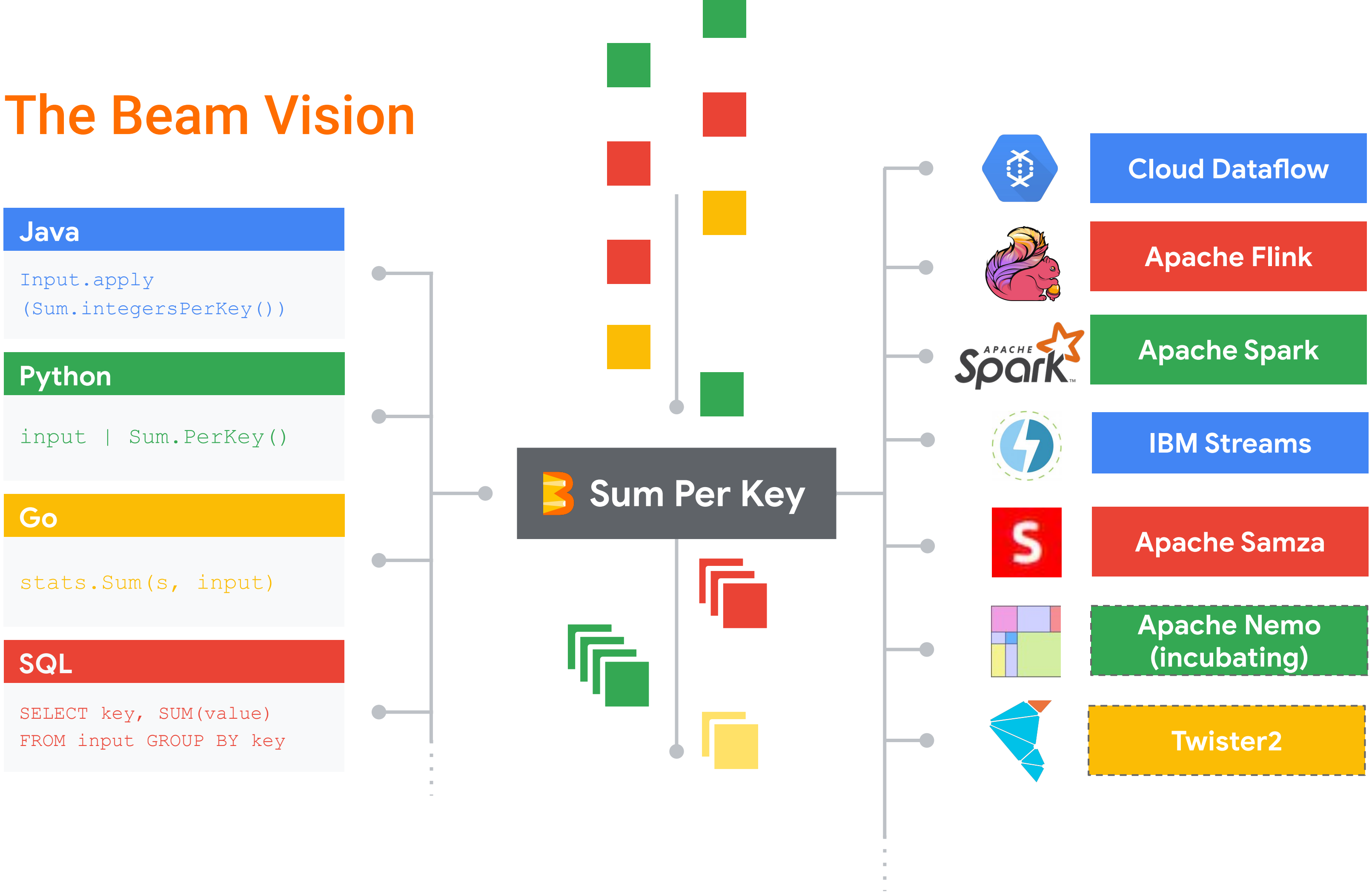
Runner Architecture, Management & Autotuning

Harsh Vardhan
ananvay@

The Beam Vision

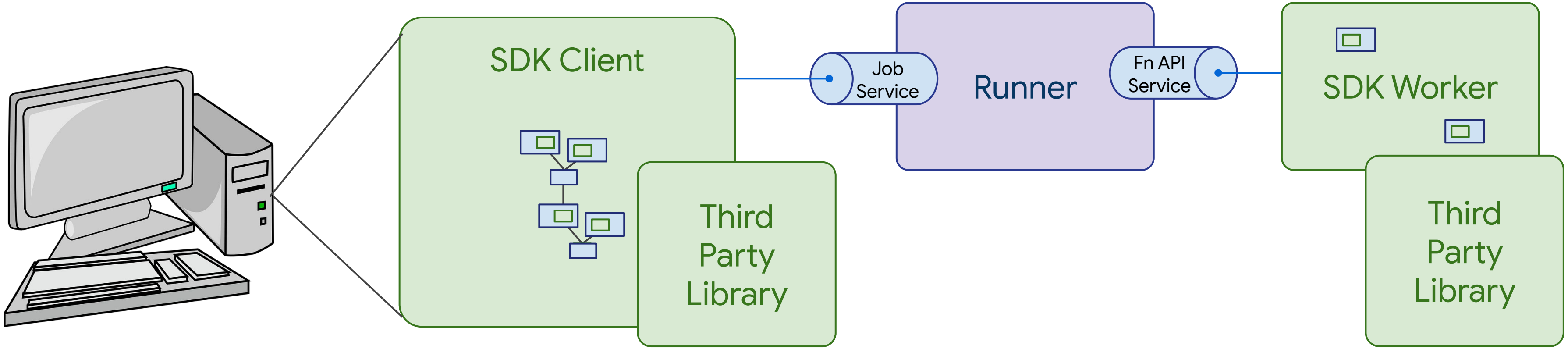
Provide a **comprehensive portability framework** for data processing pipelines, one that allows you to write your pipeline once in the **programming language of choice** and run it with minimal effort on the **execution engine of choice**

The Beam Vision



A choice of language: the SDK

- An **SDK** is the tool/library that lets the **user** author **Beam pipelines**...
- ...and **submit** it to a **runner**...
- ...with support for **executing user code** in an **appropriate environment**.



A choice of language: the SDK

- An SDK must be written for **each supported language**.
  - Provides **Beam Model Concepts** in a **language specific** way.

