Google Cloud

Designing Reliable Systems

Learning objectives

- Design services to meet requirements for availability, durability, and scalability.
- Implement fault-tolerant systems by avoiding single points of failure, correlated failures, and cascading failures.
- Avoid overload failures by leveraging the circuit breaker and truncated exponential backoff design patterns.
- Design resilient data storage with lazy deletion.
- Design for normal operational state, degraded operational state, and failure scenarios.
- Analyze disaster scenarios and plan, implement, and test/simulate for disaster recovery.

Reliable systems continuously provide their services. This section discusses the architecture and design of reliable systems. Design and architecture guidelines are provided for distributed systems to try to prevent cascading and correlated failures and to improve fault tolerance. Resilient storage solutions are considered, and disaster recovery strategies are introduced.
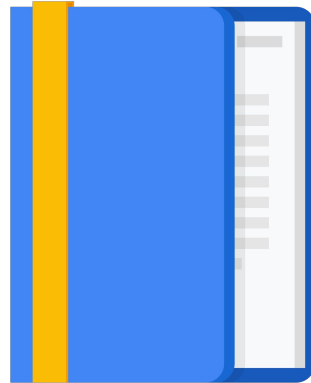
Agenda

Key Performance Metrics

Designing for Reliability

Design Activity #10

Disaster Planning

Design Activity #11

Google Cloud

When designing for reliability, consider these key performance metrics

Availability	Durability	Scalability
The percent of time a system is running and able to process requests	The odds of losing data because of a hardware or system failure	The ability of a system to continue to work as user load and data grow
- Achieved with fault tolerance. - Create backup systems. - Use health checks. - Use white box metrics to count real traffic success and failure.	- Achieved by replicating data in multiple zones. - Do regular backups. - Practice restoring from backups.	- Monitor usage. - Use capacity auto-scaling to add and remove servers in response to changes in load.

Google Cloud

Achieving reliable insights into how the software is functioning is vital. These insights are typically provided by monitoring, logging, and alerting. The performance metrics availability, durability, and scalability all contribute to the reliability of the system.

Availability
To achieve high availability, monitoring is vital. Health checks can detect when an application reports that it is ok. More detailed monitoring of services using the four golden signals from Google's Site Reliability Engineering book will help determine more insights into the health of a service. Removing single points of failure is also vital for improving availability.

Durability
Ensuring that data is preserved and available is a mixture of replication and backup. Regular restores from backup should be performed to confirm that the process works as expected.

Scalability
Monitoring and autoscaling should be used to respond to variations in load. The metrics for scaling can be the standard CPU/memory, or you can use custom metrics.
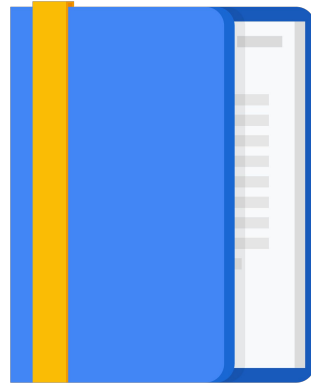
Agenda

Key Performance Metrics

Designing for Reliability

Design Activity #10

Disaster Planning

Design Activity #11

Google Cloud

Avoid single points of failure

A spare spare, N+2

- Define your unit of deployment.
- N+2: Plan to have one unit out for upgrade or testing and survive another failing.
- Make sure that each unit can handle the extra load.
- Don't make any single unit too large.
- Try to make units interchangeable stateless clones.

Single points of failure are avoided through replication of data and multi-instance deployment for compute. The recommendation in the slide depends on the service being used. It applies if you are deploying services to Compute Engine on your own managed VMs.

It is possibly not enough just to deploy N+2. The deployments should possibly be in different zones as well to mitigate for zone failure. N+2 is there for handling the upgrade case. Again, depending on the upgrade process, compute capacity can be lost. Consider 3 VMs that are load balanced (N+2). If one is being upgraded, so out of service, and another fails, then 50% of the capacity of the compute is removed, which potentially doubles the load on the remaining instance and increases the chances of that failing. This is where capacity planning is important.

Beware of correlated failures

Correlated failures occur when related items fail at the same time.

