The Road to Broadway

@plataformatec / plataformatec.com.br

Broadway

A new library for concurrent and multi-stage data ingestion and data processing with Elixir

Problem

1. Fetch data from Amazon SQS / Google Cloud PubSub
2. Process data concurrently
3. Batch data for ack / publishing
4. Publish data to another source and ack it

Broadway

```elixir
defmodule MyApp.Broadway do
  use Broadway

  def start_link(options)
  def handle_message(procesor, message, config)
  def handle_batch(batcher, messages, batch_info, config)
end
```

```elixir
Broadway.start_link(__MODULE__,
  name: __MODULE__,
  producers: [
    default: [
      module: {BroadwaySQS.Producer, queue_name: "my_queue"},
      stages: 4
    ],
  ],
  processors: [
    default: [stages: 20]
  ],
  batchers: [
    default: [batch_size: 10, batch_timeout: 1500, stages: 2]
  ]
)
```

Schedule

- Collections
- GenStage
- Broadway

Collections

Requirements

- Polymorphic
- Extensible / open
- General: in-memory and resources
- Tunable: Eager -> Lazy -> Concurrent -> Distributed

Introducing reducees

May 21, 2015

Written by José Valim.

Elixir provides the concept of collections, which may be in-memory data structures, as well as events, I/O resources and more. Those collections are supported by the Enumerable protocol, which is an implementation of an abstraction we call “reducees”.

In this article, we will outline the design decisions behind such abstraction, often exploring ideas from Haskell, Clojure and Scala that eventually led us to develop this new abstraction called reducees, focusing specially on the constraints and performance characteristics of the Erlang Virtual Machine.

Recursion and Elixir

Elixir is a functional programming language that runs on the Erlang VM. All the examples on this article will be written in Elixir although we will introduce the concepts bit by bit.

Elixir provides linked-lists. Lists can hold many items and, with pattern matching, it is easy to extract the head (the first item) and the tail (the rest) of a list:

```
iex> [h|t] = [1, 2, 3]
iex> h
```

Collections: polymorphic

```
Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]

Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
```

Collections: extensible

```
defimpl Enumerable, for: RBTree do
  def count/1
  def member?/2
  def reduce/3
  def slice/1
end
```

Collections: general

```elixir
# in-memory
Enum.map(["list", "of", "lines"], &String.upcase/1)


# resource
Enum.map(File.stream!("README"), &String.upcase/1)
```

Collections: tunable

- Eager
- Lazy
- Concurrent
- Distributed

Collections: eager

```
iex> [1, 2, 3]
...> |> Enum.map(&print/1)
...> |> Enum.map(&print/1)
1
2
3
1
2
3
[1, 2, 3]
```

Collections: lazy

```
iex> [1, 2, 3]
...> |> Stream.map(&print/1)
...> |> Stream.map(&print/1)
#Stream<...>
```

Collections: lazy

```
iex> [1, 2, 3]
...> |> Stream.map(&print/1)
...> |> Stream.map(&print/1)
...> |> Enum.to_list()
1
1
2
2
3
3
[1, 2, 3]
```

Elixir v1.0

Collections

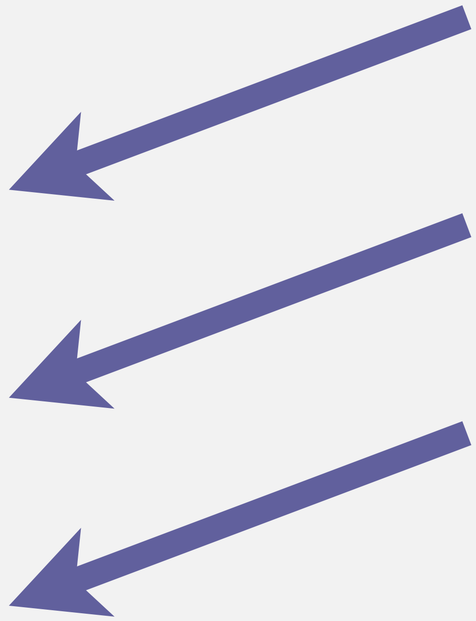
- ✓ Polymorphic
- ✓ Extensible / open
- ✓ General: in-memory and resources
- Tunable: Eager -> Lazy -> Concurrent -> Distributed

Pipeline Parallelism

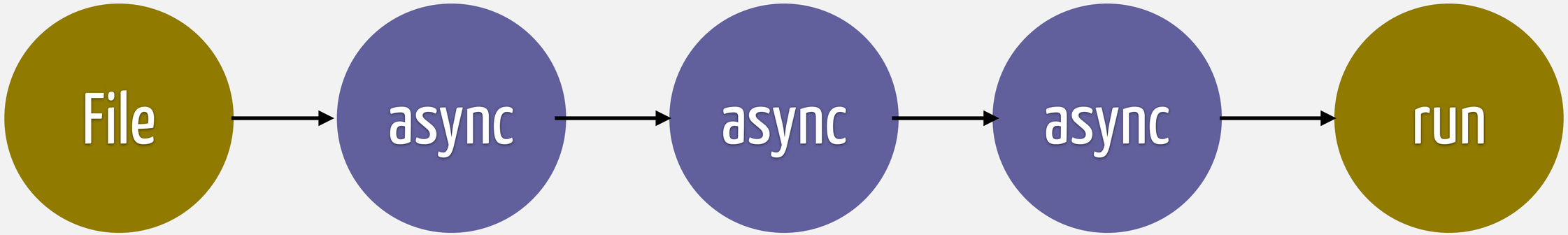
```
File.stream(path)
|> ...
|> ...
|> ...
|> ...
|> Stream.run()
```

Pipeline Parallelism

```
File.stream(path)
|> ...
|> Stream.async()
|> ...
|> Stream.async()
|> ...
|> Stream.async()
|> ...
|> Stream.run()
```



Pipeline Parallelism



Pipeline Parallelism

- Error prone as it requires manual user intervention
- Moving data vs moving computations
- How to reason about fault tolerance?
- How to provide back-pressure?

Gen?????

