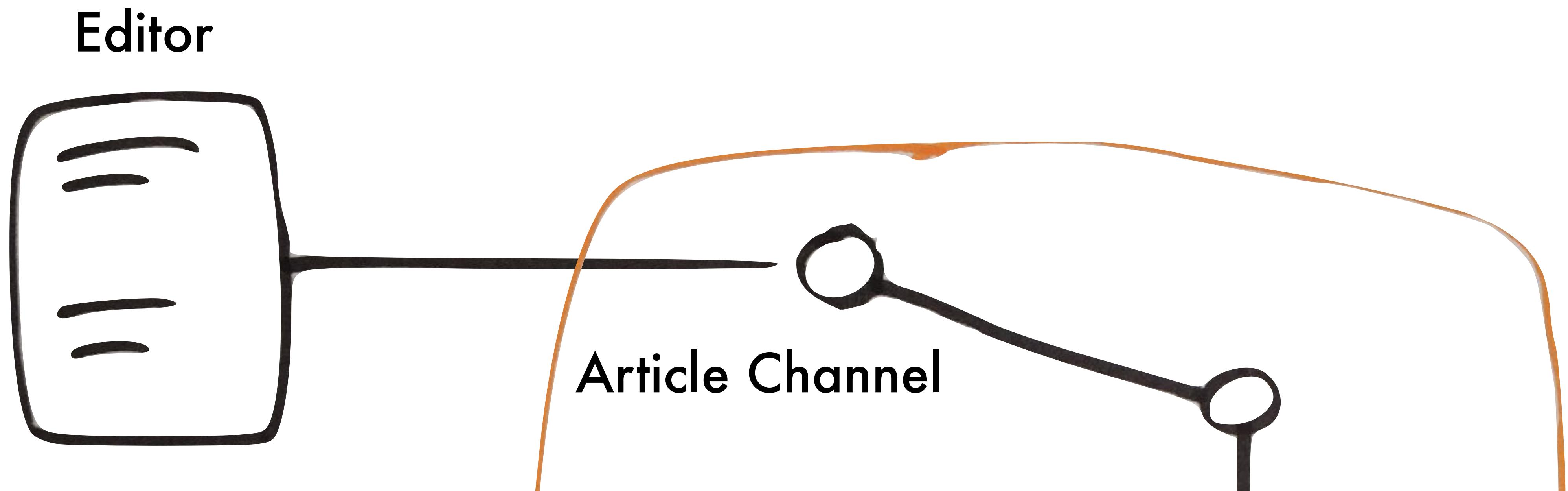








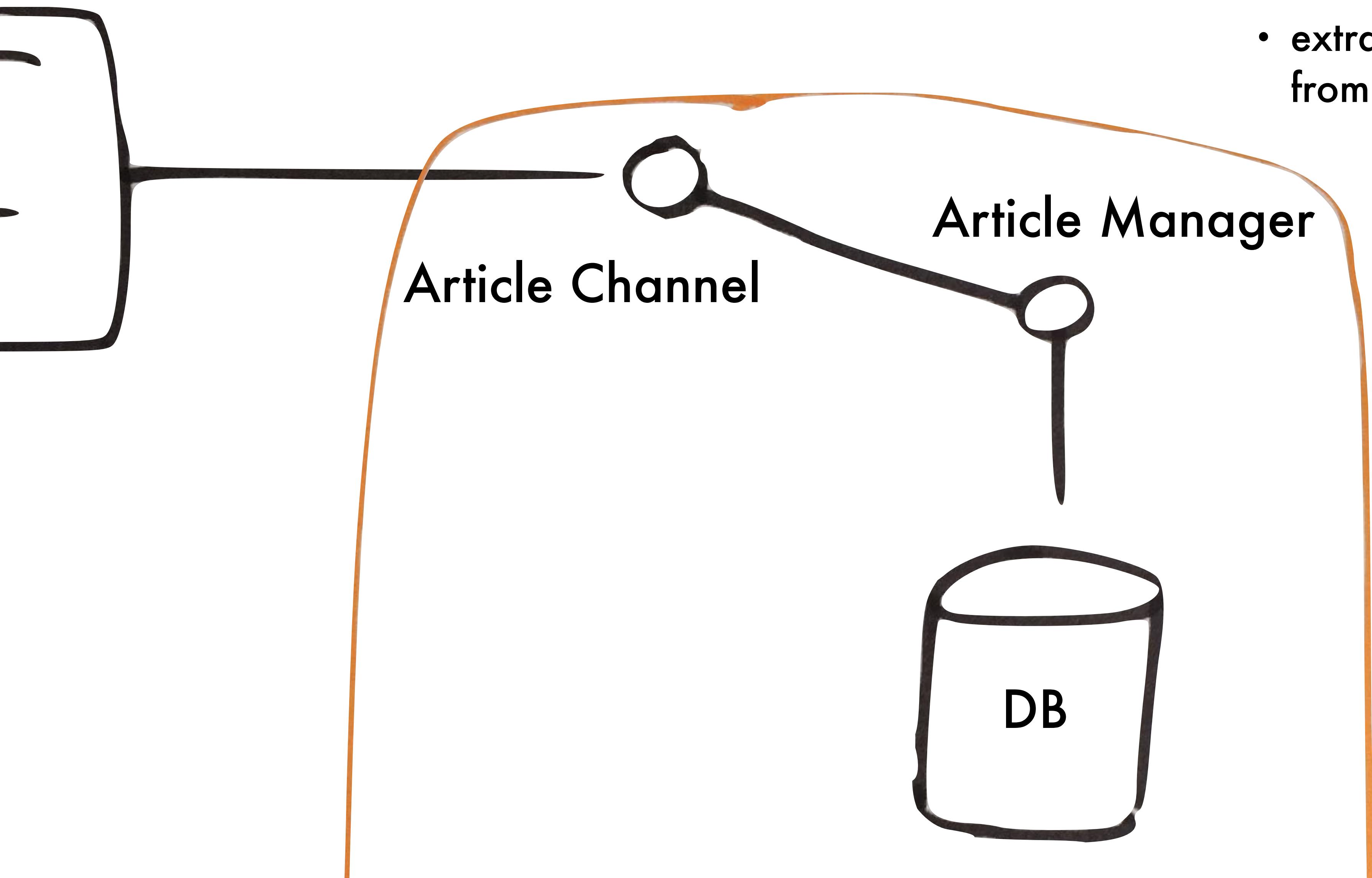
- ClojureScript & re-frame
- Transit Format – Websocket Serialiser/Ecto Type
- exchange clojure/Elixir terms (keywords ⇒ atoms)
- JSON (or MsgPack) based