- If a single machine fails, all requests served by machine fail.
- If a top-of-rack switch fails, entire rack fails.
- If a zone or region is lost, all the resources in it fail.
- Servers on the same software run into the same issue.
- If a global configuration system fails, and multiple systems depend on it, they potentially fail too.

The group of related items that could fail together is a **failure domain.**

Google Cloud

The architecture of the application and infrastructure can have a large affect on the possibility of correlated failures. Some considerations for preventing correlated failures are discussed on the next slide.

To avoid correlated failures…

Decouple servers and use microservices distributed among multiple failure domains.

- Divide business logic into services based on failure domains.
- Deploy to multiple zones and/or regions.
- Split responsibility into components and spread over multiple processes.
- Design independent, loosely coupled but collaborating services.

Google Cloud

Architecture of the application has a big effect on the potential for correlated failures. An application that uses microservices architecture has many more points of potential failure and as a result more potential for correlated failures, so care must be taken.
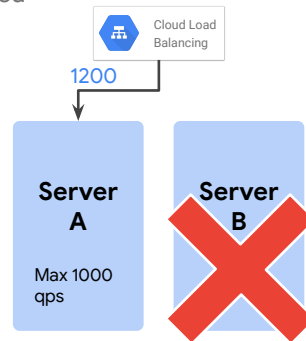
Beware of cascading failures

Cascading failures occur when one system fails, causing others to be overloaded, such as a message queue becoming overloaded because of a failing backend.

Servers A and B split the load

Cloud Load Balancing

600   600

Server A

Max 1000 qps

Server B

Max 1000 qps
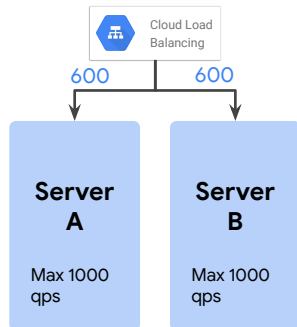
Server B fails, causing A to be overloaded

Cloud Load Balancing

1200

Server A

Max 1000 qps

Server B

Google Cloud

Cascading failures often occurs when a failure in one part of the system increases the probability that other portions of the system will fail. There are a few typical scenarios that account for the majority of cascading failures:
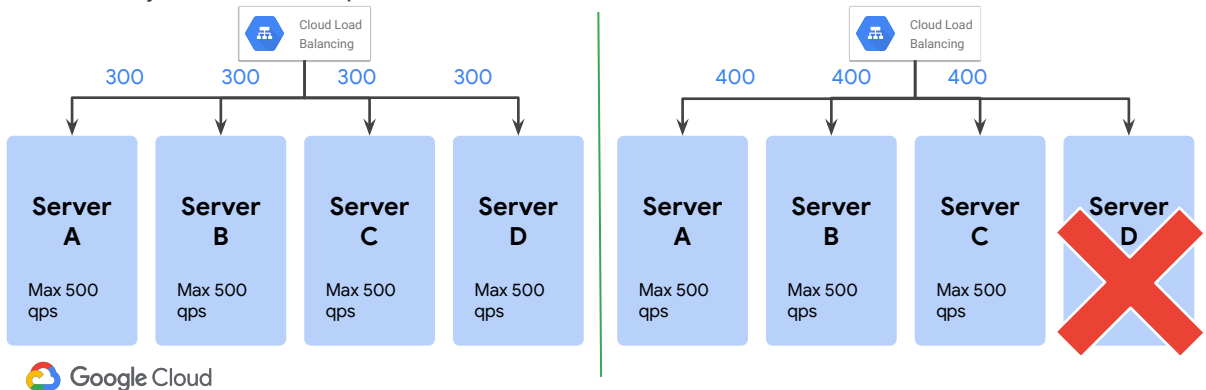
Server overload: When one server fails and others become overloaded by handling the capacity the failed server would handle under normal circumstances.

Resource exhaustion: CPU, memory, threads, and file descriptors all can cause a server to become unstable and crash.

Some strategies for preventing these are discussed on the next slide.

To avoid cascading failures

- Use health checks in Compute Engine or readiness and liveliness probes in Kubernetes to detect and then repair unhealthy instances.
- Ensure that new server instances start fast and ideally don't rely on other backends/ systems to start up.

