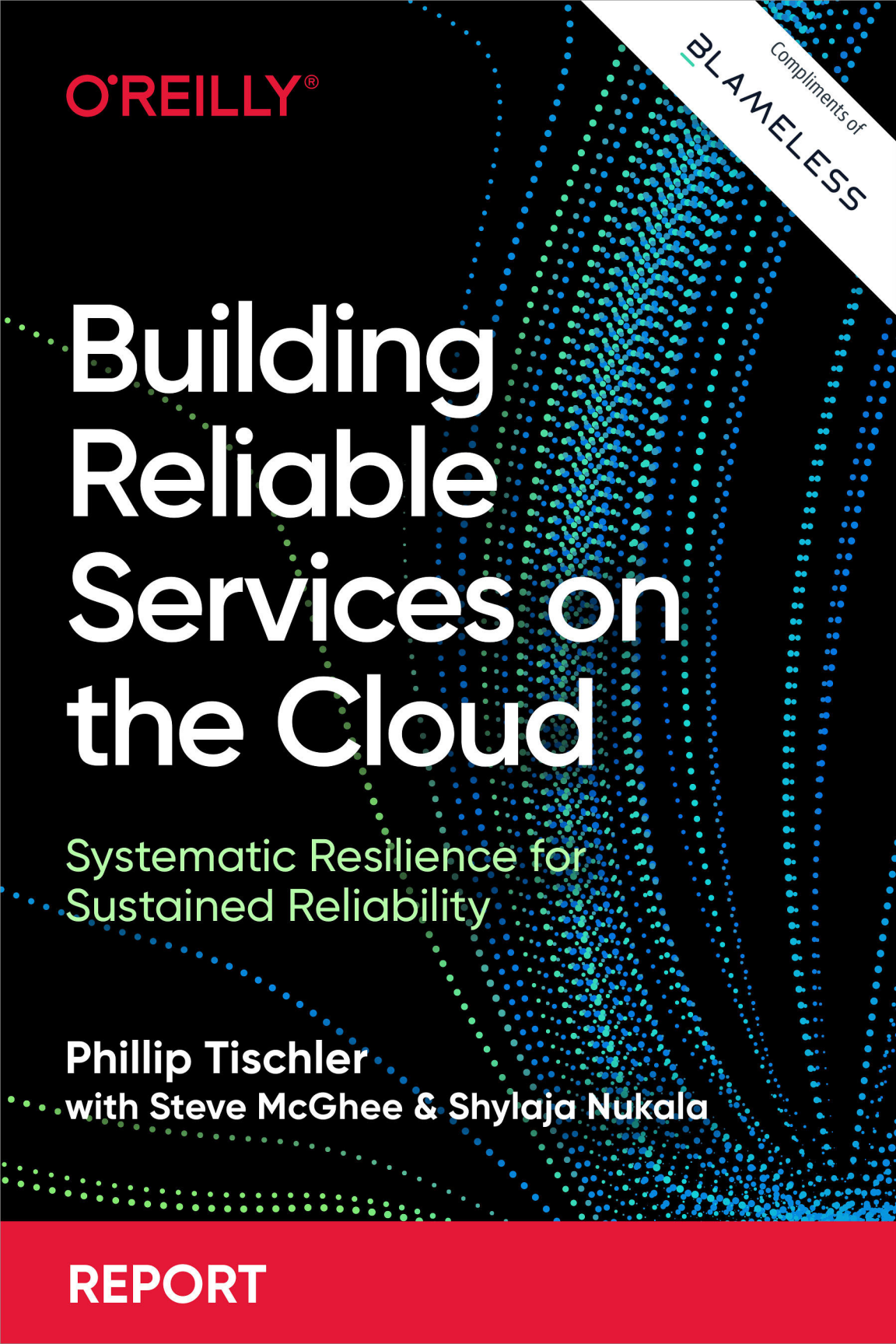
O'REILLY®
Compliments of
BLAMELESS
Building Reliable Services on the Cloud
Systematic Resilience for Sustained Reliability
Phillip Tischler
with Steve McGhee & Shylaja Nukala
REPORT

O'REILLY®

BUILDING RELIABLE SERVICES ON THE CLOUD

Google experts' tips
for cloud reliability

blameless.com

Building Reliable Services on the Cloud

Systematic Resilience for Sustained Reliability

Phillip Tischler, with Steve McGhee, and Shylaja Nukala

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Building Reliable Services on the Cloud
by Phillip Tischler

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins	**Proofreader:** Piper Editorial Consulting, LLC
Development Editor: Virginia Wilson	**Interior Designer:** David Futato
Production Editor: Kate Galloway	**Cover Designer:** Randy Comer
Copyeditor: Charles Roumeliotis	**Illustrator:** Kate Dullea

December 2021: First Edition

Revision History for the First Edition
2021-12-10: First Release

978-1-098-12033-7

[LSI]

Table of Contents

Introduction

For a product or service to be successful it must be reliable—users must trust that the service will be available when it's needed and that it won't lose the data it's entrusted to store. Outages will erode trust and motivate users to seek and adopt alternatives, and data loss is likely to destroy trust. In order to deliver a reliable system and still maintain high velocity and scalability, systematic resilience is required. Google has designed, built, and operated reliable services on the cloud for decades. In this report, you'll learn how to build similarly reliable services as a software engineer, site reliability engineer, or cloud engineer. Reliability and resiliency are very large topics, so the goal of this report is to introduce the most important concepts you should keep in mind as you design and build systems.

To build a reliable service on the cloud you should:

1. Define your objectives for the service to ensure it satisfies users and the business, and to structure the subsequent design decisions and trade-off discussions.
2. Know the dependencies that you'll use to build a service so that you can leverage them effectively to reach objectives.
3. Architect your service by developing application programming interfaces (APIs), by decomposing the system into components, and by designing the components so they contribute to the service objectives.
4. Avoid common failure modes that can create outages or otherwise cause your service to miss objectives by deploying solutions that target each failure mode.

CHAPTER 1

Define Your Objectives

Defining your service objectives will help ensure your system satisfies users while minimizing costs. Skip this step and you might design a system that is more complex than it needs to be, takes more engineers to implement, costs more to operate, or doesn't meet the needs of your users. Your objectives will influence the choice of dependencies, architectures, and means of operation.

Objectives you should define for each service include:

- Service level indicators (SLIs), service level objectives (SLOs), and service level agreements (SLAs) to set performance goals and set expectations for users
- Failure domains and redundancy so that failures don't result in service outages
- Scalability and efficiency goals so the service can grow while remaining cost effective
- Speed of change so the service remains relevant to users and your business

SLIs, SLOs, SLAs

SLIs, SLOs, and SLAs are a structure for formally defining your reliability goals as numerical measures that you can use to prioritize engineering investment, evaluate system design trade-offs, enable effective system designs across system boundaries, and reach

meaningful agreements between independent parties. These measures should cover all the needs of your users and business.

SLIs are the metrics that are measured, stored, and analyzed. Common SLIs include availability, latency, freshness, quality, and durability. SLIs also define where the measurement occurs. Measure request performance from the client to capture the user's full experience inclusive of client behavior and network performance, and also measure from the server to reduce noise and tighten evaluation of components fully under the service's control. SLOs combine an SLI with a threshold objective over a measurement window. Table 1-1 describes common SLIs and example SLOs.

Table 1-1. Common SLIs and example SLOs

SLI name	SLI definition	SLO example
Availability	Availability is the ratio of successful requests to total requests. A system is also considered unavailable if response latency is too high to be practically useful.	99.9% of requests succeed within a minute over a rolling quarter.
Latency	Latency is the time it takes to receive a response and is measured as a distribution and evaluated with statistics like average, median, 90th percentile, 95th percentile, and 99th percentile.	99% of responses were within 100ms over a rolling quarter.
Freshness	Freshness measures the staleness of a response, or the duration for which writes might not be reflected in a read, with the same statistical measures as latency. Freshness is measured for systems that aren't strongly consistent.	99% of responses had data with staleness under 60s over a rolling quarter.
Quality	Quality is a measure of the contents of a response, like the percentage of items returned in a set. Quality is measured for systems that can gracefully degrade, for example, by skipping results stored on servers that are currently unavailable.	99% of responses had 90% of the items over a rolling quarter.
Durability	Durability is the ratio of readable data to all data previously written.	99.999999999% of stored data is readable.

An SLO can also be described as an error budget and burn rate. For example, a service with an availability SLO of 99.9% over a quarter can be 100% down for about 2 hours or 10% down for about 21 hours. A service with an availability of 99.999% over a quarter can be 100% down for about 1 minute or 10% down for about 12 minutes. Humans typically cannot respond and mitigate faster than one hour, so budgets under an hour require an automated response. Figure 1-1 shows error budgets for various burn rates and SLOs.

		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

Figure 1-1. Error budgets for various error rates and SLOs over a quarter.

Choose SLO thresholds and windows based on user and business needs. Objectives that underperform stakeholder needs will make them unhappy or break obligations, whereas objectives that overperform will result in unnecessary engineering investment, more complex systems, and reduced flexibility to evolve the system. If you find that stakeholders complain even if the system is meeting SLO, then it's a good indicator that your SLO is too weak; if stakeholders don't complain when the system fails to meet SLO, then it's a good indicator your SLO is too strong. When a system fails to meet SLOs, expect to invest engineering resources to improve the system performance; if the system is meeting SLOs, then expect to save that investment. Avoid setting your objectives based solely on the system's current performance, a performance that may not satisfy your stakeholders, and one that may unnecessarily constrain the implementation to what's already built. For example, if you set your latency SLO based on current performance but faster than necessary, you may be unable to add new features that would increase latency and violate the SLO. If you later change the SLO, you may break assumptions that consumers had made and cause downstream SLO violations.

SLOs may be specialized per API, per method, or per feature if there are meaningful differences in user and business needs. For user-facing services, it's common to use different objectives for critical and optional functionality. For example, a blogging service may want a higher availability for reading blogs than for posting new entries. For infrastructure services, it's common to use different objectives for the data plane (the functionality involved in serving end-user requests like reading from a database) and for the control

plane (the functionality involved in administering the infrastructure like turning up a new replica in a new region). Figure 1-2 shows common SLOs based on layers in the tech stack and classes of functionality.

	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%

Figure 1-2. Common availability SLOs based on layer in the tech stack and class of functionality.

You can evaluate SLOs for the service as a whole or for each individual consumer. Evaluating the SLO for each consumer will prevent a poor experience in isolation, like 100% unavailability for their requests despite the service as a whole having sufficient availability. The more granular the evaluation, the more susceptible the statistic will be to noise and the more expensive the evaluation due to cardinality. For example, a service may have one million requests in a quarter with which to compute the 99th percentile latency, but for a single consumer who had one request, the 99th percentile becomes the same as the max latency for that consumer. Evaluating the SLO for each consumer may also put new constraints on the architecture. For example, evaluating the service as a whole may put constraints on general database availability, whereas evaluating the SLO for a single consumer may put constraints on availability of specific database rows. Similar to other SLO parameters, the choice of granularity should be based on user and business needs.

Publish SLOs to set performance expectations with users and to document what level of performance can be relied upon. If you don't publish the SLOs, users will set an implicit SLO based on their needs and observed performance, which may have consequences when the service fails to meet those implicit expectations. For example, users may become upset with the service and demand higher performance even though such was never an objective for the service, and may cause downstream SLO violations due to broken assumptions. Similarly, when designing your architecture, select and use dependencies based on their published SLOs so that the entire system performs as desired. For example, you may choose a less featureful

database that comes with higher availability SLOs so that the database does not limit your service's ability to meet SLOs. Later in this report, we'll learn to compose dependencies to increase aggregate reliability.

SLAs combine an SLO with a consequence should the SLO be violated, and are used between independent parties with a business relationship. For example, a cloud service provider may provide credits or refunds if virtual machine availability does not meet the objective documented in the agreement. SLAs can come with financial and reputational damage, so the objectives in SLAs tend to be more conservative than the equivalent internal SLO. For example, an SLA may give credits for failing to meet 99.9% availability even though the internal SLO targets 99.99% availability. For service owners, publishing a weaker objective in an SLA risks consumers developing an implicit SLO based on observed performance similar to as if no SLO or SLA were published. For consumers, receiving weaker objectives makes it difficult to leverage the service in other applications, which must in turn be designed to meet SLO targets. When SLAs are too weak to be theoretically useful, consumers must either demand stronger objectives from their providers, accept the risk that the dependency may underperform assumptions, or avoid the service in question altogether.

For more on SLIs, SLOs, and SLAs, see "Service Level Objectives" by Chris Jones et al. in *Site Reliability Engineering*.

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. To achieve availability objectives you must define failure domains and redundancy requirements to prevent failures from creating outages that are too large. Services must gracefully handle failures in hardware and software. Software failures can include new code bugs, misconfiguration, contention, or the inability to leverage additional resources to handle more load. Hardware failures in an on-prem datacenter can include utility power being cut to the datacenter, top of rack switches eliminating connectivity to hosts in the rack, hard drives rendering data inaccessible, CPUs corrupting requests and data, or request load exhausting available resources.

Define failure domains by ensuring there are no shared dependencies across the domains. The blast radius of a failure, or the things impacted by a failure, is thereby limited to the failure domain. For example, the blast radius of a CPU failure is a single host because multiple hosts don't depend on a single CPU. Mitigate failures of a domain by creating redundancy. For example, a service can be deployed across multiple hosts so that when a CPU fails, the other hosts can still host the service. For redundancy to be effective the replicas must be in separate failure domains and fail independently of each other. For example, having a second CPU on a host does not add redundancy because the failure of one CPU can affect the other CPU.

In a cloud environment, the hierarchy of failure domains for an application developer consists of virtual machines, hosts (physical machines), zones, and regions. Application developers should deploy services redundantly across these failure domains. When using an orchestration system like Kubernetes, the corresponding hierarchy of failure domains would be pods, nodes, clusters, and regions. Table 1-2 shows common hardware failure domains and opportunities to create redundancy.

Table 1-2. Common hardware failure domains and associated redundancy

Failures	Failure domain	Infrastructure redundancy (cloud responsibility)	Application redundancy (your responsibility)
CPU, RAM, disk	Host	Multiple hosts in a datacenter	Multiple servers and data replicas that can be distributed over multiple hosts
Top of rack switch, power rectifier, battery backup	Rack	Multiple racks in a datacenter	
Aggregation and spline switches, row transformer, diesel backup generator	Row	Multiple rows in a datacenter	
WAN routers, machine fire, cluster schedulers	Cluster	Multiple clusters within a datacenter / campus	Multiple servers and data replicas distributed over multiple zones
Exterior fiber optics, power utility, earthquake, hurricane, tornado	Datacenter / Campus	Multiple datacenters with geographic dispersion	Multiple servers and data replicas distributed over multiple regions

A failure within a domain can impact all components within that domain. Larger failure domains impact more components and can thereby create larger outages. For example, a failure of the rack's power supply will take down all hosts in the rack, whereas a failure of the datacenter's substation will take down all hosts in the datacenter. Similarly, a failure in a configuration loaded by every server will take down all servers, whereas a failure in a configuration loaded by only a single server will take down that server but not others.

Eliminate shared dependencies in larger failure domains by pushing those dependencies into the smaller failure domains, which have smaller blast radiuses and higher redundancy. For example, a configuration loaded from a global database is a dependency shared by all servers globally, which risks taking down all server instances globally. Instead, loading the configuration from parameters that can be set independently on each server ensures that updating the flags for one server doesn't impact others, thereby reducing the blast radius of the configuration. The intended state of the flags can be managed from one location, but rollout of a new flag value should be incremental and gradual by updating servers independently and staging the rollouts to gradually increase the percentage of hosts that have received the update. Figure 1-3 visually demonstrates creating smaller blast radiuses for a configuration.

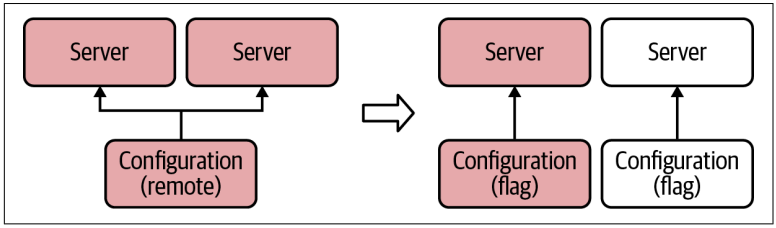

Figure 1-3. Eliminate shared dependencies to create smaller blast radiuses.

Minimize critical dependencies that can cause the service to be unavailable. Prefer soft dependencies that can be used when available and avoided otherwise, thereby ensuring the dependency does not impact the availability of the service. For example, a synchronous blocking remote procedure call (RPC) to a logging service would be a critical dependency, whereas an asynchronous nonblocking RPC, for which errors are monitored but otherwise ignored, would be a soft dependency. By switching to a soft dependency, the

availability of the service is no longer dependent on the state of the logging service.

Set the redundancy based on the number of instances necessary to host the service (`N`) and the number of failures the service should survive (`F`), or `N + F` total instances. It's typical to target surviving two failures (`F = 2`) so the service can survive one planned and one unplanned failure. A planned failure is done intentionally, like shutting down a host to upgrade the RAM. An unplanned failure is one that is unexpected, like the disk failing. For services that can operate and upgrade without planned failures, it may be sufficient to reduce redundancy to handle one failure to save the cost of an instance. Monitor the health of instances so that failed instances can be repaired or replaced to bring the service back to target redundancy.