Java

```
Input.apply(Sum.integersPerKey())
```

Go

```
stats.Sum(s, input)
```

Python

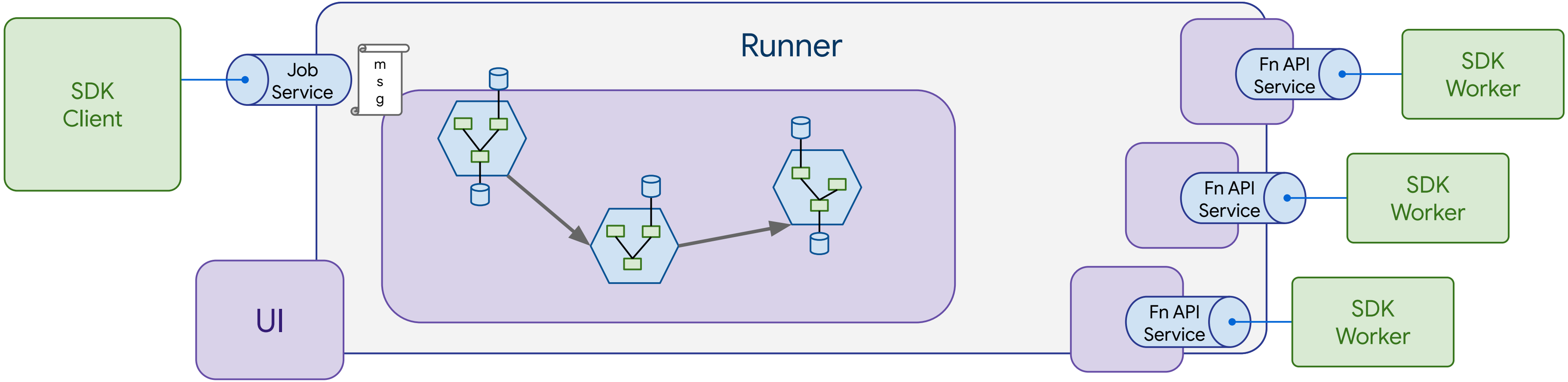
```
input | Sum.PerKey()
```

SQL

```
SELECT key, SUM(value)
FROM input GROUP BY key
```

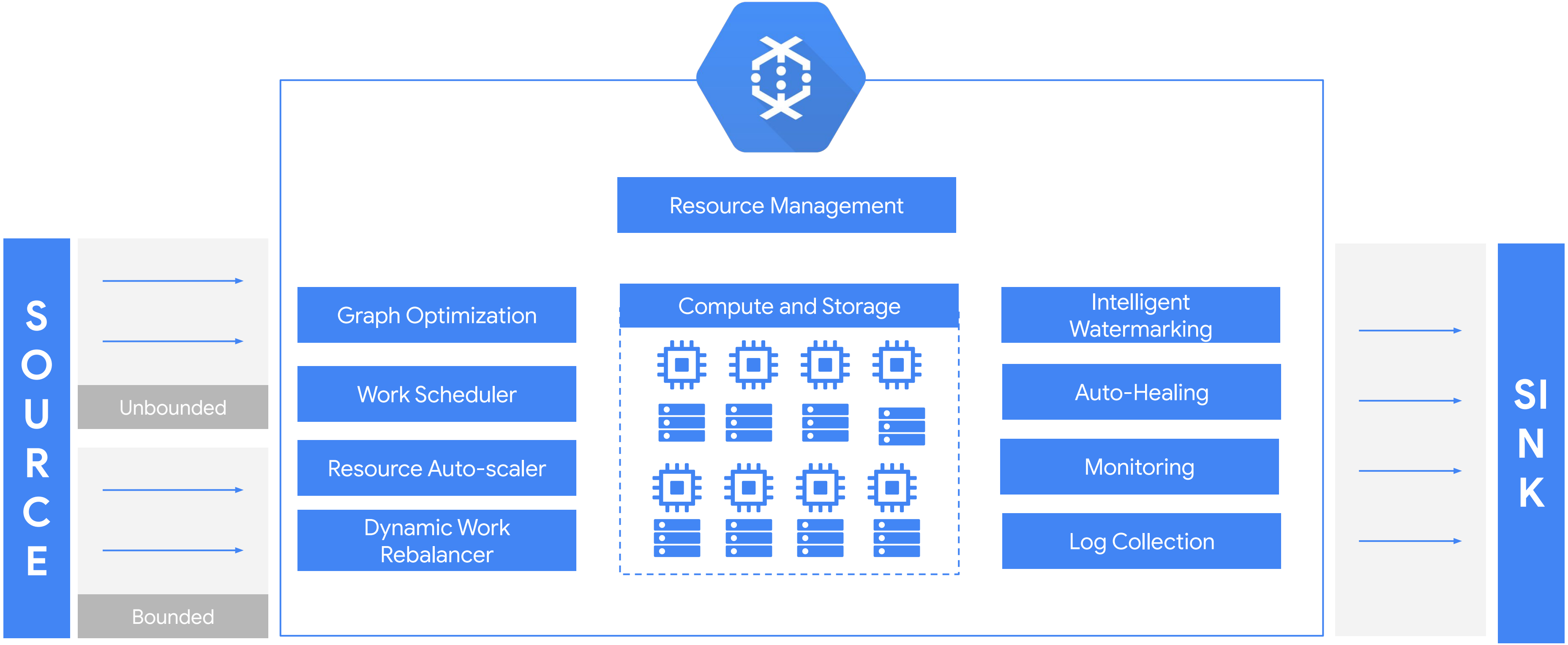
A choice of execution engine: the Runner

- A **Runner** orchestrates the **execution** of a pipeline...
- ...in a **distributed*** manner...
- ...while reporting **status** and **results** to the user.



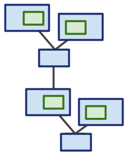
*There are also local runners for use in development and testing.

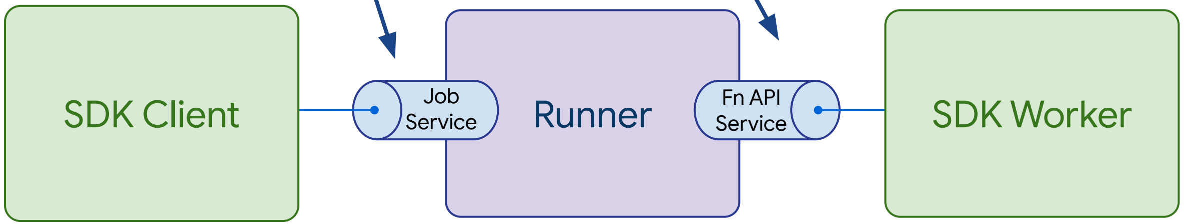
The Google Cloud Platform Runner: Dataflow



The Comprehensive Portability Framework

- A **language agnostic** way of **representing** Beam Pipelines...
    - ...and a **protocol** for **executing** them.



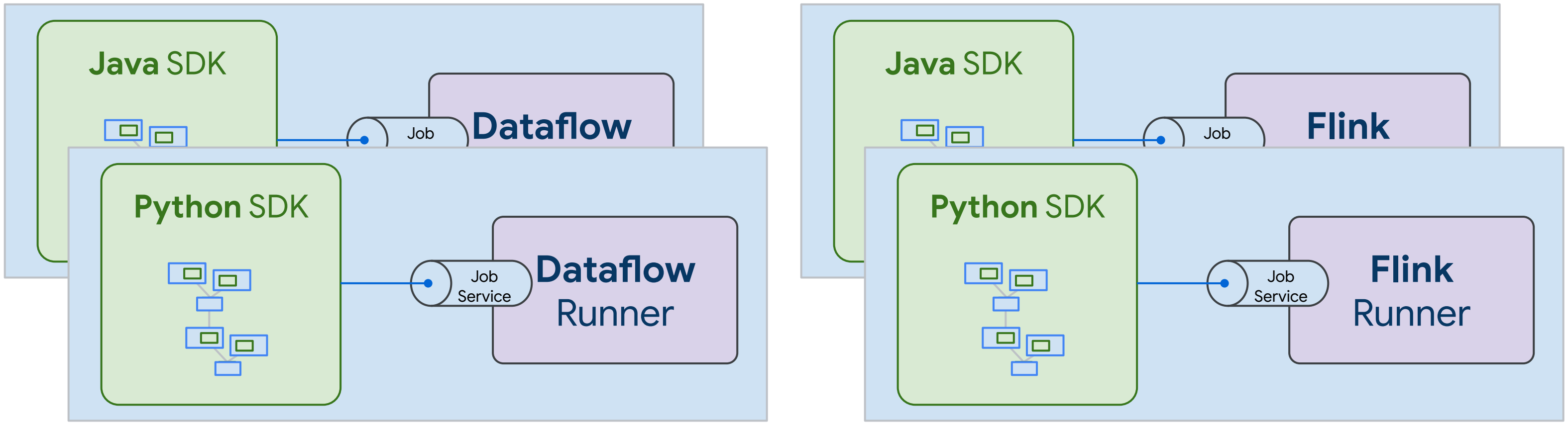


So, what's in it for me?

- **Every runner** works with **every language**
- Configurable, Hermetic **Worker Environment**
- **Multi-Language** Pipelines
- Faster/exclusive delivery of **new features**, such as Splittable DoFn (SDF)

Without Portability

- Pipelines have to be written in a single **language specific SDK**
- SDK-runner combinations require **non-trivial work** on both sides



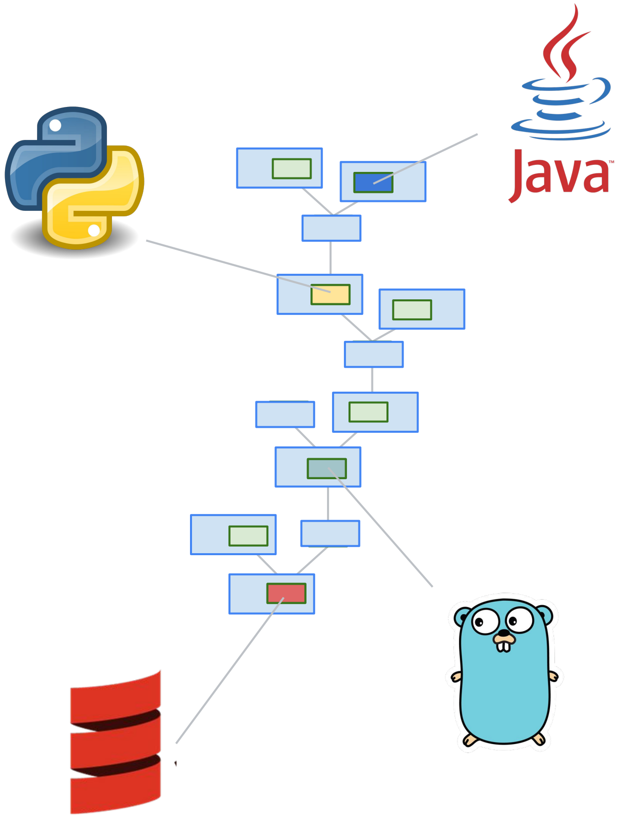
Hermetic Worker Environment

- Code deployed and executed on **remote** machines
  - Configuration is runner-specific, runner-constrained
- Shipping **dependencies** is hard
  - Especially in languages without fat jars
- Develop, test locally
  - Often **very different** than deployment environment

Beam Environments

- Each user operation has an associated environment in which to execute.
    - Typically the SDK provides a default environment
- This environment can be specified as an arbitrary Docker container
    - Ahead-of-time installation
    - Arbitrary dependencies
    - Arbitrary customization
    - Runner isolation
- Existing runtime injection of artifacts still supported
    - Jars, packages, binaries
    - Environments can be shared
    - No need to rebuild image on each compile

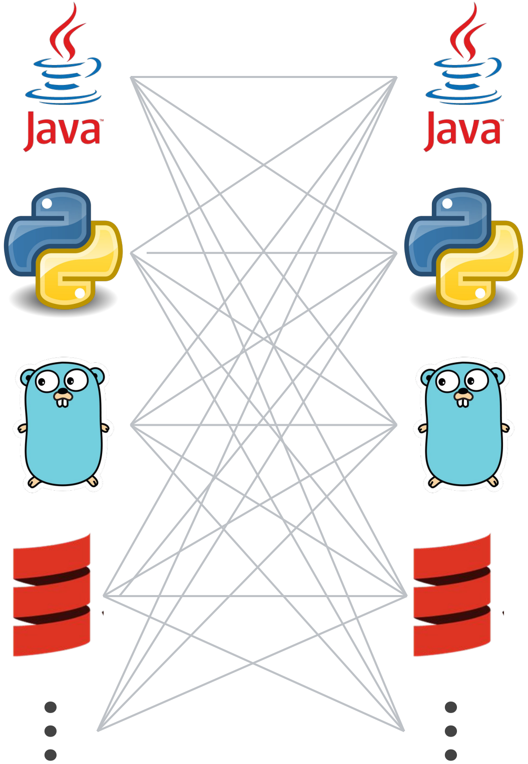
Cross-language transforms



Portability gives us

- **Language agnostic** representation of pipelines
  - Transforms, coders, ...
- Per-operation specification of **environment**
- We are **no longer bound to a single SDK** in a given pipeline