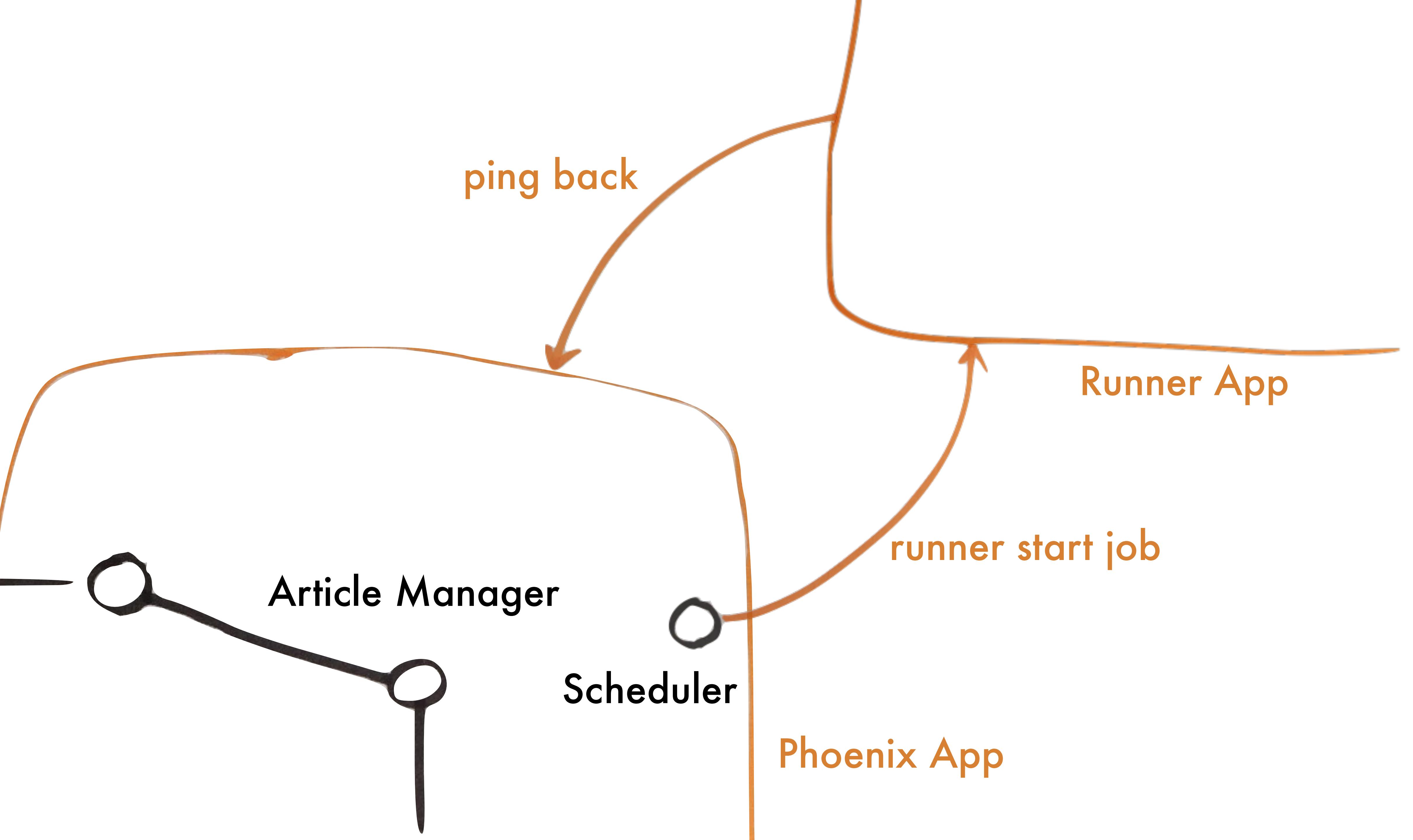


- Persists Edit Operations serialising writes to DB

- extracts CodeCells from Article State

or

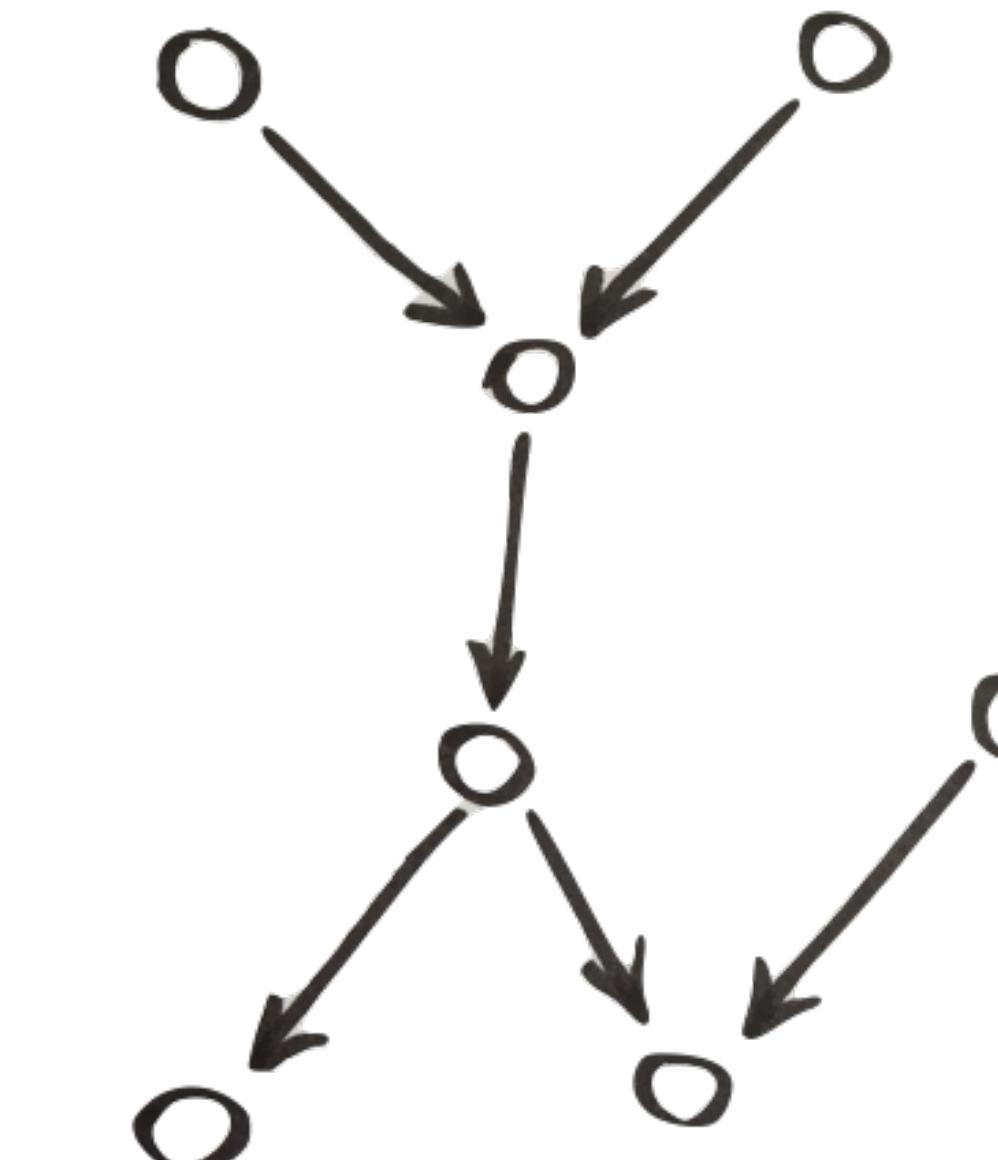




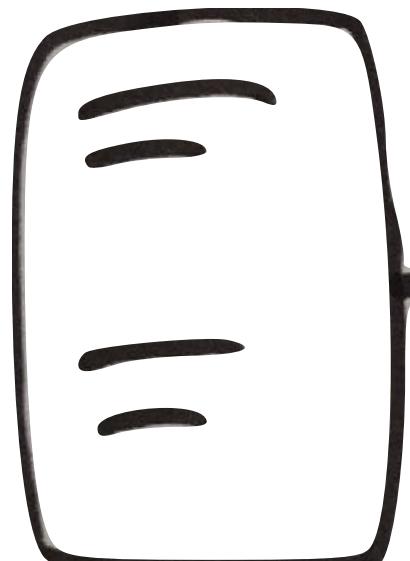


Editor

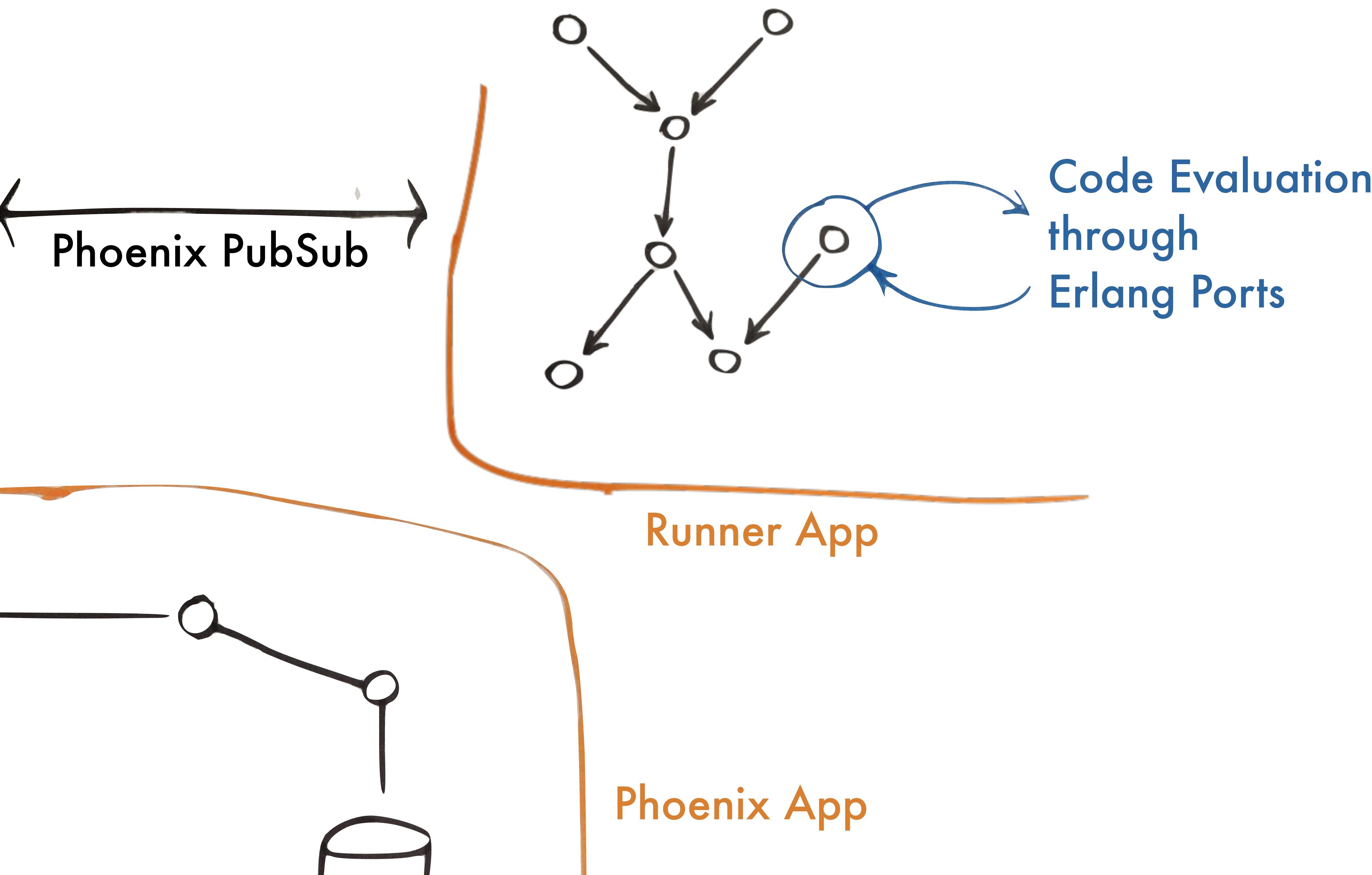
Phoenix PubSub



Runner App

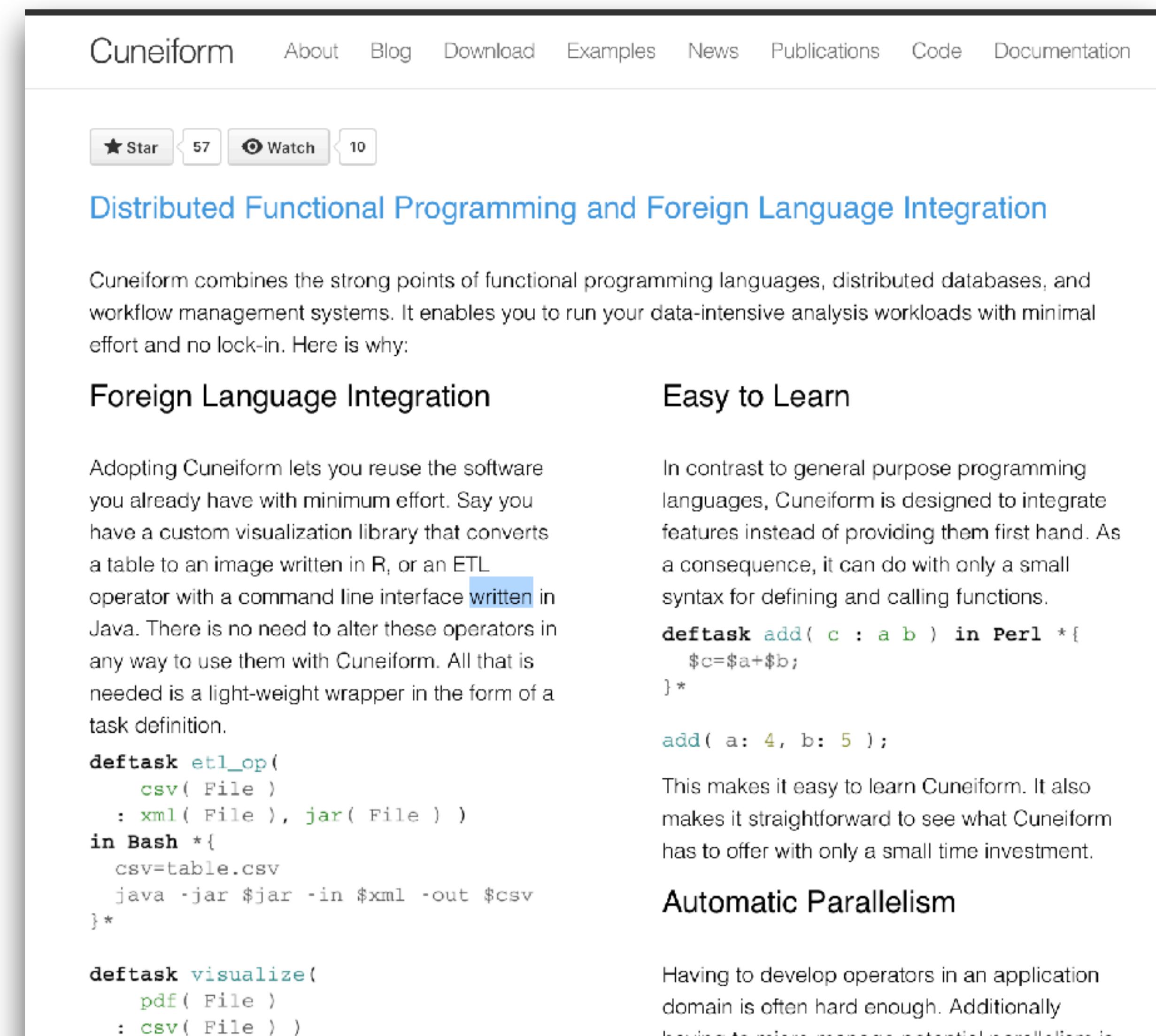


Phoenix App



Take I: Cuneiform backend

- Jörgen Brandt's (@joergenbr) cuneiform-lang.org
- *erlang based language to abstract foreign language interaction and parallelisation*
- **Query-based graph of tasks**



The screenshot shows the GitHub page for the Cuneiform project. The header includes links for Cuneiform, About, Blog, Download, Examples, News, Publications, Code, and Documentation. Below the header, there are social sharing buttons for Star (57), Watch (10), and a link to the repository. The main title is "Distributed Functional Programming and Foreign Language Integration". A brief description states: "Cuneiform combines the strong points of functional programming languages, distributed databases, and workflow management systems. It enables you to run your data-intensive analysis workloads with minimal effort and no lock-in. Here is why:"

Foreign Language Integration

Adopting Cuneiform lets you reuse the software you already have with minimum effort. Say you have a custom visualization library that converts a table to an image written in R, or an ETL operator with a command line interface written in Java. There is no need to alter these operators in any way to use them with Cuneiform. All that is needed is a light-weight wrapper in the form of a task definition.

```
deftask add( c : a b ) in Perl *{  
    $c=$a+$b;  
}  
  
add( a: 4, b: 5 );
```

Easy to Learn

In contrast to general purpose programming languages, Cuneiform is designed to integrate features instead of providing them first hand. As a consequence, it can do with only a small syntax for defining and calling functions.

```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
}*
```

This makes it easy to learn Cuneiform. It also makes it straightforward to see what Cuneiform has to offer with only a small time investment.

Automatic Parallelism

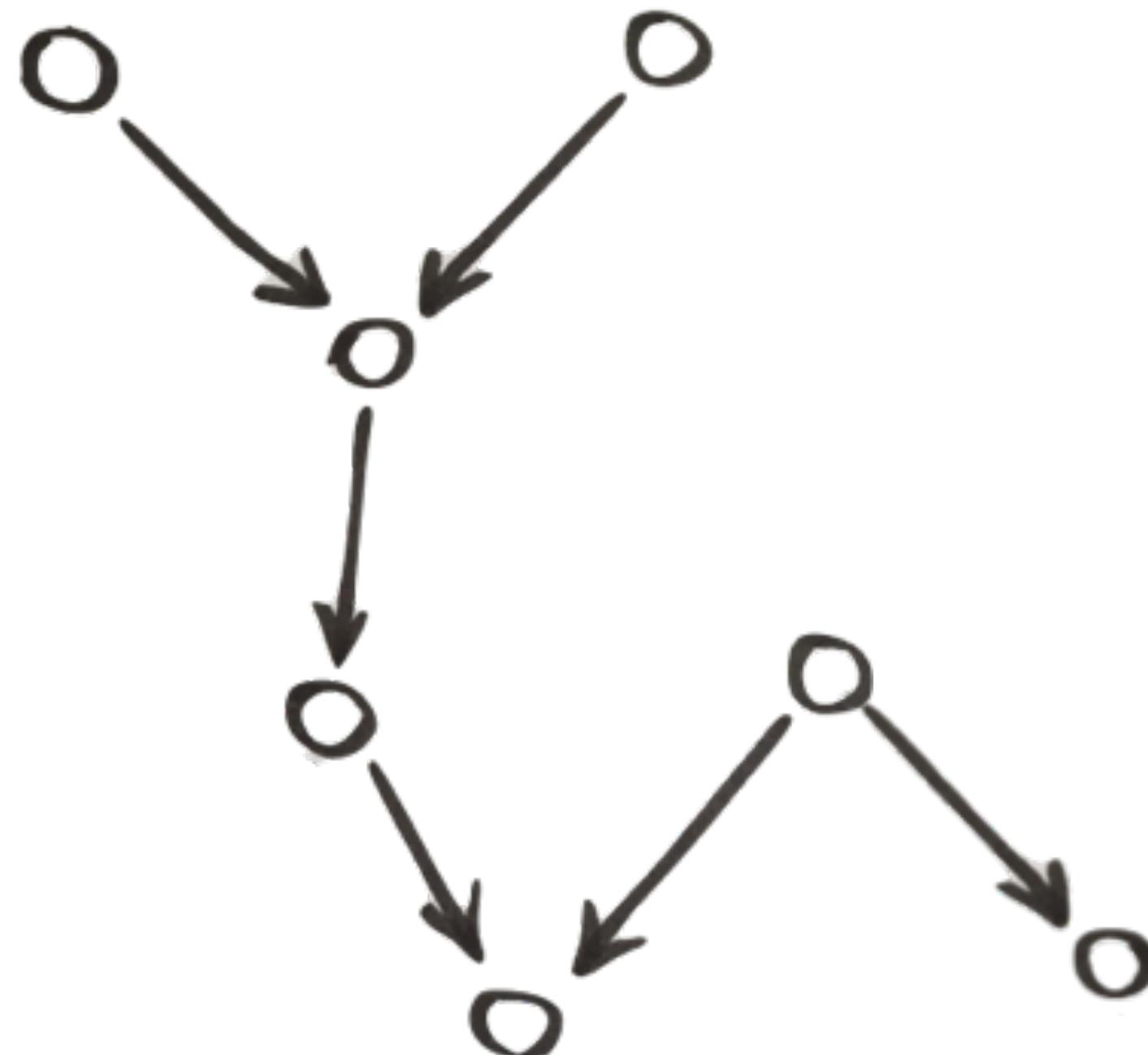
Having to develop operators in an application domain is often hard enough. Additionally having to micro-manage potential parallelism is