You can reduce the overhead and redundancy by varying the number and size of instances. If one full-sized server would be 75% utilized when serving all traffic and you want to survive two failures, then you'll need three servers. When all servers are operational, each server will be 25% utilized with 75% of each server idle. Upon failure, the load from the failed servers will move to the healthy server, resulting in a single server that is 75% utilized. Figure 1-4 shows how load is redistributed after a server failure. If instead you used four quarter-sized servers and two additional quarter-sized servers to handle failures, then each server will be 50% utilized during normal operation and 75% utilized after two failures. By going from `N = 1` to `N = 4`, we've reduced the overhead of `N + 2` from 66.7% to 33.3%. Figure 1-5 shows how overhead varies for different redundancies and number of instances. Note that too many instances can be more expensive if reducing instance size impacts efficiency due to instance overhead and the cost of instance coordination.

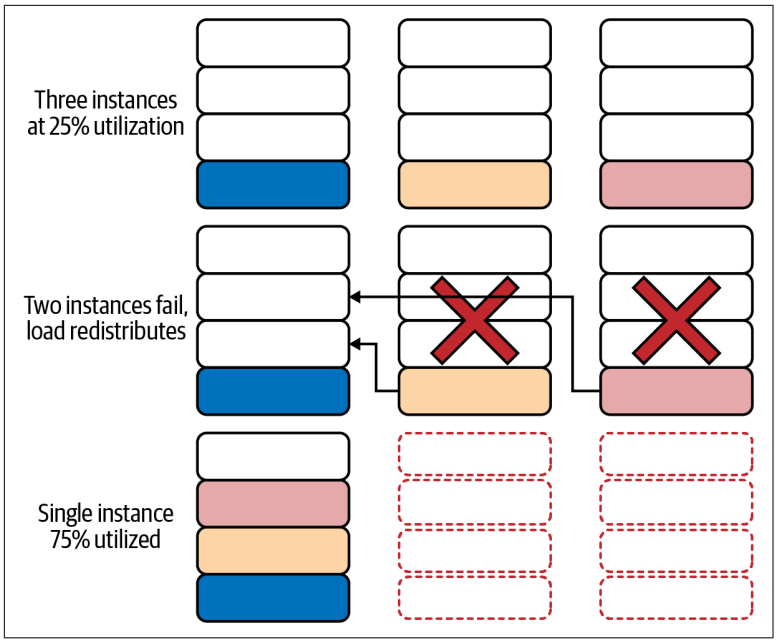

Figure 1-4. When failure occurs, load is shifted from failed to healthy instances. If there are three instances at 25% utilization each, after two failures the remaining instance will be 75% utilized.

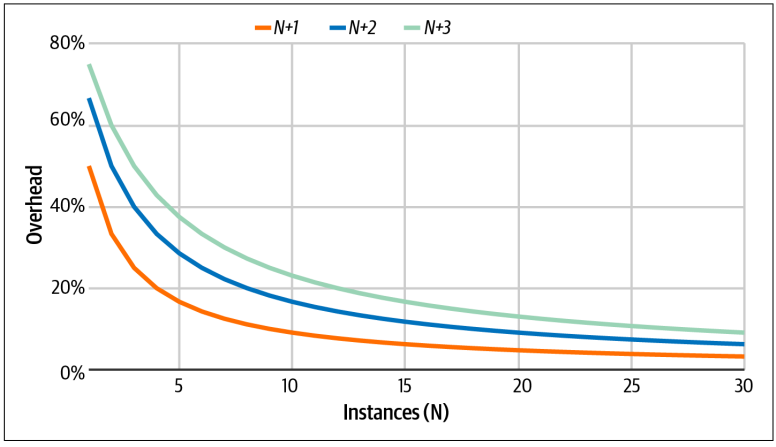

Figure 1-5. Overhead versus instances for varying levels of redundancy. Increasing the number of instances while reducing the size of an instance decreases the overhead of redundancy.

Where possible, minimize dependencies shared between servers, and minimize dependencies shared between regions. Prefer to have a dependency between servers within a region, rather than a dependency across regions. Redundancy for servers and regions is typically set at `N + 2`, unless there are no planned failures and there are significant cost savings with `N + 1`.

Scalability and Efficiency

As usage of a service grows, the service must grow in size to accommodate the additional load. That is, as queries per second and bytes stored grow, the service should be able to leverage additional computational and storage resources like CPU and disks respectively, and the service should have a process for introducing new resources. If a service fails to scale, the service may become overloaded and unhealthy, start serving errors, and ultimately not achieve SLOs.

Additional servers can make a service more reliable, but with profitability comes the opposing force of efficiency. Without considering profitability, a service could simply provision infinitely to eliminate the problem of overload. However, most services are subject to business profitability objectives, so services must be efficient with resources. As a service becomes more expensive, businesses typically increase the target utilization to reduce the cost of wasted resources.

The algorithms, data structures, and architectures used should scale at most linearly, or `O(N)`, with service usage. Prefer solutions that scale sublinearly, like `O(1)` or `O(logN)`. Polynomial solutions, or `O(N^a)` for `a>1`, will quickly become prohibitively expensive and fail to scale. For example, an `O(N^2)` operation that takes 100 milliseconds for 10 users will take over 16 minutes with 1,000 users, and will take over 30 years with 1 million users. Solutions that are `O(NlogN)` may scale but may be expensive or underperform, so those operations are typically restricted to control plane or noncritical path operations. Figures 1-6 and 1-7 compare the performance of different algorithms as data or the number of users scale.

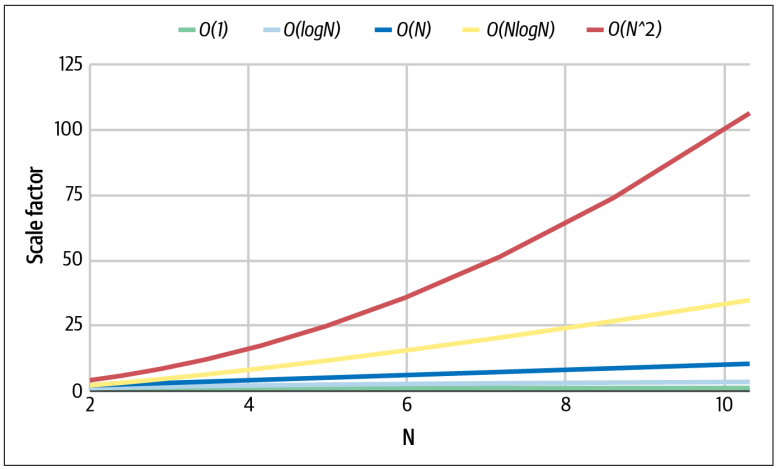

Figure 1-6. Scaling factor for cost or performance for different asymptotic complexities.

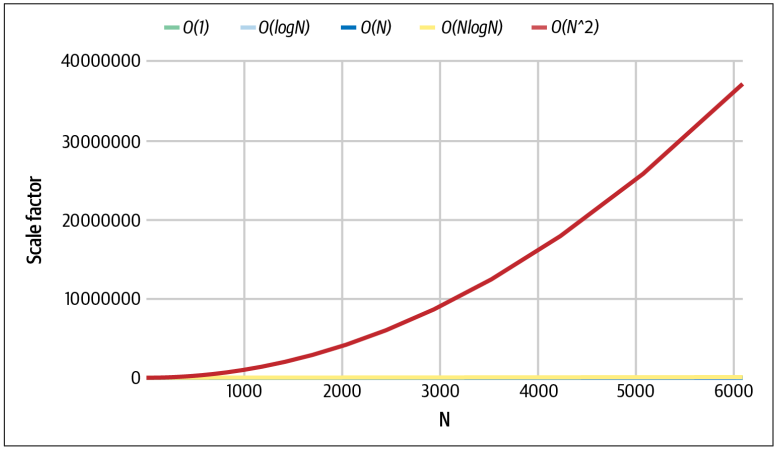

Figure 1-7. `O(N^2)` scaling quickly underperforms and becomes prohibitive.

Services should support both vertical and horizontal scaling to leverage additional resources and handle more load. Vertical scaling involves making existing components larger, which tends to improve or maintain efficiency, but fails to scale beyond a practical limit like max host or disk size. Horizontal scaling involves adding additional instances of a component without changing the size of a component, which tends to reach much larger scales, but can reduce efficiency due to increased overhead and coordination. The

Kubernetes Horizontal Pod Autoscaler is an example of horizontal scaling. Figure 1-8 shows visually the difference between vertical and horizontal scaling. With scaling as an objective, service design must consider details such as how a server can leverage additional CPUs, how work can be distributed across servers, and how state can be synchronized. Services must also avoid bottlenecks to scaling like contention. We'll dive deeper on scaling in "Horizontal and Vertical Scaling" on page 43.

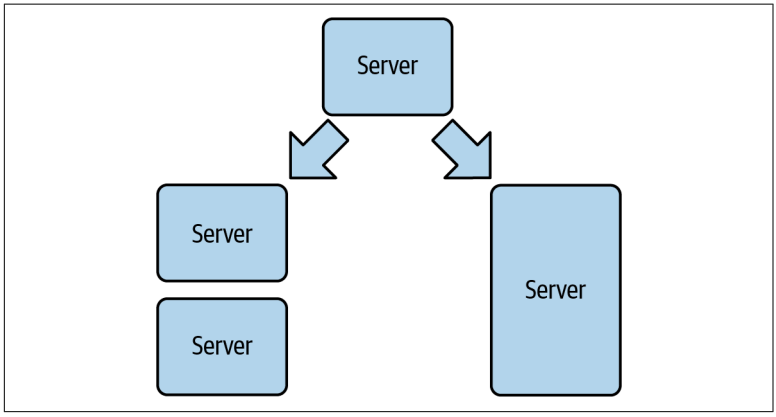

Figure 1-8. Services can leverage more resources by scaling horizontally by adding additional server instances, or by scaling vertically by creating bigger servers.

Consider both utilization and the cost of work. Utilization measures the percentage of allocated resources actively being used to do useful work. For example, if a service is provisioned with 10 CPUs but only uses 1 CPU to serve traffic then the service has 10% utilization. If the service was resized to use 2 CPUs then utilization would increase to 50% and cost would be reduced by 80%. Work cost is the resources necessary to complete a logical unit of work like responding to an HTTP request. Work cost can be measured through metrics like requests per second per CPU. Services may be able to reduce work cost through techniques like caching. Services typically target a utilization between 50% and 80%, whereas work cost targets are domain specific. Services with large periods of inactivity may leverage scale-to-zero, or shutting down all server instances, to eliminate cost during inactivity to achieve efficiency objectives.

Evolution and Velocity

Service requirements change over time due to changing user and business needs, and services must evolve to satisfy these new requirements or be rendered obsolete. Services must also evolve rapidly so that users are quickly satisfied and the service remains competitive. Design services to safely evolve at high velocity by creating environments, continuously deploying changes, updating production gradually, and designing for forward and backward compatibility.

Services typically target hourly deployments to a development environment, daily deployments to a staging environment, and weekly deployments to a production environment as part of an agile model of development where improvements are regularly delivered into production. Integrate and deploy components of a feature continuously as they are developed to minimize merge conflicts and the size of any change deployed. To maintain productivity, teams must invest in automated testing, qualification, and deployment. For detailed guidance, see "DevOps Tech: Continuous Delivery".

For safety, you must gradually deploy changes to the production environment so that problems can be detected and mitigated prior to becoming a full service outage; therefore, services should roll out a component change gradually over a week. To deploy gradually over a week while maintaining a weekly cadence, component changes are rolled out in parallel. Changes to different components must be decoupled, and individual components must maintain backward and forward compatibility between adjacent releases.

Evolution across services or with user involvement tends to be more difficult and much slower. Services typically assume breaking changes must maintain backward compatibility for years so that users can upgrade in time without disruption. For changes that don't require user work and just require redeployment, services typically assume new versions will be deployed within months. Due to the slow velocity, developers tend to be more intentional with externally visible API and feature changes, performing deeper review, and being conservative where possible. Due to the difficulty and cost of breaking changes, developers tend to design with backward and forward compatibility in mind.

Recap

In this chapter, we've defined the objectives for a service:

Service level indicators (SLIs), objectives (SLOs), agreements (SLAs)

- Measure, store, and analyze metrics for availability, latency, freshness, quality, and durability. Measure metrics both at the client side and at the server side.
- Choose SLO thresholds and windows based on user and business needs, not based on current system performance.
- Publish SLOs to set expectations with users and consumers.
- Use an SLA between independent parties with a business relationship to set expectations and the consequences for underperformance.

Failure domains and redundancy

- A failure domain is a group of resources that can fail as a unit.
- Create redundancy by leveraging multiple servers, zones, and regions.
- Eliminate shared dependencies in larger failure domains by pushing them into smaller domains with smaller blast radiuses and higher redundancy.
- Minimize critical dependencies and prefer soft dependencies.
- Set the redundancy based on the number of instances necessary to host the service (`N`) and the number of failures the service should survive (`F`), or `N + F` total instances.

Scalability and efficiency

- Algorithms, data structures, and architectures should scale at most linearly `O(N)`. Prefer solutions which scale sublinearly, like `O(1)` or `O(logN)`.
- Services should support both scaling vertically (larger servers) and horizontally (more servers).

Evolution and velocity

- Target hourly deployments to a development environment, daily deployments to a staging environment, and weekly deployments to a production environment.
- Integrate and continuously deploy components of a feature as they are developed.

- Gradually deploy changes to the production environment over a week.
- Ensure components have backward and forward compatibility between adjacent releases.
- Assume breaking API changes must maintain backward compatibility for years.

Next, we'll explore the building blocks we'll leverage to build a service that meets those objectives.

CHAPTER 2

Know Your Dependencies

Knowing your dependencies will ensure that you can select and leverage building blocks to build a system that can meet your objectives. Dependencies can vary wildly in SLOs, failure domains, flexibility, maintenance cost, and appropriate use cases. In this chapter, we introduce common considerations for when you're selecting and using dependencies like management, regionality, SLOs, and trade-offs for compute, networking, storage, and administration.

IaaS, PaaS, SaaS

Infrastructure as a service (IaaS) enables customers to use raw resources like virtual machines and durable block storage. Platform as a service (PaaS) provides additional functionality like automatic management of virtual machine groups. Software as a service (SaaS) offers an API to customers to provide services without requiring the customer to manage infrastructure. Cloud service providers may use different definitions and may not use them consistently, and a given service may not cleanly fall into one of the classes, but we'll use the preceding definitions in this section to discuss the general principles.

PaaS can save substantial effort solving common problems. For example, virtual machine groups can automatically add servers and remove servers based on resource utilization, preventing you from needing to manually adjust resources as service traffic changes. SaaS goes further and ensures that there is no infrastructure to manage, just an API to use. You don't need to worry when traffic doubles in

minutes because the SaaS service is responsible for scaling up to handle that traffic. While SaaS saves up-front engineering and operational costs, there is added risk that the dependency changes or fails to evolve, requiring a potentially disruptive migration away from the dependency. Use SaaS where possible, as the service provider can leverage economies of scale to provide more functionality at a cheaper total cost of ownership. SaaS also enables you to focus on your core problems by delegating common problems to a cloud provider, but you must understand the potential risks. Table 2-1 outlines the pros and cons of these services.

Table 2-1. Comparison of IaaS, PaaS, and SaaS services

	Examples	Pros	Cons
IaaS	Virtual machines Durable block storage	Ultimate flexibility Minimal lock-in	Highest engineering cost to develop and maintain
PaaS	Serverless Managed SQL databases	Reduces effort to solve common problems	Engineering cost to develop and maintain Less flexible and portable
SaaS	Key management systems Secret managers	No infrastructure to manage	Lock-in risk for pricing, deprecations, and evolution

Avoid premature optimization to build multicloud and cloud provider migrations ahead of a concrete need. Developing these capabilities at the start will add substantial up-front engineering and operation cost because you're forced to rebuild all functionality not part of the common denominator, and platform differences will add complexity to your service. Migrations are also unlikely to be seamless even if you plan ahead. Focusing on a single platform will enable you to leverage more advanced capabilities, like SaaS services, while minimizing your service's complexity and cost of development. If multicloud or on-premise is required, consider platforms which enable hybrid cloud while providing a unified developer experience.

Regionality

Cloud services typically provide either zonal, regional, multiregional, or global failure domains. A zonal service is physically isolated from other zones, but may not be geographically isolated. For example, a single building can house a zone, and multiple buildings on a campus can make up a region. A regional service is geographically isolated from other regions. A multiregional or global service

does not typically provide isolation guarantees. Figure 2-1 shows the hierarchy and associated blast radiuses, and Figure 2-2 shows the geographic distribution of zones.

