Patterns for scalable and resilient apps

This document introduces some patterns and practices for creating apps that are resilient and scalable, two essential goals of many modern architecture exercises. A well-designed app scales up and down as demand increases and decreases, and is resilient enough to withstand service disruptions. Building and operating apps that meet these requirements requires careful planning and design.

Scalability: Adjusting capacity to meet demand

Scalability (https://wikipedia.org/wiki/Scalability) is the measure of a system's ability to handle varying amounts of work by adding or removing resources from the system. For example, a scalable web app is one that works well with one user or many users, and that gracefully handles peaks and dips in traffic.

The flexibility to adjust the resources consumed by an app is a key business driver for moving to the cloud. With proper design, you can reduce costs by removing under-utilized resources without compromising performance or user experience. You can similarly maintain a good user experience during periods of high traffic by adding more resources. In this way, your app can consume only the resources necessary to meet demand.

Google Cloud provides products and features to help you build scalable, efficient apps:

- Compute Engine (https://cloud.google.com/compute/) virtual machines and Google Kubernetes Engine (GKE) (https://cloud.google.com/kubernetes-engine/) clusters integrate with autoscalers that let you grow or shrink resource consumption based on metrics that you define.

- Google Cloud's serverless platform (https://cloud.google.com/serverless/) provides managed compute, database, and other services that scale quickly from zero to high request volumes, and you pay only for what you use.
- Database products like BigQuery (https://cloud.google.com/bigquery/), Cloud Spanner (https://cloud.google.com/spanner/), and Cloud Bigtable (https://cloud.google.com/bigtable) can deliver consistent performance across massive data sizes.
- Stackdriver Monitoring (https://cloud.google.com/monitoring/) provides metrics across your apps and infrastructure, helping you make data-driven scaling decisions.

Resilience: Designing to withstand failures

A resilient app is one that continues to function despite failures of system components. Resilience requires planning at all levels of your architecture. It influences how you lay out your infrastructure and network and how you design your app and data storage. Resilience also extends to people and culture.

Building and operating resilient apps is hard. This is especially true for distributed apps, which might contain multiple layers of infrastructure, networks, and services. Mistakes and outages happen, and improving the resilience of your app is an ongoing journey. With careful planning, you can improve the ability of your app to withstand failures. With proper processes and organizational culture, you can also learn from failures to further increase your app's resilience.

Google Cloud provides tools and services to help you build highly available and resilient apps:

- Google Cloud services are available in regions and zones (https://cloud.google.com/docs/geography-and-regions#regions_and_zones) across the globe, enabling you to deploy your app to best meet your availability goals.
- Compute Engine instance groups and GKE clusters can be distributed and managed across the available zones in a region.
- Compute Engine regional persistent disks (https://cloud.google.com/compute/docs/disks/#repds) are synchronously replicated across zones in a region.
- Google Cloud provides a range of load-balancing options (https://cloud.google.com/load-balancing/) to manage your app traffic, including global load balancing that can direct traffic to a healthy region closest to your users.

- Google Cloud's serverless platform (https://cloud.google.com/serverless/) includes managed compute and database products that offer built-in redundancy and load balancing.
- Google Cloud supports CI/CD (https://cloud.google.com/docs/ci-cd/) through native tools and integrations with popular open source technologies, to help automate building and deploying your apps.
- Stackdriver Monitoring provides metrics across your apps and infrastructure, helping you make data-driven decisions about the performance and health of your apps.

Drivers and constraints

There are varying requirements and motivations for improving the scalability and resilience of your app. There might also be constraints that limit your ability to meet your scalability and resilience goals. The relative importance of these requirements and constraints varies depending on the type of app, the profile of your users, and the scale and maturity of your organization.

Drivers

To help prioritize your requirements, consider the drivers from the different parts of your organization.

Business drivers

Common drivers from the business side include the following:

- Optimize costs and resource consumption.
- Minimize app downtime.
- Ensure that user demand can be met during periods of high usage.
- Improve quality and availability of service.
- Ensure that user experience and trust are maintained during any outages.
- Increase flexibility and agility to handle changing market demands.

Development drivers

Common drivers from the development side include the following:

- Minimize time spent investigating failures.
- Increase time spent on developing new features.
- Minimize repetitive toil through automation.
- Build apps using the latest industry patterns and practices.

Operations drivers

Requirements to consider from the operations side include the following:

- Reduce the frequency of failures requiring human intervention.
- Increase the ability to automatically recover from failures.
- Minimize repetitive toil through automation.
- Minimize the impact from the failure of any particular component.

Constraints

Constraints might limit your ability to increase the scalability and resilience of your app. Ensure that your design decisions do not introduce or contribute to these constraints:

- Dependencies on hardware or software that is difficult to scale.
- Dependencies on hardware or software that is difficult to operate in a high-availability configuration.
- Dependencies between apps.
- Licensing restrictions.
- Lack of skills or experience in your development and operations teams.
- Organizational resistance to automation.

