Defining the Service

Architecting with Google Cloud Platform: Design and Process

Last modified 2018-08-08

Agenda

Course introduction

Your design experience in this class

State and solution

Measurement

Gathering requirements

Introducing an example photo service

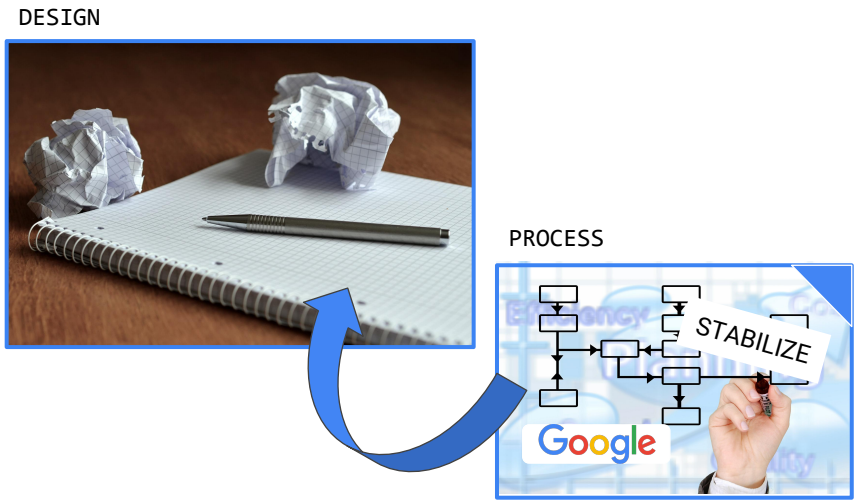
GCP lab Deployment Manager: Beginning appserver

Course introduction

This class is very linear. You will go through a step-by-step design process so that you can learn how to think through cloud design.

Along the way you will be taught process lessons that Google learned from experience so that you don't have to repeat those experiences.

This class covers cloud service design and process



In this class you will be following a sequential method to produce a cloud solution architecture that is practical and stable.
You will be working with an example application and evolving it from a simple application to a complex and robust service.

The design process is a technical activity. However, the Cloud Architect job is not complete when the design is implemented. There is usually a period of 3 to 6 months during which the system must be monitored, adjusted, and stabilized. Google has evolved very specific recommendations about developing process around the architecture and planning for behaviors that will help the solution became stable and maintainable. Where possible, some illustration will be provided.

You will evolve the design architecture and develop the surrounding elements, both technical and procedural, that will enable your solution to stabilize into a reliable, scalable, sustainable solution.

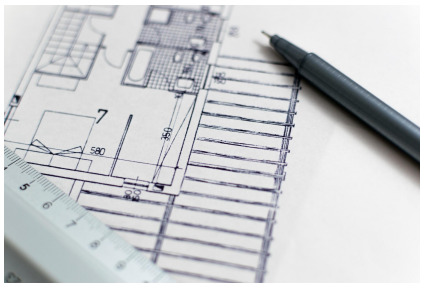
The material in this class is based on 15+ years of experience Google acquired making applications like gMail and Google Maps scalable and reliable and then applying the same methods to other Google applications. Google wants to share the lessons from this experience. Some of the knowledge is outside of the scope of the Cloud Architect's job. Not all the knowledge Google has acquired that is in scope can be covered in a two day class. Therefore, the goal with respect to process knowledge is to communicate several key lessons from Google's experience and to prepare you to continue learning after completing this class.

For your continued learning, the body of knowledge that Google has learned from deploying large-scale Cloud services goes by the name Site Reliability Engineering (SRE). There is a book available for purchased for additional study entitled "Site Reliability Engineering: How Google Runs Production Systems". *Google it!*

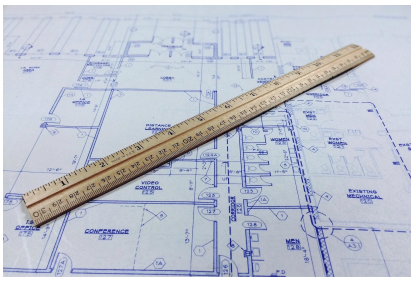
https://pixabay.com/en/notes-paper-paper-ball-memo-office-514998/
https://pixabay.com/en/mark-marker-hand-leave-516277/

1: Defining the service

Rough



Structured design



Measurable

Iterate

Defining the service. In this module you will learn about Service Level Objectives, Indicators, and Agreements.

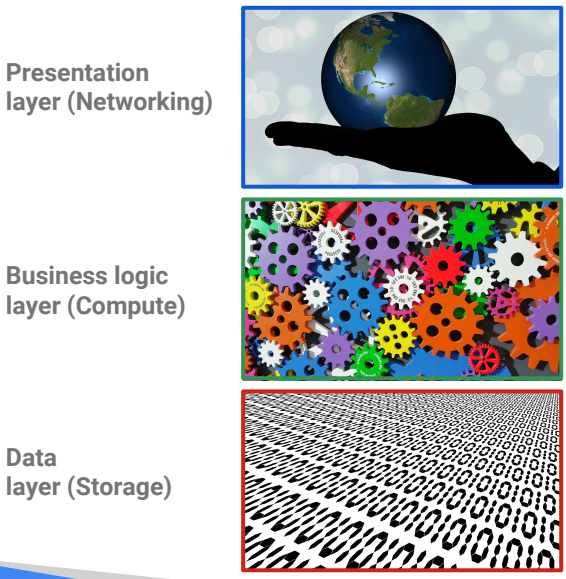
The progression shown in this slide highlights the risk of implementing without a design and a design process ("rolling the dice").
To mitigate this risk, the architect takes the application through a design process. The structured approach communicated in this class identifies the building blocks necessary to produce the service. And as those components are refined through the design, measurements are developed that serve as objective ways to determine how the service is operating and when the design has to change and grow.

The Service Level Objectives are essential to design. And Service Level Indicators are essential to implementation and operation of the solution.
SLOs and SLIs determine whether a solution is accomplishing its purpose or whether adjustment must be made. The measures are both goals and guides.

Measurement in design is a practical matter not a scientific exercise. More accurate figures can help you better mitigate risk, but a rough estimate is better than no estimate. And even quantifying orders of magnitude is better than having no measure at all. Measurement evolves and grows with experience. So rough estimates at the start of the design process become more exact and more realistic as the design evolves and as the solution is delivered and stabilized.

https://pixabay.com/en/architecture-blueprint-floor-plan-1857175/
https://pixabay.com/en/coffee-cup-coffee-cup-food-photo-2608864/
https://pixabay.com/en/blueprint-ruler-architecture-964630/

2, 3, and 4: Three-tier architecture



After the first pass design, you will begin the process of examining the design from different perspectives and implementing familiar patterns. The first three modules in this process use a three-tier approach to distinguish the separate parts of the solution.

The business logic layer incorporates all the transformations and actions that are critical to the purpose of the solution. You can think of the business logic as the application itself.

The data layer deals with the storage and retrieval of information.

The presentation layer deals with the flow of information from the user, through the service, and back out to the user. The transport of that data is networking. So although the presentation layer might sound like the user interface, in this class, it is more about the flow of information and how the data moves from one service to another.

Summing up this three-layer approach to design: The presentation layer gets the data from the user, moves it between components, and gets it back to the user. The business logic layer transforms the data, changes it, does something with it that is meaningful to the user. And the data layer stores and retrieves the data used by the business logic and moved by the presentation layer.