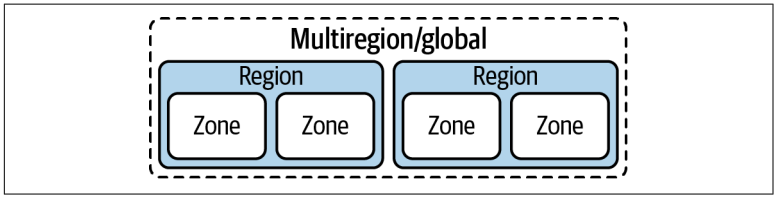

Figure 2-1. Global is composed of multiple regions, and a region is composed of multiple zones. Regions are geographically distributed, whereas zones in a region may not be.

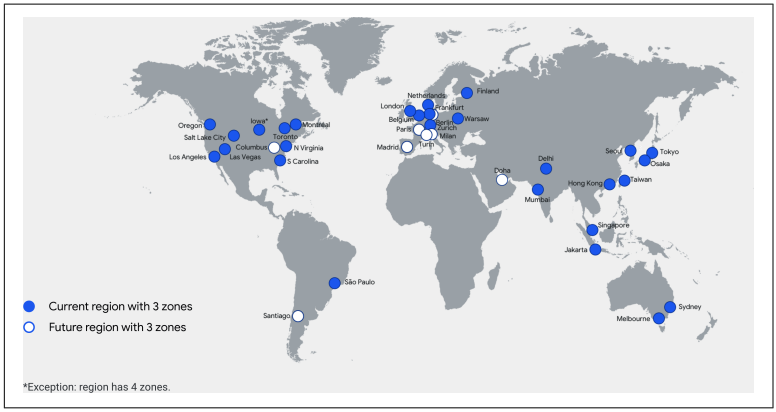

Figure 2-2. Regions in the Google Cloud Platform, October 2021. A global service will have presence and replicate data to these regions. Within every region are multiple independent zones.

A global service will have higher availability than a regional service with a single region, and a regional service will have higher availability than a zonal service with a single zone. However, failure domain isolation can enable you to leverage multiple regions of a regional service to get higher availability than a global service. While you may be able to achieve higher availability building upon isolated regions rather than a global service, your service will need to handle the multiregional complexities like data replication.

Align the failure domains of your dependencies with the desired failure domains of your system. For multiregional and global dependencies, consider if the dependency has both regional and multiregional failure domains, and how each domain can be aligned with your system. The following architecture diagrams depict how to align the failure domains of a database dependency with the rest of the application.

Figure 2-3 shows the simplest of deployments with independent regional servers and a single global SQL database. In this example, if the top region fails, then the user can no longer access the application because all regions have a dependency on a single point of failure, which only has presence in the failed region.

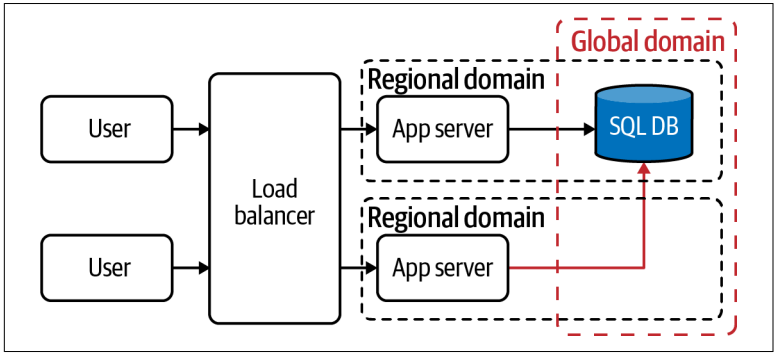

Figure 2-3. Architecture with a regional application and unaligned global storage.

Figure 2-4 shows a more reliable deployment with independent regional stacks that share a global dataset replicated through the Paxos or Raft consensus algorithms. In this example, if a region fails then the users can still access the application because there is no serving-path dependency on the failed region. However, a bad database schema, which is defined globally, could create a global outage.

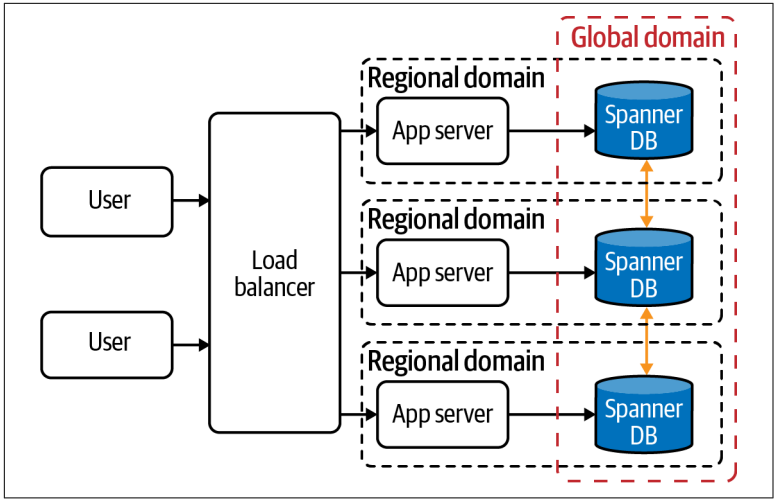

Figure 2-4. Architecture with a regional application and storage, and a global configuration.

Figure 2-5 shows the most reliable and complex deployment with independent regional stacks that are asynchronously replicated using decoupled schemas. In this example, if a region fails then the users can still access the application because there is no serving-path dependency on the failed region. Similarly, database schemas are no longer globally defined so a bad schema cannot create a multiregional failure. However, there is additional complexity and a different user experience now that the application is eventually consistent.

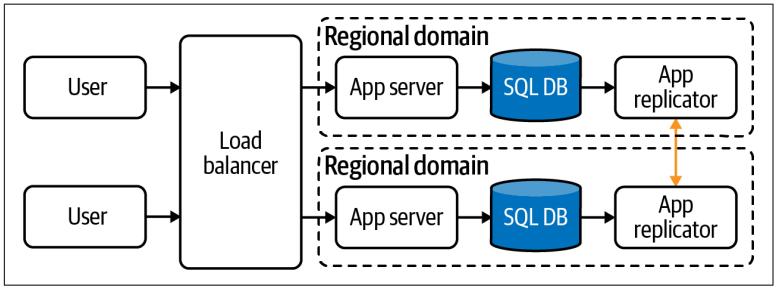

Figure 2-5. Architecture with regional alignment and asynchronous replication across regions.

For more on regional, multiregional, and global architecture alignment, see "Deployment Archetypes for Cloud Applications" by Anna Berenberg and Brad Calder.

Composed SLOs and SLAs

Select and use dependencies so that the aggregate availability, including the error budget for your application layer, can still meet your service's SLOs. How your service uses a dependency will determine whether it improves or weakens the aggregate availability of your service. To compute aggregate availability, you'll need to assume independent failure domains so that SLOs are independent and can be computed with simple probabilities.

Intersection availability is where multiple dependencies must be available. For example, an application that depends both on a login service and a database to serve a request requires both dependencies to be available. Intersection availability is lower than the availability of any single dependency. Three dependencies with 99.9% availability SLOs will have intersection availability of $0.999^3 = 99.7\%$.

Union availability is where at least one of the dependencies must be available. For example, an application that is available so long as one of the regions hosting the application is available only requires one dependency to be available. Union availability is higher than the availability of any single dependency. Three dependencies with 99.9% availability SLOs will have union availability of $1 - (1 - 0.999)^3 = 99.9999999\%$. Table 2-2 outlines some of the various computation for aggregate availability.

Table 2-2. Aggregate availability based on availability of a dependency, number of dependencies, and whether the aggregate requires union or intersection availability

	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%	99.99999999999999%

Dependencies may also have data planes and control planes with varying SLOs. The data plane consists of functionality for normal operation of the service, whereas the control plane consists of functionality necessary to administer the service. The data plane of your service should only depend on the data plane of a dependency,

whereas the control plane of your service can depend on either the control plane or data plane of a dependency. For example, a book application can read a database row in order to retrieve a book, but retrieving a book should not depend on changing a database password. However, infrastructure for deploying the book application to a new region could depend on setting a database password. Figure 2-6 shows appropriate dependencies for data and control planes.

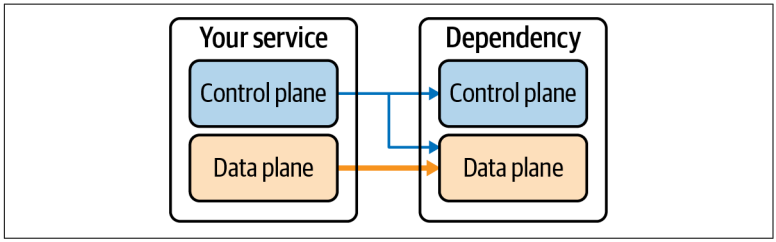

Figure 2-6. Your service's data plane should only use the data planes of dependencies. Your service's control plane can use both the control and data planes of dependencies.

Compute, Networking, Storage

Select compute, networking, and storage dependencies so they contribute to the overall performance, reliability, and efficiency of the service. For example, managed compute platforms can solve common problems like overload, dedicated inter-datacenter bandwidth can reduce exposure to network congestion, and SaaS storage services can scale to more data and traffic while providing backup and recovery. These choices may also come with trade-offs like flexibility and portability.

Virtual machines are complete computing environments including virtual hardware, operating systems, and applications that are multiplexed onto a physical host by a hypervisor. Containers package applications and their dependencies to run in an isolated process space and filesystem while sharing an OS kernel. Kubernetes enables you to schedule containers over a fleet of hosts and provides management functionality like autoscaling of servers, health checking and restarting of servers, and service discoverability. Serverless platforms enable you to leverage containers without having to worry about the underlying infrastructure. For example, serverless platforms handle node management tasks like sizing nodes, adding

additional nodes, and managing bin-packing of workloads onto nodes. Where appropriate, prefer serverless over Kubernetes over virtual machines in order to reduce development and maintenance cost while improving reliability and security. While Kubernetes was inspired by Borg, Google's internal computing environment, Borg has a user experience more closely aligned with serverless platforms like Knative and Cloud Run. Table 2-3 compares the different compute environments. For more guidance on selecting a compute platform, see "Choosing the Right Compute Option in GCP: A Decision Tree" by Terrence Ryan and Adam Glick.

Table 2-3. Comparison of compute environments

	Virtual machines	Kubernetes	Serverless
Pros	Flexibility Compatibility	Container scheduling Discoverability	No infra to manage Advanced automation
Cons	OS management VM scheduling	Cluster management Tenancy management	Reduced feature set Less portable

Containerized platforms like Kubernetes and Cloud Run can improve reliability of your service when used optimally. You should use techniques like:

- Making images smaller for faster autoscaling by using minimalistic base images like Alpine, using build stages to remove dependencies needed at build time but not at runtime, and using compiled languages, which have fewer dependencies
- Optimizing container startup for faster autoscaling by initializing and loading dependencies in parallel
- Implementing readiness checks and liveness checks so that the platform can detect and mitigate unhealthy containers
- Terminating gracefully by finishing inflight requests, saving state, and releasing dependencies so that the platform can scale down the service and migrate containers to healthier hosts

Cloud services provide functionality to improve your service's network performance. Private wide-area networks can provide you stronger guarantees for reliability, latency, and throughput than the public internet. Edge load balancers can terminate network connections closer to the user, reducing connection setup round-trip time and getting data onto private networks as soon as possible. Content delivery networks (CDNs) can cache data closer to the user,

reducing network bandwidth requirements and data transfer latency. Inter- and intra-datacenter load balancers can reduce latency and minimize overhead through load balancing. Support for protocols like HTTP/2 improves connection handling through request and response multiplexing, reduced round trips for setup, stream prioritization, and header compression. Support for RPC frameworks like gRPC improves efficiency through encoding, and type checking improves developer productivity and reduces bugs.

Service mesh frameworks improve service-to-service communication by enforcing security, managing traffic distribution, implementing resilience practices, creating observability, and implementing best practices through configuration of the mesh rather than writing code. These frameworks can reduce your effort developing, maintaining, and enhancing systems requiring features like:

Security
- Identity and certificate management
- Authentication of client and server
- Authorization of clients to access APIs and resources
- Encryption of traffic

Traffic management
- Load balancing
- Change management with gradual rollouts

Reliability
- Request deadlines
- Request retries on errors
- Fault injection for testing resilience

Observability
- Metric collection and dashboard generation
- Request tracing across the distributed system
- Log generation and auditing

Object storage typically provides nontransactional reads and writes, with eventual or causal consistency, as a SaaS offering. Object storage is best suited for serving static assets like JavaScript and CSS, serving large binary objects like photos, or archiving data for

analytics or recovery. Relational databases typically provide atomicity, consistency, isolation, and durability (ACID) with SQL language semantics. Relational databases typically have a single primary instance, which serves all writes and strongly consistent reads, with backup instances that can serve stale reads and can be promoted to primary during a failure. Depending on the cause of failure, failover to a backup instance may be manual and take significant time to complete, with the application suffering downtime in the interim. NoSQL databases like Bigtable provide unstructured storage, transactional semantics for a row at a single replica, eventual consistency across rows and replicas, and the ability to scale to many servers and regions. Spanner provides scalability and availability similar to NoSQL databases, transactions and a SQL interface similar to relational databases, and global external consistency.

Caches enable services to scale reads by leveraging more servers, to reduce latency by fetching results from fast storage (e.g., RAM) rather than computing the results from scratch or retrieving from slower storage (e.g., HDD), and to reduce cost by avoiding recomputing results or retrieving data from more expensive storage (e.g., HDD spindles). Local caches like an in-memory hashmap can be simple to add while avoiding the cost and latency of transport serialization and over-retrieval of data due to key granularity. Remote caches like Memcached or Redis can improve cache hit rate by sharding data across servers so caches need only be populated once for a given key. Servers with local caches can obtain similar hit rates to remote caches by having the clients route requests to specific servers based on a similar sharding strategy, though this increases coupling to the clients and may make server autoscaling slower due to state management.

Publish–subscribe messaging services are a class of systems for storing and distributing messages with at-least-once delivery. This type of messaging enables multiple publishers to send messages to a topic and multiple subscribers to receive messages on the topic, decoupling senders from receivers. Publish–subscribe messaging decouples workflows and is a building block for reliable asynchronous execution.

Google's MapReduce and Flume are batch and stream processing systems designed to process large volumes of data reliably and with high throughput. MapReduce provides a functional API for defining data transforms, whereas Flume goes further to enable higher order

operations and optimal chaining of operations. Cloud service providers typically provide managed forms of Apache's Hadoop and Beam, open source implementations of the MapReduce and Flume designs. Dremel and Google Cloud Platform's BigQuery product is another mechanism for processing data through columnar data formats and SQL query interfaces.

Administrative Services

Cloud service providers have administrative services that are necessary for setup and maintenance of applications that run on the cloud, and it is easy to overlook these services as a source of failures. To achieve availability objectives you must configure these administrative services so that they don't introduce a failure domain that can create a large outage. Most applications on the cloud will interact with a resource management service, an identity and access management service, and a key management service.

Cloud service providers typically have a resource management service for organizing cloud resources into a hierarchy like projects/accounts, folders/groups, and organizations. Services and components within a project are part of the same failure domain due to the shared dependency on common settings, like API enablement, and resource constraints, like service quotas. For example, all workloads within a project will share a single quota for a given API provided by the cloud service, so one misbehaving workload can cause quota rejections and subsequent outages for all workloads within the project. Separate the services, environments (dev, staging, prod), and components into separate projects to create independent failure domains.

Cloud service providers typically have an identity and access management (IAM) service for defining identities and assigning permissions. This configuration represents a shared dependency as well. For example, a single IAM binding that grants servers access to the database is a single configuration that can be broken, creating a simultaneous global outage. If your service needs the highest availability, use granular identities and permissions where possible to create independent failure domains.

Cloud service providers typically have a key management service (KMS) for defining keys for symmetric and asymmetric cryptographic operations. Services typically allow customers to set the KMS key that will be used to encrypt data. Data encrypted by the same KMS key thus has a shared dependency on the key material. If your service needs the highest availability, use separate keys where possible to create independent failure domains.

Recap

In this chapter, we've reviewed the building blocks that are available on the cloud:

Infrastructure as a service (IaaS), platform (PaaS), software (SaaS)

- Infrastructure as a service provides raw resources like virtual machines.
- Platform as a service solves common problems and reduces overhead.
- Software as a service provides an API with no infrastructure to manage.
- Prefer SaaS to PaaS to IaaS, unless the service doesn't meet your requirements.

Regionality

- Zones provide physical isolation, whereas regions provide geographic isolation.
- Leverage multiregion and global services to make the cloud service provider responsible for multiregion complexities like data replication.
- Align failure domains of your dependencies with the desired failure domains of your system.

Composed SLOs and SLAs

- Select and use dependencies so that the aggregate availability, including the error budget for your application layer, can still meet your service's SLOs.
- Intersection availability is where multiple dependencies must be available, and in aggregate has lower availability than that of any single dependency.

- Union availability is where at least one of the dependencies must be available, and in aggregate has higher availability than that of any single dependency.