Take I: Cuneiform backend

- Jörgen Brandt's (@joergenbr)
cuneiform-lang.org
- *erlang based language to abstract foreign language interaction and parallelisation*
- Query-based graph of tasks

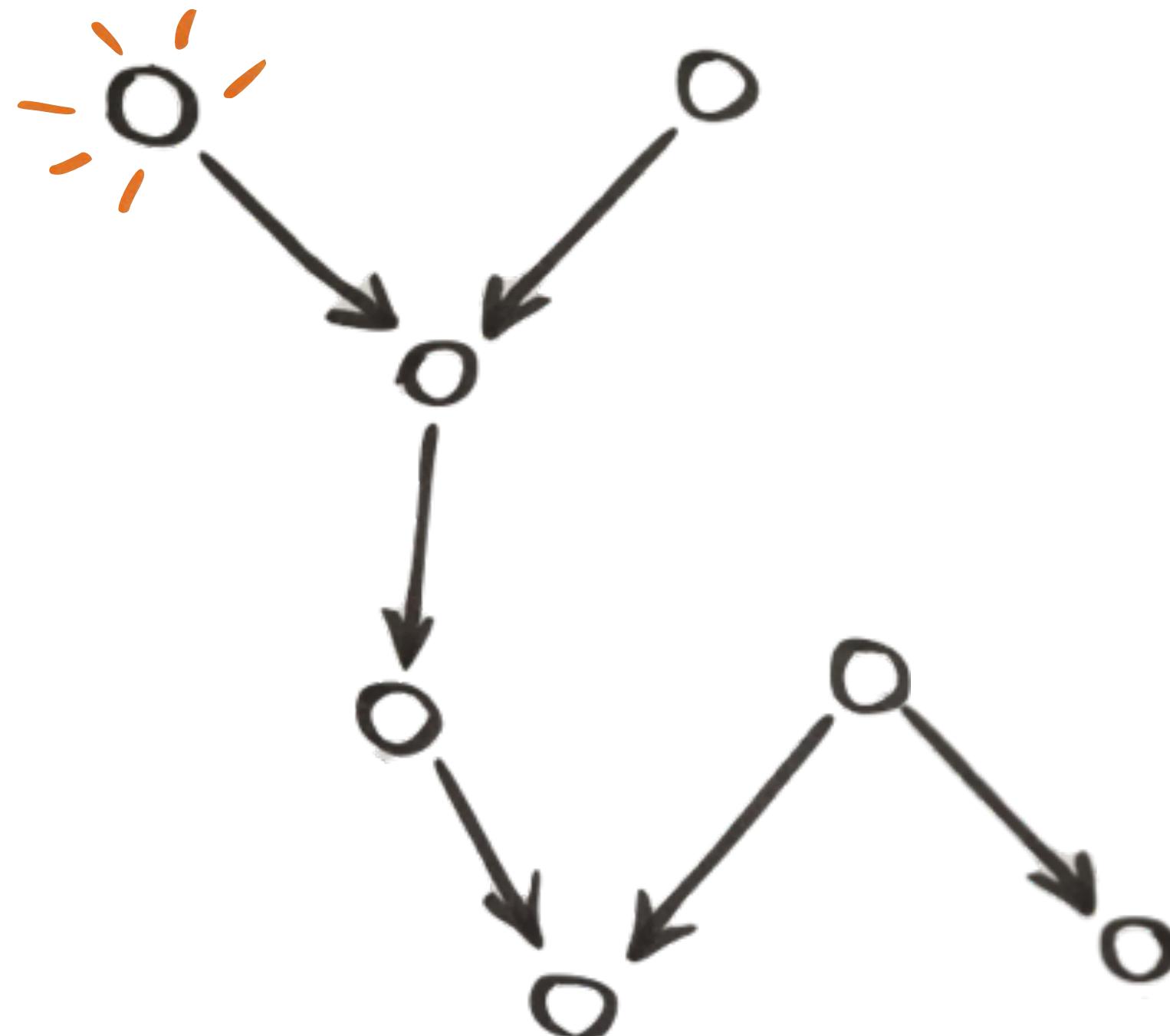
```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
}*  
  
deftask visualize(  
    pdf( File )  
    : csv( File ) )  
in R *{  
    library( my_fancy_r_library )  
    data <- read.csv( csv )  
    img <- visualize.data( data )  
    pdf <- write.pdf( img, "img.pdf" )  
}*  
  
jar = "etl.jar";  
xml = "my_data.xml";  
csv = etl_op( xml: xml, jar: jar );  
pdf = visualize( csv: csv );
```

Take I: Cuneiform backend



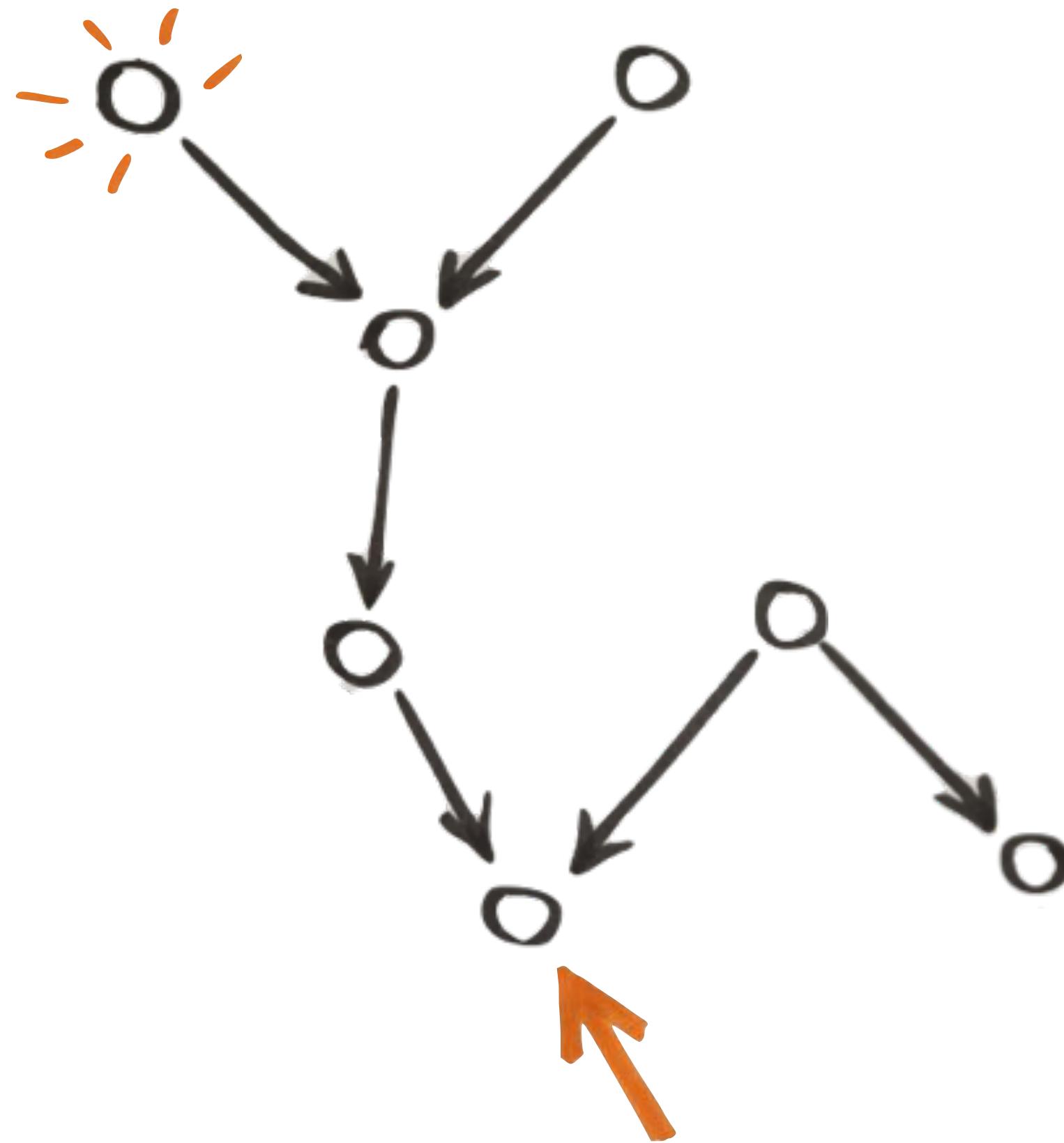
```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
}*  
  
deftask visualize(  
    pdf( File )  
    : csv( File ) )  
in R *{  
    library( my_fancy_r_library )  
    data <- read.csv( csv )  
    img <- visualize.data( data )  
    pdf <- write.pdf( img, "img.pdf" )  
}*  
  
jar = "etl.jar";  
xml = "my_data.xml";  
csv = etl_op( xml: xml, jar: jar );  
pdf = visualize( csv: csv );
```

Take I: Cuneiform backend



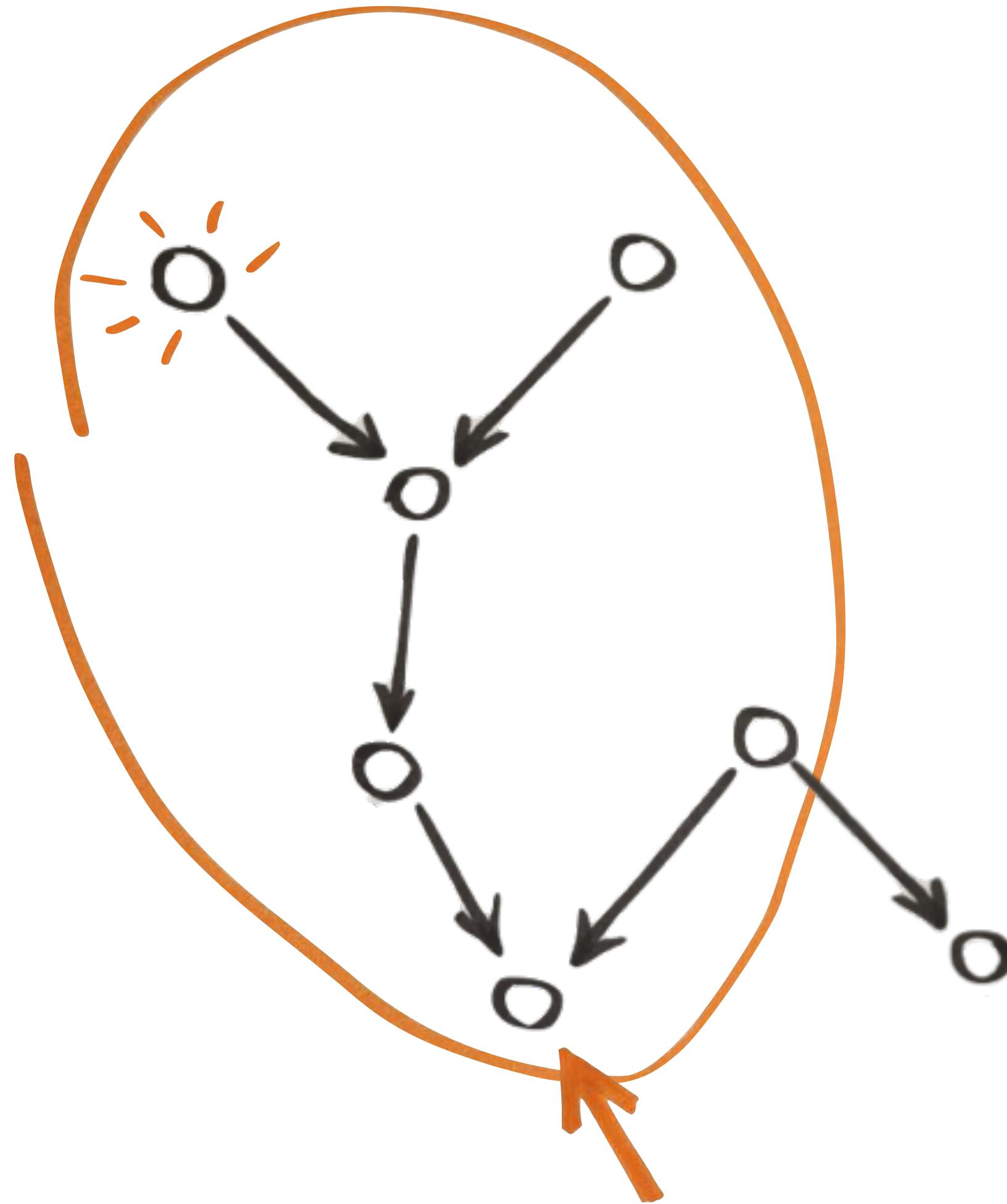
```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
} *  
  
deftask visualize(  
    pdf( File )  
    : csv( File ) )  
in R *{  
    library( my_fancy_r_library )  
    data <- read.csv( csv )  
    img <- visualize.data( data )  
    pdf <- write.pdf( img, "img.pdf" )  
} *  
  
jar = "etl.jar";  
xml = "my_data.xml";  
csv = etl_op( xml: xml, jar: jar );  
pdf = visualize( csv: csv );
```

