ROBUST DATA PROCESSING

PIPELINE WITH ELIXIR AND FLOW

LÁSZLÓ BÁCSI, 100STARLINGS

LACKAC @ICANSCALE

BIT.LY/EX18-FLOW

DISCLAIMER

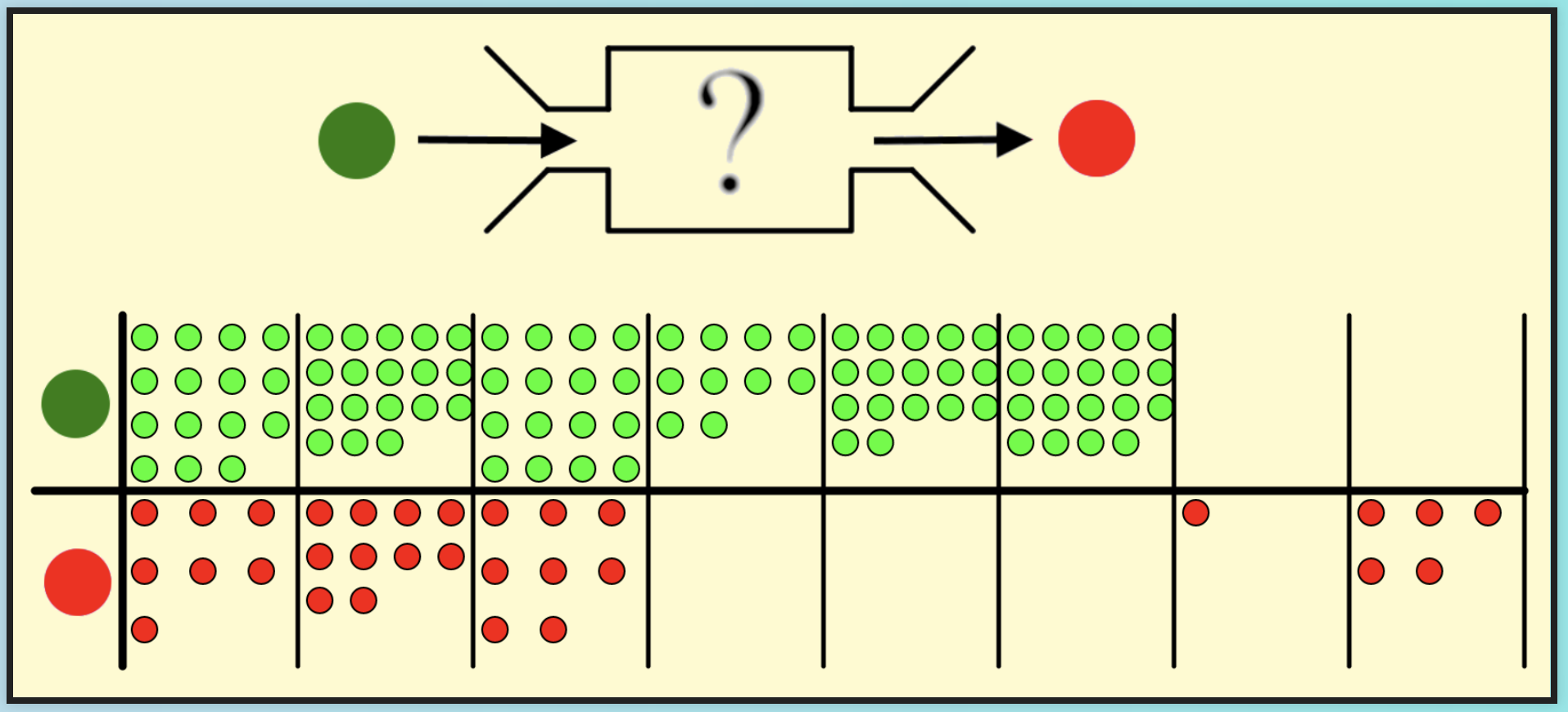
- first Elixir talk
- talk is based on uncompleted work
- based on client work*
- pipeline target is AWS Redshift
- all pipelines are custom

* CollectPlus: parcel service in the UK

AGENDA

- Mental Model of a Pipeline
- Elixir Flow
- Maintainability
- Failure Tolerance
- Scheduling

WHAT DOES THE MACHINE DO?