Compute, networking, storage

- Improve reliability through containerized platforms like Kubernetes and Cloud Run. Make container images smaller, optimize startup time, implement readiness and liveness checks, and terminate gracefully.
- Use serverless to eliminate infrastructure management and scale to zero.
- Leverage your cloud service provider's network features like private WANs, edge load balancers, content delivery networks, and support for HTTP/2 and RPC frameworks.
- Leverage service mesh frameworks to improve service-to-service communication.
- Leverage your cloud service provider's storage like object storage, relational databases, NoSQL databases, and databases like Spanner.
- Leverage caches to scale reads, reduce latency, and to reduce cost.
- Leverage publish–subscribe services to decouple senders from receivers, and to increase reliability through durable long-duration retries.
- Leverage frameworks like MapReduce and Flume for processing large volumes of data reliably and with high throughput.

Administrative services

- Configure services like resource management (projects, folders, organizations), identity and access management (IAM), and key management (KMS) so that their failure domains align with that of your application.

Next we'll learn how to architect a service using these building blocks.

CHAPTER 3
Architect Your Service

Architecting your service involves decomposing a larger system into smaller components, each with well-defined interfaces and coupling, so that in aggregate the system behaves and performs as desired.

Follow these guidelines to architect a reliable system on the cloud:

- Define application programming interfaces (APIs) to structure the coupling between components while creating an abstraction from implementation details.
- Deploy a service-oriented architecture to create a product from a collection of services, and deploy microservices architectures to create individual services.
- Determine whether operations are synchronous, asynchronous, or batch based on desired simplicity, responsiveness, coupling, reliability, and cost efficiency.
- Design how each service will scale horizontally and vertically to accommodate more traffic and data by leveraging additional computational resources.
- Leverage frameworks and platforms to more efficiently develop services across the organization while making it easier to adopt best practices.

In this chapter, we'll explore each of these steps in depth to architect a reliable service.

Application Programming Interfaces (APIs)

APIs define the coupling between components like a client and server. They define a contract between components to ensure their interactions work as intended. API standardization improves the usability and stability of the interface, and reduces the cognitive load for developers who leverage many APIs.

APIs should follow a resource-oriented design pattern to leverage repeatable reliability patterns, developer mindshare and familiarity, and proven ability to be integrated across systems at scale. An API should provide resources (nouns like a book), relationships and hierarchies between resources, the schema of each resource, and methods (verbs like create) acting upon those resources. Resources should be modeled in a hierarchy, where each node is either a resource or a collection. Standard methods over a resource include creating (`HTTP POST`), reading (`HTTP GET`), updating (`HTTP PUT`), and deleting (`HTTP DELETE`), typically referred to as the CRUD methods. Standard methods over a collection include listing (`HTTP GET`). APIs should have a stateless protocol where the server is responsible for persisting data and a request made by a client is independent of other requests. Table 3-1 provides examples of a resource-oriented API for a book application.

Table 3-1. Example resources and methods in an API

Resources	Methods
`publishers` Collection of publishers	`GET /publishers/foo/books` Retrieve a list of books published by `foo`
`publishers/foo` Publisher named `foo`	`POST /publishers/foo/books?name=bar` Create a new book named `bar`
`publishers/foo/books` Collection of books by `foo`	`GET /publishers/foo/books/bar` Retrieve the details for the book named `bar`
`publishers/foo/books/bar` Book named `bar` published by `foo`	`PUT /publishers/foo/books/bar?page_count=10` Update the book named `bar` with the count of pages
`name="bar", page_count=10` Fields in a book resource	`DELETE /publishers/foo/books/bar` Deletes the book named `bar`

In create and update methods, your API should return the resource in the response so the client has access to the updated output-only fields. List methods that return resources in a collection should support pagination through a total item count and a pagination token used by the server to efficiently resume, to bound the cost of a single request so it doesn't overload the client or the server. List methods may also support ordering and filtering to perform these operations more efficiently at the server before being transferred to the client. APIs may also support custom methods that express actions that better adhere to user intent than any of the standard methods.

APIs must handle failures, retries, and concurrency so that the API does not limit the reliability of clients. Read methods should be nonmutating and should not have side effects. Create methods should support idempotency by having a request identifier parameter, which can be used by the server to detect and merge duplicate requests, a situation that may occur when a client doesn't receive a response and subsequently retries. Update methods should support optimistic concurrency control by having a request ETag parameter, which can be used by the server to detect conflicts in read-modify-write flows across API calls.

Methods that take a significant amount of time to complete, typically more than 10 seconds, should return a long-running operation (LRO), which is similar to a promise or future in a programming language. LROs are represented as resources that can be polled, or repeatedly read, to monitor the state of the operation. Figure 3-1 shows a sample LRO sequence of operations. LROs may support update or delete methods while running to modify in-flight execution. Services may or may not support concurrent LROs. Upon completion, LROs typically expire after 30 days after which the resource is deleted. When a method may need to run repeatedly, or when the permission to edit configuration should be separated from the permission to run a method, a job resource can be used for setting configuration and a run method to initiate the LRO.

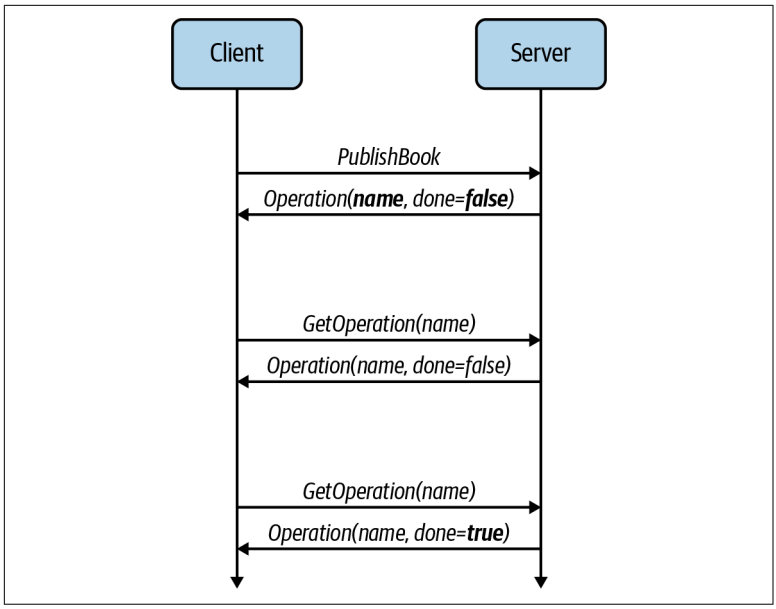

Figure 3-1. Example long-running operation used to publish a book. The `PublishBook` method returns the `Operation`, and the `GetOperation` method calls eventually return that it's done.

A strongly consistent API ensures that requests to read data will contain the effect of all prior writes. An eventually consistent API is one where the effect of writes may not be immediately in future reads, but at some point in the future, the write will be reflected in a read. A causally consistent API will ensure that all writes up to a specific time will be reflected in a read, coordinated via a consistency token that is returned during writes and provided during reads, so that logically related operations can see a causal ordering of operations. Strongly consistent APIs are significantly easier to use and result in fewer bugs, whereas eventually consistent APIs tend to be more performant and scalable due to reduced need for server coordination and asynchronous execution. Causally consistent APIs can perform as well as eventually consistent APIs, but require extra coordination to receive and provide consistency tokens. Strongly consistent APIs tend to be more reliable than eventually consistent APIs by preventing bugs that are difficult to detect and resolve.

APIs must return consistent error codes and messages so that clients can react to the error appropriately. For example, a `RESOURCE_EXHAUSTED` error indicates requests should be sent slower because the service is overloaded, whereas an `ABORTED` error indicates the request should be retried to mitigate a transaction conflict. Table 3-2 describes common error codes that your clients and servers should support to handle common scenarios.

Table 3-2. Common error codes and their meanings

gRPC status code	HTTP code	Description
`OK`	200 - OK	Success.
`INVALID_ARGUMENT`	400 - Bad request	Client specified an invalid argument.
`FAILED_PRECONDITION`	400 - Bad request	Operation rejected because the system was not in a state required to execute it.
`UNAUTHENTICATED`	401 - Unauthorized	Request does not have valid authentication credentials.
`PERMISSION_DENIED`	403 - Forbidden	Client does not have permission.
`NOT_FOUND`	404 - Not found	Requested resource was not found or not authorized for the entire class of users.
`ABORTED`	409 - Conflict	Operation aborted, typically due to concurrency issues.
`ALREADY_EXISTS`	409 - Conflict	Client attempted to create a resource that already exists.
`RESOURCE_EXHAUSTED`	429 - Too many requests	Resources have been exhausted, like a per-user quota, so the request was rejected.
`CANCELED`	499 - Client closed request	Operation canceled, typically by the caller.
`INTERNAL`	500 - Internal server error	Service is unavailable, potentially transient. May indicate a bug or problem.
`UNAVAILABLE`	503 - Service unavailable	Service is unavailable, likely transient and can be mitigated by retrying with backoff.
`DEADLINE_EXCEEDED`	504 - Gateway timeout	Operation could not be completed within specified time.

API components have different stability levels based on the life cycle of an API. An alpha component is one that is undergoing rapid iteration, where breaking changes are allowed and expected and the users of the component are curated and intentionally kept small. A beta component is one that is feature complete and ready to be declared stable subject to public testing, where breaking changes are

allowed but unexpected and small or infrequent. A beta component is expected to advance to a stable component if no issues are found, typically within a reasonable time period like 90 days. A stable component must be supported for the lifetime of a major API version, a duration that is defined when the API is marked stable. Stable APIs cannot receive breaking changes without releasing a new major version, after which a predefined deprecation clock is started on the prior version. Instead of alpha, beta, and stable, sometimes APIs instead use the names private preview, public preview, and generally available (GA).

For more on APIs and standardization, see the API Improvement Proposals (AIPs) documentation.

Tiered, Service-Oriented, Microservices

Separate a service into smaller services coupled by APIs so that each service can approach development, scaling, and failure independently. Services frequently undergo an evolution through various architectures like two-tier, three-tier, service-oriented, and microservices architectures.

A two-tier architecture typically consists of a frontend server and a backend database. A two-tier architecture is the fastest to set up, quickest to market, and has low deployment maintenance overhead. However, the application quickly becomes highly coupled, which forces all components to evolve together, making it harder and more time consuming to evolve. To improve velocity and maintenance costs, components of the application are broken into separate services, thereby developing a three-tier architecture, which typically consists of a frontend server, a series of product infrastructure services like a user service, and a backend database layer. For single-product organizations, a three-tier architecture tends to work for a long time. Figure 3-2 shows examples of two- and three-tier architectures for a book application. As additional teams start contributing and new products are developed, the architecture may need to evolve again.

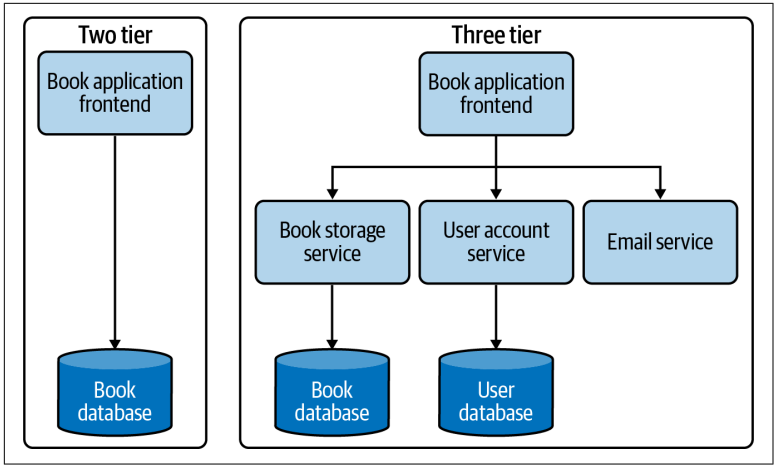

Figure 3-2. Two- and three-tier architectures for a book reading application.

As the number of teams increase and the product and feature set expand, organizations typically move to a service-oriented architecture. A service-oriented architecture is one where each business function develops a separate service with its own API, where services are loosely coupled only through those APIs. Figure 3-3 shows an example of a service-oriented architecture for a large organization that publishes books. After APIs are established, the services can be developed, maintained, operated, and evolved independently. Team coordination is reduced to details of the API, a much narrower surface than the implementation details of the services themselves. Individual services are typically owned by a single development team, with cross-functional teams involved as necessary. A single development team may own multiple services depending on the size of the team and the services.

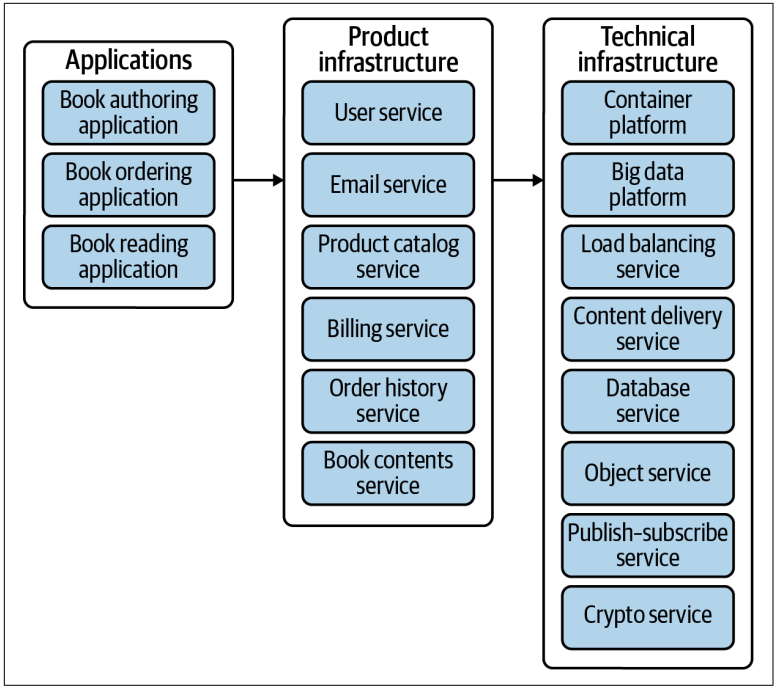

Figure 3-3. Service-oriented architecture with services coupled only through APIs. Each service is owned, developed, maintained, operated, and evolved by a single team.

Services can internally adopt a microservices architecture where the implementation of a service internally reflects a service-oriented architecture. That is, the service is broken into components coupled by APIs rather than implementation details. Figure 3-4 shows an example microservices architecture for a single service within the service-oriented architecture. Microservices enable a team to develop a reliable and maintainable service by creating a loose coupling that creates components that can be independently developed and also fail independently. Microservices also result in more scalable services by separating stateful and stateless services, each of which can be independently scaled. Breaking a service into independent components, however, does add developmental overhead in the form of API development, build and release pipelines, and production management. Smaller services should weigh whether the benefits outweigh the overhead costs. For more on microservices, see *Monolith to Microservices* (O'Reilly) by Sam Newman.

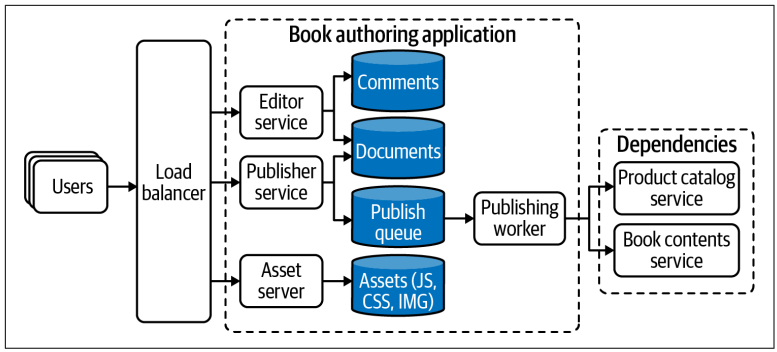

Figure 3-4. Microservices architecture where an API is implemented using a set of components loosely coupled through APIs. Dependencies may themselves be microservices.

Prefer remote services where possible to increase flexibility and reduce coupling, but sometimes functionality must be local to the client on the same production host for reasons like efficiency, latency, and security. You can publish client libraries for direct integration into a client application, but this can become expensive if multiple programming languages must be supported due to language-specific interfaces. Client libraries also limit evolution due to the dependence on a consumer's build and deployment process. Prefer to deploy sidecars, which are independent local processes that expose an API over a local communication channel. Sidecars enable you to develop in a single language and version the sidecar independently of the consuming application.

Synchronous, Asynchronous, Batch

Synchronous operations are ones where a client waits for the service to complete the operation before proceeding, and these operations are typically implemented using RPCs through frameworks like gRPC. Synchronous operations can be strongly or causally consistent, enabling you to more easily compose them to form high-order operations. However, synchronous operations tightly couple the client and the service, so if the service is unavailable or slow the client will be unable to make progress. Figure 3-5 shows an example synchronous operation for the book application to view the book catalog.

