Building Reliable Services on the Cloud

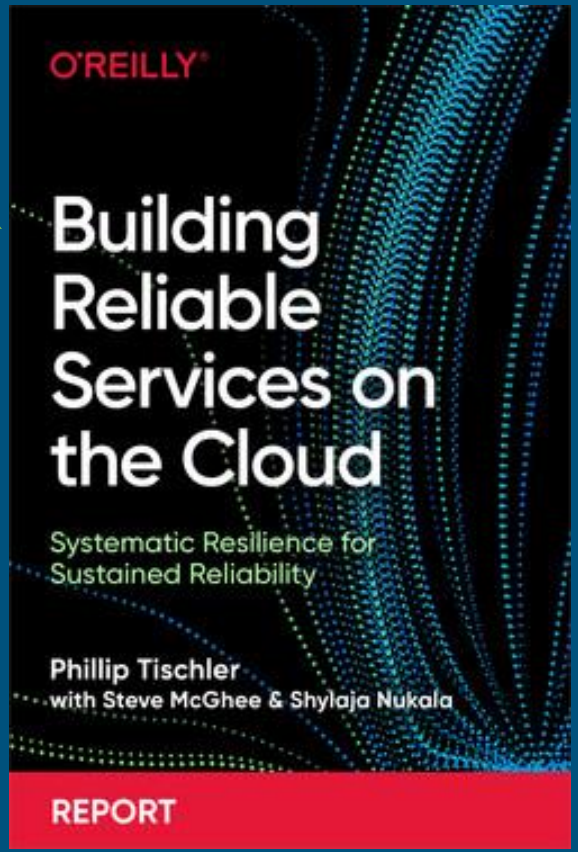
smcghee@google.com

The Report

Authored by Phillip Tischler
with Steve McGhee
and Shylaja Nukala

smcghee@google.com
Reliability Advocate, SRE

10+ years in SRE
Now: helping cloud customers



https://info.blameless.com/oreilly-building-reliable-services-on-the-cloud

TOC

1. **Define objectives**
2. **Identify the dependencies**
3. **Architect your service**
4. **Avoid common failure modes**

0. Motivation

Outages will erode trust and motivate users to adopt alternatives
Data loss is likely to destroy trust

Why?

Google has designed, built, and operated **reliable services** on the cloud for **decades**. This is what we've learned.

What?

Who?

Here, learn **how to build similarly reliable services** as a software engineer, site reliability engineer, or cloud engineer.

1. Define Your Objectives

Reliability is a **non-functional requirement**, and it has **gradations**.

- You can have "**no**" reliability
- You can have "**some**" reliability
- You can have "**too much**" reliability (because $$$)

So how do we decide "how much" to **invest**?

SLOs

Service Level Objectives (SLOs)

By using SLOs, you can choose and enforce your acceptable levels of reliability.

Choose SLOs based on user or business needs.

	Percent of service unavailable				
		0.1%	1%	10%	100%
Availability SLO	90%	Forever	Forever	Forever	9d
	99%	Forever	Forever	9d	21h
	99.9%	Forever	9d	21h	2h
	99.99%	9d	21h	2h	12m
	99.999%	21h	2h	12m	1m

Different parts of your system can have different reliability objectives.

Rule of thumb: every extra 9 costs ~10x what the last one did.

	User-facing service (e.g., blog website)	Product infra (e.g., login service)	Technical infra (e.g., database)
Critical features/ data plane	99.99%	99.99%	99.999%
Optional features/ control plane	99.9%	99.9%	99.9%

https://cloud.google.com/architecture/defining-SLOs

Understand Failure Domains and Redundancy

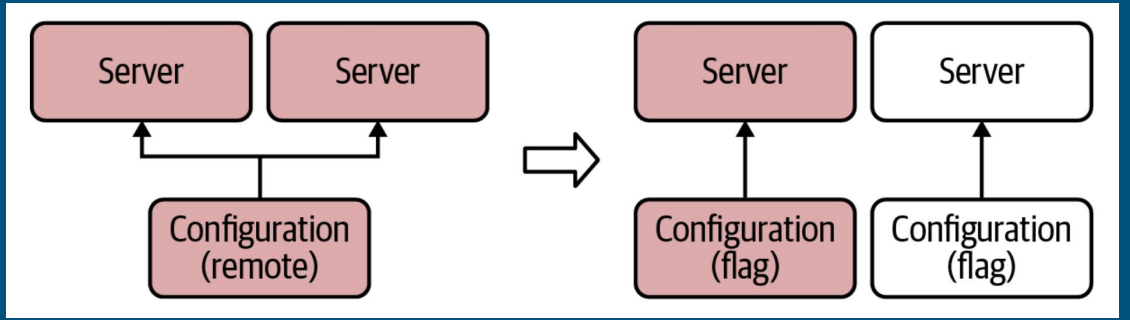
A failure domain is **a group of resources** that can **fail as a unit**, making services deployed within that unit unavailable.

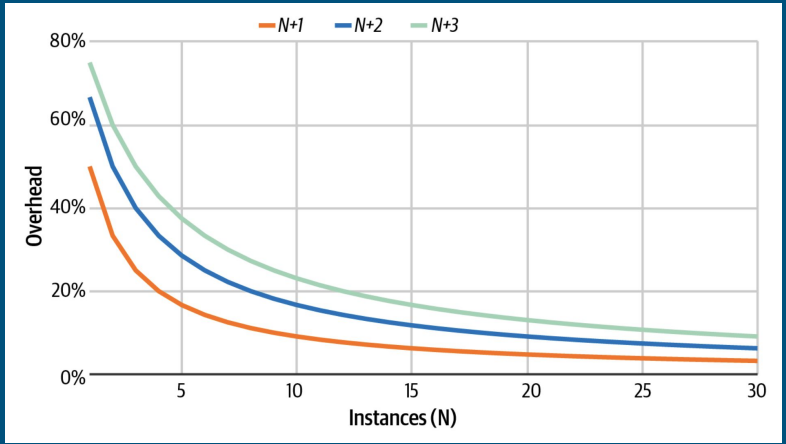
eg: a Host, Rack, Row, Cluster, Datacenter, or Campus

Push changes into **smaller** failure domains, try not to **span** failure domains.

Redundancy across failure domains provides resilience, at an overhead.