GenStage

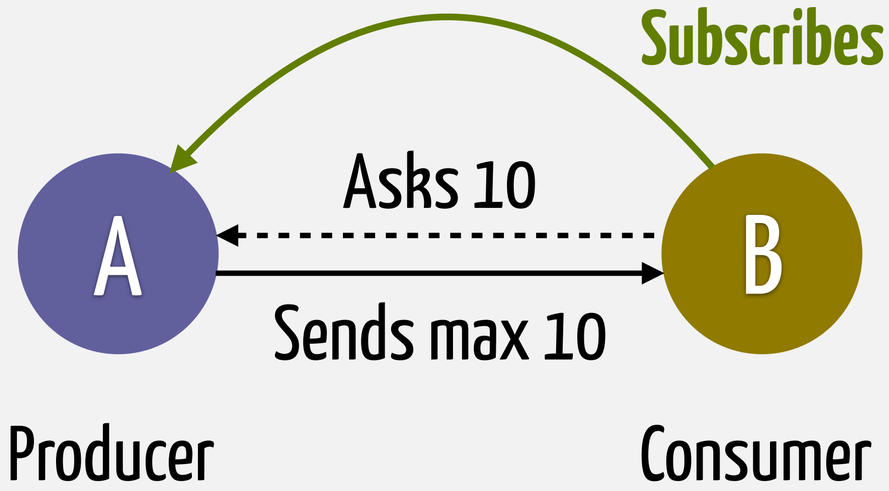
GenStage

- It is a new behaviour
- Exchanges data between stages transparently with back-pressure
- Provides producers, consumers and producer_consumers

GenStage

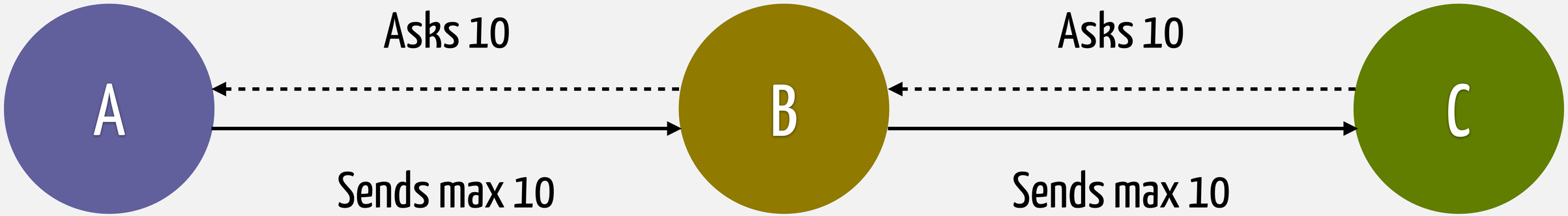

producer
producer
consumer
producer
consumer
producer
consumer
consumer

GenStage: Demand-driven



1. consumer subscribes to producer
2. consumer sends demand
3. producer sends events

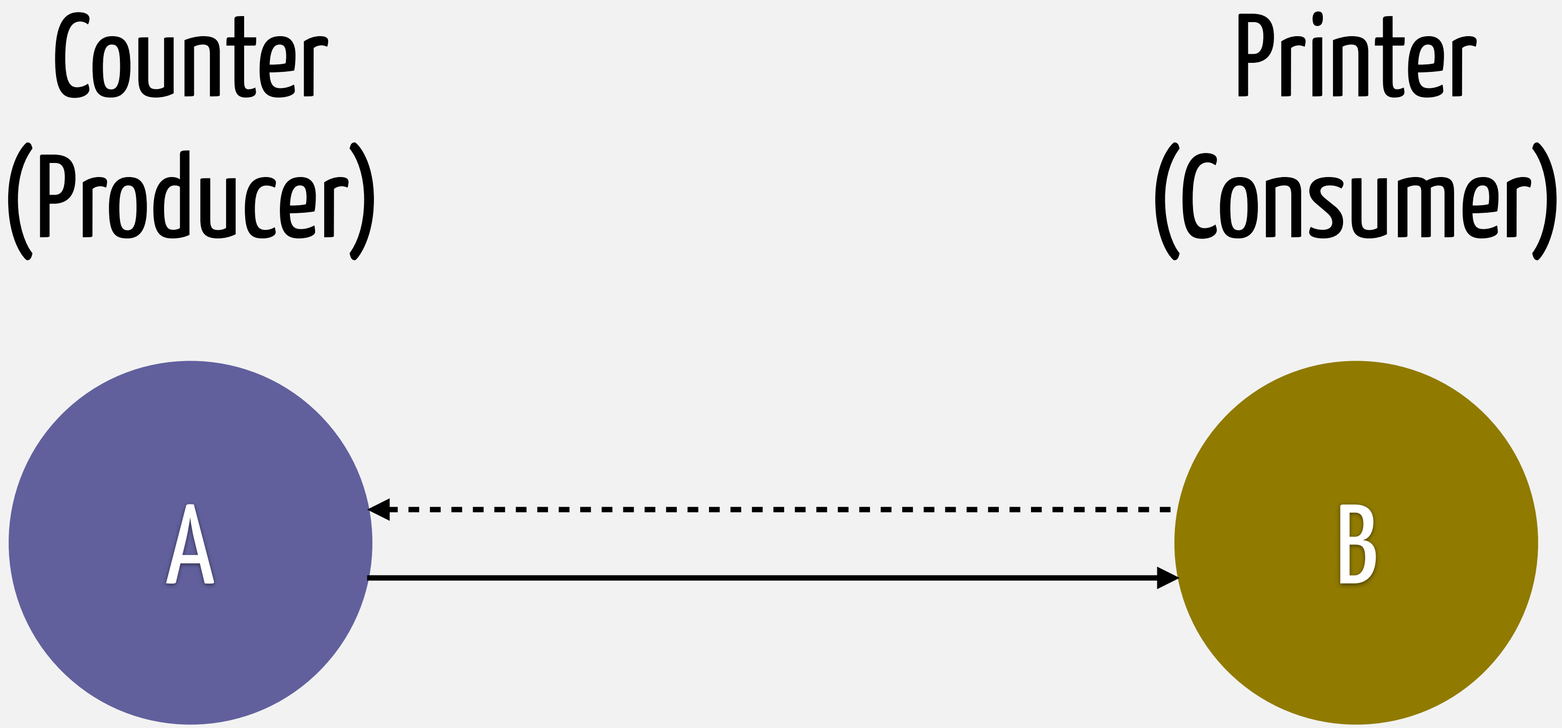
GenStage: Demand-driven



GenStage: Demand-driven

- It pushes back-pressure to the boundary
- It is a message contract
- GenStage is one implementation of this contract
- Inspired by Akka Streams