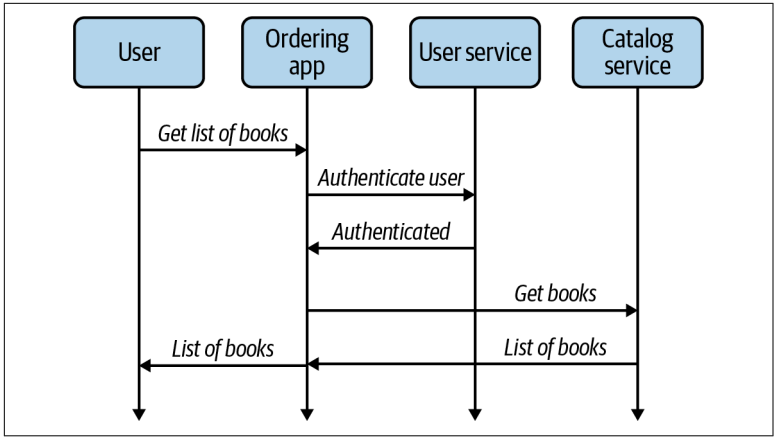

Figure 3-5. Example of synchronous operations to view the book catalog.

Asynchronous operations are ones where a client does not wait for completion and instead the operation completes independently in the background. This creates a loose coupling between the client and the service, enabling the client to make progress independent of the service, and these operations can be made more reliable through durable storage and retries. You can implement asynchronous operations with queueing systems, like publish–subscribe messaging systems, which can provide at-most-once, at-least-once, or exactly-once execution. At-most-once execution is where a system marks work done before attempting it, whereas at-least-once execution is where a system completes work and then marks it done. Exactly-once execution is where a system either transactionally completes work and marks it done, or it uses an idempotency token to dedupe multiple attempts to complete the work. Services leverage systems like publish–subscribe messaging for reliable at-least-once execution, optionally using an idempotency token to create exactly-once execution. Figure 3-6 shows an example asynchronous operation to send a receipt email after completing a book order. In the example you can see that delivery of the receipt is decoupled from responding that the order was successful.

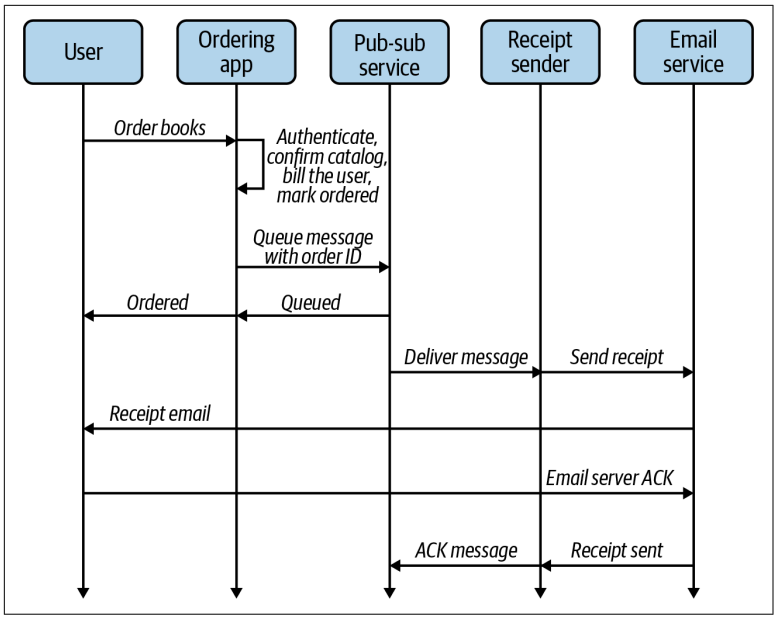

Figure 3-6. Example of asynchronous operations to send a receipt email.

Batch operations are ones where large computations or large datasets are processed all at once, typically on a periodic schedule, through a framework like MapReduce / Apache Hadoop, Flume / Apache Beam, or Dremel / GCP BigQuery. Batch operations can be more efficient through global operations like external sort, coalescing of incremental operations into a single periodic computation, and leveraging cheaper resources like preemptible compute environments. Figure 3-7 shows an example batch operation, triggered periodically via a cron system, to create a mapping of a book to a set of books that were purchased by the same set of users, for use in a simple recommendation system. In the example, you can see the shuffle operation represents a global sorting and joining of data, an operation that may be prohibitively expensive or generate too many transaction conflicts to do synchronously or asynchronously.

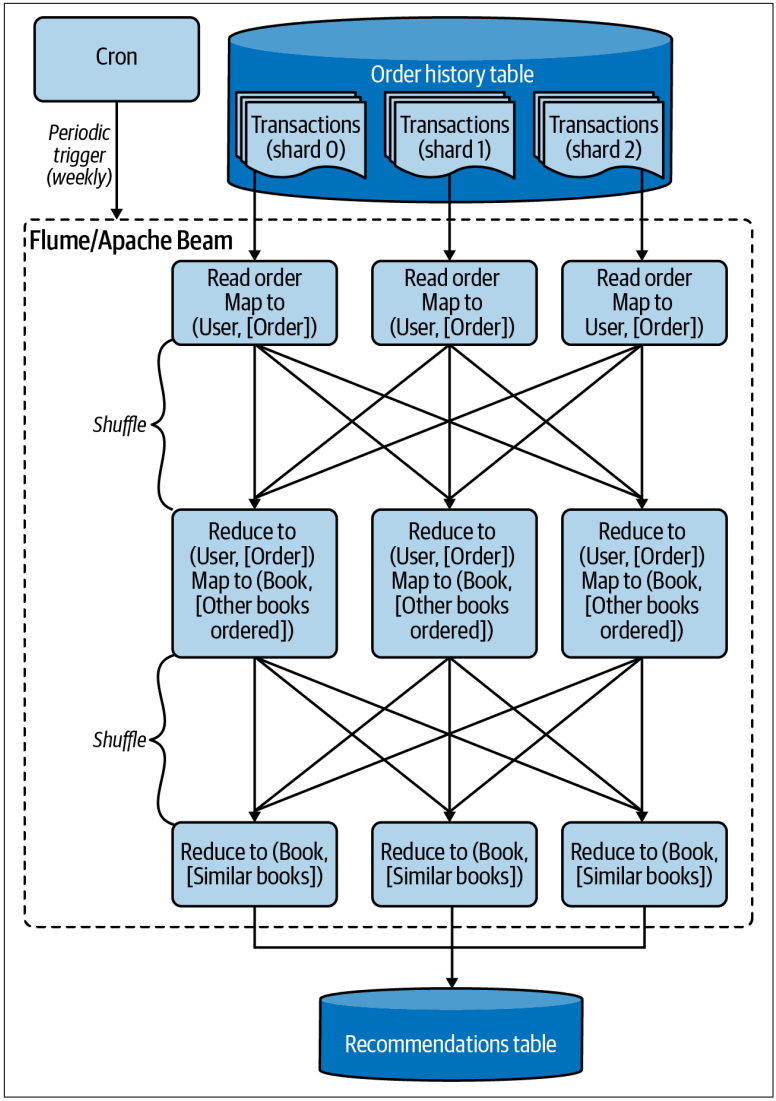

Figure 3-7. Example of a batch operation processing transactions to make recommendations.

Horizontal and Vertical Scaling

Design services to scale horizontally to serve more traffic and store more data by leveraging additional resources. Design services to scale vertically for more efficiency by leveraging locality and reducing overhead. System designers must consider how state is distributed over servers, how both larger and additional servers are introduced into a live system, how load is distributed over servers, and how servers coordinate.

Vertical scaling is the process of increasing or decreasing the size of a server to react to changes in traffic or stored data. Increasing the size of a server is more efficient than adding an additional server due to shared overhead like program data, higher efficiency of caches, and the lower cost of local operations like memory access. Remote operations, by contrast, require network requests, which can involve serialization, data copying, and higher latency. Figure 3-8 shows how doubling the size of a server doubles the variable cost capacity for serving requests without doubling the static costs, resulting in a resource savings compared to adding an additional server. Changing the shape of a server involves restarting the server, a process that purges ephemeral state like cached data and network connections, which can reduce service efficiency and reduce service capacity while scaling. You should gradually update servers when vertically scaling so that the reduction in efficiency and capacity does not risk service overload. Vertical scaling is thus not a good choice for reacting to rapid changes in traffic or stored data.

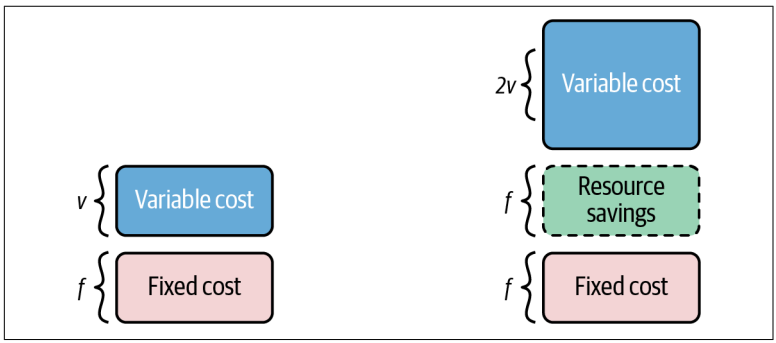

Figure 3-8. Vertical scaling of a server reduces fixed costs compared to horizontal scaling.

Horizontal scaling is the process of adding or removing replicas to react to changes in traffic or stored data. You can add or remove replicas without impacting the operation of others because the stateless servers don't store durable information and aren't responsible for state consistency, thereby enabling the service to be scaled up or down at high speed. Stateful servers, however, must first coordinate distribution of state before and after servers are added or removed, a process that can limit the speed at which service can scale. Subsequently, services typically push as much work as possible into stateless servers, which are easier to scale, and push stateful servers deeper into the request flow and stack to reduce the amount and frequency of work.

Horizontal scaling of stateful services is accomplished by sharding the state by keys that identify the state, like in a key-value pair in a map. Three methods of sharding are typical:

Hash-mod sharding

- Shard identifier is the hash of the key modulo the number of shards.
- Simple approach that can uniformly distribute state to shards through the uniform distribution properties of the underlying hash and bucketing properties of modulus.
- Eliminates keyspace locality necessary for in-order traversal, and modulus redistributes keys when the number of shards changes and requires migration of almost all data stored.
- Hash-mod sharding can be used to shard ephemeral state over cache servers where resizing is rare and redistribution cost is acceptable.

Consistent hash sharding

- Involves hashing keys to get identifiers, which yields a uniform distribution of identifiers, and then maps hashes to shards in a way that prevents large redistribution of keys when the number of shards changes.
- Eliminates keyspace locality necessary for in-order traversal.
- Consistent hashing can be used to shard ephemeral state over cache servers to enable rapid autoscaling of caches for efficiency and rapid scalability.

Range sharding

- Involves splitting the keyspace into ranges with an inclusive start key and exclusive end key, which maintains locality for operations like in-order traversal.
- Split points that divide a range into separate shards are typically selected by identifying keys that divide the state evenly through techniques like reservoir sampling, while also considering dynamic properties like traffic volume.
- Sharding over live systems is typically performed only by splitting a shard or merging adjacent shards recursively until state and load are balanced.
- More easily results in hotspots when traffic is not well distributed over the keyspace.
- Distributed storage systems like Bigtable and Spanner leverage range sharding in order to provide in-order scans of data.

Whereas sharding creates horizontal scaling of a dataset, replication creates horizontal scaling of an individual item by copying data to multiple servers that can each handle traffic. Algorithms like Paxos and Raft can be used to replicate data with strong consistency for writes and snapshot reads for scaling stale read traffic. Asynchronous replication can copy data through simpler methods with reduced coupling, but requires the application to handle eventual consistency, which can easily lead to application bugs. Finally, causally consistent replication can be achieved using tokens that ensure reads are logically as fresh as a reference. Replication increases the cost to store and write data, so some applications choose to dynamically replicate a subset of data, with a dynamic number of copies, to balance max throughput and cost. Figure 3-9 shows how stateful services can be horizontally scaled in two dimensions to handle more traffic or data.

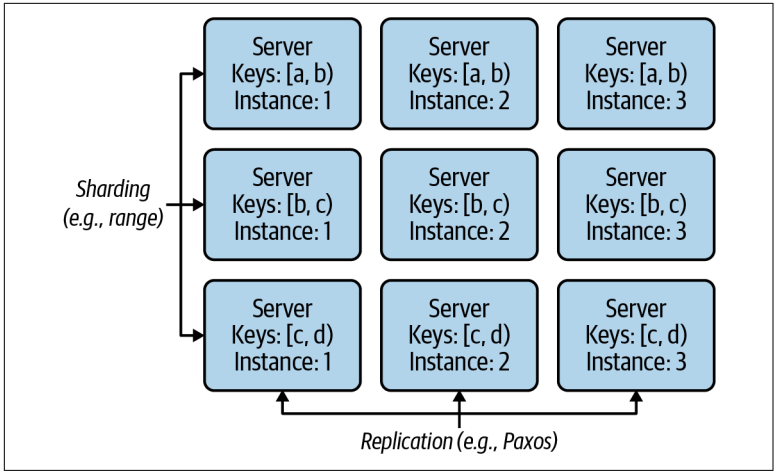

Figure 3-9. Horizontal scaling of stateful services through sharding and replication.

Horizontal and vertical scaling can also be applied to datacenters, however some of the trade-offs are different. Vertically scaling a datacenter involves adding more hosts, network capacity, power, and cooling to an existing datacenter. For traffic generated by nearby users, this ensures additional traffic can be accommodated without increasing WAN latency, and requires developing and maintaining fewer sites. However, datacenters have scaling limits at which additional land, power, or water may not be readily available. Horizontally scaling a datacenter involves creating additional facilities in other geographic locations. This can improve WAN latency for users who are further away than prior datacenters while increasing aggregate global capacity. Horizontal scaling of datacenters also creates additional failure domains, which can be leveraged to improve reliability. As a user of the cloud or datacenter provider, you must develop applications that can scale both horizontally and vertically, even if you don't need the reliability or efficiency wins, because your services will be otherwise limited by the size of the underlying hardware infrastructure.

Load balance the requests to leverage multiple datacenters and servers to scale to larger volumes of traffic. Datacenters are typically load balanced through a technique called waterfall where the nearest datacenter is sent all traffic until it reaches capacity, after which the next nearest datacenter receives the excess traffic, and so on until all datacenters are at capacity. This reduces WAN network cost and

latency by prioritizing the local datacenter while ensuring that datacenters don't exceed capacity. Figure 3-10 shows the waterfall algorithm in action. For external clients not under control, use edge load balancers exposing virtual IP addresses (VIPs) to quickly establish connections and balance traffic early. Once in a datacenter, it's typical to load balance across servers using a weighted round robin that's seeded with a random order at each client. Round robin ensures even load distribution over servers and weighted accounts for nonuniform request costs and server sizes, and the random seed ensures there aren't wave effects that synchronize the round-robin sequence across clients.

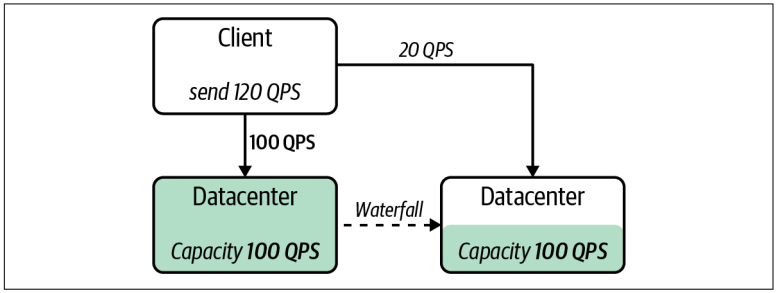

Figure 3-10. Waterfall load balancing across datacenters.

Figure 3-11 shows the weighted round-robin algorithm in action. For more on load balancing, see "Load Balancing at the Frontend" by Piotr Lewandowski and "Load Balancing in the Datacenter" by Alejandro Forero Cuervo in *Site Reliability Engineering*.

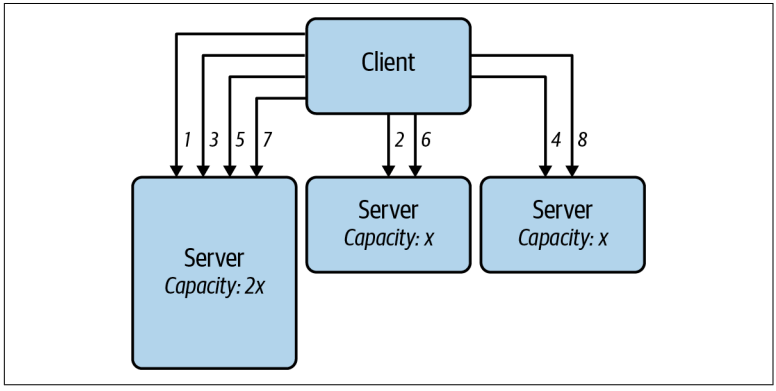

Figure 3-11. Weighted round-robin load balancing across servers with sequence of requests.

Capacity can be multidimensional when considering resource costs like CPU, RAM, and disk spindles, servers can have nonuniform capacity due to heterogeneous servers/hosts or in-flight vertical scaling, and requests can have nonuniform costs when they vary with things like request fields or response sizes. Load balancing must be multidimensional and weighted to accurately reflect the underlying properties of the system.