N+2 redundancy provides for two independent failures (one planned, one unplanned)





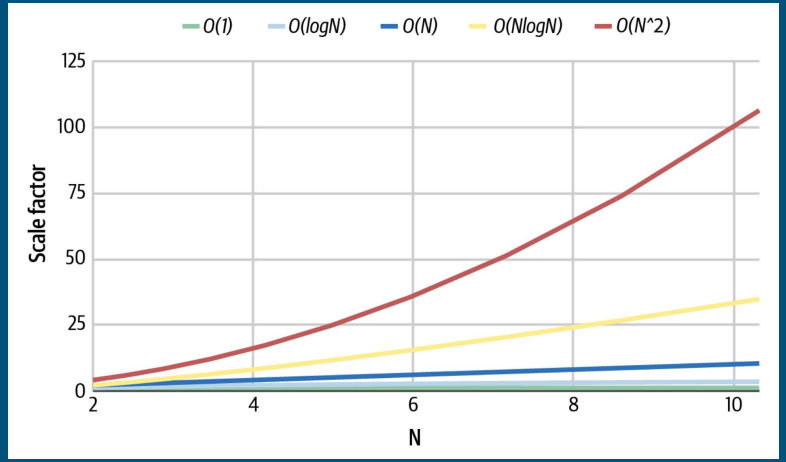
Consider Scaling

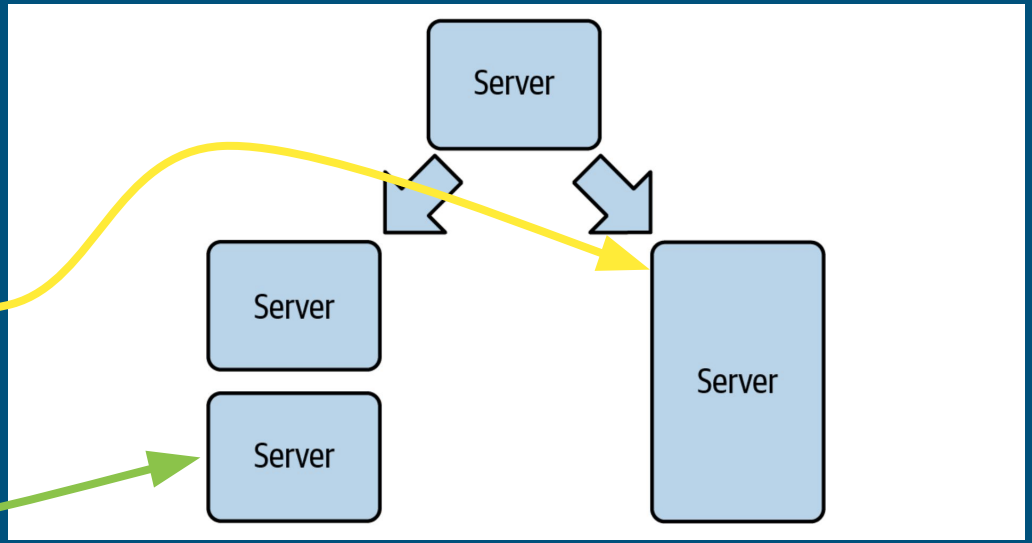
Input-to-consumption relationship is often expressed in algorithmic "Big O" notation:
`O(1)`, `O(log N)`, `O(N)`, `O(NlogN)`, `O(N^2)`, `O(N^N)`

Changing a system to a lower asymptotic complexity class can have **dramatic improvements**

Scaling can be done by increasing the capacity of a resource (**vertical scaling**),

or by using more instances of a resource (**horizontal scaling**).





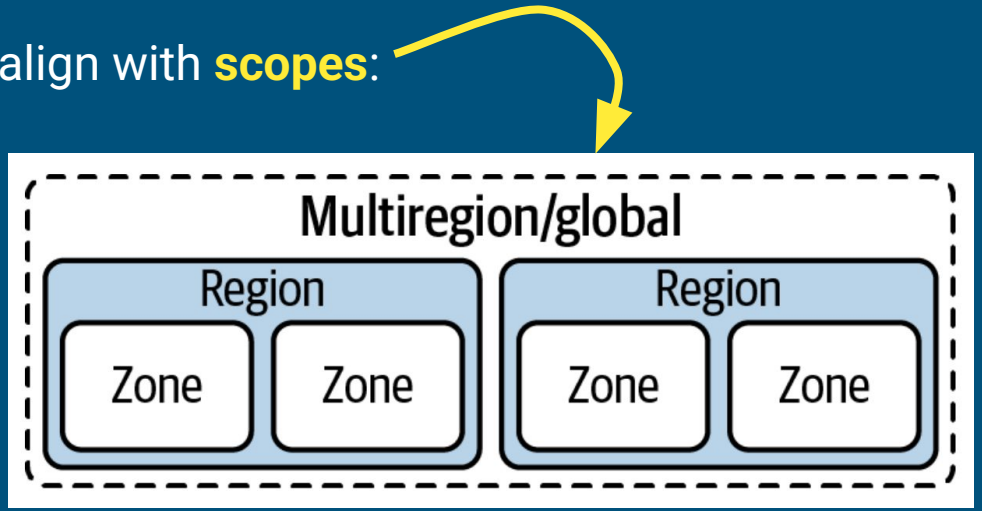
2. Know Your Dependencies

Once you have the goals down, it's time to **build**.

Our building blocks have limitations, even in Cloud!

First, Cloud **services** and **resources** align with **scopes**:

"Lower" scopes (zone) have lower availability, but there are far more of them!