Example


Counter
(Producer)
Printer
(Consumer)
A
B

```elixir
defmodule Producer do
  use GenStage

  def init(counter) do
    {:producer, counter}
  end

  def handle_demand(demand, counter) when demand > 0 do
    events = Enum.to_list(counter..counter+demand-1)
    {:noreply, events, counter + demand}
  end
end
```

state	demand	handle_demand
0	10	{:noreply, [0, 1, …, 9], 10}
10	5	{:noreply, [10, 11, 12, 13, 14], 15}
15	5	{:noreply, [15, 16, 17, 18, 19], 20}

```elixir
defmodule Consumer do
  use GenStage

  def init(:ok) do
    {:consumer, :the_state_does_not_matter}
  end

  def handle_events(events, _from, state) do
    Process.sleep(1000)
    IO.inspect(events)
    {:noreply, [], state}
  end
end
```

```
{:ok, counter} =
  GenStage.start_link(Producer, 0)

{:ok, printer} =
  GenStage.start_link(Consumer, :ok)

GenStage.sync_subscribe(printer, to: counter)
```

```
(wait 1 second)
[0, 1, 2, ..., 499] (500 events)
(wait 1 second)
[500, 501, 502, ..., 999] (500 events)
```

Subscribe options

- max_demand: the maximum amount of events to ask (default 1000)
- min_demand: when reached, ask for more events (half of max_demand)
- cancel: how to act when the producer cancels/terminates

max_demand: 10, min_demand: 0

1. the consumer asks for 10 items
2. the consumer receives 10 items
3. the consumer processes 10 items
4. the consumer asks for 10 more
5. the consumer waits

max_demand: 10, min_demand: 5

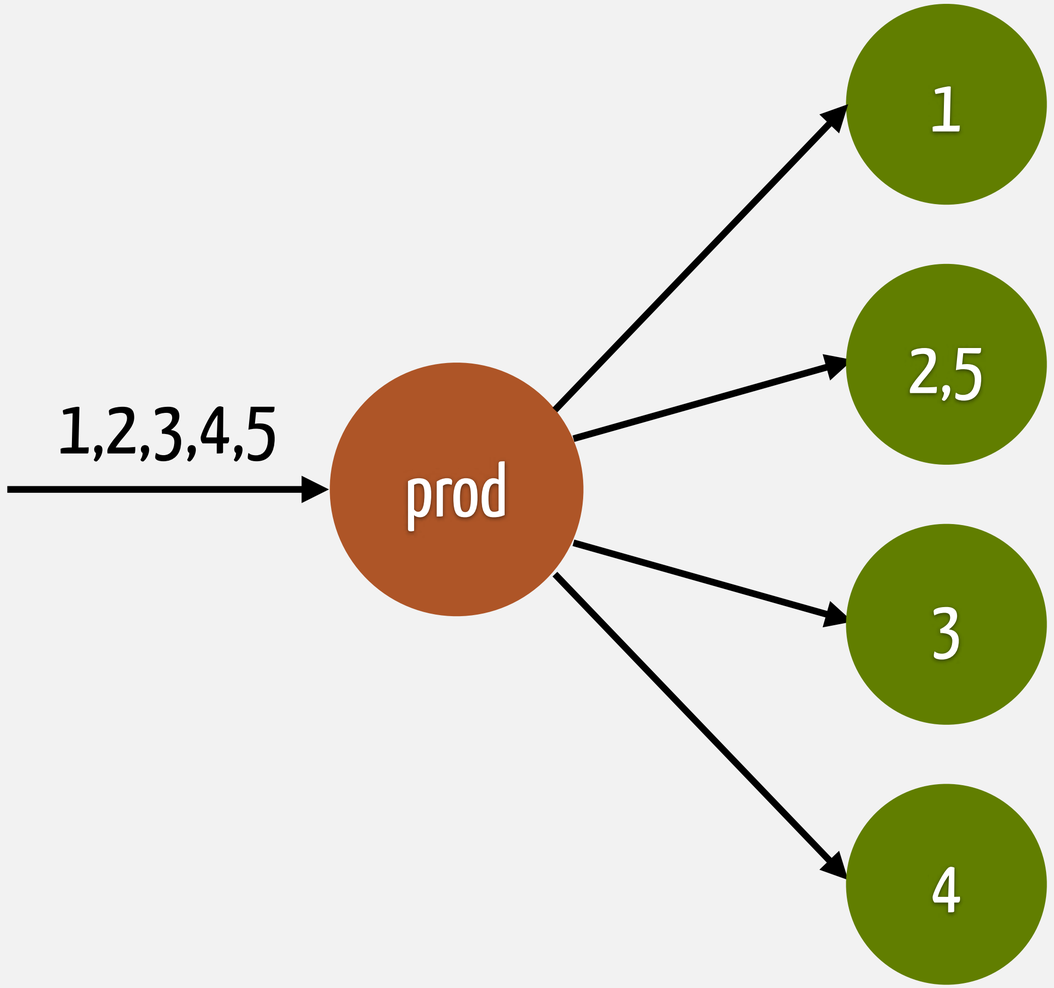
1. the consumer asks for 10 items
2. the consumer receives 10 items
3. the consumer processes 5 of 10 items
4. the consumer asks for 5 more
5. the consumer processes the remaining 5