Cross-language transforms



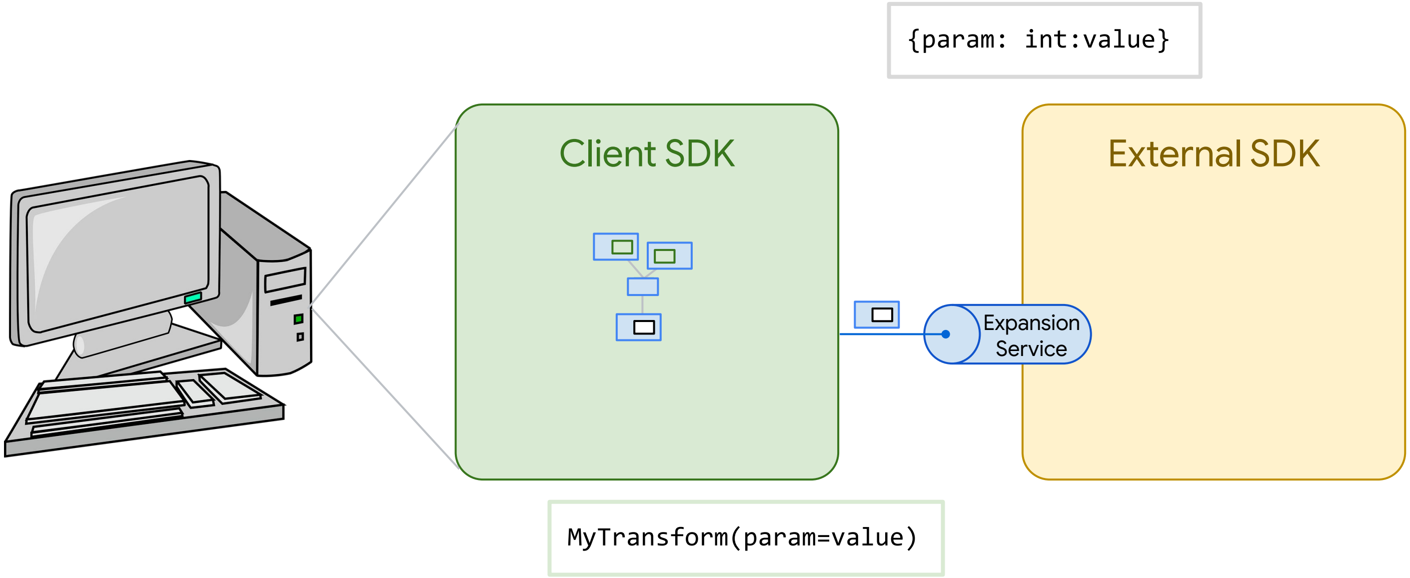
- Transforms can be **shared** among SDKs
- Rich set of **IOs** from Java **available everywhere**
- **Tensorflow** TFX transforms in non-Python jobs
- Leverage **SQL** work in Python and Go
- Bootstrap SDKs in **other languages**
- More libraries available in language of **your choice**.

Cross-language transforms

1 – User constructs pipeline using **SDK-native** conventions

2 – An **ExternalTransform** is applied

3 – The transform identifier, with its parameters is sent to an **ExpansionService**

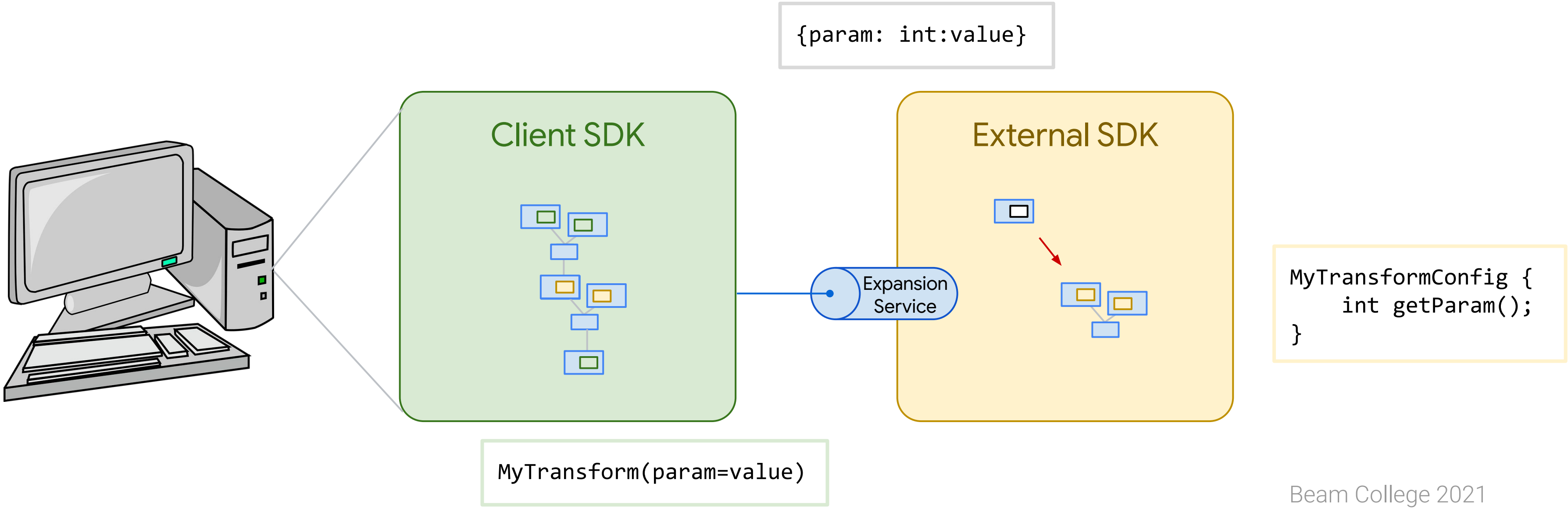


Cross-language transforms

4 – The transform is **expanded** in the external SDK.

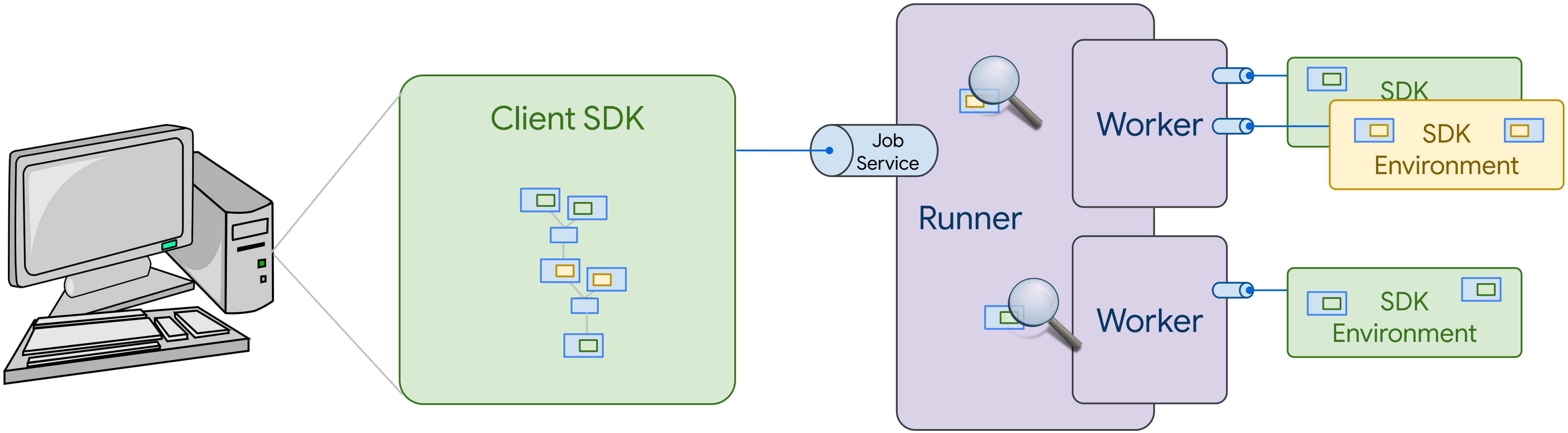
5 – The expansion is **returned** to the client SDK and **plugged** into the graph.

6 – Construction **continues as before**.



Cross-language transforms

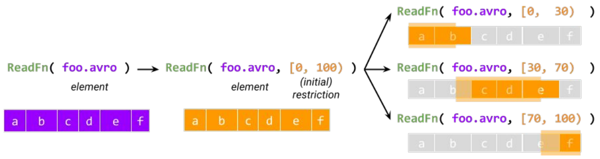
- **Execution** happens by interacting with **multiple environments**.