Cloud Load Balancing

| 300 | 300 | 300 | 300 | | 400 | 400 | 400 | |

| Server A	Server B	Server C	Server D	Server A	Server B	Server C	Server D
Max 500 qps	Max 500 qps	Max 500 qps	Max 500 qps	Max 500 qps	Max 500 qps	Max 500 qps	

Google Cloud

Cascading failures may be handled by support from the deployment platform with, for example, Kubernetes and the desired state. It is always important to have an efficient startup sequence for services, and they should start up even if dependencies are not available but just not make themselves available to process requests.

Other strategies should also be considered. For instance, to prevent server overload, consider serving degraded results, load shedding, or graceful degradation. Servers should try to prevent themselves from becoming overloaded. Also consider implementing rate limiting on services to prevent overload.

The importance of good capacity planning cannot be overestimated. The load at which a service will fail should be known so that the deployment can be suitably provisioned.

Query of death overload

Problem: Business logic error shows up as overconsumption of resources, and the service overloads.
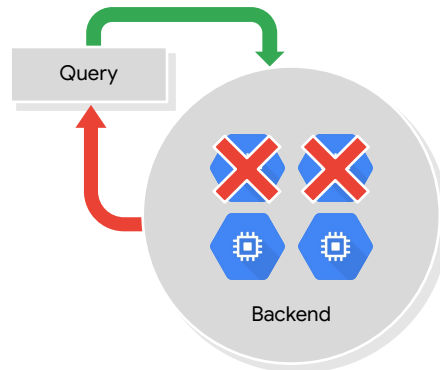
Solution: Monitor query performance. Ensure that notification of these issues gets back to the developers.

Query

Backend

Google Cloud

In the query of death, a request is made to a service that may cause a failure in the service. This is referred to as the query of death because the error manifests itself as overconsumption of resources, but in reality is due to an error in the business logic itself. This can be difficult to diagnose and requires good monitoring, observability, and logging to determine the root cause of the problem.

Positive feedback cycle overload failure

Problem: You try to make the system more reliable by adding retries, and instead you create the potential for an overload.

Solution: Prevent overload by carefully considering overload conditions whenever you are trying to improve reliability with feedback mechanisms to invoke retries.

The challenge with issuing a retry is that you do not know why the first request failed. Is it because the service you called is overloaded? If this is the case, the chances are that a retry can add more load to an already overloaded service. Not only that, your callers may start issuing retries to you because you have not replied in a given time period, and as a result trying to build in resilience actually causes system instability. If retries are to be used, some recommendations are on the next slide.

Use truncated exponential backoff pattern to avoid positive feedback overload at the client

- If service invocation fails, try again:
  - Continue to retry, but wait a while between attempts.
  - Wait a little longer each time the request fails.
  - Set a maximum length of time and a maximum number of requests.
  - Eventually, give up.
- Example:
  - Request fails; wait 1 second + random_number_milliseconds and retry.
  - Request fails; wait 2 seconds + random_number_milliseconds and retry.
  - Request fails; wait 4 seconds + random_number_milliseconds and retry.
  - And so on, up to a maximum_backoff time.
  - Continue waiting and retrying up to some maximum number of retries.

Google Cloud

Always use exponential backoff when scheduling retries. Retries should be randomly distributed over the retry window; otherwise, retries can be scheduled at the same time from different clients, which causes more load on the struggling service. See the following article for more details:
https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/.

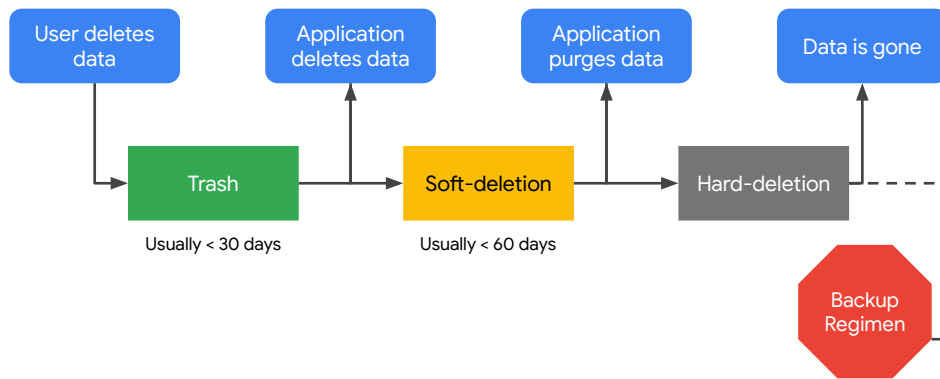# Use the circuit breaker pattern to protect the service from too many retries