The result of these three design disciplines is a well-functioning and practical solution. However, that solution might not scale, it may not be secure, and it might not recover

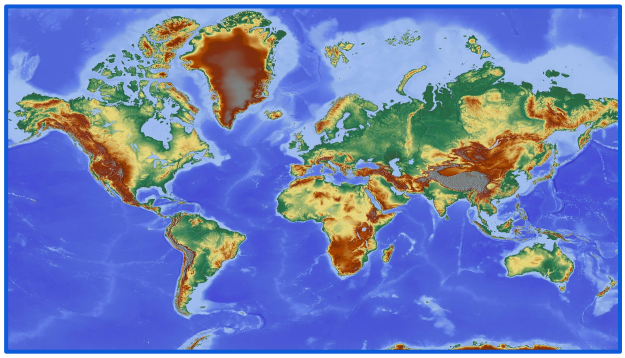
well from a catastrophe. So there is more design work to be done.

Three-tier design
Distributed design

https://stackoverflow.com/questions/7510396/business-layer-in-3-tier-architecture

https://pixabay.com/en/art-cogs-colorful-colourful-1866468/
https://pixabay.com/en/hand-keep-globe-earth-continents-1030564/
https://pixabay.com/en/binary-digitization-null-one-pay-2007350/

5: Resiliency, scalability, and disaster recovery



- Scalability
- Reliability
- Availability
- Disaster Recovery
- Resiliency

In the next design phase, you will examine the solution with respect to distributed design. Distributed design adds scalability, reliability, high availability, disaster recovery, and the ability to recover well, which is called resiliency.

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

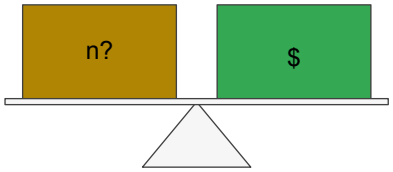
6: Security

- Privacy
- Denial of Service

After that, you will examine the solution from a security perspective. Service security is primarily about protecting against the user's data being compromised (destroyed or stolen), and the user experiencing a service outage (denial of service). There are many techniques available in service design to prevent these occurrences, limit them, or recover from them if they occur.

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

7: Capacity planning and cost optimization



In the next phase of the design process, you will examine cost optimization and capacity.

Some efficiencies exist at scale. In the final analysis you may choose to adopt an alternate technological solution or design for cost benefits.
You will also have the opportunity to do some capacity calculations for forecasting to see whether a design change is required to handle growth.

https://pixabay.com/en/coins-calculator-budget-1015125/

8: Deployment, monitoring and alerting, and incident response

- Launching
- Monitoring
- Alerting
- Responding

Developing a plan

Being prepared
How to respond to incidents

The design is complete, it has been implemented, and now it is time to deploy the solution.

In this part of the class you will explore monitoring, alerting, and incident response using some of the native tools in GCP, Deployment Manager, and Stackdriver.

The design and process elements you have incorporated into the solution enable you to monitor and adjust the solution during the period of stabilization.
After it has stabilized, you will have a service that can be operated, maintained, and adapt to changing demands.

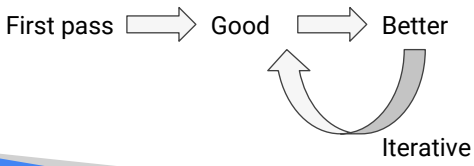
https://pixabay.com/en/pressure-measurement-2176274/
https://pixabay.com/en/book-handbook-manual-read-reading-148481/
https://pixabay.com/en/helmet-firefighter-bright-yellow-2074963/

Your design experience in this class

This class relies heavily on design exercises—challenges—to teach you how to think through design problems.

In this section, you will learn some of the expectations about how you should approach the design and what to expect during the design challenges.

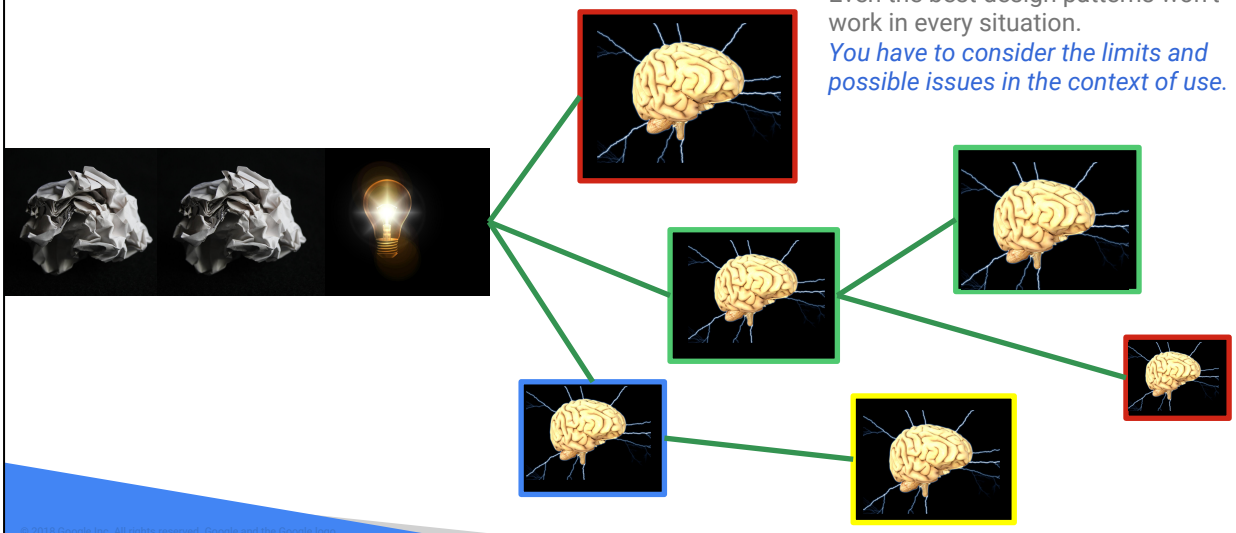
Ideas will evolve … iteratively



Through the process, you will see original ideas evolve and be replaced with new and better ideas.
You will come up with a lot of plans that will be supplanted by better ideas.
It's important for you, both individually and when working in teams, to rapidly recognize better ideas, verify that an idea actually IS better, and adopt it as quickly as possible.
This is not a linear destination, but an iterative process.

https://pixabay.com/en/pear-think-idea-questions-response-2010022/
https://pixabay.com/en/paper-screwed-up-paper-ball-crinkle-1484048/
https://pixabay.com/en/thought-idea-innovation-imagination-2123970/

There are no universal solutions, only contextual solutions



Even the best design patterns won't work in every situation.
You have to consider the limits and possible issues in the context of use.

There are no universal remedies for all situations. So as you learn about design patterns, take note of their best uses and their limitations and issues so you can consider the design in a realistic context.

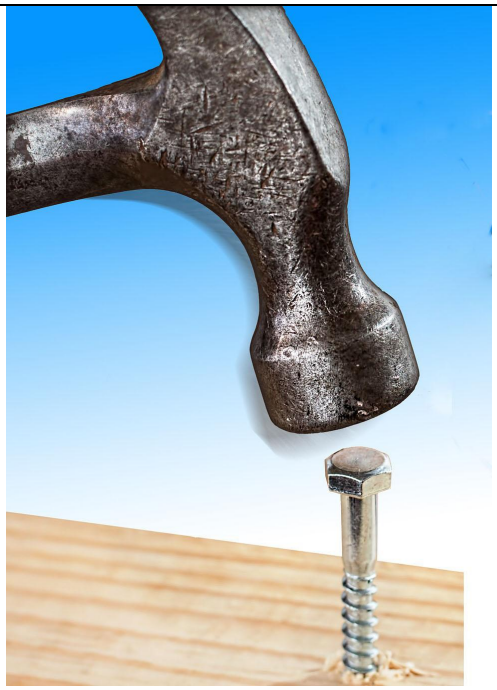
https://pixabay.com/en/paper-screwed-up-paper-ball-crinkle-1484048/
https://pixabay.com/en/brain-energy-thought-mental-1845940/
https://pixabay.com/en/thought-idea-innovation-imagination-2123970/

Recency bias

People tend to grab a great new idea or design pattern and use it for everything—*overuse it*—which can create problems later.