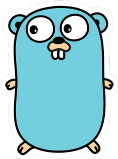


New Features

Some new features exclusively available via Portability

- **SplittableDoFn**
  - Radically Modular IO Connectors
- **Beam Metrics**
  - System metrics, richer user counter types
- **New runners and SDKs**
  - E.g. GoLang, Samza runner
- **Interactive, Visualization tools**



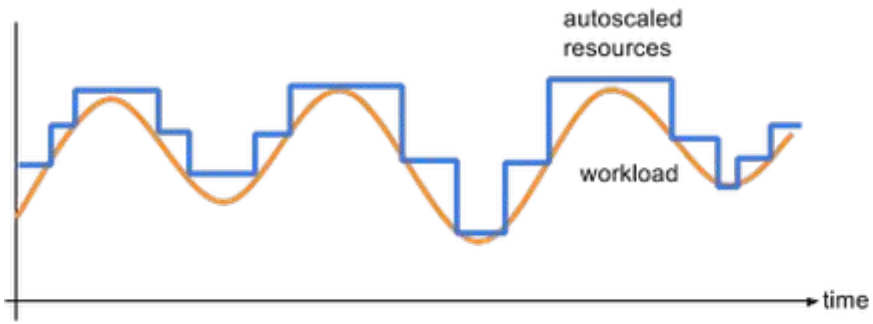


samza

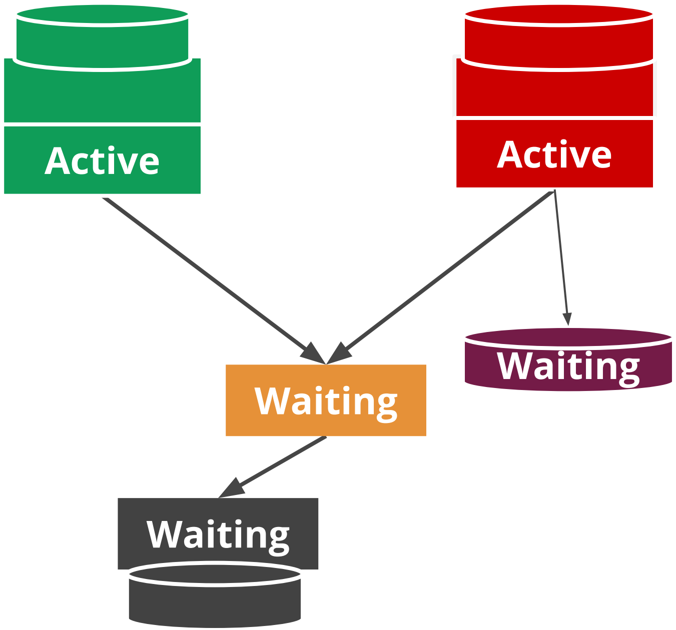
Portability is the future.

Autoscaling Mechanics (Google Cloud Dataflow)

- Batch Work items

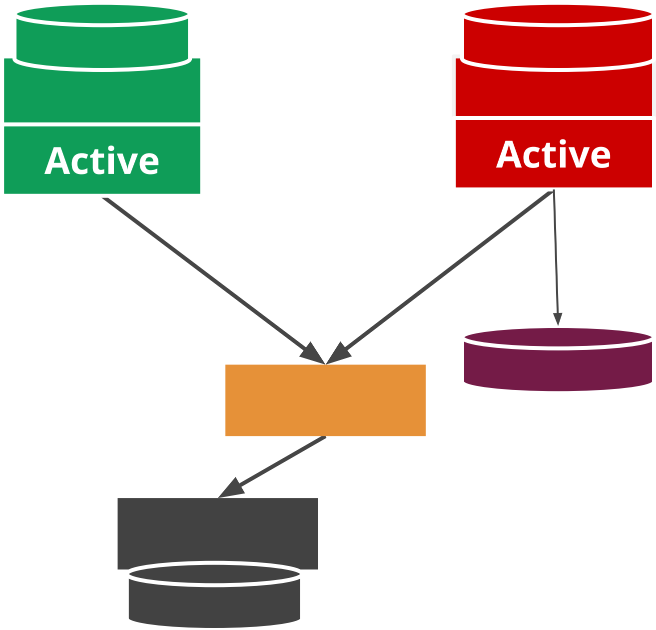


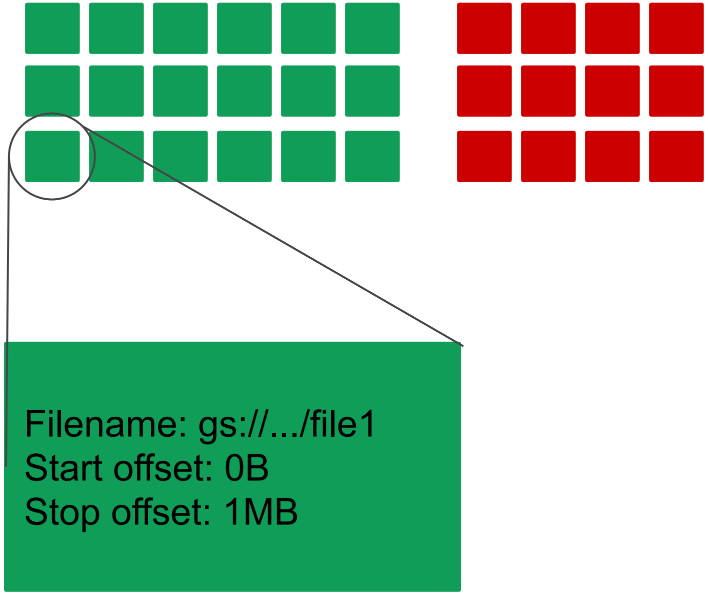
Execution: Batch stages run sequentially



- Stages wait for their complete input to be ready.

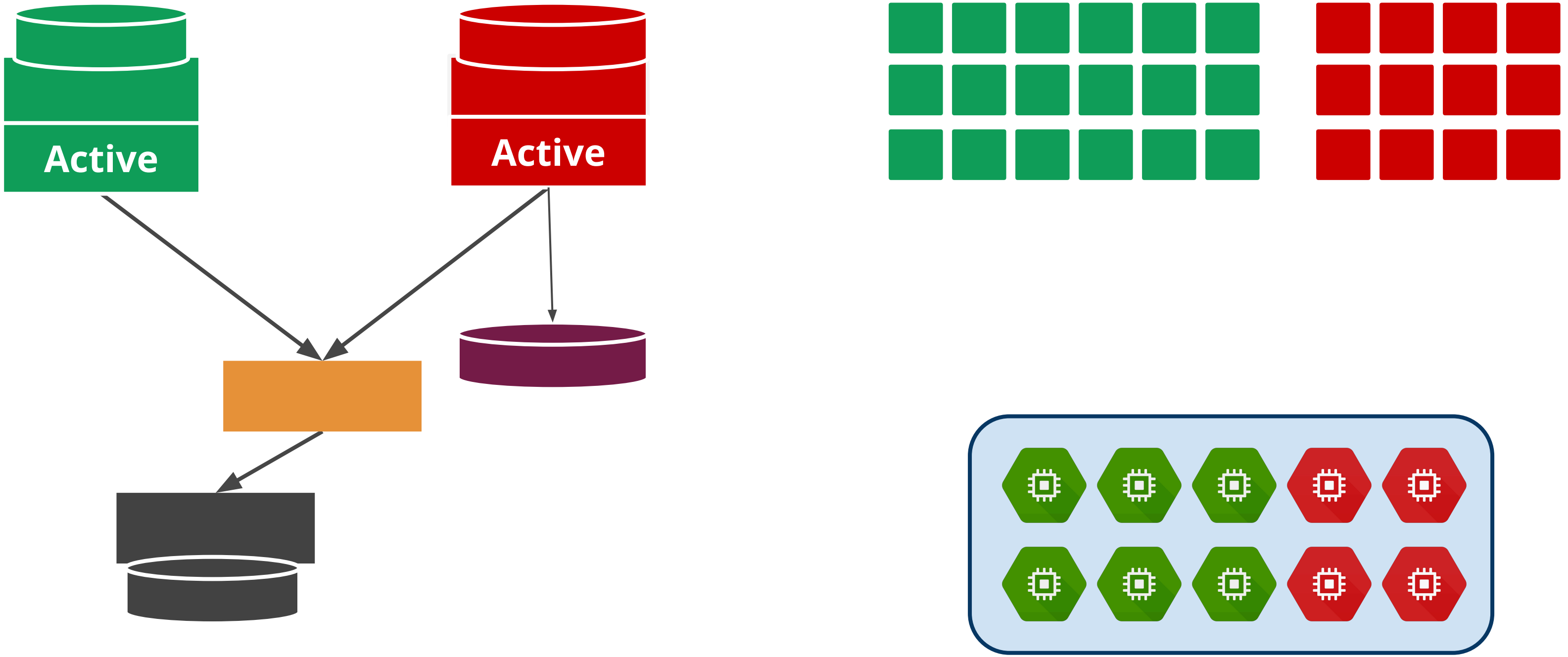
Execution: Generate Work Items (initial splitting)





