ElixirConf EU

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

Andrea Amantini

lo_zampino

zampino

nextjournal

ElixirConf EU

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

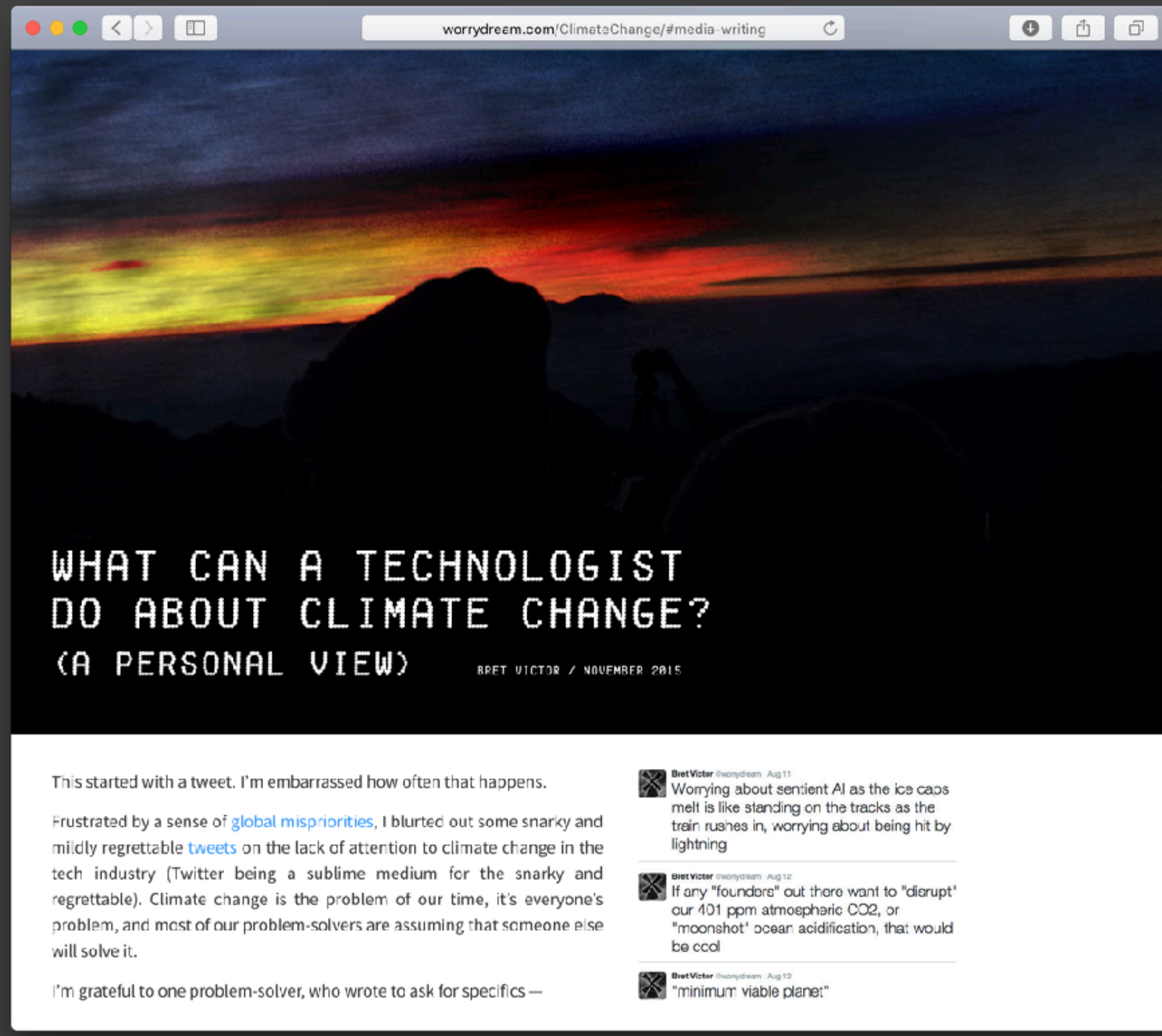
“Imagine an authoring tool designed for *arguing from evidence*…

…literally *autocompleting* sourced facts directly into the document…

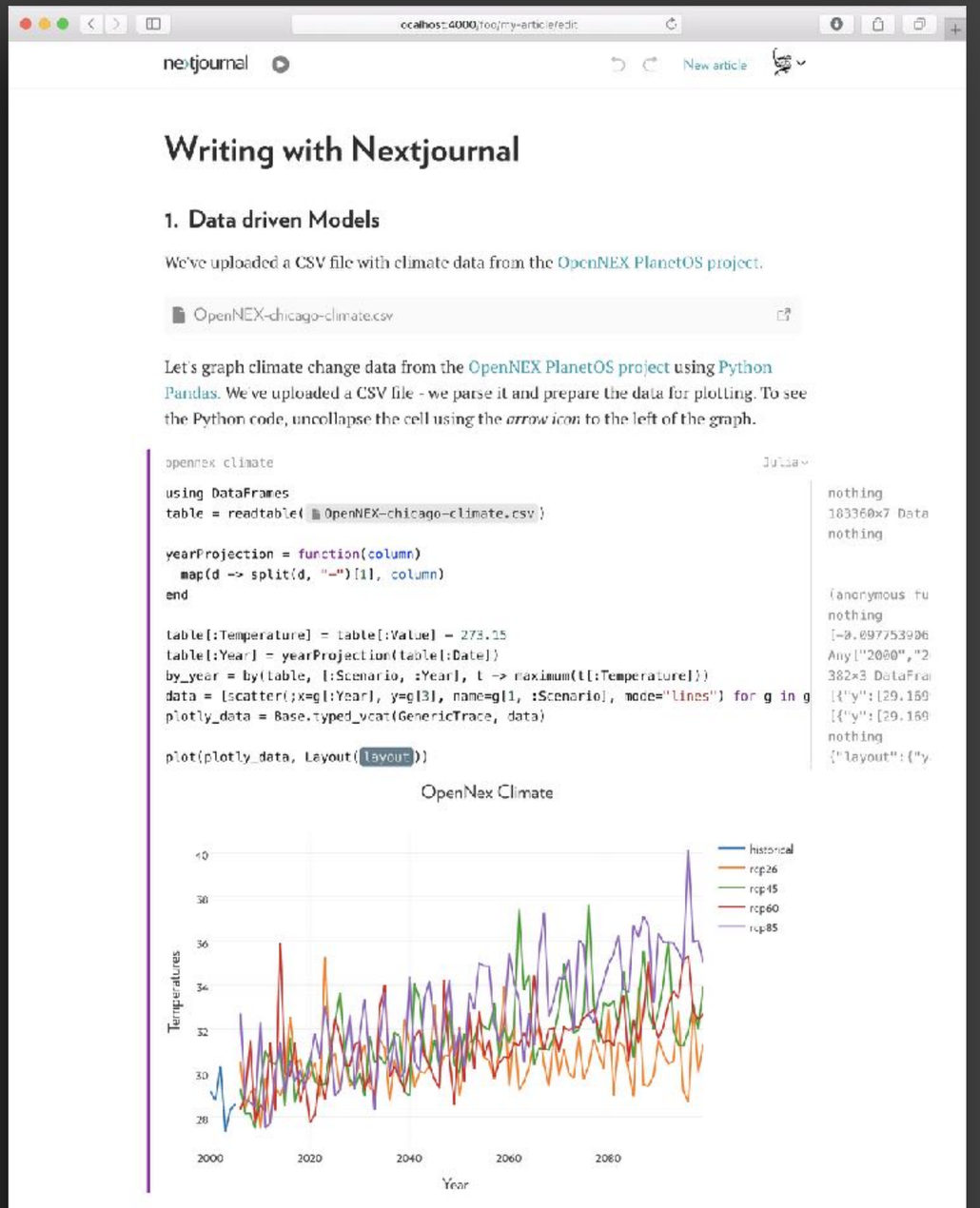
…what if it were as easy to insert facts, data, and models as it is to insert emoji and cat photos?”

—Bret Victor

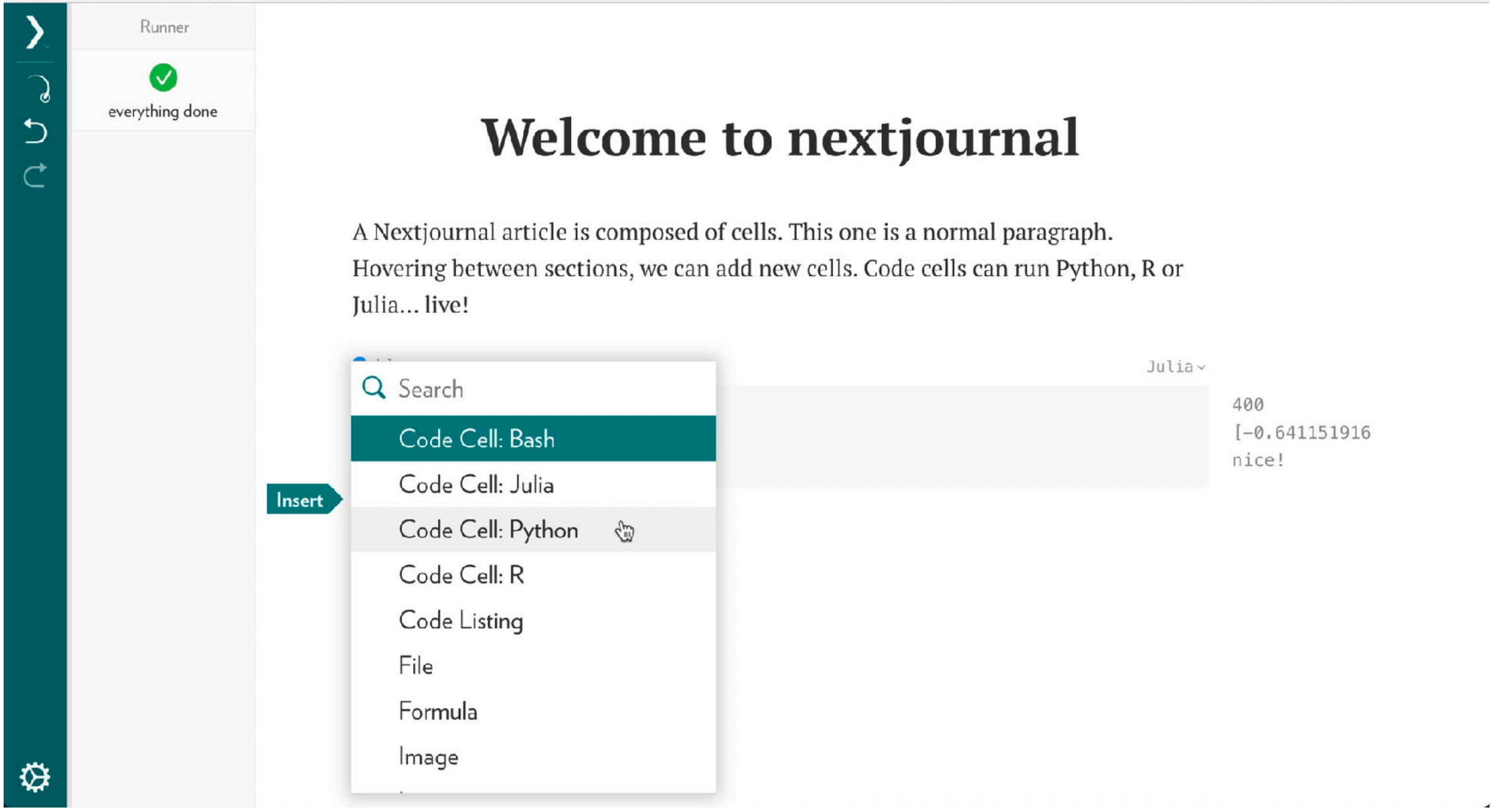
worrydream.com/ClimateChange



nextjournal



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!
Julia
400
[-0.641151916
nice!
Insert
Search
Code Cell: Bash
Code Cell: Julia
Code Cell: Python
Code Cell: R
Code Listing
File
Formula
Image

Runner

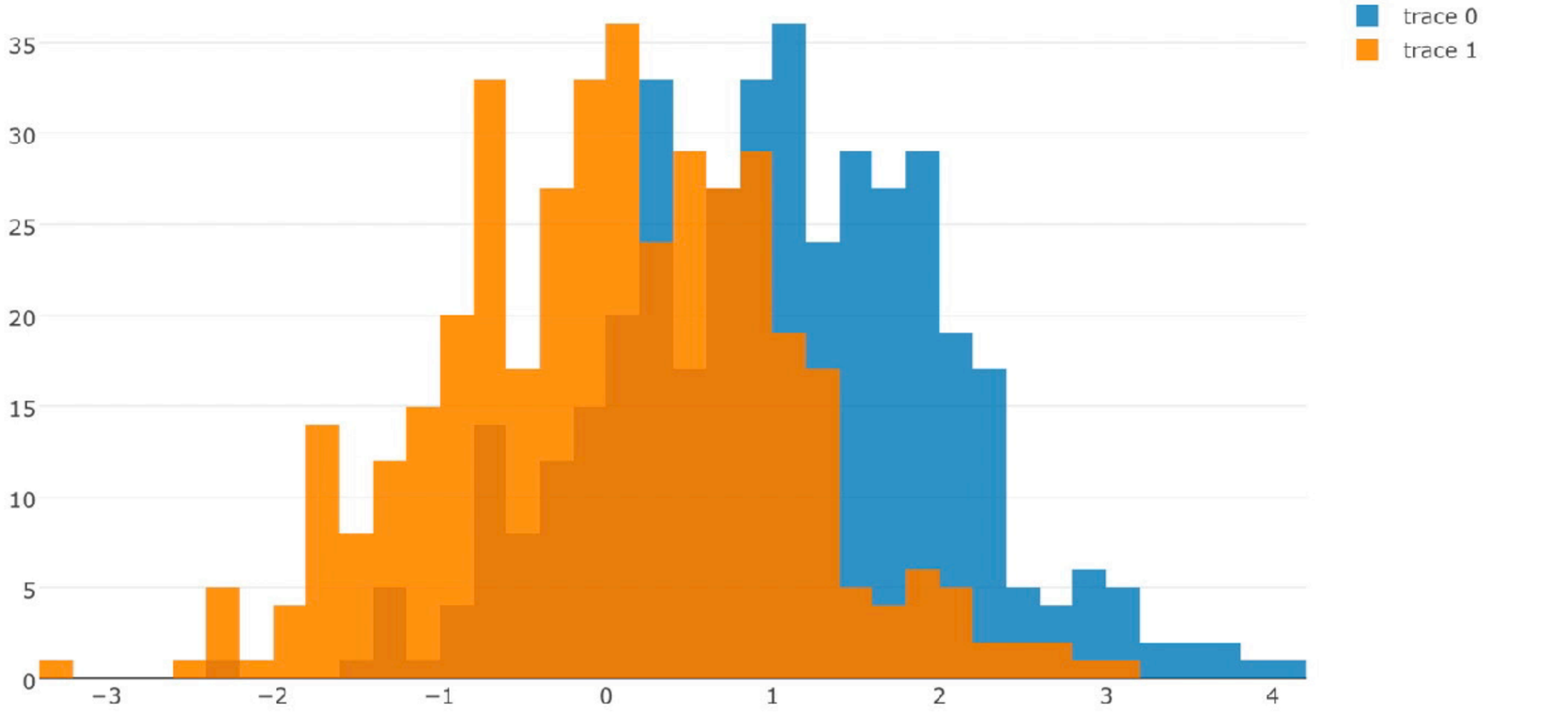
everything done

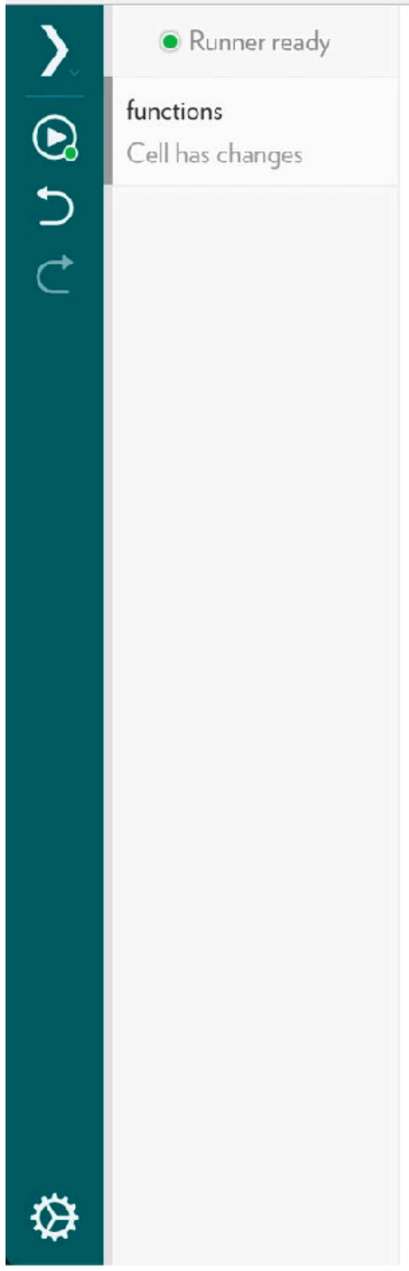
purple

R

```
plot_ly() %>%
  add_trace(x=cyan, type='histogram', opacity='.8') %>%
  add_trace(x=blue.random_list, type='histogram', opacity='.8') %>%
  layout(barmode='overlay')
```

