



Google Cloud

Designing Reliable Systems

Stephanie Wong

Developer Advocate, Google Cloud

In this module, we talk about how to design 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.

Specifically, we'll go over how to design services to meet requirements for availability, durability, and scalability. We will also discuss how to implement fault-tolerant systems by avoiding single points of failure, correlated failures, and cascading failures. Overload can also be a challenge in distributed systems, and we will see how to avoid overload failures by using design patterns such as the circuit breaker and truncated exponential backoff.

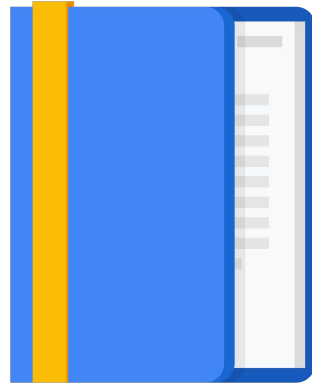
How to design resilient data storage systems with lazy deletion is introduced. When things go wrong, we want our systems to stay in control, so we discuss design decisions for different operational states including normal, degraded, and failure scenarios. Finally, we cover how to analyze disaster scenarios and plan, implement, and test for disaster recovery.

Agenda

Key Performance Metrics

Designing for Reliability

Disaster Planning



Let's start by considering the key performance metrics for reliable systems.

When designing for reliability, consider these key performance metrics

Availability	Durability	Scalability
<p>The percent of time a system is running and able to process requests</p> <ul style="list-style-type: none">• Achieved with fault tolerance.• Create backup systems.• Use health checks.• Use white box metrics to count real traffic success and failure.	<p>The odds of losing data because of a hardware or system failure</p> <ul style="list-style-type: none">• Achieved by replicating data in multiple zones.• Do regular backups.• Practice restoring from backups.	<p>The ability of a system to continue to work as user load and data grow</p> <ul style="list-style-type: none">• Monitor usage.• Use capacity auto-scaling to add and remove servers in response to changes in load.

When designing for reliability, consider availability, durability, and scalability as the key performance metrics. Let me explain each of these:

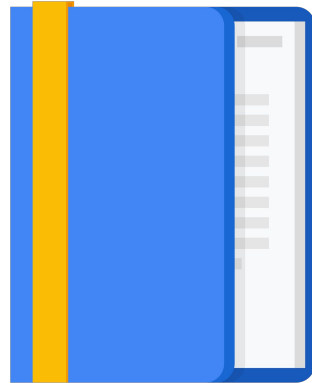
- Availability is the percent of time a system is running and able to process requests. To achieve high availability, monitoring is vital. Health checks can detect when an application reports that it is ok. More detailed monitoring of services using white box metrics to count traffic success and failures will help predict problems. Building in fault tolerance by, for example, removing single points of failure is also vital for improving availability. Backup systems also play a key role in improving availability.
- Durability is the chance of losing data because of hardware or system failure. Ensuring that data is preserved and available is a mixture of replication and backup. Data can be replicated in multiple zones. Regular restores from backup should be performed to confirm that the process works as expected.
- Scalability is the ability of a system to continue to work as user load and data grow. Monitoring and autoscaling should be used to respond to variations in load. The metrics for scaling can be the standard metrics like CPU or memory, or you can create custom metrics like “number of players on a game server.”

Agenda

Key Performance Metrics

Designing for Reliability

Disaster Planning



Now that we've covered the key performance metrics, let's design for reliability.

Avoid single points of failure

A spare spare, N+2

- Define your unit of deployment
- N+2L: 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

Avoid single points of failure by replicating data and creating multiple virtual machine instances. It is important to define your unit of deployment and understand its capabilities. To avoid single points of failure, you should deploy two extra instances, or $N + 2$, to handle both failure and upgrades. These deployments should ideally be in different zones to mitigate for zonal failures.

Let me explain the upgrade consideration: Consider 3 VMs that are load balanced to achieve N+2. If one is being upgraded and another fails, 50% of the available 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 and knowing the capability of your deployment unit is important. Also, for ease of scaling, it is a good practice to make the deployment units interchangeable stateless clones.

