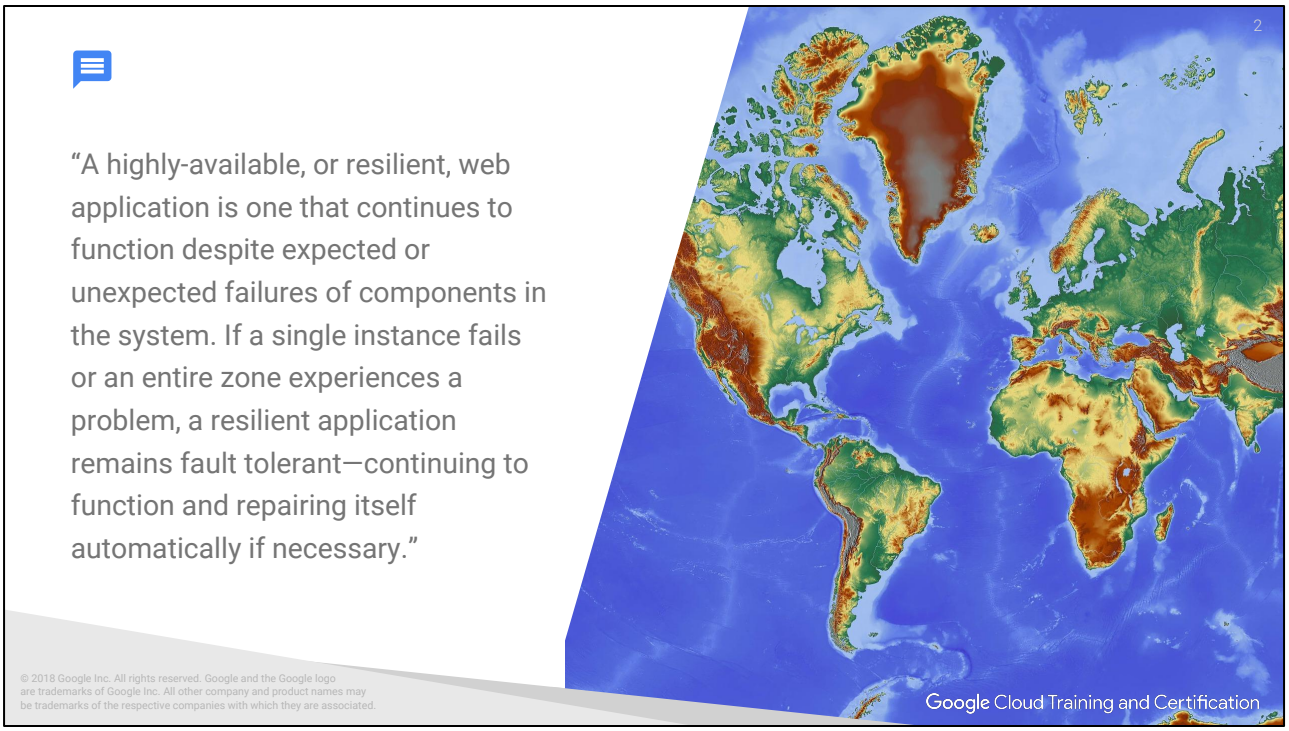
Design for resiliency, scalability, and disaster recovery

Architecting with Google Cloud Platform: Design and Process

Last modified 2018-08-08



https://cloud.google.com/solutions/scalable-and-resilient-apps

https://pixabay.com/en/map-map-of-the-world-relief-map-221210/

Designing an application architecture for a resilient application typically involves:

- Load balancers to monitor servers and distribute traffic to servers that can best handle the requests
- Hosting servers in multiple data centers
- Configuring a robust storage solution

Reliability is the outcome cloud service providers strive for – it's the result.
Resiliency is the ability of a cloud-based service to withstand certain types of failure and yet remain functional from the customer perspective. In other words, reliability is the outcome and resilience is the way you achieve the outcome.

Agenda

Failure due to loss

Failure due to overload

Coping with failure

Business continuity and disaster recovery (DR)

Scalable and resilient design

Photo service: Out of service!

Design challenge #4: Redesign for time

No GCP lab in this module

Resilience is the ability to handle issues gracefully, to remain reliable and consistent during problems or failures, and to bounce back to full service after.

Failure due to loss

Failure due to loss of resources required by the service.

Single points of failure

Correlated failures

Coping with failure is a path of reasoning that leads to distributed consensus.
Handling failure : load balancing, replication

Failure is not optional, it is mandatory

Hardware will fail!

Software will fail!

People will fail!

Communications will fail!

Hardware: Don't depend on it.

Software: Modularize, monitor, test, canary.

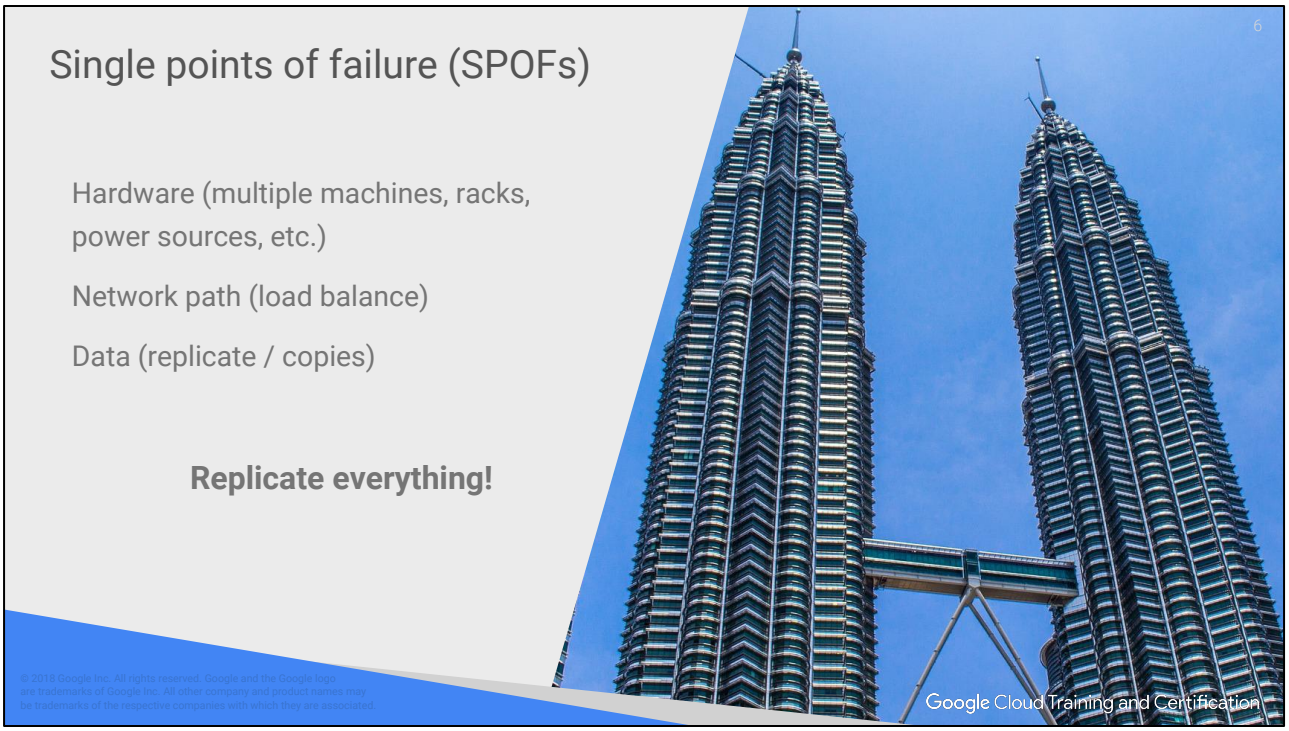
People: Review procedures, escalate.

Communication: Don't assume it happened.

The Internet is designed to route around failure.

Anticipate failure, design for failure, and fail gracefully

As your service grows, what will start to fail and how will that show itself?
What failure modes can you prevent before they start?
How can you keep serving and growing your service in spite of failures?
What is likely to fail due to scaling?



Single Points of Failure = SPOF

One example of symmetric design.

https://pixabay.com/en/kuala-lumpur-petronas-towers-kl-1609699/
https://en.wikipedia.org/wiki/Petronas_Towers

Design to avoid SPOFs

A spare spare, N+2

- N+2: Plan to have one unit out for upgrade and survive another failing.
- Make sure that each unit can handle the extra load.
- Don’t make any single unit too large.
- Don't concentrate responsibility into a single process or server
- Try to make units interchangeable clones

fun·gi·ble: Units of compute (process, server) should be able to replace or be replaced by another identical item; mutually interchangeable

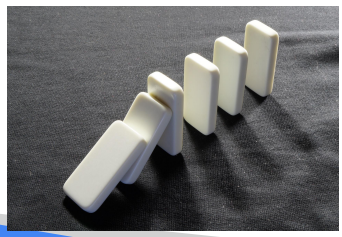
"If you are worried about what happens to a single server, something needs to be corrected in your design."