GenStage dispatchers

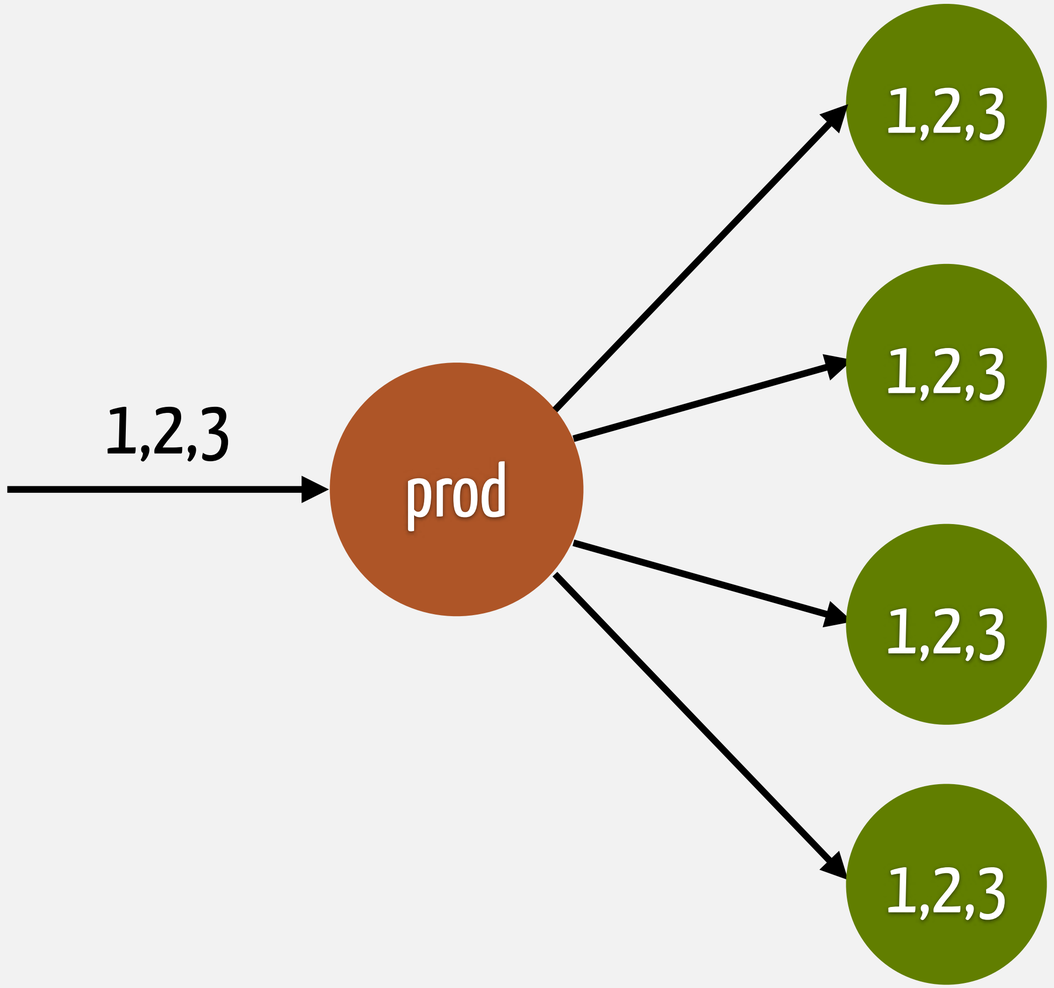
Dispatchers

- Per producer
- Effectively receive the demand and send data
- Enable concurrency by dispatching to multiple stages

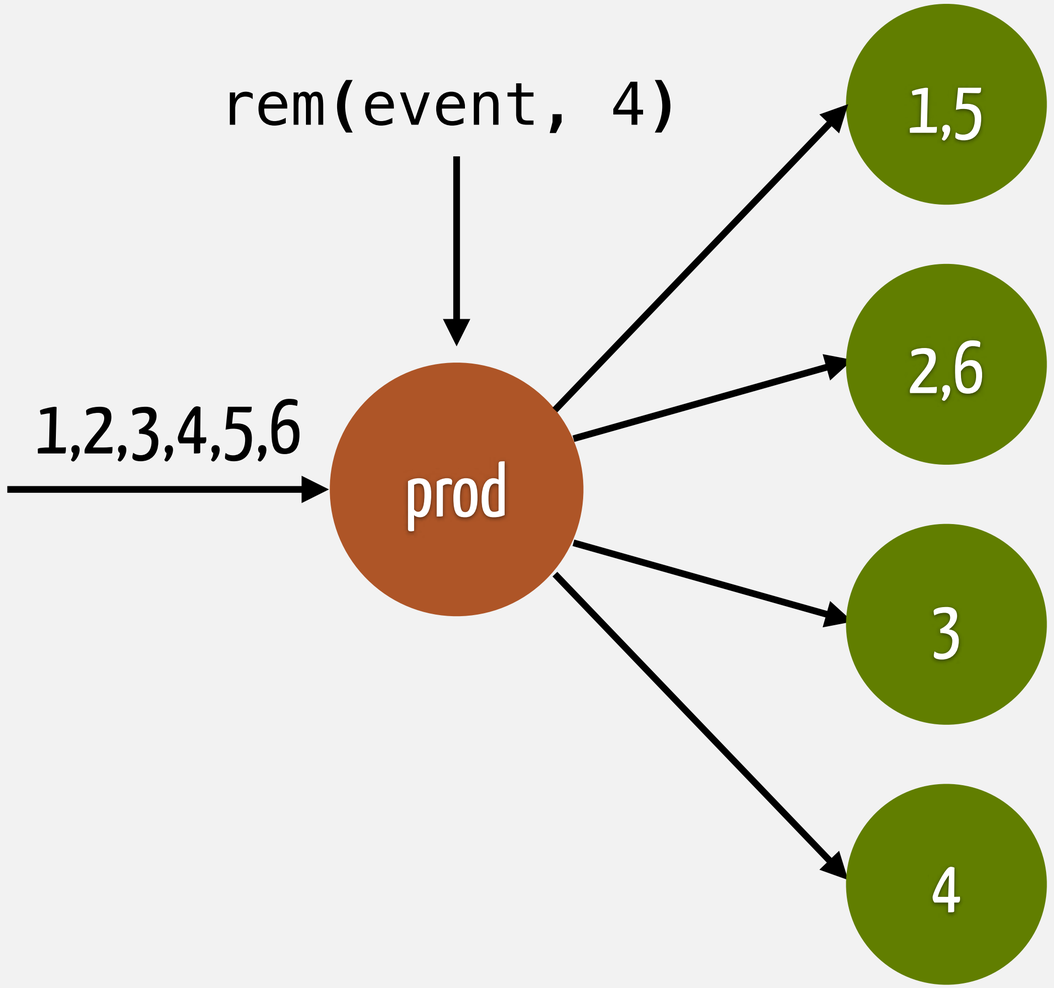
DemandDispatcher



BroadcastDispatcher



PartitionDispatcher



Announcing GenStage

July 14, 2016 · by José Valim · in Announcements

Today we are glad to announce the official release of GenStage. GenStage is a new Elixir behaviour for exchanging events with back-pressure between Elixir processes. In the short-term, we expect GenStage to replace the use cases for GenEvent as well as providing a composable abstraction for consuming data from third-party systems.

In this blog post we will cover the background that led us to GenStage, some example use cases, and what we are exploring for future releases. If instead you are looking for a quick reference, check the project source code and access its documentation.

Background

One of the original motivations for creating and designing Elixir was to introduce better abstractions for working with collections. Not only that, we want to provide developers

News: Elixir v1.8 released

Search...

BLOG CATEGORIES

- Internals
- Releases
- Announcements

OFFICIAL CHANNELS

- #elixir-lang on freenode IRC
- @elixirlang on Twitter

JOIN THE COMMUNITY

- Elixir Forum

How Discord handles push request bursts of over a million per minute with Elixir's GenStage

Top highlight

Jesse Howarth Follow

Dec 12, 2016 · 5 min read

Discord has seen tremendous growth. To handle this growth, our engineering team has had the pleasure of figuring out how to scale the backend services.

One piece of technology we've seen great success with is Elixir's GenStage.