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**.

It is also important to be aware of correlated failures. These occur when related items fail at the same time. At the simplest level, if a single machine fails, all requests served by that machine fail. At a hardware level, if a top-of-rack switch fails, the complete rack fails. At the cloud level, if a zone or region is lost, all the resources are unavailable. Servers running the same software suffer from the same issue: if there is a fault in the software, the servers may fail at a similar time.

Correlated failures can also apply to configuration data. If a global configuration system fails, and multiple systems depend on it, they potentially fail too. When we have a group of related items that could fail together, we refer to it as a *failure* or *fault* domain.

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.

Several techniques can be used to avoid correlated failures. It is useful to be aware of failure domains; then servers can be decoupled using microservices distributed among multiple failure domains. To achieve this, you can divide business logic into services based on failure domains and deploy to multiple zones and/or regions.

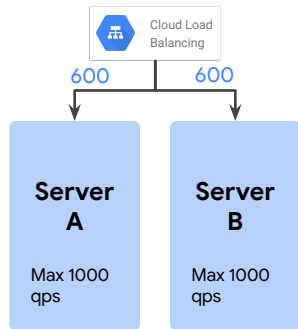
At a finer level of granularity, it is good to split responsibilities into components and spread these over multiple processes. This way a failure in one component will not affect other components. If all responsibilities are in one component, a failure of one responsibility has a high likelihood of causing all responsibilities to fail.

When you design microservices, your design should result in loosely coupled, independent but collaborating services. A failure in one service should not cause a failure in another service. It may cause a collaborating service to have reduced capacity or not be able to fully process its workflows, but the collaborating service remains in control and does not fail.

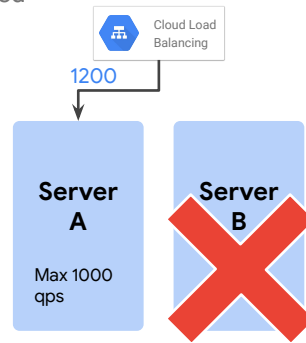
Beware of cascading failures

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



Server B fails, causing A to be overloaded



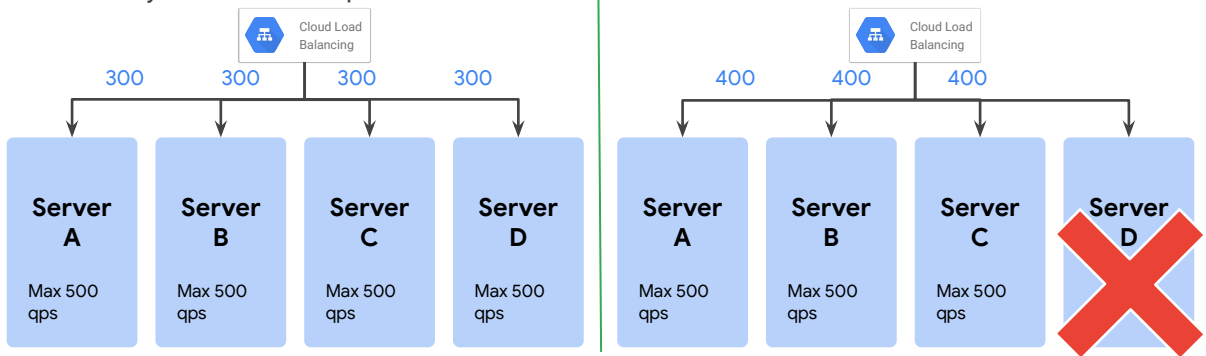
Cascading failures occur when one system fails, causing others to be overloaded and subsequently fail. For example, a message queue could be overloaded because a backend fails and it cannot process messages placed on the queue.

The graphic on the left shows a Cloud Load Balancer distributing load across two backend servers. Each server can handle a maximum of 1000 queries per second. The load balancer is currently sending 600 queries per second to each instance. If server B now fails, all 1200 queries per second have to be sent to just server A, as shown on the right. This is much higher than the specified maximum and could lead to cascading failure.

So, how do we avoid cascading failures?

To avoid cascading failures

- Use health checks in Compute Engine or readiness and liveness 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.