- Plan for degraded state operations.
- If a service is down and all its clients are retrying, the increasing number of requests can make matters worse.
  - Protect the service behind a proxy that monitors service health (*the circuit breaker*).
  - If the service is not healthy, don't forward requests to it.
- If using GKE, leverage Istio to automatically implement circuit breakers.

The circuit breaker pattern https://martinfowler.com/bliki/CircuitBreaker.html is a recognized solution to preventing too many retries. It is built into some infrastructure; for example, Istio. Also, several libraries in different languages provide support. For instance, in Java, the Microprofile and Spring frameworks provide support and libraries such as resilience4j (https://github.com/resilience4j/resilience4j)

Similar libraries are available for many other languages.

Use lazy deletion to reliably recover when users delete data by mistake

| User deletes data | Application deletes data | Application purges data | Data is gone |

Trash — Usually < 30 days
Soft-deletion — Usually < 60 days
Hard-deletion
Backup Regimen

Google Cloud

With lazy deletion, a deletion pipeline is initiated, and the deletion progresses in phases. The first stage is that the user deletes the data but it can be restored within a predefined time period. This protects against mistakes by the user. Once the predefined period is over, the data is no longer visible to the user but has moved to the soft deletion phase. Here the data can be restored by user support or administrators. This deletion protects against mistakes in the application. After the soft deletion period, the data is deleted and is no longer available. The only way to restore the data is by whatever backups/archives were made of the data.

Review Activity 10: Designing Reliable, Scalable Applications

Refer to your Design and Process Workbook.

- Draw a diagram that depicts how you can deploy your application for high availability, scalability, and durability.
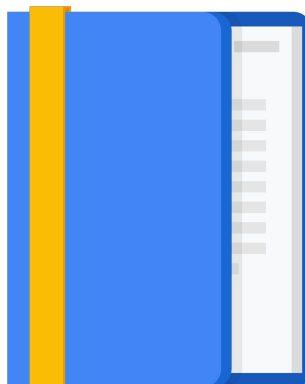

Google Cloud

Agenda

Key Performance Metrics

Designing for Reliability

Design Activity #10

Disaster Planning

Design Activity #11

Google Cloud

High availability can be achieved by deploying to multiple zones in a region

- Deploy multiple servers.
- Orchestrate servers with a regional managed instance group.
- Create a failover database in another zone or use a distributed database like Firestore or Spanner.

Google Cloud
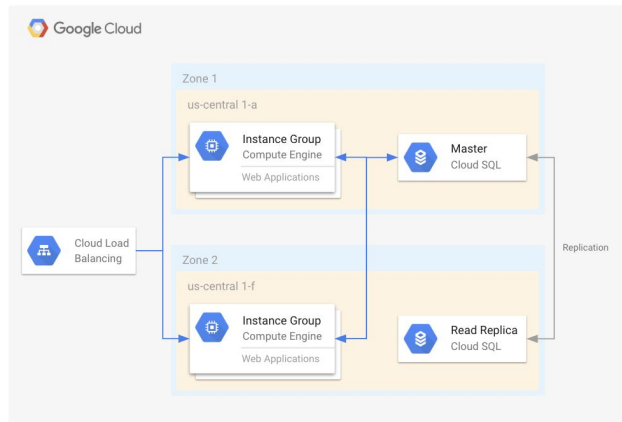
When you're using Compute Engine, for higher availability you can use a regional instance group, which provides built-in functionality to keep instances running. Use auto-healing with an application health check and load balancing to distribute load. For data, the storage solution selected will affect what is needed to achieve high availability. For Cloud SQL, the database can be configured for HA, which provides data redundancy and a standby instance of the database server in another zone.