list(visdat =





Alanine Dipeptide Analysis

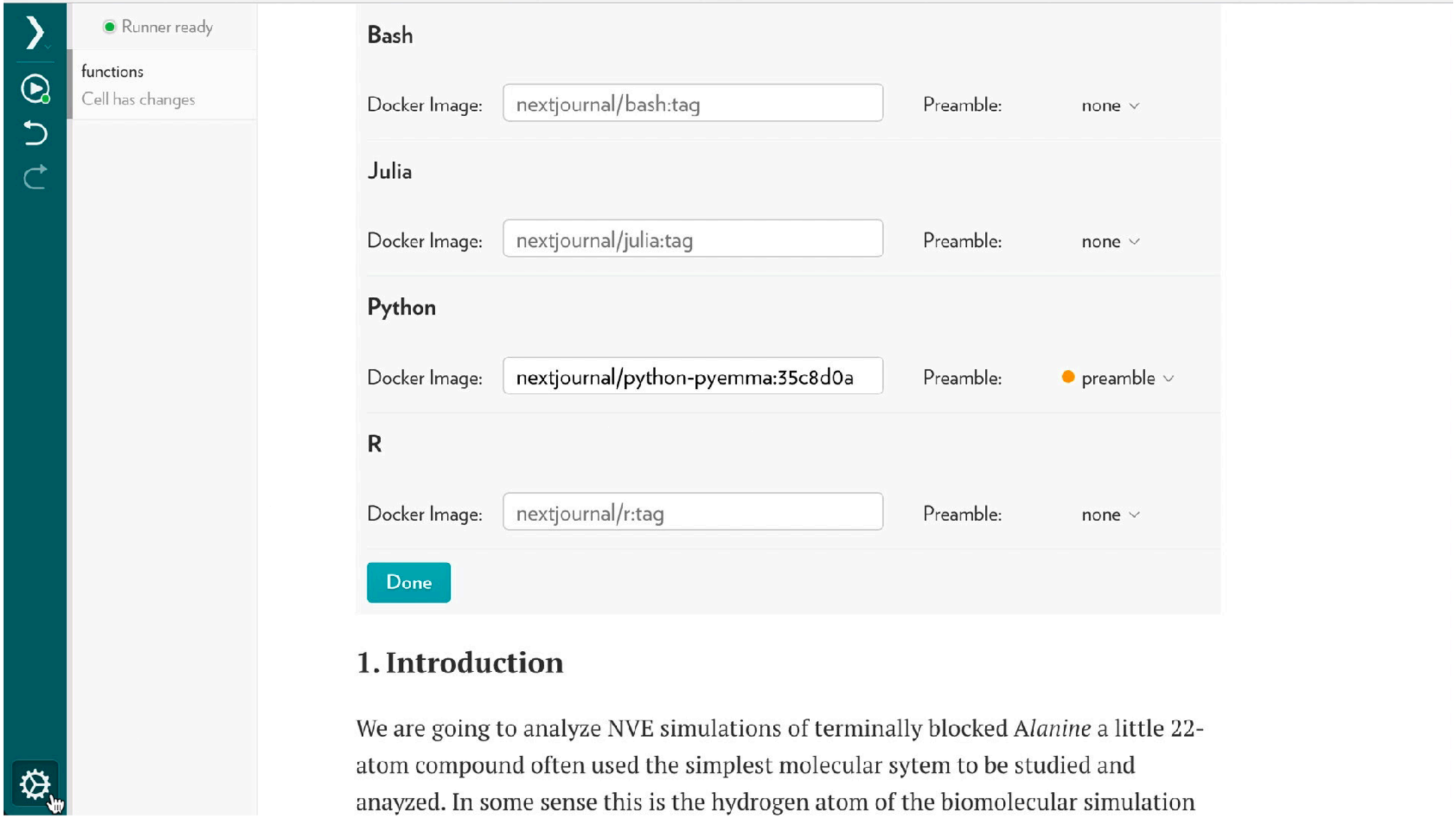
1. Introduction

We are going to analyze NVE simulations of terminally blocked A*lanine* a little 22-atom compound often used the simplest molecular sytem to be studied and anayzed. 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 realilty proteins are build out of chains of the (20 natural occuring) 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**.

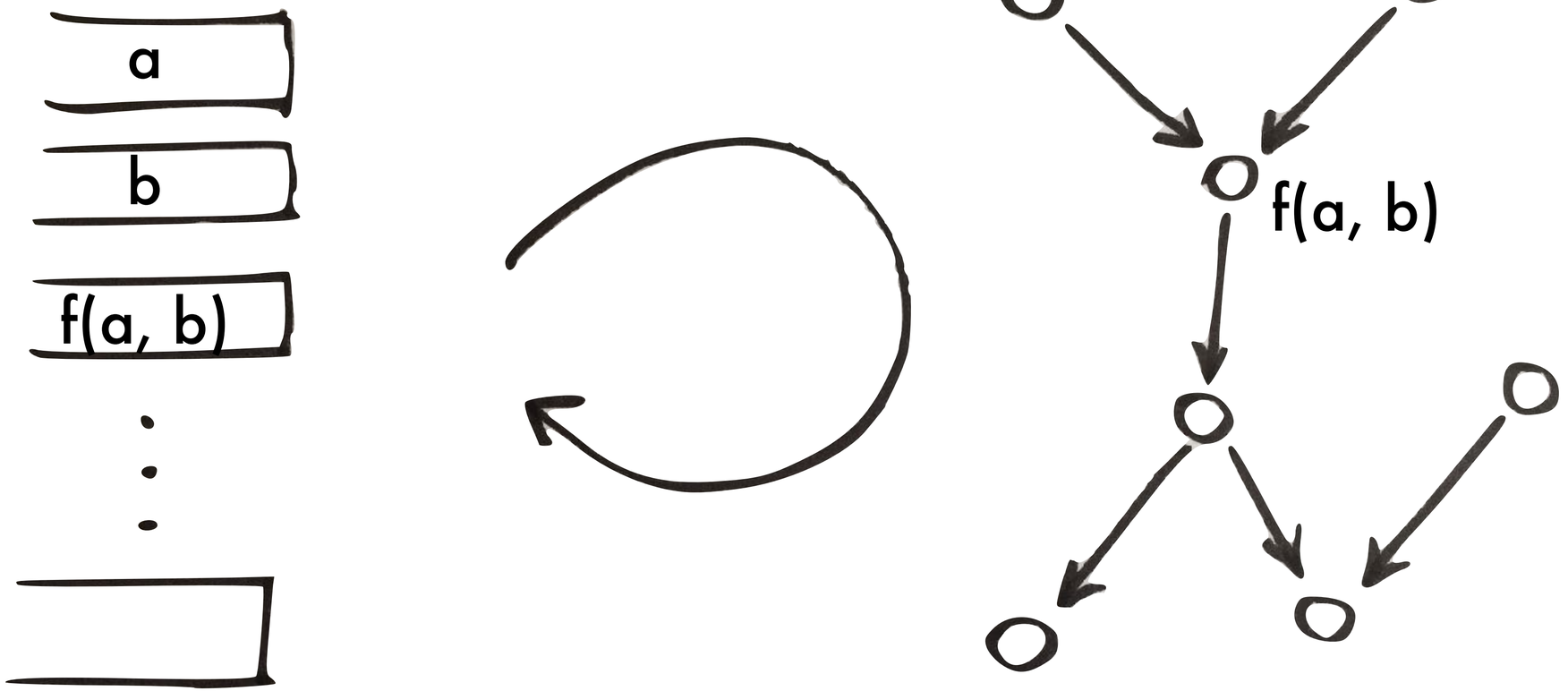
2. File imports

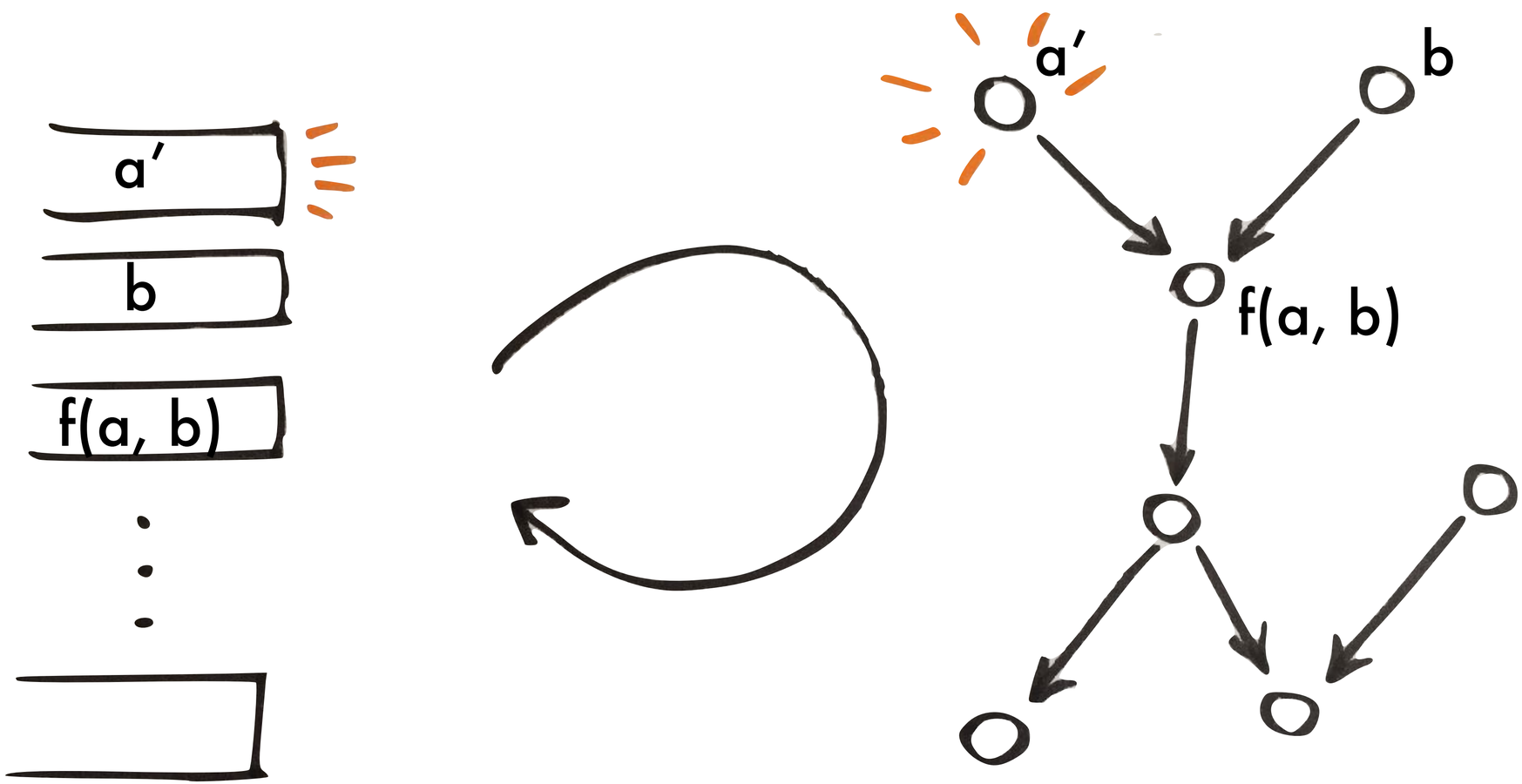
the file uploaded contains the so called replica-exchange trajectories