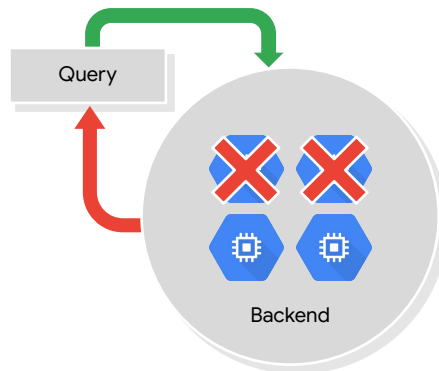


Cascading failures can be handled with support from the deployment platform. For example, you can use health checks in Compute Engine or readiness and liveness probes in GKE to enable the detection and repair of unhealthy instances. You want to ensure that new instances start fast and ideally do not rely on other backends/systems to start up before they are ready.

The graphic on this slides illustrates a deployment with four servers behind a load balancer. Based on the current traffic, a server failure can be absorbed by the remaining three servers, as shown on the right-hand side. If the system uses Compute Engine with instance groups and autohealing, the failed server would be replaced with a new instance. As I just mentioned, it's important for that new server to start up quickly to restore full capacity as quickly as possible. Also, this setup only works for stateless services.

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.

You also want to plan against query of death, where a request made to a service causes 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. When the requests are made, latency, resource utilization, and error rates should be monitored to help identify 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.



You should also plan against positive feedback cycle overload failure, where a problem is caused by trying to prevent problems!

This happens when you try to make the system more reliable by adding retries in the event of a failure. Instead of fixing the failure, this creates the potential for overload. You may actually be adding more load to an already overloaded system.

The solution is intelligent retries that make use of feedback from the service that is failing. Let me discuss two strategies to address this.

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.

If a service fails, it is ok to try again. However, this must be done in a controlled manner. One way is to use the exponential backoff pattern. This performs a retry, but not immediately. You should wait between retry attempts, waiting a little longer each time a request fails, therefore giving the failing service time to recover. The number of retries should be limited to a maximum, and the length of time before giving up should also be limited.

As an example, consider a failed request to a service. Using exponential backoff, we may wait 1 second plus a random number of milliseconds and try again. If the request fails again, we wait 2 seconds plus a random number of milliseconds and try again. Fail again, then wait 4 seconds plus a random number of milliseconds before retrying, and continue until a maximum limit is reached.

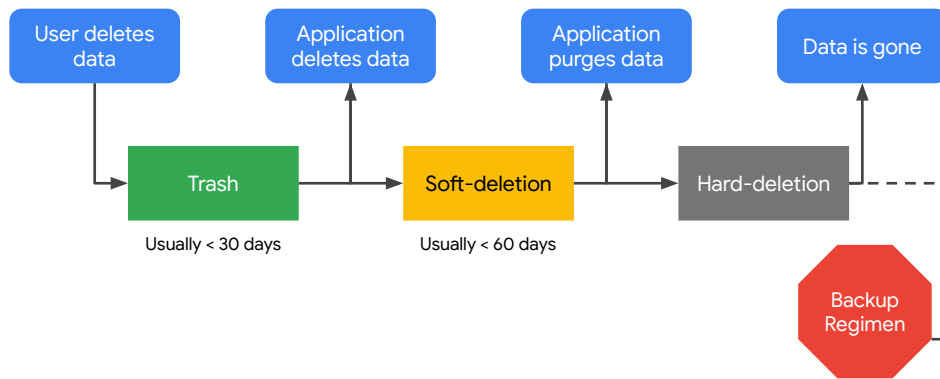
Use the circuit breaker pattern to protect the service from too many retries

- 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 can also protect a service from too many retries. The pattern implements a solution for when a service is in a degraded state of operation. It is important because if a service is down or overloaded and all its clients are retrying, the extra requests actually make matters worse. The circuit breaker design pattern protects the service behind a proxy that monitors the service health. If the service is not deemed healthy by the circuit breaker, it will not forward requests to the service. When the service becomes operational again, the circuit breaker will begin feeding requests to it again in a controlled manner.