Take I: Cuneiform backend



```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
} *  
  
deftask visualize(  
    pdf( File )  
    : csv( File ) )  
in R *{  
    library( my_fancy_r_library )  
    data <- read.csv( csv )  
    img <- visualize.data( data )  
    pdf <- write.pdf( img, "img.pdf" )  
} *  
  
jar = "etl.jar";  
xml = "my_data.xml";  
csv = etl_op( xml: xml, jar: jar );  
pdf = visualize( csv: csv );
```

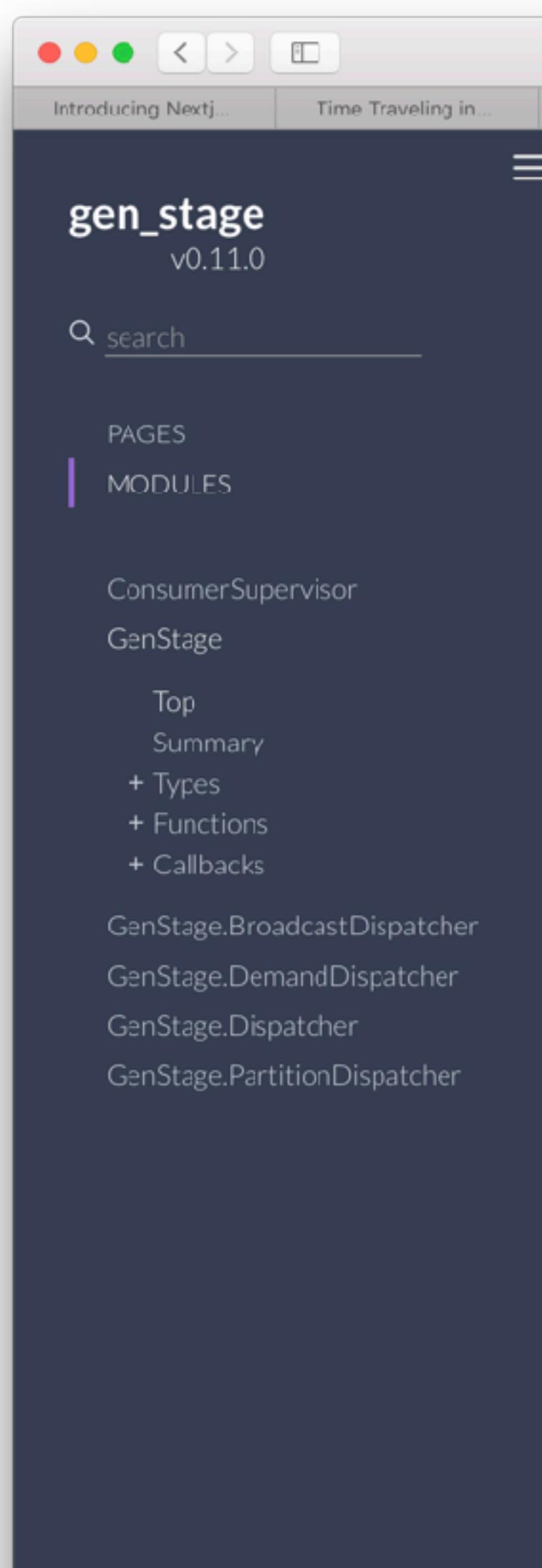
Take I: Cuneiform backend



```
deftask etl_op(  
    csv( File )  
    : xml( File ), jar( File ) )  
in Bash *{  
    csv=table.csv  
    java -jar $jar -in $xml -out $csv  
}*  
  
deftask visualize(  
    pdf( File )  
    : csv( File ) )  
in R *{  
    library( my_fancy_r_library )  
    data <- read.csv( csv )  
    img <- visualize.data( data )  
    pdf <- write.pdf( img, "img.pdf" )  
}*  
  
jar = "etl.jar";  
xml = "my_data.xml";  
csv = etl_op( xml: xml, jar: jar );  
pdf = visualize( csv: csv );
```

Take II: GenStage

- $[A] \rightarrow [B] \rightarrow [C]$
- “thin” layer on top of **GenServer**
- **producers, consumers and producer-consumers**
- **demand-based batch-processing pipelines**
- “rich” stateful subscriptions
- **configurable dispatchers**



GenStage behaviour

Stages are computation steps that send and/or receive data from other stages.

When a stage sends data, it acts as a producer. When it receives data, it acts as a consumer. Stages can take both producer and consumer roles at once.

Stage types

Besides taking both producer and consumer roles, a stage may be a sink if it only consumes items or called “sink” if it only consumes items.

For example, imagine the stages below where A sends data to B which then sends data to C.

$[A] \rightarrow [B] \rightarrow [C]$

we conclude that:

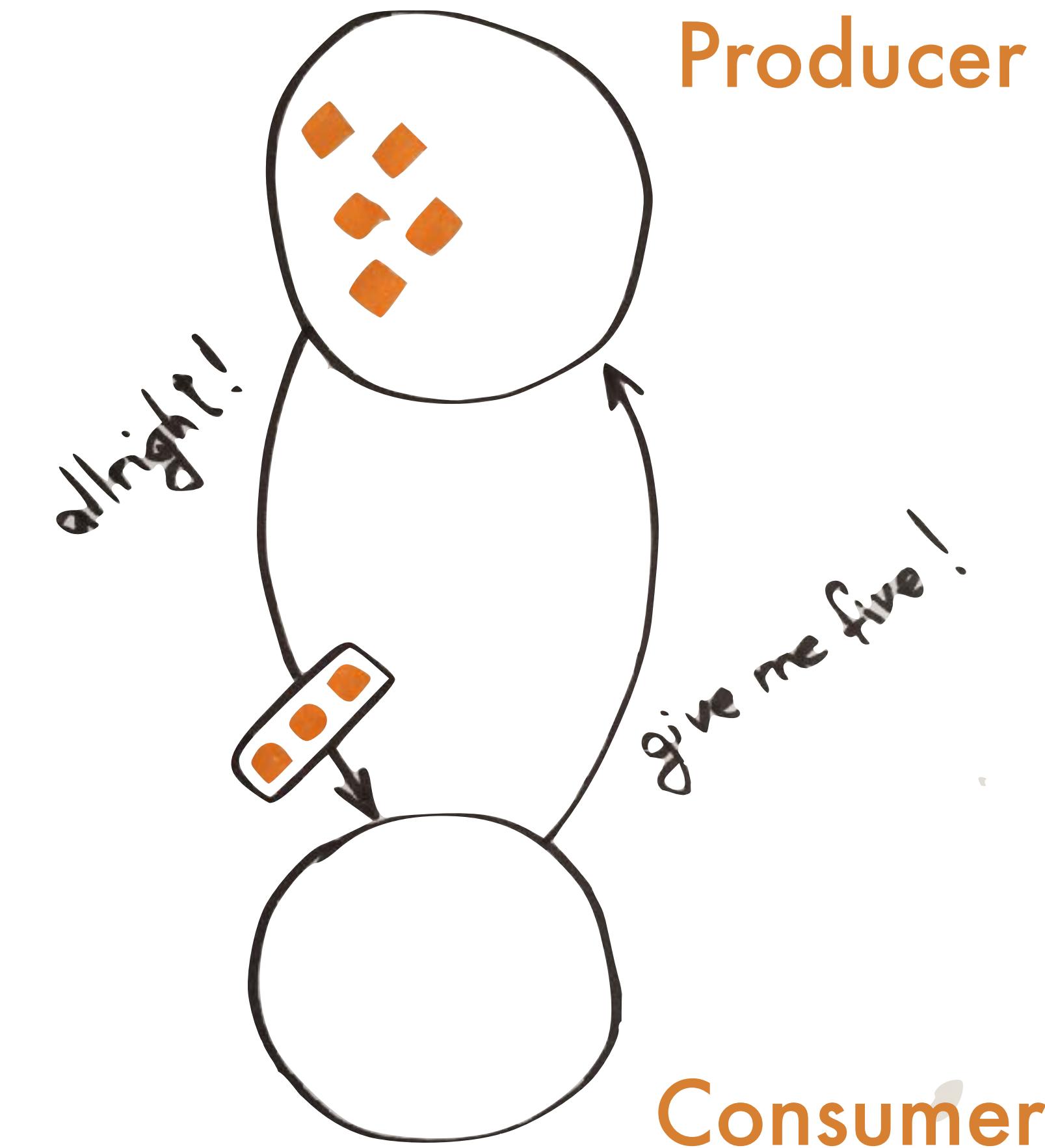
- A is only a producer (and therefore a source)
- B is both producer and consumer
- C is only a consumer (and therefore a sink)

As we will see in the upcoming Examples section, we must specify which stage type to implement for each of them.

To start the flow of events, we subscribe consumers to producers. Once a connection between them is established, consumers will ask the producers for data. Producers will then send data upstream. Once demand arrives, the producer will send more items than the consumer asked for. This provides a back-pressure mechanism for the consumer.

