DATA+AI SUMMIT 2022

Elixir: The wickedly awesome batch and stream processing language you should have in your toolbox

Brian Femiano

Senior Data Engineer. Ads Platform @ Apple

ORGANIZED BY databricks

What is Elixir?

- Dynamic functional programming language.
- Erlang virtual machine (BEAM) compatible bytecode.
- Leverage lightweight Erlang processes for distributed computing.
- Tooling ecosystem is about as dev-friendly as it gets.
- Erlang can be a difficult onboard ramp for nOObs.
- Elixir is the exact opposite but still gets to use the Erlang VM. Fantastic community.

Why Elixir for data engineering?

- Actor model helps facilitate scalable distributed communication.
- Immutability on data in memory.
- Pattern matching for analytics is very useful.
- Strong support for lazy evaluation over data structures.
- Agent/GenServer modules make managing state a breeze.
- Error handling/Fault tolerance is both robust and graceful.
- Erlang VM is extremely battle-tested and hard to crash. Great for constant-running applications.

Services architecture

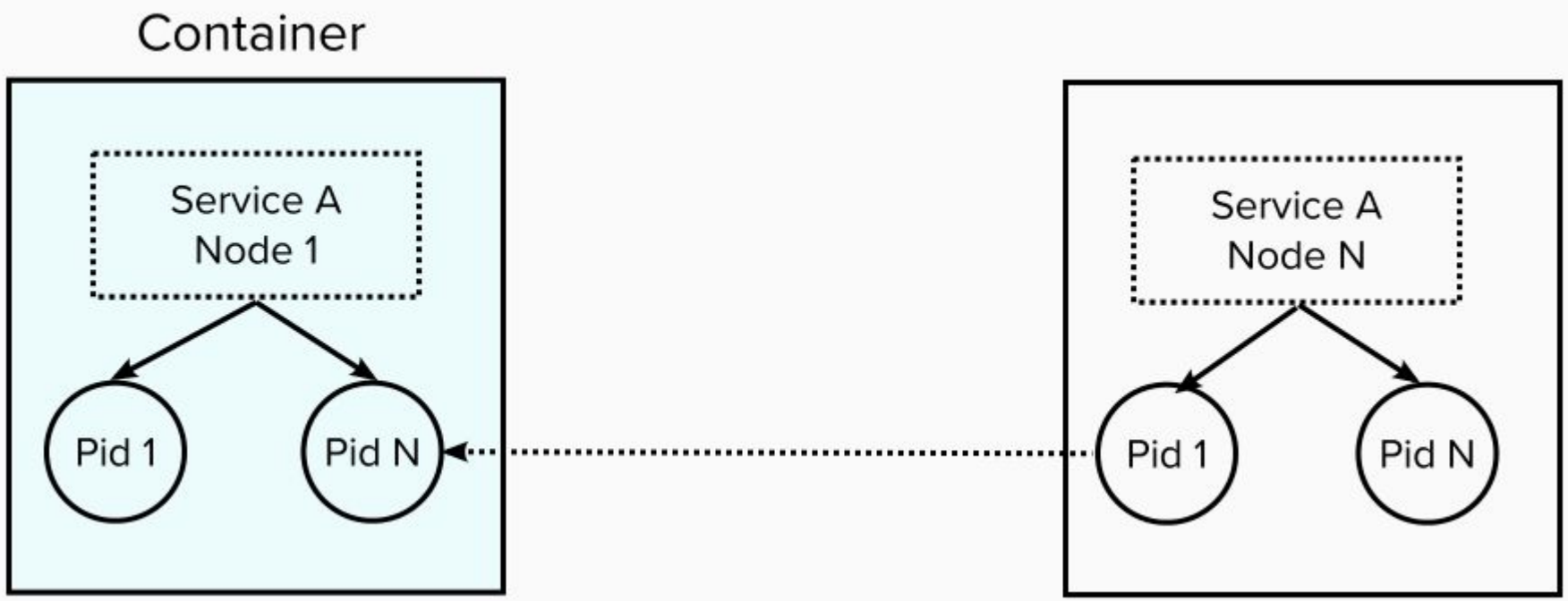
- Services configured as Erlang clusters with N nodes.
- Nodes deployed on containers.
- Nodes running the service will spawn Erlang processes.
- Processes can easily communicate across container boundaries.
- Caravan: library to help Erlang/Elixir processes communicate in a container setting with Consul.
- Airflow tasks that sends HTTP POST to our processes.
- Configure Elixir workloads as Airflow DAG tasks.
- Airflow sensor polls processes every until either:

  A) Success response

  B) Failure response

  C) Timeout.