Patterns and practices

The remainder of this document defines patterns and practices to help you build resilient and scalable apps. These patterns touch all parts of your app lifecycle, including your infrastructure

design, app architecture, storage choices, deployment processes, and organizational culture.

Three themes are evident in the patterns:

- **Automation**. Building scalable and resilient apps requires automation. Automating your infrastructure provisioning, testing, and app deployments increases consistency and speed, and minimizes human error.
- **Loose coupling**. Treating your system as a collection of loosely coupled, independent components allows flexibility and resilience. Independence covers how you physically distribute your resources and how you architect your app and design your storage.
- **Data-driven design**. Collecting metrics to understand the behavior of your app is critical. Decisions about when to scale your app, or whether a particular service is unhealthy, need to be based on data. Metrics and logs should be core features.

Automate your infrastructure provisioning

Create immutable infrastructure through automation to improve the consistency of your environments and increase the success of your deployments.

Treat your infrastructure as code

Infrastructure as code (IaC) is a technique that encourages you to treat your infrastructure provisioning and configuration in the same way you handle application code. Your provisioning and configuration logic is stored in source control so that it's discoverable and can be versioned and audited. Because it's in a code repository, you can take advantage of continuous integration and continuous deployment (CI/CD) pipelines, so that any changes to your configuration can be automatically tested and deployed.

By removing manual steps from your infrastructure provisioning, IaC minimizes human error and improves the consistency and reproducibility of your apps and environments. In this way, adopting IaC increases the resilience of your apps.

Cloud Deployment Manager (https://cloud.google.com/deployment-manager/) lets you automate the creation and management of Google Cloud resources with flexible templates. Google Cloud also has built-in support for popular third-party IaC tools, including Terraform, Chef, and Puppet. For more information, see the Infrastructure as Code (https://cloud.google.com/solutions/infrastructure-as-code/) solution page.

Create immutable infrastructure

Immutable infrastructure is a philosophy that builds on the benefits of infrastructure as code. Immutable infrastructure mandates that resources never be modified after they're deployed. If a virtual machine, Kubernetes cluster, or firewall rule needs to be updated, you can update the configuration for the resource in the source repository. After you've tested and validated the changes, you fully redeploy the resource using the new configuration. In other words, rather than tweaking resources, you re-create them.

Creating immutable infrastructure leads to more predictable deployments and rollbacks. It also mitigates issues that are common in mutable infrastructures, like configuration drift and snowflake servers (https://martinfowler.com/bliki/SnowflakeServer.html). In this way, adopting immutable infrastructure further improves the consistency and reliability of your environments.

Design for high availability

Availability is a measure of the uptime (https://wikipedia.org/wiki/Uptime) of a system. In software systems, high availability is typically achieved through redundantly deploying components. In simplest terms, highly available architectures typically involve distribution of compute resources, load balancing, and replication of data.

Physically distribute resources

Google Cloud services are available in locations across the globe. These locations are divided into regions and zones (https://cloud.google.com/docs/geography-and-regions#regions_and_zones). How you deploy your app across these regions and zones affects the availability, latency, and other properties of your app. For more information, see best practices for Compute Engine region selection (https://cloud.google.com/solutions/best-practices-compute-engine-region-selection).

Redundancy is the duplication of components of a system in order to increase the overall availability of that system. In Google Cloud, redundancy is typically achieved by deploying your app or service to multiple zones, or even in multiple regions. If a service exists in multiple zones or regions, it can better withstand service disruptions in a particular zone or region. Although Google Cloud makes every effort to prevent such disruptions, certain events are unpredictable and it's best to be prepared.

With Compute Engine managed instance groups (https://cloud.google.com/compute/docs/instance-groups/), you can distribute virtual machine instances across multiple zones in a region, and you can manage the instances as a logical unit. Google Cloud also offers regional persistent disks (https://cloud.google.com/compute/docs/disks/#repds) to automatically replicate your data to two zones in a region.

You can similarly improve the availability and resilience of your apps deployed on GKE by creating regional clusters (https://cloud.google.com/kubernetes-engine/docs/concepts/regional-clusters). A regional cluster distributes GKE masters, nodes, and pods across multiple zones within a region. Because your masters are distributed, you can continue to access the cluster's control plane even during an outage involving one or more (but not all) zones.

Favor managed services

Rather than independently installing, supporting, and operating all parts of your application stack, you can use managed services to consume parts of your application stack as services. For example, rather than installing and managing a MySQL database on virtual machines (VMs), you can instead use a MySQL database provided by Cloud SQL (https://cloud.google.com/sql/). You then get an availability Service Level Agreement (SLA) (https://cloud.google.com/sql/sla) and can rely on Google Cloud to manage data replication, backups, and the underlying infrastructure. By using managed services, you can spend less time managing infrastructure, and more time on improving the reliability of your app.