The perfect storm: Overwatch and Pokémon GO

This past summer, our mobile push notification system started having a struggle. /r/Overwatch's Discord had just passed 25,000 concurrent users,

Quaff that potion: saving $millions with Elixir and Erlang

Written by Mike Watters, January 08, 2018

We slashed our DynamoDB costs by over 75% using Kinesis, DynamoDB streams, and Erlang/OTP (and now Elixir) to implement a global cache warming system. We present that system and two new open-source libraries for processing Kinesis and DynamoDB streams in a similar way using Elixir and Erlang.

`15-20 minute read`

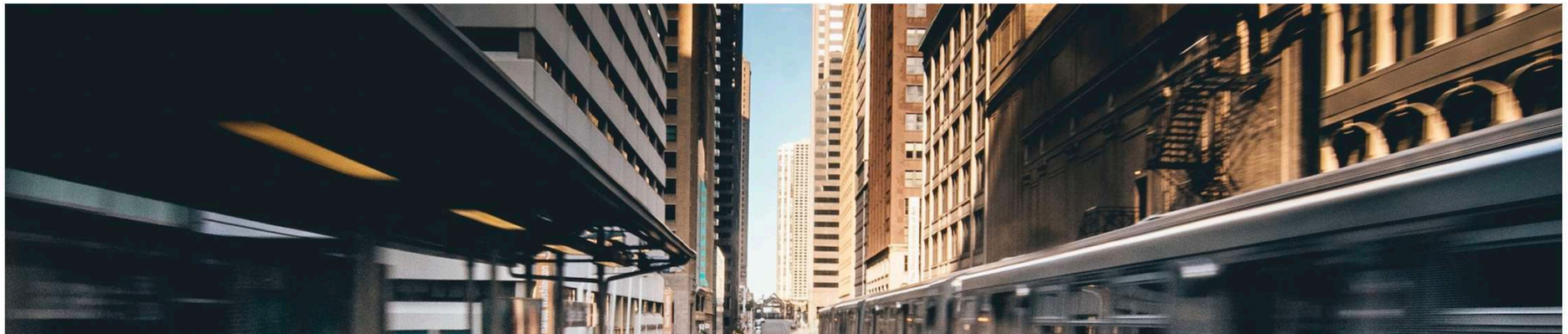
AdRoll uses Erlang/OTP as the basis for several internal products, including a real-time bidding platform running on Amazon EC2. Erlang/OTP is the king of robust highly-concurrent soft real-time systems such as these.

This article describes how we substantially reduced the cost of an element of our real-

Maximizing HTTP/2 performance with GenStage

07 NOV 2017 on Elixir, HTTP/2, and GenStage

A core feature of our Forza Football app is push notifications about live match events. With Apple moving their push notifications services to HTTP/2, we wanted to take advantage of the functionalities that their new API provides and at the same maximize performance and improve resource usage with the new platform.