Correlated failures

Correlated failures occur when related items fail at the same time

- Single machine fails, all requests served by machine fails
- Top-of-rack fails, entire rack fails
- A zone or region is lost, all the resources in it fail
- Servers on the same software run into the same issue
- A global configuration system upon which multiple systems depend is lost

The group of related items that could fail at one time is a **failure domain**



A set of things that might experience correlated failure

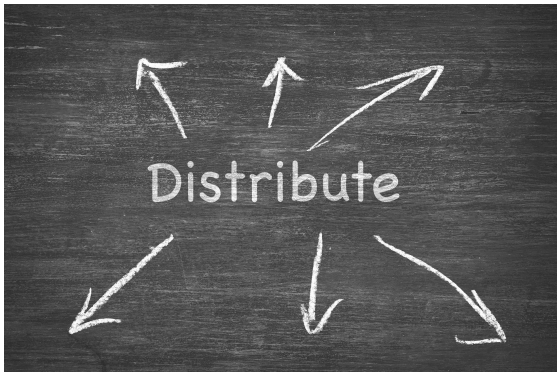
A single process
A machine
A rack
A datacenter
Servers on the same software
A global configuration system

https://pixabay.com/en/dominoes-fall-fall-over-game-row-719199/

Design to avoid correlated failures

Decouple servers, use microservices

- Divide business logic into services based on failure domains
- Split responsibility into components and spread over multiple processes
- Separate and isolate the risks
- Design independent but collaborating services



Identify where failures could occur in your design, and separate risk into independent but collaborating domains.
https://cloud.google.com/docs/tutorials#architecture

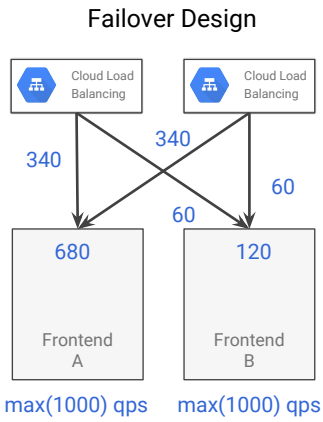
Imagine business function "A". Maybe "A" consists of reading a document, processing it, and writing it out. From a logic perspective, there is no reason that the activity of read-process-write could not be atomic. It could be handled by a single server, a single process, a single thread. But imagine for a moment that the failure risk in reading the document is different from and separate from the failure risk in processing the document and the failure risk of writing the document. When reading, the network might drop preventing access to the source document. When processing, the code might have a bug causing the process to crash. When writing, the storage might be full or the storage service might be slow. In this case you could conceive of three VMs, one handling read-and-queue, one pulling items from the queue and processing them and placing them on an output queue, and one pulling items from the output queue and writing them to storage. NOTE: The logic remains the same. What has changed is the implementation into separate domains or services. This separation buys reliability and scalability and potentially replaceability of parts.

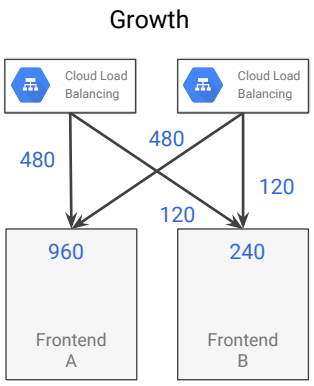
https://pixabay.com/en/delegate-board-applying-empower-1971162/

Failure due to overload

Overload failure is when a resource crosses into non-linear behavior. It can crash, thrash disk, stop responding, or break adjacent resources that the service depends on.

Failover design for reliability

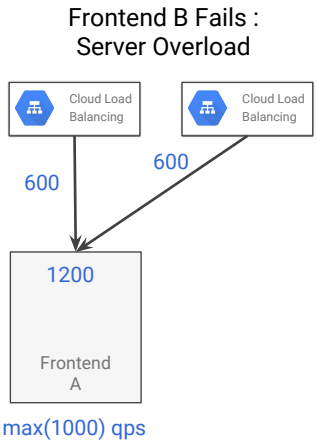




In the original design, there are two clusters A and B to provide reliability and scale for dealing with bursty demand. Each cluster is capable of 1000 queries per second. Cluster A has a load of about 680 qps and Cluster B has a load of 120. That's 800, which is 20% below the capacity of a single cluster.

Now growth occurs. Cluster A has a load of 960 and cluster B has a load of 240. The combined load is now 1200, which is below the aggregated capacity of 2000 for both A and B.

Cascading failure



CPU
Increase number of in-flight requests
Excessively long queue lengths
Thread starvation
CPU or request starvation
Missed RPC deadlines
Reduced CPU caching / lower CPU efficiency

MEMORY
Dying tasks
Increased rate of garbage collection
Reduction in cache hit rates

SERVICE DEPENDENCIES
High garbage collection in front end taxes CPU
Front end runs out of CPU
CPU exhaustion slows completion requests
Increased in-progress requests consumes RAM
Less RAM available for caching
Reduced cache size means lower hit rate
Cache misses tax the backend causing requests to fail
Back end runs out of CPU or threads
Basic health check fails … ***cascading failure begins***

System seeks resources to compensate, spreads the problems

Now one of the clusters, B, fails. The full load of 1200 is now on cluster A, that only has resources to support 1000.
The list on the right is typically what happens as the service faces overload and begins to fail. The overload spreads from the frontend server to other parts of the service, and parts begin to overload and work against other parts, until the system itself loses integrity.

Design to prevent cascading failure

This is a case where prevention is the best strategy.

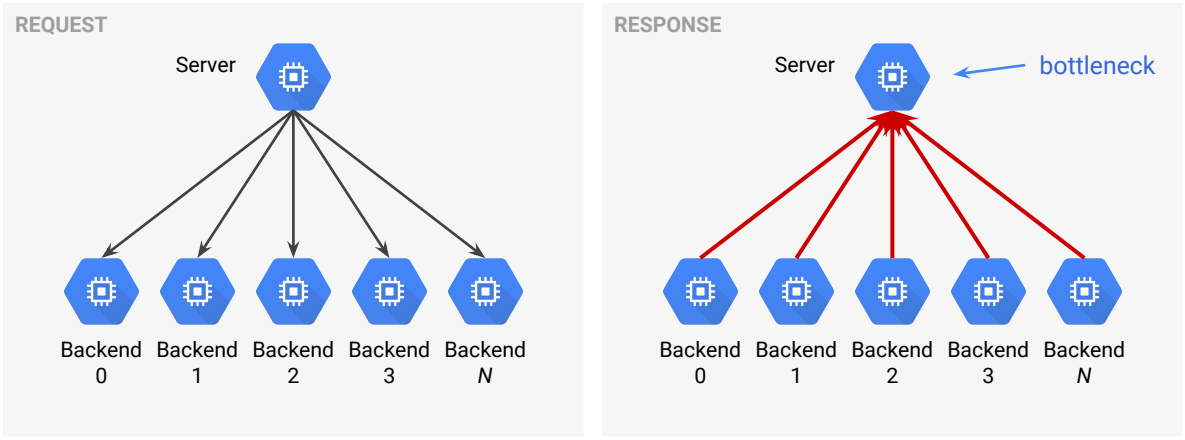
Monitor "safety" size.

Increase size for failover and not just operating capacity.

For example: No single frontend should have more than 80% CPU utilization.

- Note that CPU utilization is an internal metric, not an SLI or SLO, because it is not directly visible to the users.
- This design requirement is supported by the existing availability and performance SLOs, which are observable by the users.

Fan-in (incast) overload failure



So far, this course has referred to scaling horizontally.

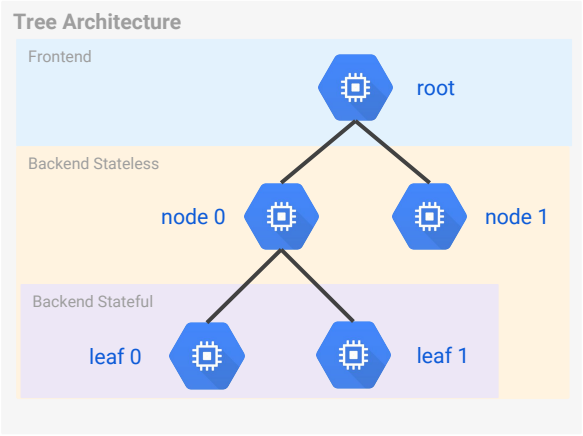
However, there could be an issue: fan-in (incast)

- Responses from thousands of backends overwhelm the server
- The server becomes a bottleneck (CPU, network, latency, etc.)

This could occur with a frontend server or with an intermediate server that performs work within the backend.

The ratio of backend servers can become an issue at scale. Consider a media library: the incoming message might be a simple request with an ID, but the response might be much larger than the request. Consequently, the server (frontend) becomes overloaded by the backend responses.

Design to mitigate incast overload

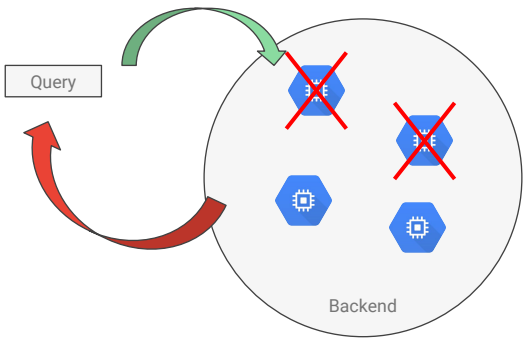


- Tree architecture limits single-server fan-in.
- Leaves generate data and forward to parent node.
- Nodes combine data from children and send merged response to parent node
- Cost is distributed and bottlenecks are reduced.

Another way to deal with incast is to simply make sure that the frontend is always provisioned to handle the worst-case replies from the backend.
Minor latencies can be introduced with this design, but it overcomes the issue of a single server becoming a bottleneck.

Queries of death overload failure

Problem: Business logic error shows up as overconsumption of resources, and the service overloads.



Solution: Monitor query performance iteratively. Ensure notification of these issues gets back to the developers.