Many of Google Cloud's managed compute, database, and storage services offer built-in redundancy, which can help you meet your availability goals. Many of these services offer a *regional* model, which means the infrastructure that runs your app is located in a specific region and is managed by Google to be redundantly available across all the zones within that region. If a zone becomes unavailable, your app or data automatically serves from another zone in the region.

Certain database and storage services also offer *multi-regional* availability, which means that the infrastructure that runs your app is located in several regions. Multi-regional services can withstand the loss of an entire region, but typically at the cost of higher latency.

Load-balance at each tier

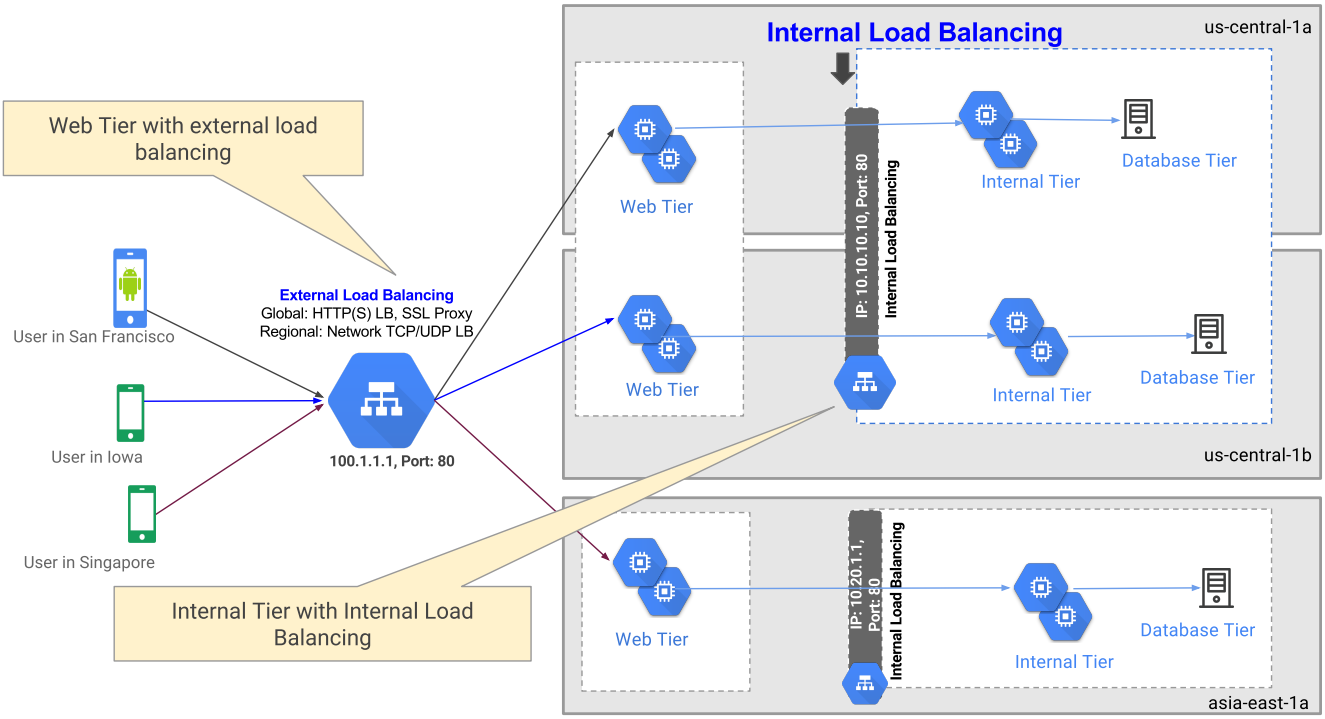
Load balancing lets you distribute traffic among groups of resources. When you distribute traffic, you help ensure that individual resources don't become overloaded while others sit idle. Most load balancers also provide health-checking features to help ensure that traffic isn't routed to unhealthy or unavailable resources.

Google Cloud offers several load-balancing choices. If your app runs on Compute Engine or GKE, you can choose the most appropriate type of load balancer depending on the type, source, and other aspects of the traffic. For more information, see the load-balancing overview (https://cloud.google.com/load-balancing/docs/load-balancing-overview) and GKE networking overview (https://cloud.google.com/kubernetes-engine/docs/concepts/network-overview).

Alternatively, some Google Cloud-managed services, such as App Engine and Cloud Run, automatically load-balance traffic.

It's common practice to load-balance requests received from external sources, such as from web or mobile clients. However, using load balancers between different services or tiers within your app can also increase resilience and flexibility. Google Cloud provides internal layer 4 (https://cloud.google.com/load-balancing/docs/internal/) and layer 7 (https://cloud.google.com/load-balancing/docs/l7-internal/) load balancing for this purpose.

The following diagram shows an external load balancer distributing global traffic across two regions, `us-central1` and `asia-east1`. It also shows internal load balancing distributing traffic from the web tier to the internal tier within each region.