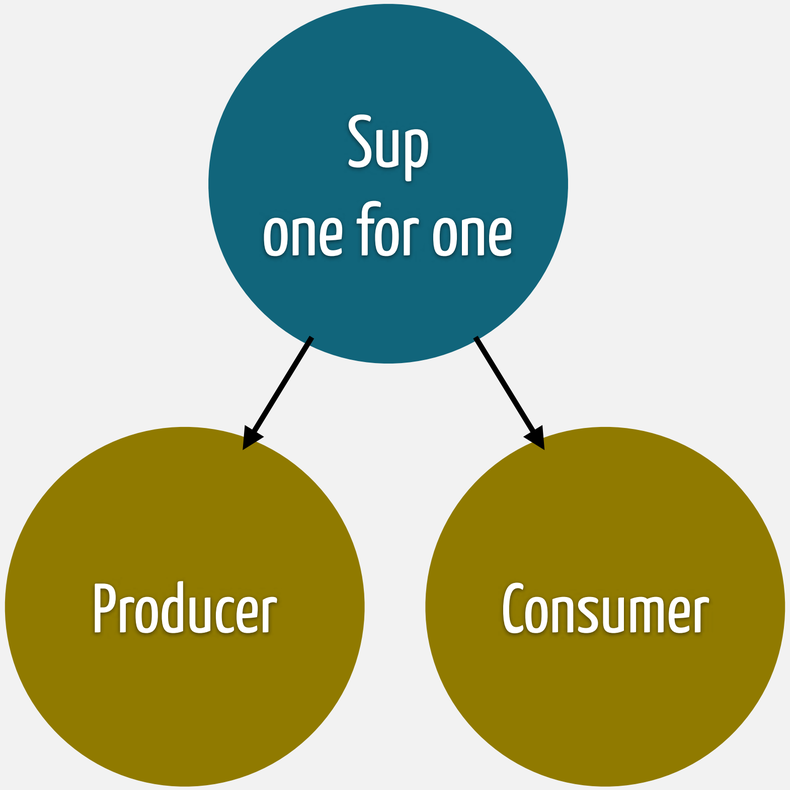


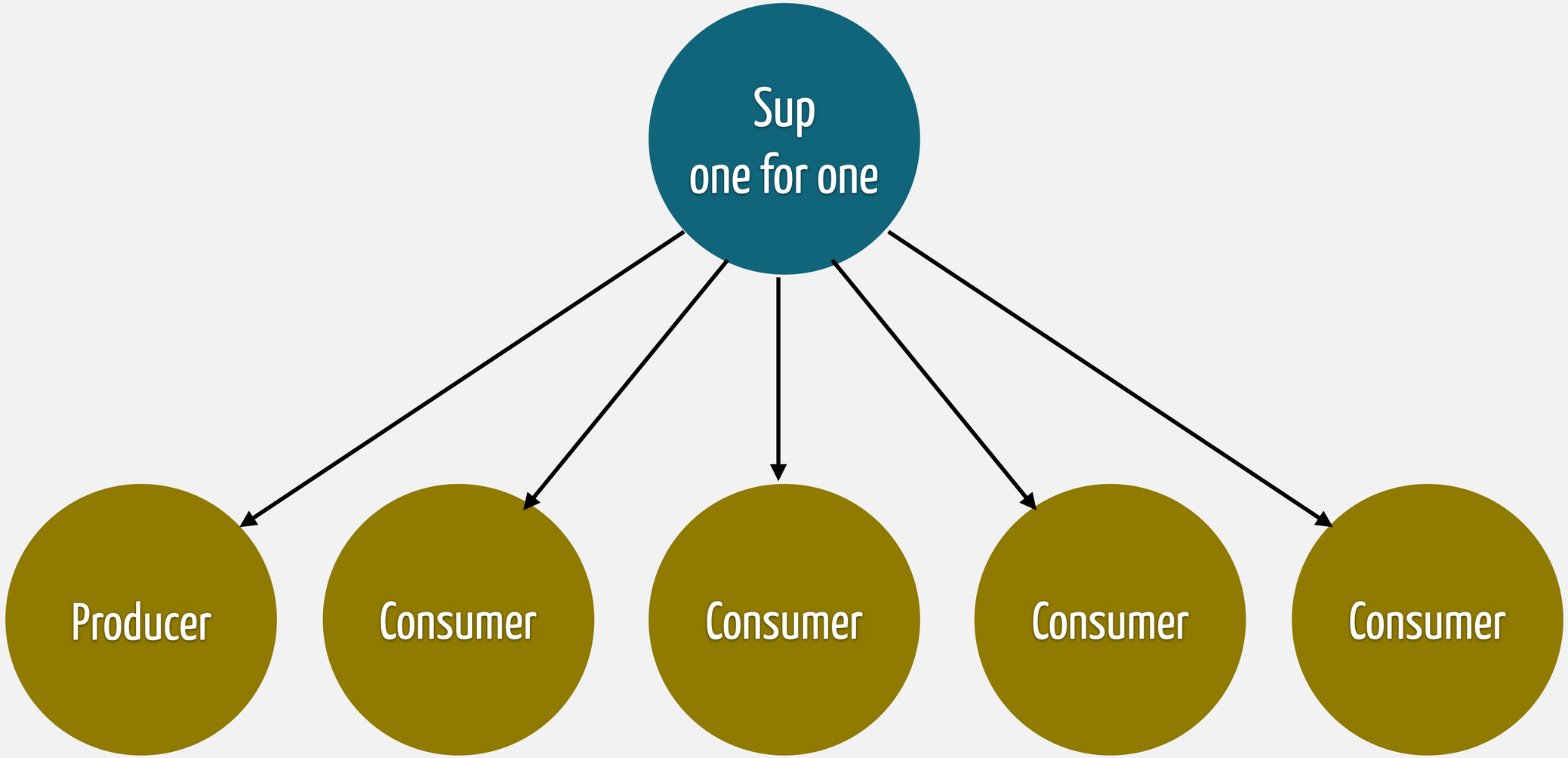
Data ingestion
Data processing

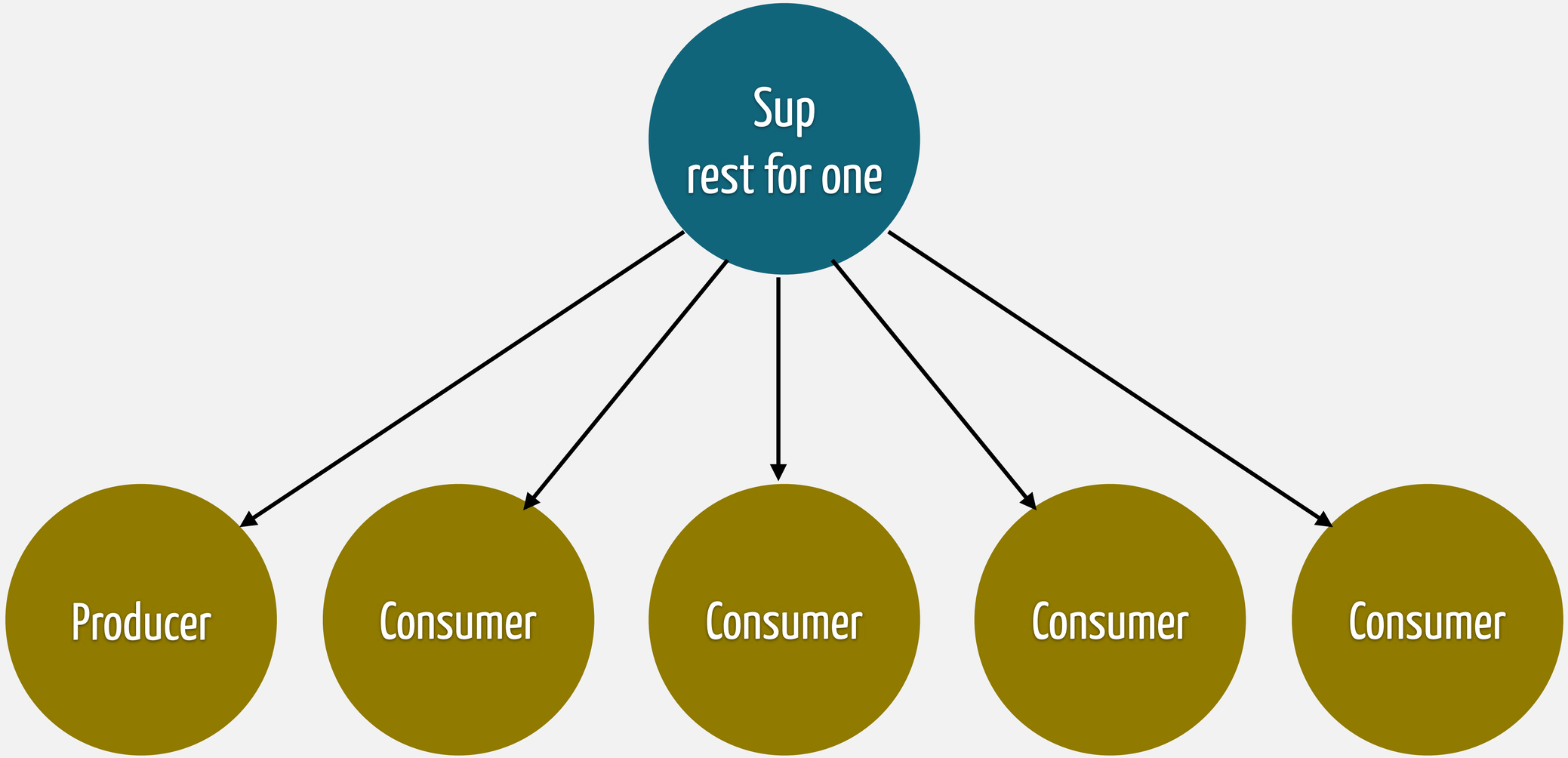
Data ingestion / Data processing

