bigdata

Challenges faced and lessons learnt while building a multi-cloud Data platform

Antonio Murgia – Big Data conference EU 2023

Who am I

agilelab

👨‍💻 Data Architect at AgileLab.
I spent the last 7 years first implementing, then leading teams and ultimately designing from the ground-up scalable streaming and batch data solutions.

I'm an open source contributor 👨‍🎓
(Apache Spark ✨ and Project Nessie 🦕)

In my free time I love to hike 🥾 and climb 🧗 mountains 🏔️ and I'm up for playing any kind of sport 🏐

Great place to work 😎

agilelab

witboost

Born as a Data management consultancy company in 2013, based in Turin (Italy)
Focused on scalable technologies and data practices - which is a way to scale too

We are now 150 engineers all over europe!

In recent years we started to develop witboost, a product that helps big organizations build their internal data development platform based on their own needs, processes, practices and policies

Great Place To Work.®

⚠️ DISCLAIMER ⚠️

This is not a guide on «how to» implement a multi-cloud data platform.

We were NOT implementing a «greenfield» data platform.

This is the story of how we managed to transform an AWS centric platform to a multi-cloud one.

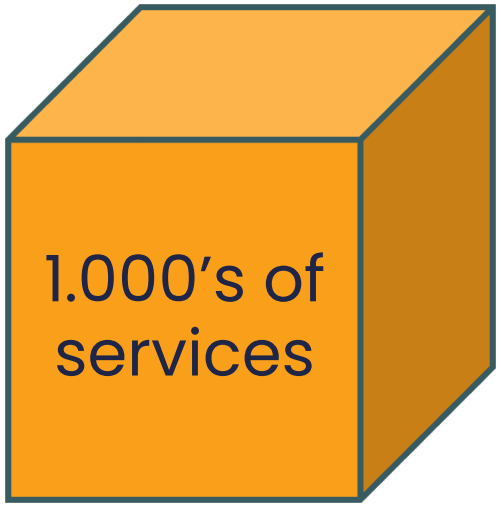
Once upon a time

We started building a service + data platform on AWS in 2019
The platform follows an “holistic” approach:

domain teams 🔨 build, 📦 deploy and 👷 operate
in the same platform both operational and data services/products

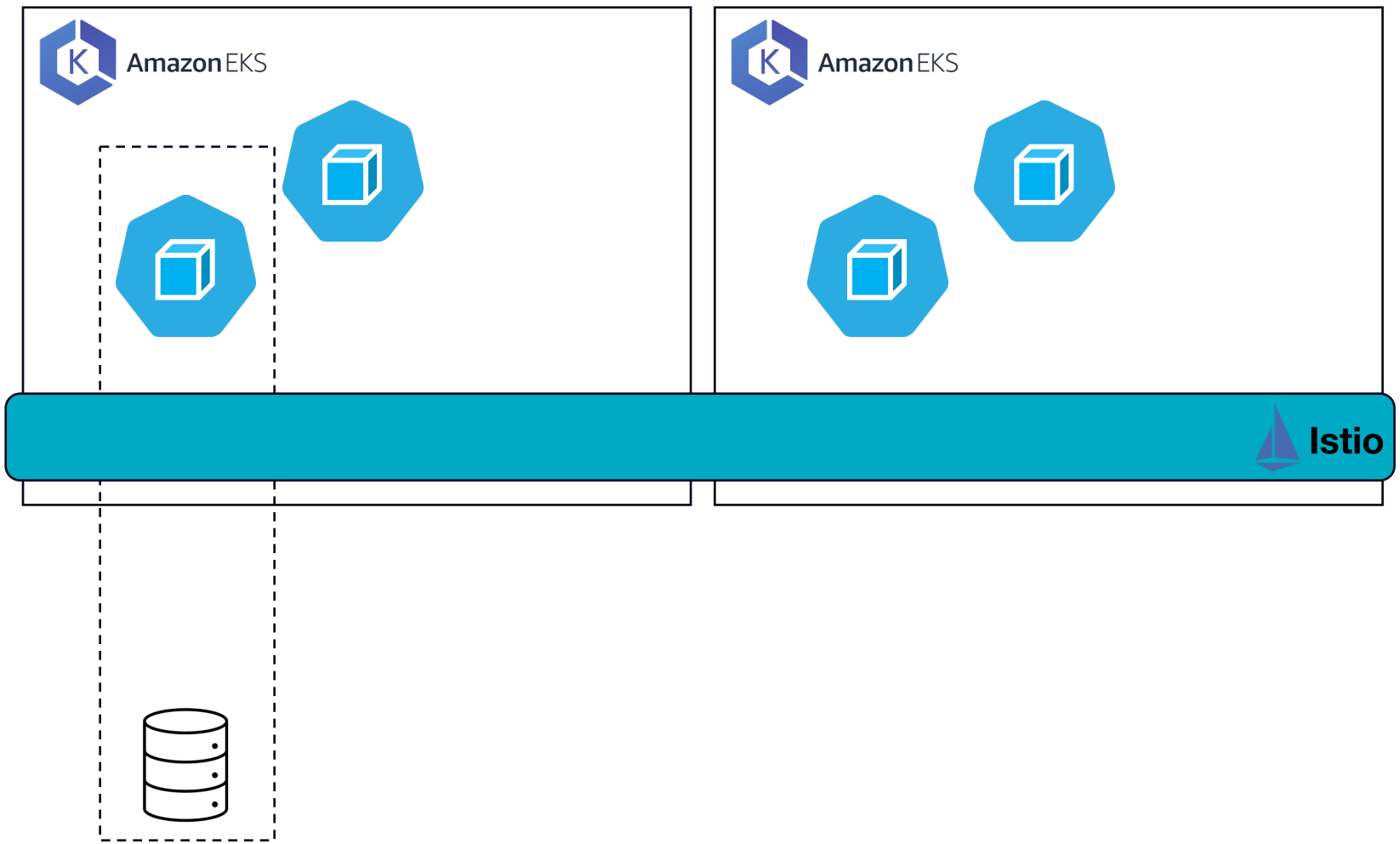
In retrospective we were building a very prehistoric kind of Data Mesh +1 year before Zhamak blogged about it on Martin Fowler website