Monitor your infrastructure and apps

Before you can decide how to improve the resilience and scalability of your app, you need to understand its behavior. Having access to a comprehensive set of relevant metrics and time series about the performance and health of your app can help you discover potential issues before they cause an outage. They can also help you diagnose and resolve an outage if it does occur. The monitoring distributed systems (https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems) chapter in the Google SRE book (https://landing.google.com/sre/books/) provides a good overview of some approaches to monitoring.

In addition to providing insight into the health of your app, metrics can also be used to control autoscaling behavior for your services.

Stackdriver Monitoring (https://cloud.google.com/monitoring/) is Google Cloud's integrated monitoring tool. Monitoring ingests events, metrics, and metadata, and provides insights through dashboards and alerts. Most Google Cloud services automatically send metrics (https://cloud.google.com/monitoring/api/metrics) to Stackdriver Monitoring, and Google Cloud also supports many third-party sources. Stackdriver Monitoring can also be used as a backend for

popular open source monitoring tools, providing a "single pane of glass" with which to observe your app.

Monitor at all levels

Gathering metrics at various levels or tiers within your architecture provides a holistic picture of your app's health and behavior.

Infrastructure monitoring

Infrastructure-level monitoring provides the baseline health and performance for your app. This approach to monitoring captures information like CPU load, memory usage, and the number of bytes written to disk. These metrics can indicate that a machine is overloaded, which might warrant triggering a scaling event or generating an alert indicating that there is a problem with the machine.

In addition to the metrics collected automatically, Monitoring provides an agent (https://cloud.google.com/monitoring/agent/) that can be installed to collect more detailed information from Compute Engine VMs, including from third-party apps running on those machines.

App monitoring

We recommend that you capture app-level metrics. For example, you might want to measure how long it takes to execute a particular query, or how long it takes to perform a related sequence of service calls. You define these app-level metrics yourself. They capture information that the built-in Monitoring metrics cannot. App-level metrics can capture aggregated conditions that more closely reflect key workflows, and they can reveal problems that low-level infrastructure metrics do not.

We also recommend using OpenCensus (https://cloud.google.com/monitoring/custom-metrics/open-census) to capture your app-level metrics. OpenCensus is open source, provides a flexible API, and can be configured to export metrics to the Monitoring backend.

Service monitoring

For distributed and microservices-driven apps, it's important to monitor the interactions between the different services and components in your apps. These metrics can help you diagnose problems like increased numbers of errors or latency between services, that once again warrant alerting or scaling.

Istio (https://istio.io/docs/concepts/what-is-istio/) is an open source tool that provides behavioral insights and operational control over your network of microservices. Istio generates detailed telemetry for all service communications, and it can be configured to send the metrics to Stackdriver.

End-to-end monitoring

End-to-end monitoring, also called *black-box monitoring*, tests externally visible behavior the way a user sees it. This type of monitoring checks whether a user is able to complete critical actions within your defined thresholds. This coarse-grained monitoring can uncover errors or latency that finer-grained monitoring might not, and it reveals availability as perceived by the user.

Expose the health of your apps

A highly available system must have some way of determining which parts of the system are healthy and functioning correctly. If certain resources appear unhealthy, the system can send requests elsewhere. Typically health checks involve *pulling* data from an endpoint to determine the status or health of a service.

Health checking is a key responsibility of load balancers. When you create a load balancer that is associated with a group of virtual machine instances, you also define a health check (https://cloud.google.com/load-balancing/docs/health-check-concepts). The health check defines how the load balancer communicates with the virtual machines to evaluate whether particular instances should continue to receive traffic. Load-balancer health checks can also be used to autoheal (https://cloud.google.com/compute/docs/instance-groups/creating-groups-of-managed-instances#monitoring_groups) groups of instances such that unhealthy machines are re-created. If you are running on GKE and load-balancing external traffic through an ingress resource, GKE automatically creates appropriate health checks for the load balancer.

Kubernetes has built-in support for liveness and readiness probes. These probes help the Kubernetes orchestrator decide how to manage pods and requests within your cluster. If your app is deployed on Kubernetes, it's a good idea to expose the health (https://cloud.google.com/solutions/best-practices-for-operating-containers#expose_the_health_of_your_application) of your app to these probes through appropriate endpoints.

Establish key metrics

Monitoring and health checking provide you with metrics on the behavior and status of your app. The next step is to analyze those metrics to determine which are the most descriptive or impactful. The key metrics vary, depending on the platform that the app is deployed on, and on the work that the app is doing. For example, some parts of your app might be CPU bound, while others might be I/O bound.

You're not likely to find just one metric that indicates whether to scale your app, or that a particular service is unhealthy. Often it's a combination of factors that together indicate a certain set of conditions. With Monitoring, you can create custom metrics (https://cloud.google.com/monitoring/custom-metrics/) to help capture these conditions. The Google SRE book advocates four golden signals (https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/#xref_monitoring_golden-signals) for monitoring a user-facing system: latency, traffic, errors, and saturation.