A single query might trigger a bug in the backend servers, or the application might have an unanticipated and poisonous traffic mix. The bug might cause immediate failure, or it could be more insidious, consuming resources that slowly push the service out of alignment with SLOs. This is a circumstance where the fix is in the backend software, but you detect the problem with infrastructure monitoring of resource consumption. You catch these by monitoring system performance metrics, such as the query response times, and reviewing them iteratively.

CASE 1
A request triggers a bug and kills the first backend instance.
The client software (or the user) retries the request.
It kills the next instance.
The service is now potentially trying to start up instances that are almost immediately being killed, and the service becomes overloaded.

CASE 2
Some requests take longer than expected to process, perhaps due to timeouts, retries, and so forth.
Those requests occupy processing for a longer time than anticipated. And for a period of time the service runs at reduced capacity until the long-running requests are cleared.
But what happens if the long-running requests take much longer than expected -- 1000 times longer?
As more requests come in, the service is occupied and can't handle them. The

backlog builds up and the service overloads.

Positive feedback cycle overload failure

Retries: Adding retries where there were none before

Other examples:

- Changes to Load Balancing without considering the behavior during overload conditions
- Crash looping
- Herd behavior

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

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

You saw how "queries of death" can harm system reliability. A similar situation can occur within your system if you try to make it more reliable.

Adding retries example:

- Backend has a simple breaking point: it can handle 10,000 QPS. Every request beyond that fails.
- Frontend issues calls to SimpleRequest() at a rate of 10,100 QPS, so it's going to slightly overload the Backend. The Backend fails 100 QPS worth of requests.
- These 100 failed QPS are retried in SimpleRequest(). They probably succeed.
- Unfortunately, that 100 QPS of retries actually becomes another 100 QPS of requests to the Backend. So now the Backend is receiving 10,200 QPS of traffic, and now 200 QPS of requests are failing.
- Next round, the Backend starts receiving 10,400 QPS of traffic. Then 10,800 QPS, then 11,600 QPS, and so on.

The load increase continues and eventually the Backends start to melt down and crash under the sheer load of requests and retries, successfully handling requests at a significantly reduced rate.
Meanwhile, the requests to the Backends can grow up to 10x the rate of calls to SimpleRequest() to 101,000 QPS.

Feedback: Outputs of a system are routed back as inputs as part of a chain of cause-and-effect that forms a circuit or loop.
Positive feedback occurs when each iteration results in amplification.
https://en.wikipedia.org/wiki/Positive_feedback

Avoid anything in your system that generates positive feedback: Changes can create larger changes, can create larger changes -> a vicious cycle that is difficult to root-cause and to fix.

Crash looping: In response to overload, an application enters an unknown state and crashes.

Herd behavior: Individuals mimic the actions of a larger group. Problem comes from 'gossip' during coordination. Often centered on startup time, time of day, or is event driven.

Java user-facing systems can introduce positive feedback for busy systems without taking care to avoid it, especially in overload situations.

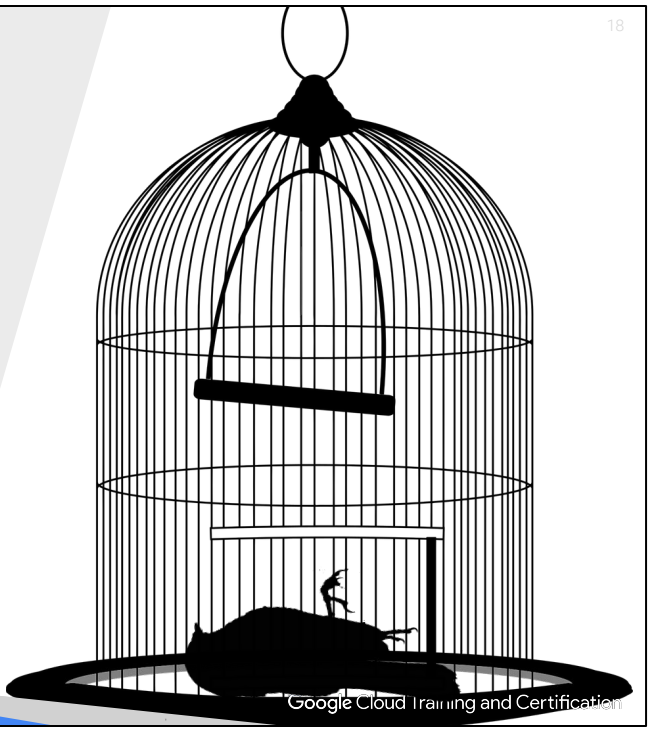
https://pixabay.com/en/audio-communication-equipment-15604/
https://pixabay.com/en/speaker-loudspeaker-volume-sound-153637/

Detect overload: Early warning systems (canaries)

An early warning system that tests key activities before the real activity occurs or before the action is scaled up

Example:

Launching and testing a single server before enabling autoscaling on an instance group



What do you do with the foreknowledge provided by an early warning system? Perhaps, graceful degradation covered on the next slide.

The common term "canary" is derived from an English idiom. An allusion to caged canaries (birds) that miners would carry down into the mine tunnels with them. If dangerous gases such as carbon monoxide collected in the mine, the gases would kill the canary before killing the miners, thus providing a warning to exit the tunnels immediately. https://en.wiktionary.org/wiki/canary_in_a_coal_mine

https://pixabay.com/en/bird-birdcage-cage-victorian-1296205/
Dead bird silhouette is original artwork created for the class by the curriculum developer.

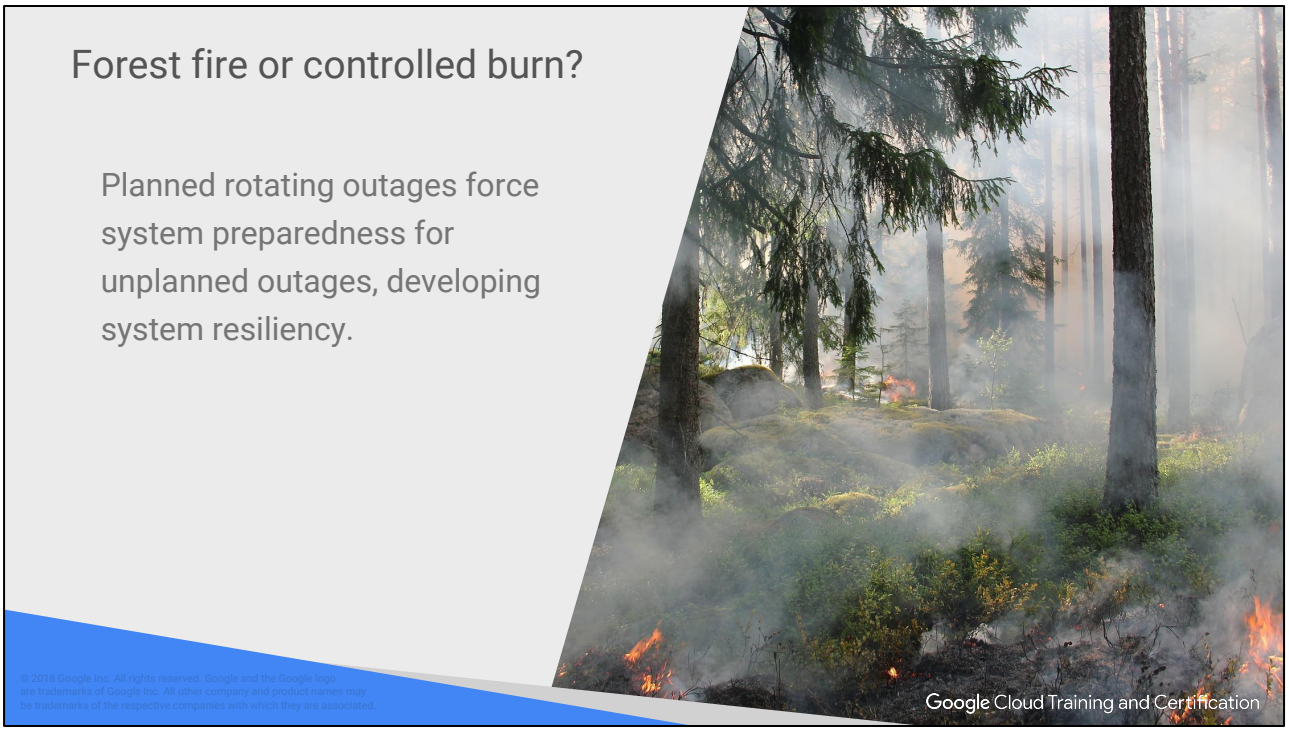
Coping with failure

Preparedness

- The time to prepare for an emergency is *before* it happens
- **No surprises!**

"People don't plan to fail. They fail to plan."

-- John L. Beckley



Scheduled planned maintenance outages.
A service is running in multiple zones and regions. Once a day, at random, a server instance is shut down. Once every week all the resources in a zone are taken out of service, simulating the loss of a zone. Once each quarter, all the resources in a region are taken out of service, simulating a region loss. By scheduling and exercising planned outages as part of the system design and process, when an actual outage does occur, the system is proven to handle be able to handle the outage while maintaining SLOs.

The forest fire and controlled burn analogy:
Forest fires are generally disliked and you may want to avoid them. But the forest has natural ablation. Branches die and fall off. Leaves fall. The refuse from the forest rots, increasing the collection of nitrates and burnable substances. Sooner or later lightning will strike and the forest will catch fire. The longer the time between forest fires, the more fuel accumulates, and the higher the risk of a huge and uncontrollable forest fire. Controlled burns are planned activities that burn off some of the fuel in a defined area under controlled conditions. The planned activities reduce the chances of a devastating forest fire. The planned activity builds resiliency into the forest by helping it learn to adapt and survive.

https://pixabay.com/en/styggk%C3%A4rret-reserve-burning-fire-433688/

Prepare the team using simulation sessions

PROCESS

Experienced member presents a failure scenario

Team member(s)

- Investigate using system's monitoring and diagnostics
- Determine the cause
  - Loss failure
  - Overload failure
- Present strategies for fixing the issue

Google Cloud Training and Certification

Simulation
An experienced team member presents a scenario based on system knowledge. The other team member or team members must investigate using system's monitoring and diagnostics, determine the cause, and present strategies for fixing the issue.
Great for training new team members and developing troubleshooting and system knowledge. Reminds everyone to think about system weaknesses.