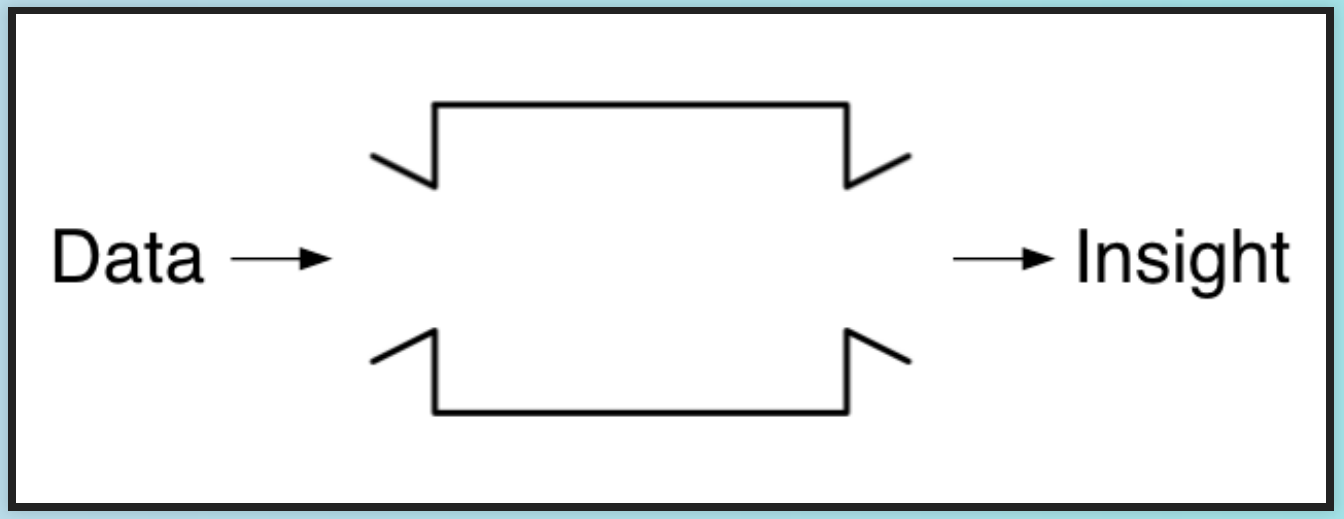


100 STARLINGS

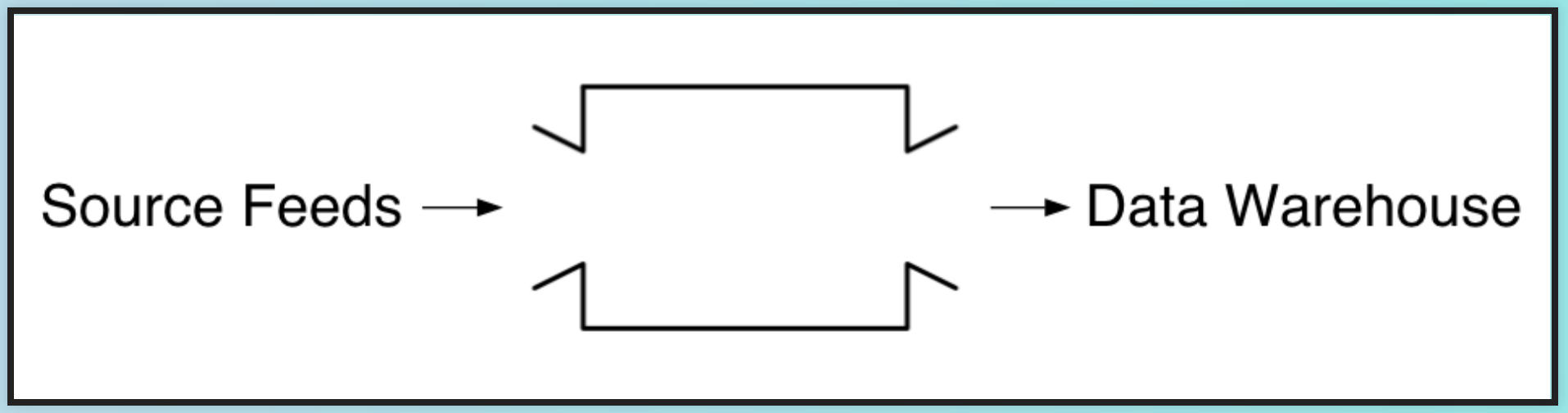
MENTAL MODEL OF A PIPELINE

Photo by Scott Webb on Unsplash

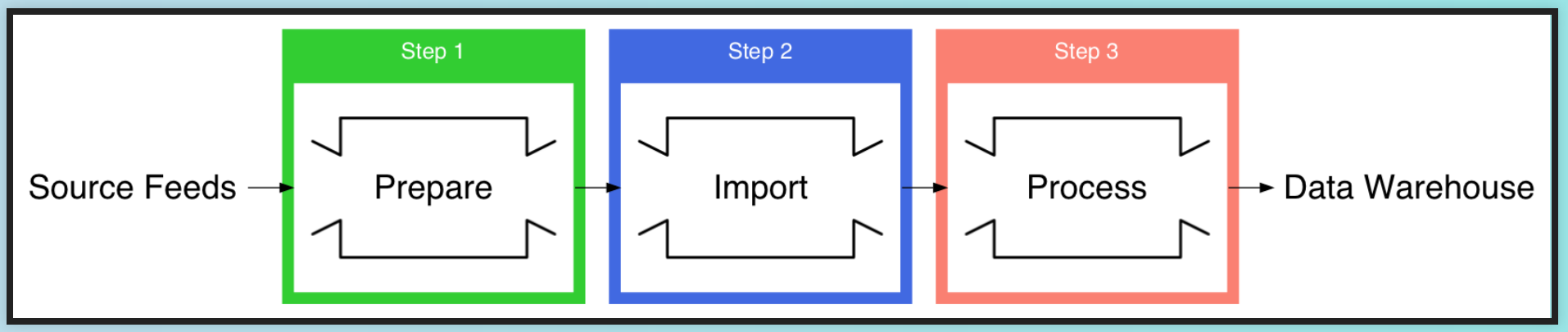
APPROACHING A DATA PROJECT



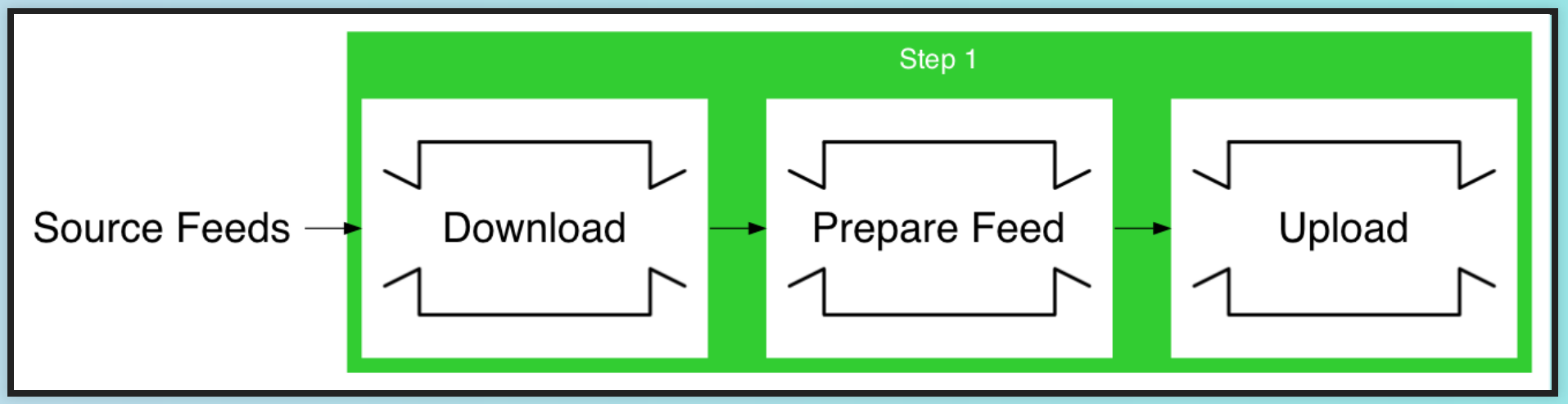
GOING OUTSIDE IN



ETL, ELT, POTAYTO, POTAHTO



LET IT FLOW



GLIMPSE OF A FLOW

```elixir
def flow(input) do
  input
  |> Flow.from_enumerable(opts)
  |> Flow.filter(&feed_file?/1)
  |> Flow.map(&get_feed/1)
  |> Flow.map(&prepare_feed/1)
  |> Flow.map(&upload_feeds/1)
  |> Flow.partition(stages: 1, window: opts[:window])
  |> Flow.reduce(fn -> [] end, fn feed, acc -> [feed | acc
  |> Flow.emit(:state)
end
```

IT'S A SERIES OF TUBES

PIPES, PIPES, AND MORE PIPES

```
def prepare_feed({key, contents}) do
  feed_id = Path.basename(key, ".csv")

  contents
  |> String.splitter("\n")
  |> Stream.with_index()
  |> Stream.map(fn {line, i} ->
    Enum.join([String.trim(line), feed_id, i + 1], ",")
  end)
  |> Enum.reduce(%Feed{}, &collect_feed_rows/2)
end
```

FLOW CRASH COURSE

Photo by paul morris on Unsplash

RESEARCHING ELIXIR FLOW

Elixir Of Love - Cocktail Flow

www.cocktailflow.com/cocktail/elixir-of-love ▾