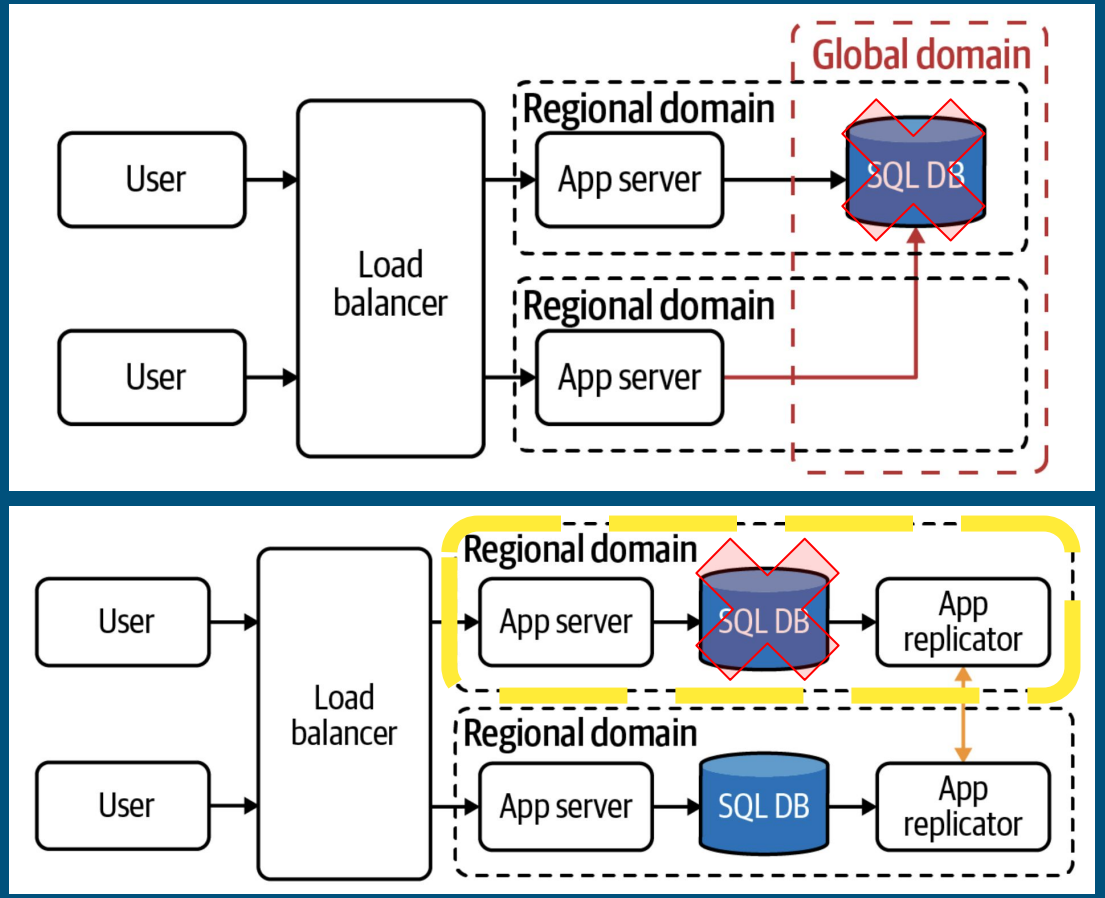
Align Failure Domains

"Stacks" of services should **align** in terms of failure domain.

Misalignment can cause **coordinated failure** (which is bad)

If you align your services, the **loss** of a service in one failure domain will **only affect the stack in that failure domain**.

Note the stack itself doesn't change, just the **deployment archetype** did.



"Deployment Archetypes for Cloud Applications" B. Calder, A. Berenberg
https://arxiv.org/pdf/2105.00560.pdf

Consider Cloud Service SLOs

When **composing** Cloud services, consider the SLOs of each service you're using and compare it to the SLO you intend to hold yourself to.

How do you depend on these?

Intersectional dependency:

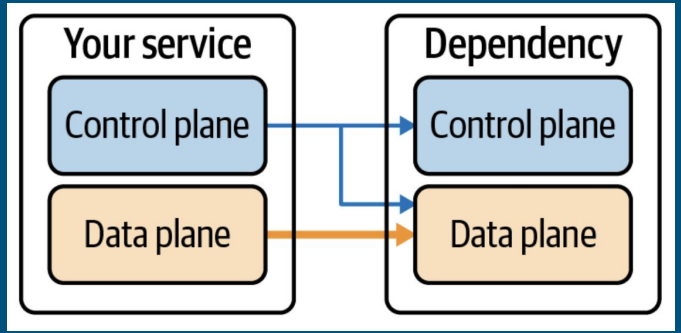
★ **All** must be up

Union dependency:

★ **At least one** must be up

Explicitly consider your Control/Data planes separately!

	Intersection availability per number of components			Union availability per number of components	
Component availability	3	10	100	2	3
99%	97%	90%	37%	99.99%	99.9999%
99.9%	99.7%	99%	90%	99.9999%	99.9999999%
99.99%	99.97%	99.9%	99%	99.999999%	99.9999999999%
99.999%	99.997%	99.99%	99.9%	99.99999999%	99.9999999999999%



Choose Services to Depend On

Compute:

- Use containers! Keep them small, start with serverless.
- Optimize for startup time, implement ready/live checks, terminate gracefully.

Network:

- Use provider's CDN, Load Balancers, private WANs, service meshes

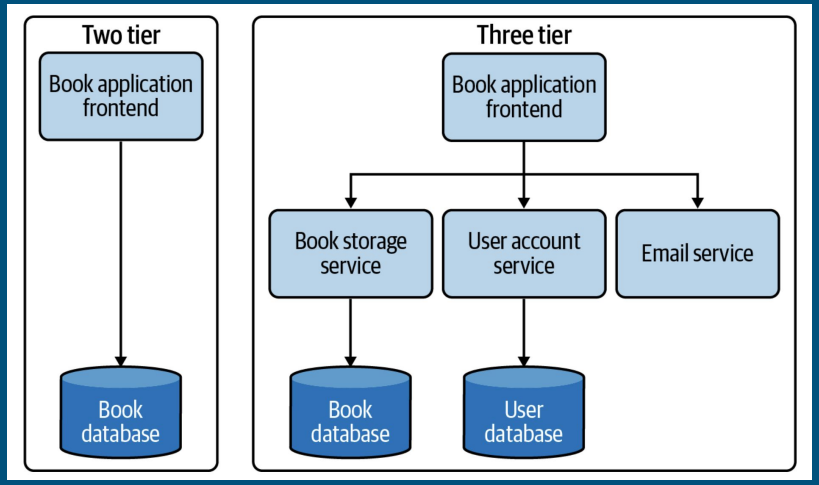
Storage:

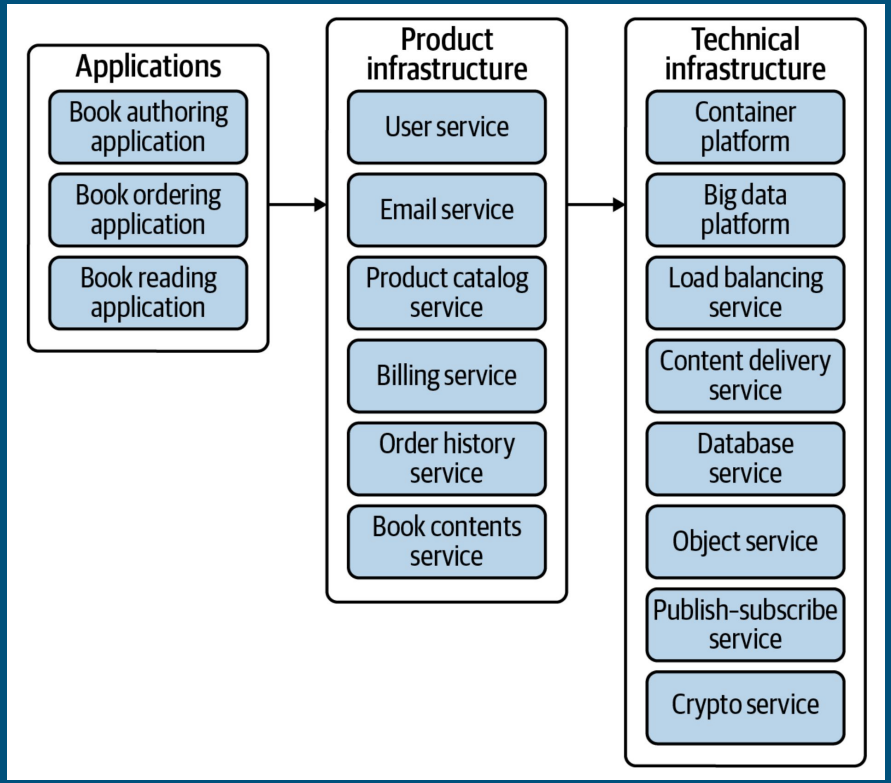
- Consider object stores, NoSQL databases, multi-regional database services
- Use Publish/Subscribe service to decouple readers/writers, improve retries
- Consider MapReduce/Flume for high volumes of data

3. Architect Your Service

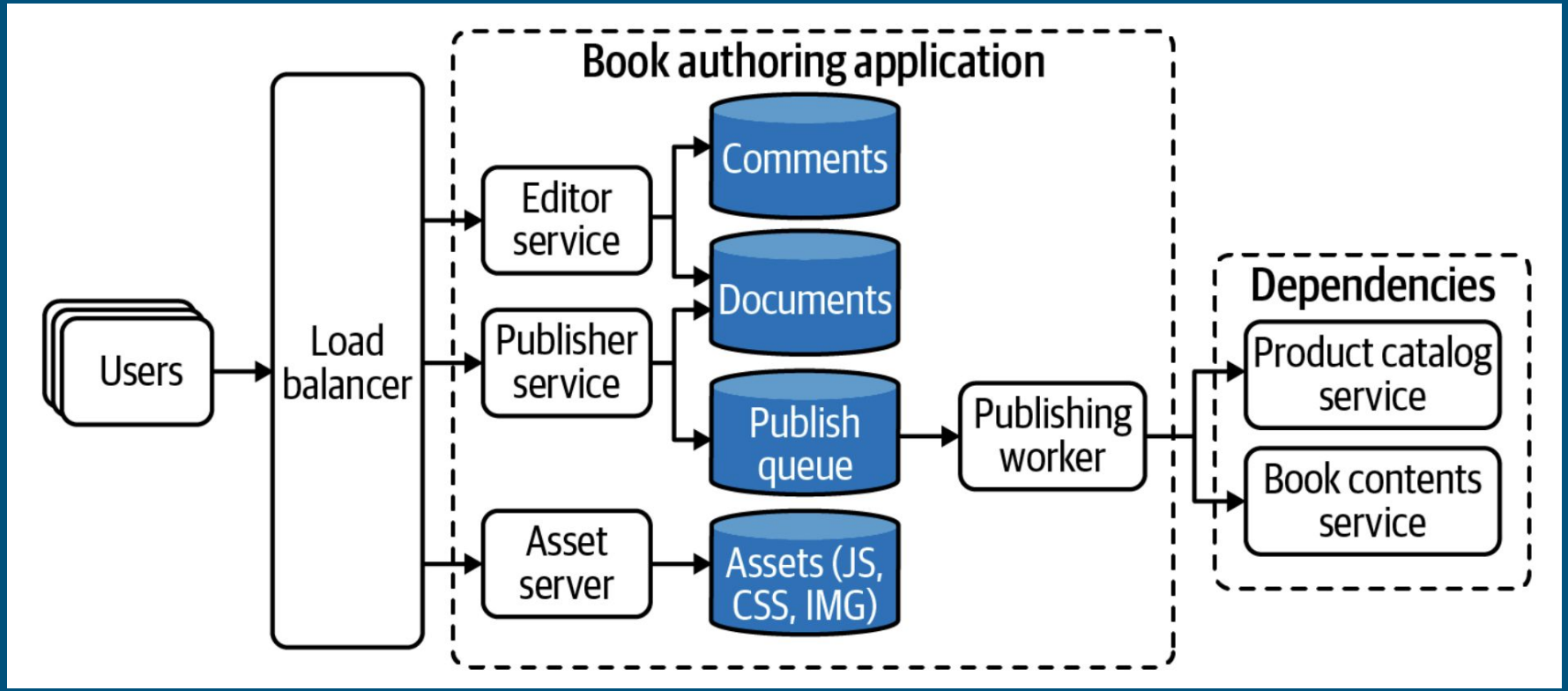
- ★ Decompose a larger system into smaller components (Services)
- ★ Use well-defined interfaces and loose-coupling (APIs)
- ★ So the aggregate system behaves and performs as desired (Scaling)