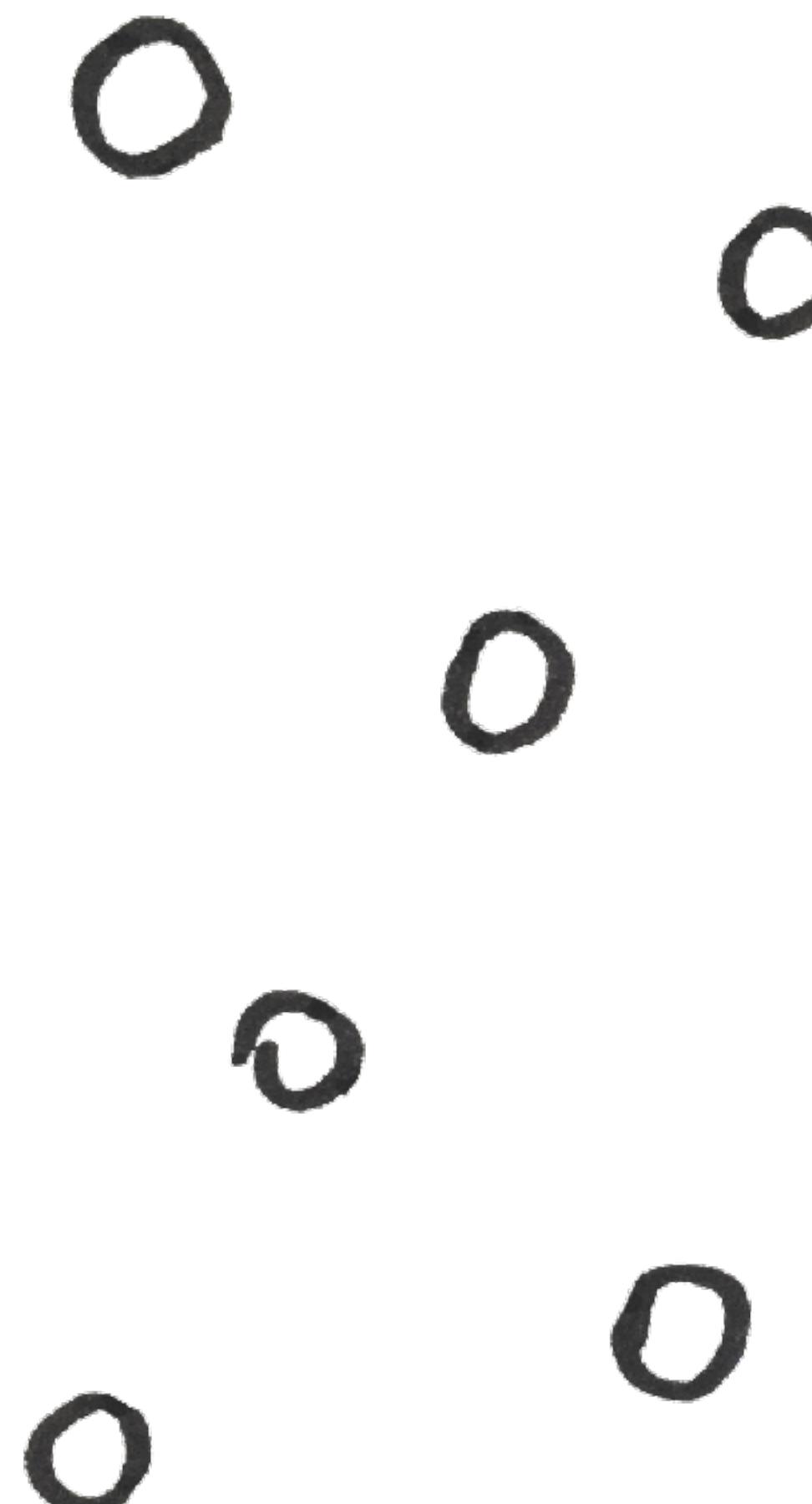
Take II: GenStage

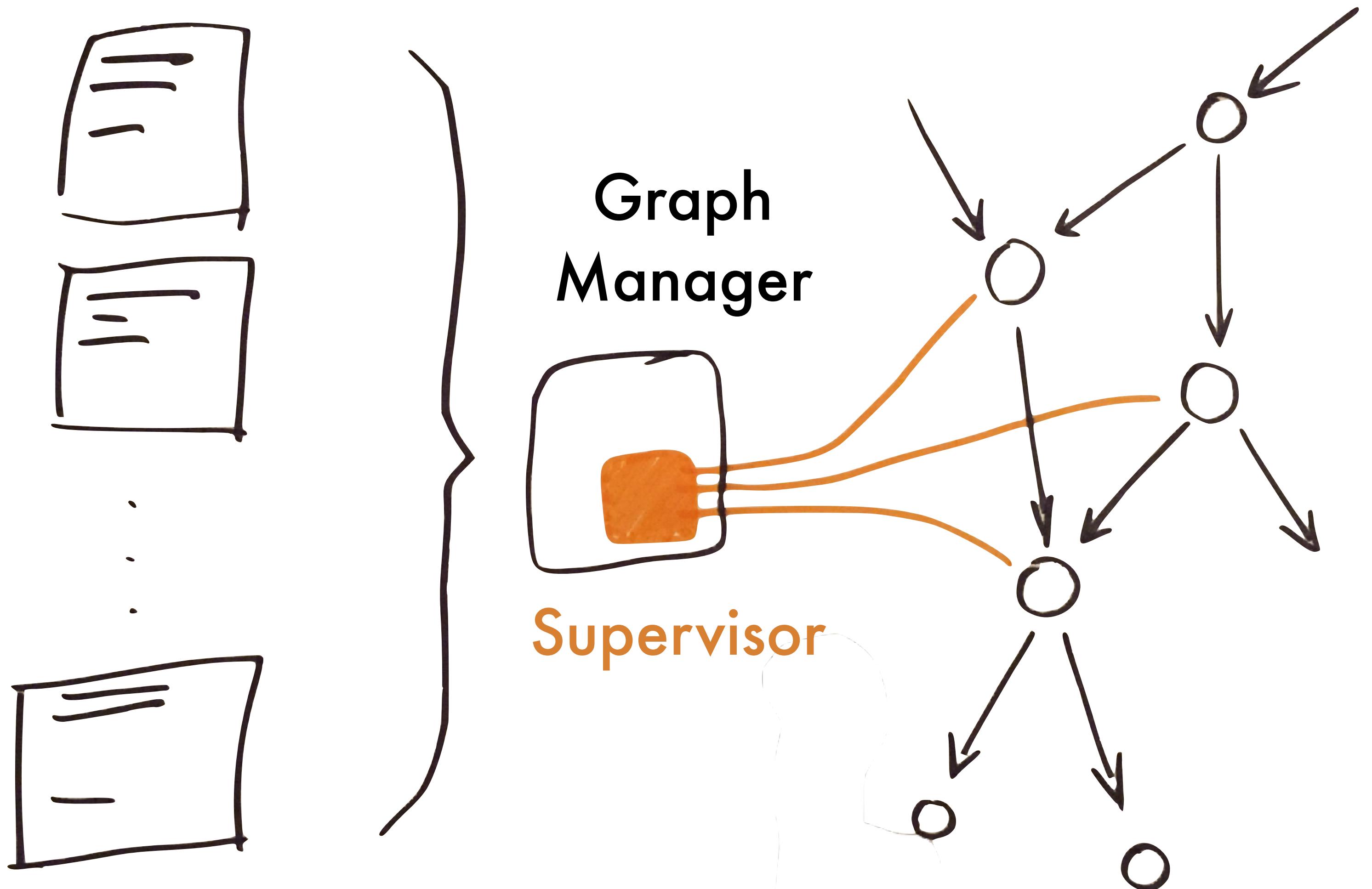
- $[A] \rightarrow [B] \rightarrow [C]$
- “thin” layer on top of GenServer
- producers, consumers and producer-consumers
- demand-based batch-processing pipelines
- “rich” stateful subscriptions
- configurable dispatchers



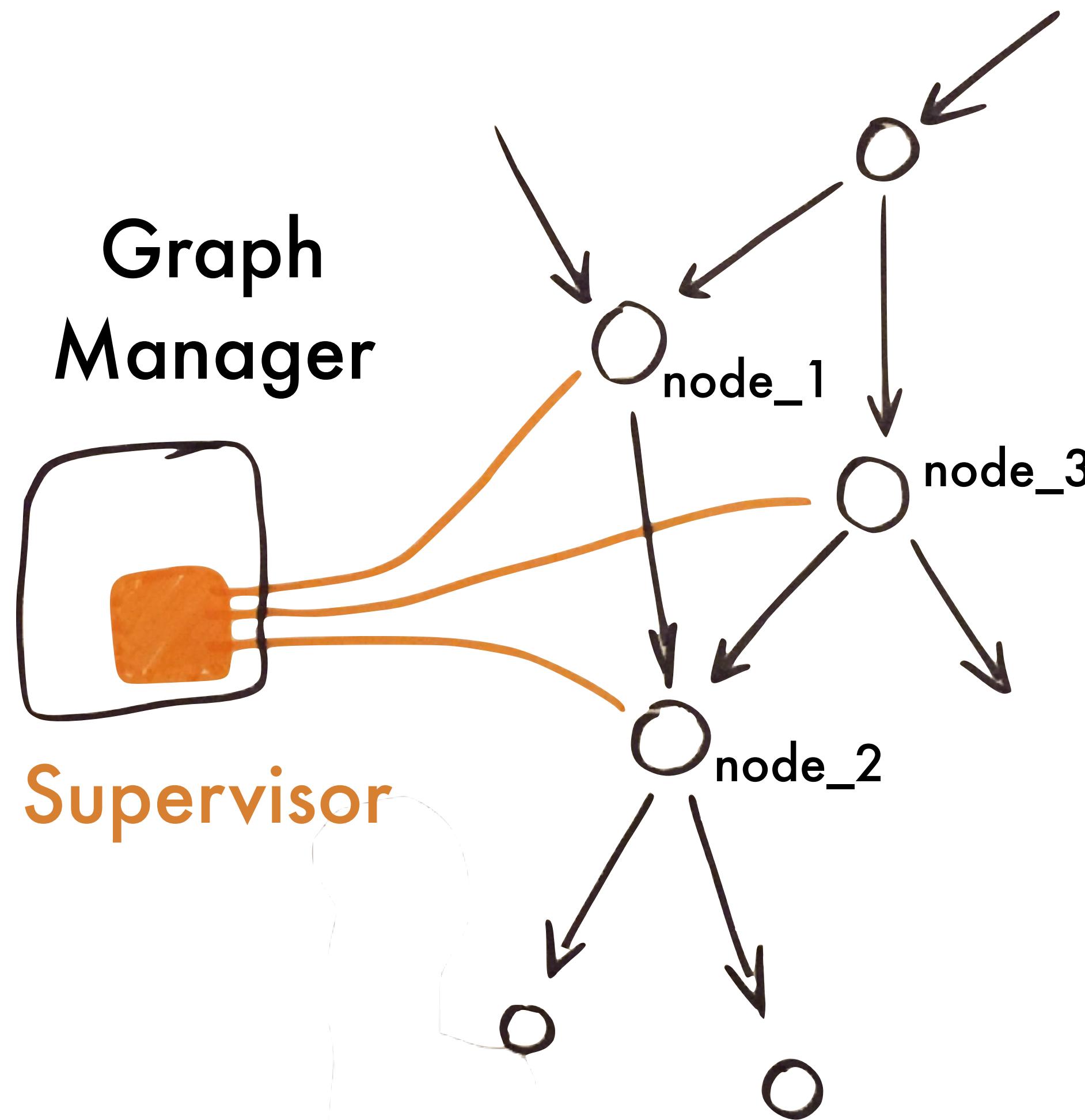
Take II: GenStage

- $[A] \rightarrow [B] \rightarrow [C]$
- “thin” layer on top of **GenServer**
- producers, consumers and producer-consumers
- demand-based batch-processing pipelines
- “rich” stateful subscriptions
- configurable dispatchers

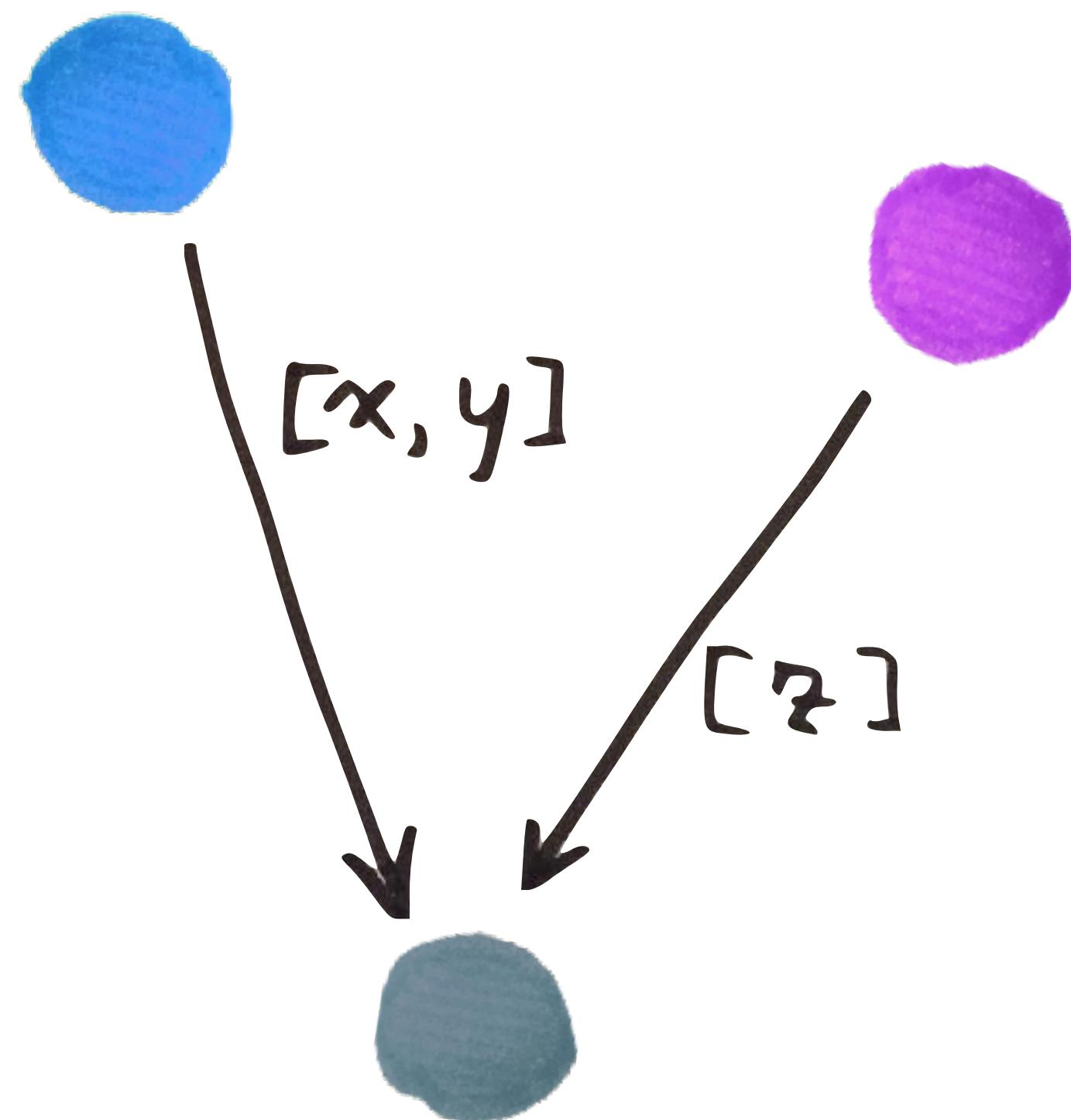




```
"node_1":  
  type: "value"  
  value: 12  
  
"node_2":  
  type: "code-cell"  
  body: "$(node_1) + $(node_3)"  
  lang: "julia"  
  dependencies:  
    - id: "node_1"  
    - id: "node_3"  
    refs: []  
  
"node_3":  
  type: "code-cell"  
  body: "3"  
  lang: "python"
```