Ingredients. 1 part white rum; 2 parts Amaretto; 1 part White Crème de Cacao; 2 parts half & half; ice cubes. How to Mix. fill up the glass with ice; fill up the shaker with ice; pour half & half,White Crème de Cacao,Amaretto,white rum into the shaker; shake well; strain into the old fashioned glass; garnish with sprinkled ...

FLOW Elixír. Drognak számít? (1707766. kérdés) - Gyakori kérdések

https://www.gyakorikerdesek.hu/szorakozas__egyeb-kerdesek__17... ▾ Translate this page

Apr 12, 2011 - Mi is most voltunk bulizni és kaptunk a bulin fejenként egy **Flow elixírt**. A mi társaságunknak (voltunk 9-en) nagyon bejött! Az elöttem szólók valószínű elfogyaszthattak valami mást is az elixíren kívül. Ha mást nem akkor lehet, hogy belecsepegtettek valamit az italukba- manapság soha nem lehet tudni.

SO WHAT *IS* ELIXIR FLOW?

Flow allows developers to express computations on collections, similar to the Enum and Stream modules, although computations will be executed in parallel using multiple GenStages.

COUNTING WORDS – THE CANONICAL EXAMPLE

```
File.stream!("path/to/some/file")
|> Enum.flat_map(&String.split(&1, " "))
|> Enum.reduce(%{}, fn word, acc ->