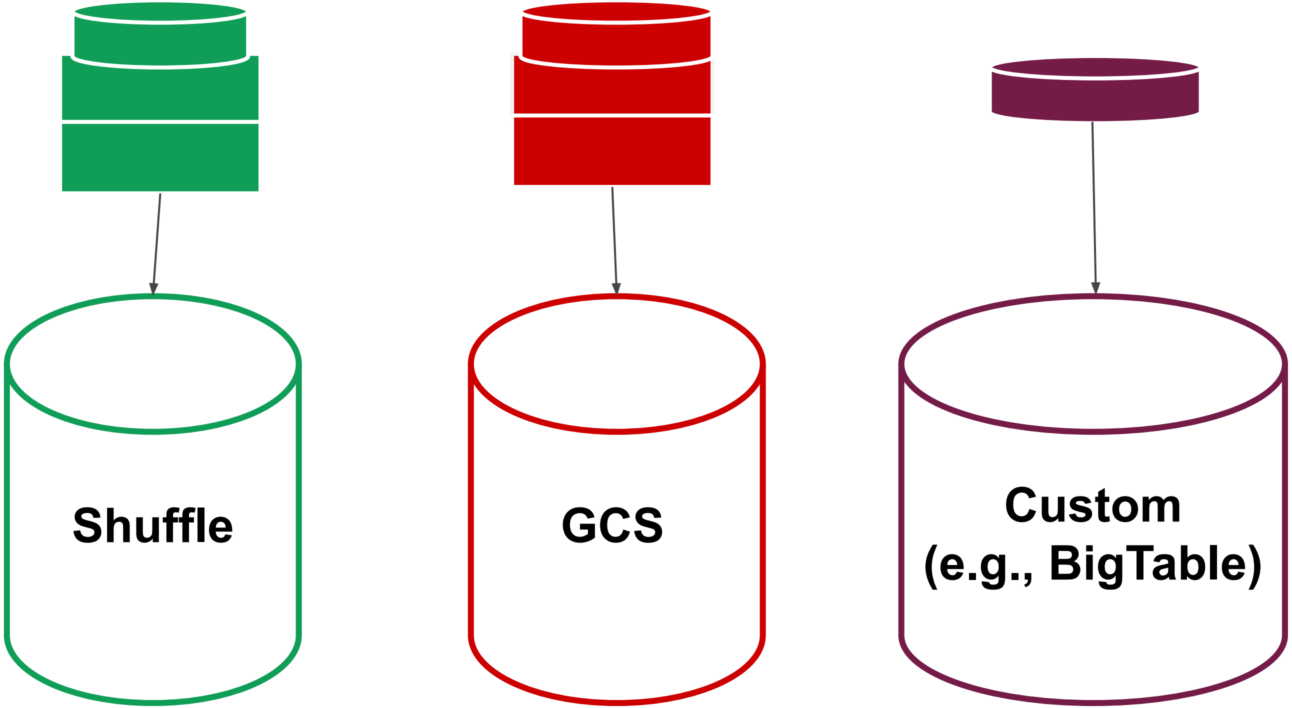
Unit of failure.