Consider which designs and tools work best in which contexts.

Every tool has an appropriate or best application.



One of the jobs of an architect, beyond delivering a solution, is to deliver a stable, maintainable, and reliable solution. Technologists tend to "fall in love" with their technology. This leads to overuse of the technology and application of it in circumstances where it's not the best fit or might not even be maintainable. Part of the job of the architect is to rein in the technology and make suggestions and decisions from a larger context: the system, the business, and the people.

https://pixabay.com/en/hammer-screw-shock-violent-895666/

Design process

1. Begin simply. Iterate.

Compose simple services.
Build **up** to complexity.
Iterate.

2. Plan for failure.

Avoid bottlenecks.
Avoid single points of failure.

3. Measure stuff.

Consider the hardware and services:
What are the limits?
How might they fail?

Architectural principles: Compose simple services and evolve them. Plan to avoid conditions that would lead to failure, such as bottlenecks, consider the hardware (or service limitations in the cloud). But also plan for resiliency and fast recovery when failure occurs.

https://pixabay.com/en/bricks-concrete-rock-stone-1839553/
https://pixabay.com/en/speedometer-kilometers-dashboard-309118/
https://pixabay.com/en/oops-reminder-post-note-sticker-1432954/

State and solution

State is the cornerstone of cloud-based design

Many design discussions in this class will focus on the question of where state information is located in the system and how state is being maintained.

This section discusses the problem of where to store system state and shows how that leads to a general solution for the design of large-scale cloud-based services.

Stateful versus stateless: Job shop and assembly line

All work in all systems can be divided up into "job shops" and "assembly lines." In a job shop, a single server, service, or unit performs multiple steps. In an assembly line each server, service, or unit performs one well-defined step and hands the work off to another worker for the subsequent step. These are the two extremes. There are variations where one station performs multiple steps as part of a conveyor belt of activities. If you think about the parts of your system in this way, it will highlight to you where work is handed off between stations and where multiple steps are performed within a single station. That will help you think about how to measure the work and how to adjust the relationships. For example, it is easier to queue up work between stations than within a single station. So there are natural breaks and tools. It makes it much easier to consider the operational performance characteristics of the overall system.

The job shop has to keep track of where it is in the process: STATE. The assembly line doesn't need to keep track of state. *However*, the units (microservices) have to coordinate with each other, so queuing and messaging becomes important as an alternative to keeping state. You can't get away from state in all cases. And in those cases, you have to try to make state scalable and reliable.

One nice aspect of this approach is that it applies equally to human organizations and to electronic systems. So you can define entire processes that include both automated and manual components.

https://pixabay.com/en/kitchen-culinary-cooks-helpers-81644/

https://pixabay.com/en/chef-smell-cook-spice-eat-939437/

Where is truth in a system?

A central control mechanism is a choke point on performance and a single point of failure.

How can parts of a system work together without a central control?

Truth = state.
This is a big subject in distributed design and game theory and even in military strategy.
How many people can be working from stale information or incorrect information and still have the organization accomplish its mission?

The "Byzantine Generals" problem (1982 paper):
https://en.wikipedia.org/wiki/Byzantine_fault_tolerance

https://pixabay.com/en/compass-compass-rose-north-south-2188475/

How should you deal with state?

The best state is no state

Make as much of your system stateless as possible.

Easier to apply more workers to a problem.

Easier to relocate tasks.

More fault tolerant, less to recovery when something breaks.

Sources of state

Objects in a database

Shared documents in memory

Key-value pairs for user

Critical state

Assuming you need state, what is the best way to implement it and manage it?

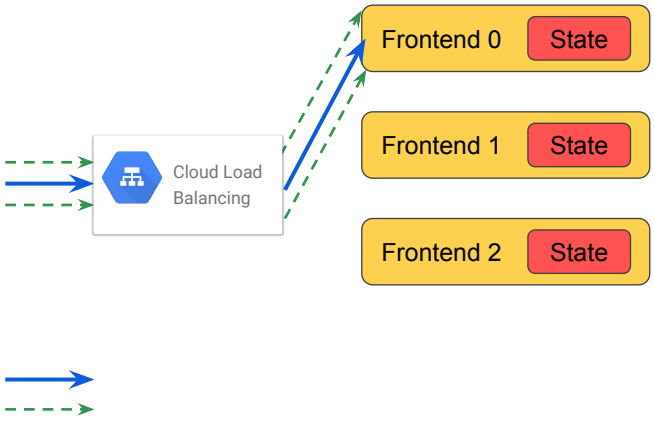
Stateless servers are easier to work with than stateful servers.

The worst case for state is a hotspot

State and Design

In the **worst case**, all requests for the same user route to the same frontend.

The balancer is doing the opposite of balancing: it is creating "hotspots."



Hotspots tend to make autoscaling less efficient because the capacity of one stateful server is consumed more rapidly than if the traffic is shared among multiple stateless frontend servers. The scaling decisions can become more rapid.

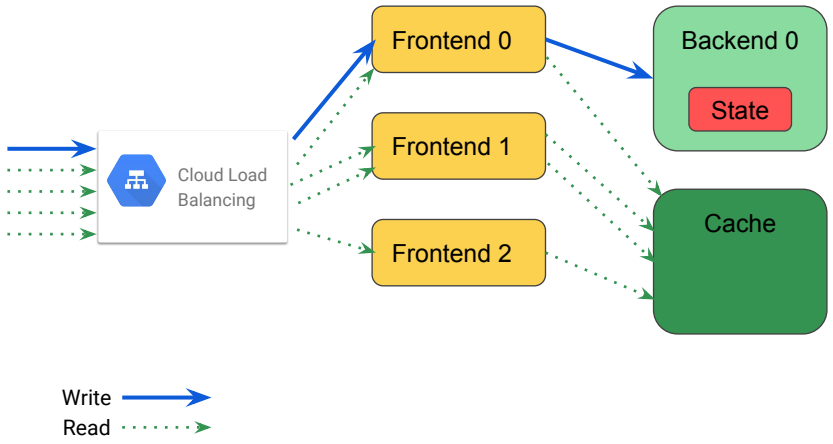
Stateful servers pushed to lower in the stack

Stateless frontends

Separate read and write paths

Caching of the read path

Push stateful servers lower in the stack



This design could create problems with strong consistency and caching. However, it is a common design that is useful in many situations.

