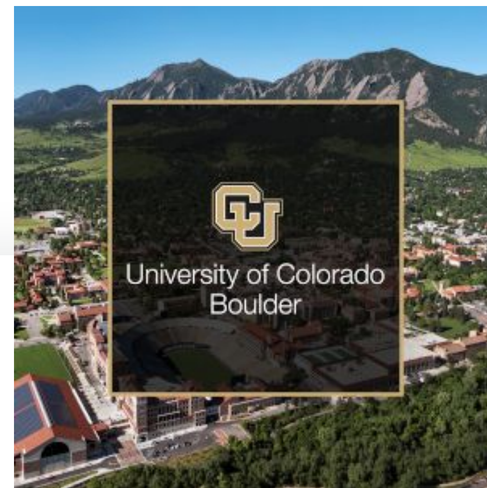
Reliability, Cost, Performance

Pick 3

Suyog Soti
2026

About Me

Why am I here?



University of Colorado Boulder (2020)

Applied Math and Computer Science

Google (2020 - 2023)

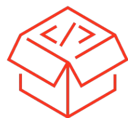
Worked on resource allocation and isolation for multi-tenant services on GCP

Databricks (2023 - Present)

Overload Protection, Service Discovery and Load balancing

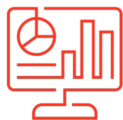
Obligatory: About Databricks :)

The unified data platform



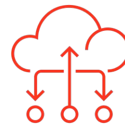
Store big data

- Delta/Iceberg Formats
- Store it in S3
- Metadata in Unity Catalog



Query big data

- Spark to query data fast
- Governance via Unity Catalog
- Build business dashboards



Build on top of the data

- Create agents
- Fine tune models
- Lakewatch
- Lakehouse Apps

Common Wisdom

Build a low latency, highly scalable service with a 99.999 uptime

Dimensions

Autoscale

- 0 downtime deployments
- Scale horizontally with load
- Resource Utilization

Why care about these

- Reliability
- Cost

Storage

- Keep consistent and durable state via some remote storage layer

- Performance
- Reliability
- Cost

Caching

Scalable and eventually consistent reads

- Fall back to storage layer

- Performance
- Reliability
- Cost

The Setup

How healthy are these replica's?

Do these replicas need to hold state?

Replica 1

Client

API Gateway (Envoy)

Replica 2

DB

Replica 3

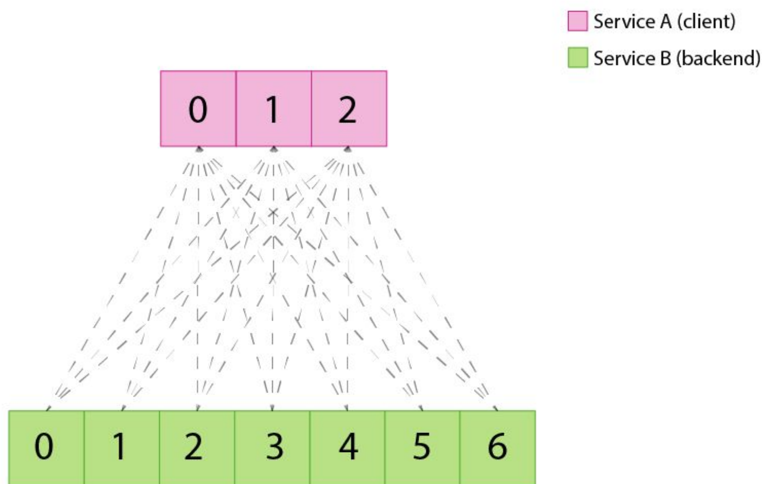
Are all requests created equal?

What LB Policy Should We Use?

Notice there are no LB's

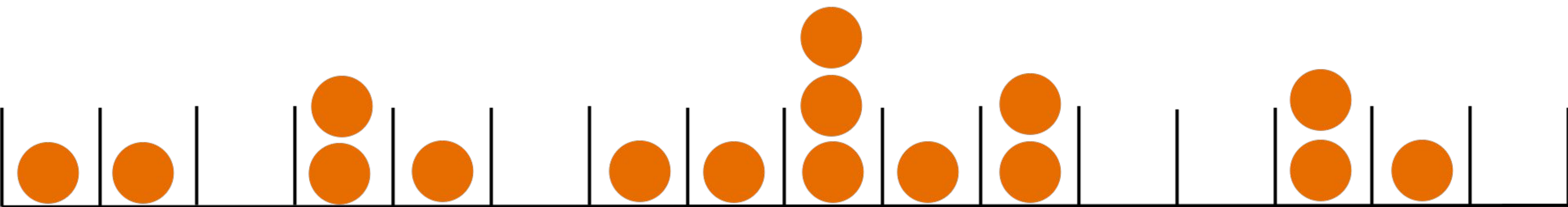
LB Algorithms – Stateless

- Random Routing
  - Uniform Distribution
- P2C
  - Assign weights, pick 2 random replicas, choose replica with lower weight