  Map.update(acc, word, 1, & &1 + 1)
end)
|> Enum.to_list()
```

THE SAME WITH `Stream`

```
File.stream!("path/to/some/file")
|> Stream.flat_map(&String.split(&1, " "))
|> Enum.reduce(%{}, fn word, acc ->
  Map.update(acc, word, 1, & &1 + 1)
end)
|> Enum.to_list()
```

AND NOW WITH FLOW

```
File.stream!("path/to/some/file")
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split(&1, " "))
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, acc ->
  Map.update(acc, word, 1, & &1 + 1)
end)
|> Enum.to_list()
```

READ THE DOCS

EXCELLENT AS USUAL

MAINTAINABILITY

- Composition
- Branching flows

COMPOSITION

- Two modes of operation
  - Historical
    - prepare, import, and process all data from 2012 to 2015
    - reimport and reprocess everything for year 2016
  - Real-time
    - monitor source for new files
    - run incoming data through the whole pipeline

COMPOSABLE FLOWS

```elixir
defmodule ImportFeeds do
  @spec flow(Enumerable.t()) :: Flow.t()
  def flow(%Flow{} = flow) do
    flow
    |> Flow.map(&ensure_loaded/1)
    # |> ...
    |> Flow.emit(:state)
  end

  def flow(input) do
    input |> Flow.from_enumerable() |> flow()
  end
```