Divide state to reduce the impact of an outage

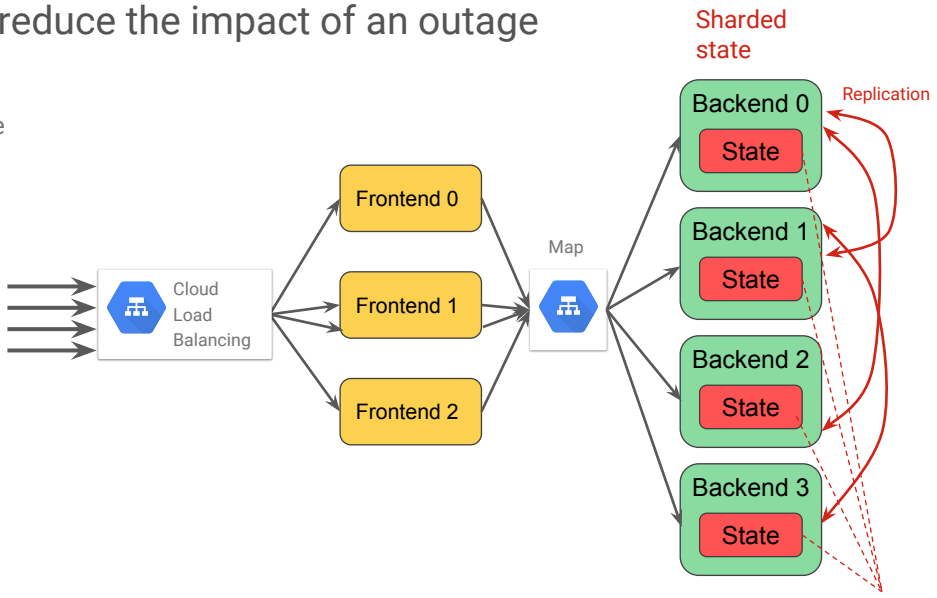
Divide and distribute state among stateful backend servers

Load balancer maps ID to backend

Single machine = 100% unavailability

Multiple machines = partial unavailability

Redundant state = ability to recover state

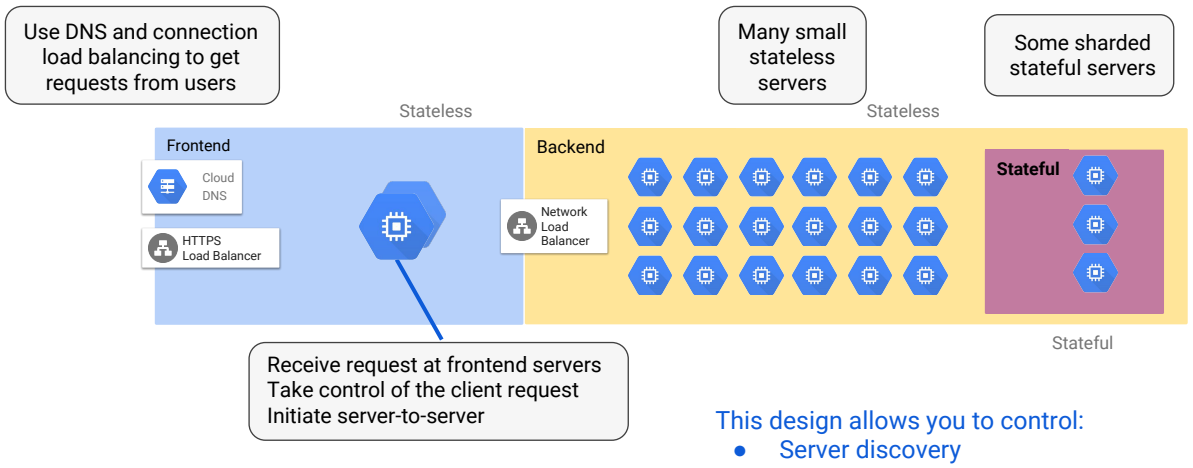


In this illustration, state is distributed to make it more reliable. The backend systems are each primary for serving state for a range of IDs of transaction and act as secondary or alternates for their peers. The main point here is that dividing state into smaller elements and mapping to it through a load balancer changes the durability, scalability, and reliability characteristics.

Note that the backend state are replicas and there is replication occurring between them. So there are backups if any single machine is lost. And the existence of the replicas is transparent to the front end servers. Sharding and replication is an approach that resolves the single point of failure and the performance limitations of a centralized control design.

Later in this course, some of these concepts of state are revisited as they are applied to data.

A general solution for large-scale cloud-based systems



This design allows you to control:

- Server discovery
- Request balancing
- Throttling

Throughout this class you will learn in detail, step-by-step, how to arrive at a design that has similar features to this one: Stateless servers combined with sharded stateful servers. Along the way you will learn about many nuances, features, and variations in the design and how to make design decisions.

In network communications the phrase "locally terminating" a session, means that the communication is acknowledged and a new communications loop is started with the backend. This gives you the flexibility of controlling the server-to-server activities without also simultaneously trying to control the client-to-server communications.

Measurement

Defining a Service Level:

- Service Level Indicators (SLI)
- Service Level Objectives (SLO)
- Service Level Agreements (SLA)

Defining a Level of Service

Service Levels help you:

- Qualify and quantify the user experience.
- Balance investment in reliability vs. features
- Set user expectations for availability and performance.

To define a level of service for a workload:

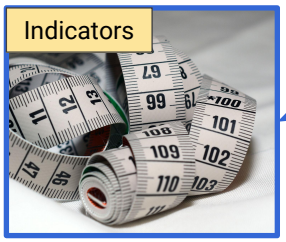
1. Identify the Service Level Indicators (SLIs).
2. Determine an appropriate Service Level Objective (SLO).
3. Optionally, establish a Service Level Agreement (SLA).

As the designer of your service, it's up to you to define how your service should look and feel to your users. It's equally important to establish metrics that tell your users what they can expect from your service in terms of reliability and performance. We'll call this the Service Level. To define a Service Level, you need to identify what the users care about. Over the next few slides, we'll discuss each of these terms in more detail.

Our first step is to identify the Service Level Indicators, or SLIs. For example, the availability of a web page, or how fast the system responds when the users click a button. Next, we determine the threshold of pain for each of these SLIs, meaning at what point the users will decide that the service isn't working properly. We call this the Service Level Objective, or SLO. Finally, some services are so critical that a formal business agreement will exist between the provider and the users to guarantee a specific behavior, and to compensate the users if the service fails to meet this expectation.

https://pixabay.com/en/user-satisfaction-user-feedback-2800863/

Service Level Indicators, Objectives, and Agreements



An SLI is a measurable attribute of the service that represents its availability and/or performance.

The SLO defines how the service should perform.

It's based on the users' experience, which is measured using the SLIs.

An SLA is a binding contract to provide the customer compensation if the service doesn't meet specific expectations.

The SLA is a more restrictive version of the SLO.

Chances are that you've seen terms like Service Level Agreement, or SLA. Perhaps you've even wondered how these figures are determined, or more importantly, how they are monitored. Service levels define how a given service or product should behave. There are a number of factors that often go into service levels, ranging from availability specifications - such as 99.99% uptime, or a performance specification, such as a minimum response time of 1 second. Service levels help set the user's expectation, and help keep their expectations in line with the design of the service.

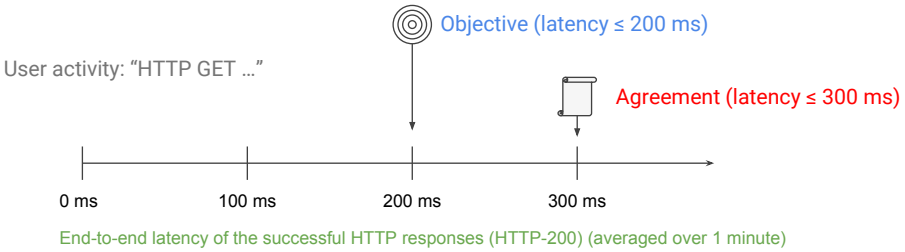
Let's start on the left with Service Level Indicators. SLIs should be directly observable and measurable by the users. SLIs should not be internal system metrics, such as server CPU utilization, or the number of server failures. This is done for two reasons: One, the users can't directly measure those internal factors because they are on the OUTSIDE of the system. And two, utilization isn't really an effective measure of the usability of a system. Instead, utilization is a metric best used by auto-scalers to maintain a consistent user experience.

Once we have identified WHAT the users care about, we need to quantify their threshold for pain. This is represented as the Service Level Objective, or SLO. The SLO is a threshold value for an SLI, and the threshold should be set at the lowest or poorest level of service, where the users will still consider the service to be in good working order. That is to say, the SLO represents the point at which a user would consider opening a support ticket because the service failed to meet his/her expectation.