If you are using GKE, the Istio service mesh automatically implements circuit breakers.

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



Lazy deletion is a method that builds in the ability to reliably recover data when a user deletes the data by mistake. With lazy deletion, a deletion pipeline similar to that shown in this graphic 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; in this example it is 30 days. This protects against mistakes by the user.

When the predefined period is over, the data is no longer visible to the user but moves 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 of 15, 30, 45, or even 50 days, 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.

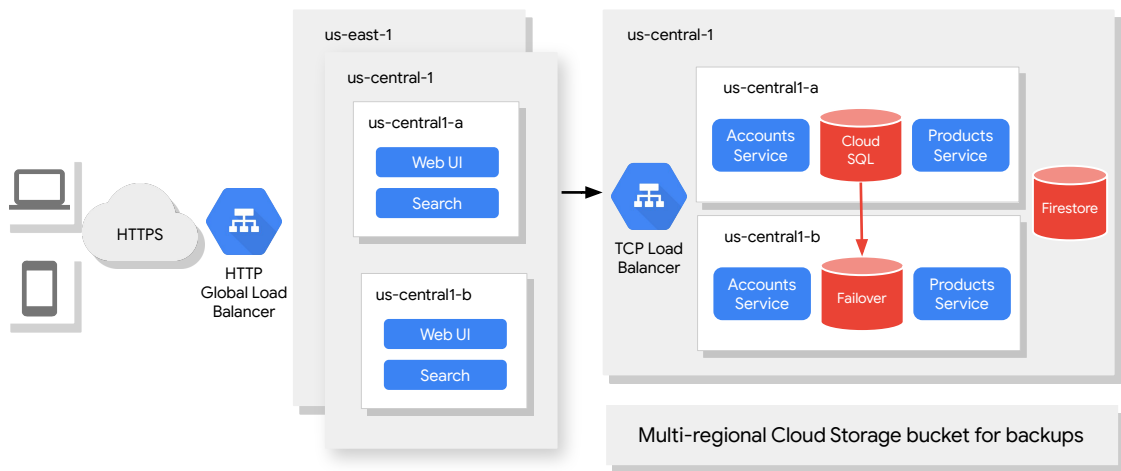
Review Activity 10: Designing Reliable, Scalable Applications

Refer to your Design and Process Workbook.

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



In this design activity, you draw a diagram that depicts how you can deploy your application for high availability, scalability, and durability. Let me show you an example of what to draw.



This diagram is for an online bank application with customers in the United States.

I want the web UI to be highly available, so here I depict it as being deployed behind a global HTTP Load Balancer across multiple regions and multiple zones within each region. I chose us-central-1 as the main region because it's somewhat in the middle of the US. I also have a backup region in us-east-1, which is on the east coast of the US.

I deploy the accounts and products services as backends to just the us-central1 region, but I'm using multiple zones (us-central1-a and us-central1-b) for high availability. I even have a failover Cloud SQL database. The Firestore database for the products service is multi-regional, so I don't need to worry about a failover.

In case of a disaster, I'll keep backups in a multi-regional Cloud Storage bucket. That way, if there is a regional outage, I can restore in another region.

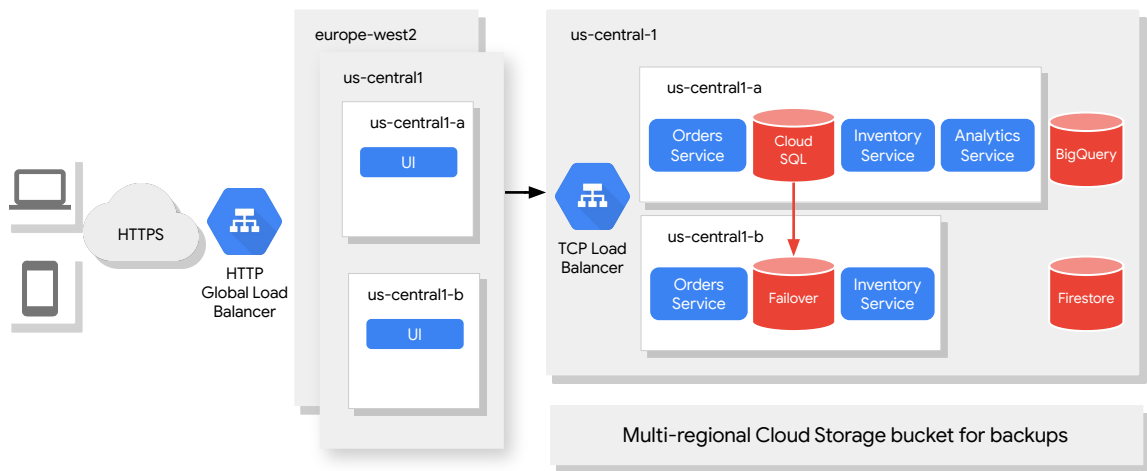
Refer to activity 10 in your workbook to create a similar diagram for your services.