Context



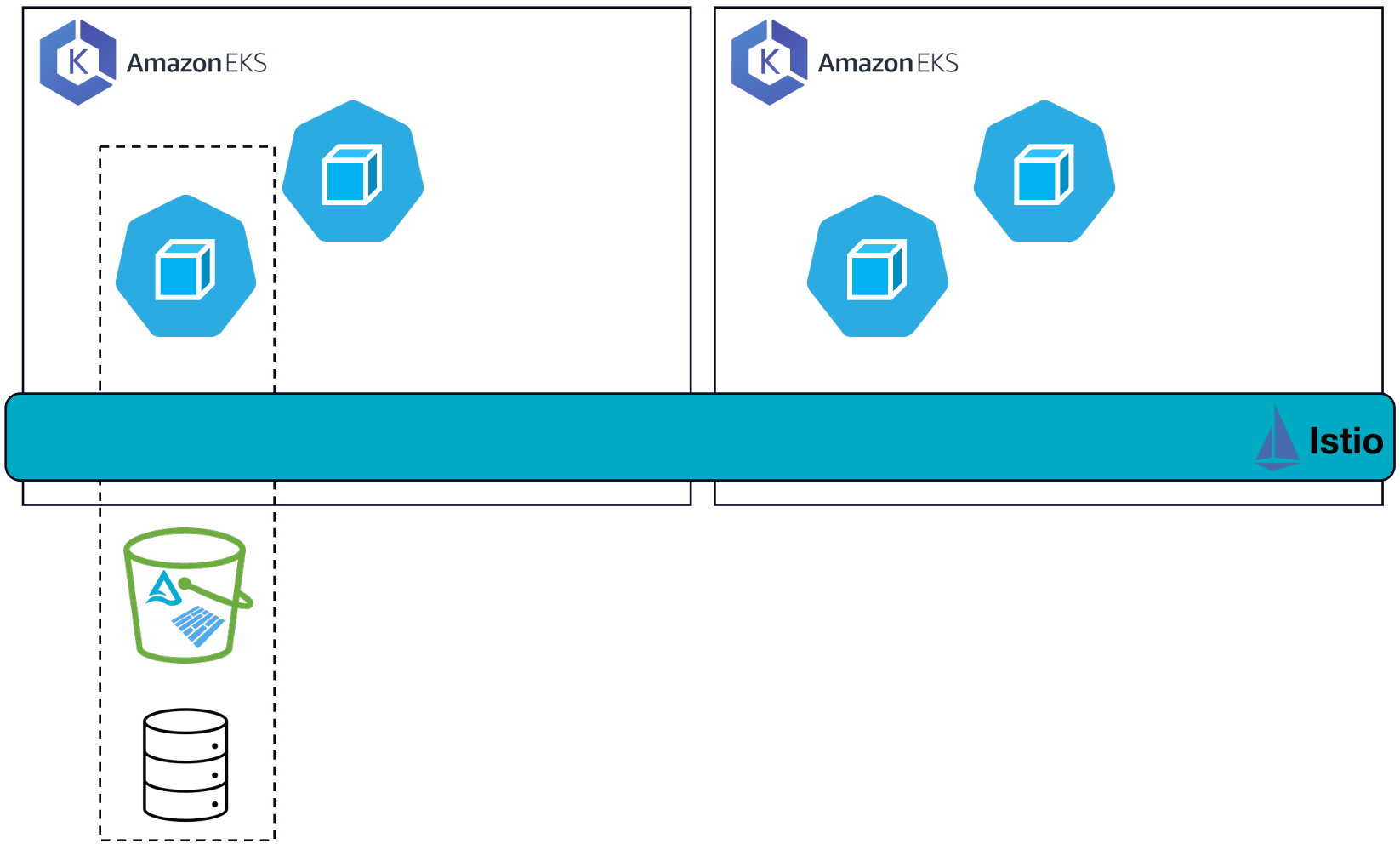
Technology Stack

Technology Stack

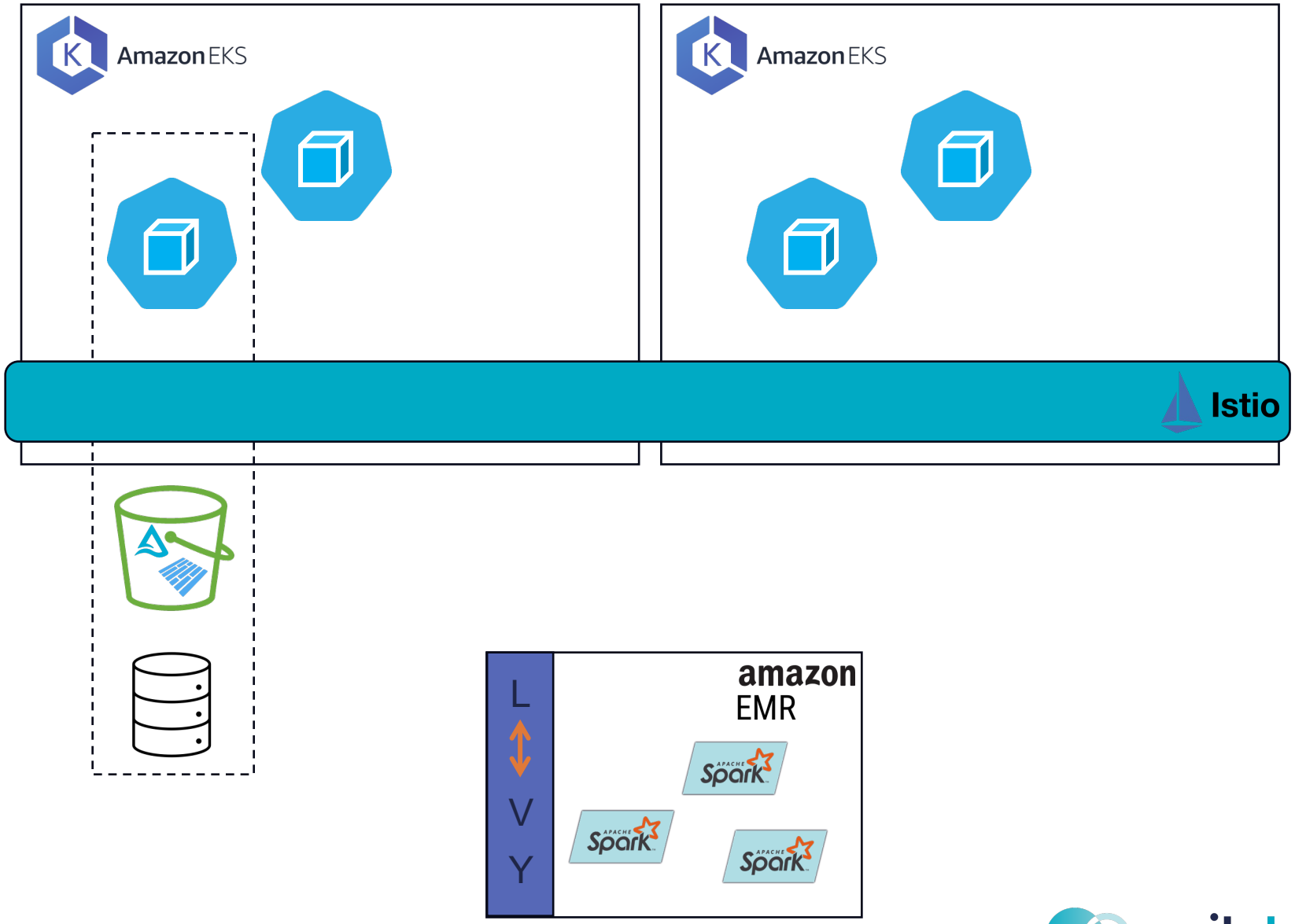


agilelab

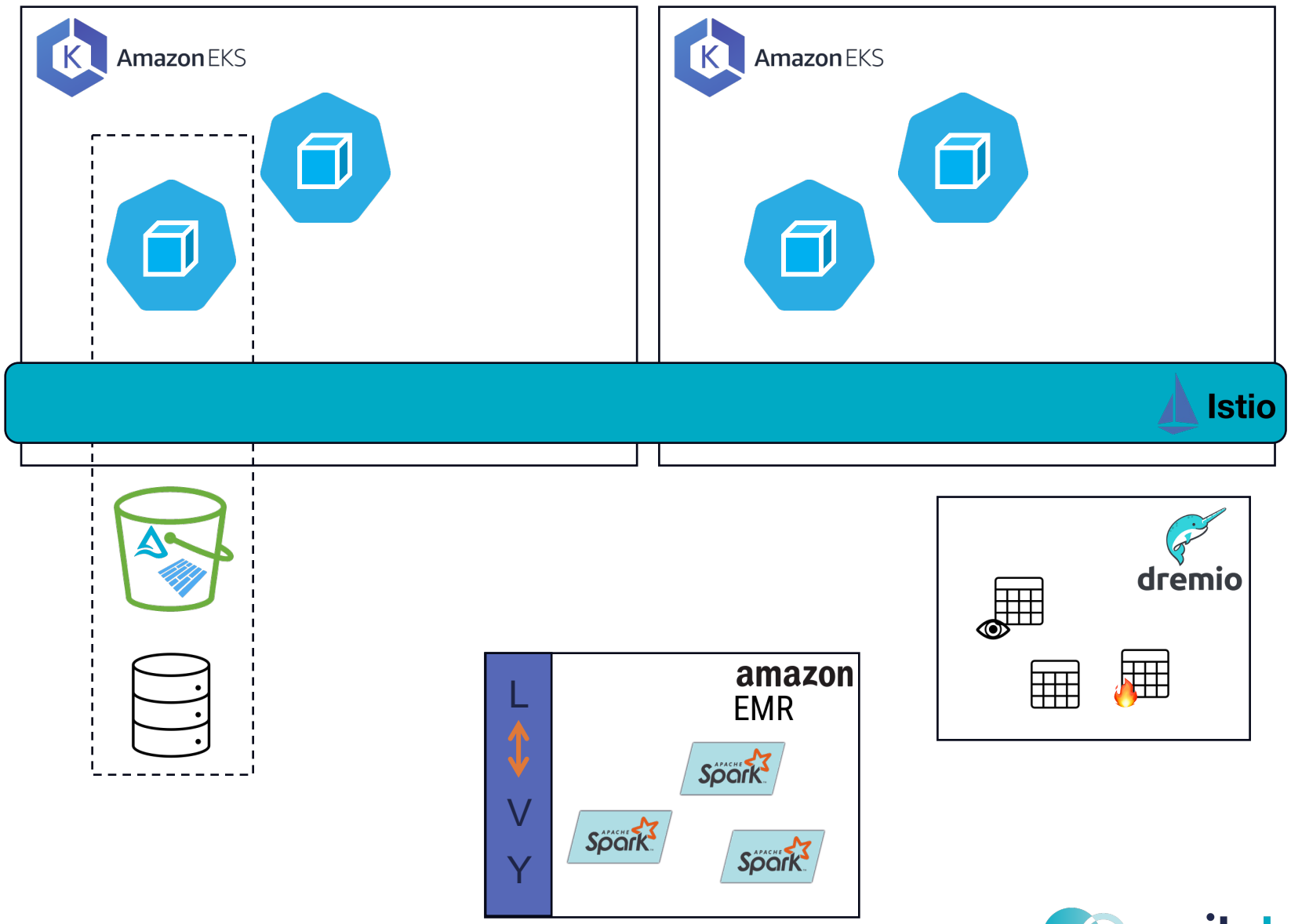
Technology Stack



Technology Stack

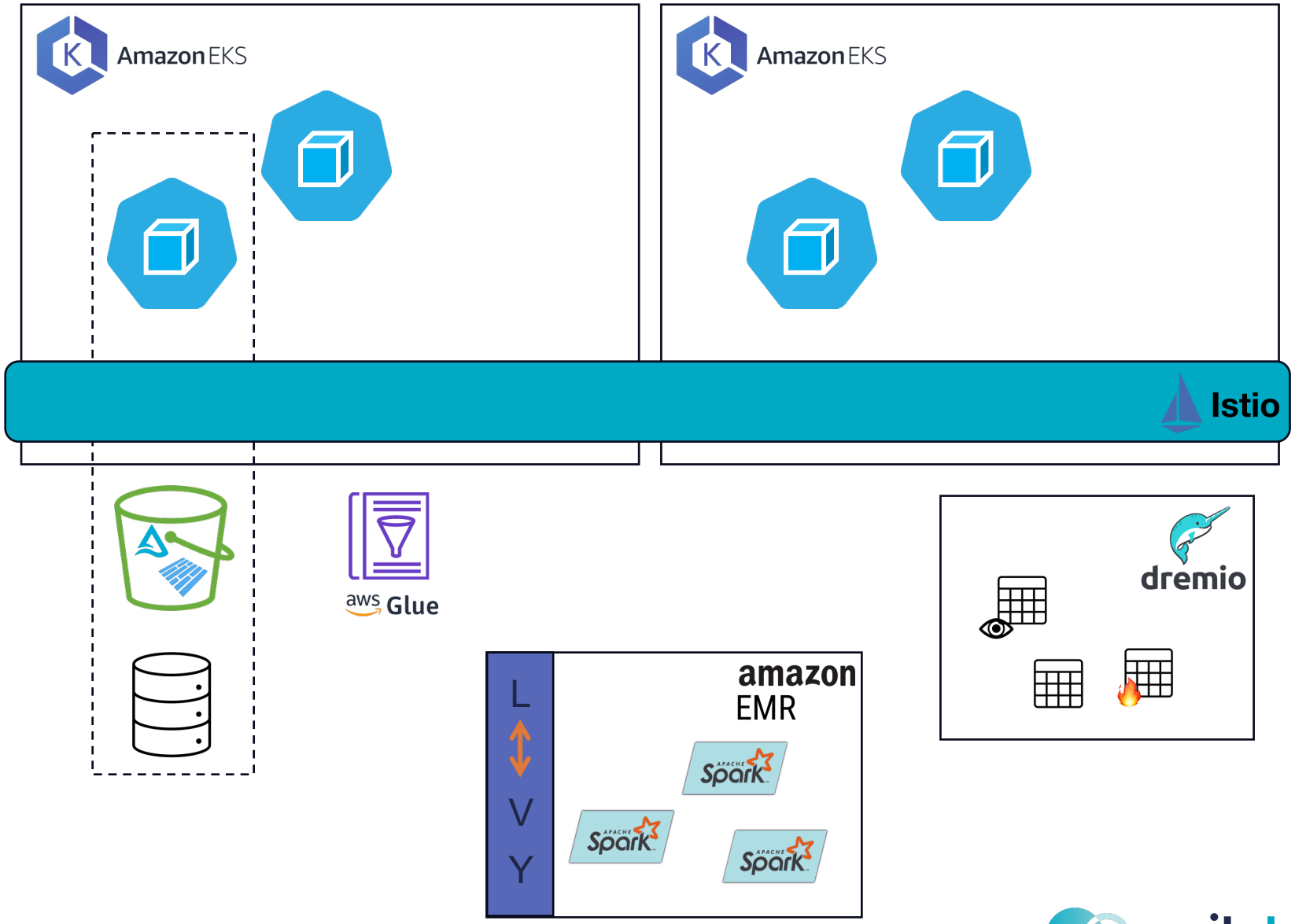


Technology Stack