Regional managed instances groups evenly distribute VMs across zones in the region specified

Location

To ensure higher availability, select a multiple zone location for an instance group.
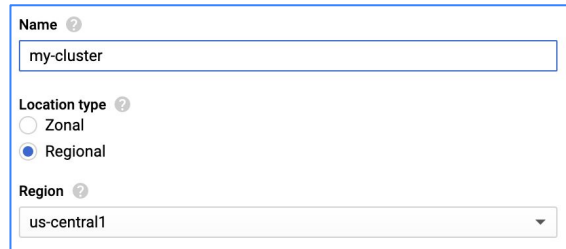Learn more

◯ Single zone

⦿ Multiple zones
Only managed instance groups can exist in multiple zones.

Google Cloud

The multiple zone instance group discussed on the previous slide ensures that VMs are evenly distributed across the available zones.

Kubernetes clusters can also be deployed to single or multiple zones

- Kubernetes cluster consist of a collection of node pools.
- Selecting Regional location type replicates node pools in multiple zones in the region specified.

Name ⊘
my-cluster

Location type ⊘
○ Zonal
● Regional

Region ⊘
us-central1   ▼

Google Cloud

It is possible to use multiple Google Kubernetes Engine clusters to host applications for better resilience, scalability, isolation, and compliance. In addition, users expect low-latency access to applications from anywhere around the world. It is possible to configure ingress using Cloud Load Balancing for multi-cluster Google Kubernetes Engine (GKE) environments. This allows the use of a Kubernetes Ingress definition to leverage Cloud Load Balancing along with multiple GKE clusters running in regions around the world, to serve traffic from the closest cluster using a single anycast IP address, taking advantage of Google Cloud's 100+ Points of Presence and global network.

For more information about how Cloud Load Balancing handles cross-region traffic, see https://cloud.google.com/kubernetes-engine/docs/how-to/multi-cluster-ingress.

Istio can also be used to integrate multiple clusters in different regions. For more details, see https://istio.io/docs/examples/platform/gke/#create-the-gke-clusters.

Create a health check when creating instance groups to enable auto healing

- Create a test endpoint in your service.
- Test endpoint needs to verify that the service is up, and also that it can communicate with dependent backend database and services.
- If health check fails, the instance group will create a new server and delete the broken one.
- Load balancers also use health checks to ensure that they send requests only to healthy instances.

Google Cloud

Name ⓘ
Name is permanent

my-health-check

Description (Optional)

Protocol   **Port** ⓘ

HTTP   ▾   80

Proxy protocol ⓘ

NONE   ▾

Request path ⓘ

/test

⌄ More

Health criteria

Define how health is determined: how often to check, how long to wait for a response, and how many successful or failed attempts are decisive

Check interval ⓘ   **Timeout** ⓘ

10   seconds   5   seconds

Healthy threshold ⓘ   **Unhealthy threshold** ⓘ

2   consecutive successes   3   consecutive failures

The test endpoint is there to indicate that the service is available and ready to accept requests. A challenge here is that if dependent services are not available, a consumer service cannot flag itself as being available, and thus can lead to a cascading failure effect.

Use multi-region storage buckets for high availability if latency impact is negligible

Bucket type	Availability	Price (us-central1)
Multi-region	99.95%	$0.026 per GB
Single region	99.90%	$0.020 per GB

Google Cloud

The multi-region availability benefit is a factor of 2—0.1% to 0.05% unavailability. But in terms of time it is not so much. 99.9% availability means 43 minutes 12 seconds per month unavailability, compared to 99.95%, which means 21 minutes 36 seconds unavailability per month. Multi-region also needs to be carefully considered if data governance is a concern.

If using Cloud SQL, create a failover replica for high availability

- Replica will be created in another zone in the same region as the database.
- Will automatically switch to the failover if the master is unavailable.
- Doubles the cost of the database.