NOW THIS WORKS

```
feeds
|> PrepareFeeds.flow()
|> ImportFeeds.flow()
|> Flow.run()
```

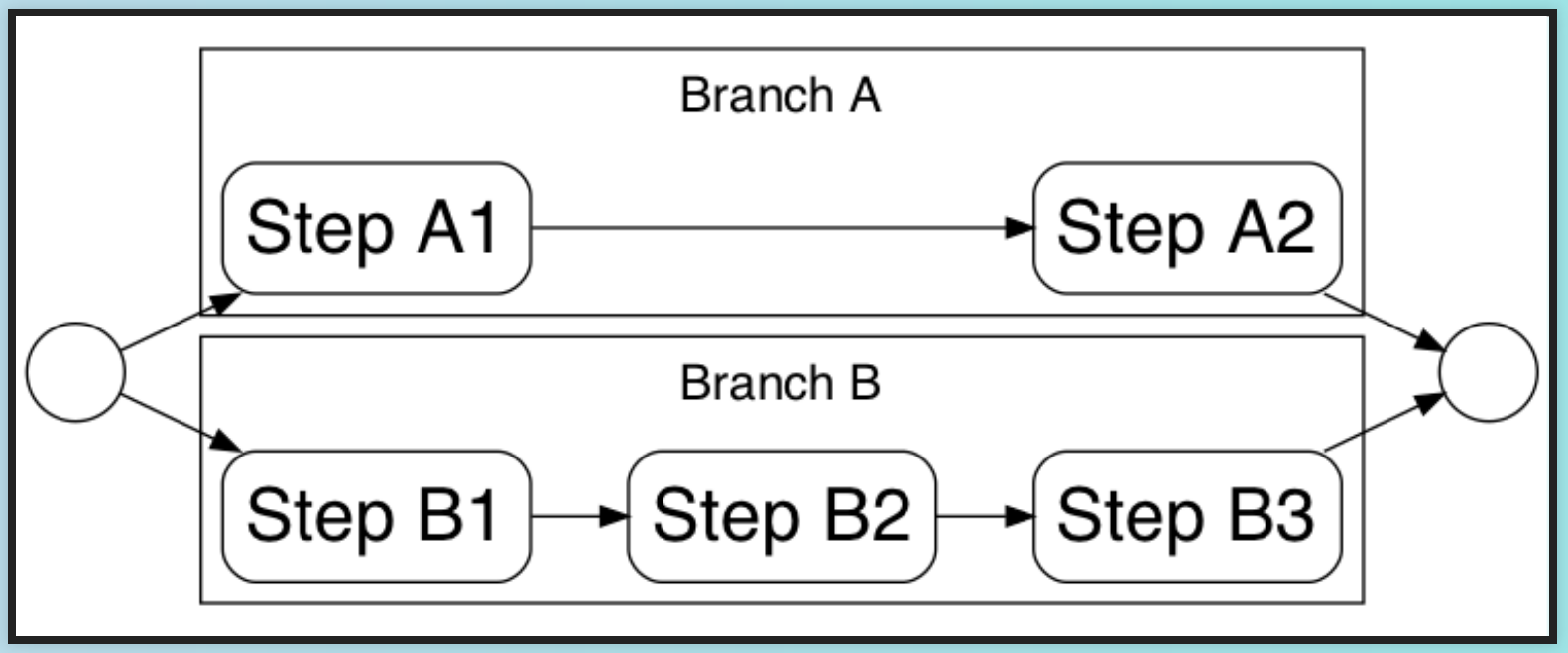
AS DOES THIS

```
batch_ids
|> ImportFeeds.flow()
|> PorcessEvents.flow()
|> Flow.run()
```

WE MAY ALSO EXTEND THIS TO SUPPORT DIFFERENT KIND OF SOURCES

```
def flow(stage) when is_pid(stage) do
  stage |> Flow.from_stage() |> flow()
end
```