Workers Claim Work Items

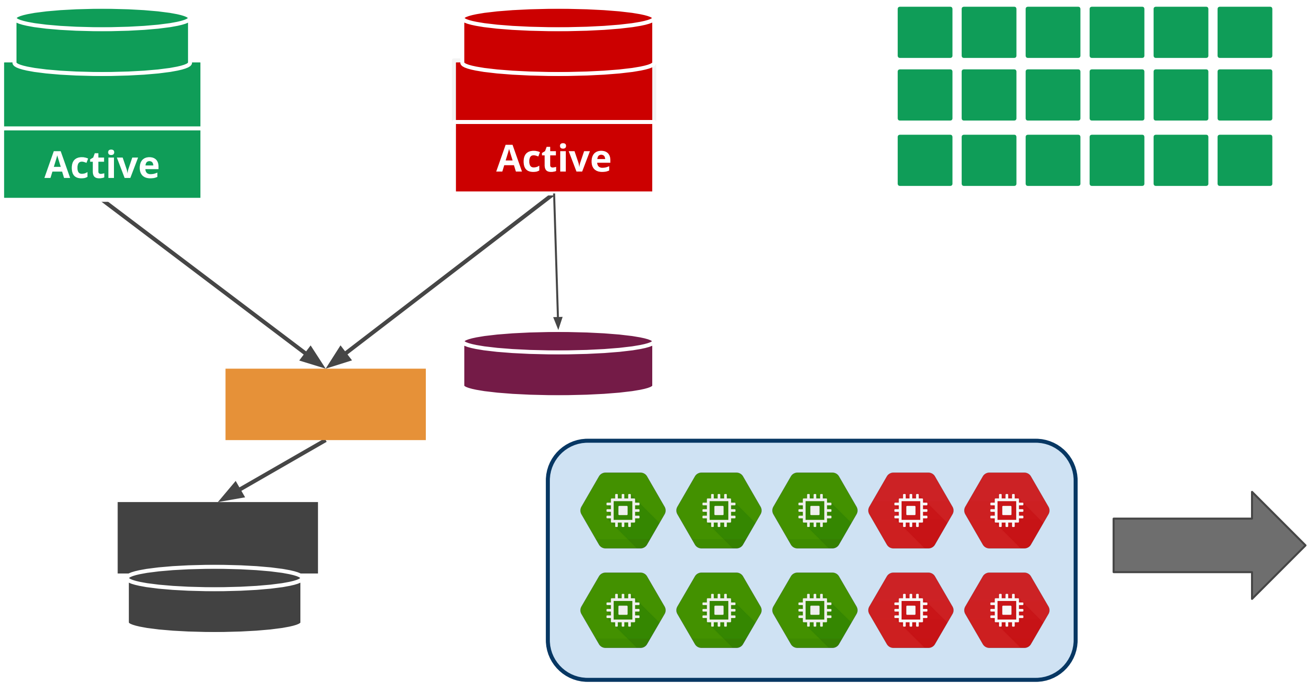


Output Written to Persistent Storage

Every completed work item is checkpointed

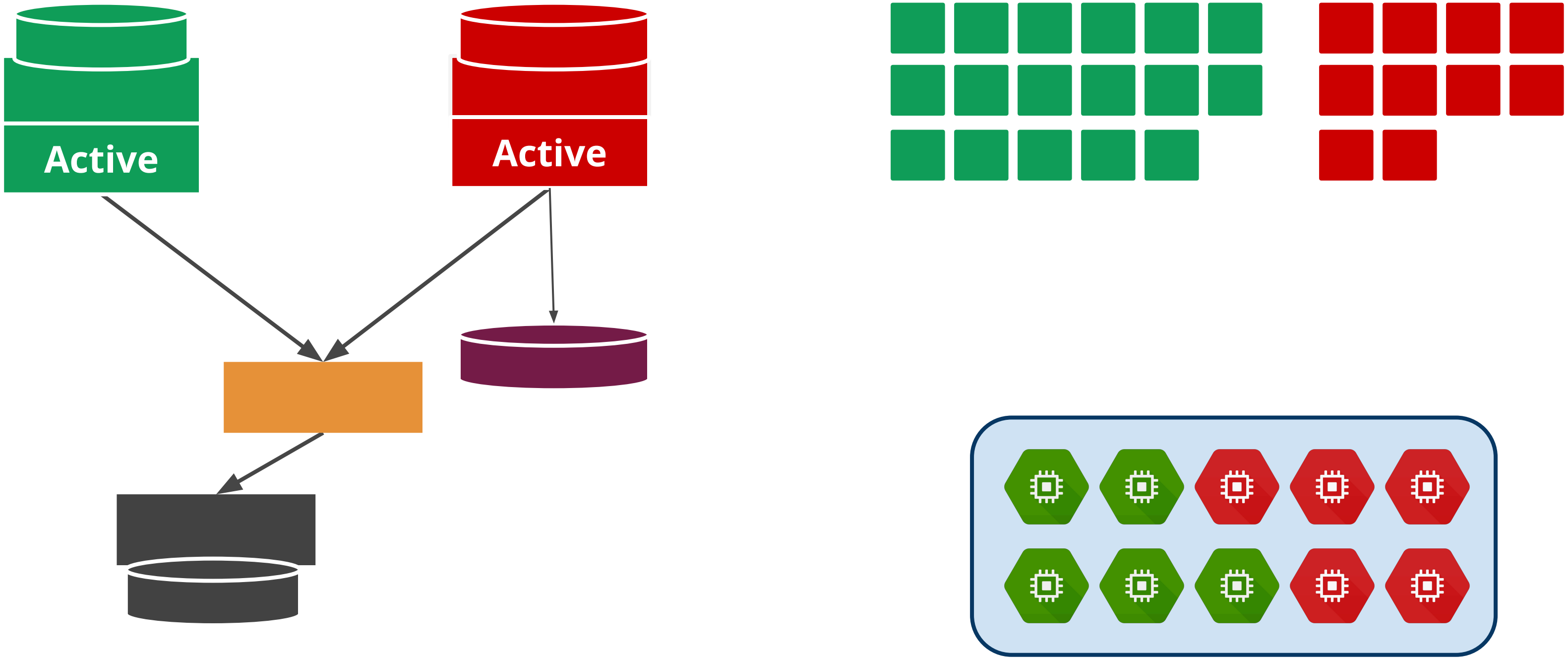


Workers Update Work Item Progress

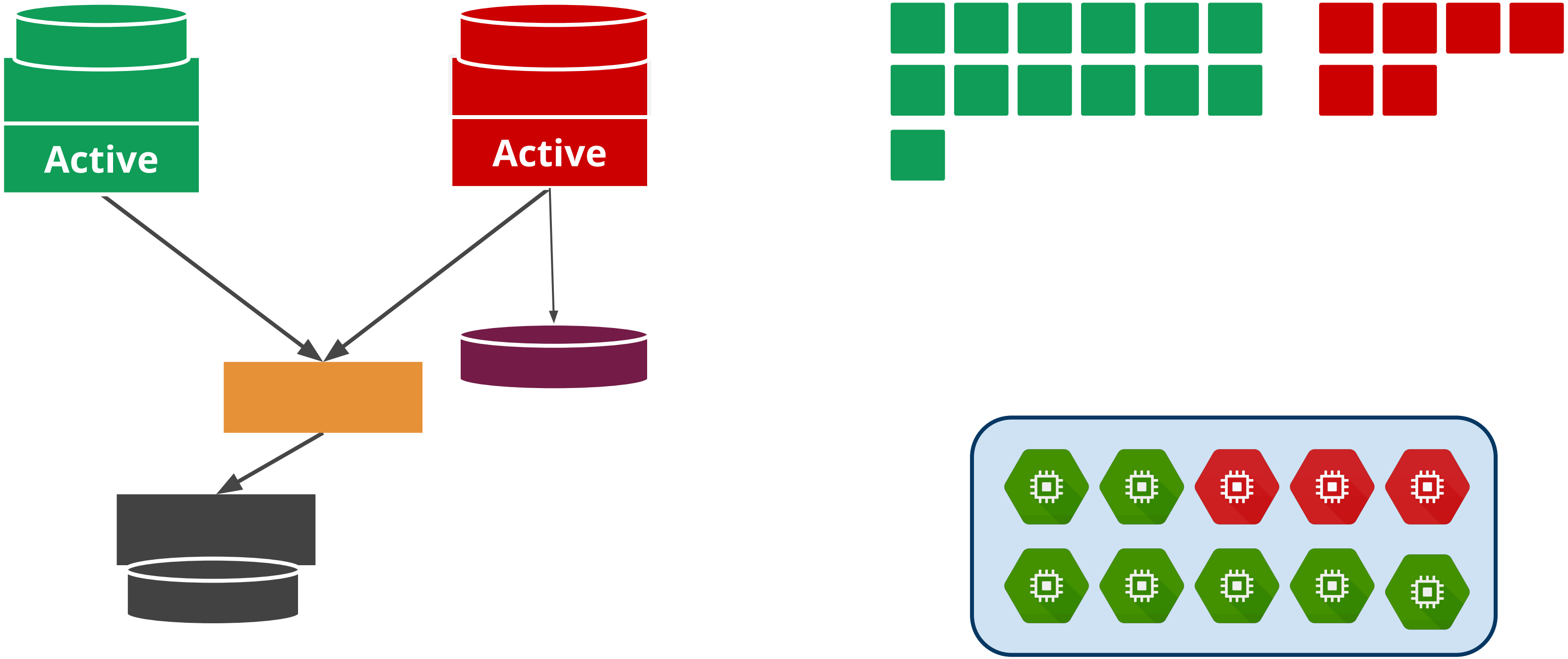


Update(work_item_2)
File read progress
Update counters
Extend lease
Complete? y/n