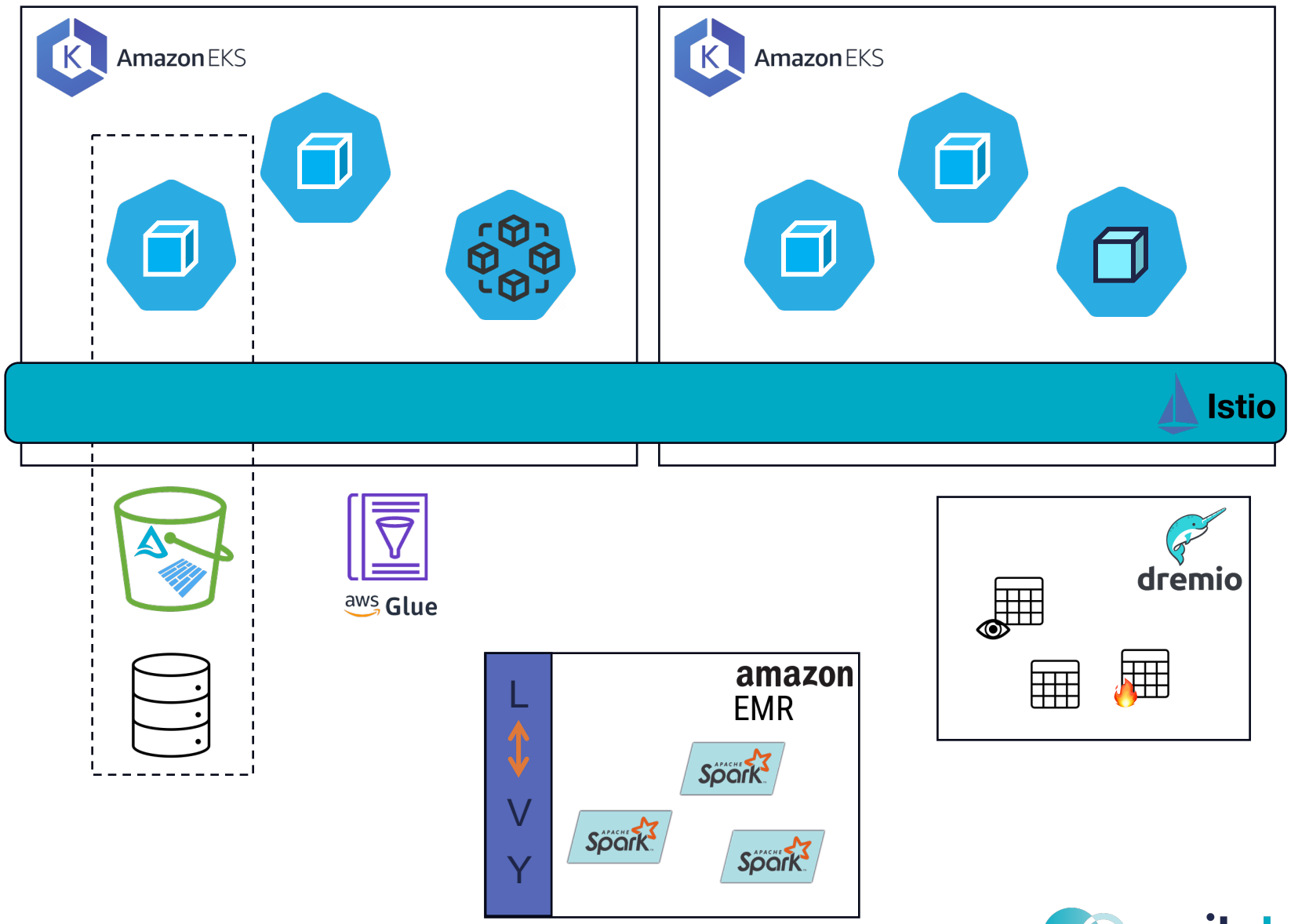
agilelab

Technology Stack



agilelab

Technology Stack



agilelab

Organization

Each domain team has an associated AWS IAM Role for each environment (dev/test/prod)

Each service has its own s3 bucket and operational DB