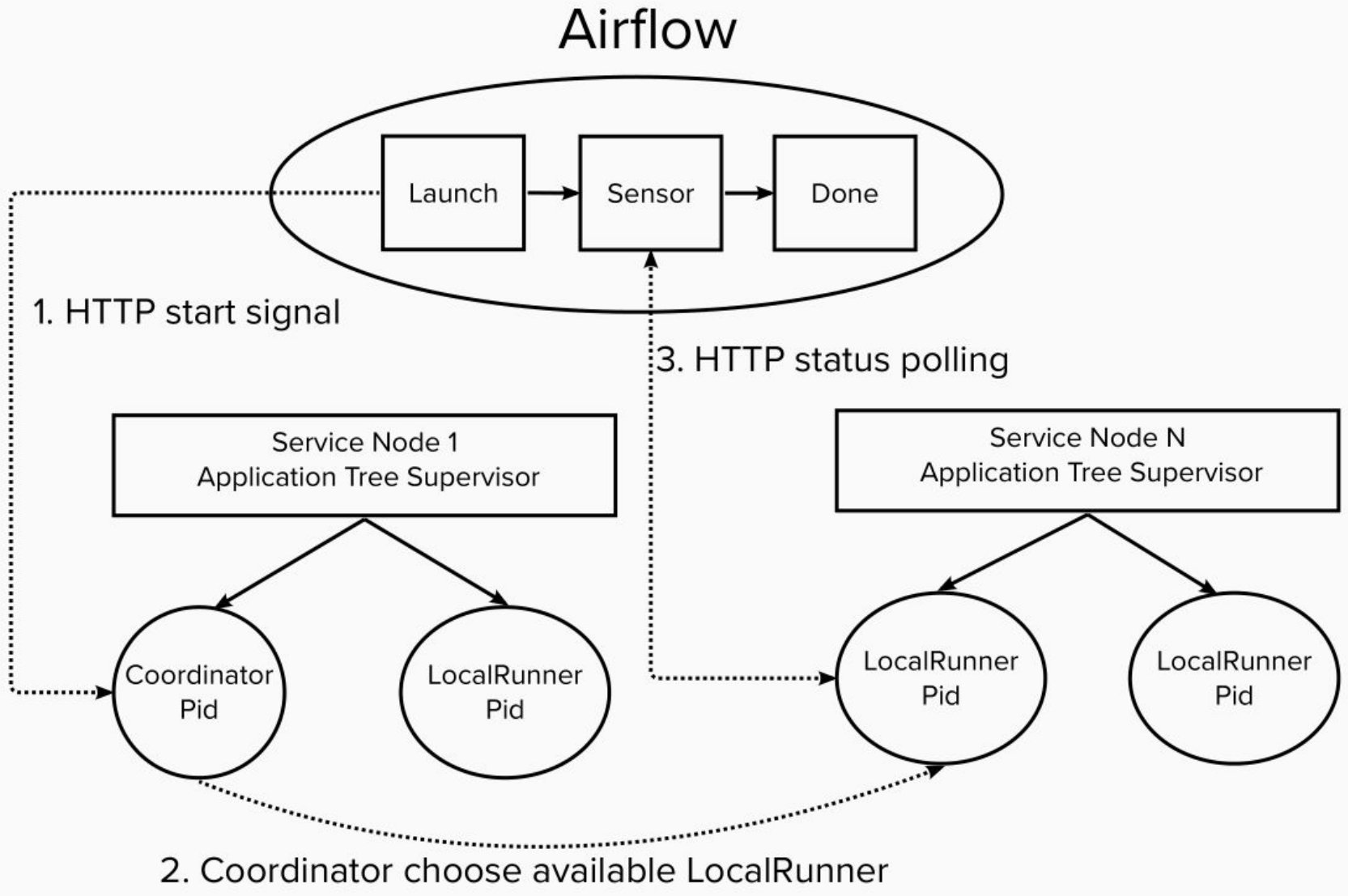
Services architecture



ScheduledJob framework written in Elixir

- Handles coordination of jobs across the Erlang nodes in a service cluster, rerunning failed jobs and persisting of status logs.
- Does not handle triggering jobs or inter-job dependencies. Designed to be triggered by an HTTP POST from an Airflow operator.
- Unchecked errors automatically flag the scheduled job as failed.
- Coordinator service interacts with different LocalRunners across the Erlang cluster as a locking service to ensure only one concurrent instance of a job is spawned and running.