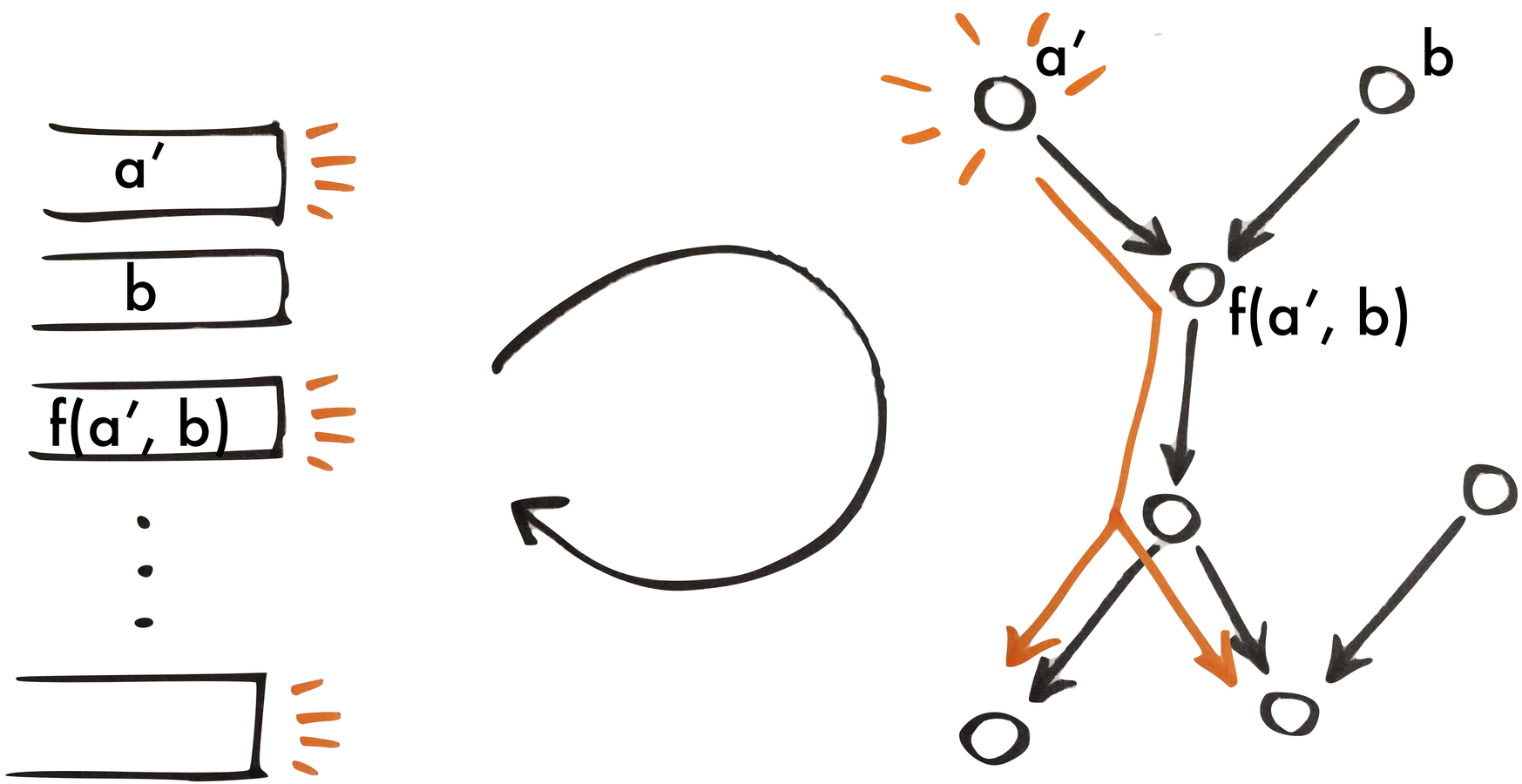


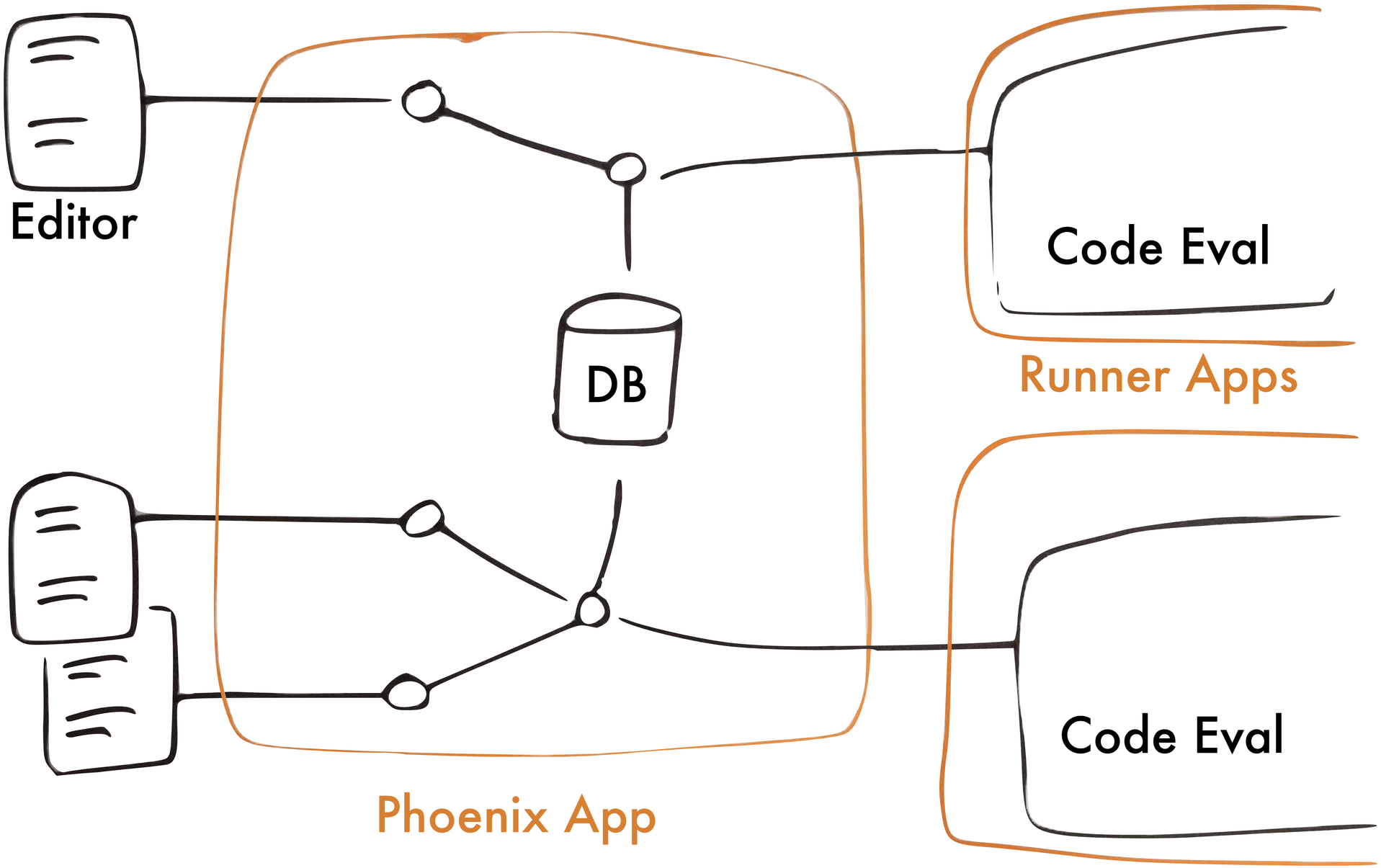
1. Introduction

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

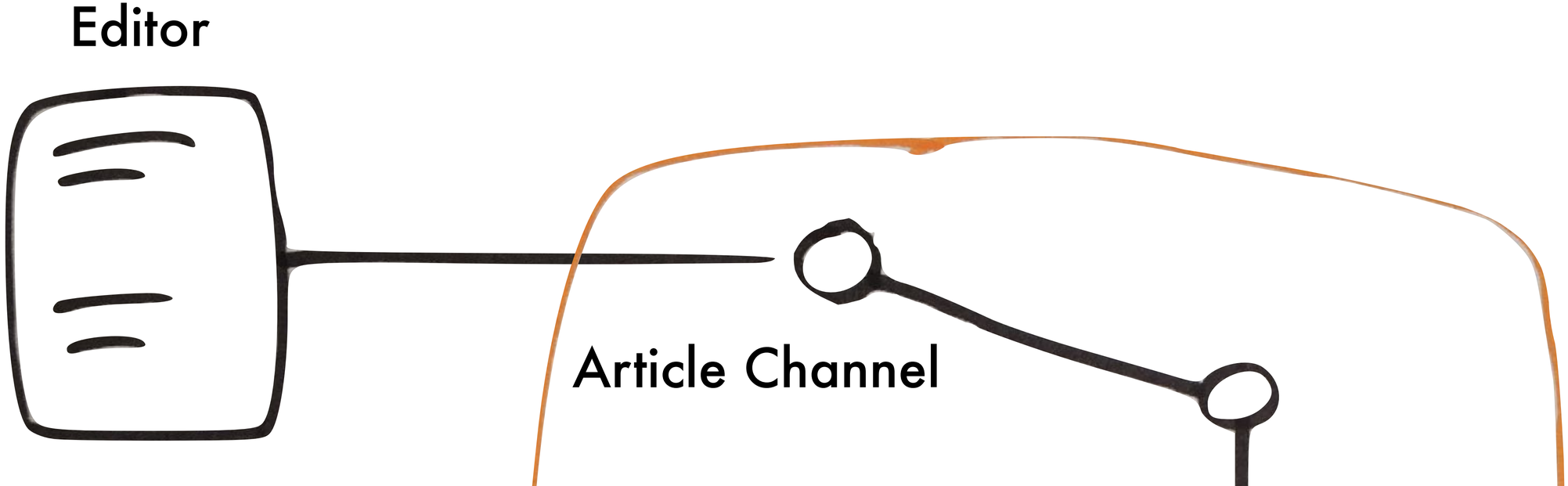

a
b
f(a, b)
a
b
f(a, b)

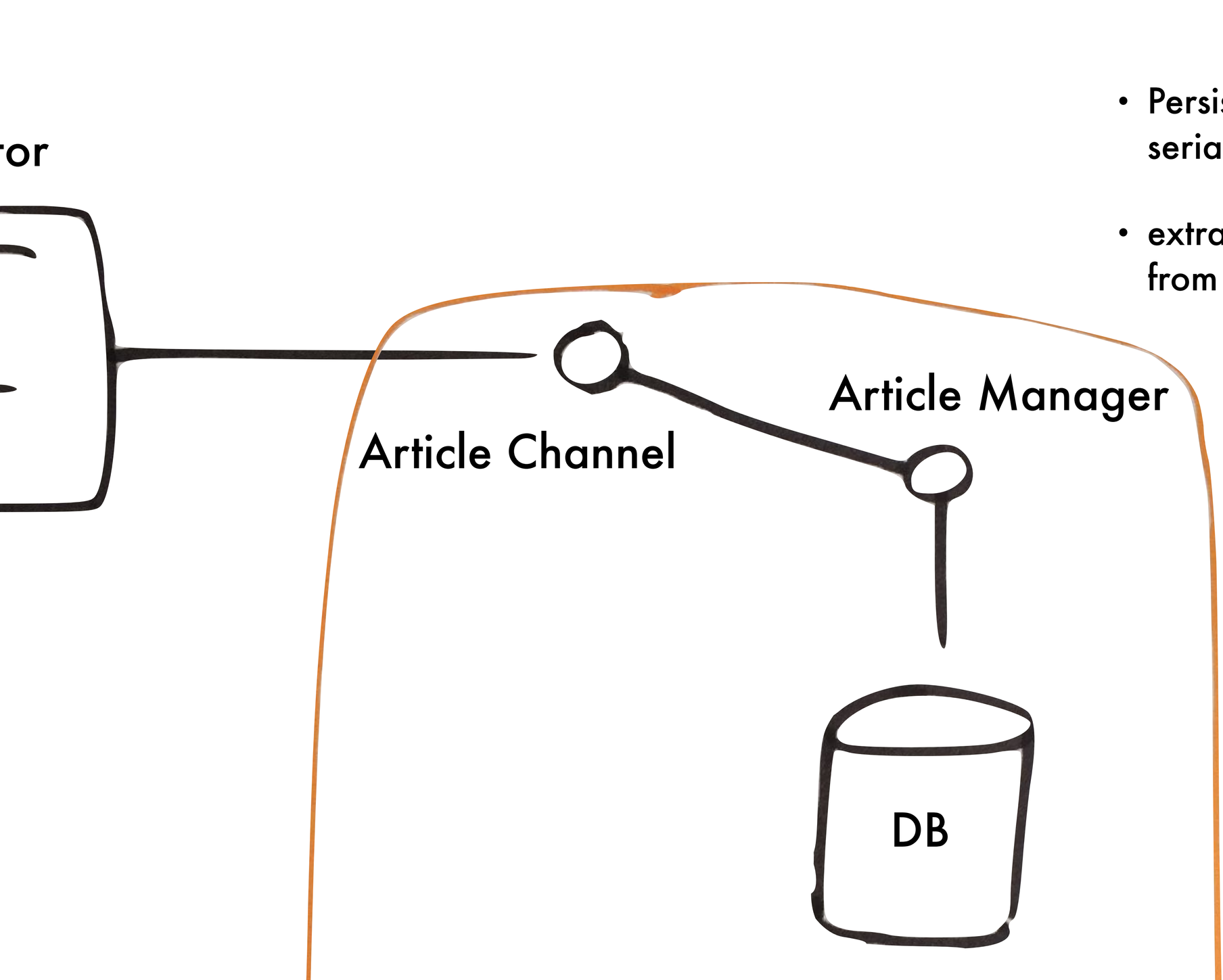

a′
b
f(a, b)
a′
b
f(a, b)


a′
b
f(a′, b)
a′
b
f(a′, b)