BRANCHING



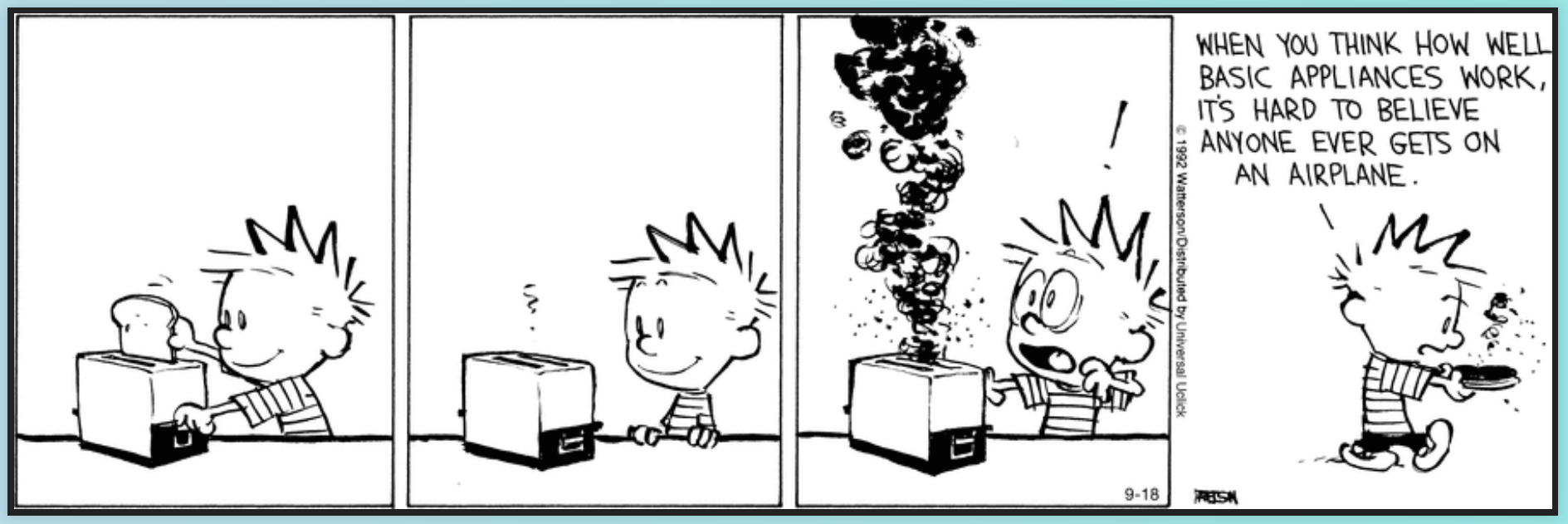
MULTIPLE CASES AND STRATEGIES

- keep it inside the Flow for simple cases especially if you need to join the data together again later
- use `Flow.into_stages/3` for complex situations

BRANCHING EXAMPLE

```
flow
|> Flow.flat_map(&[{&1, :type_a}, {&1, :type_b}])
|> Flow.partition(stages: 2, max_demand: 1,
  hash: fn
    {_, :type_a} = event -> {event, 0}
    {_, :type_b} = event -> {event, 1}
  end
)
|> Flow.map(&import_type/1)
|> Flow.map(&mark_imported/1)
|> Flow.partition(stages: 1, window: window)
|> Flow.group_by(&elem(&1, 0).id, &elem(&1, 1))
```

FAILURE TOLERANCE



TWO SIDES

ERROR HANDLING

SUPERVISION

SUPERVISING ELIXIR FLOWS

```
FeedStorage.list_feeds(opts)
|> SplitFeeds.flow(window: window)
|> Flow.start_link()
```

ADDING A FLOW TO A SUPERVISION TREE

```
def start_prepare_flow(opts) do
  {window, opts} = Keyword.pop(opts, :window)
  FeedStorage.list_feeds(opts)
  |> SplitFeeds.flow(window: window)
  |> start_flow_supervised()
end
```

CREATE A DynamicSupervisor

```
# in your supervisor
children = [
  # ...,
  {DynamicSupervisor, name: FlowSupervisor, strategy: :one
]
```

START AN ARBITRARY FLOW SUPERVISED

```
defp start_flow_supervised(flow) do
  child_spec = %{
    id: make_ref(),
    start: {Flow, :start_link, [flow]},
    restart: :temporary,
    type: :supervisor
  }

  DynamicSupervisor.start_child(FlowSupervisor, child_spec
end
```

ERROR HANDLING STRATEGIES

- Let it crash
- Flag and filter
- Branch

Photo by Daniel Tausis on Unsplash

LET IT CRASH

Works well if you have an idempotent pipeline you can just restart, or if the occasional crash has a very low impact.