Example 1: Problem

- You are paged because your system is returning a lot of 500s. Your monitoring system has alerted you properly.
- You check your playbook - no entry for this. That is an action item for later.
- You check your graph that shows how many frontend servers are healthy - this has been steadily decreasing. Why?
- Spot check server logs - the binary is crashlooping - can't reach the configuration server.
- Configuration server is down because it's OOMing. (Generating Out Of Memory errors).

Example 1: Solution

- **Mitigate:** increase memory if possible.
- **Long-term fix:** clean up old config data, monitor health of config server,

- volume of data etc.

Example 2: Problem

- You are paged that your service needs more resources in 'GAS'.
- Your playbook, uselessly states the obvious "get more resources".
- Looking in the system that allocates resources there seem to be plenty of machines but some of them aren't being used.
- You notice in the details that the exact resource you don't have enough of is a thing called "SSD".
- Looking at the machines that are not being allocated, you see that they know nothing of this "SSD" of which you speak. But they knew about it yesterday....

Example 2: Solution

- You note that you canary new OS images in 'GAS'.
- A new OS image is generated which remembers what SSD is!

https://pixabay.com/en/males-lifebelt-support-1002779/

Incorporate failure into SLOs

The level of service over which the user is accessing your application limits the level of service they will be able to sense in your application:

- Don't over-engineer

There are different kinds of reliability:

- Users may not differentiate them
- You still have to address them separately in your design

Define the gap between user expectations and system reliability:

- Establish a margin of safety
- Incorporate scheduled downtime

Failure and SLOs

- If users access your service over a 99% reliable network, 99.9% reliability in your service won't be visible or noticed. Don't over engineer.
- Define the gap between user expectations and system reliability, establish a margin of safety in your SLOs
- Incorporate *scheduled* downtime into your SLOs
- Users may not differentiate between sources of downtime, you still have to address them separately

Regarding scheduled downtime and SLOs:
If a service is consistently more reliable or available than what is defined in its SLOs, users will begin to rely upon it in a matter that is not consistent with the SLOs. Meaning that the users will grow accustomed to a level of service that was not intended to be delivered, and will expect the service to continue to perform at these new levels even though the SLO has not been increased. For example, if a service has an SLO of 99.0% availability, but it routinely achieves 99.99% availability, the users could begin to expect the service to maintain an availability of 99.99%. The users could potentially incorporate this perceived increase in the SLO when designing dependent applications and services, creating a risky situation. Refer to the Global Chubby Planned Outage in the SRE book, chapter 4.
https://research.google.com/archive/chubby.html
By incorporating scheduled outages and downtime, the SREs were able to keep the user expectations in line with the SLOs of the service.

Though the SLO may change, the SLI does not. The SLI is the metric by which the user is able to determine if the SLO is met. For example, the availability SLO can be observed using an SLI that tracks the cumulative downtime of the service. The planned downtime is reflected in the change of the SLO from 99.9% to 99.0%, however the SLI remains the same (the measurement of cumulative downtime of the service).

Google learned that holding regular production meetings is important to the stability and reliability of the service. The business environment is changing, the technology is changing, and there could be assumptions made in the design that are no longer true.

Coping with failure requires ongoing intelligence: responsible people meeting and discussing the right subjects.

Reference: SRE Book: Chapter 31 - Communication and Collaboration in SRE
Reference: SRE Book: Chapter 5 - Eliminating Toil

If your team is running a live system you should hold regular production meetings

Examples of items to review:

- Recent outages/postmortems
- Major system changes - recent, planned
- Recent alerts - interesting patterns?
- How the system is performing against its SLO
- Releases and rollouts
- Capacity
- Any other concerns

https://pixabay.com/en/people-girls-women-students-2557396/

Strategies for dealing with failure

Redesign	Remove the faulty or at-risk component from the design.
Prevention	Take steps to ensure a failure that is possible does not occur.
Detection and Mitigation	Detect a failure before or as it is happening and take steps to reduce or eliminate the effects of it.
Graceful Degradation	Instead of failing completely, handle stress and return to full service once the issue passes.
Repair	Fix the problem. Hopefully, it won't come back. At least not in the same way.
Recover	Allow the problem to occur and get service back as quickly as possible. Measure indicators of recovery and work to improve them.

Business continuity and disaster recovery

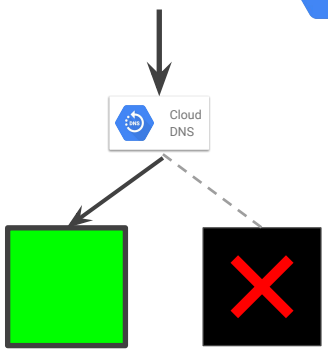
Cloud DNS

Helps users find your service

- A Low latency, high availability way to make your applications and services available to your users.

Resilient design

- If the primary service goes down, use DNS to redirect users to the backup service.



One common DR strategy is to automate the roll-out of a complete alternate system in a different region. If the service is lost, or the region is lost, the alternate system is reconstituted in another region. If you do this, make sure there are periodic realistic tests to make sure that alternate can be built and brought to operation.

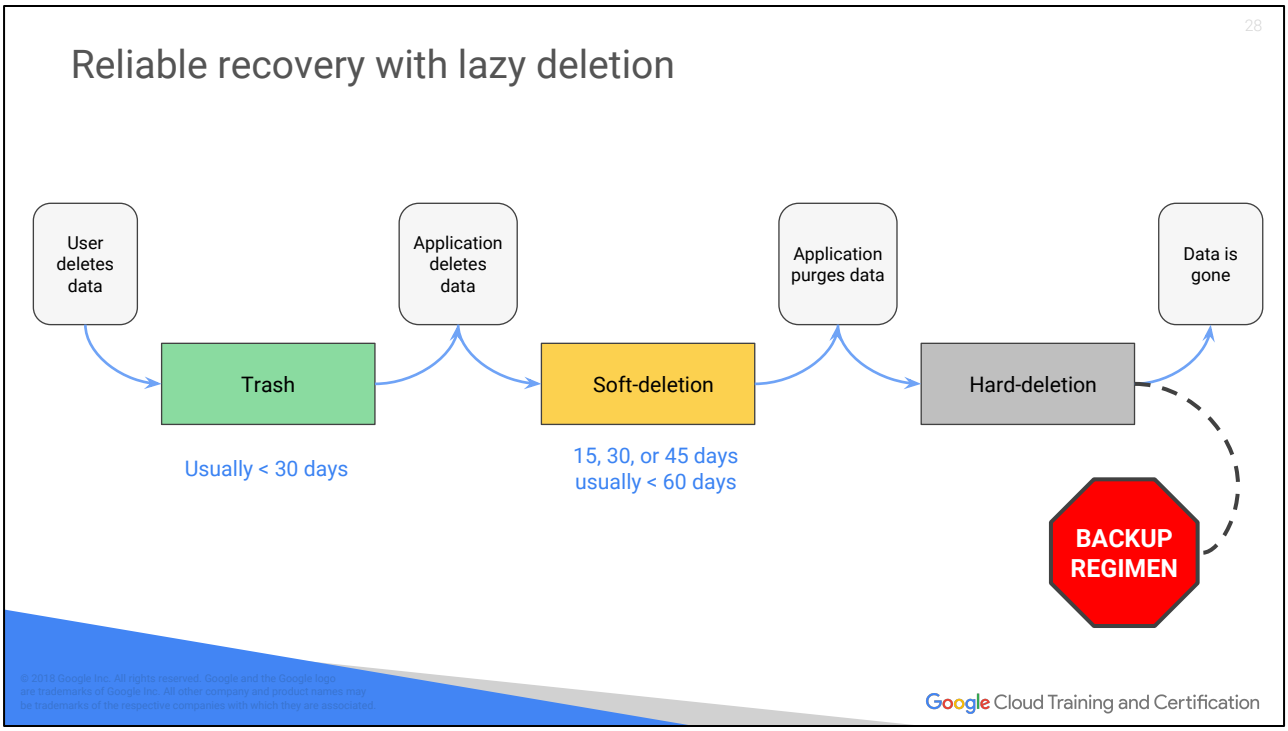
Users care about **data integrity**

Data integrity is the quality of the data remaining accurate and accessible

- If the data loses accuracy or accessibility, it is no longer reliable to the system (and the users), and therefore has lost *integrity*.

Many kinds of failures, including human error, can lead to data loss.

- Strategy is defense in depth



When the user deletes data, it goes into a receptacle where it can be 'undeleted' by the user. This protects against user error.
In some designs, trash can be effectively replaced by version history, with the ability to restore previous versions.

When the application deletes data, it is soft-deleted. It is no longer visible to the user but it can be 'undeleted' by application administration. This protects against bugs in the application as well as mistakes made by services or service providers.

When the application purges data, either the application, a storage service, or a garbage scrubber service hard-deletes the data. Once the application purges or requests the data be purged there is no recovery without going to backups.

Lazy deletion is not a solution for all circumstances. A long deletion period could be costly. Some systems require guarantees of deletion within standard times. Verify privacy and standards with respect to your system requirements and especially if certification is required.