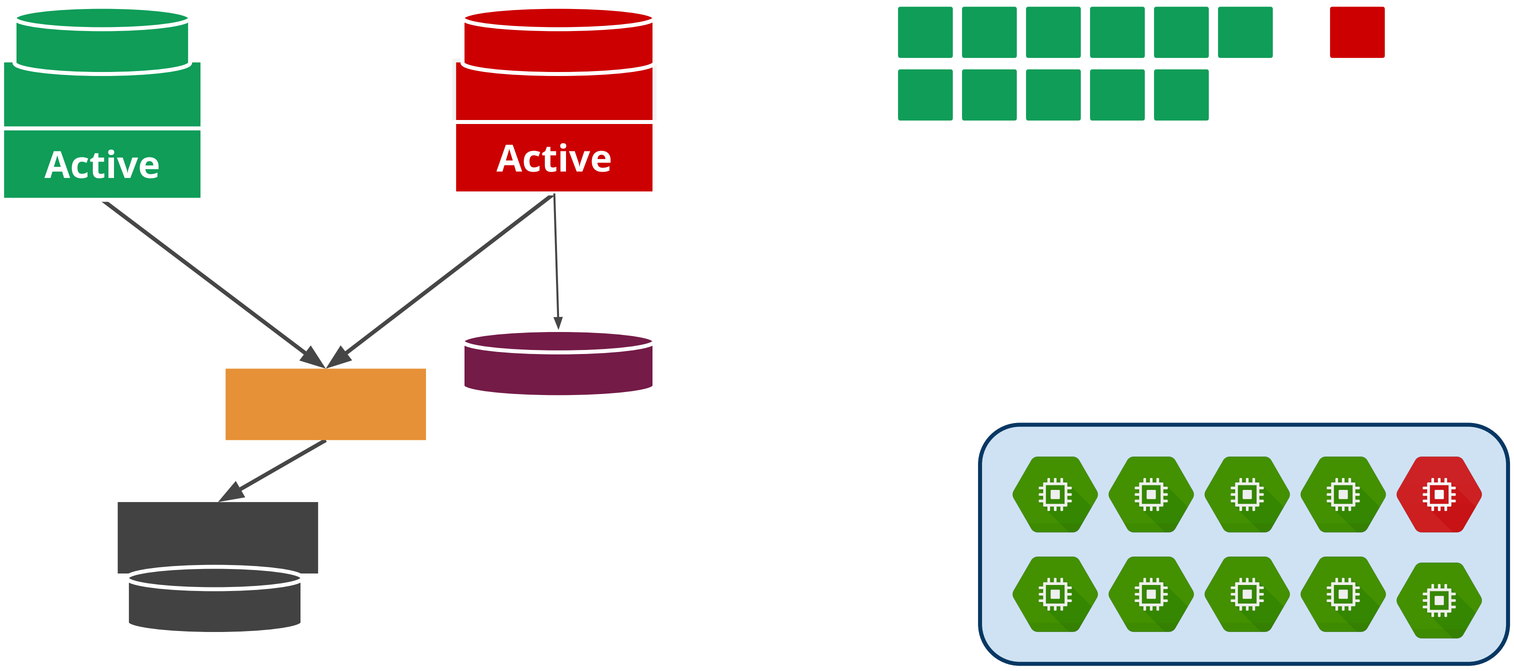
Two stages share the same pool



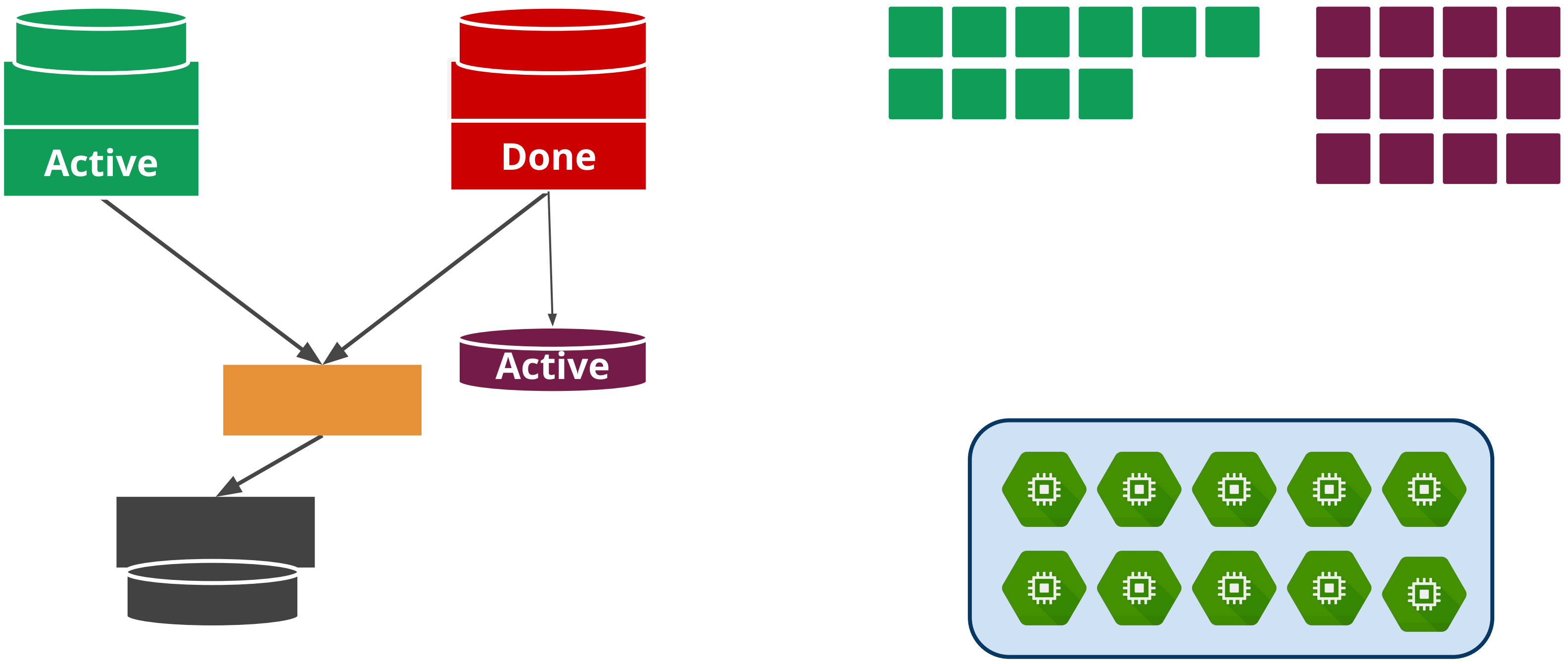
Making steady progress



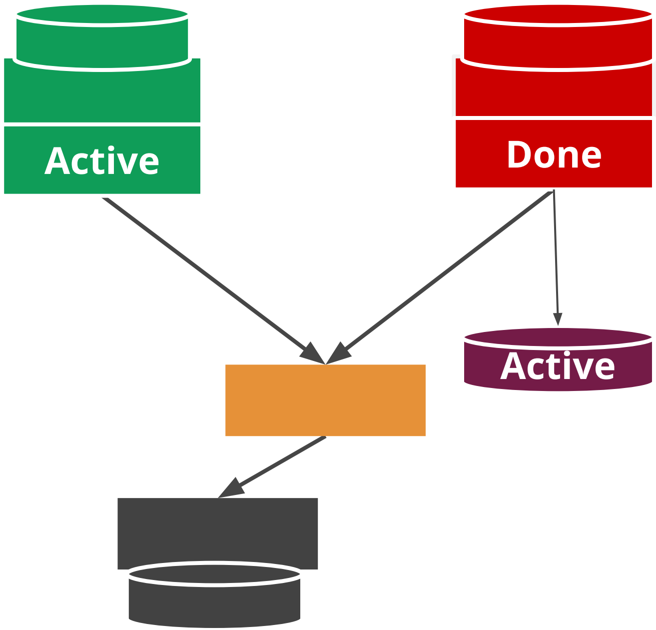
Red stage almost done

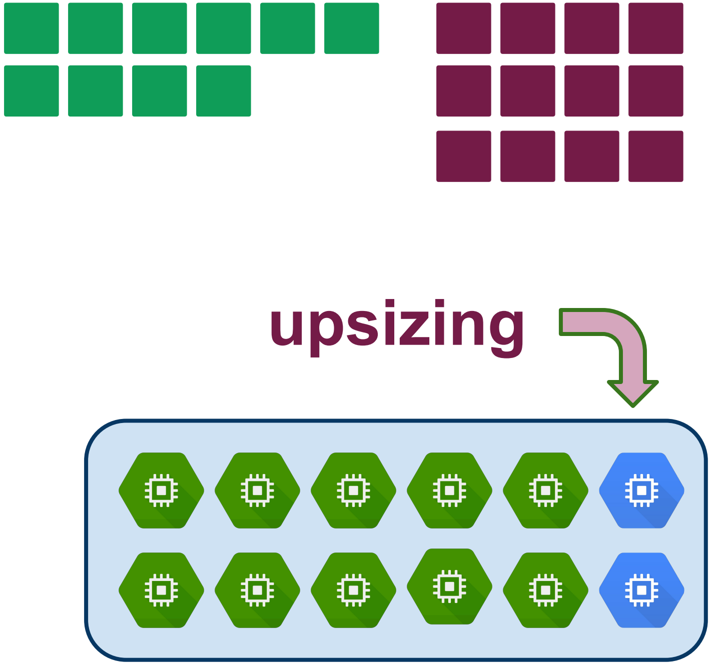


Red stage done, green running, purple unblocked

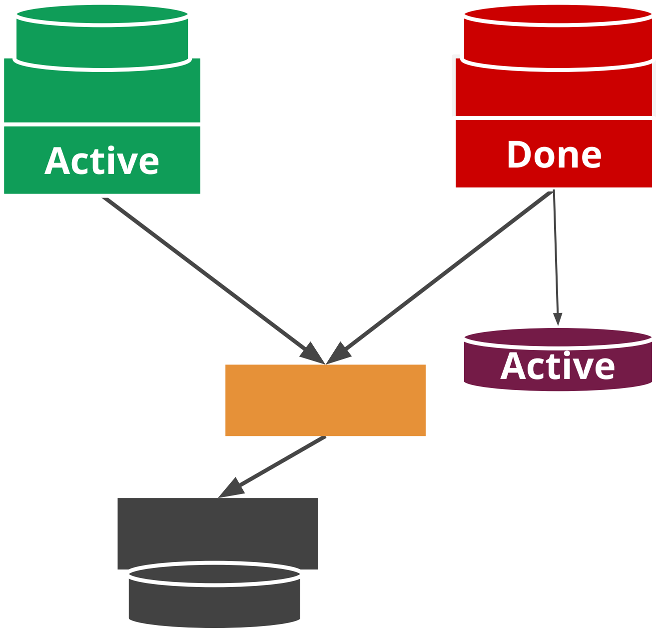


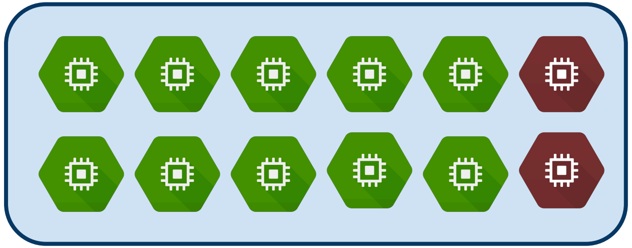
Autoscaling: Add more workers



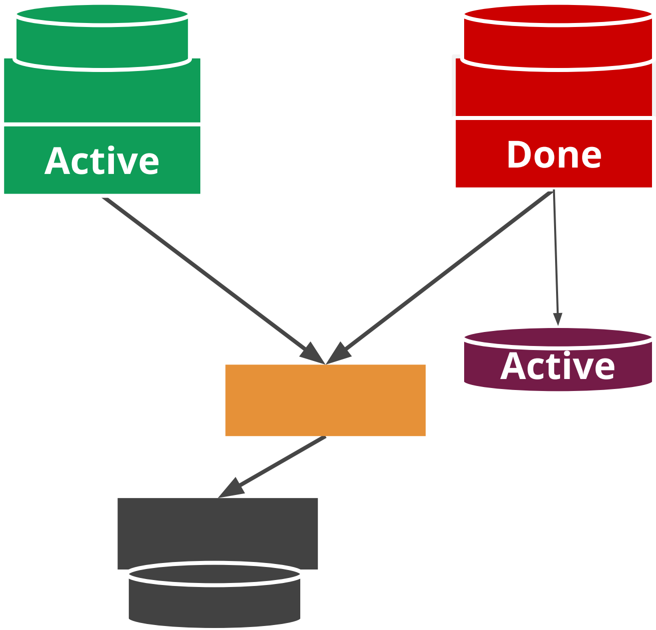


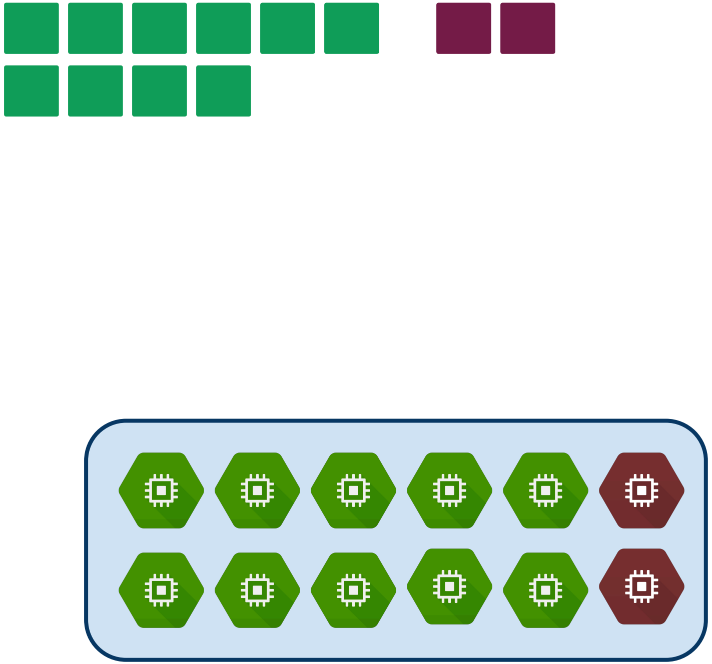
Autoscaling



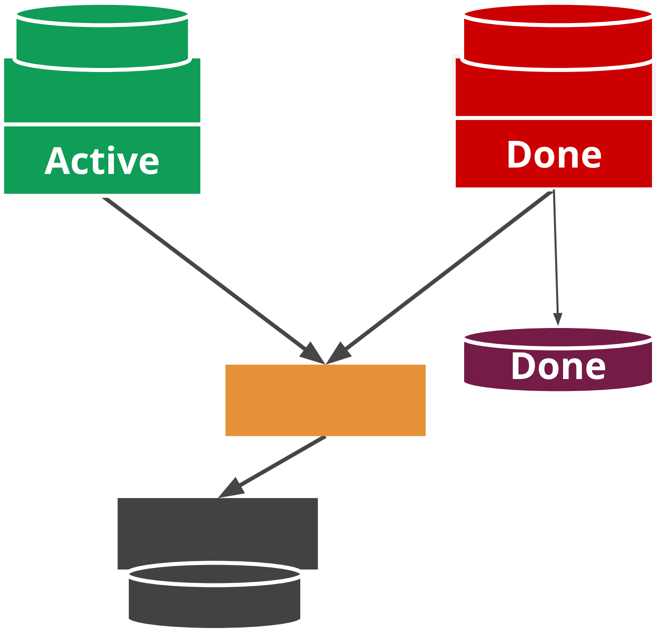


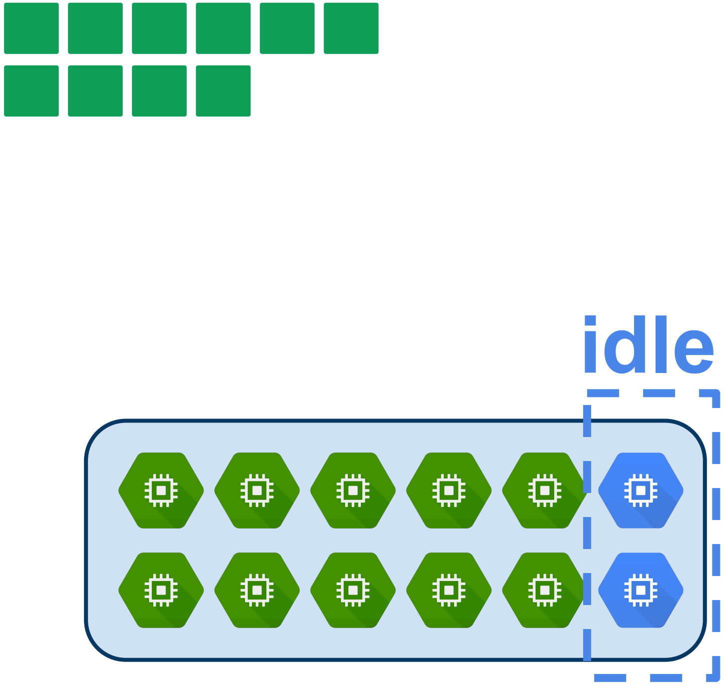
Autoscaling: Purple stage finishing quickly