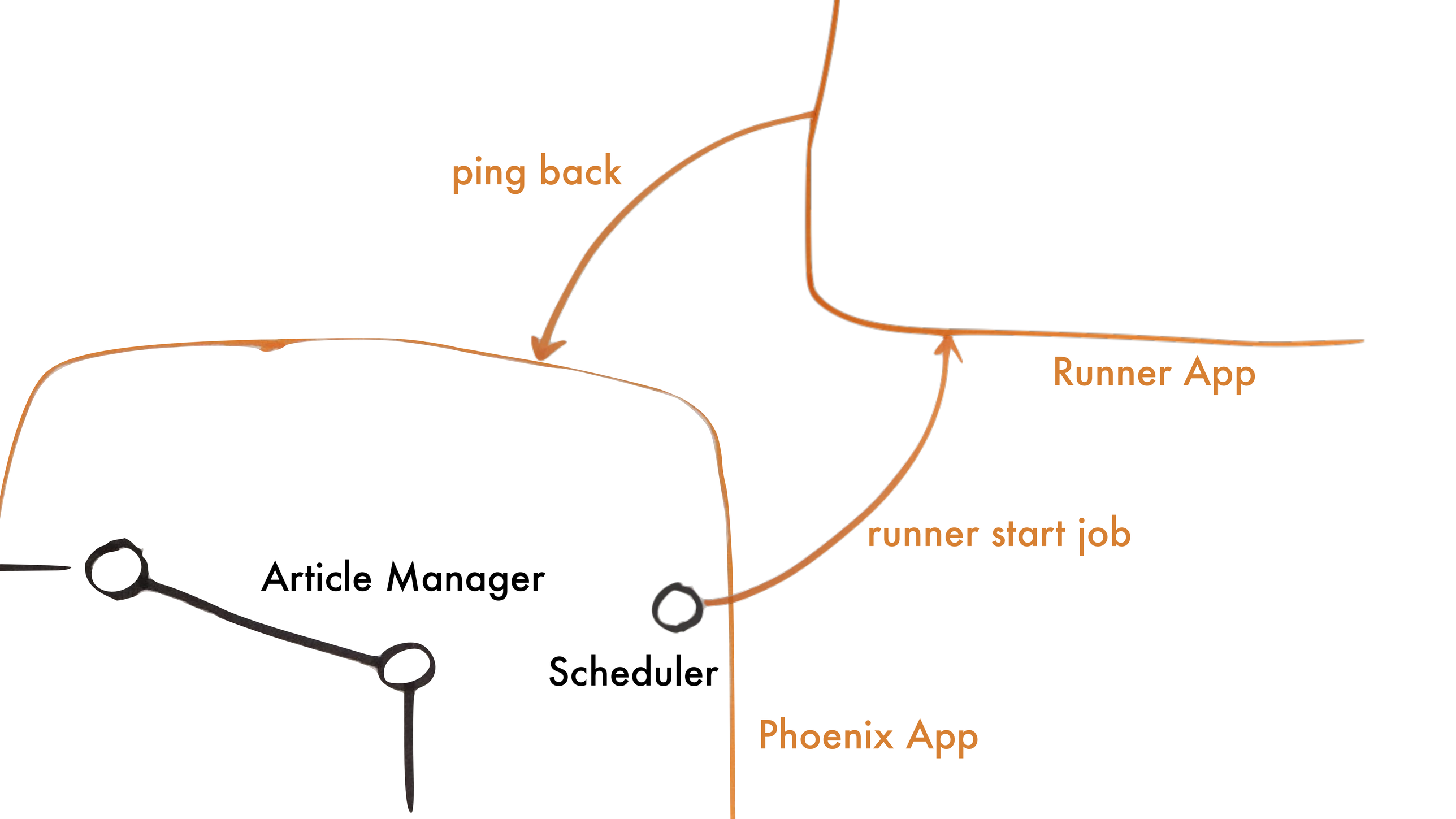

Editor
DB
Code Eval
Runner Apps
Code Eval
Phoenix App

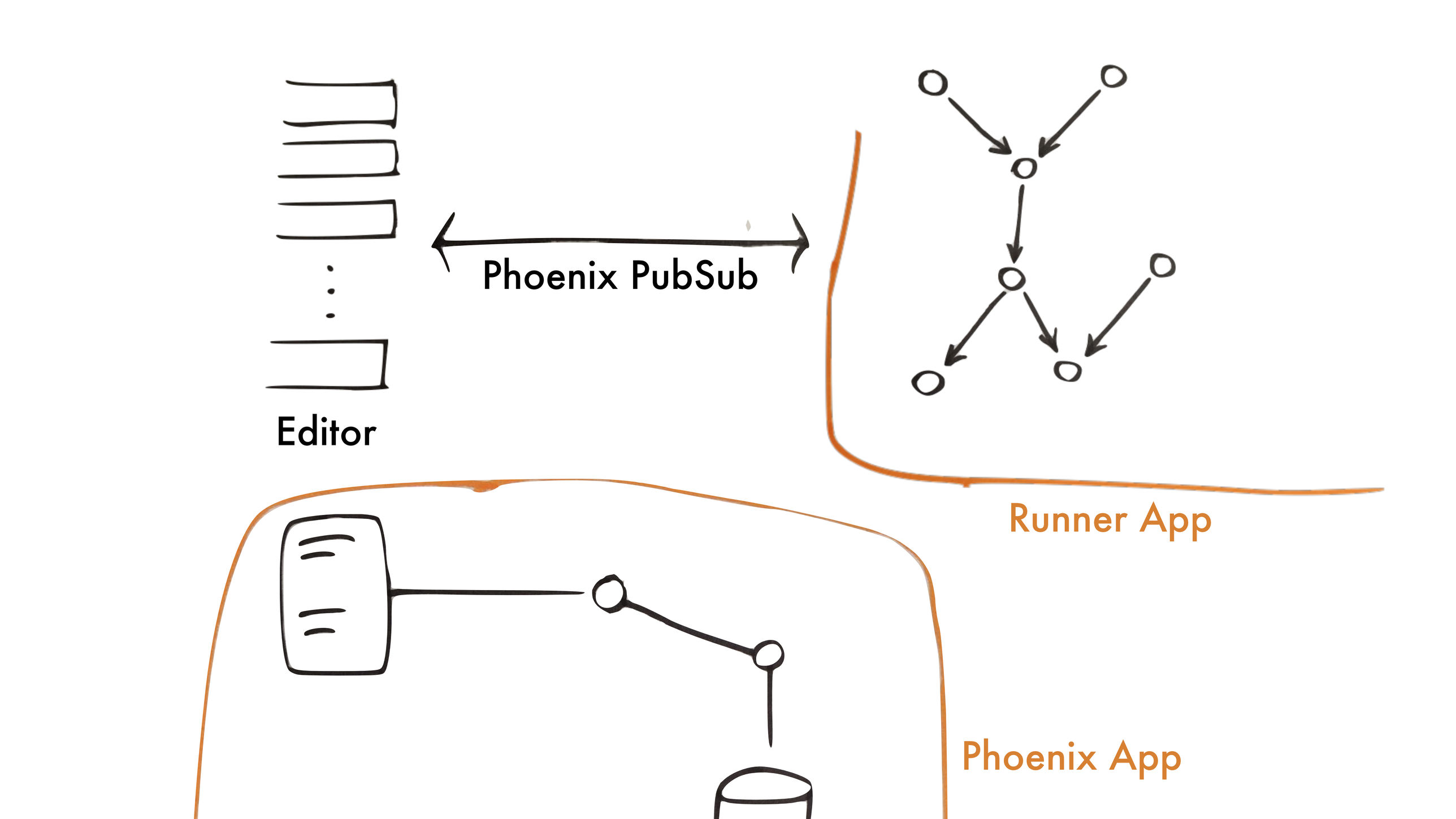
- ClojureScript & re-frame
- Transit Format – Websocket Serialiser/Ecto Type
- exchange clojure/Elixir terms (keywords ⇒ atoms)
- JSON (or MsgPack) based

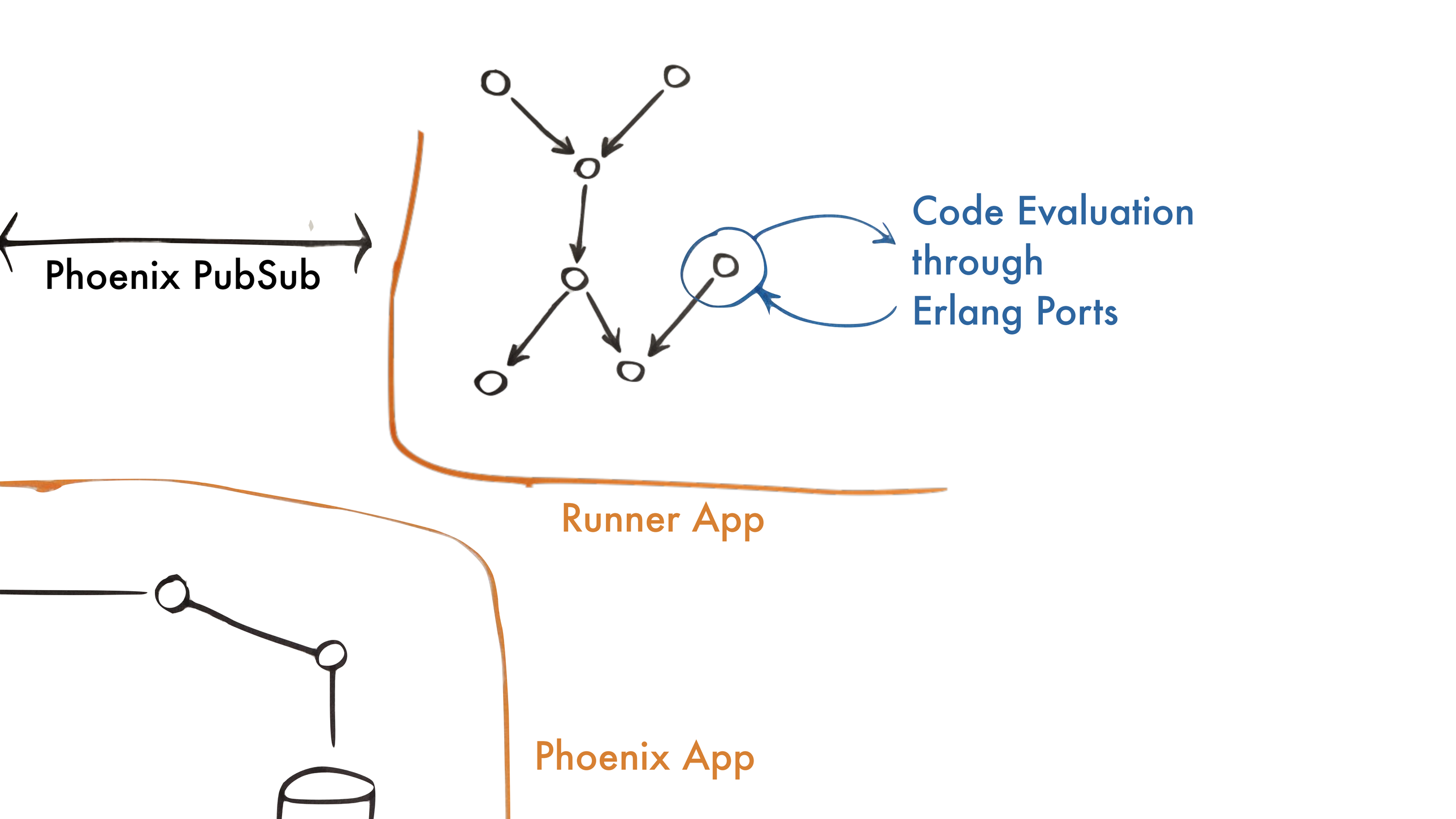




- Persists Edit Operations serialising writes to DB
- extracts CodeCells from Article State

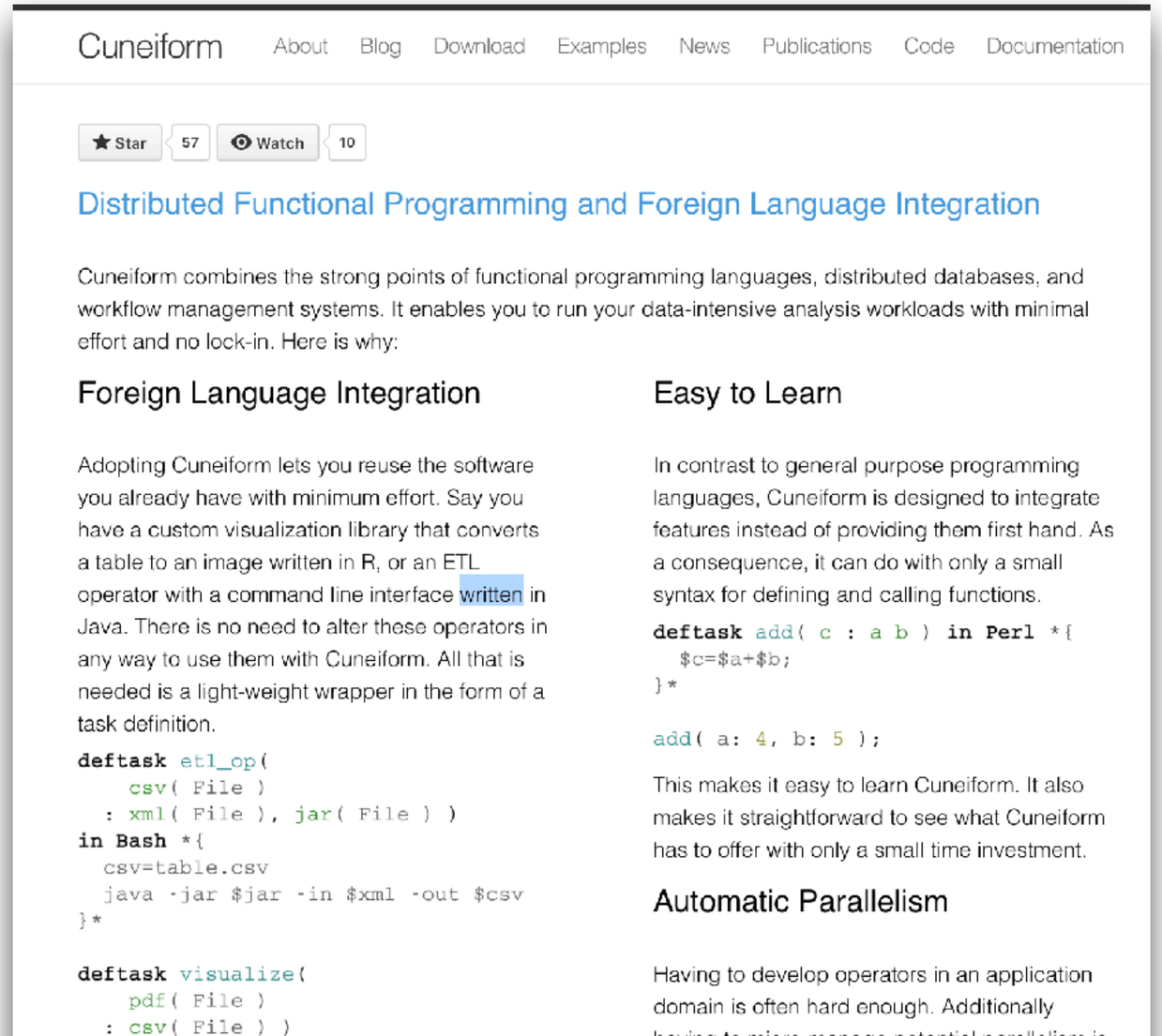

ping back
Runner App
runner start job
Article Manager
Scheduler
Phoenix App


Phoenix PubSub
Editor
Runner App
Phoenix App


Phoenix PubSub
Code Evaluation
through
Erlang Ports
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



Cuneiform About Blog Download Examples News Publications Code Documentation

Star 57 Watch 10

Distributed Functional Programming and Foreign Language Integration

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 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 ) )
```

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 add( c : a b ) in Perl *{
  $c=$a+$b;
}*

add( a: 4, b: 5 );
```

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

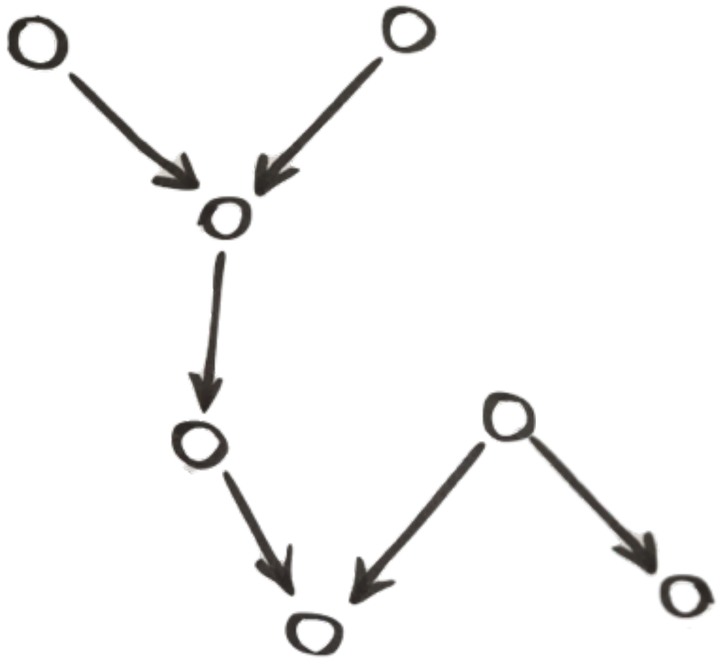
Having to develop operators in an application domain is often hard enough. Additionally