The goal isn't backup or archive, it's **RESTORE**

Backups often are *really* archives

In general, don't care about *backups* care about ***restore***

How long would it take to restore from backup, and does it work?

For confidence

- Periodic testing = low confidence, high chance of interruption
- Automate restore from backup and have it running continuously or periodically to gain confidence

Incorporate recovery testing into your design

What's the point of keeping old versions of data if not to restore them?

https://pixabay.com/en/blood-neutrophil-2194498/

The image of blood cells is an analogy. Think of the white blood cells running around the body, ready to act at any time.
This is how you want your recovery processes to work. You don't want them to be an afterthought.

Tiered backup for resiliency

1st tier	Frequent backups Quickly restored Close to data store May be the same technology as the data store	Backups are kept for hours to days, usually < 10 days
2nd tier	Backups twice a week May take hours to restore Backup to random access distributed file system local to the application	Backups kept for days, usually < 35 days, most common 1 to 2 weeks
3rd tier	Nearline storage Protect against site-level risks Balance risks against costs	Retained longer depending on policy
+tiers	Offline storage / archive	Retained longer; policy

You can shard backup data, by establishing known "trust points" (full backup) and then backing up incrementally from those trust points.
This approach can enable you to take more frequent backups and lower your verification processing requirements.

Also, segmenting backups into "slices" can help parallelize processing and provides benefits by horizontal scaling.

Cloud Storage features for backup and DR

Lifecycle Management

- Changing the storage class of an object
- Deleting an object

Versioning

- Live versus archived *N* versions
- Whole objects, not incremental
- Works with Lifecycle Management.

Enable GCS Access Logs to monitor what actions have been taken as a result of both Versioning and Lifecycle Management

Delete archived versions - they are whole objects and you are charged for each of them

Lifecycle Management

- Changing the storage class of an object in GCS when criteria are met
- Deleting an object from GCS when criteria are met

Versioning

- Live versus archived N versions
  - Whole objects, not incremental
- Works with Lifecycle Management.
  - Example: Keep only the most recent three versions of an object

Criteria

- Age (TTL)
- CreatedBefore
- IsLive (Versioning)
- MatchesStorageClass
- NumberOfNewerVersions

Enable GCS Access Logs to monitor what actions have been taken as a result of both Versioning and Lifecycle Management

Delete archived versions - they are whole objects and you are charged for each of them

GCS versioning is NOT incremental.

Versioning: This is NOT like other versioning systems that establish an incremental change lineage. The whole object is stored. That means you can delete the live version or delete the first archived version, and it will not prevent you from restoring one of the other archived versions and making it live. If you are using versioning, you will want to use lifecycle management to avoid accumulating charges.

Age: You set a TTL (Time To Live) from the date-time-stamp of object creation. So if the object is created on day 1. And you set an Age of 10 days on day 3, then it has 7 days to live. And even if it changes storage class or is archived it will still match the criteria on inception+10.
CreatedBefore: You specify a date, and if inception was before Midnight UTC of that date, then it is a match.
IsLive: "**true**" matches the current version only. "**false**" matches archived versions only.
MatchesStorageclass: *as expected*
NumberOfNewerVersions: 0 for a live object, 1 for most recent archived object if a live object exists, otherwise 0

https://cloud.google.com/storage/docs/managing-lifecycles
https://cloud.google.com/storage/docs/using-object-versioning
https://cloud.google.com/storage/docs/lifecycle#configuration

Prepare the team for disasters by using drills

PROCESS

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.

A drill is a rehearsal of a predetermined set of actions.

Example drills

- Failovers
- Data recovery processes - restoring or rebuilding state
- Running with reduced capacity (lost a rack or a datacenter)
- Detecting and recovering from internal corruption
- Detecting and correcting invalid configurations

Note: *Playbooks* and *post-mortems* are introduced in "Alerting and Incident Response", later in this class.
These items can be useful for inspiring drills and games

https://pixabay.com/en/white-male-3d-model-isolated-3d-1834096/

Scalable and resilient design

Resilient design: Part 1: Use health checks to monitor instances

Used by a load balancer to determine if an instance is healthy or not

Supported Protocols

- HTTP health checks
- HTTPS health checks
- TCP health checks
- SSL (TLS) health checks

https://cloud.google.com/compute/docs/load-balancing/health-checks#overview

A health checker polls instances at specified intervals. Instances that do not respond successfully to a specified number of consecutive probes are marked as UNHEALTHY. No new connections are sent to such instances, though existing connections are allowed to continue. The health checker continues to poll unhealthy instances. If an instance later responds successfully to a specified number of consecutive probes, it is marked HEALTHY again and can receive new connections.

https://pixabay.com/en/stestoskop-heart-curve-course-ad-64276/

Resilient design: Part 2 Instances

Automatically replace instances that have failed or become unavailable

What a new instance needs to do:

- Understand its role in the system
- Configure itself automatically
- Discover any of its dependencies
- Start handling requests automatically

Key technologies

- Startup scripts
- Instance group
- Instance group manager
- Instance template
- Health check

Resilience requires addressing issues in Compute, Networking, and Storage.

https://cloud.google.com/solutions/scalable-and-resilient-apps

Resilient design: Part 3 Storage

Cloud Storage

A location for state information so that no individual server risks the integrity of the system

Redundant, durable, virtually infinite, multiple zone replication

Cloud SQL

Part of the Cloud SQL services is that it can replicate data across the region to multiple zones with a replications instance.

That means it offers reliability as if you were running replication on a standard SQL database

Resilient design: Part 4 Network

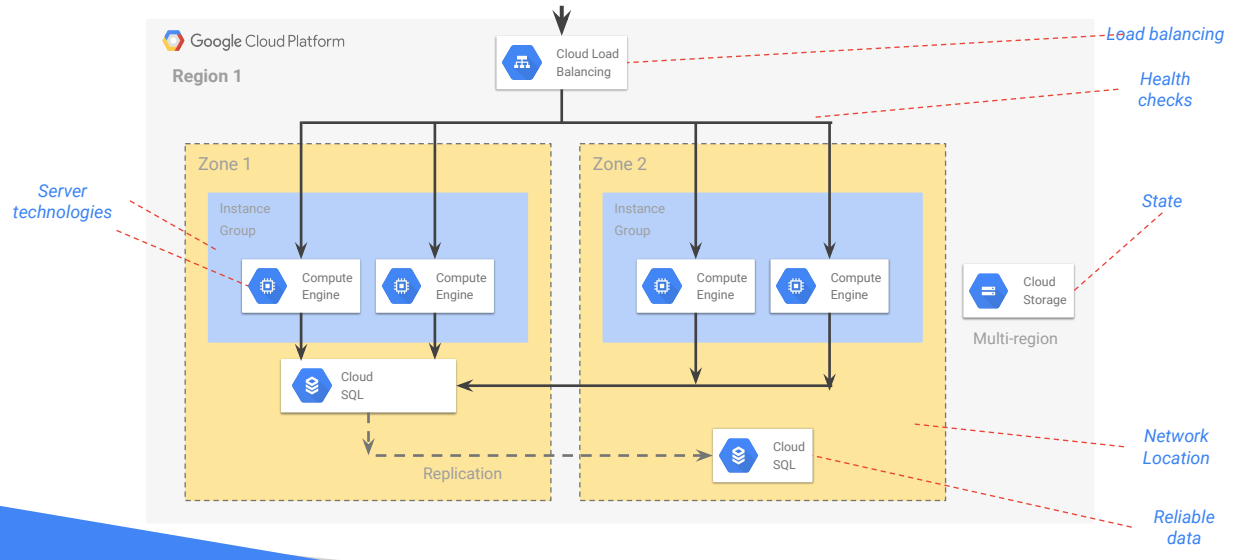
Load balancers

Load balancers to monitor servers and distribute traffic to servers that can best handle the requests

Location

Hosting servers in multiple data centers (zones and regions)