Deploy automation to automatically scale the service horizontally and vertically so that the service can react to changes in traffic, remain efficient, and avoid human operator toil. Horizontal scaling should ideally react within minutes to a change in load, adjusting capacity as often as every ~5 minutes and growing as fast as ~10x in size within that time. Automatic scaling is purposely rate limited to ensure performance and utilization signals stabilize before further adjusting, avoiding feedback loops, which may scale inappropriately. Vertical scaling, by contrast, is typically adjusted weekly, monthly, or quarterly and is deployed as part of the next binary or configuration rollout. Services not sensitive to overload or latency, like batch and analytic pipelines, may use real-time vertical scaling in a similar fashion to horizontal scaling. Automatic scaling should have safety limits on the range of acceptable values, like ensuring servers aren't made too small and ensuring the aggregate capacity doesn't starve other services of resources.

Utilization-based balancing is where servers respond with capacity utilization on each request, enabling load balancers to infer a capacity QPS (queries per second) from observed traffic and utilizations. These capacity QPS targets are then used for waterfall and weighted round-robin balancing. This simplifies system design by eliminating global capacity configuration. Inferred capacities are automatically recalculated, so during autoscaling the capacities and balancing decisions are automatically adjusted.

Frameworks and Platforms

Building a resilient service on the cloud requires solving numerous problems and applying many best practices. Over time the problems, solutions, and best practices change and require evolving systems. This is a significant challenge that can impede a development team's velocity. At scale these problems are best handled once centrally, typically by a company's platform development organization.

There are three major types of frameworks/platforms:

- Proxies that provide common functionality to incoming and outgoing requests. Service mesh frameworks like Istio can add common functionality like multiformat request parsing, monitoring, and audit logging.
- Server frameworks that make developing server software easier, making it trivial to perform common tasks like establishing database connections, reporting server health, and annotating requests with capacity utilization.
- Deployment platforms that enable a team to rapidly define and deploy a new microservice, handling common tasks like setting up build and deployment, load-balancing configurations, and resource provisioning.

Together, these frameworks enable a service developer to make their services resilient while focusing their time on the unique problems of the service, delegating responsibility for common problems to the framework and platform developers. For example, a service owner may need to specify the build target for their server's binary, but they shouldn't need to set up deployment, autoscaling, and load balancing. When best practices change, platform developers update the platform and the improvements are deployed to all services in the organization.

Standardize on a narrow set of developer experiences to minimize cognitive load and enable teams to flex to changing requirements. For example, if all services in an organization use the same programming language, a developer needs to be an expert on fewer topics and can more easily contribute to other services within the organization as needs evolve. Of course, this must be balanced with using the right tool for the job. At Google, teams use C++, Kotlin, Java, Python, and Go for general-purpose programming; JavaScript, TypeScript, and Dart for web programming; and Objective-C/C++ and Swift for programming on Apple platforms. Within a smaller organization of 100 to 1,000 developers, it's typical to pick exactly one general-purpose and web programming language.

Recap

In this chapter, we've covered how to architect a reliable service:

Application programming interfaces (APIs)

- Use a resource-oriented design where there are resources (nouns like a book) and methods (verbs like create). Provide methods to create (`HTTP POST`), read/list (`GET`), update (`PUT`), and delete (`DELETE`).
- APIs must handle failures, retries, and concurrency.
- Use the long-running operation pattern for methods that take significant time.
- Specify whether an API provides strong, causal, or eventual consistency.
- Use standard error codes for predictable and reliable client-server interactions.
- Use stability levels like alpha, beta, and generally available (GA) to set expectations for how the API will evolve.

Tiered, service oriented, microservices

- Two-tier architectures consist of a frontend server and a back-end database.
- Three-tier architectures consist of a frontend server, a series of product infrastructure services like a user service, and a back-end database layer.
- Service-oriented architectures are where each business function develops a separate service with its own API, where services are loosely coupled only through those APIs.
- Microservices architectures are where a service is internally composed of components coupled by APIs rather than implementation details.

Synchronous, asynchronous, batch

- Synchronous operations are ones where a client waits for the service to complete the operation before proceeding.
- Asynchronous operations are ones where a client does not wait for completion and instead the operation completes independently in the background.

- Use asynchronous operations to decouple the client and service and increase reliability through durable storage and long-duration retries.
- Understand whether queueing systems like publish–subscribe are providing at-most-once, at-least-once, or exactly-once execution.
- Batch operations are where large computations or large datasets are processed all at once to be more efficient.

Horizontal and vertical scaling

- Design services to scale both horizontally and vertically.
- Horizontal scaling is the process of adding or removing replicas and is best for reacting to changes in traffic or stored data.
- Vertical scaling is the process of increasing or decreasing the size of a server and is best for optimizing the efficiency of a service.
- Horizontally scale stateful services using sharding and replication.
- Load balance across data centers using the waterfall algorithm, and within a data center using the weighted round-robin algorithm.

Frameworks and platforms

- Leverage frameworks and platforms to focus on the unique problems of the service, delegating responsibility for common problems to the framework and platform developers.
- The three major classes of frameworks/platforms include request proxies, server development frameworks, and deployment platforms.

Next, we'll learn about common failure modes and how to mitigate and prevent them.

CHAPTER 4

Avoid Common Failure Modes

Resilient services must defend against common failure modes like deploying a bad change, error handling exacerbating a problem, resource exhaustion, thundering herds[1] and hotspots creating overload, and data corruption losing data. This chapter introduces these common failure modes and discusses techniques you can use to defend against them.

Bad Changes

At Google, the most common trigger of an outage is the deployment of a bad change. For example, in one major outage, an error introduced in a load-balancing configuration was rapidly deployed to all load balancers, causing them to establish additional backend connections, run out of memory, and disrupt global network traffic. In another major outage, a quota management system received an erroneous usage report, which resulted in a global resizing of quota, resulting in Paxos write failures that led to Paxos log staleness, bounded-staleness read failures, authentication failures, and global service disruption. To defend against these outages, Google has established a set of change management principles shown in Table 4-1.

1 A sudden and rapid increase in traffic typically initiated by an event.

Table 4-1. Change management principles

Principle	Description
Change Supervision	Monitor services to detect faults, including dependencies and dependents for end-to-end coverage.
Progressive Rollout	Make changes slowly across isolated failure domains so issues can be detected and mitigated before having a large impact.
Safe & Tested Mitigations	Have rapid, low-risk, and tested mitigations like change rollback and automatically execute them on issue detection.
Defense in Depth	Independently verify correctness of changes at each layer of the stack for multiple defenses against failure.

Supervise changes by monitoring instrumentation of the components. Monitoring must be fine-grained enough to detect anomalies in a new version when it has only been rolled out to a small fraction of clients, servers, or systems. Time-series metrics can provide indicators like server crash rate, request error ratio, and request latency. Probers can automatically evaluate a service end to end by periodically executing a test that exercises the API the way a consumer would. Use metrics and probers to detect performance deviations and unhealthiness. For example, you can detect that a change notably increased latency by 10% or produced a high error rate of 1%. Change management systems must monitor these signals while rolling out a change to detect and mitigate an issue. Once an issue is detected, you'll want further instrumentation like dashboards, logging, tracing, and profiling in order to identify, mitigate, and prevent the contributing factors.

Rollout changes progressively by creating independent and small failure domains and then changing those failure domains one at a time. If the change is bad it'll cause the domain to fail, but the other domains will remain healthy. Change management systems should detect issues in the updated domain and halt the rollout before many domains are broken. It's typical to create independent failure domains at the granularities of server, zone, region, and continent. For global services, it's also common to create independent failure domains at data or service boundaries, like sharding users across 10 independent instances of the service. Resilient services are typically rolled out over a week, waiting at least several minutes between server updates. Regions are typically grouped into waves of 0.1%, 1%, 10%, 50%, and 100%, where waves are spaced by one business day to expose issues tied to daily traffic cycles and global demand

fluctuations. Figure 4-1 shows an example gradual rollout schedule over a week.

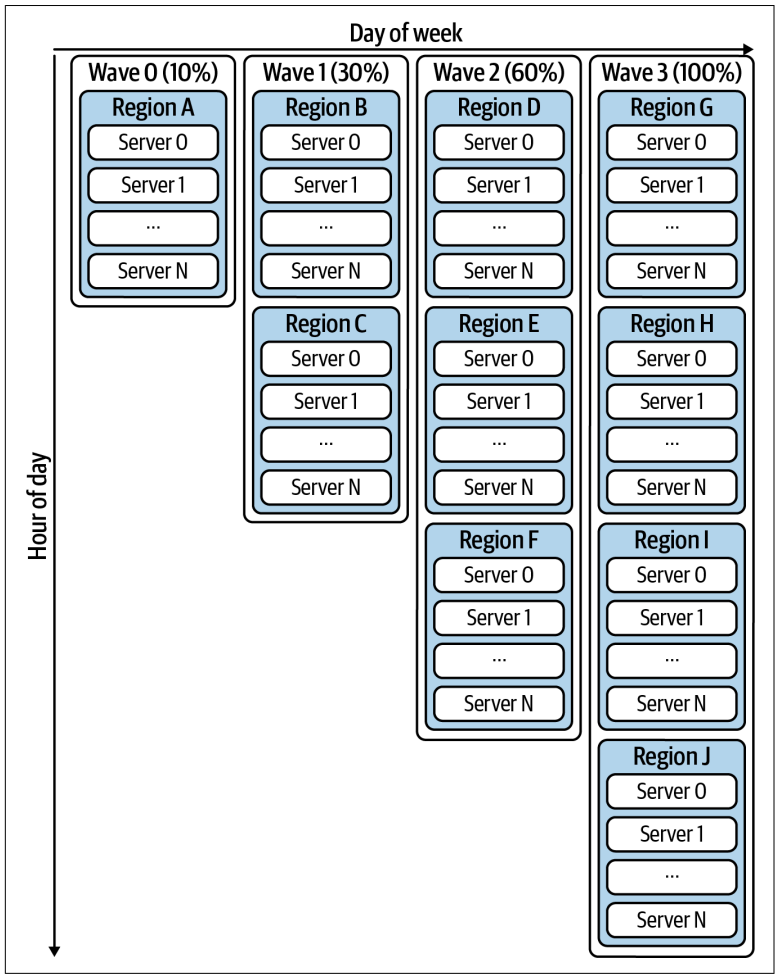

Figure 4-1. Example gradual rollout schedule.

Automatically apply safe and tested mitigations, like change rollback, once an issue has been detected in order to reduce the duration of outages and minimize the error budget exhausted. For example, an automatic rollback can reduce repair time from a human response time of hours to a machine response time of minutes. For automatic rollbacks to be safe, the service and the changes must be forward and backward compatible between adjacent versions.

Verify service configuration for correctness and sanity. For example, both the service as well as systems that store and propagate configuration changes should detect and alert when configuration is empty, partial or truncated, corrupted, logically incorrect or unexpected, or not received within expected time. The service should gracefully degrade by continuing to operate in the previous state, or in a fallback mode, until the bad input can be corrected. Operational tools should reject invalid configurations, configurations that change too much from the previous version, and potentially destructive changes (e.g., revoke all permissions from all users). Source control should similarly evaluate changes with required pre-merge checks. Changes deemed risky should only be applied if operators use emergency overrides like a command-line flag, a configuration option, and disabling pre-merge checks.

Deploy isolated environments for development (dev), staging, and production (prod). Changes will be deployed first to dev, evaluated in dev, deployed to staging, evaluated in staging, and finally deployed to prod. By unit, integration, and load testing changes in dev and staging before prod, issues can be detected and mitigated before reaching end users. The dev environment's purpose is to rapidly test and iterate on changes in a realistic setup. The staging environment will act as a dry run for updating production, and will also be a more stable environment than dev with which to test the interactions with dependencies and dependents before reaching prod. Figure 4-2 shows the coupling between services across environments.

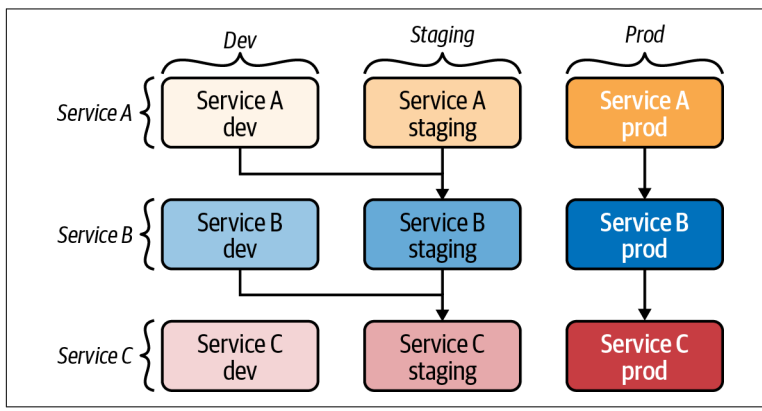

Figure 4-2. Coupling between services across environments.

Finally, use infrastructure-as-code (IaC) and multiparty authorization (MPA) to safely make changes to production. IaC ensures that infrastructure configuration changes undergo review, are evaluated with pre-merge checks, and can be versioned and deployed similar to code. Multiparty authorization of commands can ensure one-off processes also undergo review to avoid common mistakes like typos.

Error Handling

How errors are handled can improve the reliability of the service by mitigating issues that would otherwise be exposed to consumers. Error handling can also exacerbate a problem and become the cause of an outage. Ensure services and clients generate appropriate errors and are well behaved in the presence of errors.

Gracefully degrade in the presence of errors by leveraging soft dependencies and having fail-safe alternatives. For example, a shopping page that contains reviews could be rendered without those reviews if the backend storing reviews is returning errors, enabling the customers to browse and checkout even if the experience isn't ideal. Similarly, if there are 100 backends serving reviews but only a subset respond, the page can still be rendered with partial results, ensuring the customer gets some benefit from the review system. As another example, a load balancer distributing traffic based on server utilizations can fall back to round robin if the system gets bad utilization data, ensuring the system is able to serve some traffic even if load distribution isn't uniform.

Services must return appropriate error codes and clients must be well behaved by responding appropriately to those error codes. Certain error codes like `UNAVAILABLE` indicate an issue may be transient and can be safely retried. Retry these errors across backends to increase the probability of requests succeeding so issues are mitigated and consumers are not impacted.

Error codes for overload or resource exhaustion should differentiate between server overload, service overload, and quota exhaustion. Server overload should be retried as there's a chance the request may succeed on another server. For the other error codes, immediate retries are unlikely to succeed so short-deadline traffic like user requests should not be retried. However, batch traffic, which can afford to wait, can mitigate longer issues by retrying after a backoff

delay. Table 4-2 summarizes the overload errors and when a client should retry the request.

Table 4-2. Overload error codes and retry conditions

Overload error	Error code	Retry for user traffic	Retry for batch traffic with a delay
Server	`UNAVAILABLE`	Yes	Yes
Quota	`RESOURCE_EXHAUSTED`	No	Yes
Service	`INTERNAL`	No	Yes

Error codes are usually propagated directly to upstream systems. However, if retrying a request fails multiple times due to local server overload, the propagated error should be a service overload to prevent retry amplification by upstream components, which could further overload the system. Figure 4-3 shows a request flow where a service retries an `UNAVAILABLE` error three times before returning an `INTERNAL` error to the consumer. If the service had instead propagated the `UNAVAILABLE` error, the upstream consumer may have itself retried three times leading to nine total requests to the dependency servers. Those extra six requests from retry amplification may further overload the dependency.

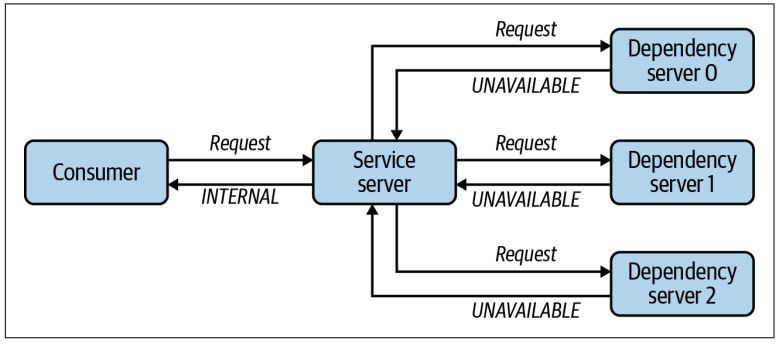

Figure 4-3. Preventing retry amplification by propagating an `INTERNAL` error.

Clients must slow down the sending of requests (backoff) when the service is overloaded to limit the cost of rejected requests and ensure the service isn't pushed into cascading failure. The TCP network protocol leverages exponential backoff to ensure the connection remains healthy and data isn't sent faster than it can be received. While exponential backoff will also work at the application layer for

sending RPCs, it is commonly misimplemented to back off in the scope of an individual request rather than the connection between processes. Consider Figure 4-4, which shows the rate a client will send requests over time with different sending strategies. In this scenario, the backend has capacity for 2k QPS (dotted line) and the client is attempting to send 10k QPS (blue line), yielding 8k QPS of overload errors. A client that retries each throttled request up to three times (red line) will send 26k QPS to the backend, yielding further overload. A client that leverages per-request exponential backoff (yellow line) will spread a single request's load over a longer period, potentially mitigating transient issues, but will still send 26k QPS and fail to reduce load on the backend. A technique called adaptive throttling (green line) throttles RPCs based on observed throughput and a target error ratio. This tunable target error ratio balances overhead of rejected requests with the speed at which the client adapts to a change in backend capacity. For more on adaptive throttling, see "Handling Overload" by Alejandro Forero Cuervo in *Site Reliability Engineering*.