Understand Service Evolution: Tiers, SOA





Understand Service Evolution: Microservices

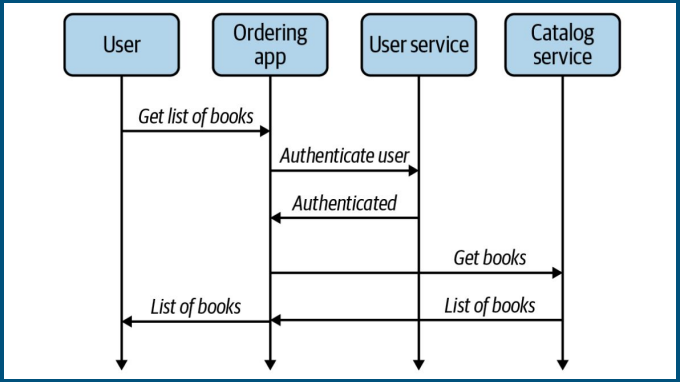


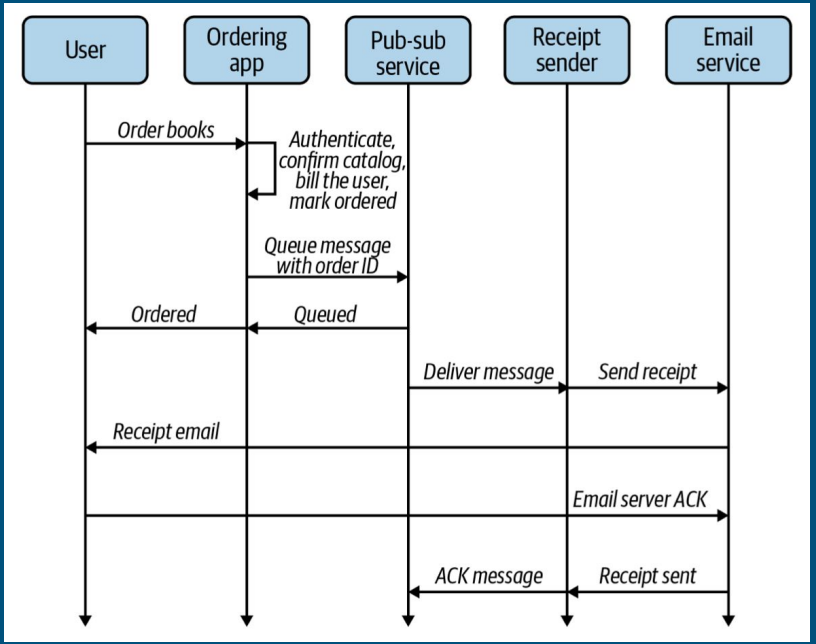
Choose Synchronous / Asynchronous

Synchronous operations:
- client **waits for the service**
- strongly/causal consistency
- **tight coupling** of server/client

Asynchronous operations:
- client does not wait
- operation completes independently
- looser coupling
- client **not blocked**

Consider using a **Pub/Sub** queue:
- consider idempotency to dedupe retries
- might provide: at-most-once, at-least-once, or exactly-once execution



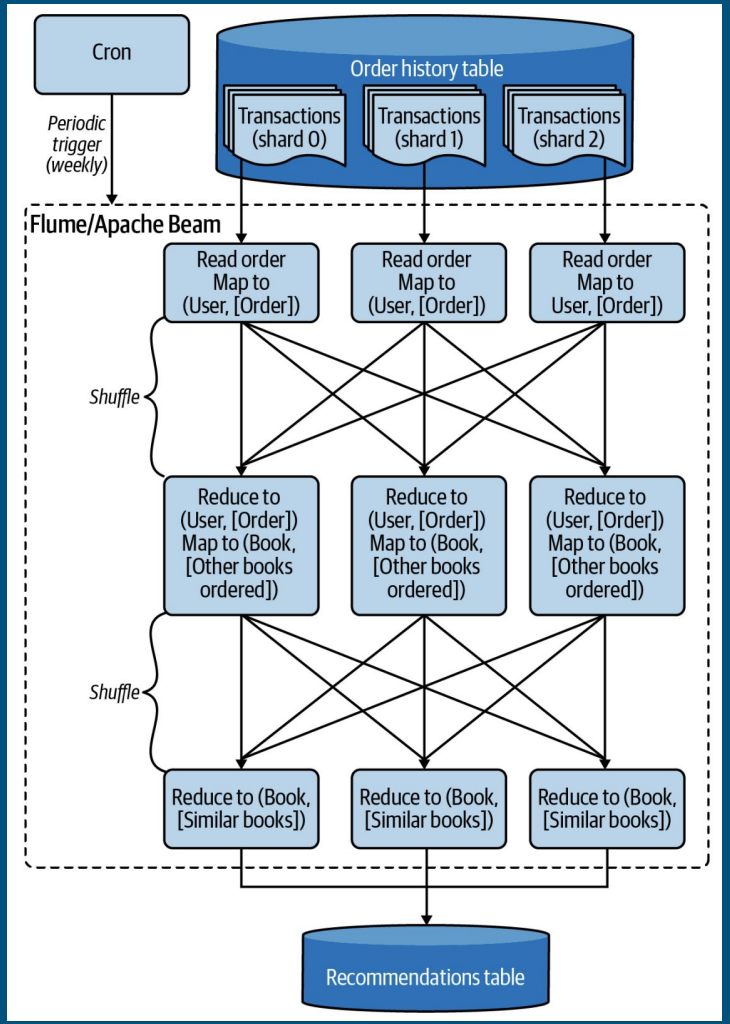


Utilize Batch Computing

When large computations or large datasets are processed all at once.

can be **more efficient** through:

- global operations like external sort and join
- use cheaper preemptible compute

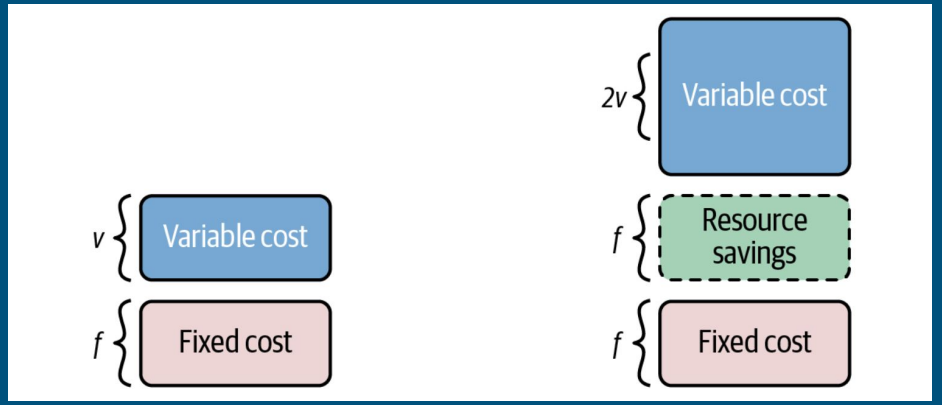


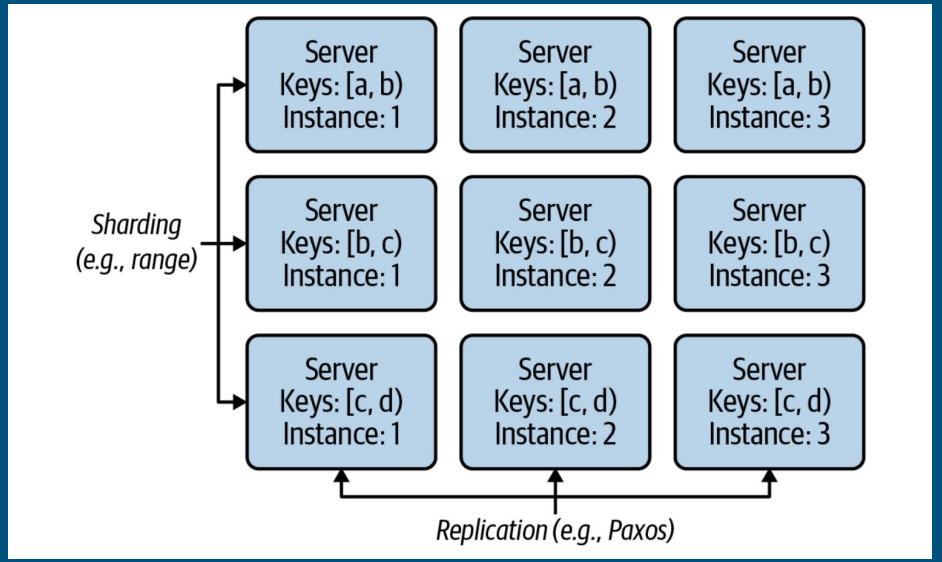
Understand Horizontal vs Vertical Scaling