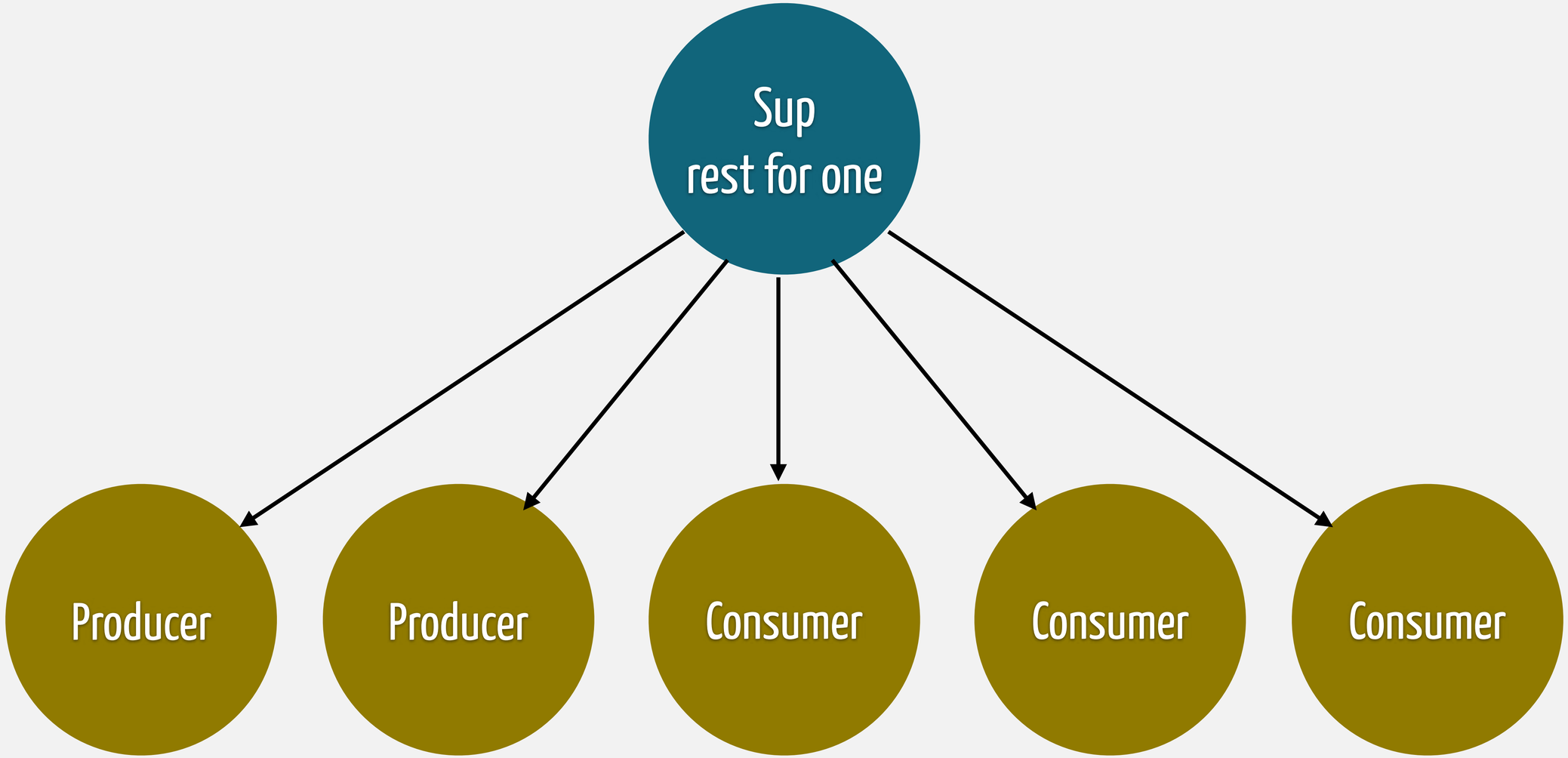
- Using GenStage for concurrency / back-pressure
- Many projects reimplementing the same feature set: rate limiting, batching, metrics, etc
- Woes regarding complex pipelines