FLAG AND FILTER

```
flow
|> Flow.map(&get_feed/1)
|> Flow.map(&prepare_feed/1)
|> Flow.map(&upload_feeds/1)
|> Flow.filter(&filter_and_log/1)

def prepare_feed({:ok, feed}), do: do_prepare_feed(feed)
def prepare_feed({:error, _} = error), do: error

def filter_and_log({:ok, _}), do: true
def filter_and_log({:error, err}) do: log(err) && false
```

BENEFITS

- minimal impact on the pipeline structure
- failure handling checkpoints make it easier to see what's going on
- recommended to put checkpoint just before reduce step so that it deals with clean data

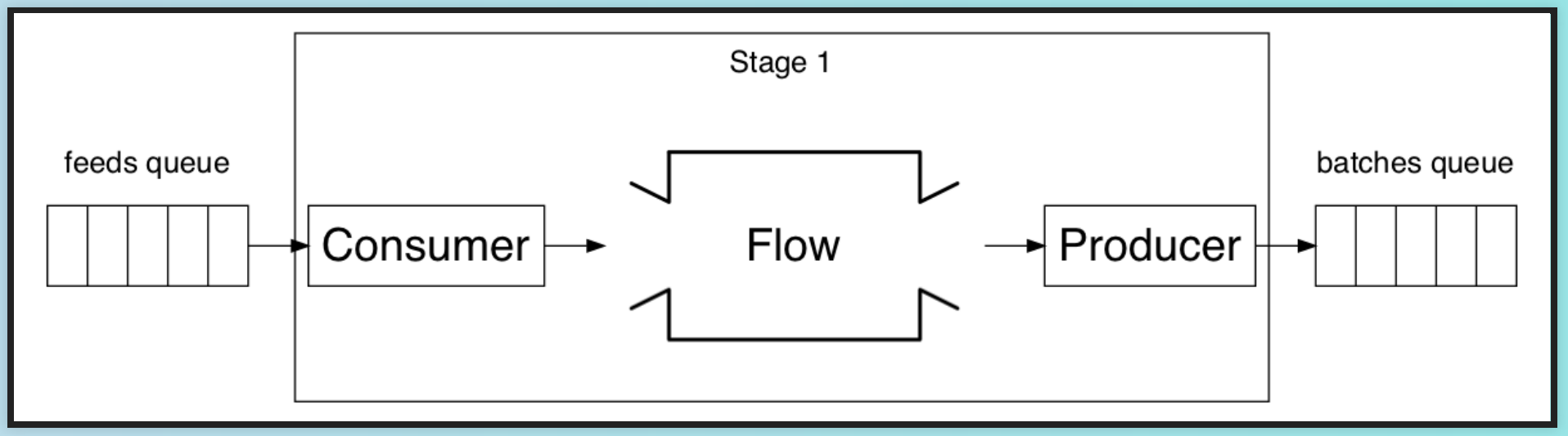
BRANCHING ERRORS

- A combination of "Flag and Filter" and the "Branching flows" approach
- Branch `{:ok, value}` and `{:error, error}` into two partitions
- Only advised for complex error handling situations and when the number of errors is in the same ballpark as the success values

SCHEDULING

- Simple scheduling is easy to achieve based on our composing functions (e.g. `start_prepare_flow/1` from earlier)
- for real-time processing we need something more
- looking to use RabbitMQ there

SCHEDULING FLOWS WITH RABBITMQ



STILL TBD

- The RabbitMQ consumer would be a GenStage producer for the flow
- And the RabbitMQ producer could be a GenStage consumer of the same flow
- This is still just on the drawing board
- Just the other day found pma/wabbit on GitHub that looks interesting we might be able to use

SUMMARY

- Elixir with Flow is more than just parallel processing
- Break up a complex flow into composable steps
- Handle errors at the right place and supervise flows
- Combine flows with RabbitMQ for scheduling

THANK YOU! FURTHER READING

- Flow Documentation
- Architecting Flow in Elixir Programs – Dr. René Föhring's excellent article series
- These slides