Vertical scaling can gain more efficiency, to a limit. Well-controlled scaling can have controlled costs. Each node has a fixed cost.

Horizontal scaling can allow more traffic and storage, quickly. Rapid growth and large geographical spread work better with horizontal scaling.

Utilize algorithmic sharding and replication to allow for automated growth.





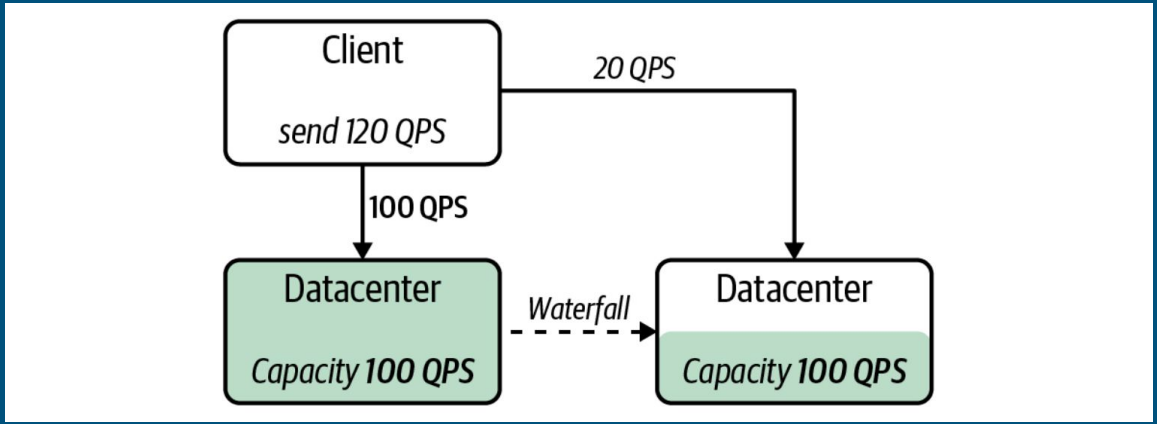
Load Balancing

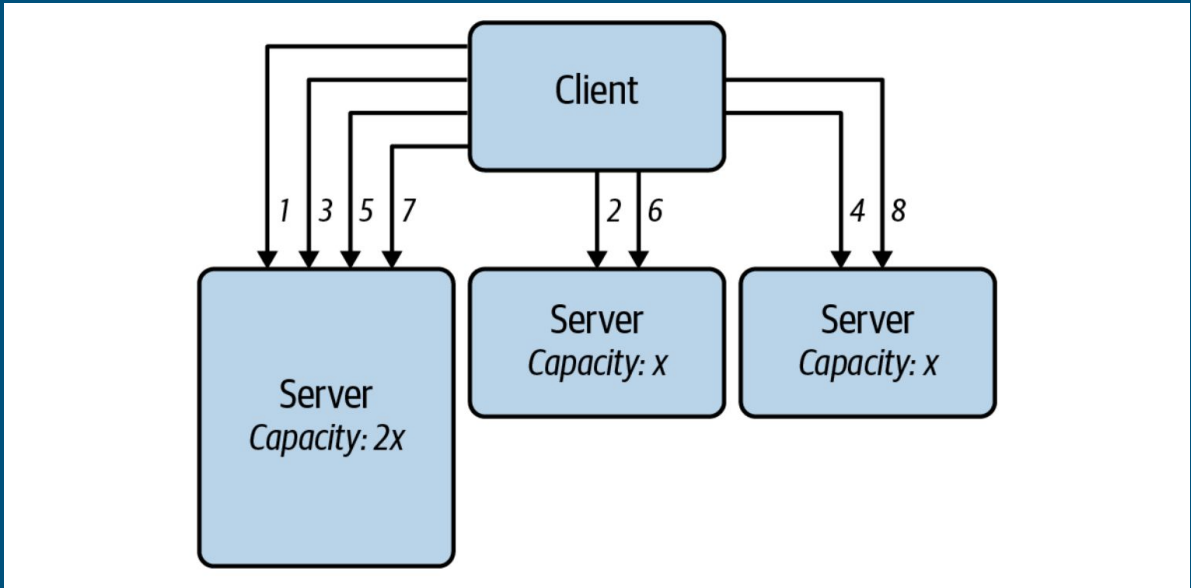
Automated deployment and **autoscaling** require dynamic load balancing.

Simple "waterfall" load balancing for homogeneous services with stable, consistent requests.

Some services may require knowing the "cost" of a query, as well as "capacity" of a server.

Utilization-based balancing might measure the servers directly and **infer** their capacity via other metrics.





4. Avoid Common Failure Modes

Resilience is not about building a perfect system, but designing methods for handling issues like:

- Bad Changes
- Cascading Failures
- Thundering Herds
- Hotspots
- Data Loss or Corruption