Review Activity 10: Designing Reliable, Scalable Applications

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



In this activity, you were tasked to draw a diagram that depicts how your application can be deployed for high availability, scalability, and durability.



For our online travel application, ClickTravel, I'm assuming that this is an American company, but that I have a large group of customers in Europe.

I want the UI to be highly available, so I've placed it into us-central1 and europe-west2 behind a global HTTP Load Balancer. This load balancer will send user requests to the region closest to the user, unless that region cannot handle the traffic.

I could also deploy the backends globally but I'm trying to optimize cost. I could start by just deploying those in us-central1. This will create latency for our European users but I can always revisit this later and have a similar backend in europe-west2.

To ensure high availability, I've decided to deploy the Orders and Inventory services to multiple zones. Because the Analytics service is not customer-facing, I can save some money by deploying it to a single zone. I again have a failover Cloud SQL database, and the Firestore database and BigQuery data warehouse are multi-regional, so I don't need to worry about a failover for those.

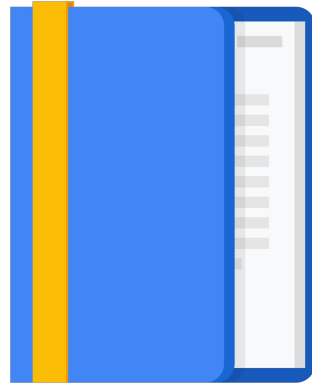
In case of a disaster, I'll keep backups in a multi-regional Cloud Storage bucket. That way if there is a regional outage, I can restore in another region.

Agenda

Key Performance Metrics

Designing for Reliability

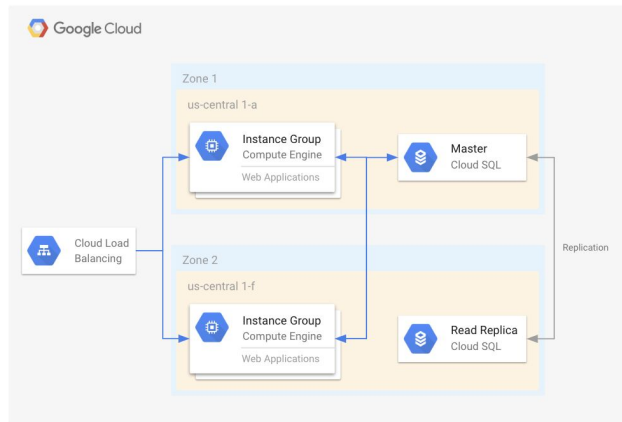
Disaster Planning



Now that we've designed for reliability, let's explore disaster planning.

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.



High availability can be achieved by deploying to multiple zones in a region. When 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 high availability, which provides data redundancy and a standby instance of the database server in another zone. This diagram shows a high availability configuration with a regional managed instance group for a web application that's behind a load balancer. The master Cloud SQL instance is in us-central1-a, with a replica instance in us-central1-f.

Some data services such as Firestore or Spanner provide high availability by default.