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

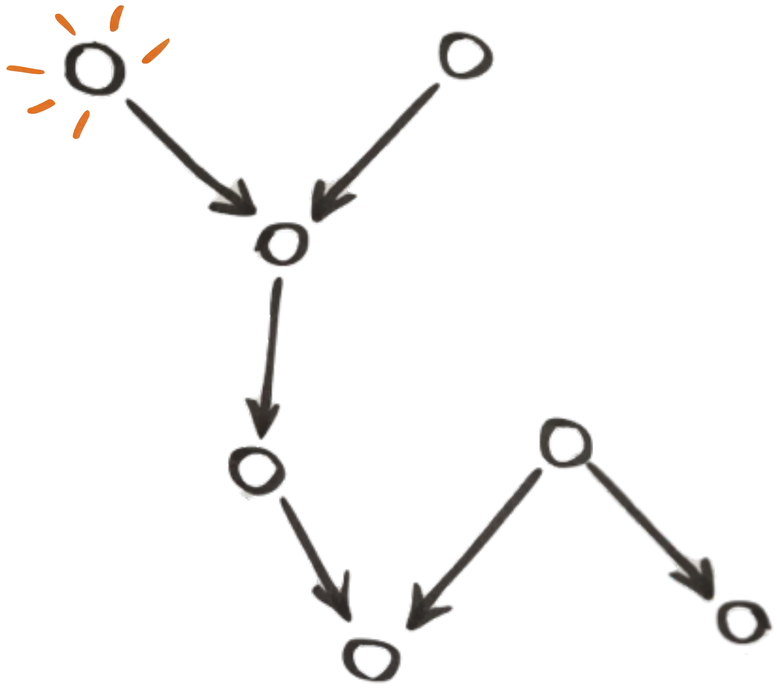
```
deftask etl_op(
    csv( File )
  : xml( File ), jar( File ) )
in Bash *{
  csv=table.csv
  java -jar $jar -in $xml -out $csv
}*

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



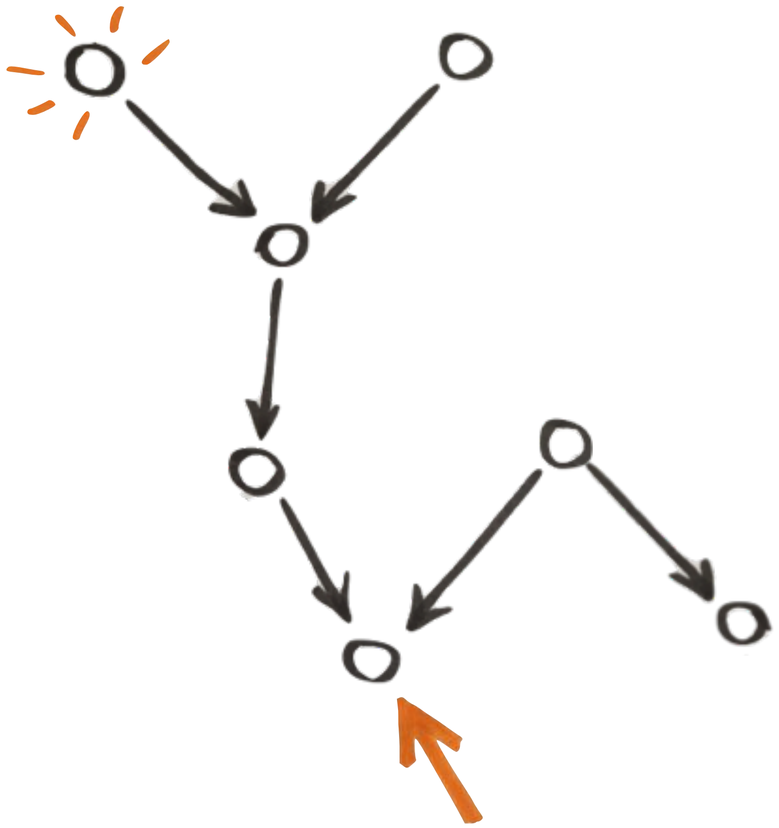
```
deftask etl_op(
    csv( File )
  : xml( File ), jar( File ) )
in Bash *{
  csv=table.csv
  java -jar $jar -in $xml -out $csv
}*

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 1: Cuneiform backend



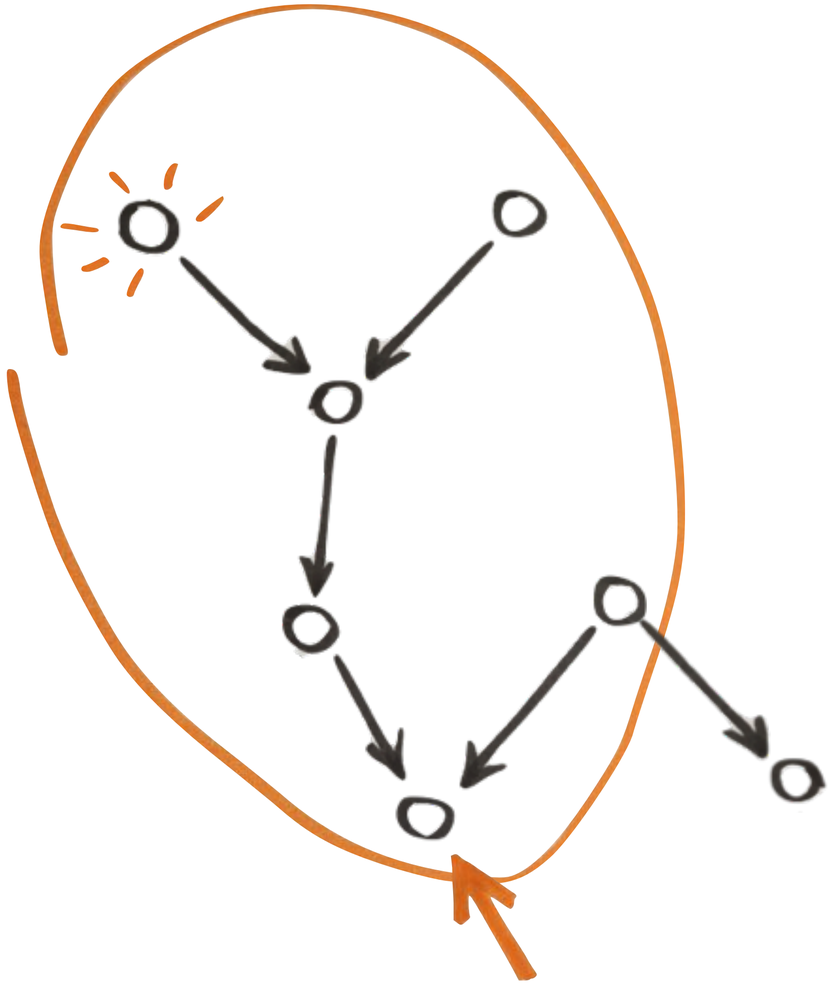
```
deftask etl_op(
    csv( File )
  : xml( File ), jar( File ) )
in Bash *{
  csv=table.csv
  java -jar $jar -in $xml -out $csv
}*

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 );
```

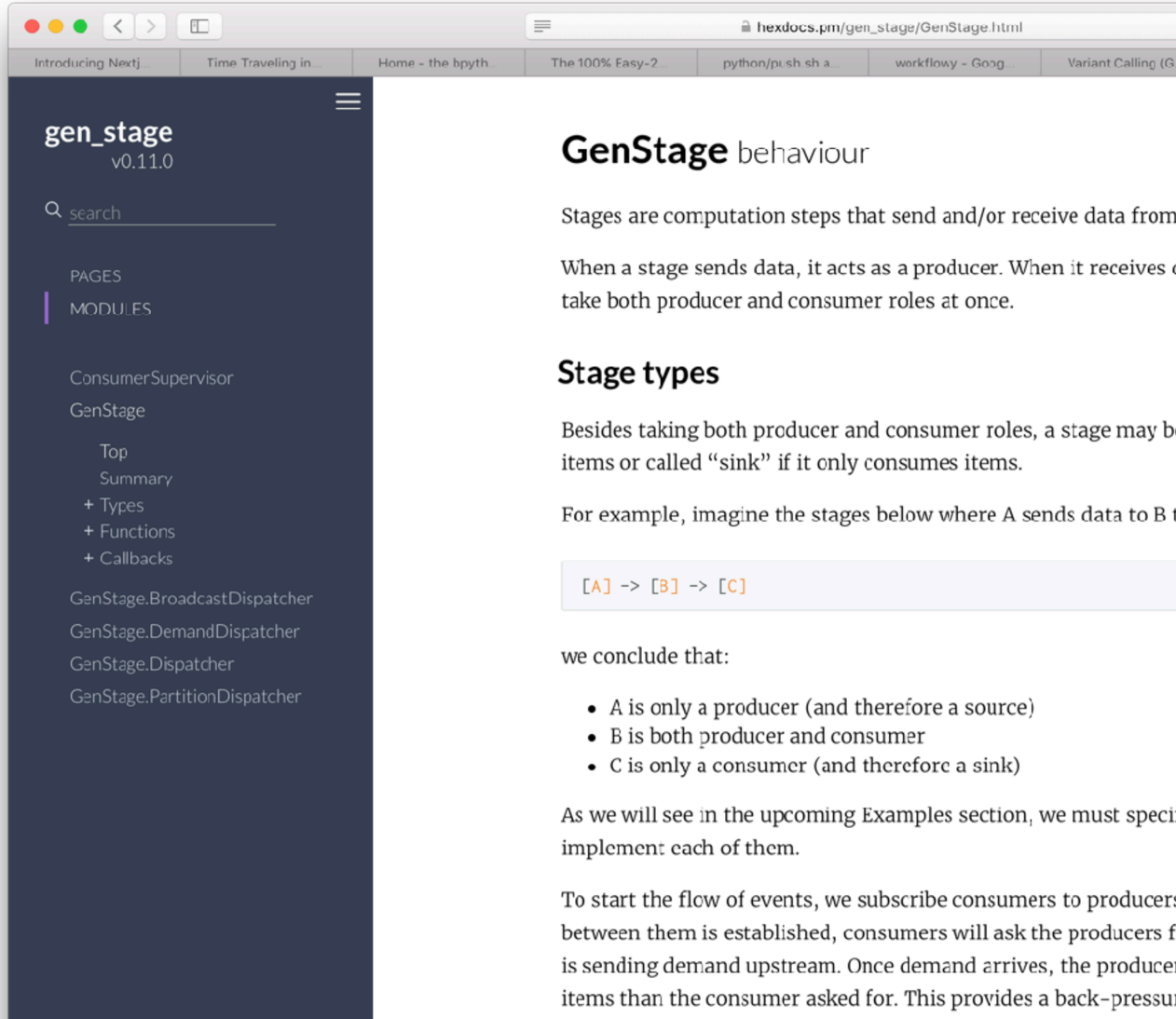
Take I: Cuneiform backend



```
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 );
```