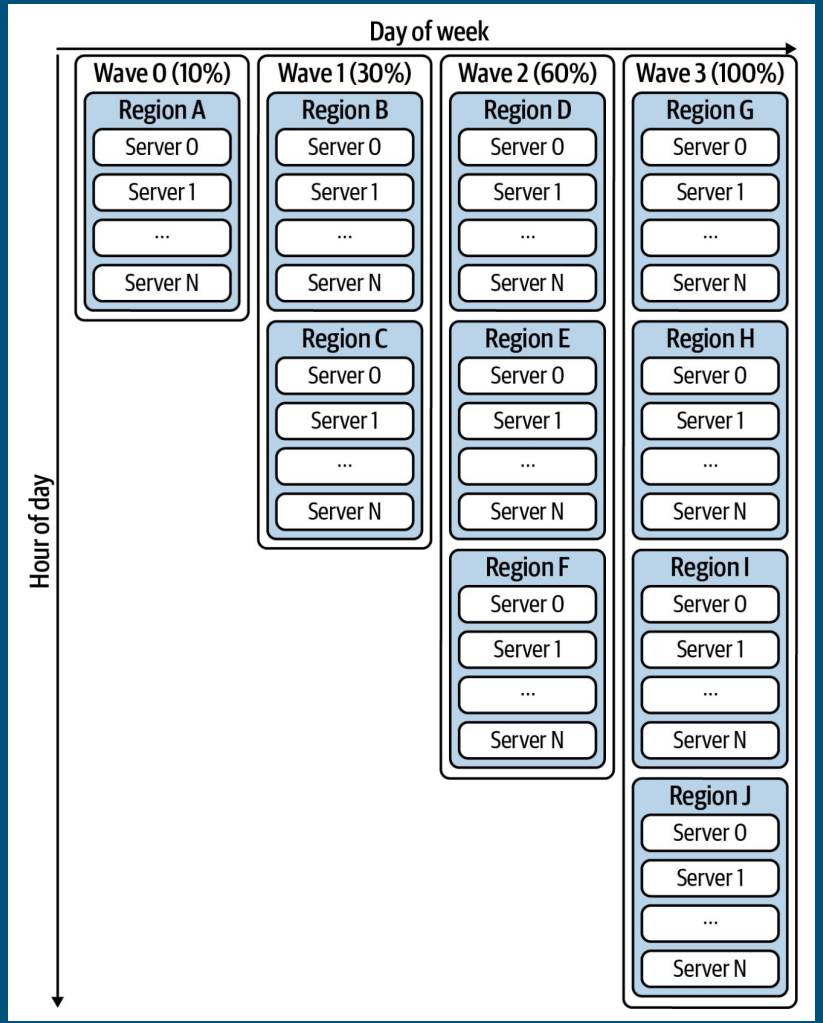
Avoid Bad Changes

Change Supervision	Monitor services to detect faults, including dependencies and dependents for end-to-end coverage.	- Alert on SLO burn - Monitor service health metrics - Add synthetic end-to-end probes
Progressive Rollout	Make changes slowly across isolated failure domains so issues can be detected and mitigated before having a large impact.	- Create small, independent failure domains - Update those domain one at a time
Safe & Tested Mitigations	Have rapid, low-risk, and tested mitigations like change rollback and automatically execute them on issue detection.	- Apply well-tested mitigations when failures are detected - eg: immediate change rollback
Defense in Depth	Independently verify correctness of changes at each layer of the stack for multiple defenses against failure.	- Deploy to dev, staging, then prod - Use IaC and 2FA/MFA to improve confidence in changes

Utilize Gradual Rollouts

A deeper look into a model for gradual rollout of a large service:

- **start small** (canary)
- spread change across **time** (hour, day)
- spread change across **space** (regions)
- don't alter two regions (failure domains) at the same time



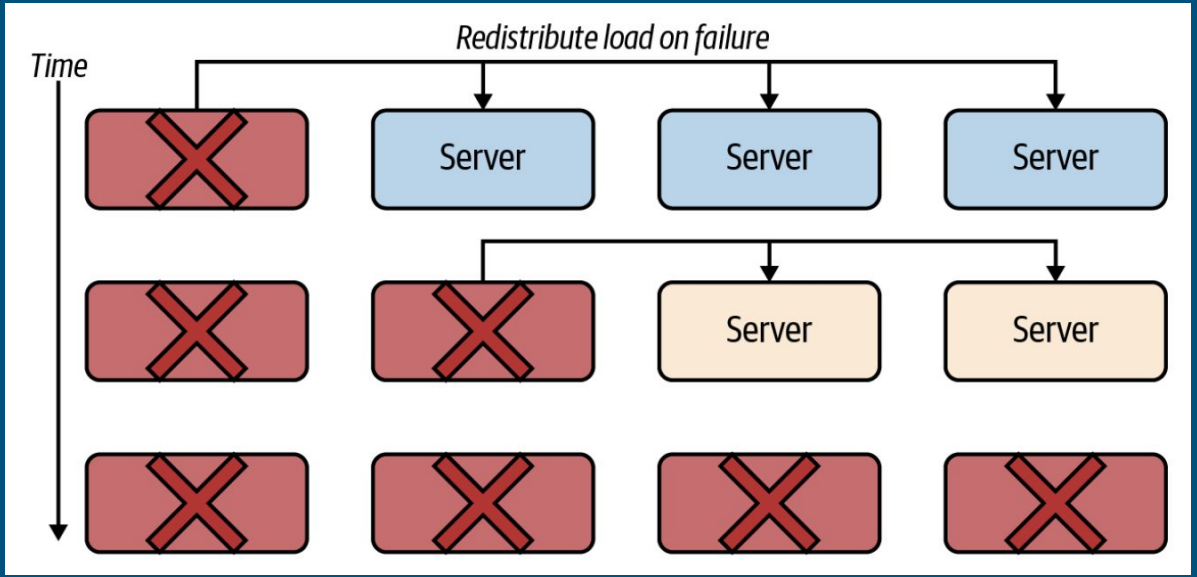
Avoid Cascading Failure

Avoid the model where failure of one component results in a retry to another component, which then fails in turn.

This can be especially nefarious in capacity caches.

Mitigations may include **explicit dropping** of excessive traffic while more capacity can be made available.

See also: Cost Modeling, DoS protection, Load Shedding, Quotas, Criticality, Autoscaling, Capacity Planning.