Google Cloud
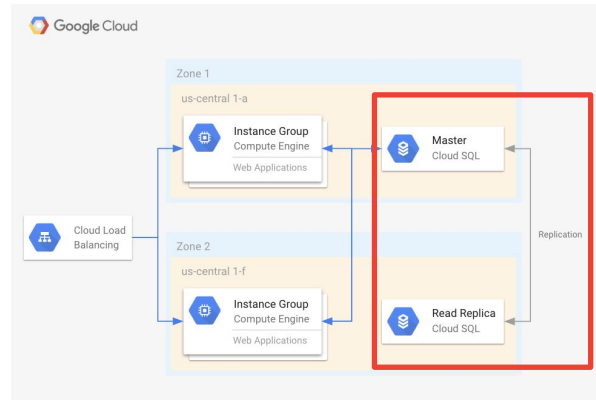
The HA configuration is available for all three Cloud SQL offerings: MySQL, PostgreSQL, and SQL Server. Full details for the configuration are here:
https://cloud.google.com/sql/docs/mysql/high-availability

The master and standby share the same IP address so that in the event of failure, clients will seamlessly be redirected to the standby.

Spanner and Firestore can be deployed in 1 or multiple regions

Database	Availability SLA
Firestore single region	99.99%
Firestore multi-region	99.999%
Spanner single region	99.99%
Spanner multi-region (nam3)	99.999%

Both Firestore and Spanner offer single and multi-region deployment. Multi-region locations can withstand the loss of entire regions and maintain availability without losing data. However, multi-region deployments also cost more money. Which you choose ties back to your application requirements and service-level objectives.

Deploying for high availability increases costs

A Risk/Cost analysis is needed.

Deployment	Estimate Cost	Availability %	Cost of being down
Single zone			
Multiple zones in a region			
Multiple regions			

The aim here is to stimulate the conversation as to how to perform this risk analysis. If you wanted to try, consider a simple deployment using GCE and a VM, and evaluate the cost of distributing this across zones and regions. How does the availability increase compared to the cost? The cost of being down is hard to determine, and much depends on the type of application, business criticality, etc.

Disaster recovery: Cold standby

- Create snapshots, machine images, and data backups in multi-region storage.
- If main region fails, spin up servers in backup region.
- Route requests to new region.
- Document and test recovery procedure regularly.

Google Cloud
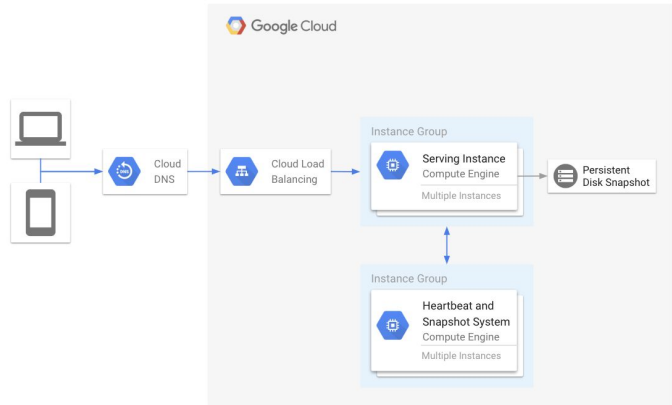
Your disaster recovery strategy depends on the business impact and the key metrics recovery time objective and recovery point objective. With a more forgiving recovery time objective, a cold standby can be acceptable. Here data is backed up, and the recovery process involves recreating the infrastructure in a different zone or region. Here infrastructure as code is a big help for automating the provisioning of a replica. This is a cost-effective solution because standby resources will only be used when needed. It is important to regularly test the ability to restore an infrastructure in order to ensure that the process for restoration and the configuration data for infrastructure are correct.

Disaster recovery: Hot standby

- Create instance groups in multiple regions.
- Use a global load balancer.
- Store unstructured data in multi-region buckets.
- For structured data, use a multi-region database such as Spanner or Firestore.
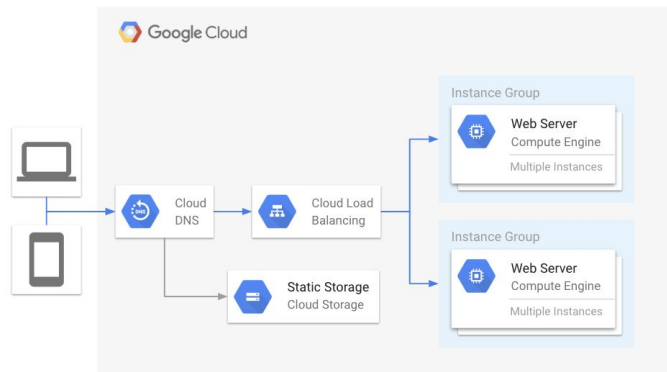
Google Cloud

The hot standby is about using Google Cloud features like instance groups deployed across multiple regions and traffic distributed using a global load balancer. Depending on the storage service, multi-region options or high availability configurations must be selected.