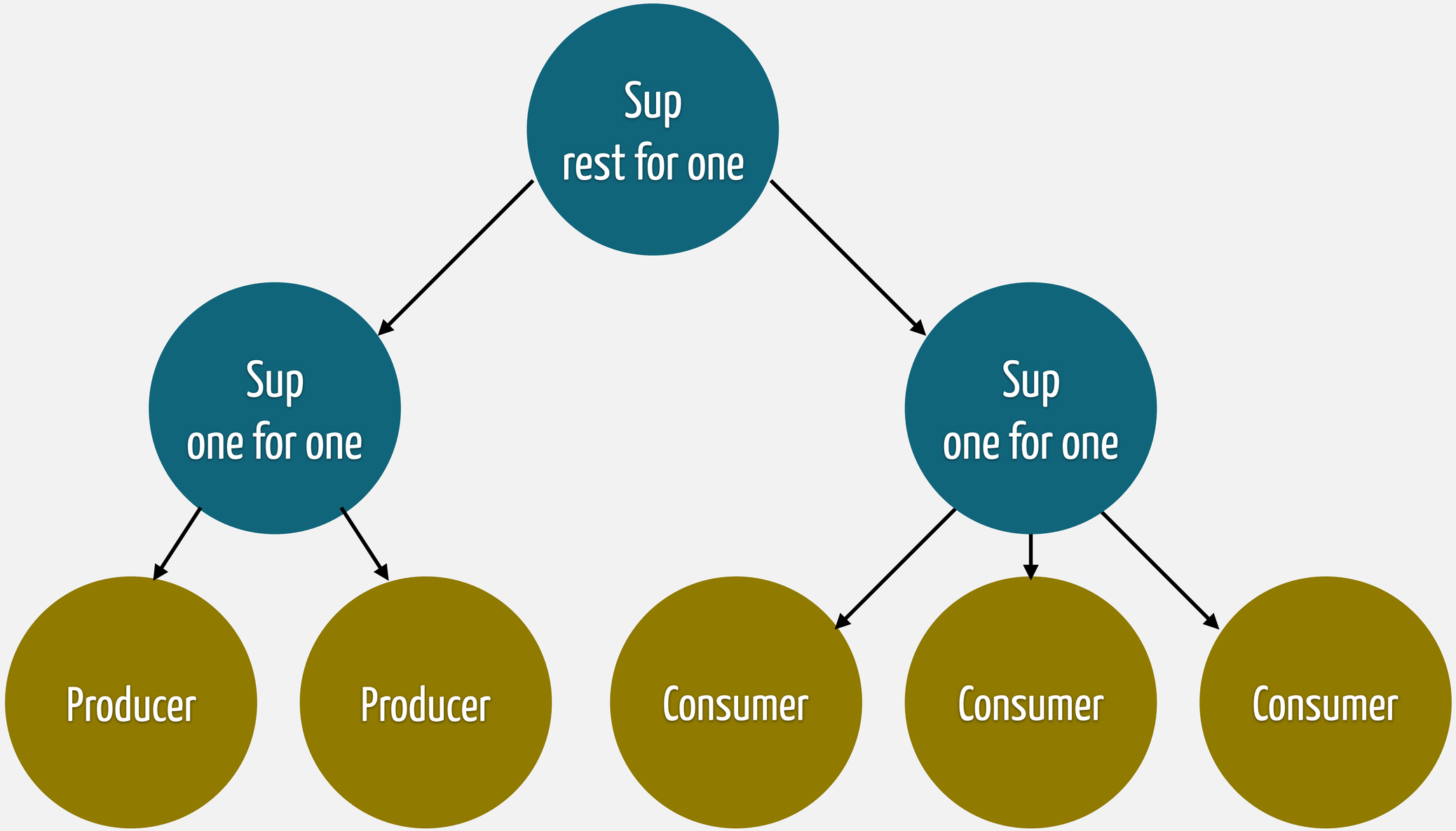
Complex GenStage pipelines

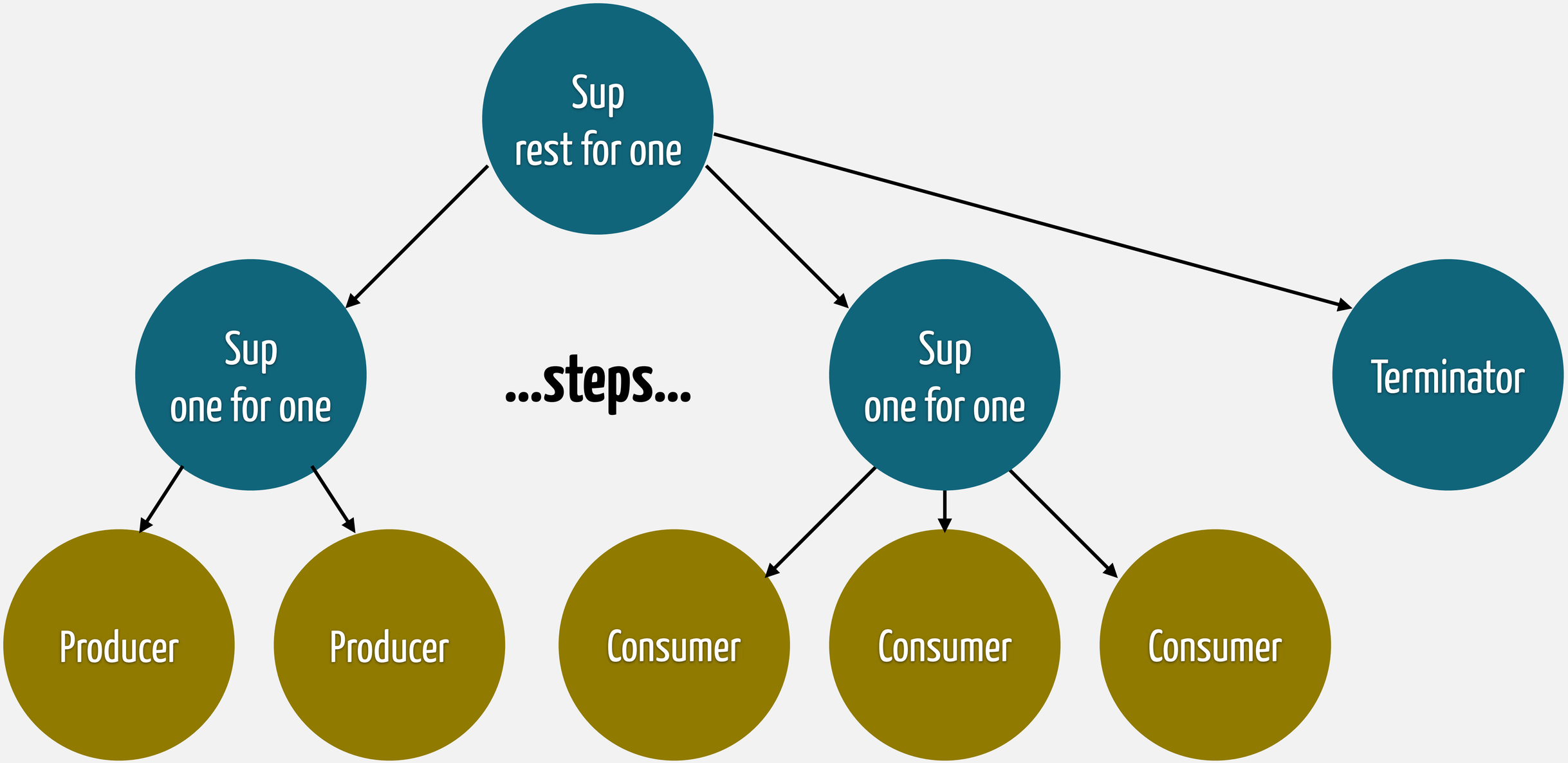



Sup
one for one
Producer
Consumer
Consumer
Consumer
Consumer


Sup
rest for one
Producer
Consumer
Consumer
Consumer
Consumer


Sup
rest for one
Producer
Producer
Consumer
Consumer
Consumer


Sup
rest for one
Sup
one for one
Sup
one for one
Producer
Producer
Consumer
Consumer
Consumer


Sup
rest for one
Sup
one for one
...steps...
Sup
one for one
Terminator
Producer
Producer
Consumer
Consumer
Consumer

Complex GenStage pipelines

- How to structure supervisions trees correctly?
- How to handle graceful shutdown without data loss?
- How to reduce the amount of events lost during failures?

Broadway

Broadway

A new library for concurrent and multi-stage data ingestion and data processing with Elixir

Broadway

- Back-pressure and concurrency
- Automatic acknowledgements at the end of the pipeline
- Fault-tolerance with minimal data loss
- Graceful shutdowns
- Batching and partitioning

Broadway

```elixir
defmodule MyApp.Broadway do
  use Broadway

  def start_link(options)
  def handle_message(procesor, message, config)
  def handle_batch(batcher, messages, batch_info, config)
end
```

vNEXT

- Batchless pipelines (for RabbitMQ and others)
- Metrics and statistics
- Back-off in case of failures
- etc

Your turn

- Give it a try: v0.1 is out!
- Write a Broadway producer for your favorite thing
  - BroadwaySQS is currently available
  - But you can also plug any GenStage producer

plataformatec

consulting and software engineering

Questions?

@plataformatec / plataformatec.com.br