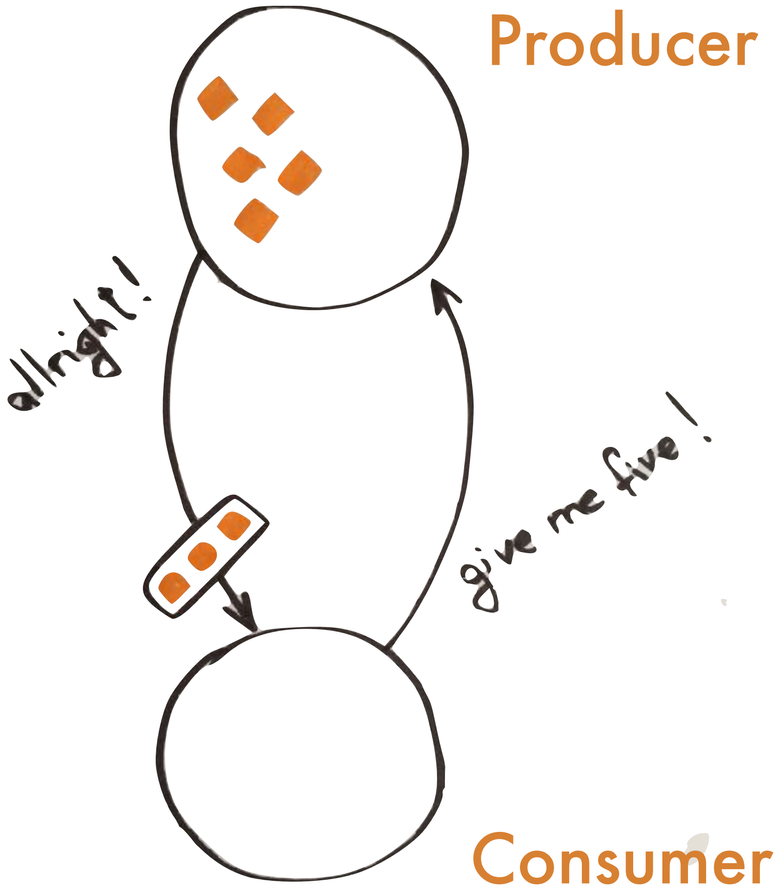
Take II: GenStage

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



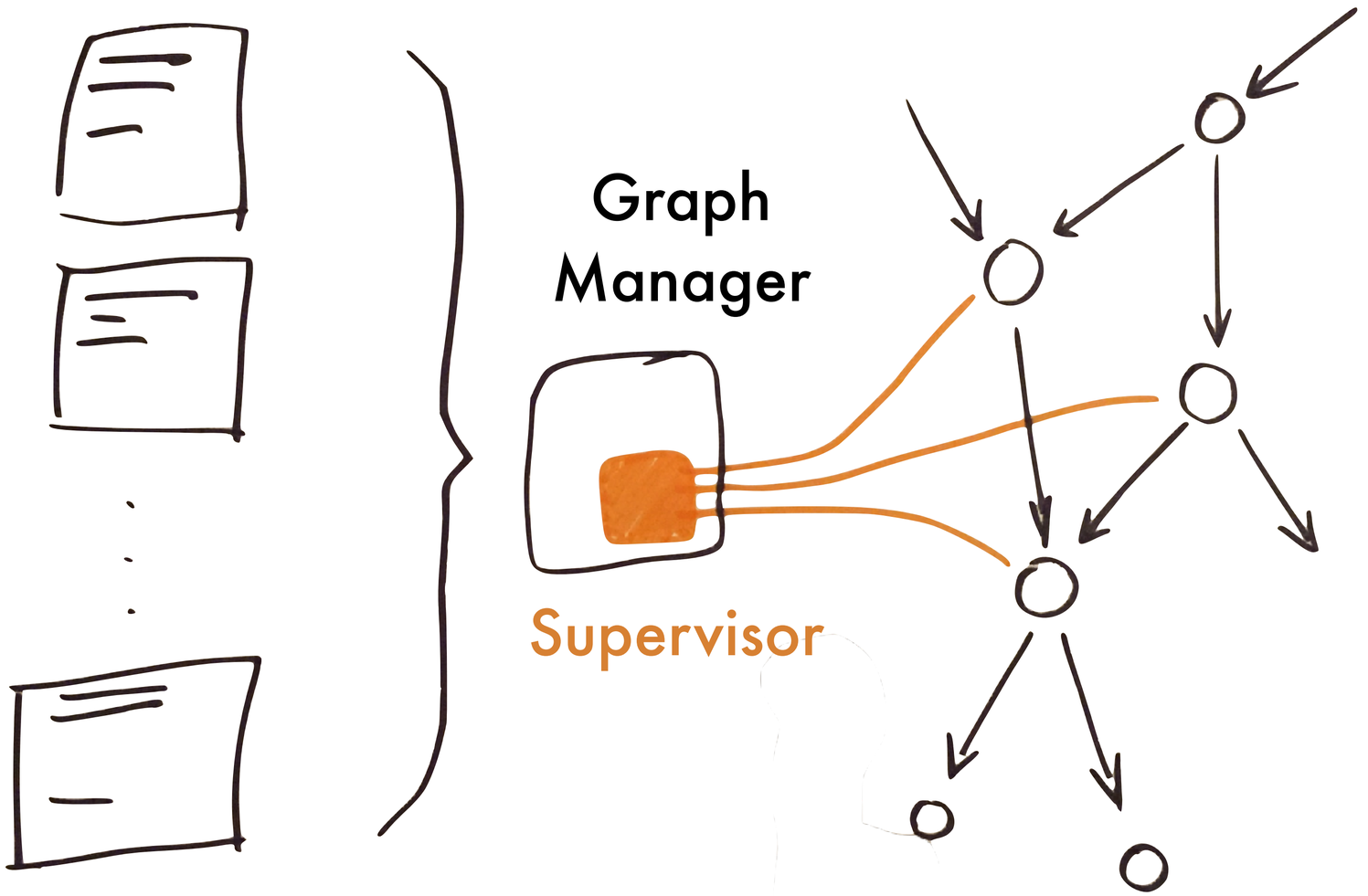
Take II: GenStage

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

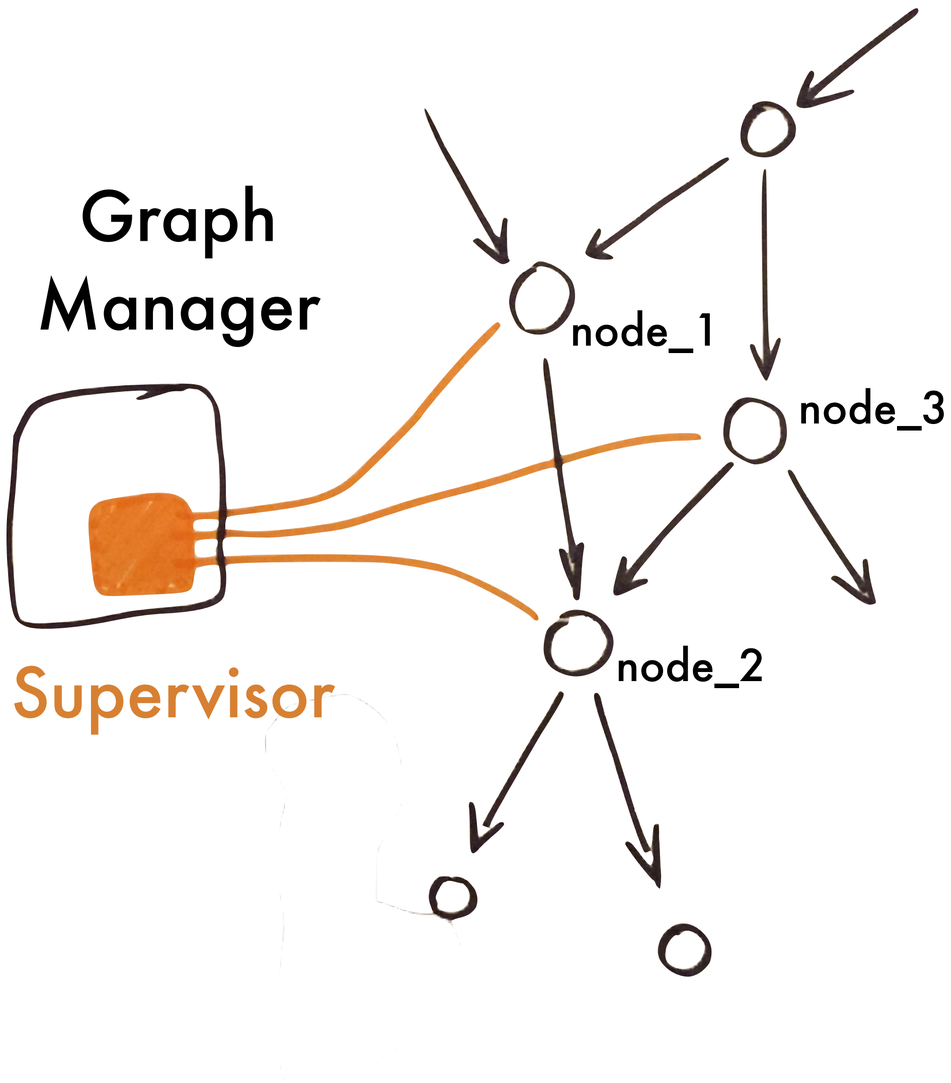


Take II: GenStage

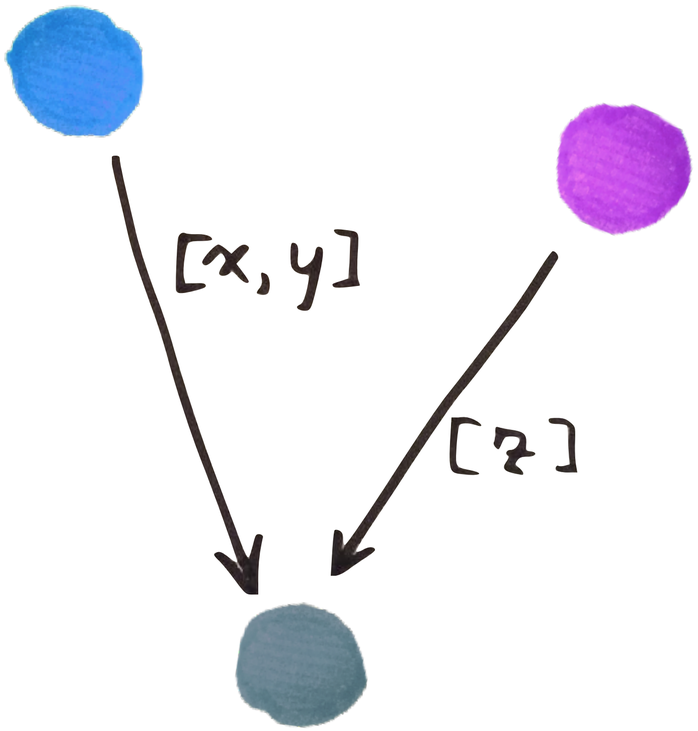
- [A] → [B] → [C]
- “thin” layer on top of GenServer
- producers, consumers and producer-consumers
- demand-based batch-processing pipelines
- “rich” stateful subscriptions
- configurable dispatchers


Graph
Manager
Supervisor

```yaml
"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

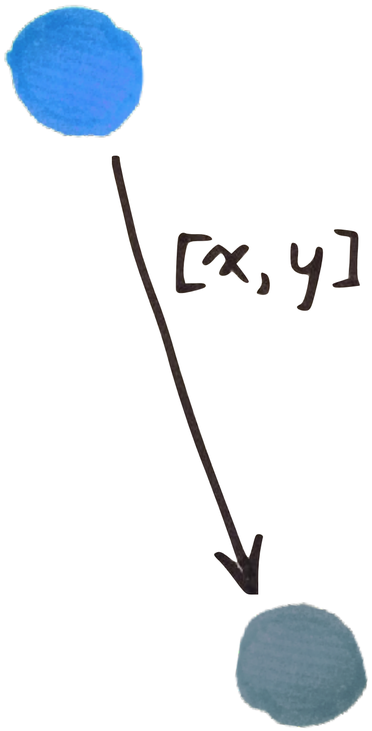
```
x = 1
y = 2
```

purple

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

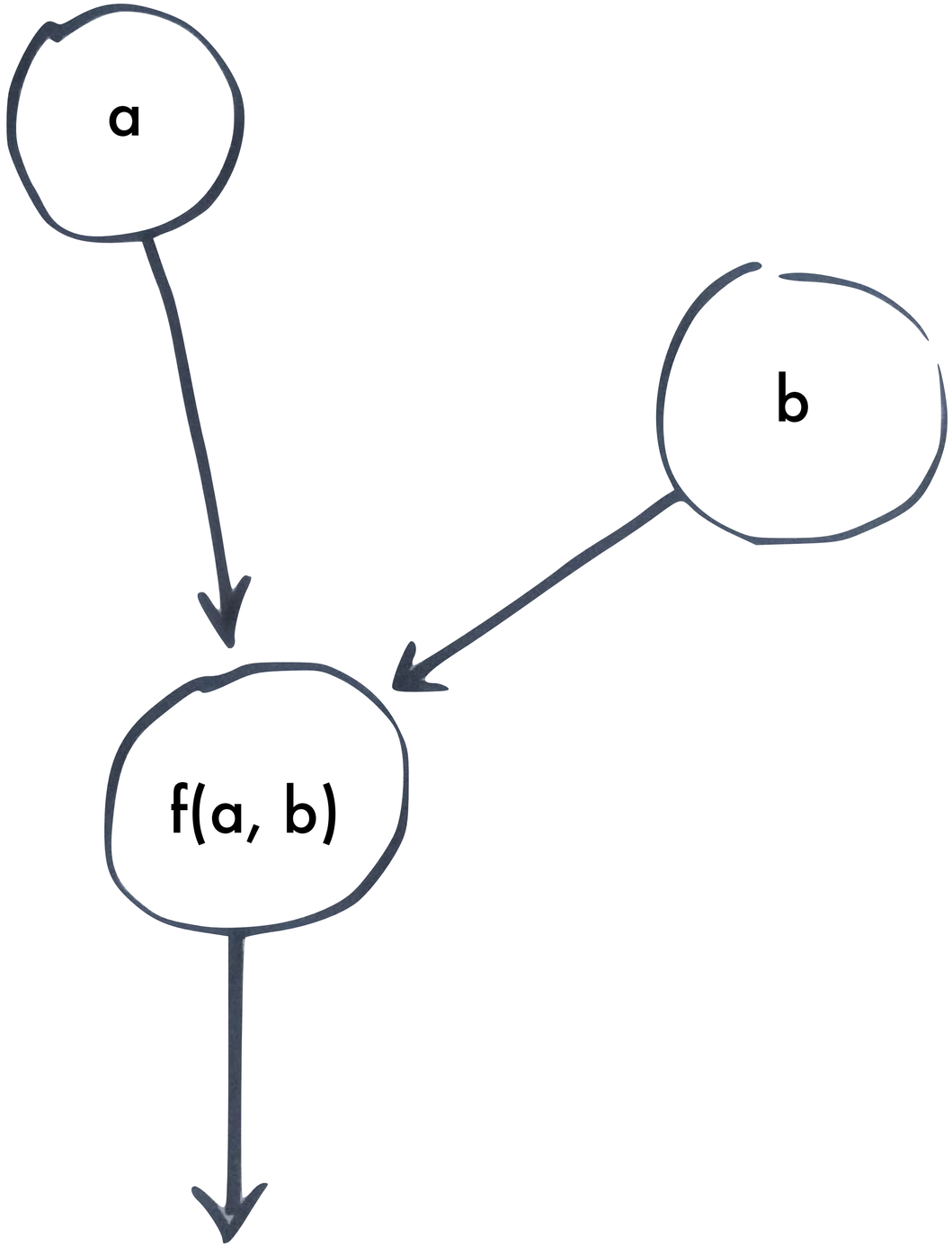
bluegray

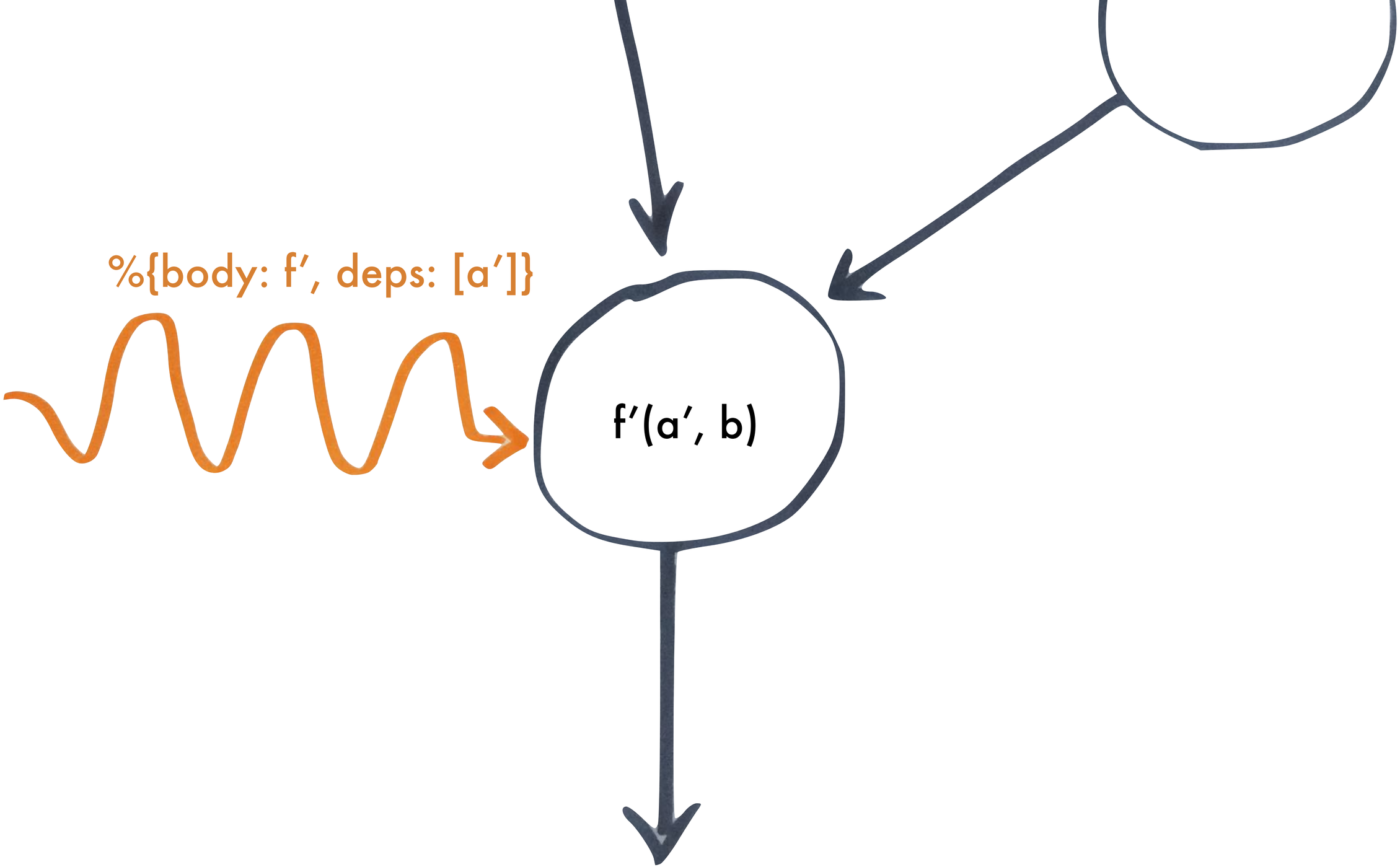
```
out <- c(blue.x, blue.y) + purple.z
```

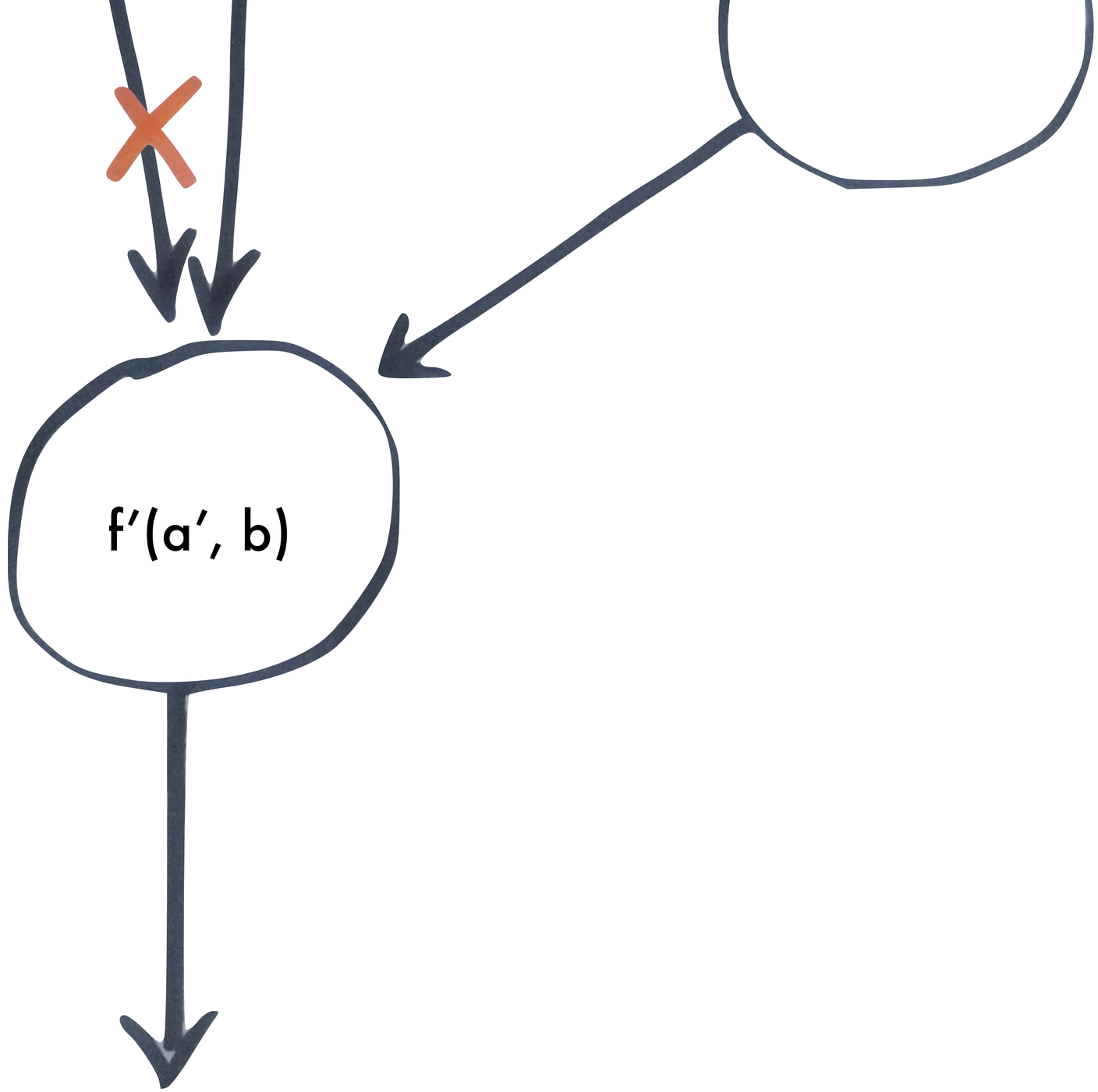


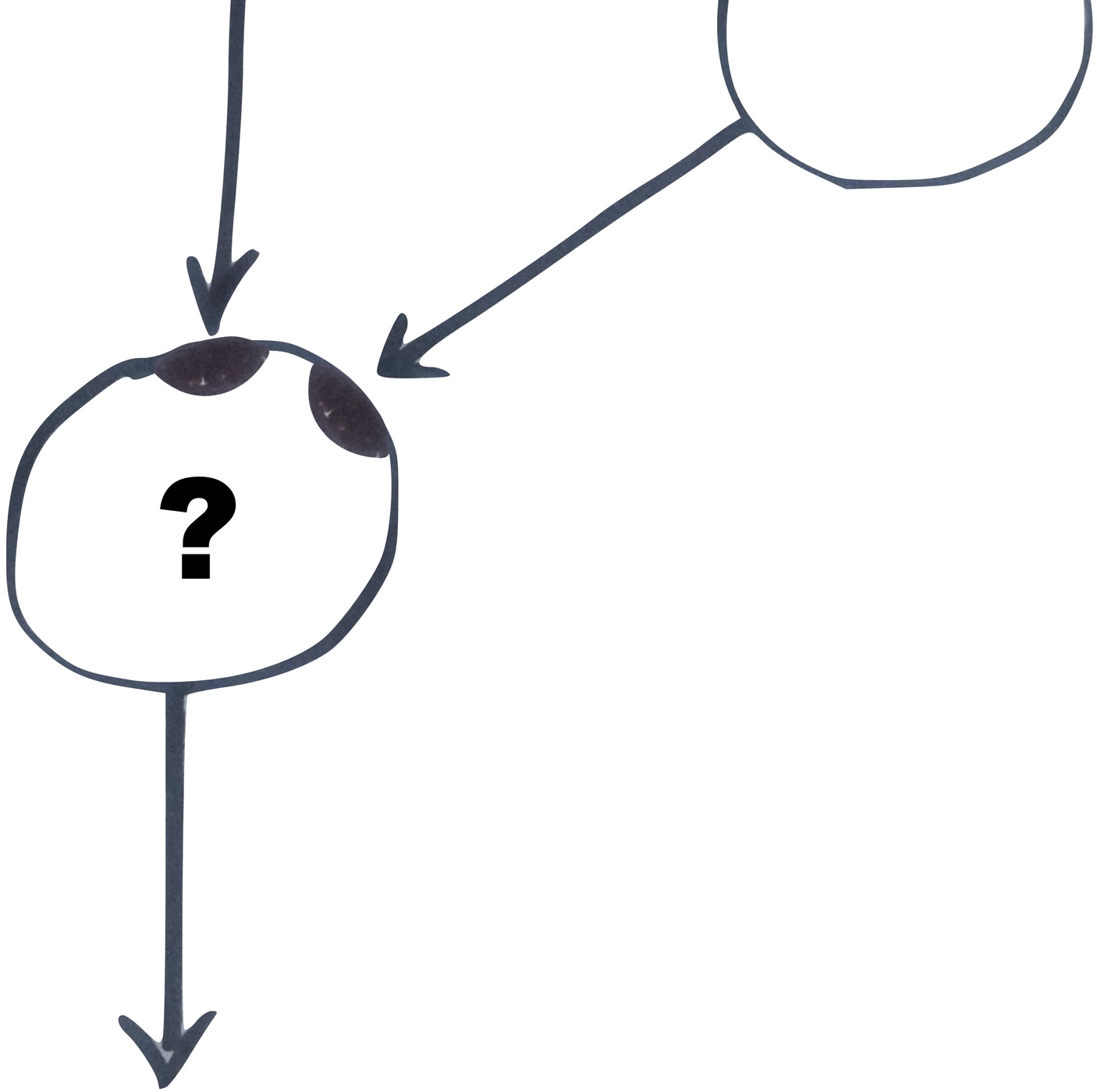