Also consider your tolerance for outliers. Using an average or median value to measure health or performance might not be the best choice, because these measures can hide wide imbalances. It's therefore important to consider the metric *distribution*; the 99th percentile might be a more informative measure than the average.

Store the metrics

Metrics from your monitoring system are useful in the short term to help with real-time health checks or to investigate recent problems. Monitoring retains your metrics for several weeks (https://cloud.google.com/monitoring/quotas#data_retention_policy) to best meet those use cases.

However, there is also value in storing your monitoring metrics for longer-term analysis. Having access to a historical record can help you adopt a data-driven approach to refining your app architecture. You can use data collected during and after an outage to identify bottlenecks and

interdependencies in your apps. You can also use the data to help create and validate meaningful tests.

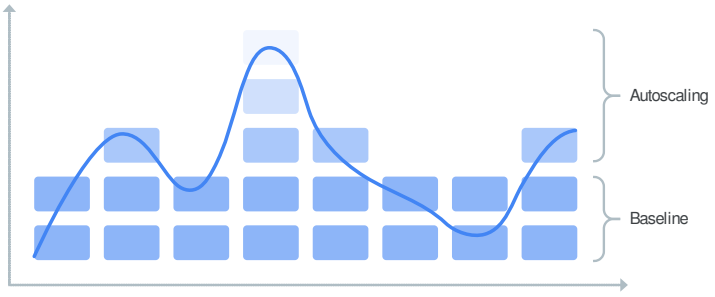
Historical data can also help validate that your app is supporting business goals during key periods. For example, the data can help you analyze how your app scaled during high-traffic promotional events over the course of the last few quarters or even years.

For details on how to export and store your metrics, see the Monitoring metric export (https://cloud.google.com/solutions/stackdriver-monitoring-metric-export) solution.

Determine scaling profile

You want your app to meet its user experience and performance goals without over-provisioning resources.

The following diagram shows how a simplified representation of an app's scaling profile. The app maintains a baseline level of resources, and uses autoscaling to respond to changes in demand.



Balance cost and user experience

Deciding whether to scale your app is fundamentally about balancing cost against user experience. Decide what your minimum acceptable level of performance is, and potentially also where to set a ceiling. These thresholds vary from app to app, and also potentially across different components or services within a single app.

For example, a consumer-facing web or mobile app might have strict latency goals. Research shows (https://developers.google.com/web/fundamentals/performance/why-performance-matters/) that even small delays can negatively impact how users perceive your app, resulting in lower

conversions and fewer signups. Therefore, it's important to ensure that your app has enough serving capacity to quickly respond to user requests. In this instance, the higher costs of running more web servers might be justified.

The cost-to-performance ratio might be different for a non-business-critical internal app where users are probably more tolerant of small delays. Hence, your scaling profile can be less aggressive. In this instance, keeping costs low might be of greater importance than optimizing the user experience.

Set baseline resources

Another key component of your scaling profile is deciding on an appropriate minimum set of resources.

Compute Engine virtual machines or GKE clusters typically take time to scale up, because new nodes need to be created and initialized. Therefore, it might be necessary to maintain a minimum set of resources, even if there is no traffic. Again, the extent of baseline resources is influenced by the type of app and traffic profile.

Conversely, serverless technologies like App Engine, Cloud Functions, and Cloud Run are designed to scale to zero, and to start up and scale quickly, even in the instance of a cold start. Depending on the type of app and traffic profile, these technologies can deliver efficiencies for parts of your app.

Configure autoscaling

Autoscaling (https://wikipedia.org/wiki/Autoscaling) helps you to automatically scale the computing resources consumed by your app. Typically, autoscaling occurs when certain metrics are exceeded or conditions are met. For example, if request latencies to your web tier start exceeding a certain value, you might want to automatically add more machines to increase serving capacity.

Many Google Cloud compute products have autoscaling features. Serverless managed services like Cloud Run, Cloud Functions, and App Engine are designed to scale quickly. These services typically offer configuration options to limit or influence autoscaling behavior, but in general, much of the autoscaler behavior is hidden from the operator.

Compute Engine and GKE provide more options to control scaling behavior. With Compute Engine, you can scale based on various inputs

(https://cloud.google.com/compute/docs/load-balancing-and-autoscaling), including Stackdriver custom metrics and load-balancer serving capacity. You can set minimum and maximum limits on the scaling behavior, and you can define multiple scaling policies (https://cloud.google.com/compute/docs/autoscaler/multiple-policies) to handle different scenarios. As with GKE, you can configure the cluster autoscaler (https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler) to add or remove nodes based on workload or pod metrics (https://cloud.google.com/kubernetes-engine/docs/tutorials/custom-metrics-autoscaling), or on metrics external (https://cloud.google.com/kubernetes-engine/docs/tutorials/external-metrics-autoscaling) to the cluster.

We recommend that you configure autoscaling behavior based on key app metrics, on your cost profile, and on your defined minimum required level of resources.

Minimize startup time

For scaling to be effective, it must happen quickly enough to handle the increasing load. This is especially true when adding compute or serving capacity.

