Design for scale and high availability

Last reviewed 2023-08-05 UTC

This document in the Google Cloud Architecture Framework (/architecture/framework) provides design principles to architect your services so that they can tolerate failures and scale in response to customer demand. A reliable service continues to respond to customer requests when there's a high demand on the service or when there's a maintenance event. The following reliability design principles and best practices should be part of your system architecture and deployment plan.

Create redundancy for higher availability

Systems with high reliability needs must have no single points of failure, and their resources must be replicated across multiple failure domains. A failure domain is a pool of resources that can fail independently, such as a VM instance, zone, or region. When you replicate across failure domains, you get a higher aggregate level of availability than individual instances could achieve. For more information, see Regions and zones (/compute/docs/regions-zones).

As a specific example of redundancy that might be part of your system architecture, to isolate failures in DNS registration to individual zones, use zonal DNS names (/compute/docs/networking/zonal-dns) for instances on the same network to access each other.

Design a multi-zone architecture with failover for high availability

Make your application resilient to zonal failures by architecting it to use pools of resources distributed across multiple zones, with data replication, load balancing and automated failover between zones. Run zonal replicas of every layer of the application stack, and eliminate all cross-zone dependencies in the architecture.

Replicate data across regions for disaster recovery

Replicate or archive data to a remote region to enable disaster recovery in the event of a regional outage or data loss. When replication is used, recovery is quicker because storage systems in the remote region already have data that is almost up to date, aside from the possible loss of a small amount of data due to replication delay. When you use periodic archiving instead of continuous replication, disaster recovery involves restoring data from backups or archives in a new region. This procedure usually results in longer service downtime than activating a continuously updated

database replica and could involve more data loss due to the time gap between consecutive backup operations. Whichever approach is used, the entire application stack must be redeployed and started up in the new region, and the service will be unavailable while this is happening.

For a detailed discussion of disaster recovery concepts and techniques, see Architecting disaster recovery for cloud infrastructure outages (https://cloud.google.com/architecture/disaster-recovery).

Design a multi-region architecture for resilience to regional outa

If your service needs to run continuously even in the rare case when an entire region fails, design it to use pools of compute resources distributed across different regions. Run regional replicas of every layer of the application stack.

Use data replication across regions and automatic failover when a region goes down. Some Google Cloud services have multi-regional variants, such as Spanner (/spanner). To be resilient against regional failures, use these multi-regional services in your design where possible. For more information on regions and service availability, see Google Cloud locations (/about/locations).

Make sure that there are no cross-region dependencies so that the breadth of impact of a region-level failure is limited to that region.

Eliminate regional single points of failure, such as a single-region primary database that might cause a global outage when it is unreachable. Note that multi-region architectures often cost more, so consider the business need versus the cost before you adopt this approach.

For further guidance on implementing redundancy across failure domains, see the survey paper Deployment Archetypes for Cloud Applications (PDF) (https://storage.googleapis.com/pub-tools-public-publication-data/pdf/cd6b7106c4decf552edc20c125dcb587c4cdcba9.pdf)
.

Eliminate scalability bottlenecks

Identify system components that can't grow beyond the resource limits of a single VM or a single zone. Some applications scale vertically, where you add more CPU cores, memory, or network bandwidth on a single VM instance to handle the increase in load. These applications have hard limits on their scalability, and you must often manually configure them to handle growth.

If possible, redesign these components to scale horizontally such as with sharding, or partitioning, across VMs or zones. To handle growth in traffic or usage, you add more shards. Use standard VM types that can be added automatically to handle increases in per-shard load. For

more information, see Patterns for scalable and resilient apps (/architecture/scalable-and-resilient-apps).

If you can't redesign the application, you can replace components managed by you with fully managed cloud services that are designed to scale horizontally with no user action.

Degrade service levels gracefully when overloaded

Design your services to tolerate overload. Services should detect overload and return lower quality responses to the user or partially drop traffic, not fail completely under overload.

For example, a service can respond to user requests with static web pages and temporarily disable dynamic behavior that's more expensive to process. This behavior is detailed in the warm failover pattern from Compute Engine to Cloud Storage (/architecture/warm-recoverable-static-site-failover-load-balancer). Or, the service can allow read-only operations and temporarily disable data updates.

Operators should be notified to correct the error condition when a service degrades.

Prevent and mitigate traffic spikes

Don't synchronize requests across clients. Too many clients that send traffic at the same instant causes traffic spikes that might cause cascading failures.

Implement spike mitigation strategies on the server side such as throttling, queueing (https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_queue-management), load shedding (https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation) or circuit breaking (https://martinfowler.com/bliki/CircuitBreaker.html), graceful degradation (https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation), and prioritizing critical requests (https://sre.google/sre-book/handling-overload/#criticality-00sDCK).

Mitigation strategies on the client include client-side throttling (https://sre.google/sre-book/handling-overload/#client-side-throttling-a7sYUg) and exponential backoff with jitter (https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/).

Sanitize and validate inputs

To prevent erroneous, random, or malicious inputs that cause service outages or security breaches, sanitize and validate input parameters for APIs and operational tools. For example, Apigee and Google Cloud Armor can help protect against injection attacks (/architecture/owasp-top-ten-mitigation#a1_injection).

Regularly use *fuzz testing* where a test harness intentionally calls APIs with random, empty, or too-large inputs. Conduct these tests in an isolated test environment.