Design pattern: General design for scalable and resilient apps

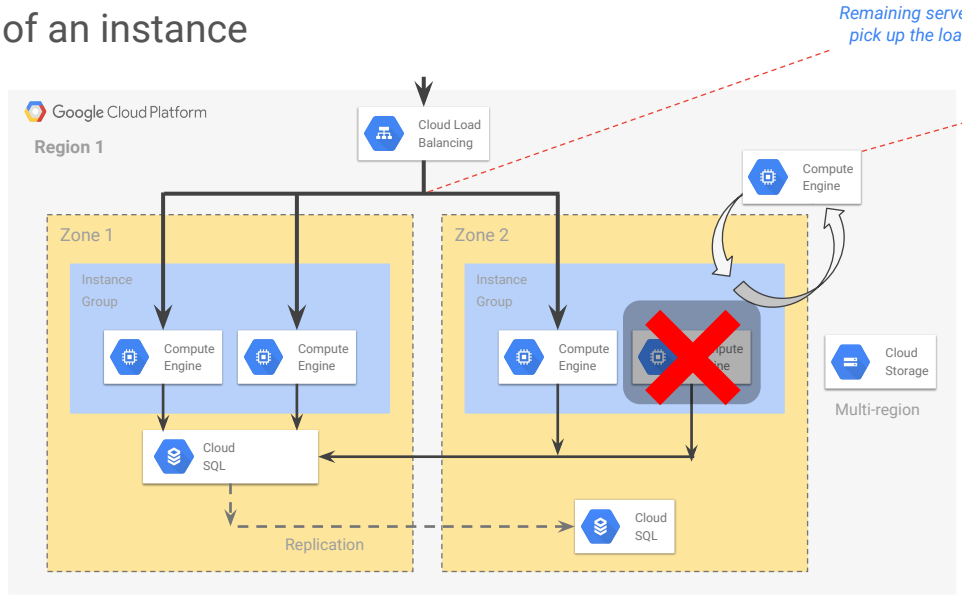


https://cloud.google.com/solutions/scalable-and-resilient-apps

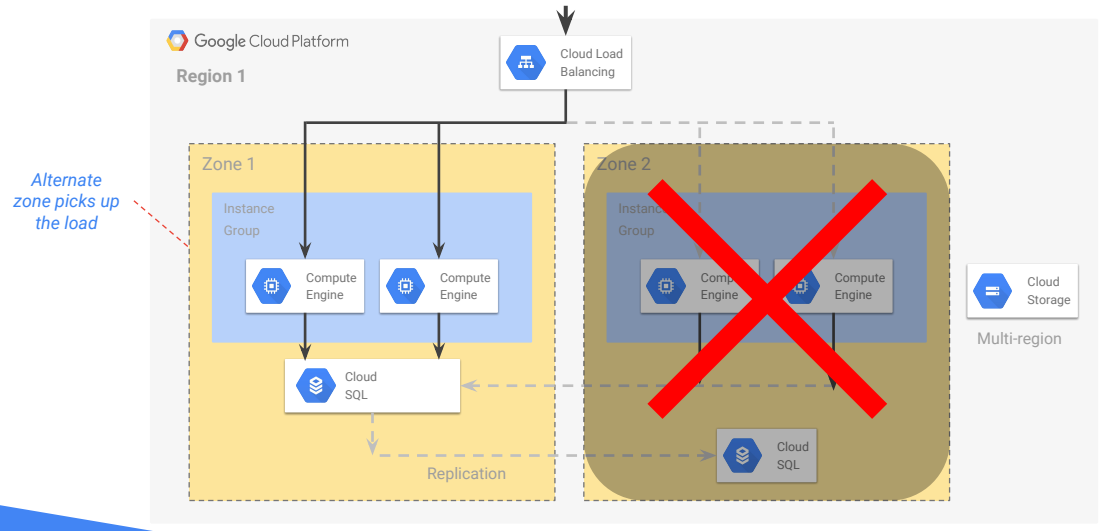
Load balancer distributes traffic over two instance groups in two zones. Each zone contains minimally one VM and an N+1 VM capable of handling the traffic if the other VM is lost.
The instance groups could have a static number of instances or could be autoscaling.
Data is stored in Cloud SQL in one zone and replicated to an Cloud SQL instance in the other zone.
If the entire zone 1 is lost, Zone 2 can pick up the load and switch to the backup Cloud SQL instance.

State information, instance templates, and backups are stored in Cloud Storage. In the event of a disaster, the system can be reconstituted in another region from Cloud Storage.

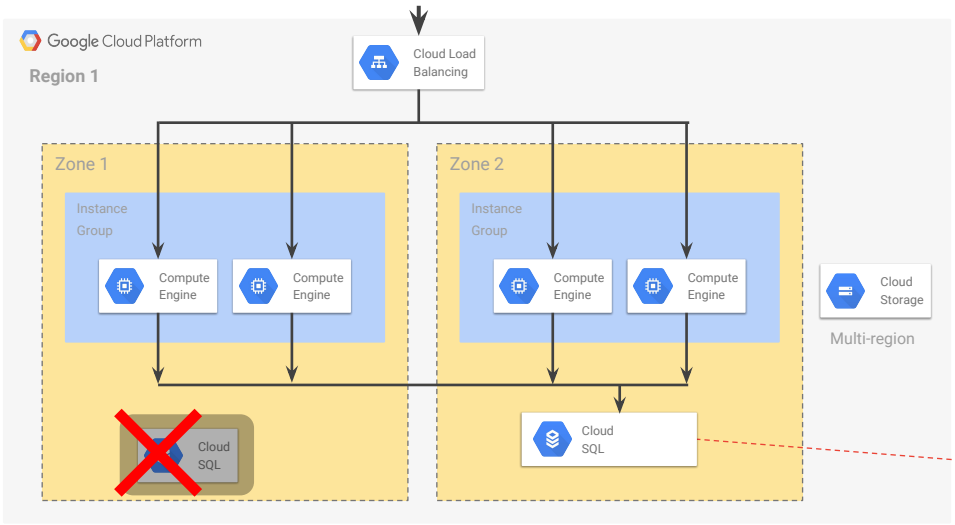
Loss of an instance



Loss of a zone

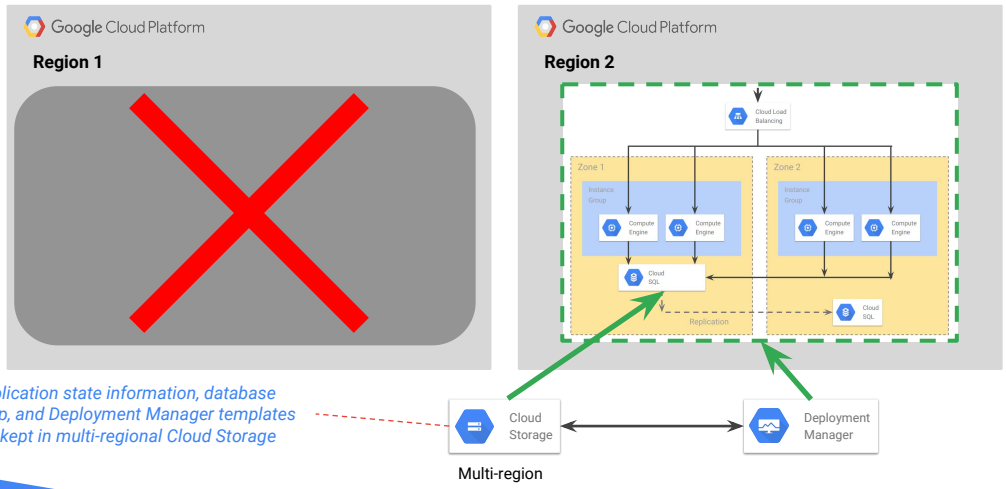


Loss of database

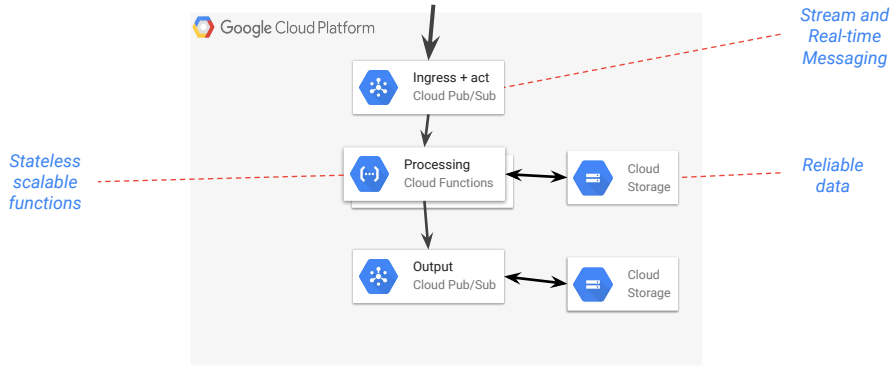


Disaster recovery

Deployment Manager recreates infrastructure in a different region. Database restores state from backup. Service recovers state from database.



Microservices design for scalable and resilient streaming



Tutorial on using Cloud Functions and Cloud Pub/Sub together:
https://cloud.google.com/functions/docs/tutorials/pubsub

https://cloud.google.com/pubsub/
https://cloud.google.com/pubsub/docs/
https://cloud.google.com/functions/

State information, instance templates, and backups are stored in Cloud Storage. In the event of a disaster, the system can be reconstituted in another region from Cloud Storage.

12-factor system and application design in GCP

Treat backing services as attached resources

- Network Load Balancer

Export services via port binding

- Cloud Endpoints, Kubernetes, Load Balancing

Execute the application as one or more stateless processes; scale out via the process model

- Compute Engine Autoscaling, App Engine, Cloud Functions
- Offload state: Datastore, Cloud SQL

Backing Services: The system relies on services; try to abstract them so location makes no difference to your application.
Example minimal access: Endpoint (URL), Username, Password
Your application shouldn't care if the backing service is running on the same VM, on a different VM, in a cloud service, or in containers, or provided remotely

Port Binding: The world connects to your application through a single URL (bound to a single port). Free if the front end of your application is a web server. If the world connects to your application through an API, consider services for wrapping, securing, and managing that interface. Cloud Endpoints. Consider offering a trusted (fast) and untrusted (more secure) API endpoint Also, third party: APIgee.

Processes: Horizontal scaling for fault tolerance and capacity. Ideally, the application code should be stateless. Store state in databases and shared storage services. Never store state in the running code. If a server or process goes down, another process or server should pick up and reacquire state, and the application should continue without the user being impacted.

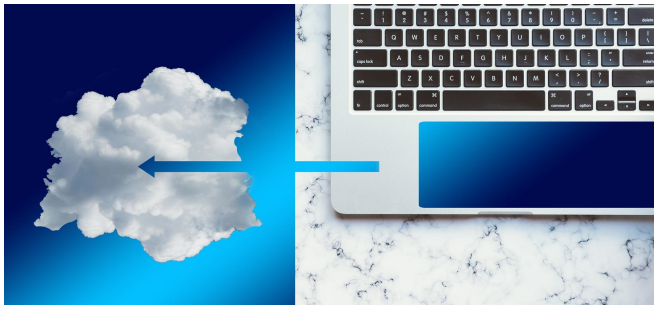
https://pixabay.com/en/binary-hands-keyboard-tap-enter-2372131/

Keep development simple, iterative, and aligned

PROCESS

As much as possible, try to keep code and systems simple:

- Common tools and systems across the organization
- Continuous integration and continuous deployment
- IT enterprise process (e.g., ITIL) to align IT with business requirements



Google learned from experience that making the development environment stable, reliable, iterative, and "boring" actually increased developer productivity and code quality, decreasing the chances of problems in the service.