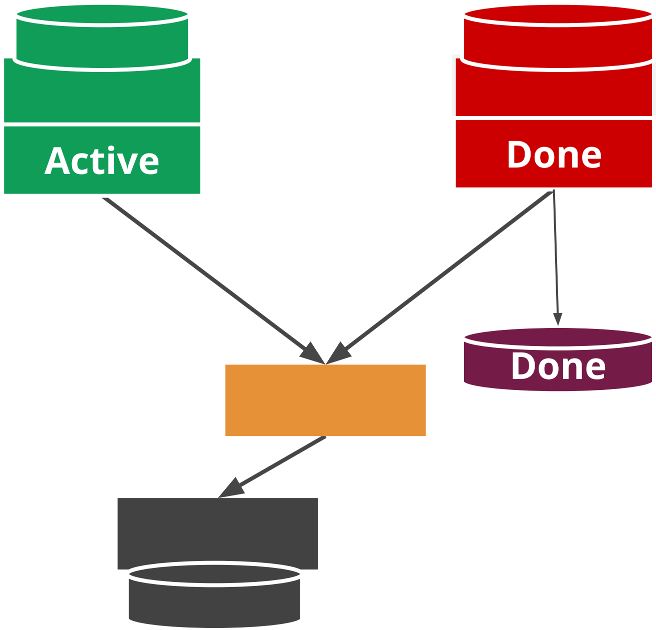


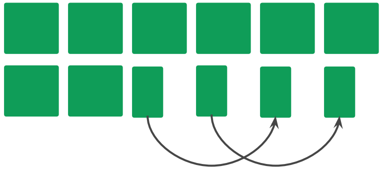
Autoscaling: Purple done, green still active

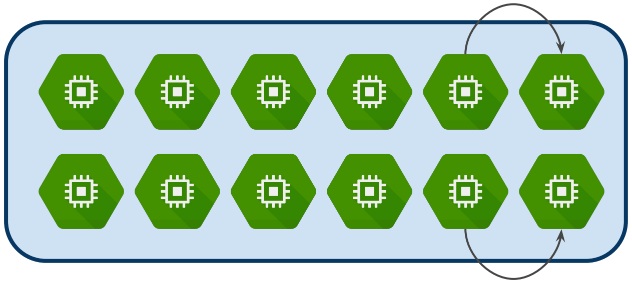




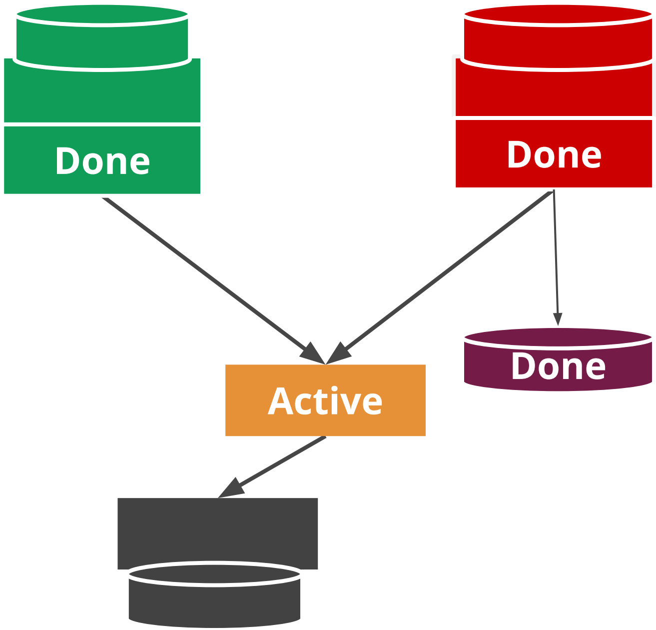
Dynamic Work Rebalancing: Splitting Work

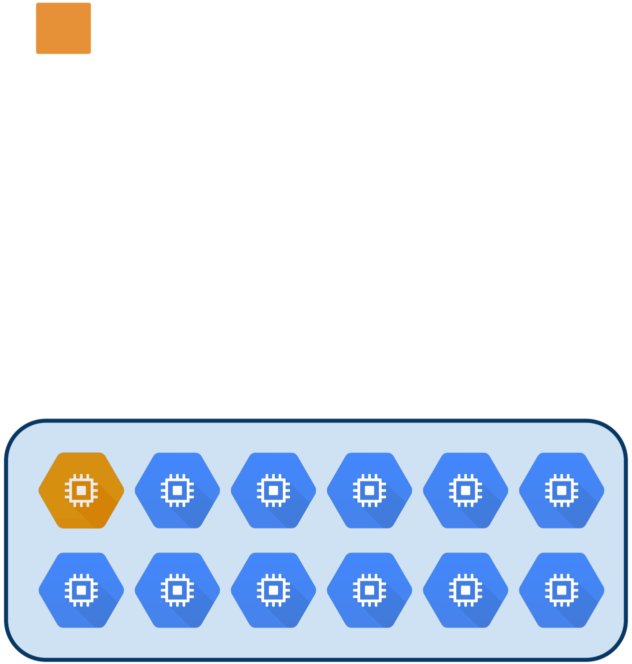




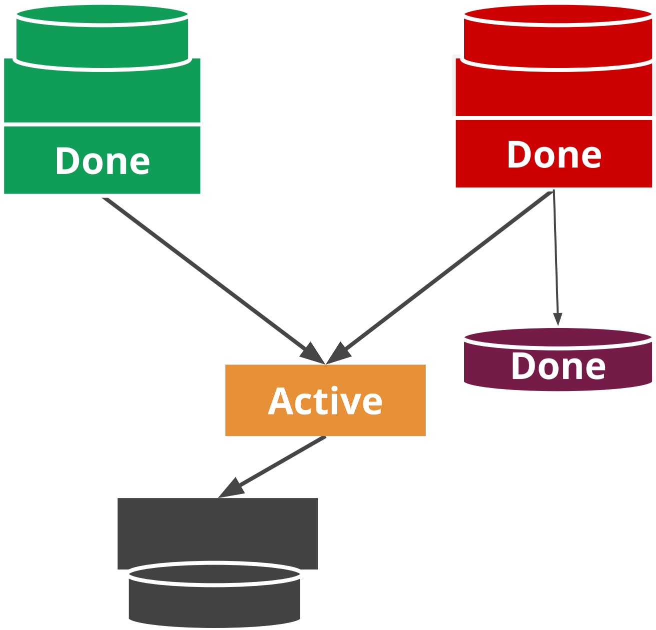


Next Stage: Limited Parallelism

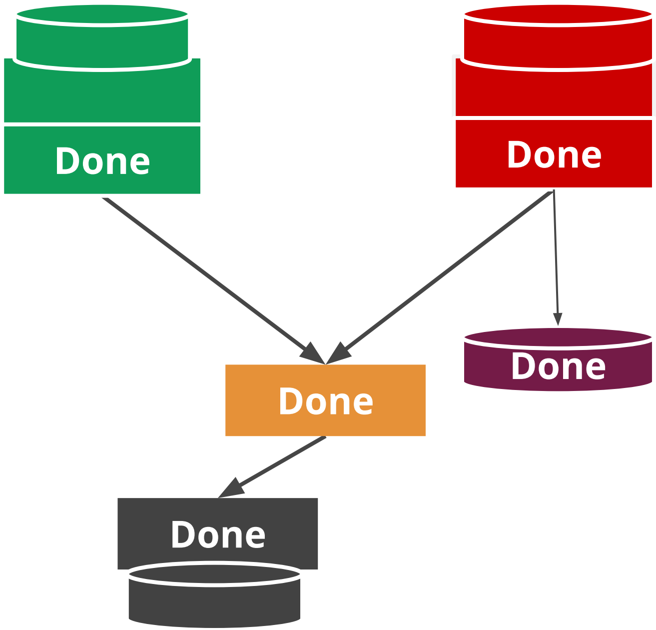




Downscaling



And, finally



Pipeline shuts down

All resources released

All output and counters are available

Pipeline is marked as done in the UI and API

Summary

1. Beam vision
2. Basics of the SDK
3. Basics of Runners
4. Portability framework & its advantages
5. Autoscaling (Dataflow specific)

Thank you!

Questions?