Going one step further, we have SLAs. As previously mentioned, some services are so critical that the loss of the service could represent loss of money for the customer, or even loss of life in extreme situations, such as autonomous cars or air traffic control. These contracts define an even more restrictive level of service that is lower or poorer than what is defined in the SLO. The key difference here is that the SLO is a soft target used by the owner of the service to set expectations. The SLA is a business contract that grants a user compensation if the service falls below a certain threshold. Therefore, developers and operations teams strive to maintain the SLO, which inherently meets the SLA figures.

https://pixabay.com/en/writing-pen-man-boy-male-ink-1149962/
https://pixabay.com/en/massband-tape-measure-measure-gage-1189947/
https://pixabay.com/en/darts-dart-game-bull-s-eye-target-155726/

Example: SLI, SLO, and SLA



Service Level Indicator (SLI): The latency of successful HTTP responses (HTTP-200).

Service Level Objective (SLO): The latency of 99% of the responses must be ≤ 200 ms.

Service Level Agreement (SLA): The user is compensated if 99th percentile latency exceeds 300 ms.

Here we have an example web application where we have defined a level of service using both SLO and SLA figures. In this case, we are quantifying the performance of the application using the latency of the HTTP responses as a Service Level Indicator. The SLO is set at 200ms, and represents the lowest performing point at which the system is still usable, and simultaneously performing acceptably to the users. Improving the service performance beyond 200ms (such as to 100ms) is not required or beneficial because there might be circumstances beyond our control between the user and the service, such as poor Wifi service, which would mask any benefits gained by reducing the SLO further.

In this example, the service provider has also established an SLA value at 300ms, which is a lower performance value than the SLO of 200ms. The SLA represents the threshold at which the service violates the written expectation of performance or availability, and entitles the user to compensation from the service provider.

Service Level Indicators (SLI)

An SLI is a quantitative measure of an attribute of the service—one that the users care about—such as:

- Throughput
- Latency
- Availability
- Correctness

Not all statistics should be SLIs.

An SLI should:

- Represent the users' experience.
- Reliably represent the health of the system.

Service Level Indicators are a quantitative measure of an attribute of the service. SLIs could represent things such as throughput, latency, availability, or even correctness.

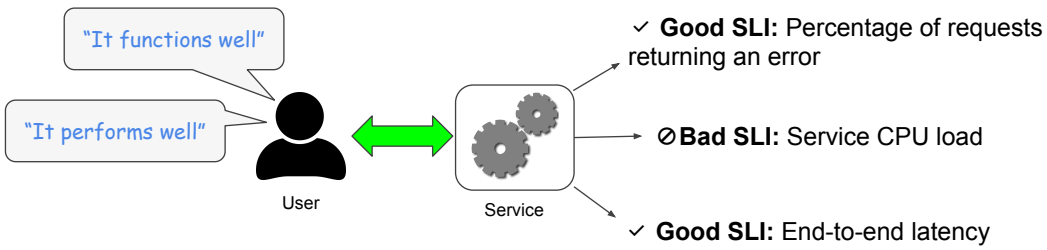
Be careful not to include internal system metrics such as CPU utilization or disk latency in your SLIs because the users cannot directly measure these values. Users can only measure what they see and feel, so stick with values that are both meaningful to the users and that are tied to the core functionality of the service.

https://pixabay.com/en/massband-tape-measure-measure-gage-1189947/

Identifying the SLIs

When identifying SLIs, consider:

- What functions or behavior do the users care about?
- How do the users **quantify** a good or bad experience?



Consider the following example. The users know if the service functions properly because they are not receiving errors. They know it performs well because the latency between operations is very quick. However, they have no idea how heavily loaded the CPUs of the servers are, nor do they care if the first two components are being met.

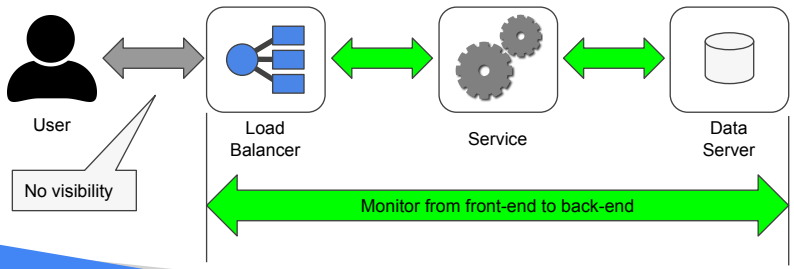
Good SLOs are based on SLIs that effectively represent the service's user experience.

https://pixabay.com/en/cog-gears-icon-settings-simple-2766559/
https://pixabay.com/en/user-person-people-profile-account-1633249/
https://pixabay.com/en/massband-tape-measure-measure-gage-1189947/

SLI considerations

To properly determine the availability/performance of a service, you might need to observe factors that surround the service.

For instance, it might be difficult to monitor the performance at the client, so you should monitor it from a load balancer or a data server instead.



It's also important to include other factors when determining the SLI for your service. To provide an accurate measurement of the latency a user would experience, we need to monitor additional systems. In this case, to know the true latency of our application we must account for the latency of the load balancer and that of the data server.

Note that in this case we don't have visibility of the user's internet connection, so our SLI will be the end-to-end latency from the front-end load balancer, through our service, and back to the data server.

https://pixabay.com/en/cog-gears-icon-settings-simple-2766559/
https://pixabay.com/en/user-person-people-profile-account-1633249/

Service Level Objective (SLO)

An SLO is the threshold at which point you will change your behavior to improve the reliability of the service.

For example, you might:

- Change the velocity of updates.
- Change your development process.
- Devote engineering efforts to reliability.

A good SLO focuses on the users' experience instead of the internal system.

An SLO represents the users' "pain tolerance", where "pain" is a measurement of an SLI.

Service Level Objectives are thresholds. They represent the point at which you must do something to improve the reliability or performance of the service. Sometimes, the thing you need to do isn't fixing a component. Sometimes it's delaying a fix, or postponing updates because the system has been offline too much. At other times, it might be necessary to devote engineering time to resolve the problems. And as we said before, SLOs are different from SLAs. An SLO defines the users' pain tolerance to performance or availability issues; whereas an SLA is a legal contract. SLAs are known to the consumers of the service, but SLOs might not be; though there have been cases where providers have disclosed their SLOs just to ensure that users do not become overly dependent upon the service, beyond the point that the designers of the application or service originally intended.

https://pixabay.com/en/darts-dart-game-bull-s-eye-target-155726/

Determining the SLO

SLOs are driven by business requirements, not necessarily current performance.

Minimize the number of SLOs for your service, although if you add features you might need to add SLOs.

Focus on SLIs reflected in the four golden monitoring signals:

- Latency
- Traffic
- Errors
- Saturation (resource usage)

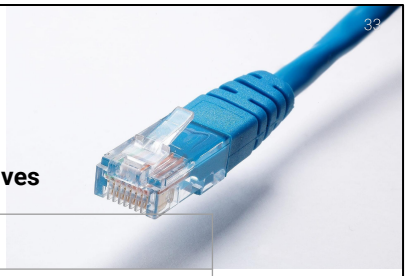
Examples of an SLO:
Your service must respond successfully to user requests 99.9% of each month, and the latency of 95% of those responses must be less than 400 ms.

- **SLI:** The "HTTP-200" responses
- **SLO:** (This SLO has two components)
  1. Receiving an HTTP-200 99.9% of the month
  2. The 95th percentile (p95) of responses is ≤ 400 ms.