Use pre-baked images

If your app runs on Compute Engine VMs, you likely need to install software and configure the instances to run your app. Although you can use startup scripts (https://cloud.google.com/compute/docs/startupscript) to configure new instances, a more efficient way is to create a custom image (https://cloud.google.com/compute/docs/images#custom_images). A custom image is a boot disk that you set up with your app-specific software and configuration.

For more information on managing images, see the image-management best practices (https://cloud.google.com/solutions/image-management-best-practices) article.

When you've created your image, you can define an instance template (https://cloud.google.com/compute/docs/instance-templates/). Instance templates combine the boot disk image, machine type, and other instance properties. You can then use an instance template to create individual VM instances or a managed instance group (https://cloud.google.com/compute/docs/instance-groups/creating-groups-of-managed-instances). Instance templates are a convenient way to save a VM instance's configuration so you can use it later to create identical new VM instances.

Although creating custom images and instance templates can increase your deployment speed, it can also increase maintenance costs because the images might need to be updated more frequently. For more information, see the balancing image configuration and deployment speed (https://cloud.google.com/solutions/dr-scenarios-building-blocks#balancing_image_configuration_and_deployment_speed) docs.

Containerize your app

An alternative to building customized VM instances is to containerize your app. A container (https://cloud.google.com/containers/) is a lightweight, standalone, executable package of software that includes everything needed to run an app: code, runtime, system tools, system libraries, and settings. These characteristics make containerized apps more portable, easier to deploy, and easier to maintain at scale than virtual machines. Containers are also typically fast to start, which makes them suitable for scalable and resilient apps.

Google Cloud offers several services to run your app containers. Cloud Run (https://cloud.google.com/run/) provides a serverless, managed compute platform to host your stateless containers. The App Engine Flexible (https://cloud.google.com/appengine/docs/flexible/) environment hosts your containers in a managed platform as a service (PaaS). GKE (https://cloud.google.com/kubernetes-engine/) provides a managed Kubernetes environment to host and orchestrate your containerized apps. You can also run your app containers on Compute Engine (https://cloud.google.com/compute/docs/containers/) when you need complete control over your container environment.

Optimize your app for fast startup

In addition to ensuring your infrastructure and app can be deployed as efficiently as possible, it's also important to ensure your app comes online quickly.

The optimizations that are appropriate for your app vary depending on the app's characteristics and execution platform. It's important to do the following:

- Find and eliminate bottlenecks by profiling the critical sections of your app that are invoked at startup.
- Reduce initial startup time by implementing techniques like lazy initialization, particularly of expensive resources.
- Minimize app dependencies that might need to be loaded at startup time.

Favor modular architectures

You can increase the flexibility of your app by choosing architectures that enable components to be independently deployed, managed, and scaled. This pattern can also improve resiliency by eliminating single points of failure.

Break your app into independent services

If you design your app as a set of loosely coupled, independent services, you can increase your app's flexibility. If you adopt a loosely coupled design, it lets your services be independently released and deployed. In addition to many other benefits, this approach enables those services to use different tech stacks and to be managed by different teams. This loosely coupled approach is the key theme of architecture patterns like microservices and SOA.

As you consider how to draw boundaries around your services, availability and scalability requirements are key dimensions. For example, if a given component has a different availability requirement or scaling profile from your other components, it might be a good candidate for a standalone service.

For more information, see Migrating a monolithic app to microservices (https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke).

Aim for statelessness

A stateless app or service does not retain any local persistent data or state. A stateless model ensures that you can handle each request or interaction with the service independent of previous requests. This model facilitates scalability and recoverability, because it means that the service can grow, shrink, or be restarted without losing data that's required in order to handle any in-flight processes or requests. Statelessness is especially important when you are using an autoscaler, because the instances, nodes, or pods hosting the service can be created and destroyed unexpectedly.

It might not be possible for all your services to be stateless. In such a case, be explicit about services that require state. By ensuring clean separation of stateless and stateful services, you can ensure easy scalability for stateless services while adopting a more considered approach for stateful services.

Manage communication between services

One challenge with distributed microservices architectures is managing communication between services. As your network of services grows, it's likely that service interdependencies will also grow. You don't want the failure of one service to result in the failure of other services, sometimes called a *cascading failure*.

You can help reduce traffic to an overloaded service or failing service by adopting techniques like the circuit breaker (https://martinfowler.com/bliki/CircuitBreaker.html) pattern, exponential backoffs (https://wikipedia.org/wiki/Exponential_backoff), and graceful degradation (https://landing.google.com/sre/sre-book/chapters/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation). These patterns increase the resiliency of your app either by giving overloaded services a chance to recover, or by gracefully handling error states. For more information, see the addressing cascading failures (https://landing.google.com/sre/sre-book/chapters/addressing-cascading-failures/) chapter in the Google SRE book.