Requiring Exports



● blue

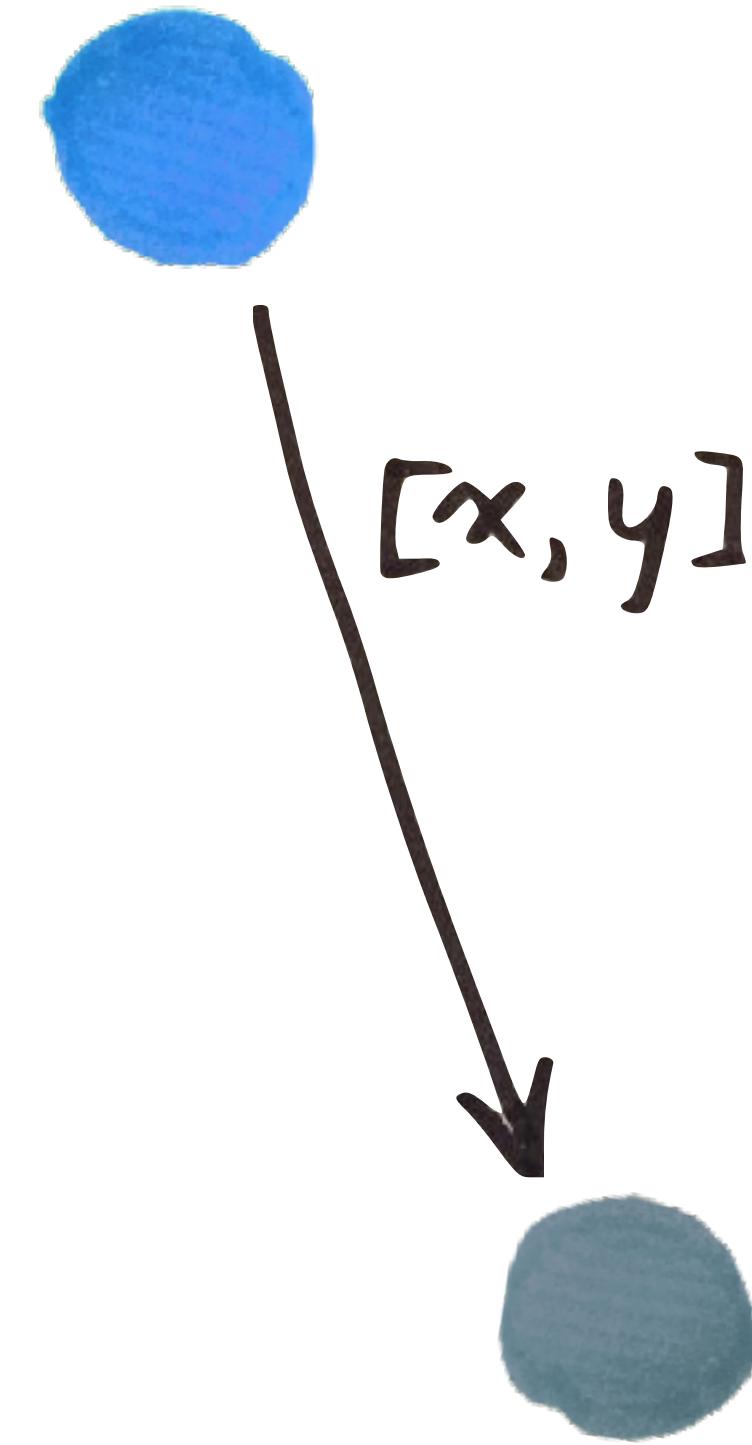
```
x = 1  
y = 2
```

● purple

```
z = [3, 4]
```

● bluegray

```
out <- c(blue.x, blue.y) + purple.z
```



```
GenStage.async_subscribe(grey,  
  to: blue,  
  refs: [x, y])
```

```
handle_subscribe(:consumer,  
  [refs: [x, y]], ..., state)
```

```
handle_subscribe(:producer,  
  [refs: [x, y]], ..., state)
```

PRODUCER-CONSUMERS

```
def handle_events(incoming_events, from, state) do
    # ... TRANSFORM EVENTS ...
    {:noreply, outgoing_events, state}
end
```

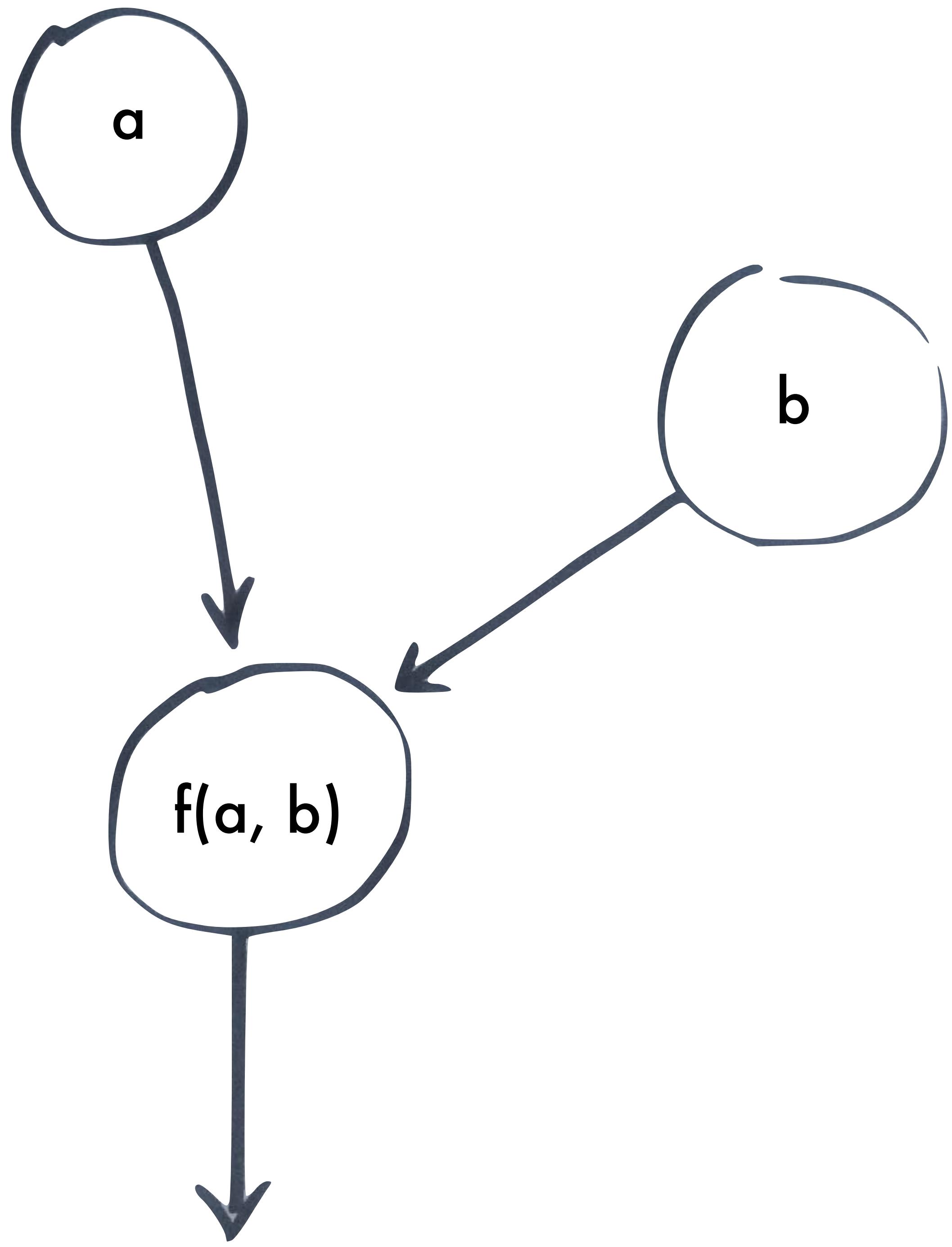
GEN-SERVER CALLBACKS

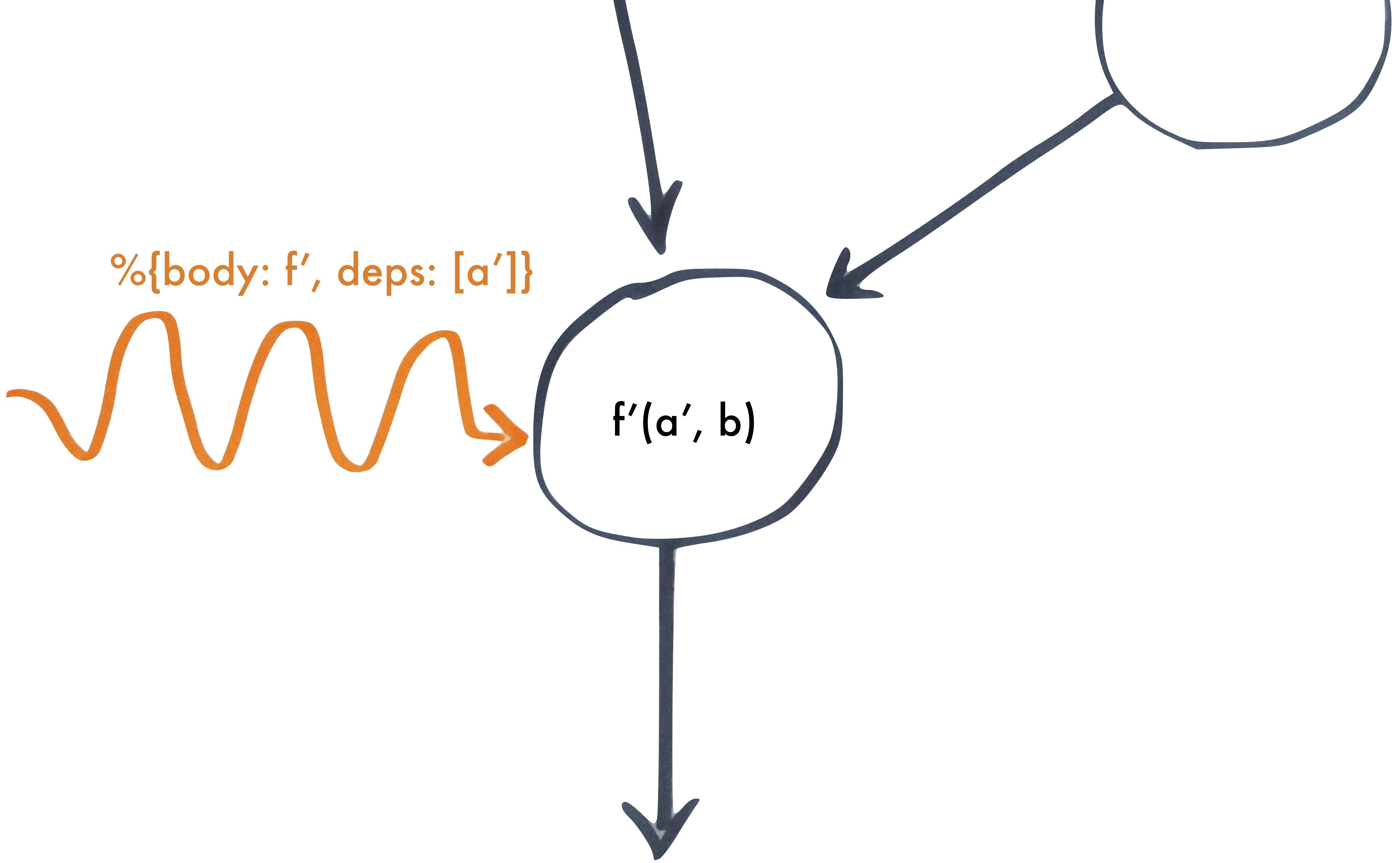
```
def handle_cast(msg, state) do
```

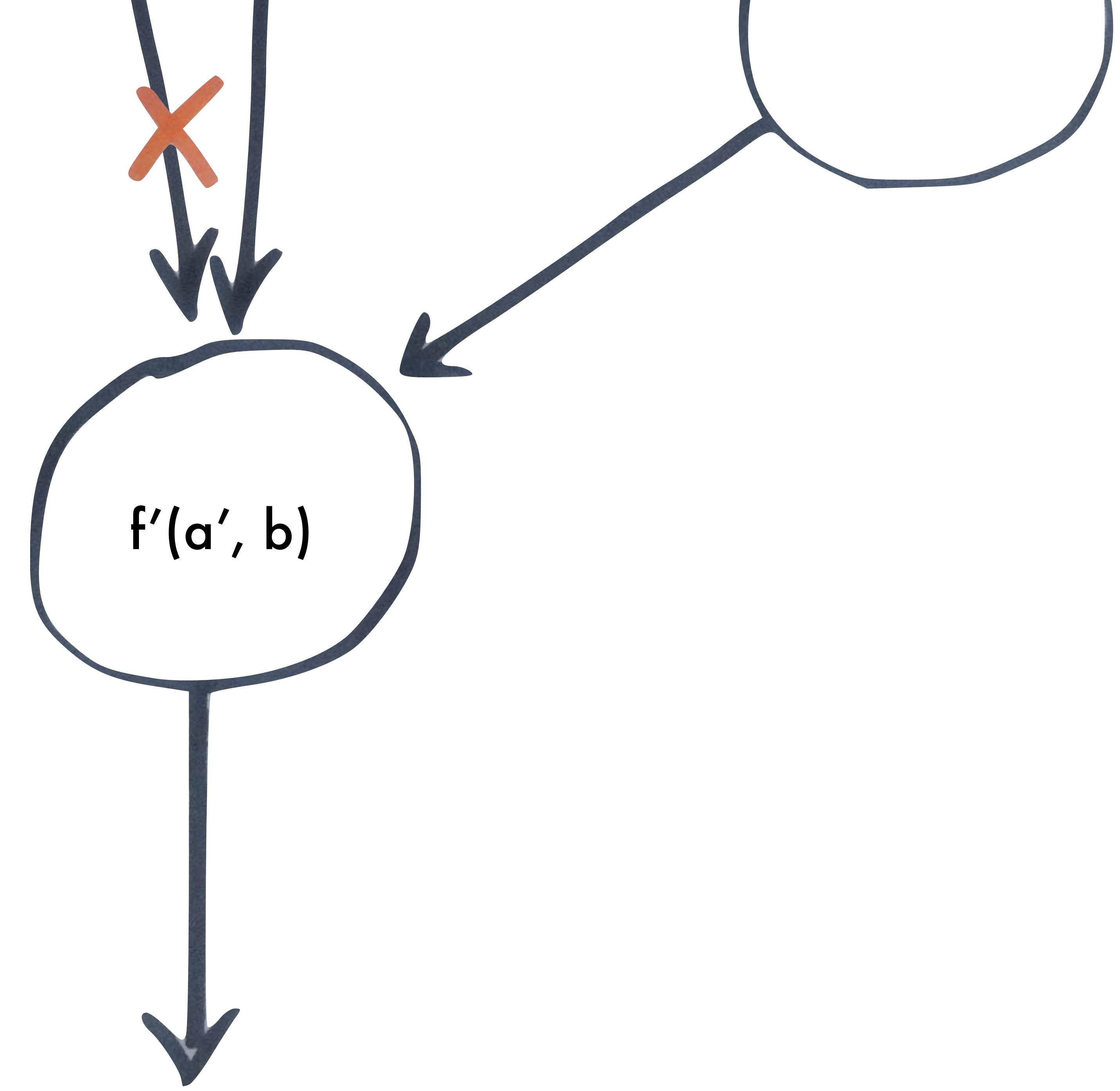
... PRODUCE EVENTS ...