Steps of a scheduled job workflow



Wiring up scheduled job

```
defmodule Notifications.Jobs do
   def job_configs do
      %{
         ingest_job: %{
            retry_policy: %{retries: 2},
            default_arguments: [~T[11:00:00]]
         }
      }
```

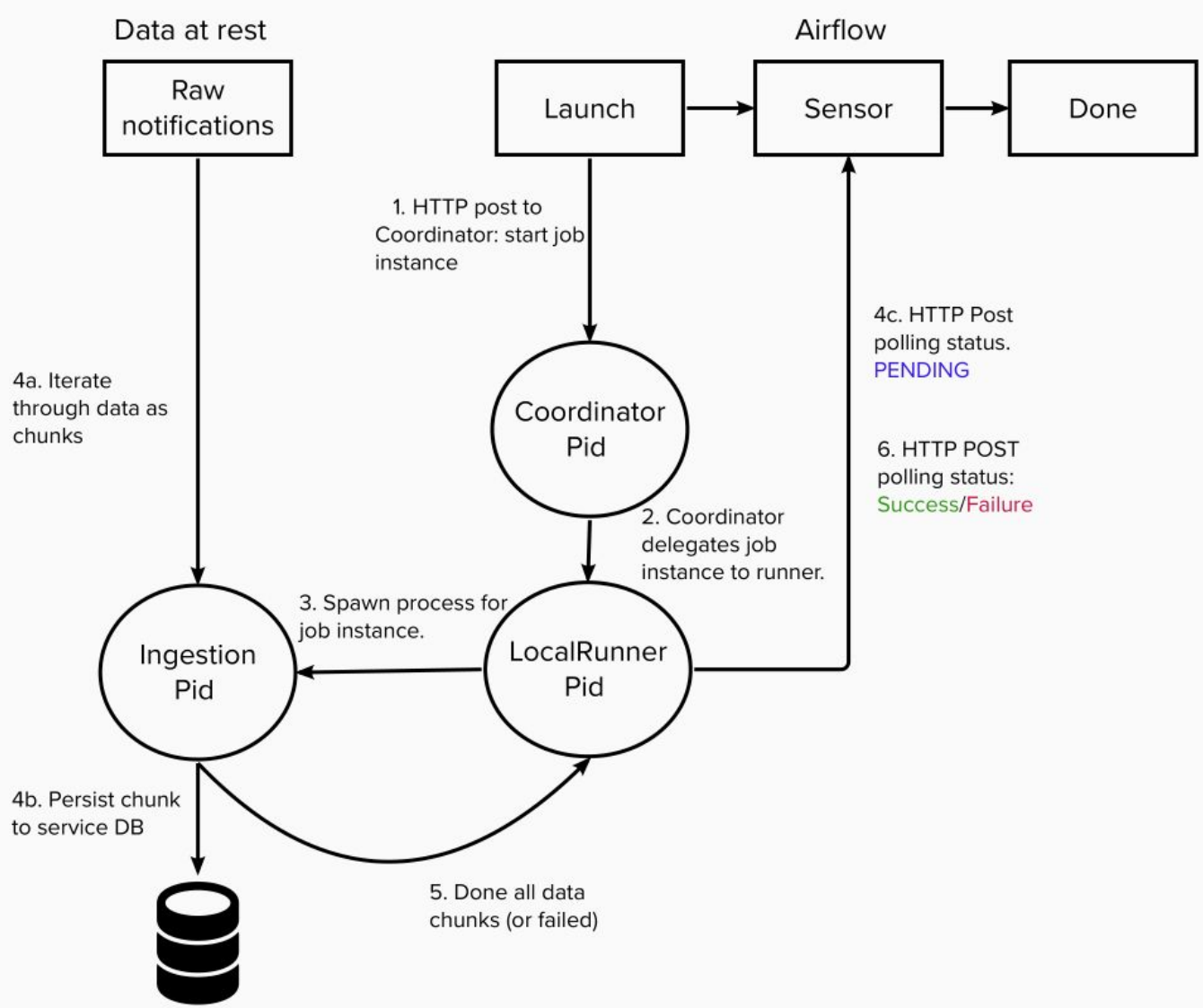
Scheduled jobs in Application start()

```
def start(_type, _args) do
  # List all child processes to be supervised
  children = [
    {ScheduledJob,
     [
        job_providers: [[module: Notifications.Jobs, config_provider:
:job_configs]],
       repo: Notifications.Repo # Ecto database operations module.
     ]}
  ]
  Supervisor.start_link(children, opts)
end
```