Using a service mesh (https://cloud.google.com/blog/products/networking/welcome-to-the-service-mesh-era-introducing-a-new-istio-blog-post-series) can help you manage traffic across your distributed services. A service mesh is software that links services together, and helps decouple business logic from networking. A service mesh typically provides resiliency features like request retries, failovers, and circuit breakers.

Use appropriate database and storage technology

Certain databases and types of storage are difficult to scale and make resilient. Make sure that your database choices don't constrain your app's availability and scalability.

Evaluate your database needs

The pattern of designing your app as a set of independent services also extends to your databases and storage. It might be appropriate to choose different types of storage for different parts of your app, which results in heterogeneous storage.

Traditional apps often operate exclusively with relational databases. Relational databases offer useful functionality such as transactions, strong consistency, referential integrity, and

sophisticated querying across tables. These features make relational databases a good choice for many common app features. However, relational databases also have some constraints. They are typically hard to scale, and they require careful management in a high-availability configuration. A relational database might not be the best choice for all your database needs.

Non-relational databases, often referred to as NoSQL databases, take a different approach. Although details vary across products, NoSQL databases typically sacrifice some features of relational databases in favor of increased availability and easier scalability. In terms of the CAP theorem (https://wikipedia.org/wiki/CAP_theorem), NoSQL databases often choose availability over consistency.

Whether a NoSQL database is appropriate often comes down to the required degree of consistency. If your data model for a particular service does not require all the features of an RDBMS, and can be designed to be eventually consistent, choosing a NoSQL database might offer increased availability and scalability.

In addition to a range of relational and NoSQL databases, Google Cloud also offers Cloud Spanner (https://cloud.google.com/spanner/), a strongly consistent, highly available, and globally distributed database with support for SQL. For information about choosing an appropriate database on Google Cloud, see Google Cloud databases (https://cloud.google.com/products/databases/).

Implement caching

A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer.

Caching supports improved scalability by reducing reliance on disk-based storage. Because requests can be served from memory, request latencies to the storage layer are reduced, typically allowing your service to handle more requests. In addition, caching can reduce the load on services that are downstream of your app, especially databases, allowing other components that interact with that downstream service to also scale more easily or at all.

Caching can also increase resiliency by supporting techniques like graceful degradation (https://landing.google.com/sre/sre-book/chapters/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation). If the underlying storage layer is overloaded or unavailable, the cache can continue to handle requests. And even though the data returned from the cache might be incomplete or not up to date, that might be acceptable for certain scenarios.

Memorystore for Redis (https://cloud.google.com/memorystore/) provides a fully managed service that is powered by the Redis in-memory datastore. Memorystore for Redis provides low-latency access and high throughput for heavily accessed data. It can be deployed in a high-availability configuration that provides cross-zone replication and automatic failover.

Modernize your development processes and culture

DevOps can be considered a broad collection of processes, culture, and tooling that promote agility and reduced time-to-market for apps and features by breaking down silos between development, operations, and related teams. DevOps techniques aim to improve the quality and reliability of software.

A detailed discussion of DevOps is beyond the scope of this article, but some key aspects that relate to improving the reliability and resilience of your app are discussed in the following sections. For more details, see the Google Cloud DevOps page (https://cloud.google.com/solutions/devops/).

Design for testability

Automated testing (https://cloud.google.com/solutions/devops/devops-tech-test-automation) is a key component of modern software delivery practices. The ability to execute a comprehensive set of unit, integration, and system tests is essential to verify that your app behaves as expected, and that it can progress to the next stage of the deployment cycle. Testability is a key design criterion for your app.

We recommend that you use unit tests for the bulk of your testing because they are quick to execute and typically easy to maintain. We also recommend that you automate higher-level integration and system tests. These tests are greatly simplified if you adopt infrastructure-as-code techniques, because dedicated test environments and resources can be created on demand, and then torn down once tests are complete.

As the percentage of your codebase covered by tests increases, you reduce uncertainty and the potential decrease in reliability from each code change. Adequate testing coverage means that you can make more changes before reliability falls below an acceptable level.

Automated testing is an integral component of continuous integration (https://cloud.google.com/solutions/devops/devops-process-continuous-integration). Executing a

robust set of automated tests on each code commit provides fast feedback on changes, improving the quality and reliability of your software. Google Cloud–native tools like Cloud Build (https://cloud.google.com/cloud-build/) and third-party tools like Jenkins (https://cloud.google.com/jenkins/) can help you implement continuous integration.

Automate your deployments

Continuous integration and comprehensive test automation give you confidence in the stability of your software. And when they are in place, your next step is automating deployment (https://cloud.google.com/solutions/devops/devops-process-deployment-automation) of your app. The level of deployment automation varies depending on the maturity of your organization.

Choosing an appropriate deployment strategy is essential in order to minimize the risks associated with deploying new software. With the right strategy, you can gradually increase the exposure of new versions to larger audiences, verifying behavior along the way. You can also set clear provisions for rollback if problems occur.