A central catalog is the entry-point for governance

Datasets are declared as Glue tables

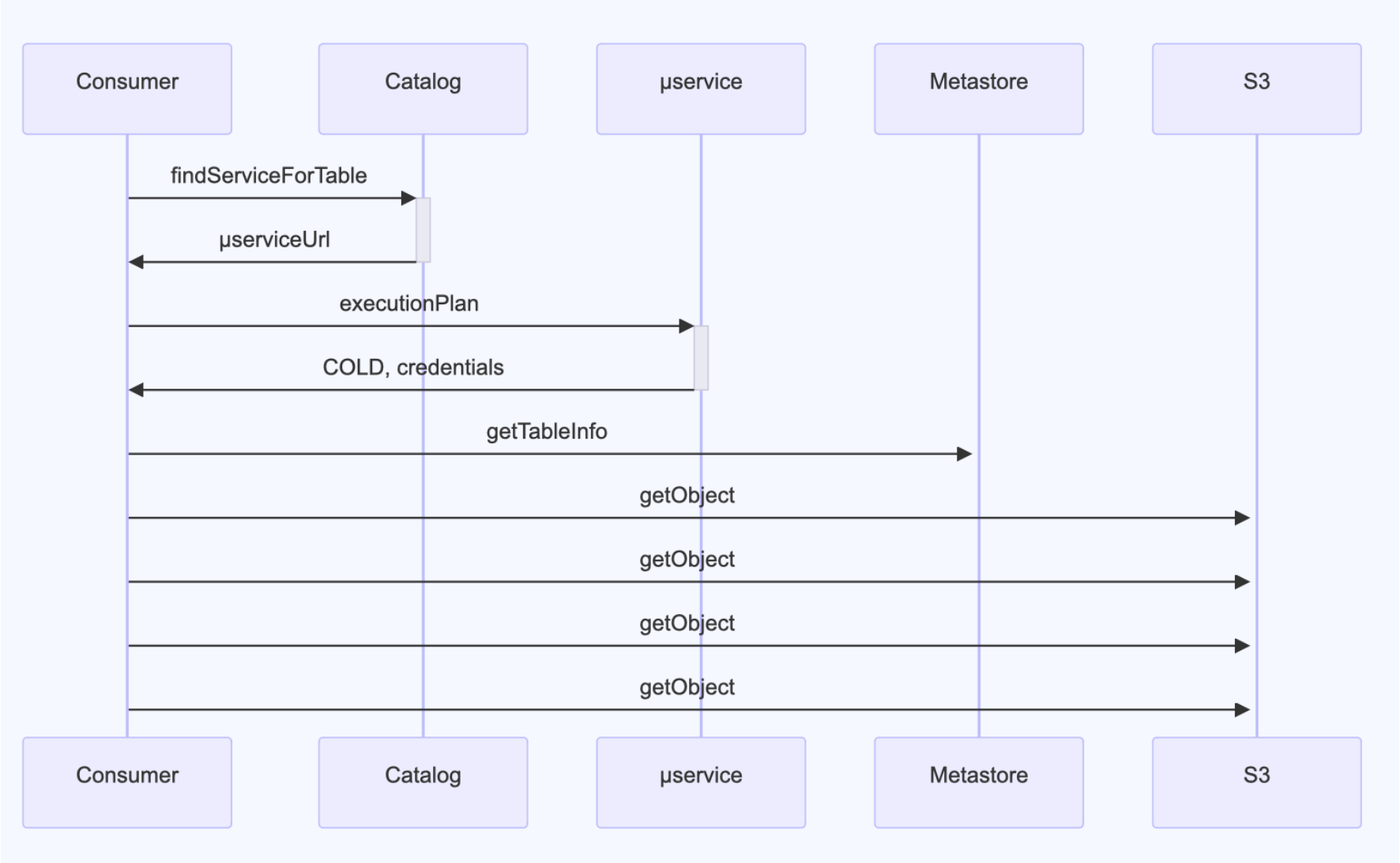
Data for analytical purposes can be exposed in one of 3 ways:

- ❄️ Parquet or Delta Lake tables on s3
- 🔥 json/http with a custom pre-defined protocol