```elixir
GenStage.async_subscribe(grey,
  to: blue,
  refs: [x, y])


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




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

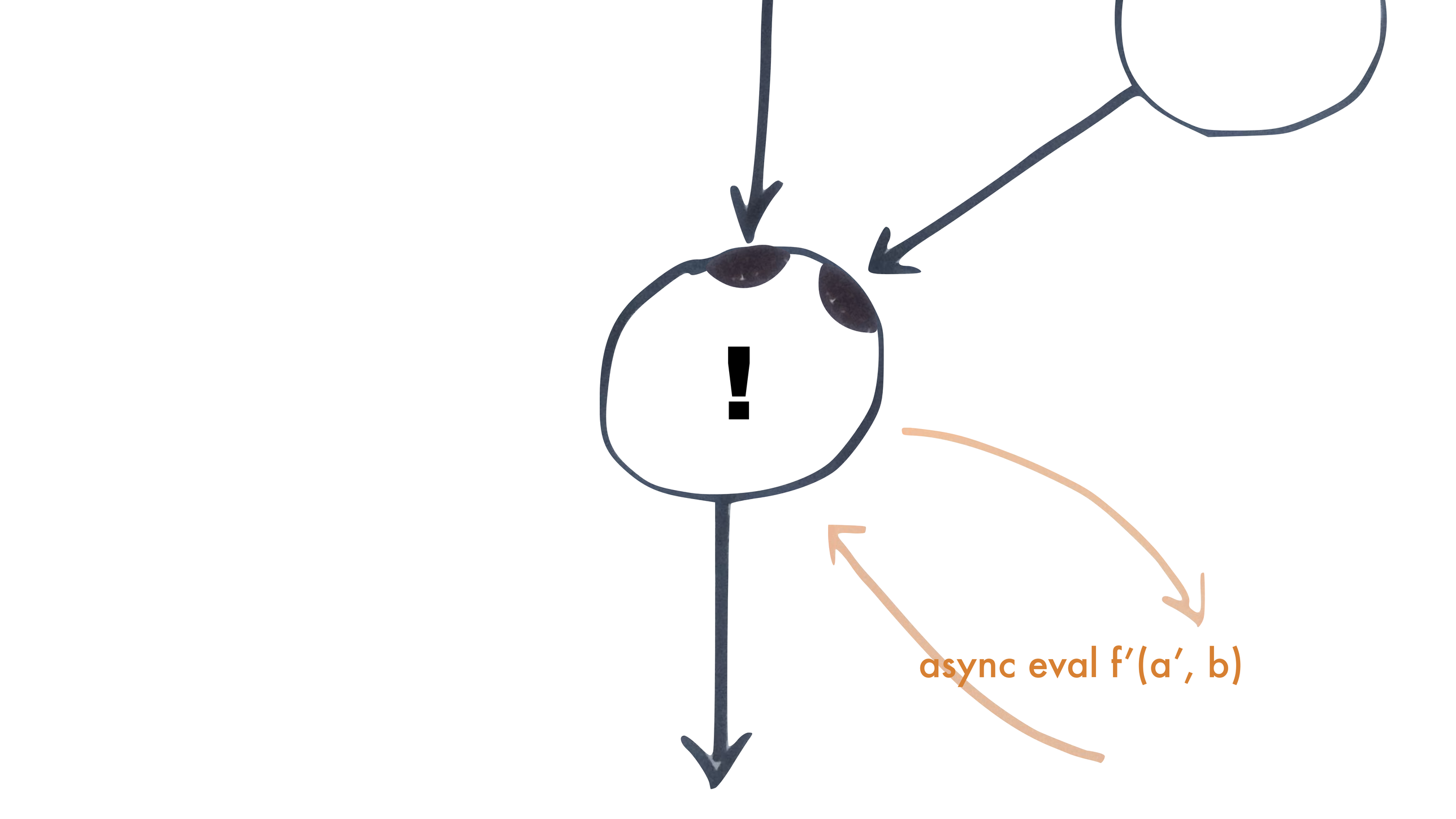
```elixir
#  PRODUCER-CONSUMERS

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...
```

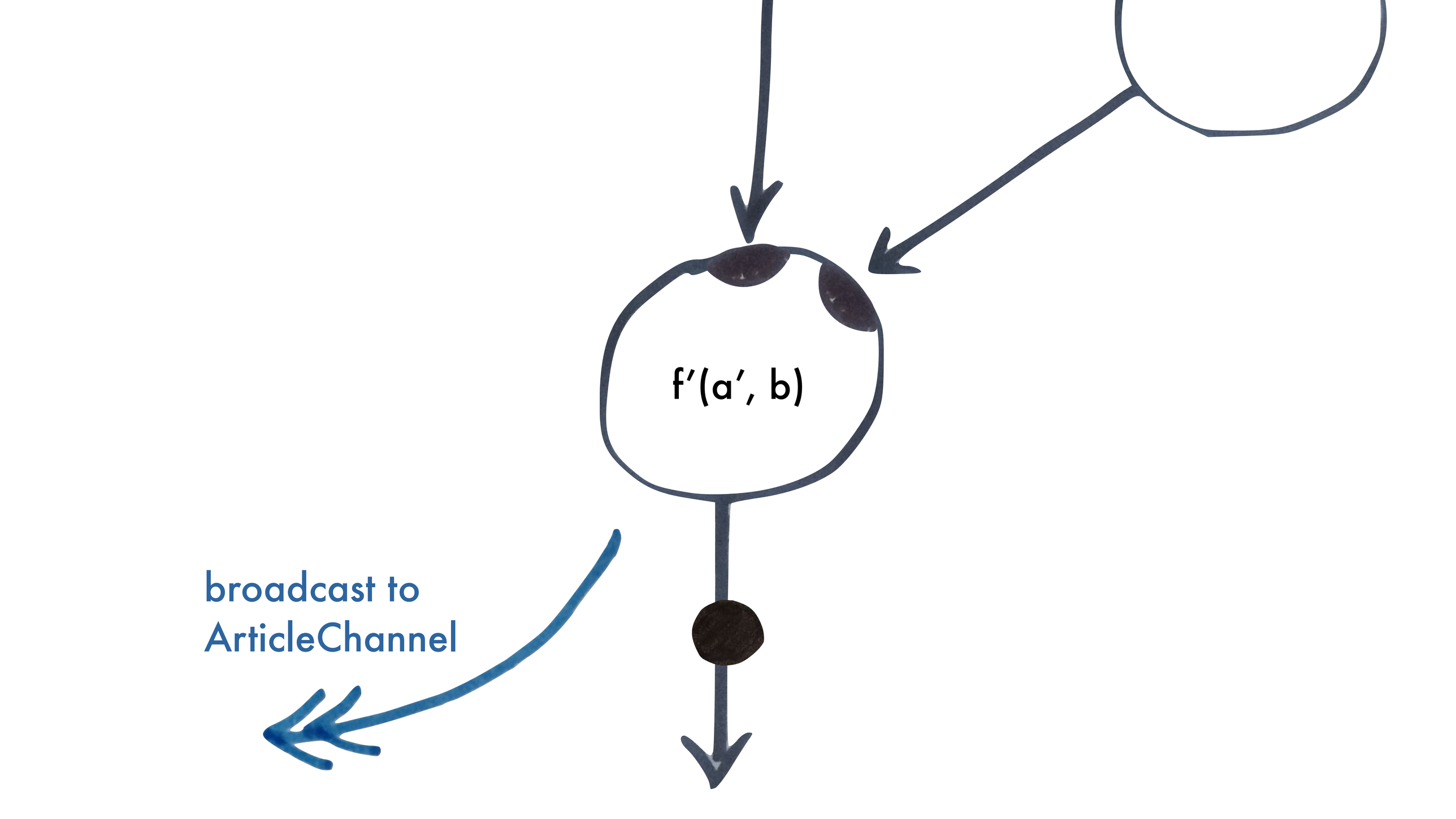

a
b
f(a, b)

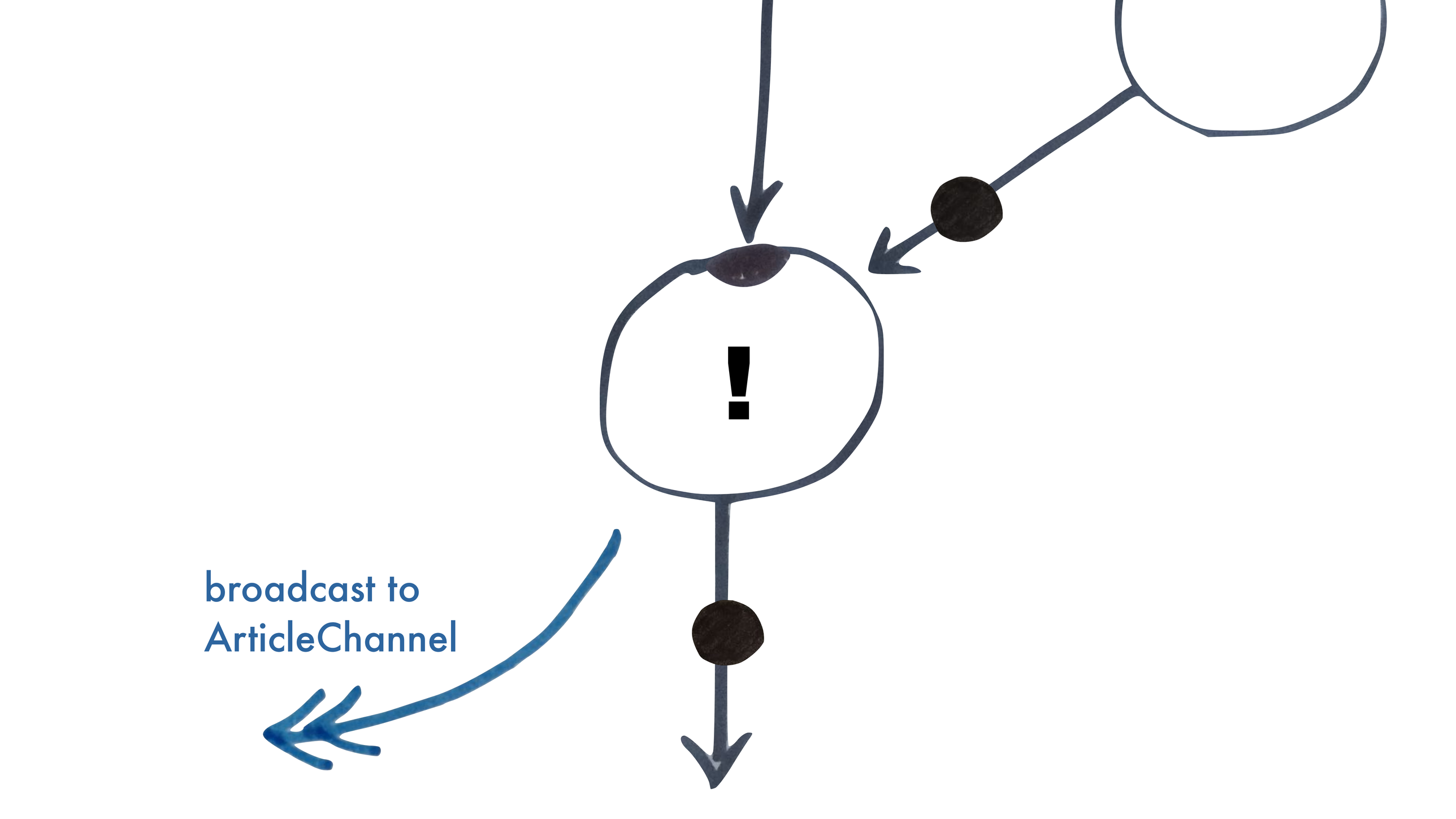

%{body: f', deps: [a']}
f'(a', b)