LB Algorithms – Static Stateful

- Recall LB Lecture
- Hash key matters
- Requests are not homogenous



Common Wisdom

Autoscale

- 0 downtime deployments
- Scale horizontally with load
- Resource Utilization

With stateless services

- 0 downtime deployments
- Scale horizontally with load

Storage

- Keep consistent and durable state via some remote storage layer

- Databases don't scale well per tenant
- Network IO Overhead
- Serialization Overhead

Caching

Scalable and eventually consistent reads

- Fall back to storage layer

- Low cache hit rates
- Network IO Overhead
- Serialization Overhead

Why not static sharding?

Autoscale

- 0 downtime deployments
- Scale horizontally with load
- Resource Utilization

With statically sharded services

- 0 downtime deployments
- Scale horizontally with load
- **Hot Spots**

Storage

- Keep consistent and durable state via some remote storage layer

- Databases scale well per tenant (batch writes/reads)
- Network IO Overhead
- Serialization Overhead

Caching

Scalable and eventually consistent reads

- Fall back to storage layer

- High cache hit rates
- Low Network IO Overhead
- Low Serialization Overhead

The Way – Dynamic Autosharder

Distributed data path*

Centralized control path*

Hash(key)

Hash(key)

Clients

Clerk

Slicelet

Deployment/StatefulSet

Dicer Service

Dynamic Autosharder

Autoscale

- 0 downtime deployments
- Scale horizontally with load
- Resource Utilization

With dynamically sharded services

- 0 downtime deployments
- Scale horizontally with load

Storage

- Keep consistent and durable state via some remote storage layer

- Databases scale well per tenant (batch writes/reads)
- Network IO Overhead
- Serialization Overhead

Caching

Scalable and eventually consistent reads

- Fall back to storage layer

- High cache hit rates
- Low Network IO Overhead
- Low Serialization Overhead

Use Case: Ratelimit and in memory stores

Critical path
non-critical path

Resource to be protected

Caller Service
RLSv2 client library

rlg_dimension_name_value_1: rejection rate = 60%
rlg_dimension_name_value_2: rejection rate = 30%

rlg_dimension_name_value_1: allowed = 10
rejected = 5
rlg_dimension_name_value_2: allowed = 20
rejected = 8

RLSv2

Redis
to be deprecated

rlg_dimension_name_value_1: 50
rlg_dimension_name_value_2: 100
...

Clients hash based on Rate Limit Group and Dimension

RatelimitV2Client

Clerk

Clerk

Dicer Assigner Service

Slicelet

Ratelimit-v2 Deployment

Use Case: Model Serving – Batch Inference

User

Control Plane

Data Plane

General Cluster

1 Create inference session

2 Update config map containing the per-model adapterID -> image prefix + SHA mapping

Llama 3.1 70b v1 us-west-2 ConfigMap

Llama 3.1 70b v1 us-east-1 ConfigMap

```
0050bd285b8143d58bca5ed2b2ca5fcd:
/multitenant_adapters/llama-3-70b/1/60519214
18418893/0050bd285b8143d58bca5ed2b2ca5fcd

839accd0610d40ed9aa4ac681ec8b6ad:
/multitenant_adapters/llama-3-70b/1/60519214
18418893/839accd0610d40ed9aa4ac681ec8b6ad

...
```

3 Each server mounts the configmap as a volume and reads the image path to download an adapter when needed from the mount path

MT serving scheduler

Serving DB

ai_query

Model Serving Proxy

dp-apiproxy

Dicer Clerk
→ [0, 1, 2]
→ [1, 2]
→ [0]
→ [0, 1]

Llama 3.1 70b server 0
slicelet

Llama 3.1 70b server 1
slicelet

Llama 3.1 70b server 2
slicelet

Dicer

4 Along the request path, MSP reads from the serving DB to attach the correct adapter ID in the request header.

5 Dicer clerk will determine which pod to forward a request to based on the adapter ID header and load reported per pod

6 Dicer monitors load of each adapter ID emitted by the slicelet and communicate assignments back to the slicelet and to the clerk

Artifact Registry

4x more PEFT adapters loaded

Save 10 mil/year

The new problems

Clients in control

- Proxies
- Pod to Pod Forwarding

Overload Protection

- Interactions with load shedder
- Load and health metrics not perfect
    - Clients often do reachability checks

Case Study: Unity Catalog

Client

API Gateway
(Envoy)