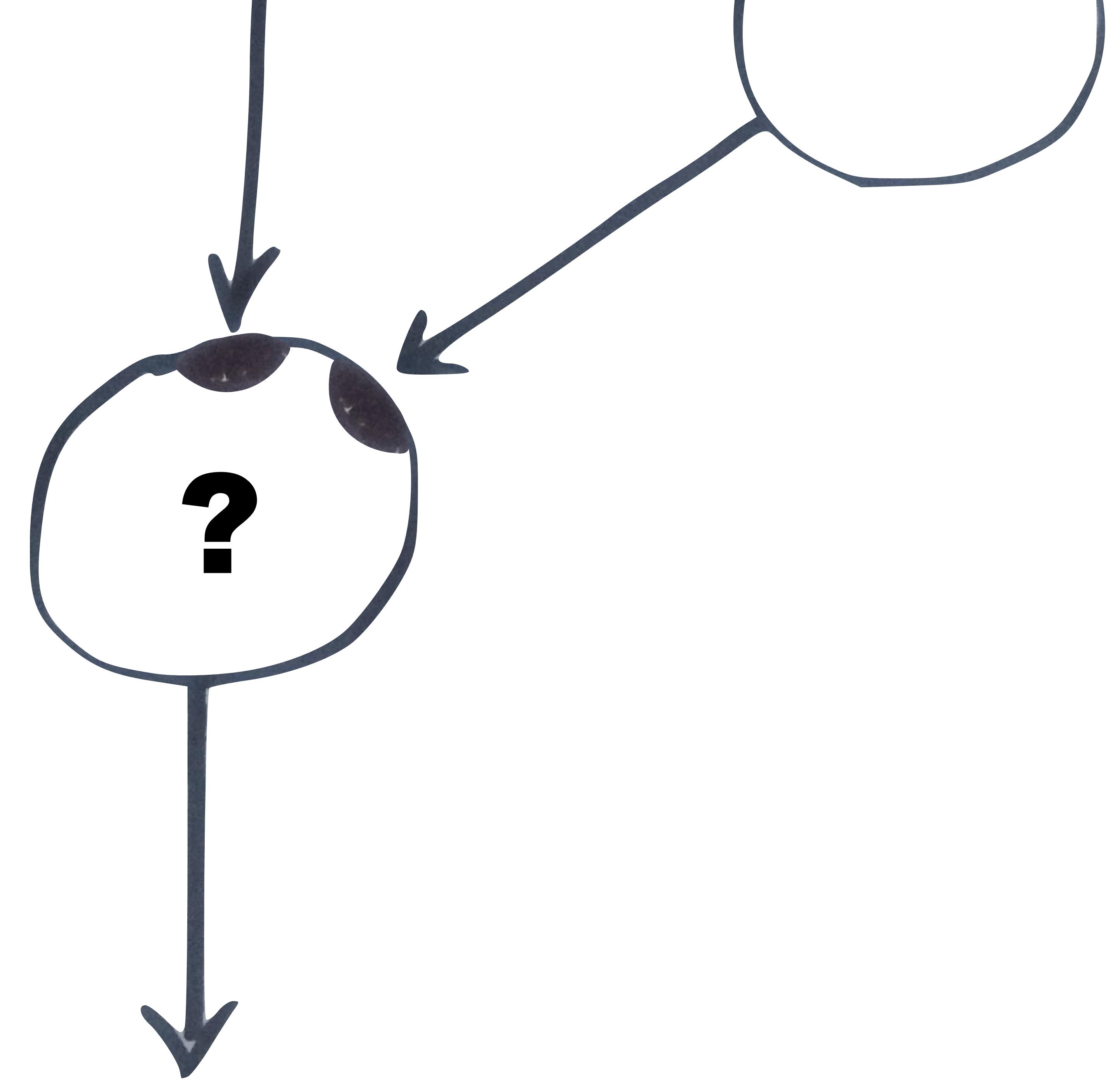
```
    {:noreply, outgoing_events, state}
end
```

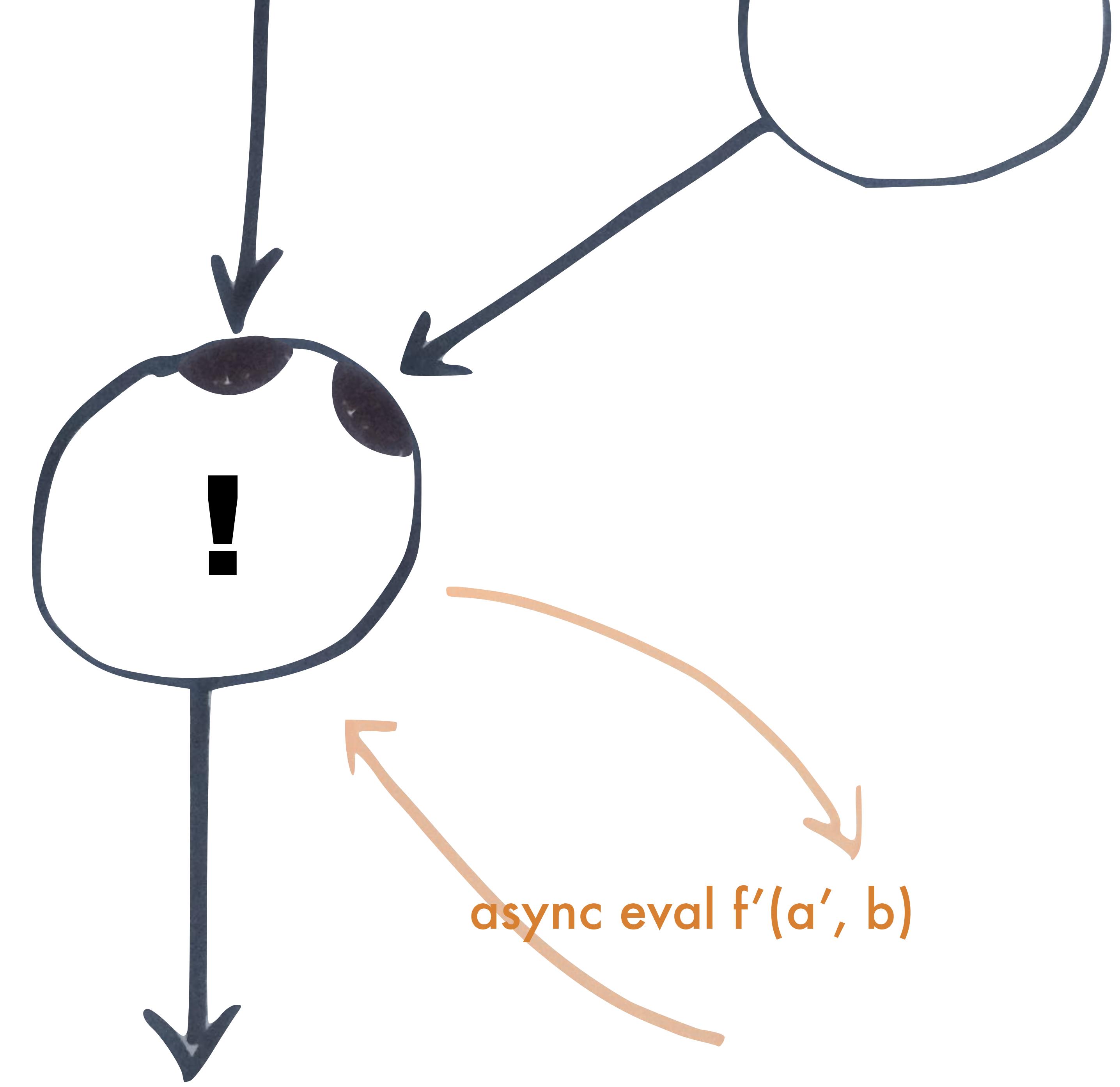
SAME FOR CALLS, INFO, ETC...