Everything is provisioned through CloudFormation

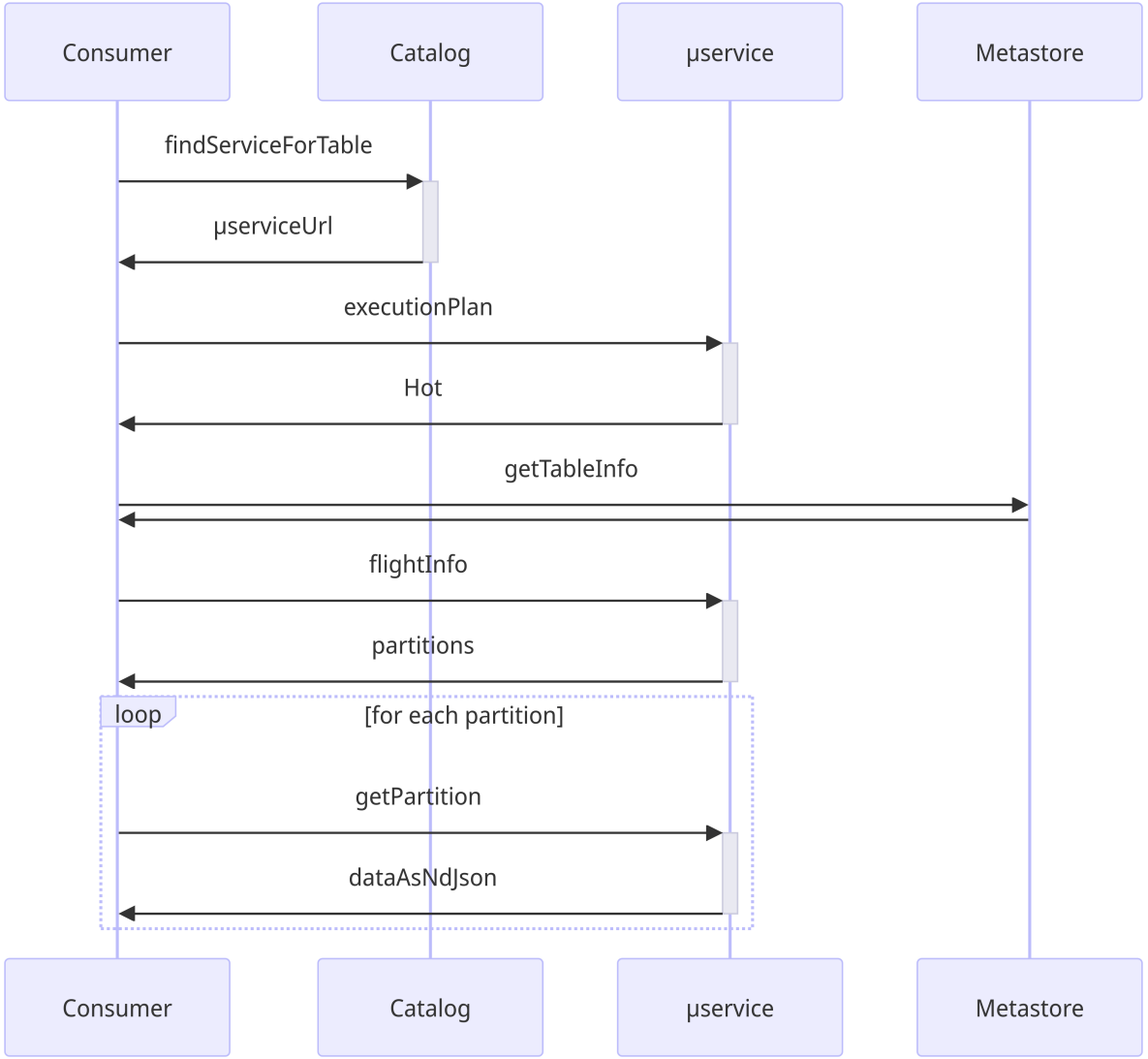
Everything is API-first

Cold Data *(Parquet/Delta Lake on s3)*



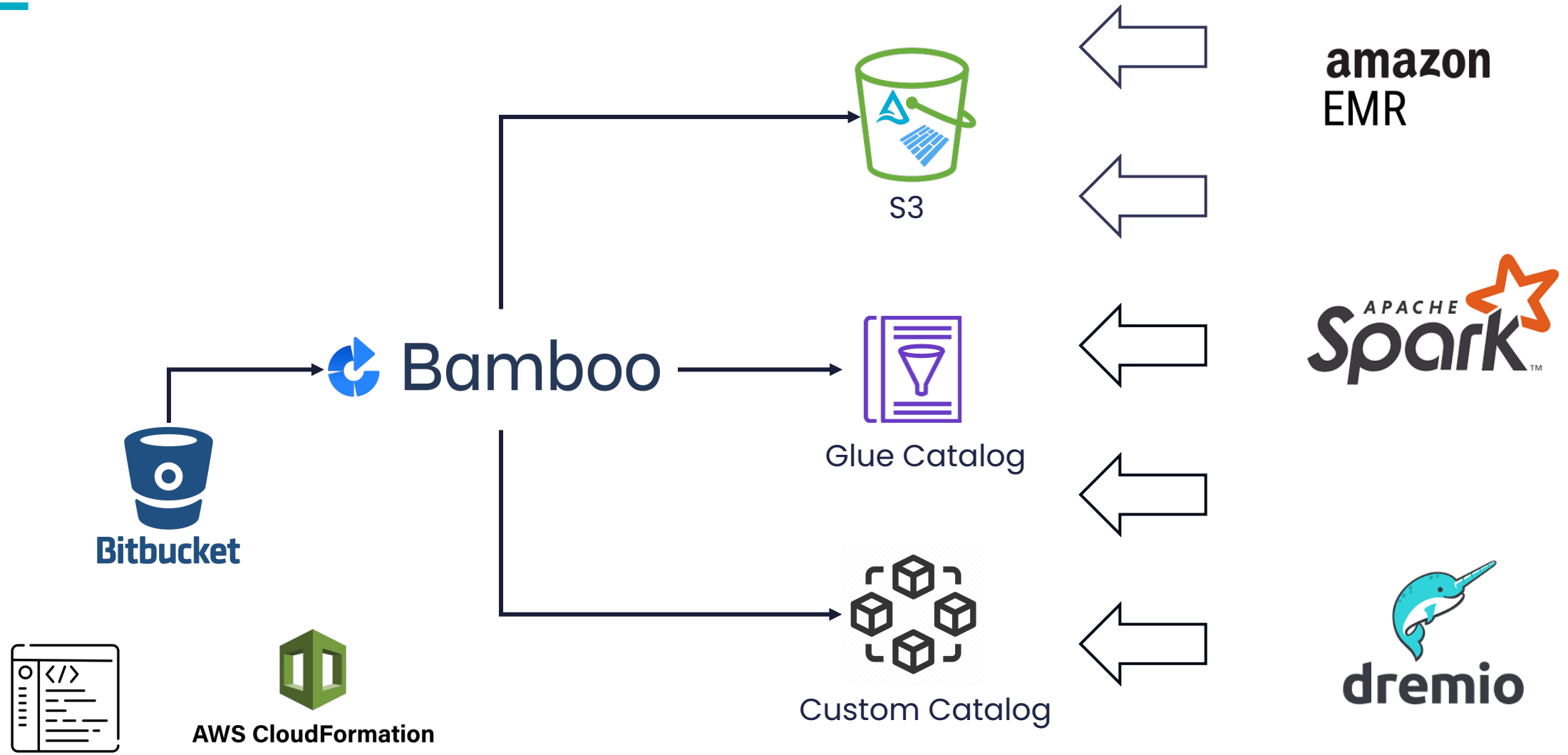
agilelab

Hot 🔥 Data *(http/json)*

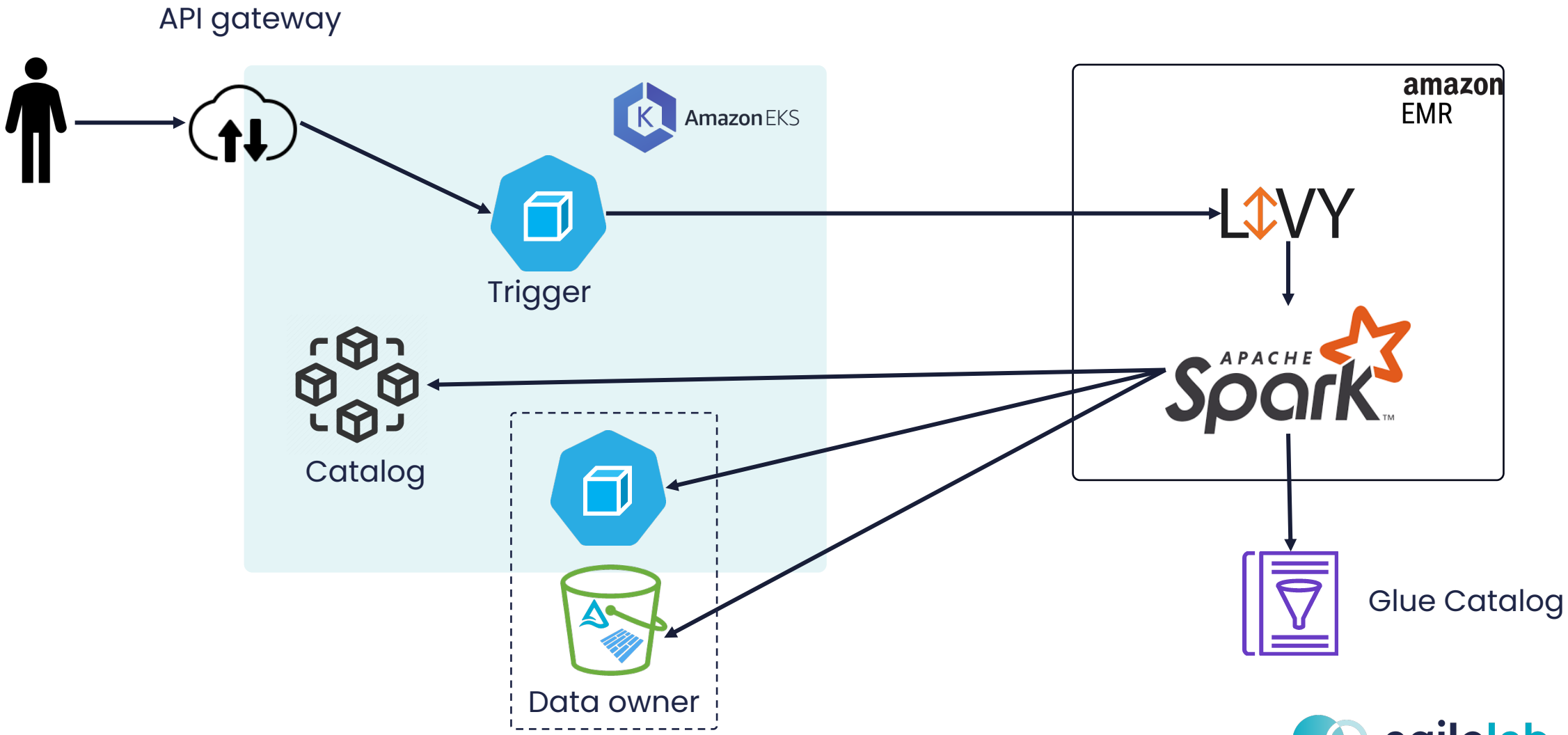


agilelab

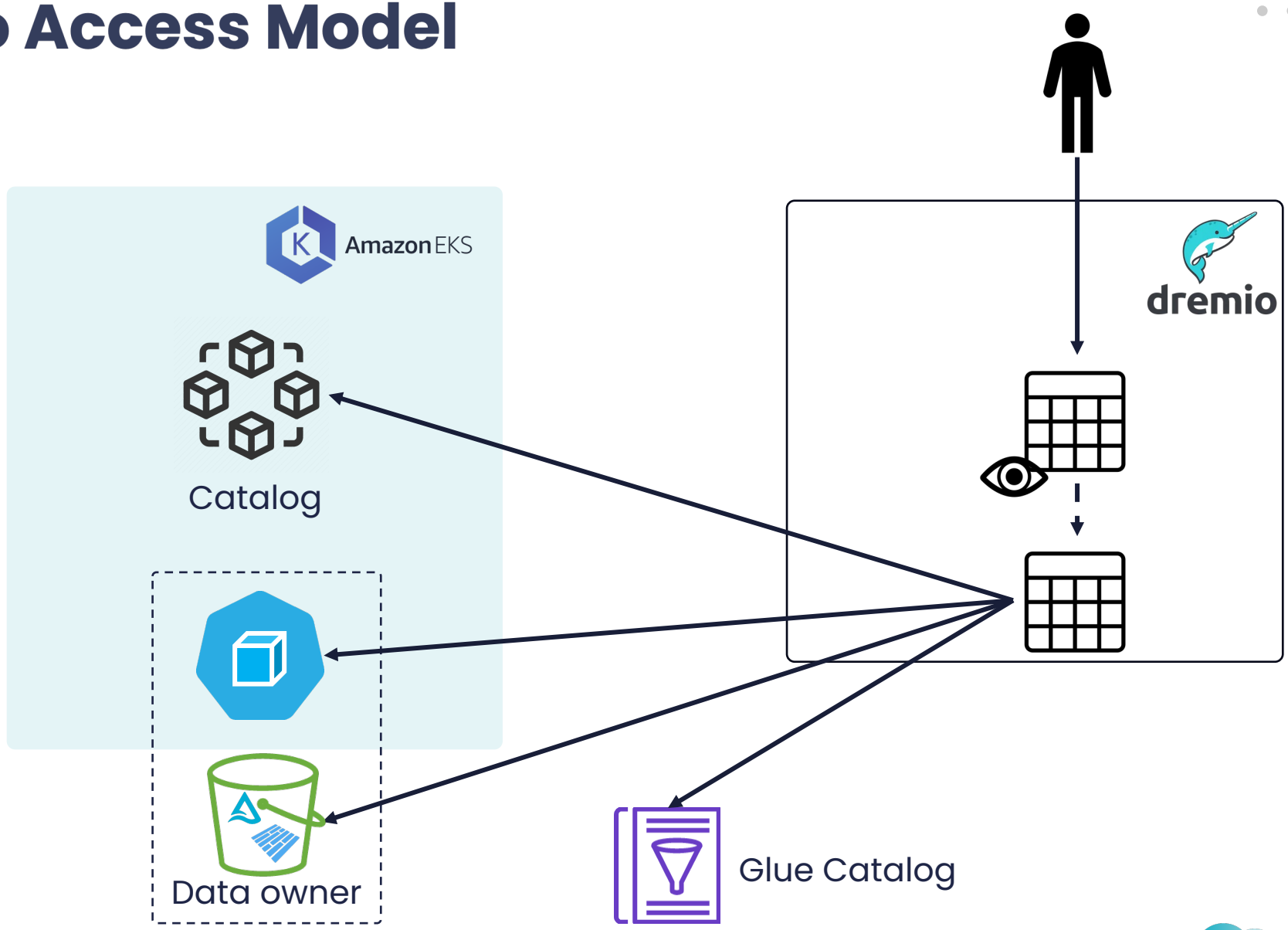
Table creation



Job submission



Dremio Access Model



Let’s dive into Multi-cloud

What's multicloud?

Definition

multiple **cloud** computing services in a single architecture

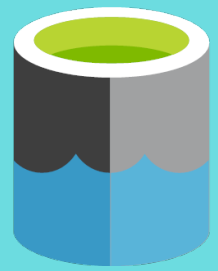
Key Points

Goals

Transparent allocation on aws and Azure of:

Compute

Storage



Already sorted out

«If I have seen further it is by standing on the shoulders of giants»

Isaac Newton

Http communication across different Kubernetes clusters on different cloud providers (backbone on on-premise data center)

Challenges

No open source Spark Glue connector

EMR is not available out of aws

amazon
EMR

Provide a non-breaking upgrade

Metastore

Challenges:

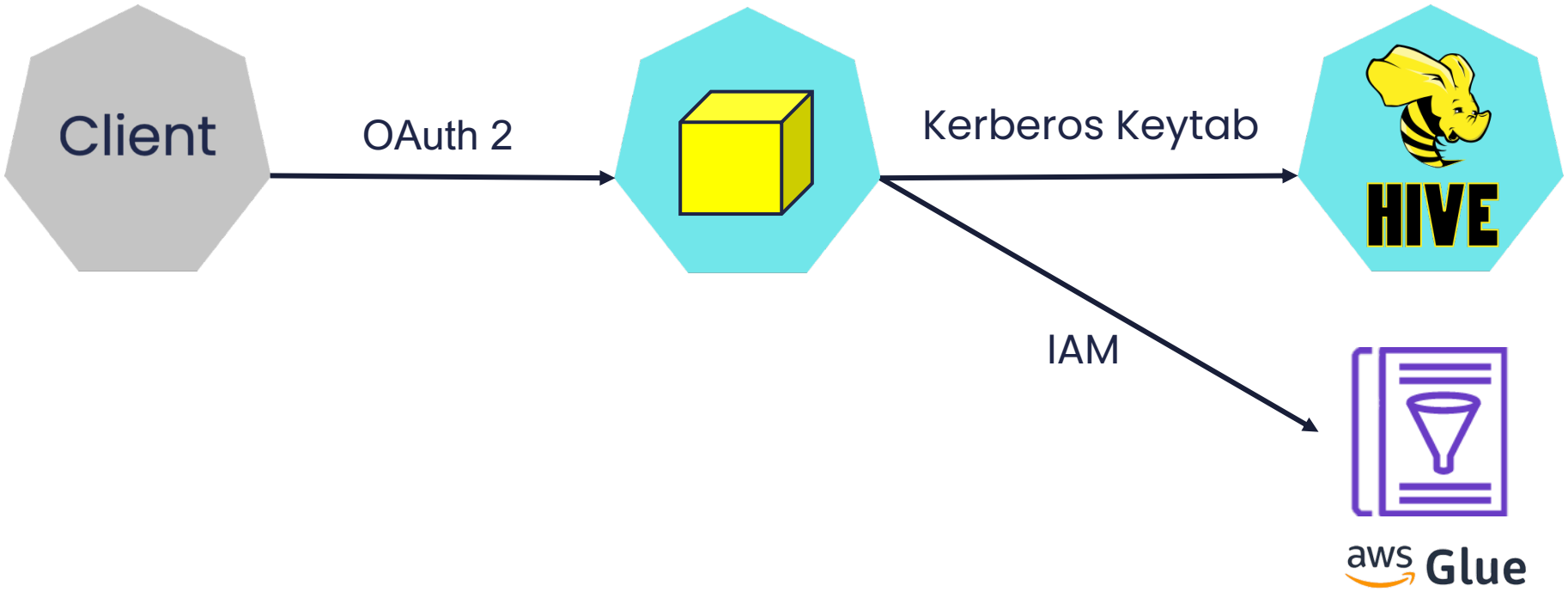
💀 Authentication is based on Kerberos

✨Spark 2 can’t easily switch between catalogs

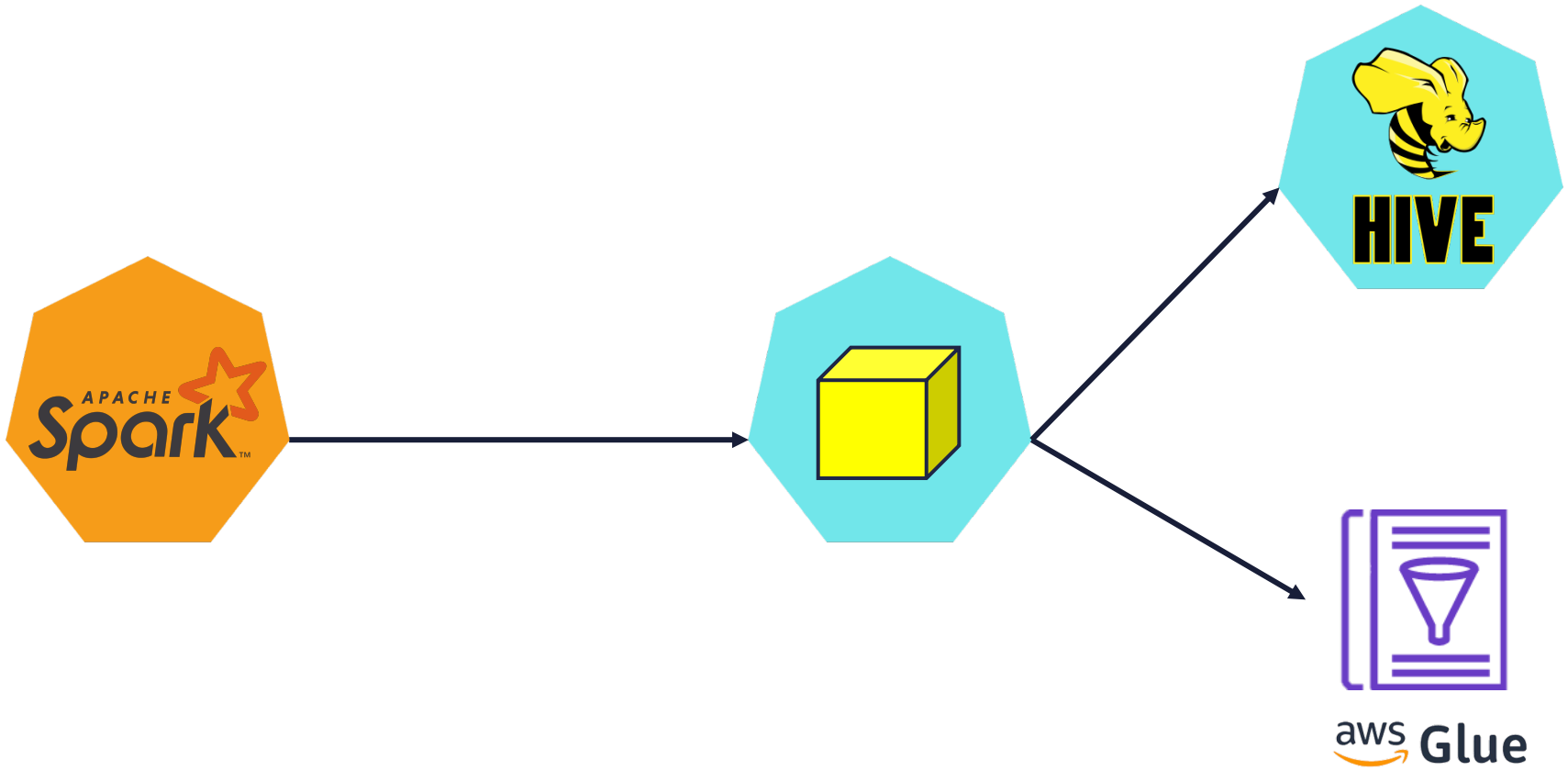
Kerberos

Hundreds of developers struggling with Kerberos?

Not on my watch



Spark single catalog limitation



Resource Manager for Spark

✅ Deployable in any cloud

❌ Huge pain to operationalise

Resource Manager for Spark

CLOUDERA

✅ Deployable in any cloud
✅ Easier to operationalise

❌ Quite hard to integrate across instances

Resource Manager for Spark

✅ Deployable in any cloud
✅ Easier to operationalise
✅ Federation ease integration

❌ Vendor lock-in

Resource Manager for Spark

✅ Deployable in any cloud
✅ No vendor lock-in
✅ Good isolation
✅ Huge configurability

❌ Huge pain to operationalise
❌ Maintain your Spark “distro”

Resource Manager for Spark

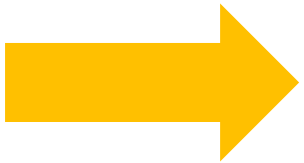
- ✅ Deployable in any cloud
- ✅ Easier to operationalise
- ✅ No vendor lock-in
- ✅ Good isolation
- ✅ Huge configurability

- ❌ Maintain your Spark “distro”

Smooth migration – evil master plan

1. Hide low level details
2. Introduce concept of deployment options
3. Build multi-cloud support
4. Add Azure deployment options
5. Gently roll out upgrades
6. Profit🤞