Reference: SRE Book: Chapter 9 - Simplicity

Common tools: bug tracking, email, text messaging / chat system, groups, documentation and collaboration systems.
https://pixabay.com/en/cloud-infrastructure-disk-space-2531028/

Related to the theme of simplicity is the Microservices Architecture. The simplicity of code units facilitates rapid incident response.

ITIL, Information Technology Infrastructure Library, is a set of detailed practices for IT service management that focuses on aligning IT services with the requirements of the business.

Iterate on the design: Auto-iterate with each release

Test

Look for bottlenecks

Look for ways to improve

Look for bottlenecks

- Verify each function can scale horizontally, load test, profile
- Work out the hardware requirements for each function
  - CPU, network, disk accesses, RAM
  - Sanity check expected concurrent connections

Look for ways to improve

- Is it efficient in terms of resources?
- How can the design accommodate change? Scope changes.
- Is there a better/simpler design?

https://pixabay.com/en/graph-growth-finance-profits-163509/

Out of service!

The photo service has crashed.

How do you handle the outage?

What happened?

How can you prevent it from happening again?

https://pixabay.com/en/summer-sunflower-flowers-sky-cloud-368224/

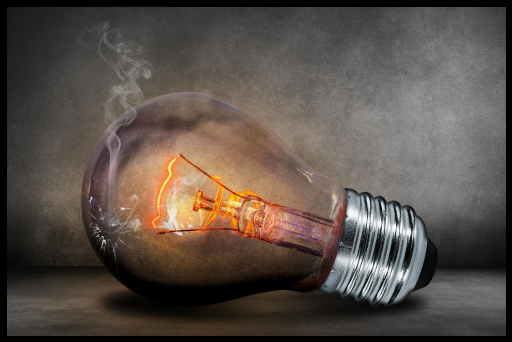
Major system outage!

The thumbnail service went down.

All of the backend thumbnail servers were in a single region and zone. The zone was lost for a brief period and the service failed to recover.

Have a plan for dealing with a major outage

PROCESS



Appoint one person to be in charge of the response

- They delegate necessary tasks to others and coordinate
- Hand off carefully if necessary until incident over
- Maintain a log of the incident state and response

Communicate

- Let those affected know you're responding
- Make it clear who to contact if you want to help

Clean up any loose ends left during incident response

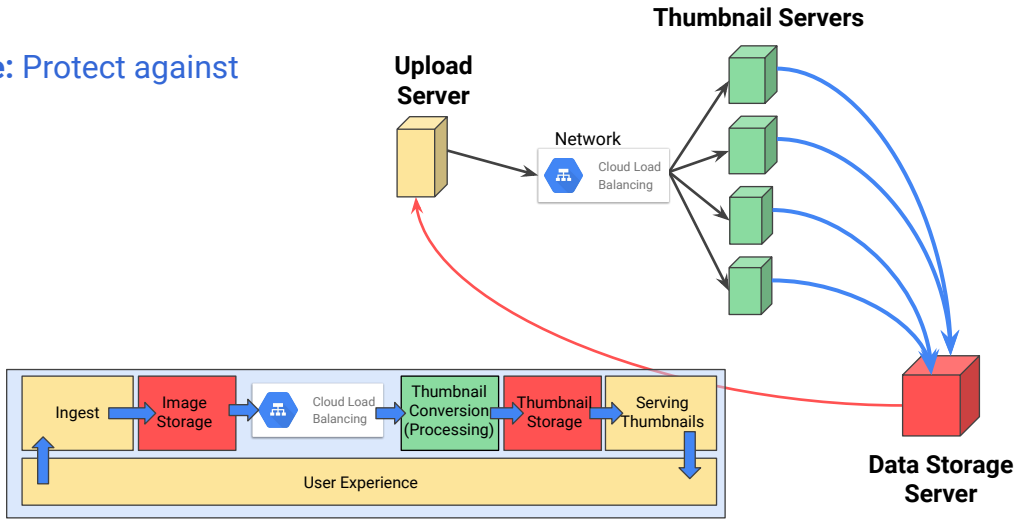
Prepare a postmortem report

- Determine root causes
- Update playbooks, consider adding DR drills

Reference: SRE Book: Chapter 16 - Tracking Outages
https://pixabay.com/en/light-bulb-current-light-glow-503881/

Service loss due to zone outage

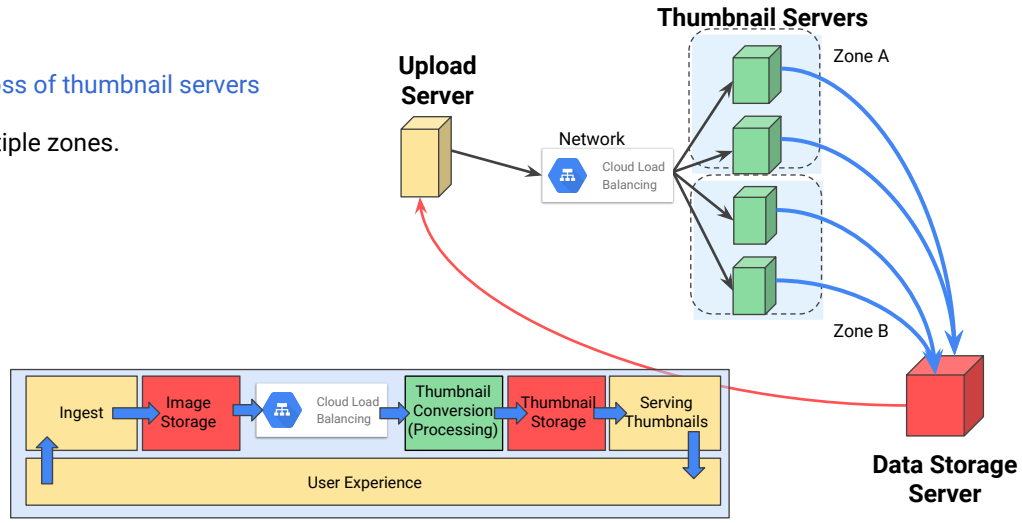
Business Issue: Protect against zone outage



Scale the backend processing of thumbnails

Business Issue: Loss of thumbnail servers

- Move to multiple zones.



A zone was lost and the whole application crashed. Need to separate servers into zones.

Objectives and Indicators

Objectives	Indicators
Availability, 23/24 hours/day = 95.83% availability	Server up/down time
99% of user operations completed in < 1 minute	End to end latency
Failure to produce a thumbnail < 0.01% (100 errors per million)	Completion errors (log entry) @ 1m images/day **Error budget** = 3,000 errors per month

The service will now span multiple zones.
Discussion: Why didn't the SLO/SLI values change?

Discussion answer: The SLOs and SLIs did not change from the previous module because the steps taken to mitigate a zonal failure are related to maintaining the **availability** SLO which is already specified. The additional steps taken to provide for zone redundancy are not visible directly to the user. Even during a zone failover event, the user will only observe that the service maintains the availability SLO. There may be many different technical methods used to maintain the service availability in the background, all of which should be transparent to the end user.

Major system outage!

The photo service has crashed again.
This time the zone did not go down.

The thumbnail service went down due to overload. The upload server is a single point of failure. That machine crashed today due to server overload.

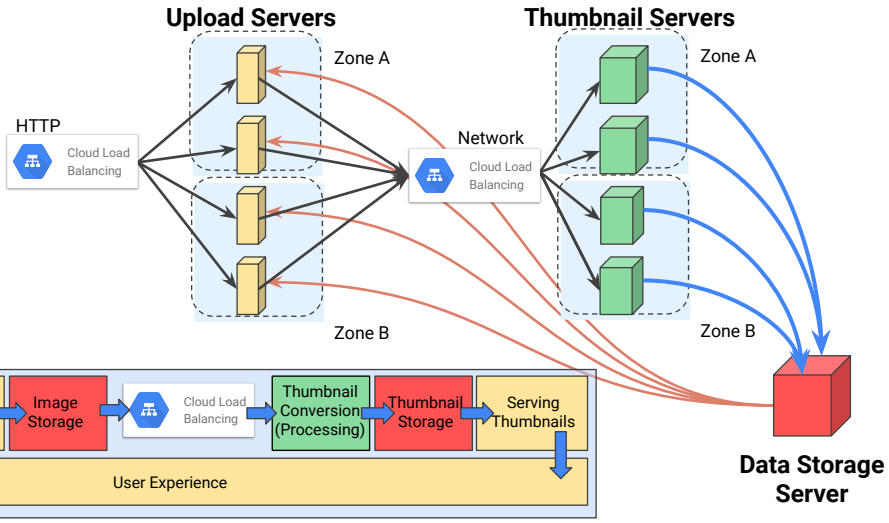
Make changes to the design to prevent overload problems and to improve resiliency.

e.

Need to scale frontend server

Business Issue: Scale frontend server.

- Issue: the single upload server stored user experience state information.
- To scale to multiple upload servers the servers must become stateless.



Prevent overload: Make load testing real

If you can, dark launch.
If not, test in pre-production with synthetic workload.