Beware Thundering Herds

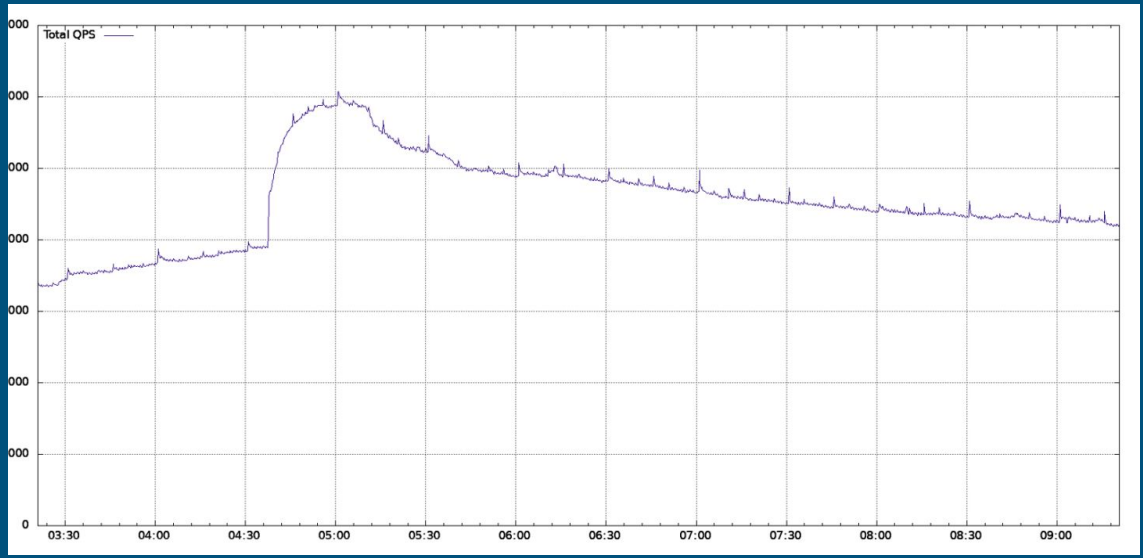
Sometimes, a "wall" of traffic can hit a service at once.

- Mobile outage resuming service
- External events: World Cup
- Synchronized cache expiration
- Your own marketing!

These can often be **too fast** for autoscaling or caching to be of use.

Mitigate instead via:

- **Exponential backoff** at client
- **Jitter** (added small, random delay)
- **Drop** excess traffic

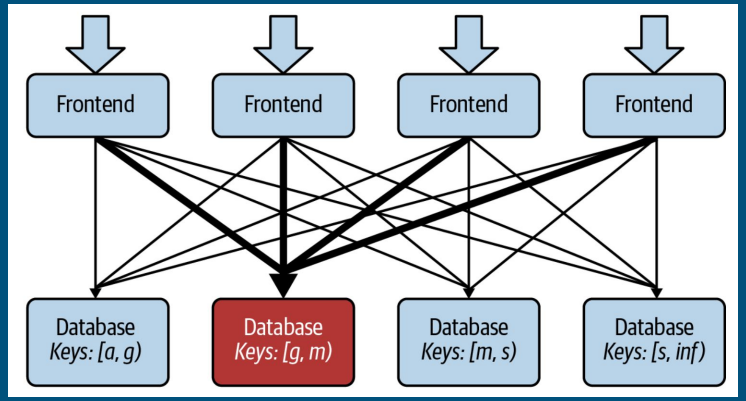


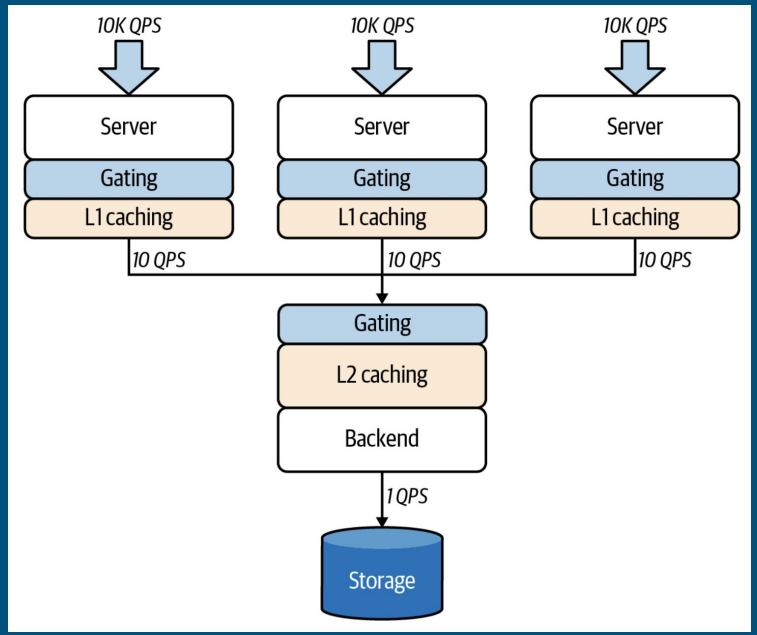
Prevent Hotspots

A hotspot is where a subset of servers receive a disproportionate amount of load and subsequently become overloaded, despite over- all service having spare capacity.

eg: popular new app is released

Mitigation: "gating" via batched request handling at multiple points in a hierarchy



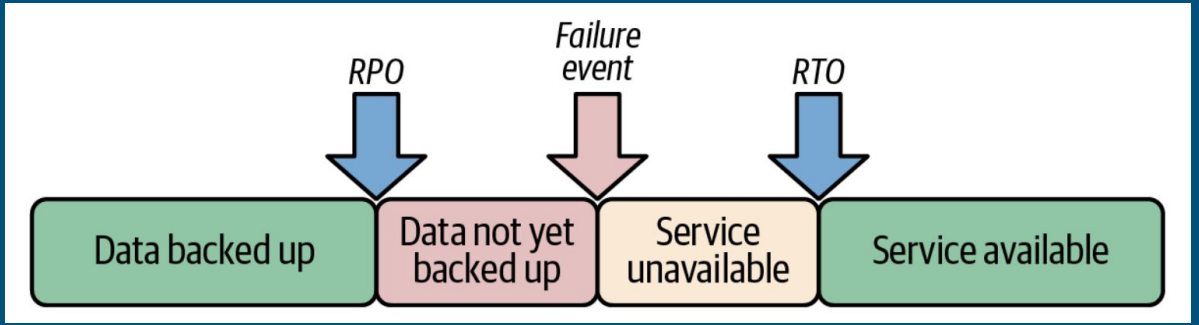


Ensure Data Integrity

SLOs may define how a service is expected to run, but you should also model how you expect to handle its data.

RPO = Recovery Point Objective

RTO = Recovery Time Objective



Know the RPO/RTO of your data storage services to ensure they match your needs

Utilize snapshots, differential backups, and full backup functionality

Test your point-in-time recovery as well as end-to-end full recovery process

Deploy probes or synthetic monitors to detect loss or corruption

Start Your Quest!

- Start with an **Archetype**
  - set expectations of reliability
  - choose an appropriate model
- Build an **Architecture**
  - know your failure domains
  - choose appropriate dependencies
- Write your **Code**
  - decouple via APIs
  - choose synch/asynch/batch
  - consider how you will scale up
- Enforce your **SLOs**
  - avoid known failure modes
  - build mitigations into your system
  - practice

Fin

* big thanks to Phil and the team at Google for writing the report and making all these great diagrams!