Contrary to popular belief, SLOs aren't set at the maximum performance a system can achieve. Rather, the business will determine an appropriate level of performance or reliability that the provider can offer without making the service too available or too responsive. Keep in mind that your SLOs might change as you improve your service, or add features to it. Also, focus on the four golden monitoring signals when creating your SLOs: Latency, Traffic, Errors, and resource saturation.

A well-defined SLO will often describe a subset of responses, such as the 95th percentile of responses must be less than 400ms. Or the service must be 99.9% available over the last 30 days using a rolling calendar. This avoids the situation of outlier statistics skewing the data, and of major outages in January preventing software updates in December because the availability target was exhausted at the beginning of the year.

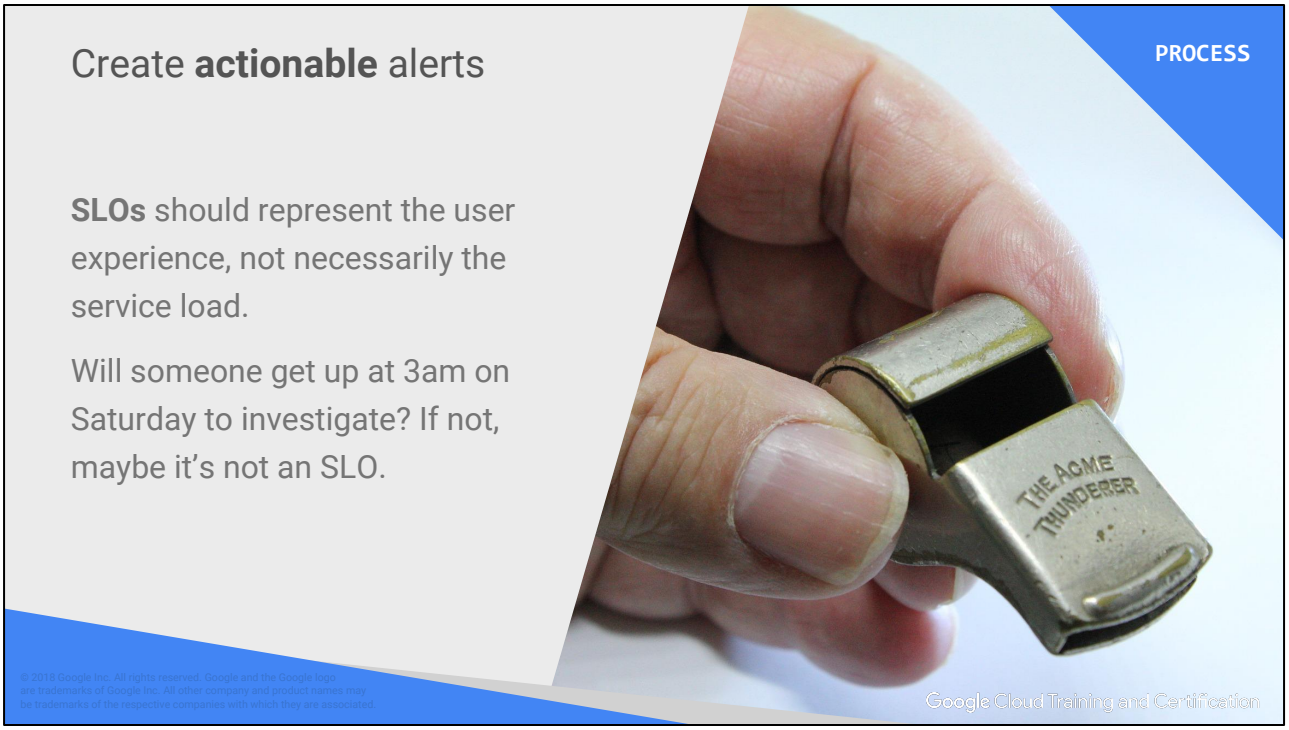
Useful concepts: Real beats ideal



Realistic Objectives	Impossible Objectives
Error budget	~~Error free~~
Unplanned downtime	~~Zero downtime~~
Risk tolerance	~~Zero risk~~
Target level of availability	~~Always available~~
Types of failures	~~No failures~~
Defined classes of service	~~One class of service: perfect~~

When designing your SLOs, it's important to put reality above utopia. While we'd love to have a service that is up 100% of the time, that also means that the service can never be updated. Google has adopted an Error Budget methodology. Instead of assuming that every service will run perfectly, services are given error budgets for their SLOs that operate a bit like spending money. Both planned and unplanned downtime count against this budget. As the days go by each month, the error budget is replenished, giving the development and operations teams the breathing room needed to implement changes and apply updates.

https://pixabay.com/en/network-cable-network-cable-data-2245837/



General guidance: measure the system and all its parts, but alert only on problems that either are currently causing or will soon cause user pain. Alerts replace the manual operation of watching the application 24x7. Alerts are designed to inform you about an issue or outage before hearing about it from users.

We will discuss monitoring in more detail in a later module, but it's important to ensure that all SLOs are based on the user experience rather than an internal system metric. For instance, if a service becomes unavailable at 3am, but no one notices, perhaps the SLO shouldn't be based on a 24-hour day.

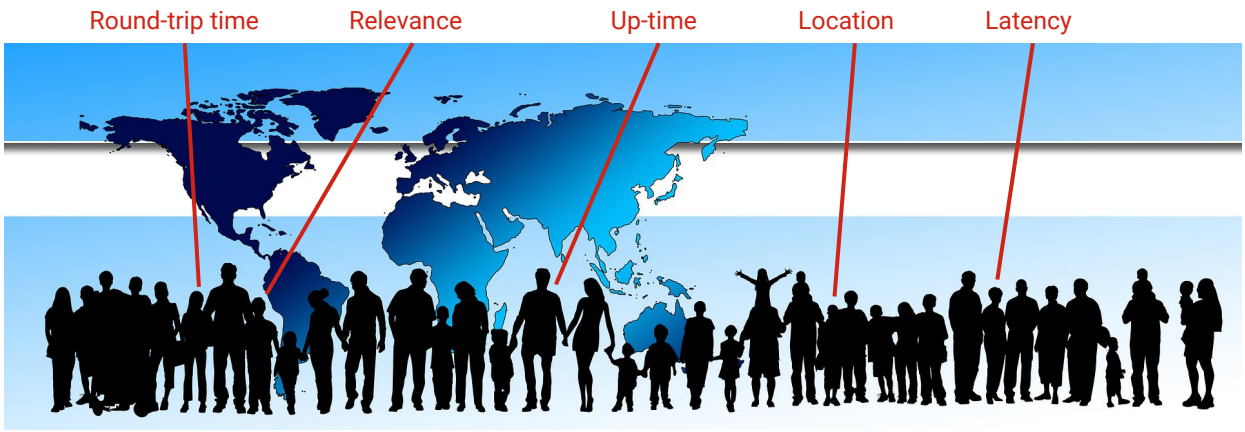
https://pixabay.com/en/whistle-attention-warning-referee-2477714/

Finally we come to the one of the most overused terms in the industry, the Service Level Agreement. An SLA is not the minimum point at which a service is considered usable. There are some services that are so mission critical that the service provider must provide a written guarantee that the service will perform above a certain specification. That specification is the SLA, and breaking the SLA grants compensation from the provider to the users.

Keep in mind that not all services have SLAs, but all services should have an SLO. It's also possible that providers might not publicly disclose their SLOs for fear that users might incorrectly associate the SLO with an SLA.

https://pixabay.com/en/writing-pen-man-boy-male-ink-1149962/

User personas for indicators: Define your users to serve them better



What do groups of users really care about? How can you identify, define, and measure that?