Correct mix:

- Read-only and state mutating operations
- Diverse workload with realistic cache hitrate
- Test environment should resemble production
- Exercise expensive operations as well
- Be aware of load testing tool limitations

PROCESS

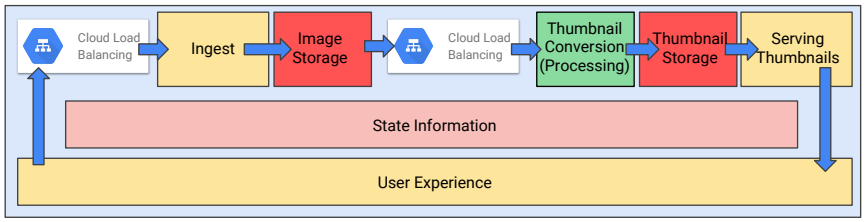
Google learned that simulating load testing can create a false sense of confidence. There are many circumstances in load testing where you can get false results by allowing simplifying assumptions to creep into the test that undermine the validity of the test.

If you can, test during a low use time, such as at night (called a 'dark launch')
If not, test in pre-production using a synthetic workload that closely resembles a real workload. The results could be misleading if the workload is not designed well:

- Correct mix of read-only and state mutating operations
- Sufficiently diverse workload (with realistic cache hitrate)
- Environment should resemble production
- If expensive operations (e.g. returning large lists) exist, exercise these
- Don't rely on load testing tools that minimize the number of concurrent operations - gives optimistic view of peak throughput

https://pixabay.com/en/hard-labour-sacks-transportation-285215/

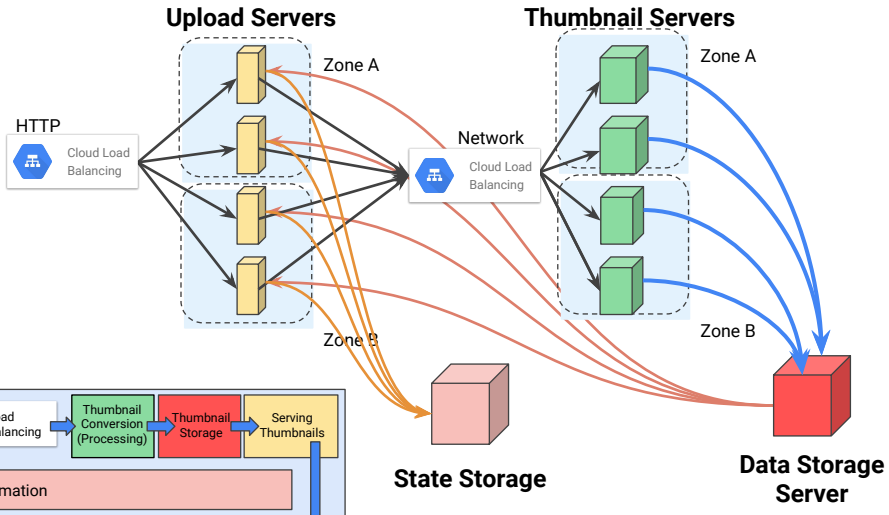
Scaling requires breaking state out of the upload server

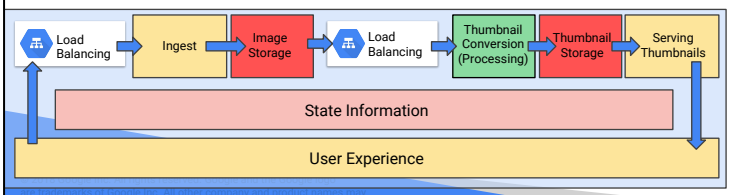


Need to make frontend servers stateless

Business Issue: Scale frontend server.

- Need to store state information to make upload servers stateless and scalable.





SLOs and SLIs

SLOs	SLIs
~~Availability, 23/24 hours/day = 95.83% availability~~ **Availability, 99.9%**	~~Server up/down time~~ **Downtime budget** *including planned and unplanned downtime* 43 minutes per month
99% of user operations completed in < 1 minute	End to end latency
Failure to produce a thumbnail < 0.01% (100 errors per million)	Completion errors (log entry) @ 1m images/day **Error budget** = 3,000 errors per month

The previous SLO of 95.83% availability is no longer appropriate to the level of service required by the users. The Availability SLO has been significantly increased to 99.9%. The SLI is a downtime budget that includes both planned and unplanned downtime.
https://uptime.is/

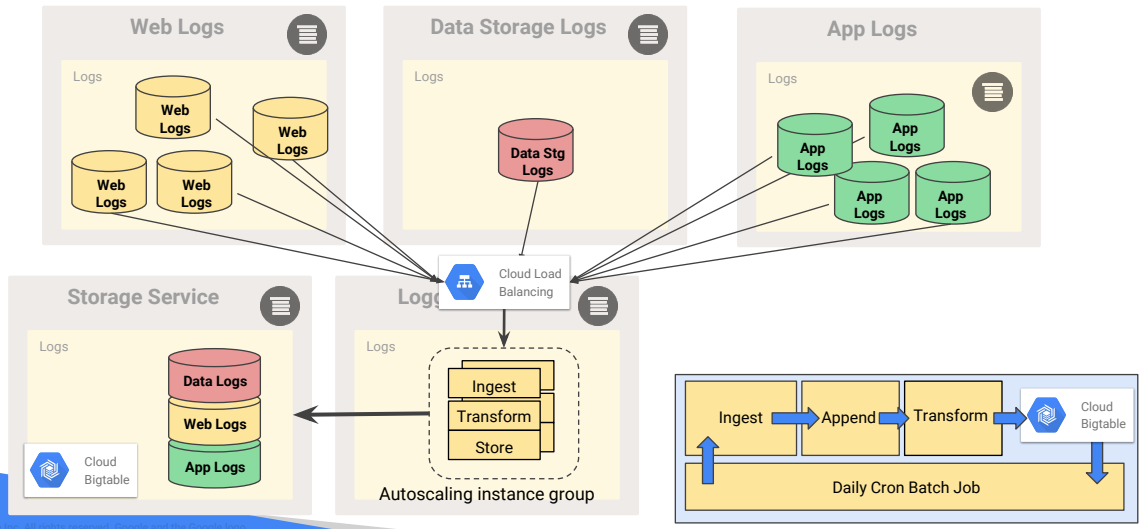
YOUR TURN

Design challenge #4

Redesign for time

https://pixabay.com/en/the-strategy-win-champion-1080527/

Troubleshooting is delayed due to lazy log aggregation



Challenge

12-factor: Treat logs as event streams

Business issue: Service resiliency

It's taking too long to troubleshoot service issues.

Batch processing of logs doesn't support live service troubleshooting. It's causing delays; can't meet the Operations team's goal for identifying and responding to incidents.

Design challenge: Replace the cron batch processing with stream processing. (Hint: consider a microservices design.)

Take a few minutes to design your solution

Problem: Lazy log aggregation is impacting the speed of troubleshooting. The log aggregation service must be made significantly faster.

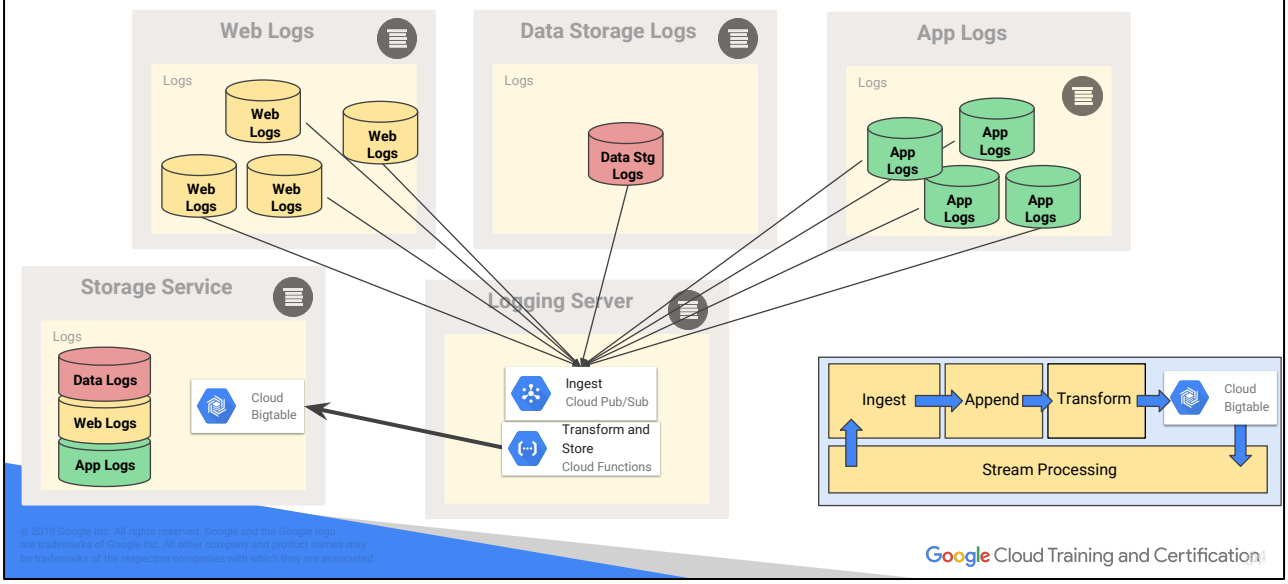
Design a solution.

There are multiple designs possible depending on your assumptions. Your solution might be better than the one shown. The point of this exercise is to "think about the design" to develop your architecting skills.

You can sketch your design in a tool like http://docs.google.com/drawings

One solution

State: Need to move to streaming for live troubleshooting



Microservices solution is one way to get more responsive results.

The cloud function can be used to both transform and store the records in Cloud BigTable.

Google Cloud Training and Certification