Step 1.1 – Hide storage and table provisioning

Cloudformation  Cloud agnostic descriptor

Step 1.1 – Hide storage and table provisioning

```yaml
version: v3
type: parallelread
entityName: customer
entityVersion: v1
description: customer data from the financial system
coldStorage:
  format: parquet
  location:
    bucket: finance_customer_${}
    prefix: bronze
    provider: aws|azure
  hasHistory: false
  partitioningColumns: [country]
columns:
  - name: fullName
    type: string
          - dgItaly1
  - name: address
    type: string
    optional: true
```

agilelab

Step 1.1 – Hide storage and table provisioning

```
version: v3
type: parallelread
entityName: customer
entityVersion: v1
description: customer data from the financial system
coldStorage:
  format: parquet
  location:
    bucket: finance_customer_${}
    prefix: bronze
    provider: aws|azure
  hasHistory: false
  partitioningColumns: [country]
columns:
  - name: fullName
    type: string
          - dgItaly1
  - name: address
    type: string
    optional: true
```

Step 1.2 – hide Spark low level details

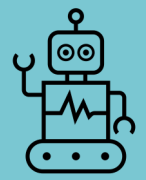
Step 1.2 – hide Spark low level details

pre-built Templates

Step 2.1 – Deployment options

Introduce batch job deployment descriptor with deployment target

The project skeleton/template is auto-configured from the deployment descriptor



The CI/CD pipeline takes care of setting everything up

Step 2.1 – Deployment options

```yaml
specVersion: v1
name: "helloworld"
sources:
  - domain: "Finance"
    name: "customer"
    version: "v1"
  - domain: "Ecommerce"
    name: "orders"
    version: "v2"
sink:
  domain: "Marketing"
  name: "most_spending_customers"
  version: "v1"
framework: #technology specific configuration
  kind: spark-2.4.4-emr6-livy-scala
  cluster: emr1
  entrypointClass: "io.contoso.app.HelloWorld"
  dependencies: []
  configurations:
    spark.task.maxFailures: 6
    spark.dynamicAllocation.enabled: false
```

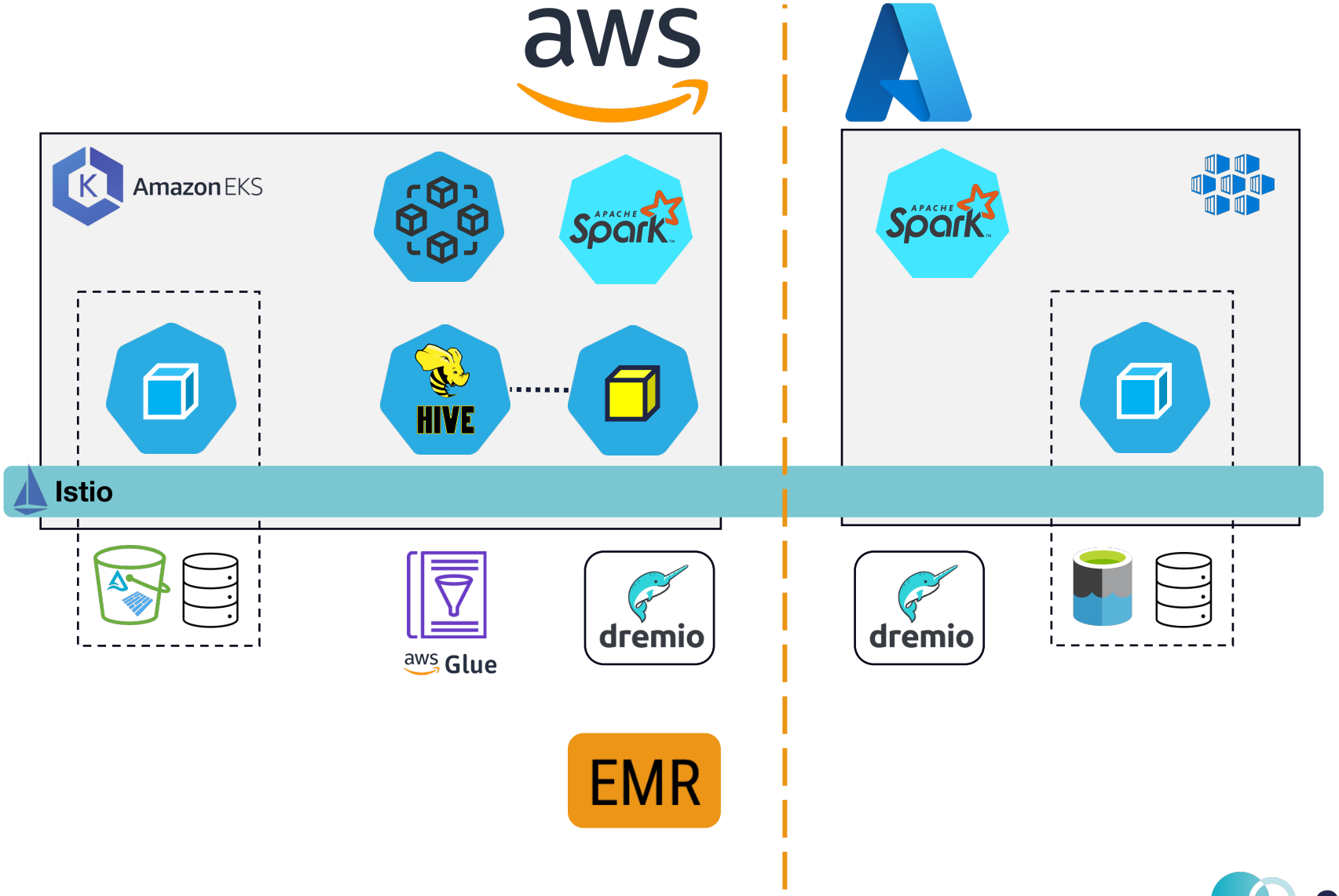
Step 3 – Build azure support

- Extend our sdk to support Azure/ADLS
- Prepare base container images
- Hide s3/adls setup on SDK side
- Setup spark context with custom catalog

Step 4 – Add azure support

```yaml
specVersion: v1
name: "helloworld"
sources:
  - domain: Finance
    name: customer
    version: v1
  - domain: Ecommerce
    name: orders
    version: v2
sink:
  domain: Marketing
  name: most_spending_customer
  version: v1
framework: # technologyspecific configuration
  kind: spark-3.4.1-k8s-livy-scala
  cluster: azure1
  entrypointClass: io.contoso.app.HelloWorld
  dependencies: [ ]
```

Full picture



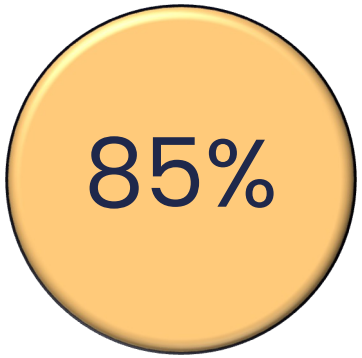
Gently roll-out

- We don’t FORCE anyone to update to Spark newest version
- New applications run on Kubernetes by default
- Applications running on EMR can’t access datasets stored on Azure
- So we give them a good reason to update!

Roll out status – 6 months in

Thanks to the platform custom catalog we know that:

multi-cloud



upgraded

New app on Azure
(3% overall)

agilelab

Lessons learnt

Abstracting infra is good way of handling a transition

Truly open standards are there and help a lot

You can/should create your internal standards (when needed)

You **need** a Platform Team

Data has gravity (egress traffic is expensive)

Next steps – caching

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

Next steps – smart scheduler

My favourite subject at school was avoiding unnecessary work

Prince Philip

agilelab

Next steps – true independence

Independence is not freedom, it's a responsibility

agilelab

agilelab

Thank you!

Questions?

www.agilelab.it

@agilelab_official

linkedin.com/company/agile-lab

@agilelabsrl