User personas are a concept that comes from user experience (UX) design and originated in marketing. If you don't know who your user is, how can you develop good measurable SLIs?

A user persona is a representation of the goals and behaviors of a group of users. It can be derived from data collected from users and characterizes patterns, goals, skills, attitudes, behaviors, and their environment. For example, a user of an application that helps identify oil pipeline fractures in the field is using the application in a certain environment: usually with satellite-level network delays. The concept is to use what you know about the users to develop smart SLIs. It can help to conduct interviews or surveys of your users to find out directly from them what they care about. In some cases, their real interest can be surprising or may have different stresses or conditions than imagined. Another good suggestion if you are using surveys is to study statistical survey techniques. It is important to get a sample that is representative of your actual users, or the results will be invalid or misleading.

https://en.wikipedia.org/wiki/Persona_(user_experience)
https://pixabay.com/en/human-banner-header-humanity-1375492/

Gathering requirements

A design starts by asking questions.

In this section you will learn about asking questions that can help define Service Level Objectives and will be helpful thinking through the design decisions later.

Gather requirements
Step 1: **Qualitative** requirements

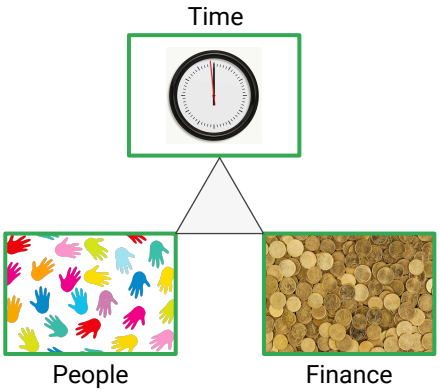
Why	Why is the system needed or desired? What problem does it solve?
Who	Who are the stakeholders and who are the users? What do they care about and not care about? Who is developing this? What is in scope? What is feasible and what is not feasible?
What	What does the system need to do? What are the priorities, required versus nice-to-have?
When	When do the users need and/or want the solution? Are there any time/value tradeoffs, such as faster versus better?

Notice that "How" is not listed. This is because implementation is not part of gathering requirements and should not be considered yet.

Sometimes the first requirements you get are not explicit or detailed enough, and you will need to dig into them to get correct and usable information.

Gather requirements
Step 2: **Quantitative** requirements

RESOURCES



CONSTRAINTS

Time
- Operational time constraints
- Cost of downtime

Data
- Cost of data lost
- Data volume
- Throughput
- Freshness
- Groups of data

Users
- Number of users
- Location of users

Identifying quantitative requirements will help with non-abstract (sizing and dimensioning) design later in the design process.

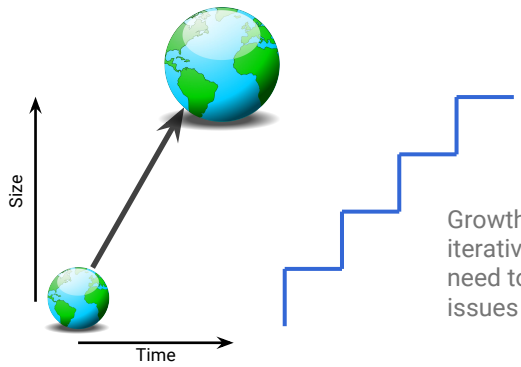
A perfect solution might be too expensive. This is the beginning of establishing the trade-offs you will need to make in the design.

It is important to understand the time/value of data and how "fresh" data must be to be useful in the system. These translate to time constraints.
Also, if there are different groupings of data, even if they are identically formed, but some are dated, can these groups be treated differently?

https://pixabay.com/en/clock-black-wall-isolated-sign-163580/
https://pixabay.com/en/hands-background-black-colorful-565603/
https://pixabay.com/en/euro-coins-currency-money-yellow-1353420/

Gather requirements
Step 3: **Scaling** requirements

SCALE



Growth in scale tends to be iterative and new solutions need to be found as scale issues arise.

LIMITING FACTORS

What will limit your ability to grow your application?

Which resource constraints are important to pay attention to?

How can you design to push past these limits when necessary?

- Scaling
  - Determine the expected traffic at launch.
  - Determine the timeline for scaling and safety factors, and design appropriately. For example, design for 10-100x growth.
- Limiting factors
  - Bottlenecks: CPU, RAM, IOPS, disk?
  - Profile to get relative costs and identify limiting resource

Load tests will help you plan and help prevent over allocation and later regression.

Eric Schmidt, former chairman of Google, is known for saying that things break every time a system scales to a new order of magnitude.

https://pixabay.com/en/world-earth-globe-planet-153534/

Gather requirements
Step 4: **Size** requirements

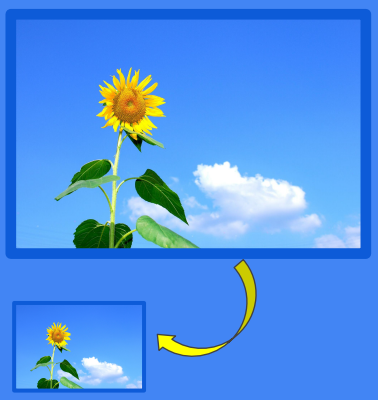
Which factors are important to the size of the solution?

- Dimensions
- Replication
- Rate of change

Which dimensions scale?
Which dimensions are dominant?
How many copies are necessary for redundancy and for availability?
How much data now?
How much data in the future?
What is the rate of change of the scale?
Replication can be costly.
Watch out for exponential rates of change.

https://pixabay.com/en/burj-tower-skyscraper-dubai-2196344/

Introducing an example photo service



This section introduces the thumbnail photo service, an example application that will evolve in design throughout the class.

Apply what you learned in this module.

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

Gathering requirements

Users = On the internet, using a web browser from a computer, care about uploading images and getting smaller sized thumbnails in return.

Speed = < 1 minute, beating estimated thumbnail time on a local computer.

Resources = Have to go with the information on hand. There will be some questions that can't be answered without research/testing. Best guess.

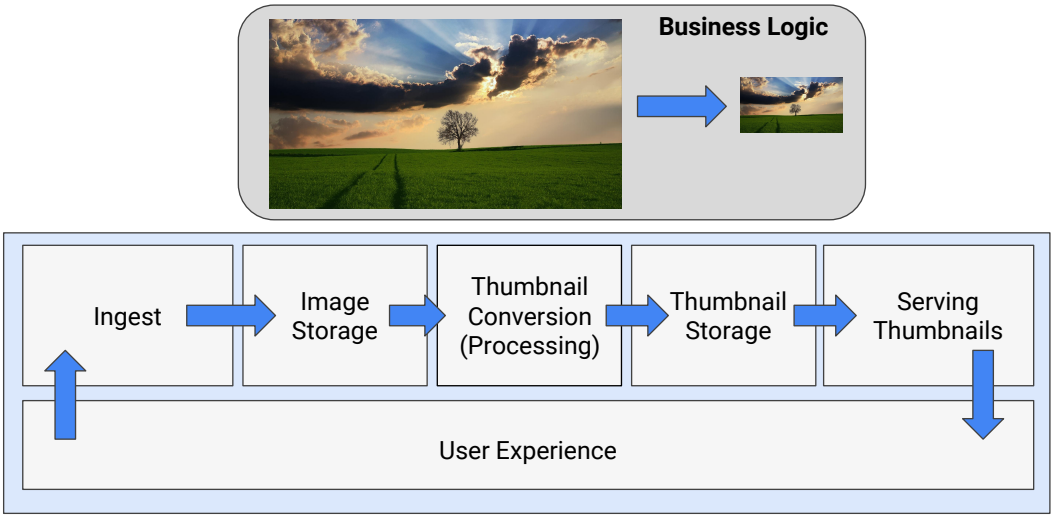
Scale = This information will be given in the design exercise. Generally, the service will become more popular over time and grow larger.