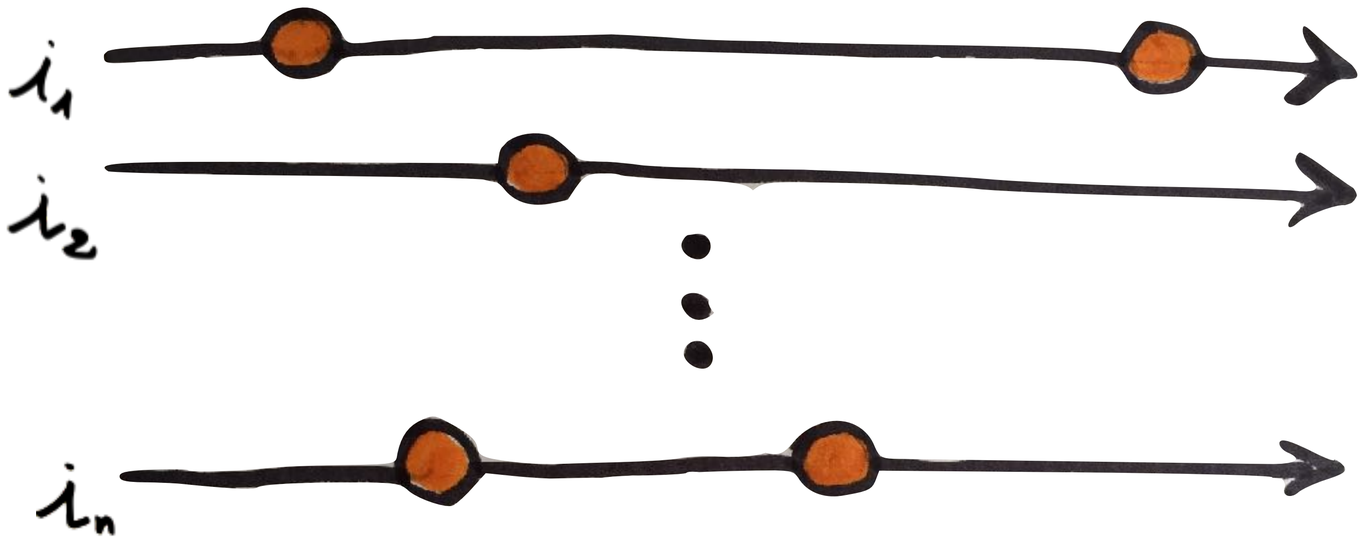

f’(a’, b)




!
async eval f'(a', b)


f’(a’, b)
broadcast to
ArticleChannel

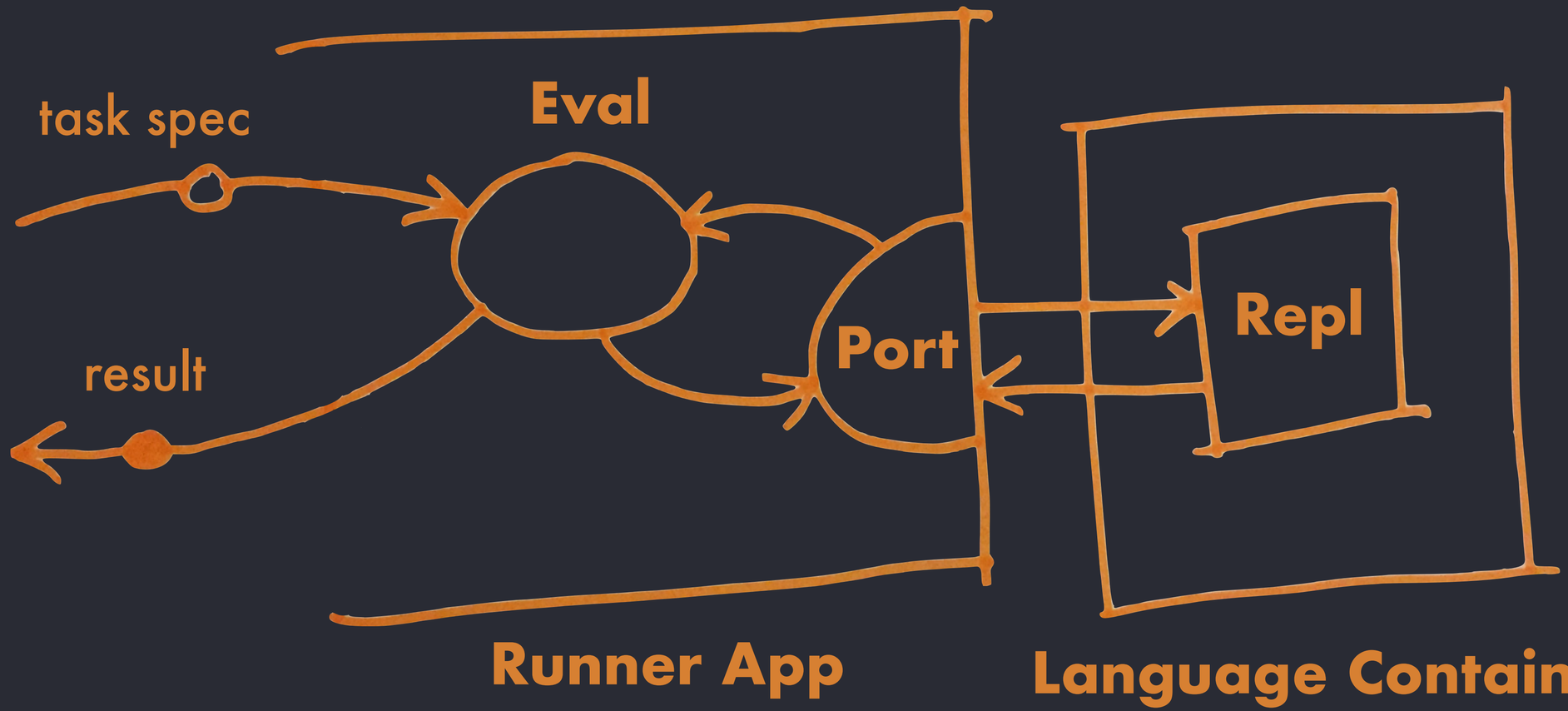

!
broadcast to
ArticleChannel



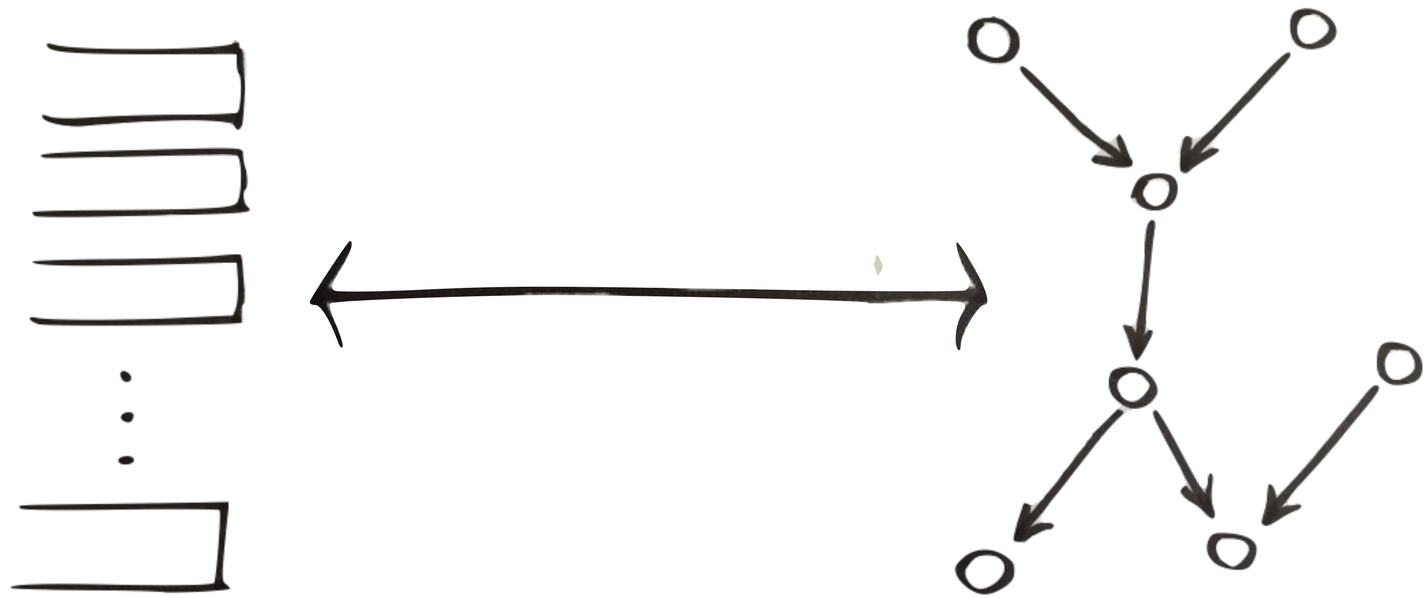
combineLatest (i_1, …, i_n) => eval(code; i_1, …, i_n)

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

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

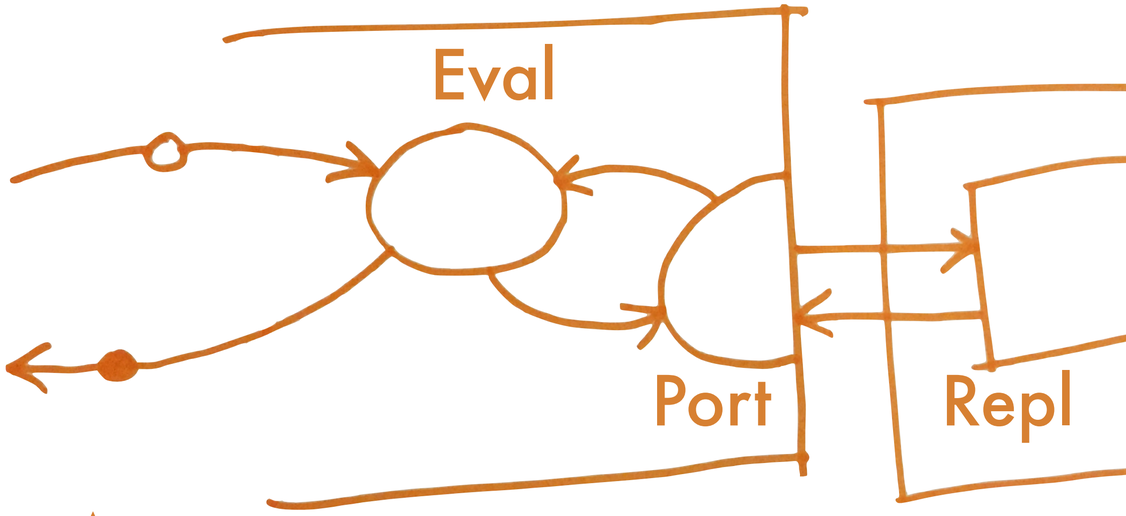
```elixir
def handle_cast(%Result{id: id} = res, %CodeCell{id: id}) do
  # ...
  broadcast(state.topic, "result", res)
  {:noreply, [res], state}
end
```




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

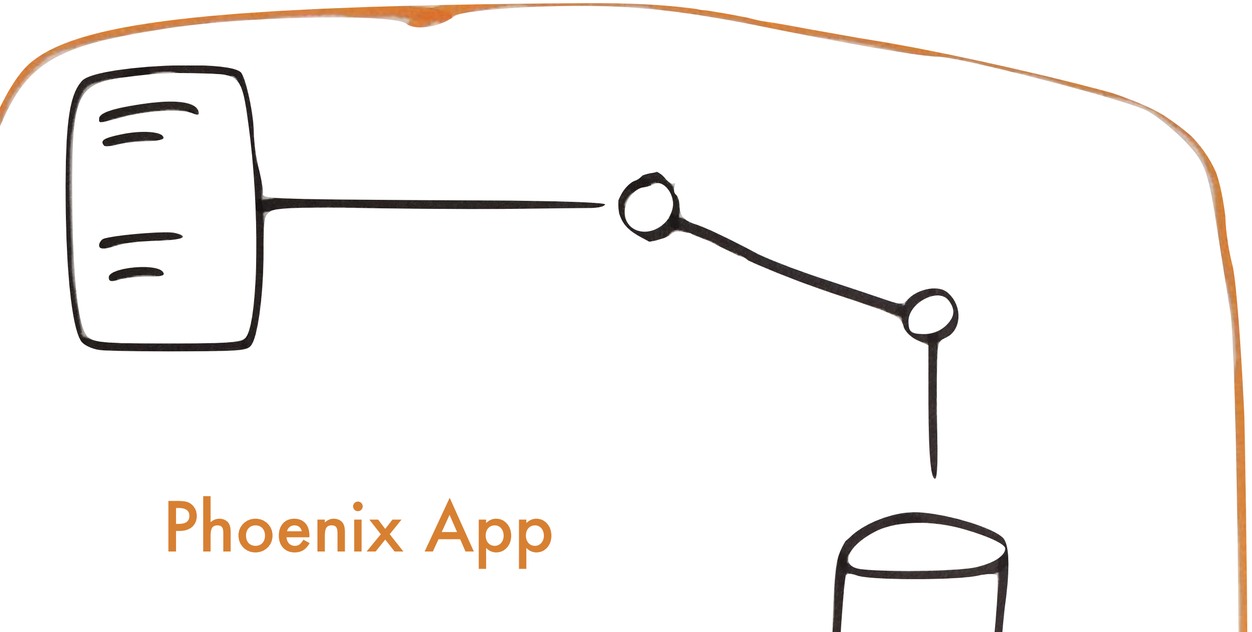


Editor



Runner App

Language Container



Phoenix App

Remarks
&
Thanks

nextjournal

@usenextjournal

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