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

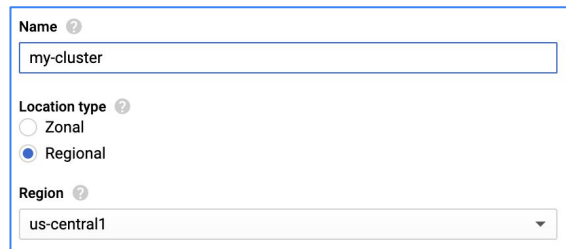
☒ Multiple zones

Only managed instance groups can exist in multiple zones.

In the previous example, the regional managed instance group distributed VMs across zones. You can choose between single zones and multiple zones (or regional) configurations when creating your instance group, as you can see in this screenshot.

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.



The screenshot shows a configuration form for a Kubernetes cluster. It has three main sections: 'Name' with a text input field containing 'my-cluster'; 'Location type' with two radio button options, 'Zonal' and 'Regional', where 'Regional' is selected; and 'Region' with a dropdown menu showing 'us-central1'.

Name ?
my-cluster
Location type ?
<input type="radio"/> Zonal
<input checked="" type="radio"/> Regional
Region ?
us-central1

Google Kubernetes Engine clusters can also be deployed to either a single or multiple zones, as shown in this screenshot.

A cluster consists of a master controller and collections of node pools. Regional clusters increase the availability of both a cluster's master and its nodes by replicating them across multiple zones of a region.

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.

The screenshot shows the configuration for a health check in the Google Cloud Platform console. The form includes the following fields and settings:

- Name:** my-health-check (Note: Name is permanent)
- Description:** (Optional, empty text box)
- Protocol:** HTTP (dropdown menu)
- Port:** 80 (text box)
- Proxy protocol:** NONE (dropdown menu)
- Request path:** /test (text box)
- More:** (expandable section)
- 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:** 10 seconds
 - Timeout:** 5 seconds
 - Healthy threshold:** 2 consecutive successes
 - Unhealthy threshold:** 3 consecutive failures

If you are using instance groups for your service, you should create a health check to enable auto healing.

The health check is a test endpoint in your service. It should indicate that your service is available and ready to accept requests, and not just that the server is running. A challenge with creating a good health check endpoint is that if you use other backend services, you need to check that they are available to provide positive confirmation that your service is ready to run. If the services it is dependent on are not available, it should not be available.

If a health check fails the instance group, it will remove the failing instance and create a new one. Health checks can also be used by load balancers to determine which instances to send requests to.

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

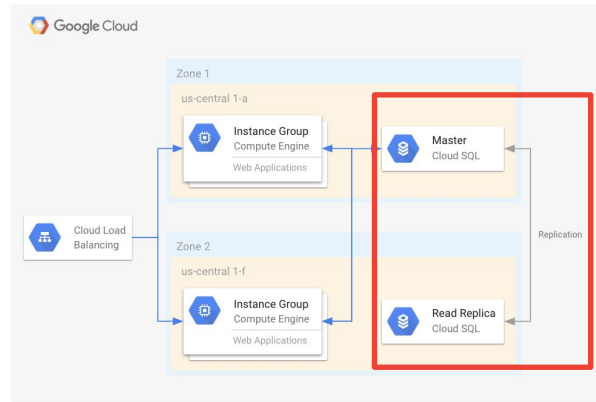
Bucket type	Availability	Price (us-central1)
Multi-region	99.95%	\$0.026 per GB
Single region	99.90%	\$0.020 per GB

Let's go over how to achieve high availability for Google Cloud's data storage and database services.

For Cloud Storage, you can achieve high availability with multi-region storage buckets if the latency impact is negligible. As this table illustrates, the multi-region availability benefit is a factor of two as the unavailability decreases from 0.1% to 0.05%.

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.



If you are using Cloud SQL and need high availability, you can create a failover replica. This graphic shows the configuration where a master is configured in one zone and a replica is created in another zone but in the same region. If the master is unavailable, the failover will automatically be switched to take over the master. Remember that you are paying for the extra instance with this design.

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%

Firestore and Spanner both offer single and multi-region deployments. A multi-region location is a general geographical area, such as the United States. Data in a multi-region location is replicated in multiple regions. Within a region, data is replicated across zones.

Multi-region locations can withstand the loss of entire regions and maintain availability without losing data. The multi-region configurations for both Firestore and Spanner offer “five nines” availability, which is less than 6 minutes of downtime per year.