Size = The application can be served to all users from one location, at least to start.

Availability = Nobody relies on it. Users would like it to be available most of the time, but if it is down some of the time for development it is not a big problem.

Compare our proposed thumbnailing service to launching an image editor on a local computer, opening an image file, shrinking it in size, and outputing a thumbnail. That takes about 1 minute. If the service is faster than that, it is useful because it saves time. Faster is better. But the difference between 1 second turn-around time and 3-second turn-around time with the current expected users is not significant.

Thumbnail service



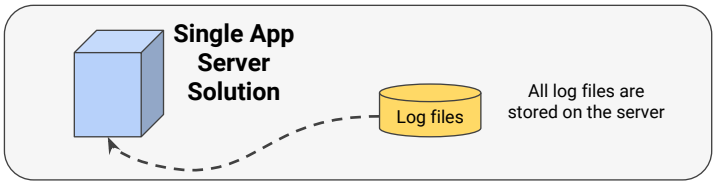
This example is a thumbnail photo service.

There are several activities in the business logic. The first is to ingest images, which is a slow transfer of an image file into the image storage. (Low queries per second (qps) and long duration transfer). Then the thumbnail conversion process occurs, creating a small preview image from the original. The thumbnail preview image is stored in thumbnail storage. Finally, the thumbnail images are served to users. Serving the thumbnails may require a high query per second response (many users/uses) and short duration transfer downloads.

https://pixabay.com/en/countryside-tree-landscape-sunlight-2175353/

Define Service Level Objectives (SLOs)

FIRST DESIGN



The Service Level Objectives will evolve as the service grows and changes.

The service will stay up 23 hours a day. It can be taken offline 1 hour each day to publish software upgrades or make changes to the service.

SLI: service online/offline

SLO: 95.83% service availability

Given the requirements expressed as an SLO of availability of 95.83% uptime, and an SLI indicator that the server is either up or down, the service can be implemented on a single compute. The service generates log files that are stored on the server and used for analysis if errors occur.

The single app server solution could be implemented as a physical computer, or it could be implemented as a Compute Engine instance.

The SLI is a check if the service is online or offline.

Test before and during launch

PROCESS

Pre-production tests

Unit tests

Integration tests

System tests

Stress test

Production tests

Rollout (staged) with user acceptance, A/B testing

It is not a good idea to deploy a new or updated service without appropriate testing. Testing your service is the first thing you can do to ensure your service is reliable.

Reference: SRE Book: Chapter 17 - Testing for Reliability

Unit tests: testing a unit of software for correct behavior independent of the larger system.

Integration tests: testing of assemblies of units. Usually includes "dependency injection", by replacing adjacent components with ones that don't meet requirements.

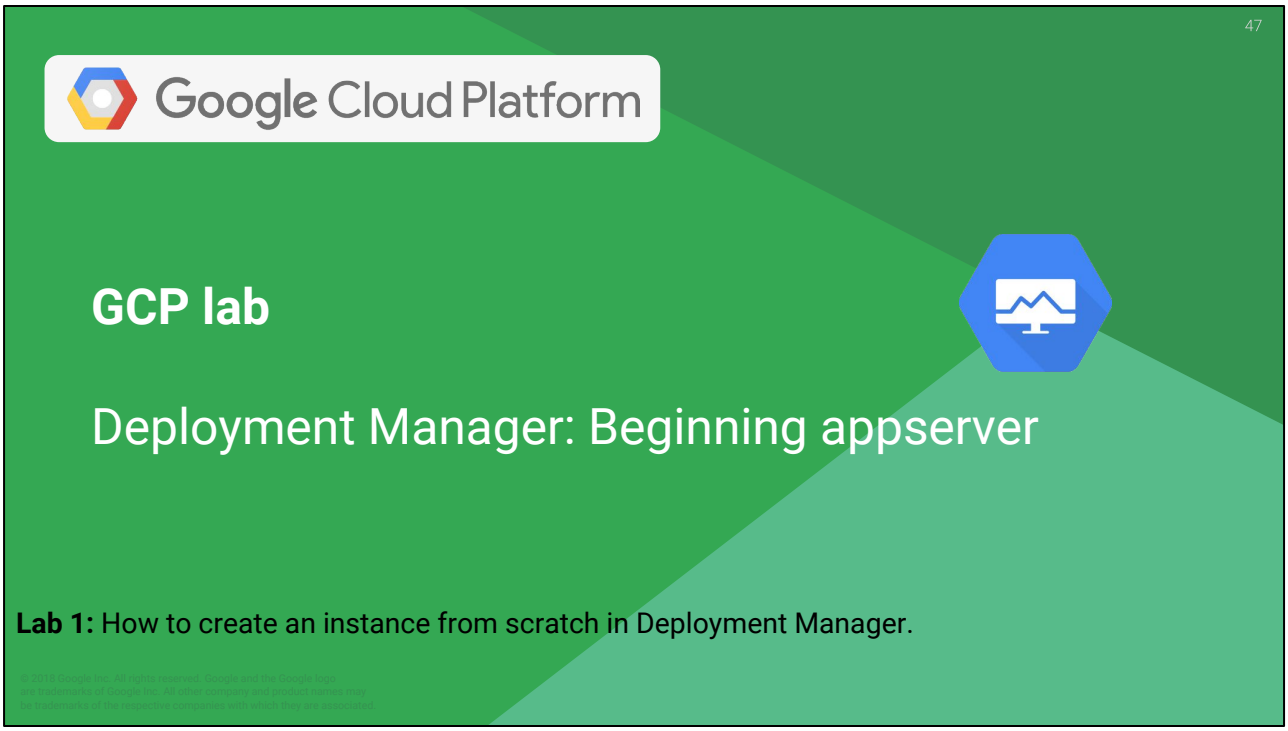
System tests:

- smoke test: test critical system behavior
- performance test: test various loads simulating a lifecycle to verify expected behavior and not unexpected degradation or resource consumption
- regression test: test of already fixed bugs to ensure they have not resurfaced

Production tests: testing in a deployed environment

- Rollout (staged) testing + user acceptance (canary test)
- Configuration validation: a test that verifies each configuration file is correct
- Stress test: push the limits of a service to verify that it degrades rather than fail

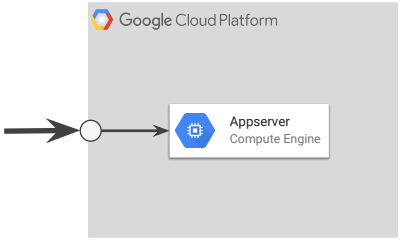
https://pixabay.com/en/laptop-notebook-computer-black-158648/
https://pixabay.com/en/smoke-fumes-black-white-curve-298243/



This is the first of several hands-on Deployment Manager labs.

This first lab is directly relevant to the example photo application. The labs in this class focus on teaching you what you need to know to use Deployment Manager as an effective Cloud Architect tool.

Lab Deployment



In this lab, you will create a Deployment Manager template for the Appserver. The goal is to create a single virtual machine.

After the machine exists, developers will use the machine to develop, host, test, and operate the thumbnail service.
So you don't need to load any software on the server, you just need to create it.

Such a server could be created from the GCP console, or from Cloudshell, or using the Cloud API from a program.
In this lab you will use Deployment Manager to create the instance.

In this trivial example, it would be faster and easier to create the instance using one of the other methods. However, as the service grows in complexity and requirements, the value of using Deployment Manager will become evident. And learning to make complex deployments using Deployment Manager requires starting by creating a single instance.

Google Cloud Training and Certification