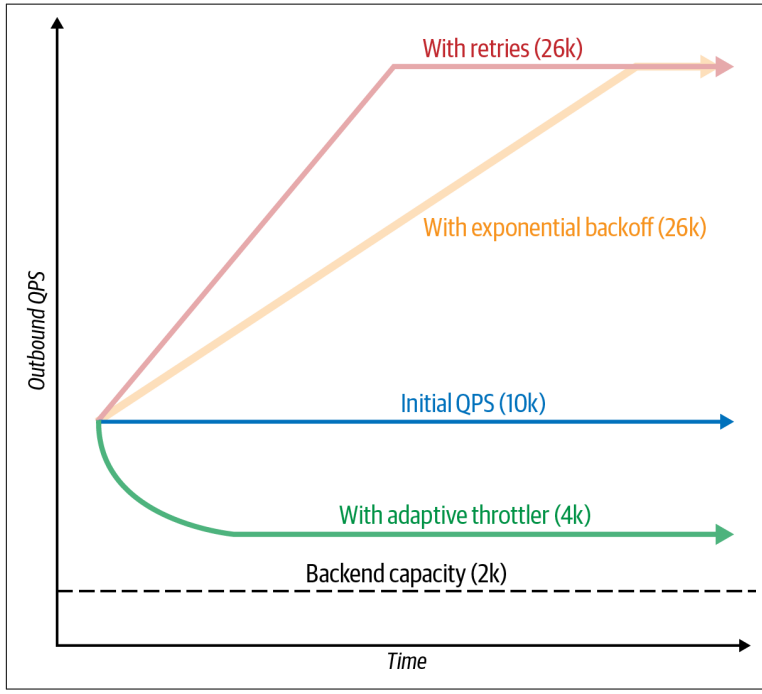

Figure 4-4. Adaptive throttler versus per-request exponential backoff.

Backoff through delays/sleeping should add jitter, or random amounts of additional delay to each retried request. Without jitter, multiple concurrent requests with the same delay may inadvertently synchronize retries and create an instantaneous traffic spike leading to overload. By adding jitter, retry traffic will be smoothed out and load spread over time.

API parameters must be validated and sanitized so that erroneous, random, or malicious inputs cannot cause service outages or security breaches. Defend in depth by validating inputs at each layer of the stack and within each component of code. Explicitly test error handling to validate error codes generated and behavior in edge cases. Test for unexpected scenarios through fuzz testing by intentionally calling service APIs with random, empty, or too-large inputs, which will expose insufficient parameter validation.

Attach deadlines or timeouts to requests and check if these have been exceeded at multiple layers of the system and call stack to prevent problematic requests from exhausting resources. Deadlines are absolute timestamps specifying when processing must complete, and timeouts are durations specifying how long processing is allowed to take. Deadlines and timeouts ensure that an unhealthy request or system eventually drops the request even if the request never generates an error. This prevents a request from being processed indefinitely, which may contend resources until the server is restarted. Propagate deadlines and timeouts across RPCs so that downstream components stop processing requests that the upstream caller has already abandoned. For example, if a service uses a 100ms timeout to process a request but a backend takes 200ms, the backend should instead abort the request after 100ms as the service will have already moved on and subsequently will ignore any response after that point.

Resource Exhaustion

Resource exhaustion occurs when the load on a service exceeds the resources provisioned. Load on a service can increase due to organic growth like the gradual adoption by more users each month, or due to events like the release of a new feature that suddenly increases user demand, the release of a new version that contains a performance degradation, or an infrastructure event like a cache flush or the start of a batch job.

Two particularly risky failure modes are cascading failures and capacity caches. Cascading failure is where the overload of a single component causes the load to shift to another component, which subsequently becomes overloaded and fails, repeating until all components have failed and the system is unable to recover. Figure 4-5 shows how cascading failures occur after the overload of a single component. For more on cascading failures, see "Addressing Cascading Failures" by Mike Ulrich in *Site Reliability Engineering*. Capacity caches are where the provisioning of the system depends on the efficiency gains of a cache, so if the cache underperforms due to a change in traffic or data distribution, the system as a whole is underprovisioned and resources become exhausted. These risky scenarios can be mitigated by dropping traffic until the system recovers or by provisioning additional resources, and these scenarios can be prevented by not introducing caches to solve capacity problems, by testing the system in overload, by testing the system without the effect of the cache via cache flushes, and by the defensive techniques discussed in this section.

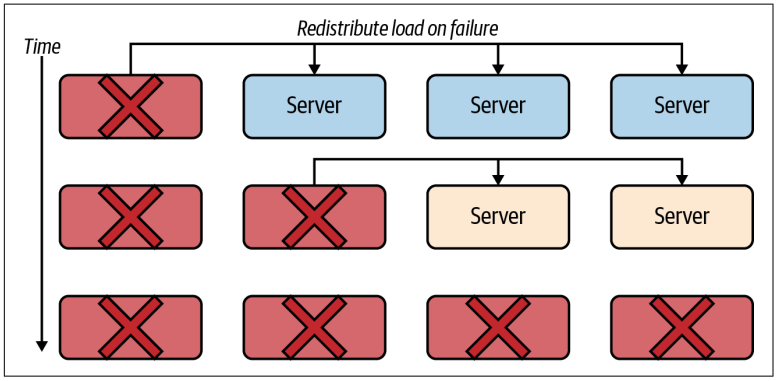

Figure 4-5. Cascading failure due to resource overload.

Services should gracefully degrade in such a way that the degradation maintains SLOs for traffic that fits within prior provisioning and only returns errors for the excess load. Defend against resource exhaustion through:

Cost modeling
: Measure resource usage and predict the cost and volume of work.

Load shedding
: Drop excess load that would risk service health if ingested.

Quotas
: Limit consumer load to pre-requested amounts and provision to reflect limits.

QoS/criticality
: Prioritize work and drop load that is less important.

Autoscaling
: Automatically react to changes in load and provision additional servers.

Caching
: Increase efficiency and thereby max load by degrading to staleness.

Capacity planning
: Forecast demand and model service cost.

Cost modeling involves measuring resource usage and predicting both the cost and volume of work. Services should measure provisioned resource dimensions like CPU, RAM, stored bytes, and disk spindles, and indirect measures like queries per second (QPS), concurrent in-flight requests, and throughput. These measurements should be made on historical production data for realistic distributions, on synthetic load tests to evaluate changes not yet deployed, and on live servers to reflect real-time conditions. From these measurements, you can predict the cost of a request, the volume of requests, and the resources necessary to serve a given load. For example, you can observe that over the last 90 days the 99th percentile usage was 100 CPUs serving 10k QPS of traffic, and the traffic is growing at a rate of 10% per month.

Load shedding involves dropping excess load that would risk service health if accepted, and is thus critical for defense against overload and cascading failure. There are two forms of load shedding:

- Request-based load shedding is where the server, on receipt of a request, evaluates current server health and immediately decides to accept or reject the request. If the request is rejected, an `UNAVAILABLE` error is returned so the client can retry the request at another server.

- Queue-based load shedding is where the server puts incoming requests into a queue, rejecting requests only if the queue is too large or the server is unlikely to process the request before its deadline. Queue scheduling ensures that request processing starts only if it wouldn't risk service health. Queue scheduling should use a last-in-first-out (LIFO) policy to maximize useful work completed by preventing a failure mode where work dequeued via first-in-first-out (FIFO) has insufficient time remaining before the request deadline. This may seem surprising because LIFO scheduling adds disproportionate queue delay to requests arriving earlier, which increases the chance those requests exceed deadlines and thus may appear to increase overall error rates. When overloaded, however, the lowest error rate will occur when resources are maximally used to produce successful responses, and LIFO scheduling minimizes resources wasted on requests that will exceed deadlines anyway.

Both load-shedding methods ensure that accepting a request does not cause the server to become unhealthy. Request-based load shedding has better end-to-end latency when the time to retry across servers is faster than an individual server's queue delay. Queue-based load shedding is more efficient by not resending requests, which saves data copying and serialization costs. At Google, most servers use request-based load shedding for its simplicity, whereas the most performance sensitive servers use queue-based load shedding and scheduling for its efficiency.

Quotas are limits on service usage established ahead of time between a consumer and the service, limits that enable the service to provision for a specific amount of load, to set usage expectations a consumer can depend on, and to isolate consumers from each other. Quotas should be defined at all layers of the stack including the application and the cloud service provider. That is, as an application developer you are subject to cloud service provider quotas, and you should define and enforce quotas for your own service and its consumers. A quota is broken into two components: a commitment, which is the amount a consumer is guaranteed to be able to use, and a ceiling, which is the largest value a commitment could be adjusted to without negotiation with the service. A consumer's ability to increase the commitment is not always guaranteed and could depend on current commitments across consumers and the service's provisioning. The separation between commitment and ceiling

enables a service to safely oversubscribe resources and to reduce the frequency of quota negotiations. A service with hard limits will reject all requests over the commitment, whereas a service with soft limits can accept requests over the commitment so long as it doesn't sacrifice isolation between consumers. Prefer soft limits to reduce headroom for traffic fluctuations and to reduce over-quota error rates when the service has spare capacity.

Quality of service (QoS)/criticality are techniques for assigning priorities to different streams or individual requests so that upon resource exhaustion the requests are dropped in priority order. For example, the service can prioritize user-facing requests over traffic from batch jobs that can be delayed without issue. Table 4-3 shows common criticality levels for requests. Request-based load shedding and quota enforcement will use different utilization thresholds at which rejections occur to create prioritization. For example, `CRITICAL_PLUS` may be dropped at 100% utilization whereas `SHEDDABLE` may be dropped at 50% utilization. Queue-based load shedding and quota enforcement will simply schedule work in criticality order.

Table 4-3. Common criticality levels for requests

Criticality	Description	Use case
`CRITICAL_PLUS`	Most important traffic class Service provisioned for 100% of traffic Expected to be <50% of traffic	Most important >user-facing requests
`CRITICAL`	Service provisioned for 100% of traffic Shedding expected only in outage	Default for user-facing requests
`SHEDDABLE_PLUS`	Partial unavailability expected	Non-user-facing requests like async work and data processing pipelines
`SHEDDABLE`	Frequent partial and occasional full unavailability expected	Best effort traffic where progress is not critical

Within an organization, multiple services are all unlikely to peak in load at the same time. Within a service, multiple components are also all unlikely to peak in load at the same time. With fixed service and component provisioning, one service or component can be overloaded while another is at relatively low utilization. Leverage autoscaling to dynamically scale down the underutilized servers, freeing up resources which can then be used to scale up the

overloaded servers. Autoscaling provisions capacity where it's needed to reduce error rates without increasing total resource cost.

Through load shedding, quotas, criticality, and autoscaling, the service should remain healthy during overload and gracefully degrade to serve as much traffic as possible in priority order. However, this will still yield errors for the excess load beyond provisioned capacity. To do better, you'll need to increase efficiency or add additional resources to the service.

Caching improves service efficiency by looking up a precomputed value from optimized storage, which can enable the service to handle more load by trading for eventual consistency and data staleness. As resource utilization increases, the service can dynamically adjust caching parameters like expiration times so the cache is more effective, thereby increasing request capacity and reducing error rates due to overload. For example, when a cached item is at expiration time, the service can attempt to lookup a fresher value, replacing it with the new data if successful and falling back to the stale data if the system is overloaded. Note this use case is different from the capacity caches previously discussed. While you should leverage caching to mitigate unexpected traffic, you should not rely on caching in order to serve expected traffic.

Finally, ensure the service has capacity for foreseeable demand through capacity planning. When forecasting resource needs, it's important to account for lead time for acquiring resources and for variability in demand. Capacity planning can be automated end to end by modeling the relationship between services and components (e.g., service A depends on service B), modeling the high-level constraints on a service (e.g., must have presence in these regions with `N + 2` redundancy), observing service usage indicators (e.g., QPS over time and CPU utilization), and feeding all of this data into an optimizer that reshapes production systems into optimal configurations and provides inputs for machine purchasing.

For more on resource exhaustion, see "Handling Overload".

Thundering Herds and Hotspots

A thundering herd is a sudden and rapid increase in traffic typically initiated by an event. For example, if an app requires users to have the latest version of the app for it to function, then on release of a new version, all users are forced to immediately and simultaneously upgrade. If the app has millions of users, the application download system may receive millions of requests within seconds of the version release. Figure 4-6 shows such an example of a thundering herd of requests. Another example is the aftermath of a mobile carrier outage, a moment when many phone apps retry requests as soon as the network recovers, leading to a thundering herd. Thundering herds are particularly dangerous because they can overload a system faster than mitigations like caching and autoscaling can react.

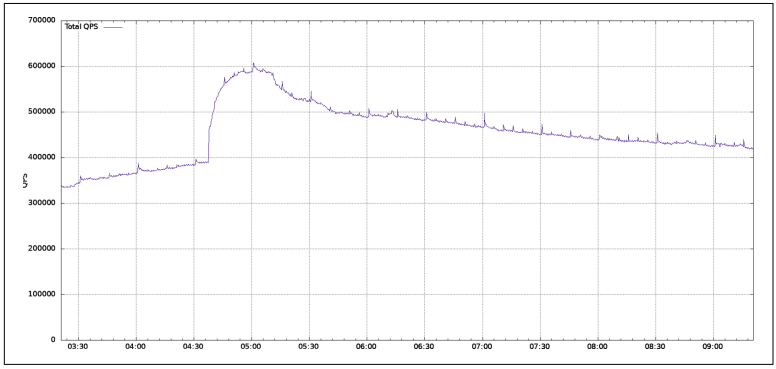

Figure 4-6. Service QPS when an app version upgrade yielded a thundering herd.

A hotspot is where a subset of servers receive a disproportionate amount of load and subsequently become overloaded, despite overall service having spare capacity. Hotspots frequently occur during thundering herds. For example, an app store serving application downloads may experience a hotspot when the previously described version release sends millions of requests for a specific resource identifier, potentially causing the canonical server storing the new version's data to receive all of the new traffic. Figure 4-7 shows a hotspot in a service architecture.

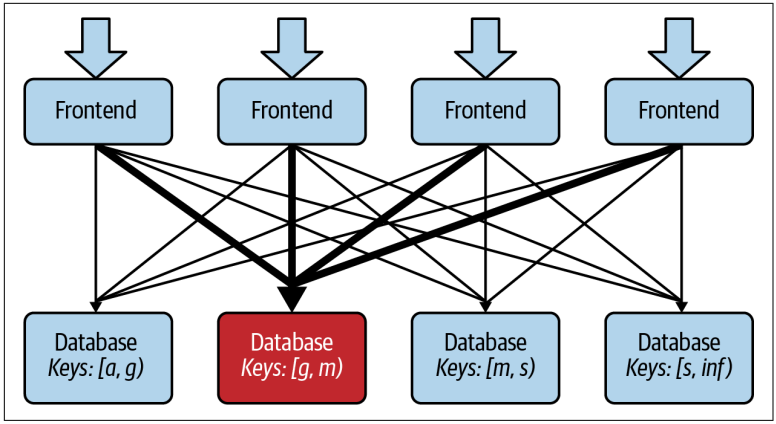

Figure 4-7. Traffic hotspot to a single database server storing data for a popular resource.

Gating is a technique for deduping equivalent operations and performing work once to satisfy multiple requests. Prior to doing an operation, the server can first check to see if there are equivalent concurrent operations in flight and if so, combine them into a batch. The server can then execute the operation once and return the result to each request as if the operation ran multiple times. Gating ensures traffic to backends is proportional to the number of servers and latency of an operation, rather than being proportional to incoming traffic, ensuring that thundering herds and hotspots cannot overload backends. For example, if a backend request takes 100ms to complete, then a single server will send at most 10 QPS of traffic to the backend regardless of whether that server receives 10 QPS of requests or 10k QPS of requests. Figure 4-8 shows the reduction in load on a distributed system visually.

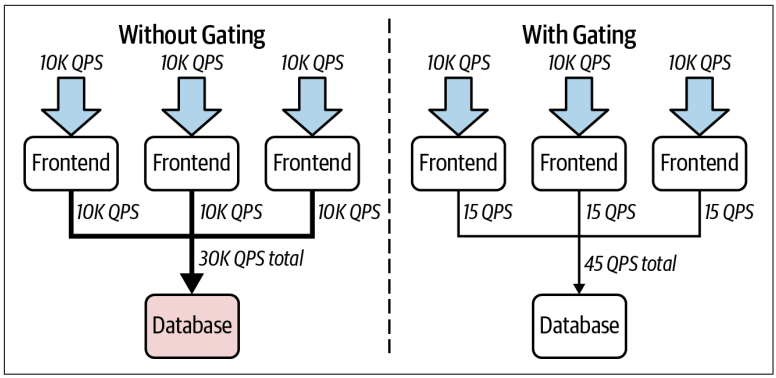

Figure 4-8. Impact of gating on distributed system load.