Case study: Notification view analytics

- We want to know which notifications are actually get viewed.
- Information is collected in raw logs.
- We want this information structured and stored in our notifications microservice database for fast retrieval by the service API layer.
- Why couple the ingest code directly with the microservice?
  1. The service API layer contains quite a bit of code we'd like to reuse for ingest transformation.
  2. We'd like to keep write access to the database restricted to just the service.

Case study: Notification view analytics



Case study: Notification analysis

```
def run(conn, scheduled_at) do
  with_timing( #telemetry helper method. Stopwatch to measure how long the first function takes to execute.
    fn -> # inline function that does the iterative chunk persistence.
       day = get_day_from_schedule(scheduled_at)
       unless upstream_ready?(day) do
         raise "Upstream data is not available yet"
       end
       objs = Storage.list_objects_with_prefix(conn, "data_directory", "day=#{Date.to_string(day)}")

       Storage.stream_objects_data(conn, objs)
         |> Stream.chunk_every(1000)
         |> Stream.map(fn lines -> parse_lines_batch(lines, day) end)
         |> Enum.map(&write_batch_to_db/1)
   end,
   &capture_telemetry/2 #callback to measure output metrics
  )
```

Stream processing

Notifications service also needs listens to several Kafka topics.

KafkaEx: Elixir client with support for Kafka 0.8+
https://hexdocs.pm/kafka_ex/readme.html

Setting up Kafka Ex

1. Add mix dependency to build.

2. Setup supervisor module to listen to consumers.

3. Wire supervisor into Application supervision tree.

4. Define different consumer_impl implementations.

Supervisor module to listen on consumers

```elixir
def start_link(args) do
 Supervisor.start_link(__MODULE__, args, name: __MODULE__)
end

@impl true
def init(_args) do
 Supervisor.init(build_child_specs(Application.get_env(:app_name, :consumers),
 strategy: :one_for_one)
end

def build_child_specs(configs) when is_list(configs) do
 configs
 |> Enum.map(&validate_config!/1)
 |> Enum.map(&build_child_spec/1)
end
```

Supervisor module to listen on consumers

```
defp build_child_spec(config) when is_list(config) do
   consumer_group_args = [
     # the implementation of KafkaEx.GenConsumer - this module does all the work
     Keyword.fetch!(config, :consumer_impl),
     Keyword.fetch!(config, :consumer_group_name),
     [Keyword.fetch!(config, :topic_name)],
     [
       uris: Config.kafka_uris(),
       auto_offset_reset: :earliest
     ]
   ]
   %{
     id: supervisor_name,
     start: {KafkaEx.ConsumerGroup, :start_link, consumer_group_args},
     type: :supervisor
   }
end
```

GenServer consumer

```elixir
defmodule CreatorNotifications.CuratedProgram.Consumer do
  use KafkaEx.GenConsumer
  alias KafkaEx.Protocol.Fetch.Message
  @impl true
  @spec handle_message_set([Message.t()], term) :: {:async_commit, term}
  def handle_message_set(messages, consumer_state) do
      messages
      |> Enum.map(&decode_avro_message(&1.value))
      |> ingest_messages()
      {:async_commit, consumer_state}
  end
 end
```

DATA+AI
SUMMIT 2022

Thank you

Brian Femiano

Ads Platform @ Apple

References

The Erlang programing language: https://www.erlang.org/

The Elixir programming language: https://elixir-lang.org/

Elixir libs: KafkaEx:
https://hexdocs.pm/kafka_ex/readme.html#usage-examples

Caravan: https://hexdocs.pm/caravan/Caravan.html

Consul: https://www.consul.io/