When disaster planning, brainstorm scenarios that might cause data loss and/or service failure

- What could happen that would cause a failure?
- What is the Recovery Point Objective (amount of data that would be acceptable to lose)?
- What is the Recovery Time Objective (amount of time it can take to be back up and running)?

Service	Scenario	Recovery Point Objective	Recovery Time Objective	Priority
Product Rating Service	Programmer deleted all ratings accidentally	24 hours	1 hour	Med
Orders service	Database server crashed	0	1 minute	High

Google Cloud

The two metrics mentioned will be major factors in determining the recovery processes that are required to be put into place. The example in the slide illustrates two services: one is critical; the other is useful but not critical.

Based on your disaster scenarios, formulate a plan to recover

- Devise a backup strategy based on risk and recovery point and time objectives.
- Communicate the procedure for recovering from failures.
- Test and validate the procedure for recovering from failures regularly.
- Ideally, recovery becomes a streamlined process, part of daily operations.

Resource	Backup Strategy	Backup Location	Recovery Procedure
Ratings MySQL database	Daily automated backups	Multi-regional Cloud Storage bucket	Run Restore Script
Orders Spanner database	Multi-region deployment	us-east1 backup region	Snapshot and backup at regular intervals, outside of the serving infrastructure; e.g. Cloud Storage

Google Cloud

Typical disaster recovery planning would need to consider requirements for capacity, security, bandwidth, and equipment, including network infrastructure. Google Cloud helps bypass most of these and helps by providing a global network, redundancy through multiple points of presence, data mirroring, scalability, security, and compliance. A useful planning guide for disaster recovery is at https://cloud.google.com/solutions/dr-scenarios-planning-guide.
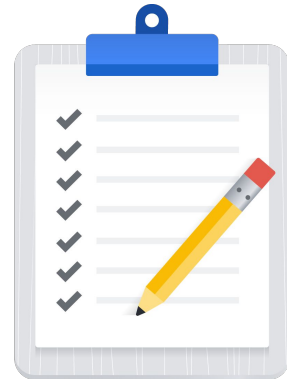
Prepare the team for disasters by using drills

Planning
- What can go wrong with your system?
- What are your plans to address each scenario?
- Document the plans.

Practice periodically
- Can be in production or a test environment as appropriate.
- Assess the risks carefully.
- Balance against the risk of not knowing your system's weaknesses.

This stage is vital and often overlooked. In particular, the periodic practice of recovery is vital to ensure that the process and procedures in place are satisfactory to achieve the recovery point objectives and the recovery time objectives. The risks should always be evaluated periodically and the processes adjusted to account for changes in risks.

Activity 11: Disaster planning

Refer to your Design and Process Workbook.

- Brainstorm disaster scenarios.
- Formulate a disaster recovery plan.

Quiz

Match the term to its definition.

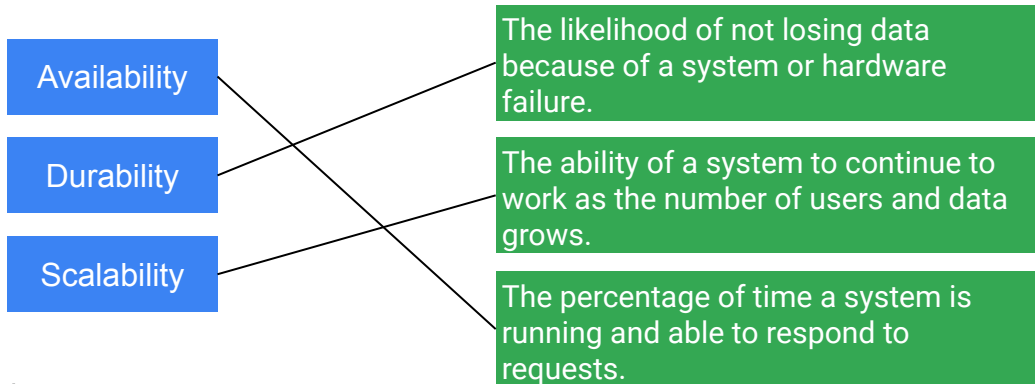|	
Availability	The likelihood of not losing data because of a system or hardware failure.
Durability	The ability of a system to continue to work as the number of users and data grows.
Scalability	The percentage of time a system is running and able to respond to requests.