Deploying for high availability increases costs

A Risk/Cost analysis is needed.

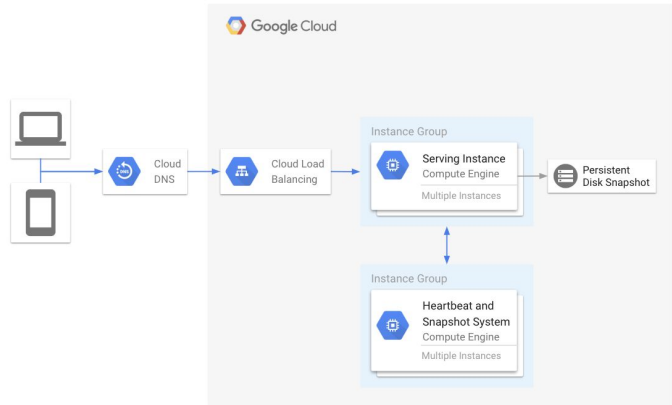
Deployment	Estimate Cost	Availability %	Cost of being down
Single zone			
Multiple zones in a region			
Multiple regions			

Now, I already mentioned that deploying for high availability increases costs because extra resources are used.

It is important that you consider the costs of your architectural decisions as part of your design process. Don't just estimate the cost of the resources used, but also consider the cost of your service being down. This table shown is a really effective way of assessing the risk versus cost by considering the different deployment options and balancing them against the cost of being down.

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.

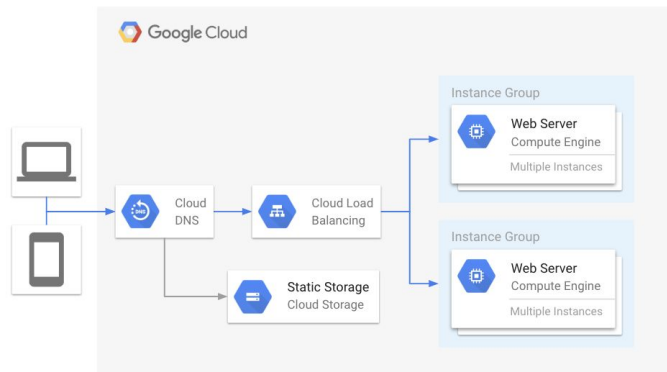


Now, let me introduce some disaster recovery strategies.

A simple disaster recovery strategy may be to have a cold standby. You should create snapshots of persistent disks, machine images, and data backups and store them in multi-region storage. This diagram shows a simple system using this strategy. Snapshots are taken that can be used to recreate the system. If the main region fails, you can spin up servers in the backup region using the snapshot images and persistent disks. You will have to route requests to the new region, and it's vital to document and test this recovery procedure regularly.

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.



Another disaster recovery strategy is to have a hot standby where instance groups exist in multiple regions and traffic is forwarded with a global load balancer. This diagram shows such a configuration.

I already mentioned this, but you can also implement this for data storage services like multi-regional Cloud Storage buckets and database services like Spanner and Firestore.

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

Now, any disaster recovery plan should consider its aims in terms of two metrics: the recovery point objective and the recovery time objective. The recovery point objective is the amount of data that would be acceptable to lose, and the recovery time objective is how long it can take to be back up and running.

You should brainstorm scenarios that might cause data loss or service failures and build a table similar to the one shown here. This can be helpful to provide structure on the different scenarios and to prioritize them accordingly. You will create a table like this in the upcoming design activity along with a recovery plan.

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

You should create a plan for how to recover based on the disaster scenarios that you define.

For each scenario, devise a strategy based on the risk and recovery point and time objectives. This isn't something that you want to simply document and leave. You should communicate the process for recovering from failures to all parties.

The procedures should be tested and validated regularly, at least once per year, and ideally recovery becomes part of daily operations, which helps streamline the process. This table illustrates the backup strategy for different resources along with the location of the backups and the recovery procedure.

This simplified view illustrates the type of information that you should capture.