For examples of automating deployments, see Continuous Delivery Pipelines with Spinnaker and GKE (https://cloud.google.com/solutions/continuous-delivery-spinnaker-kubernetes-engine) and Automating Canary Analysis on GKE with Spinnaker (https://cloud.google.com/solutions/automated-canary-analysis-kubernetes-engine-spinnaker).

Adopt SRE practices for dealing with failure

For distributed apps that operate at scale, some degree of failure in one or more components is common. If you adopt the patterns covered in this document, your app can better handle disruptions caused by a defective software release, unexpected termination of virtual machines, or even an infrastructure outage that affects an entire zone.

However, even with careful app design, you inevitably encounter unexpected events that require human intervention. If you put structured processes in place to manage these events, you can greatly reduce their impact and resolve them more quickly. Furthermore, if you examine the causes and responses to the event, you can help protect your app against similar events in the future.

Strong processes for managing incidents (https://landing.google.com/sre/sre-book/chapters/managing-incidents/) and performing blameless postmortems (https://landing.google.com/sre/sre-book/chapters/postmortem-culture/) are key tenets

of SRE. Although implementing the full practices of Google SRE might not be practical for your organization, if you adopt even a minimum set of guidelines, you can improve the resilience of your app. The appendices in the SRE book (https://landing.google.com/sre/sre-book/toc/) contain some templates that can help shape your processes.

Validate and review your architecture

As your app evolves, user behavior, traffic profiles, and even business priorities can change. Similarly, other services or infrastructure that your app depends on can evolve. Therefore, it's important to periodically test and validate the resilience and scalability of your app.

Test your resilience

It's critical to test that your app responds to failures in the way you expect. The overarching theme is that the best way to avoid failure is to introduce failure and learn from it.

Simulating and introducing failures is complex. In addition to verifying the behavior of your app or service, you must also ensure that expected alerts are generated, and appropriate metrics are generated. We recommend a structured approach, where you introduce simple failures and then escalate.

For example, you might proceed as follows, validating and documenting behavior at each stage:

- Introduce intermittent failures.
- Block access to dependencies of the service.
- Block all network communication.
- Terminate hosts.

For details, see the Breaking your systems to make them unbreakable (https://www.youtube.com/watch?v=pVYwagnFXJI) video from Google Cloud Next 2019.

If you're using a service mesh like Istio to manage your app services, you can inject faults (https://istio.io/docs/concepts/traffic-management/#fault-injection) at the application layer instead of killing pods or machines, or you can inject corrupting packets at the TCP layer. You can introduce delays to simulate network latency or an overloaded upstream system. You can also introduce aborts, which mimic failures in upstream systems.

Test your scaling behavior

We recommend that you use automated nonfunctional testing to verify that your app scales as expected. Often this verification is coupled with performance or load testing. You can use simple tools like hey (https://github.com/rakyll/hey) to send load to a web app. For a more detailed example that shows how to do load testing against a REST endpoint, see Distributed load testing using Google Kubernetes Engine (https://cloud.google.com/solutions/distributed-load-testing-using-gke).

One common approach is to ensure that key metrics stay within expected levels for varying loads. For example, if you're testing the scalability of your web tier, you might measure the average request latencies for spiky volumes of user requests. Similarly, for a backend processing feature, you might measure the average task-processing time when the volume of tasks suddenly increases.

Also, you want your tests to measure that the number of resources that were created to handle the test load is within the expected range. For example, your tests might verify that the number of VMs that were created to handle some backend tasks does not exceed a certain value.

It's also important to test edge cases. What is the behavior of your app or service when maximum scaling limits are reached? What is the behavior if your service is scaling down and then load suddenly increases again? For a discussion of these topics, see the load testing section of Peak-season production readiness (https://cloud.google.com/solutions/black-friday-production-readiness#helping_to_ensure_reliability).

Always be architecting

The technology world moves fast, and this is especially true of the cloud. New products and features are released frequently, new patterns emerge, and the demands from your users and internal stakeholders continue to grow.

As the principles for cloud-native architecture (https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it?utm_medium=email&utm_source=other&utm_campaign=partner.443.opencourse.targetedmessages.marketing%7Epartner.443.r7GztVGbEemwag6YlZVrbA) blog post defines, always be looking for ways to refine, simplify, and improve the architecture of

your apps. Software systems are living things and need to adapt to reflect your changing priorities.

What's next

- Read the principles for cloud-native architecture (https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it?utm_medium=email&utm_source=other&utm_campaign=partner.443.opencourse.targetedmessages.marketing%7Epartner.443.r7GztVGbEemwag6YIZVrbA) blog post.
- Read the SRE books (https://landing.google.com/sre/books/) for details on how the Google production environment is managed.
- Learn more about how DevOps (https://cloud.google.com/solutions/devops/) on Google Cloud can improve your software quality and reliability.
- Try out other Google Cloud Platform features for yourself. Have a look at our tutorials (https://cloud.google.com/docs/tutorials).

Last updated January 15, 2020.