Google Cloud

Match the terms availability, durability, and scalability to their definitions.

Quiz

Match the term to its definition.

Availability	The likelihood of not losing data because of a system or hardware failure.
Durability	The ability of a system to continue to work as the number of users and data grows.
Scalability	The percentage of time a system is running and able to respond to requests.

Google Cloud

These are a check on the definitions introduced on slide 4.

Availability is the percentage of time a system is running and able to respond to requests.
Durability is the likelihood of not losing data because of a system or hardware failure.
Scalability is the ability of a system to continue to work as the number of users and data grows.

Quiz

Why wouldn't you design every system for maximum availability?

Quiz

Why wouldn't you design every system for maximum availability?

Higher availability drives up system costs. You need to do a Risk/Cost analysis to determine what is appropriate for each service.

Not every system needs high availability. Some down time may be acceptable, and if a fast recovery time is achievable, the cost/benefit on optimizing recovery time may be more beneficial than aiming for higher availability. There is a cost with high availability, and higher availability usually leads to higher cost through both extra resources and added complexity in the software.

Quiz

You need a relational database for a system that requires extremely high availability (99.999%). The system must run uninterrupted even in the event of a regional outage. Which database would you choose?

A. Cloud SQL

B. Firestore

C. Spanner

D. BigQuery

Google Cloud

Quiz

You need a relational database for a system that requires extremely high availability (99.999%). The system must run uninterrupted even in the event of a regional outage. Which database would you choose?

A. Cloud SQL

B. Firestore

C. Spanner

D. BigQuery

A. This is not correct. Cloud SQL provides a relational database and can be configured for HA. However, even in such a configuration, a regional outage will cause some downtime because HA is not global.

B. This is not correct. Firestore is not a relational database.

C. This is the correct answer. Cloud Spanner meets all the requirements. It is a global relational database with high availability. Multi-regional instances have a monthly uptime of >= 99.999%.

D. This is not correct. BigQuery is not a relational database.

Quiz

You're creating a service, and you want to protect it from being overloaded by too many client retries in the event of a partial outage. Which design pattern would you implement?

A. Circuit breaker

B. Truncated exponential backoff

C. Overload feedback repudiation

D. Lazy caching

Quiz

You're creating a service and you want to protect it from being overloaded by too many client retries in the event of a partial outage. Which design pattern would you implement?

A. Circuit breaker

B. Truncated exponential backoff

C. Overload feedback repudiation

D. Lazy caching

A. This is the correct answer. The circuit breaker will attempt to prevent an operation that is likely to fail and therefore will protect the resource that is in partial outage and hopefully prevent cascading failure.

B. This is not correct. Truncated exponential backoff attempts a retry on the expectation that it is likely to succeed. This will cause problems if there is a partial outage because the retry will not succeed.

C. This is not correct. Overload feedback repudiation is shedding load to prevent services from becoming overloaded. It is normally a load balancing strategy.

D. This is not correct. Caching can reduce the load, but there is no control on when the cache is evicted/replenished, and it might not reduce load when it is needed.

Review

Designing Reliable Systems

In this module, we covered how to deploy our applications for high availability, durability, and scalability. For high availability we can deploy our resources across multiple zones and regions. For durability we can keep multiple copies of data and perform regular backups, and for scalability we can deploy multiple instances of our services and set up autscalers.

We also introduced disaster recovery planning, which is defined by coming up with the scenarios that would cause you to lose data and having a plan in place for recovery if a disaster happens.

More resources

Disaster recovery planning guide
https://cloud.google.com/solutions/dr-scenarios-planning-guide

The link provides access to a useful resources on disaster recovery planning.