Operational tools should automatically validate configuration changes before the changes roll out, and should reject changes if validation fails.

Fail safe in a way that preserves function

If there's a failure due to a problem, the system components should fail in a way that allows the overall system to continue to function. These problems might be a software bug, bad input or configuration, an unplanned instance outage, or human error. What your services process helps to determine whether you should be overly permissive or overly simplistic, rather than overly restrictive.

Consider the following example scenarios and how to respond to failure:

- It's usually better for a firewall component with a bad or empty configuration to fail open and allow unauthorized network traffic to pass through for a short period of time while the operator fixes the error. This behavior keeps the service available, rather than to fail closed and block 100% of traffic. The service must rely on authentication and authorization checks deeper in the application stack to protect sensitive areas while all traffic passes through.
- However, it's better for a permissions server component that controls access to user data to fail closed and block all access. This behavior causes a service outage when it has the configuration is corrupt, but avoids the risk of a leak of confidential user data if it fails open.

In both cases, the failure should raise a high priority alert so that an operator can fix the error condition. Service components should err on the side of failing open unless it poses extreme risks to the business.

Design API calls and operational commands to be retryable

APIs and operational tools must make invocations retry-safe as far as possible. A natural approach to many error conditions is to retry the previous action, but you might not know whether the first try was successful.

Your system architecture should make actions *idempotent* - if you perform the identical action on an object two or more times in succession, it should produce the same results as a single

invocation. Non-idempotent actions require more complex code to avoid a corruption of the system state.

Identify and manage service dependencies

Service designers and owners must maintain a complete list of dependencies on other system components. The service design must also include recovery from dependency failures, or graceful degradation if full recovery is not feasible. Take account of dependencies on cloud services used by your system and external dependencies, such as third party service APIs, recognizing that every system dependency has a non-zero failure rate.

When you set reliability targets, recognize that the SLO for a service is mathematically constrained by the SLOs of all its critical dependencies. You can't be more reliable than the lowest SLO of one of the dependencies. For more information, see the calculus of service availability (https://research.google/pubs/pub46285/).

Startup dependencies

Services behave differently when they start up compared to their steady-state behavior. Startup dependencies can differ significantly from steady-state runtime dependencies.

For example, at startup, a service may need to load user or account information from a user metadata service that it rarely invokes again. When many service replicas restart after a crash or routine maintenance, the replicas can sharply increase load on startup dependencies, especially when caches are empty and need to be repopulated.

Test service startup under load, and provision startup dependencies accordingly. Consider a design to gracefully degrade by saving a copy of the data it retrieves from critical startup dependencies. This behavior allows your service to restart with potentially stale data rather than being unable to start when a critical dependency has an outage. Your service can later load fresh data, when feasible, to revert to normal operation.

Startup dependencies are also important when you bootstrap a service in a new environment. Design your application stack with a layered architecture, with no cyclic dependencies between layers. Cyclic dependencies may seem tolerable because they don't block incremental changes to a single application. However, cyclic dependencies can make it difficult or impossible to restart after a disaster takes down the entire service stack.

Minimize critical dependencies

Minimize the number of critical dependencies for your service, that is, other components whose failure will inevitably cause outages for your service. To make your service more resilient to failures or slowness in other components it depends on, consider the following example design techniques and principles to convert critical dependencies into non-critical dependencies:

- Increase the level of redundancy in critical dependencies. Adding more replicas makes it less likely that an entire component will be unavailable.
- Use asynchronous requests to other services instead of blocking on a response or use publish/subscribe messaging to decouple requests from responses.
- Cache responses from other services to recover from short-term unavailability of dependencies.

To render failures or slowness in your service less harmful to other components that depend on it, consider the following example design techniques and principles:

- Use prioritized request queues and give higher priority to requests where a user is waiting for a response.
- Serve responses out of a cache to reduce latency and load.
- Fail safe in a way that preserves function.
- Degrade gracefully when there's a traffic overload.

Ensure that every change can be rolled back

If there's no well-defined way to undo certain types of changes to a service, change the design of the service to support rollback. Test the rollback processes periodically. APIs for every component or microservice must be versioned, with backward compatibility such that the previous generations of clients continue to work correctly as the API evolves. This design principle is essential to permit progressive rollout of API changes, with rapid rollback when necessary.

Rollback can be costly to implement for mobile applications. Firebase Remote Config (https://firebase.google.com/products/remote-config) is a Google Cloud service to make feature rollback easier.

You can't readily roll back database schema changes, so execute them in multiple phases. Design each phase to allow safe schema read and update requests by the latest version of your application, and the prior version. This design approach lets you safely roll back if there's a problem with the latest version.

Recommendations

To apply the guidance in the Architecture Framework to your own environment, follow these recommendations:

- Implement exponential backoff with randomization in the error retry logic of client applications.
- Implement a multi-region architecture with automatic failover for high availability.
- Use load balancing to distribute user requests across shards and regions.
- Design the application to degrade gracefully under overload. Serve partial responses or provide limited functionality rather than failing completely.
- Establish a data-driven process for capacity planning, and use load tests and traffic forecasts to determine when to provision resources.
- Establish disaster recovery procedures and test them periodically.

What's next

- Create reliable operational processes and tools (/architecture/framework/reliability/create-operational-processes-tools) (next document in this series)

Explore other categories in the Architecture Framework (/architecture/framework) such as system design, operational excellence, and security, privacy, and compliance.

Last updated 2023-08-05 UTC.