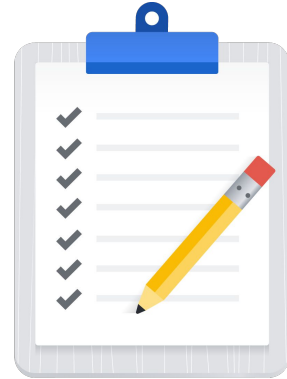
Prepare the team for disasters by using drills

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.



Before we get into our next design activity, I just want to emphasize how important it is to prepare a team for disaster by using drills.

Here you decide what you think can go wrong with your system. Think about the plans for addressing each scenario and document these plans. Then practice these plans periodically in either a test or a production environment.

At each stage, assess the risks carefully and balance the costs of availability against the cost of unavailability. The cost of unavailability will help you evaluate the risk of not knowing the system's weaknesses.

Activity 11: Disaster planning

Refer to your Design and Process Workbook.

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



In this design activity, you brainstorm disaster scenarios for your case study and formulate a disaster recovery plan.

Service	Scenario	Recovery Point Objective	Recovery Time Objective	Priority
Product Rating Service	Programmer deletes all ratings accidentally	24 hours	1 hour	Med
Orders Service	Orders database crashes	0 (can't lose any data)	2 minutes	High

Here's an example of such a brainstorming activity. The purpose of this activity is to think of disaster scenarios and assess how severely they would impact your services.

The recovery point objective represents the maximum amount of data you would be willing to lose. Obviously, this would be different for your orders database, where you wouldn't want to lose any data, and your products ratings database, where you could tolerate some loss. The recovery time objective represents how long it would take to recover from a disaster. In this example, we are estimating that we could recover our product rating service database from a backup in an hour, and our orders service within 2 minutes.

Now, like all things in life, we should prioritize. There's never time to get everything done, so work on the highest priority items first.

Resource	Backup Strategy	Backup Location	Recovery Procedure
<i>Ratings Database</i>	<i>Daily automated backups</i>	<i>Multi-Regional Cloud Storage Bucket</i>	<i>Run Restore Script</i>
<i>Orders Database</i>	<i>Failover replica plus daily backups</i>	<i>Multi-zone deployment</i>	<i>Automated</i>

Once you've identified these scenarios, come up with plans for recovery. These plans will be different based on your recovery point and recovery time objectives, as well as the priorities.

In this example, performing daily backups of our ratings database is good enough to meet our objectives. For the orders database that won't be adequate, so we'll implement a failover replica in another zone.

Refer to activities 11a, b and c in your design workbook to document similar disaster scenarios for your case study and formulate a disaster recovery plan for each.

Review Activity 11: Disaster planning

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



In this activity, you were asked to come up with disaster recovery scenarios and plans for the case study you've been working on.

Service	Scenario	Recovery Point Objective	Recovery Time Objective	Priority
Inventory Firestore	Programmer deleted all inventory accidentally	1 hours	1 hour	High
Orders Cloud SQL database	Orders database crashed	0 mins	5 minutes	High
Analytics BigQuery database	User deletes tables	0 mins	24 hours	Med

Here's an example of some disaster scenarios for our online travel portal, ClickTravel.

Each of our services uses different database services and has different objectives and priorities. All of that affects how we design our disaster recovery plans.

Resource	Backup Strategy	Backup Location	Recovery Procedure
Inventory Firestore	Daily automated backups	Multi-Regional Cloud Storage Bucket	Cloud Functions and Cloud Scheduler
Orders Cloud SQL database	Binary logging and backups Failover replica in another zone	NA	Automated failover Run backup script if needed
Analytics BigQuery database	No specific backup required	NA	Re-import data to rebuild analytics tables

Our Analytics service in BigQuery had the lowest priority; therefore, we should be able to re-import data to rebuild analytics tables if a user deletes them.

Our Orders service can't tolerate any data loss and has to be up and running almost immediately. For this we need a failover replica in addition to binary logging and automated backups.

Our Inventory service uses Firestore, and for that we can implement daily automated backups to a multi-regional Cloud Storage bucket. Cloud Functions and Cloud Scheduler can help with the recovery procedure.

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 autoscalers.

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.