Gating can be implemented two ways. To achieve eventual consistency with lower average latency, requests can be grouped into the batch that is currently being processed. To achieve strong consistency with higher average latency, requests can be grouped into batches that are only started after the batch currently being processed is completed. Figures 4-9 and 4-10 show how the two approaches work. Under heavy load both approaches reduce traffic equivalently, so the choice between them is typically based on consistency and latency requirements.

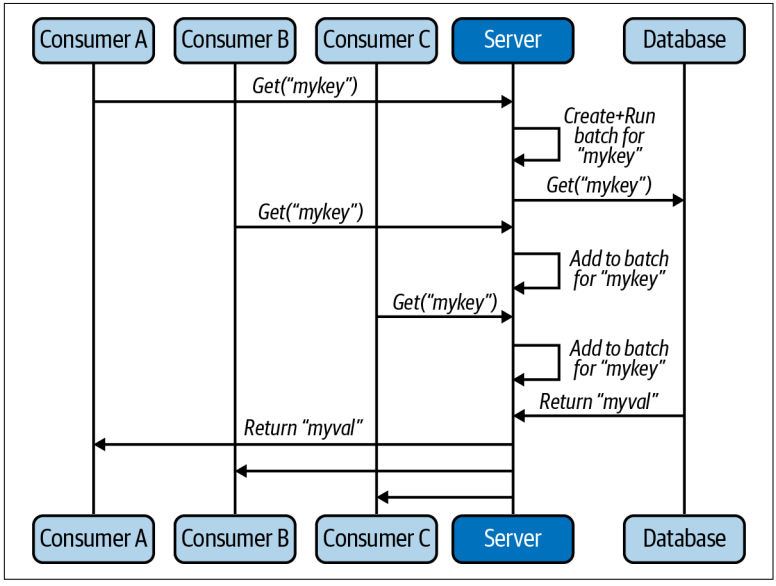

Figure 4-9. Reading a database with eventually consistent gating.

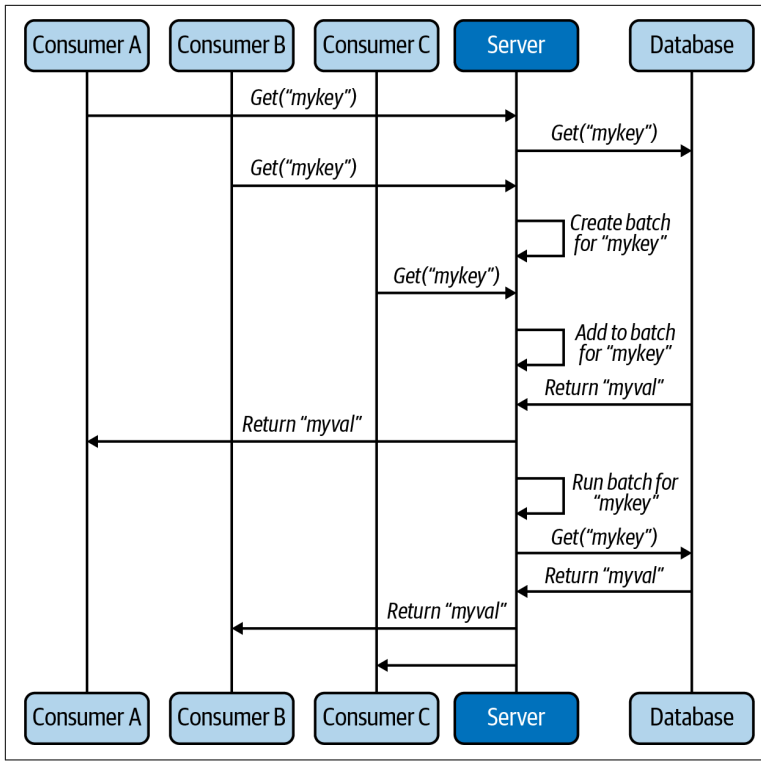

Figure 4-10. Reading a database with strongly consistent gating.

Hierarchical gating and caching is a technique where gating and caching are applied at multiple points to further dedupe load reaching a backend. This technique is sometimes referred to as L1 and L2 caching as it's similar to how CPUs cache data from RAM. Figure 4-11 shows hierarchical caching in action where a server applies gating and caching prior to sending requests, and the backend server applies gating and caching upon receiving requests. The L1 gating dedupes concurrent requests and the L1 caching ensures keys in high demand don't hotspot the backend. The L2 gating further dedupes concurrent requests and the L2 caching reduces overall load on storage.

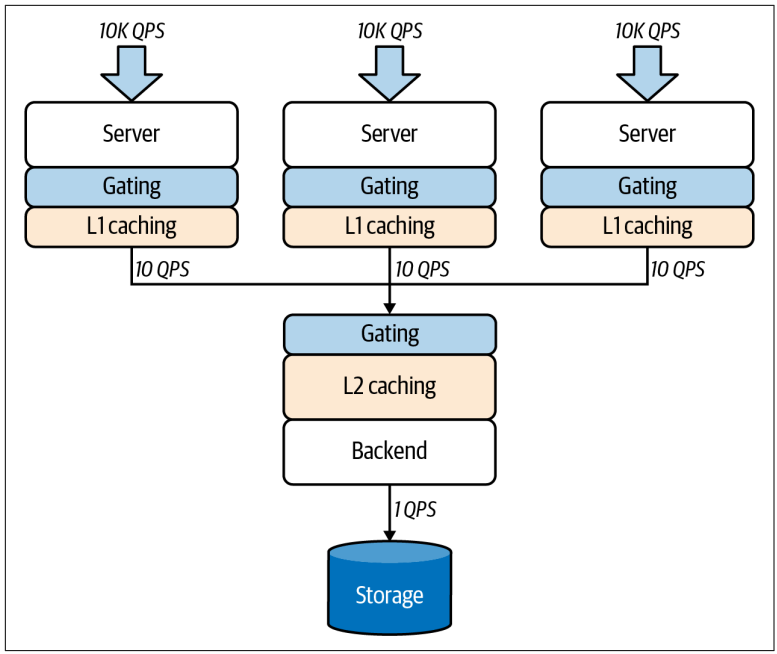

Figure 4-11. Hierarchical gating and caching.

Add jitter or random delays to cache expiries to ensure cache expiration doesn't become the cause of a thundering herd. For example, if there are 10k frontend servers each caching a particular key, if they all expire the data for the key at the same time, then the 10k servers will all send a request to the backend at the same time. Coordinated expiry frequently occurs during thundering herds, but also when the expiration time is synchronized by a backend. For example, coordinated expiry can occur if the expiration time is based on a fixed offset from when the backend first cached a storage read. By adding random delays, servers will send the backend requests at different times thereby spreading load over time.

Replication of data can also help by leveraging additional servers at the cost of eventual consistency. Static replication of all data to multiple servers will help with thundering herds and with hotspots, but this can increase the cost of storage by two or more times. Dynamic replication where data is selectively replicated based on demand will help with long-running hotspots with much lower increases in storage cost. However, dynamic replication will not help

with thundering herds because the system will become overloaded faster than demand can be measured and data replicated.

Integrity, Backup, Recovery

It's inevitable that a software bug will be released, a hard-drive disk will fail, and a mismanufactured CPU will corrupt data. Services must detect, mitigate, and recover from data loss and data corruption. You should set SLOs for recovery, use probers and checksums to detect data loss and corruption, and have backups and restore procedures to recover.

Services should have two SLOs for recovery from failure. The recovery point objective (RPO) is the point in time after which data may be permanently lost upon a failure. The recovery time objective (RTO) is the duration of time to restore the service after a failure, a period during which the service may be unavailable. For example, let's say the failure is an accidental deletion of the service's database. If your service sets an RPO of four hours, then any data written to the database in the last four hours may be lost. If your service sets an RTO of one hour then the service may be unavailable for one hour while the system recovers from the deletion. In order to meet these objectives, the service may create backups every four hours and regularly test recovery from backup to ensure it can be done within one hour. Figure 4-12 shows RPO and RTO visually on a timeline. Similar to other SLOs, the RPO and RTO should be set based on the needs of the users and business.

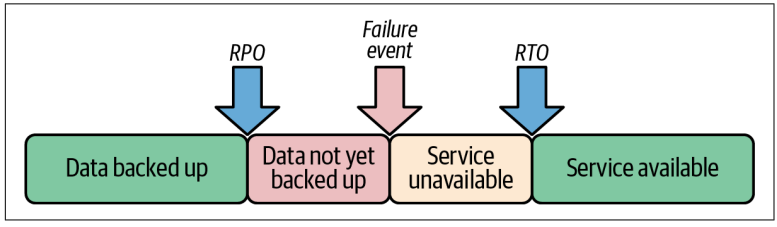

Figure 4-12. Recovery point objective (RPO) and recovery time objective (RTO).

RPO is typically measured by reporting the age of the freshest backup over time, yielding a measure similar to freshness SLOs. RTO is ideally measured by periodically executing an automated restore, yielding a measure similar to those generated by probers. For simplicity, RTO may be evaluated for just storage recovery like a

database restore, or for critical services it can be the end-to-end recovery of the service. RPO and RTO are also tested as part of Disaster Recovery Testing (DiRT) and with techniques like Chaos Engineering.

Deploy probers to detect data loss or corruption by automatically testing that data can be read after written, and that data written a long time ago can still be read. Run these probers in development and staging environments to detect bugs before they can impact production workloads. Run these probers in production to reduce the time to detect an outage and thereby reduce the duration of outages. Finally, use burn-in load tests to exercise new and existing hardware to detect sources of corruption before production workloads are scheduled on the machines.

Generate and verify checksums to validate end-to-end integrity of data. Checksums like CRC32C can detect unintended changes to data like corruption from a bad CPU. Generate checksums as close to the source of data as possible and verify checksums as close to the use of data as possible in order to detect corruption end to end. For example, generating a checksum in the client before sending a write to a server and verifying the checksum in the client after a read from a server will ensure you detect any corruption that occurs in transit, during processing at the server, or while being stored on disk. Generate overlapping checksums at multiple layers for defense in depth. For example, checksums can be separately generated and verified for data objects at the client, network traffic between client and server, and database blocks on disk. Upon detecting corruption, apply mitigating actions like retrying requests that were corrupted and overwriting corrupted data from a healthy data replica.

Periodically snapshot and backup data to alternative storage systems and develop reliable restore procedures to recover the service from a backup. It's typical to backup data every hour or every four hours. Backups are then retained for durations and granularities that reflect business needs. Hourly backups are stored in high performance storage like the Colossus distributed filesystem or Blobstore object store. Daily backups are stored in immutable media like firmware-locked hard drives for faster recovery, or tape drives for better storage cost. Restoration from firmware-locked hard drives can further reduce recovery time by restoring individual data items rather than entire datasets. Backup systems must be monitored for healthy operation and to verify the RPO.

Leverage point-in-time recovery techniques for failures at the application layer to reduce the recovery point to optimal levels for situations where the database and storage layers aren't the source of data loss or corruption. Point-in-time recovery involves retaining the database's underlying transaction logs and snapshot files so that read queries can be executed at arbitrary points in the database's history. These queries can be used to build a detailed history for a data object so it can be restored to the point right before it was corrupted or lost. This helps recover from software bugs or other failures at the application layer significantly faster than backup restoration.

Develop an end-to-end recovery process including loading backups, merging backup data with live production state, and if necessary reconfiguring servers so the service can return to normal operation. The end-to-end process must be regularly tested to verify the backups and the recovery process will actually work when the time comes.

For more on data integrity, see "Data Integrity" by Raymond Blum and Rhandeev Singh in *Site Reliability Engineering*.

Recap

In this chapter, we've covered how to defend against common failure modes:

Bad changes

- Supervise changes by monitoring with metrics and probers.
- Progressively roll out changes by creating independent and small failure domains and then changing those failure domains one at a time.
- Automatically apply safe and tested mitigations, like change rollback, once an issue has been detected.
- Deploy isolated environments for development (dev), staging, and production (prod).
- Use infrastructure-as-code (IaC) and multiparty authorization (MPA) to safely make changes to production.

Error handling

- Use soft dependencies and fail-safe alternatives.
- Return and respond to standardized error codes.

- Resource exhaustion should differentiate between server overload, service overload, and quota exhaustion.
- If retrying a request fails multiple times due to local server overload, the propagated error should be a service overload to prevent retry amplification.
- Clients must slow down sending of requests when the service is overloaded.
- Attach deadlines or timeouts to requests and check if these have been exceeded at multiple layers of the system.

Resource exhaustion

- Resource exhaustion occurs when the load exceeds the resources provisioned.
- Cascading failure is where the overload of a single component causes the load to shift and overload another component until all have failed.
- Capacity caches are those where the service depends on the efficiency gains to prevent resource exhaustion.
- Defend against resource exhaustion through cost modeling, load shedding, quotas, quality of service (QoS)/criticality, autoscaling, caching, and capacity planning.

Thundering herds and hotspots

- A thundering herd is a sudden and rapid increase in traffic typically initiated by an event. A hotspot is where a subset of servers receive a disproportionate amount of load.
- Gating is a technique for deduping equivalent operations and performing work once to satisfy multiple requests.
- Hierarchical gating and caching is a technique where gating and caching are applied at multiple points to further dedupe load reaching a backend.
- Add jitter or random delays to cache expiries to ensure cache expiration doesn't become the cause of a thundering herd.

Integrity, backup, and recovery

- Recovery point objective (RPO) is the window in which data may be permanently lost upon a failure. Recovery time objective (RTO) is the window of time the service may remain unavailable after a failure.

- Deploy probers to detect data loss or corruption.
- Generate and verify checksums to validate end-to-end integrity of data.
- Periodically snapshot and back up data to alternative storage systems.
- Leverage point-in-time recovery techniques for failures at the application layer.
- Develop and test an end-to-end recovery process.

Conclusion

In this report, you've learned to build a resilient service on the cloud by:

1. Defining objectives to ensure the service is useful and to structure design decisions
   a. Formalizing the measures (SLIs), evaluation (SLOs), and negotiation (SLAs)
   b. Structuring the failure domains and selecting the necessary redundancy
   c. Designing for scale and efficiency through horizontal and vertical scaling
   d. Planning for high-velocity evolution through frequent and gradual releases
2. Knowing the dependencies so they can be leveraged effectively to reach objectives
   a. Preferring SaaS over PaaS over IaaS for reliability and total cost of ownership
   b. Aligning regionality of the service and dependencies to align failure domains
   c. Composing dependencies so aggregate SLOs contribute to objectives
   d. Leveraging the most appropriate compute, networking, and storage options
   e. Applying resiliency best practices to administrative services in the critical path

3. Architecting the service by decomposing into components and defining the coupling
    a. Designing APIs to structure the coupling between services and components
    b. Deploying tiered architectures for simplicity or service-oriented/microservices architectures for scalability, reliability, and evolution velocity
    c. Operating synchronously for consistency and simplicity, asynchronously for decoupling and reliability, and periodically in batch for efficiency and cost
    d. Scaling horizontally to handle load and scaling vertically to increase efficiency
    e. Leveraging frameworks and patterns to apply best practices at scale
4. Avoiding common failure modes like bad changes, bad error handling, resource exhaustion, and data loss
    a. Deploying changes gradually to mitigate issues before large impact
    b. Retrying to mitigate errors, using adaptive throttling to alleviate overload, and attaching deadlines to stop unhealthy requests from executing infinitely
    c. Load shedding to prevent cascading failure and deploying quotas, criticality/QoS, autoscaling, and caching to gracefully degrade
    d. Gating requests and caching hierarchically to serve thundering herds and hotspots
    e. Verifying data integrity with checksums and developing backup and recovery to prevent data loss and recover from failures

Acknowledgments

The authors would like to thank the Googlers who reviewed and provided feedback that shaped this report including Salim Virji, Vivek Rau, Yuri Grinshteyn, Seth Vargo, Steven Ross, Christopher Heiser, and Tommy Murphy. The authors would also like to thank Casey Rosenthal, Sean P. Kane, and Virginia Wilson, who reviewed

and provided feedback as well. Finally, the authors would like to thank the Google leadership who reviewed this report and supported the team, including Marcus Mitchell and Todd Underwood.

About the Authors

Phillip Tischler graduated from Cornell University with a BS/MEng, focusing on distributed systems and autonomous robotics. He spent more than five years as a site reliability engineer (SRE) within Google working on Core Infrastructure where he was the SRE technical lead for search over shareable private data. For the last two years, Phill has been developing products for the Google Cloud Platform (GCP). He is now the technical lead and manager for GCP's Secret Manager product.

Steve McGhee graduated from University of California, Santa Barbara with a BS/MS, focusing on distributed systems. He spent more than 10 years as an SRE within Google, learning how to scale global systems in Search, YouTube, Android, and Cloud. He managed multiple engineering teams in California, Japan, and the UK. Steve left Google to help a California-based enterprise transition onto the Cloud. He then returned to Google as a solution architect to help more companies with that same transformation. He is now a reliability advocate, helping teams understand how best to build and operate world-class, reliable services.

Shylaja Nukala is a technical writing lead for Google site reliability engineering. She leads the documentation, information management, and select training efforts for SRE, Cloud, and Google engineers. She is the author of "Why SRE Documents Matter" and other external articles about SRE. Shylaja has a PhD in communication studies from Rutgers University. She joined Google in 2006 after spending seven years writing technical documentation for Epiphany and Sony.