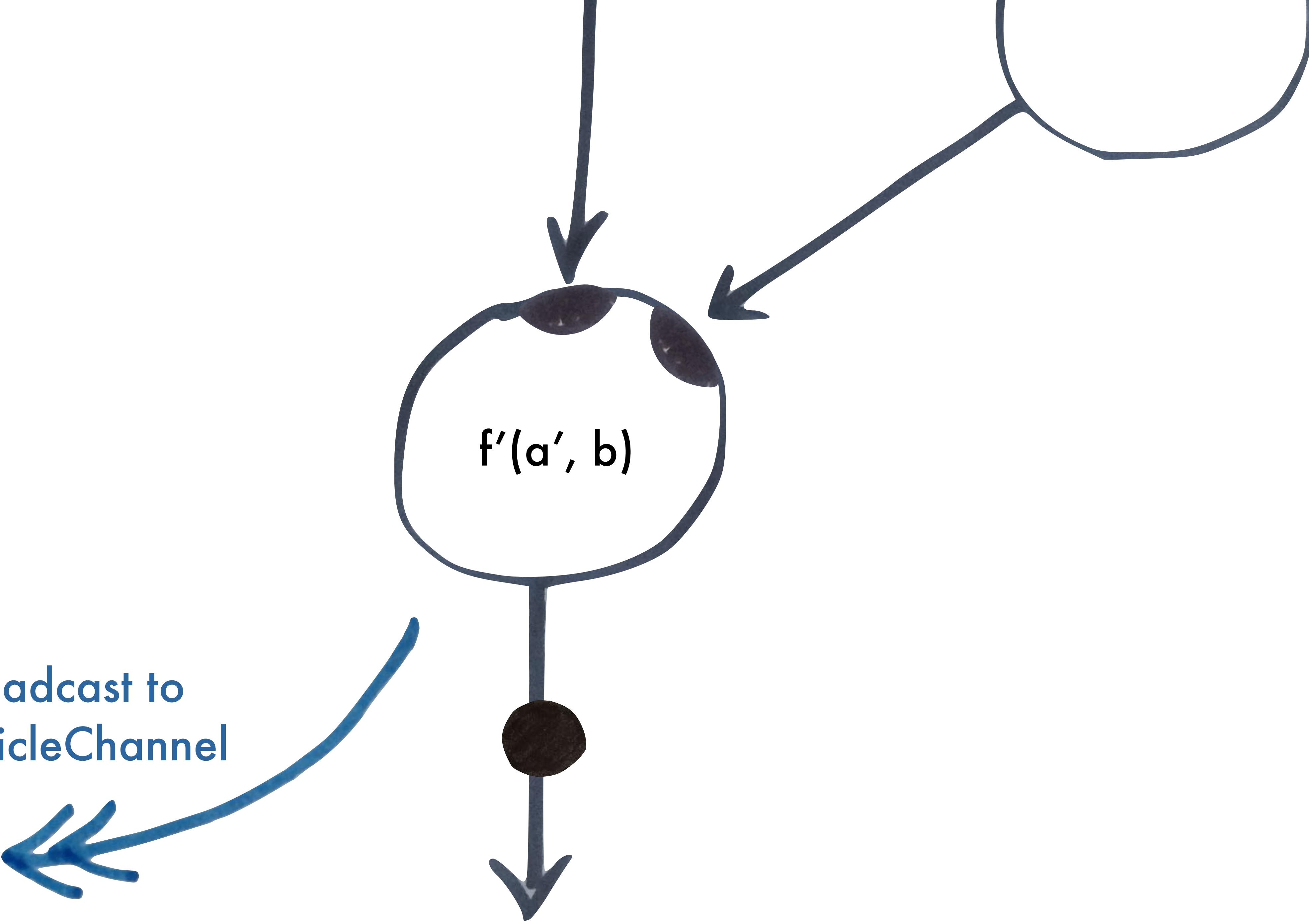




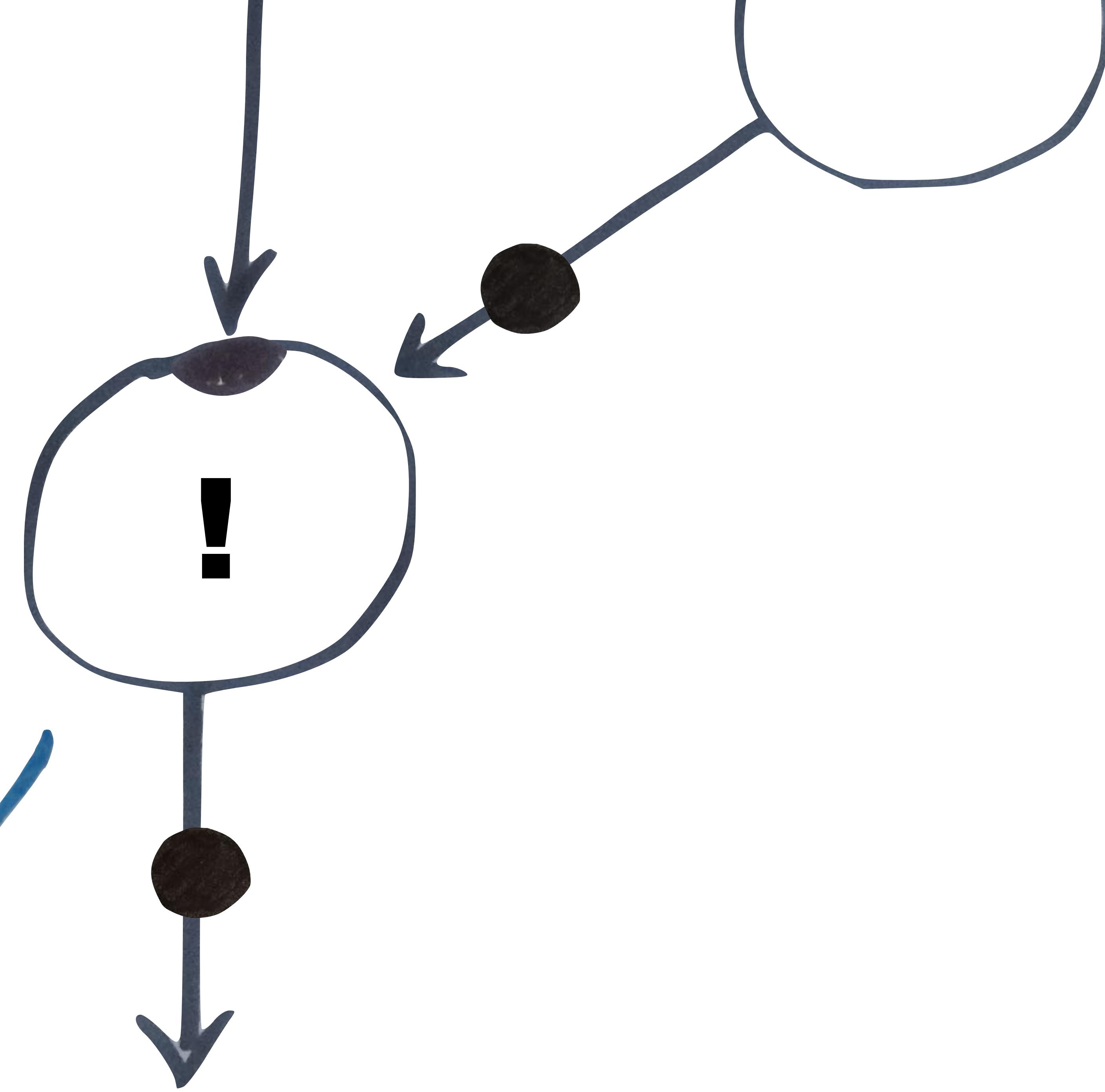


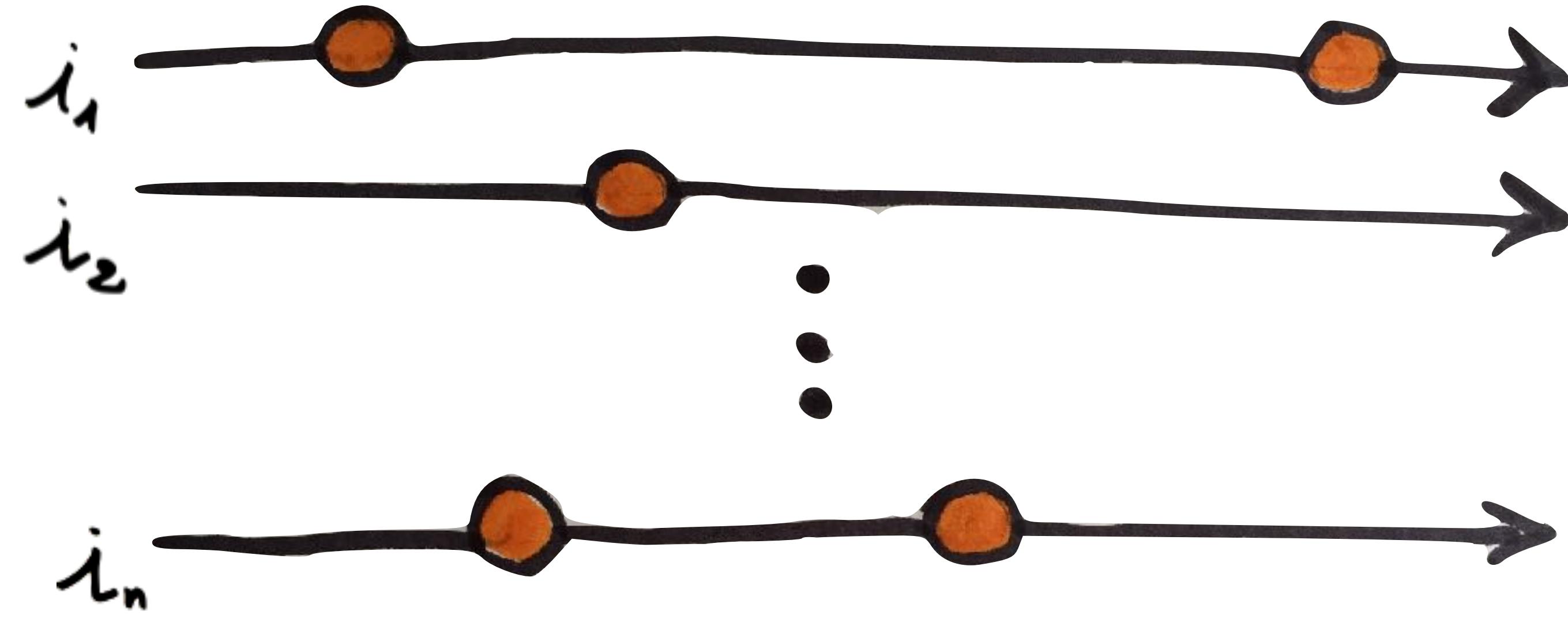


broadcast to
ArticleChannel



broadcast to
ArticleChannel





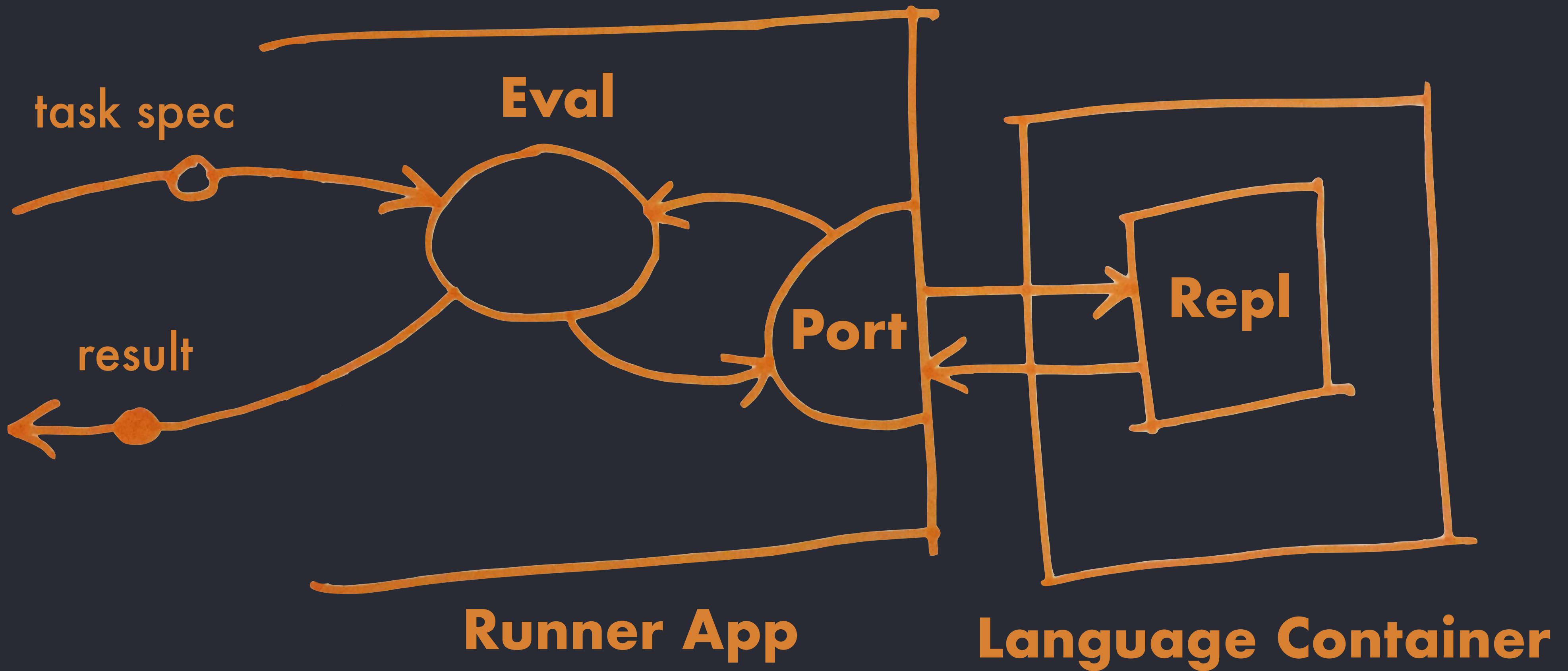
combineLatest (i_1, \dots, i_n) => eval(code; i_1, \dots, i_n)



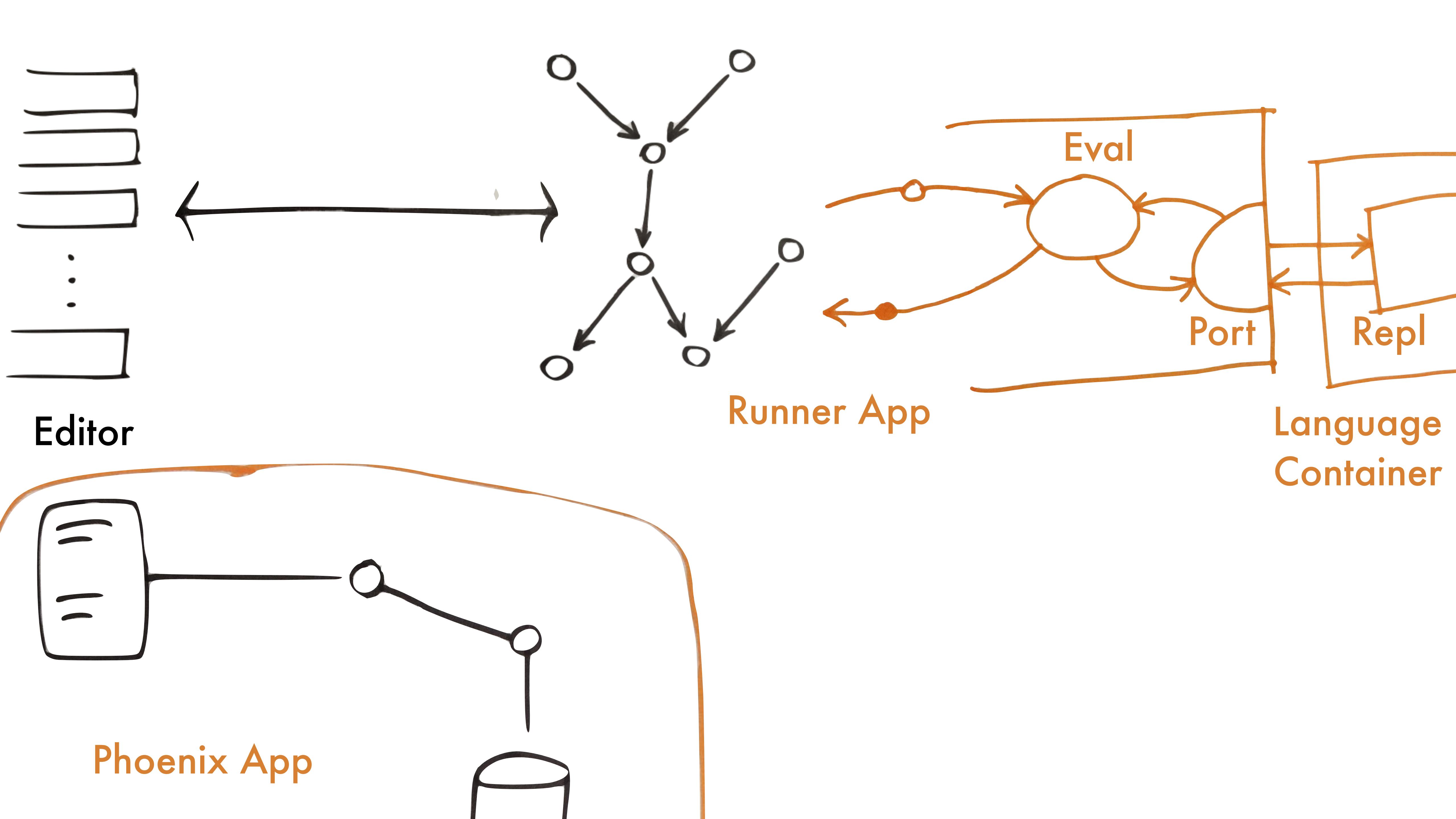
```
def handle_events( [%Result{} = res], _from, %CodeCell{}) do
  # ...
  {state, errors} =
    |> add_result(state, res)
    |> trigger?
    |> async_schedule_task()

  {:noreply, errors, state}
end
```

```
def handle_cast(%Result{id: id} = res, %CodeCell{id: id}) do
  # ...
  broadcast(state.topic, "result", res)
  {:noreply, [res], state}
end
```



```
:erlang.open_port(  
  {:_spawn_executable, 'docker'},  
  args: ['run', '-i', docker_image, command] )
```



Remarks & Thanks

nextjournal

@usenextjournal

see you at CurryOn, Barcelona, June 19-20th