

Fingers digital



Treggs

Beyond your horizon!

Design and Process on

Google Cloud

Brief description

Treggs is an American global travel agency that wants to build a scalable e-commerce platform to serve a global customer base.

Main features

- Travelers can search and book travel (hotels, flights, trains, cars)
- Pricing will be individualized based on customer preferences and demand
- Strong social media integration with reviews, posts, and analytics
- Suppliers (airlines, hotels, etc.) can upload inventory

Roles of typical users

- | | |
|----------------------|------------|
| • Customer | • Traveler |
| • Inventory supplier | • Manager |

User personas

Donna

Donna is a busy businesswoman who likes to take luxury weekend breaks, often booked at the last minute. A typical booking comprises a hotel and flight. Recommendations play a major role in the choice Donna makes, as does customer feedback. Donna likes to perform all operations from her phone

User personas

Maxwell

Maxwell is a student who likes to travel home to visit parents and also takes vacations twice yearly. His primary concern is cost, and he will always book the lowest price travel regardless of convenience. Maxwell has no loyalty and will use whichever retailer can provide the best deal.

User stories

Search for flight and hotel

As a traveler, **I want to** search for a flight-hotel combination to a destination on dates of my choice **so that** I can find the best price.

User stories

Supply hotel inventory

As a hotel operator, **I want to** bulk supply hotel inventory **so that** Treggs can sell it on my behalf.

User stories

Analyze sales performance

As a Treggs manager, **I want to** analyze the sales performance data of all our suppliers **so that** I can identify poor performers and help them improve.

Defining SLIs and SLOs

User story	SLO	SLI
Search hotel and flight	Available 99.95%	Fraction of 200 vs 500 HTTP responses from API endpoint measured per month
Search hotel and flight	95% of requests will complete in under 200 ms	Time to last byte GET requests measured every 15 seconds aggregated per 5 minutes
Supply hotel inventory	Error rate of < 0.00001%	Upload errors measured as a percentage of bulk uploads per day by custom metric
Supply hotel Inventory	Available 99.9%	Fraction of 200 vs 500 HTTP responses from API endpoint measured per month
Analyze sales performance	95% of queries will complete in under 10s	Time to last byte GET requests measured every 60 seconds aggregated per 10 minutes

Note:

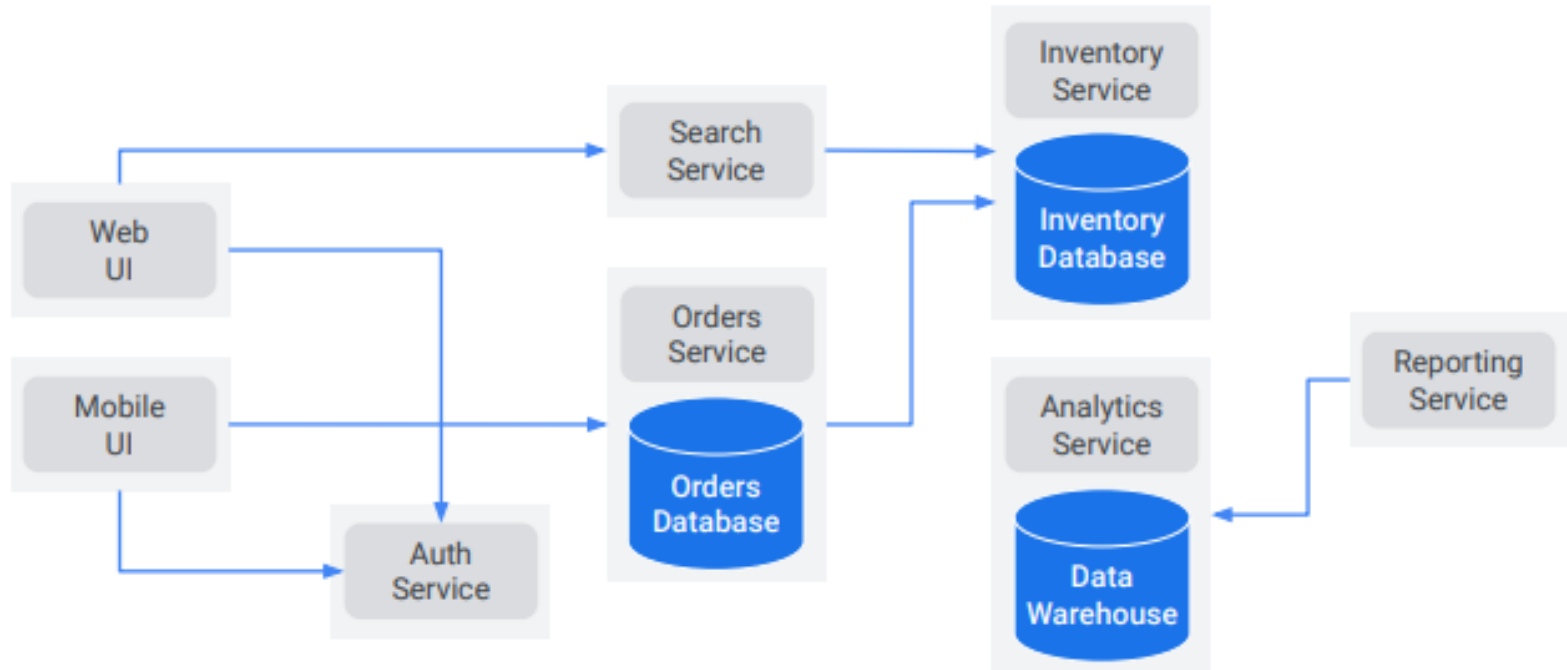
- The SLI describes what we are going to measure and how:

for example, the “Fraction of 200 vs 500 HTTP responses from API endpoint measured per month.” This example is a way of measuring availability.

- The SLO represents the goal we are trying to achieve for a given SLI.

For example, “Available 99.95%” of the time.”

Microservices design for the application



Note:

There isn't really one and only one right way to design an application.

- They are separate services for the web and mobile UIs.
- There's a shared authentication service
- Microservices for search, orders, inventory, analytics and reporting.
- Each of these services will be deployed as a separate application.
- Where possible we want stateless services, but the orders and inventory services will need databases
- The Analytics service will provide a data warehouse.

This might make a good starting point, and could be adjusted as needed when implementing the application

REST APIs design

Service name	Collections	Methods
Search	Trips (flights + hotel combination)	find save
Inventory	Items (flights + hotels)	add search get remove
Analytics	Sales	analyze get list
Order processing	Orders	add get list update

Note:

Obviously, our API would be larger than this, but in a way the APIs are all more of the same.








- Each service manages and makes available some collection of data.
- For any collection of data there are a handful of typical operations we do with that data. This is similar to Google Cloud APIs. For example in Google Cloud, there is a service called Compute Engine, which is used to create and manage virtual machines, networks, and the like. The Compute Engine API has collections like instances, instanceGroups, networks, subnetworks, and many more. For each collection, various methods are used to manage the data.

Storage characteristics

Service	Structured or Unstructured	SQL or NoSQL	Strong or Eventual Consistency	Amount of Data (MB, GB, TB, PB, ExB)	Read only or Read/Write
Inventory	Structured	NoSQL	Strong	GB	Read/Write
Inventory uploads	Unstructured	N/A	N/A	GB	Read only
Orders	Structured	SQL	Strong	TB	Read/Write
Analytics	Structured	SQL	Eventual	TB	Read only

Each of these services has different requirements that might result in choosing different Google Cloud services.

Google Cloud Storage and Data Services

Service	 Persistent Disk	 Cloud Storage	 Cloud SQL	 Firestore	 Cloud Bigtable	 Cloud Spanner	 BigQuery
Inventory				X			
Inventory uploads		X					
Orders			X				
Analytics							X

Note:




- For the inventory service Cloud Storage is used for the raw inventory uploads.
- Suppliers will upload their inventory as JSON data stored in text files. That inventory will then be imported into a Firestore database.
- The orders service will store its data in a relational database running in Cloud SQL.
- The analytics service will aggregate data from various sources into a data warehouse, for which BigQuery will be used

Network characteristics for the services

Service	Internet facing or Internal only	HTTP	TCP	UDP	Multi-Regional?
Search	Internet facing	X			Yes
Inventory	Internal		X		No
Analytics	Internet facing	X			No
Web UI	Internet facing	X			Yes
Orders	Internal		X		No

The inventory and orders service are internal and regional using TCP. The other services need to be facing the internet using HTTP and deployed to multiple regions for lower latency, higher performance, and high availability to our users who are in multiple countries around the world

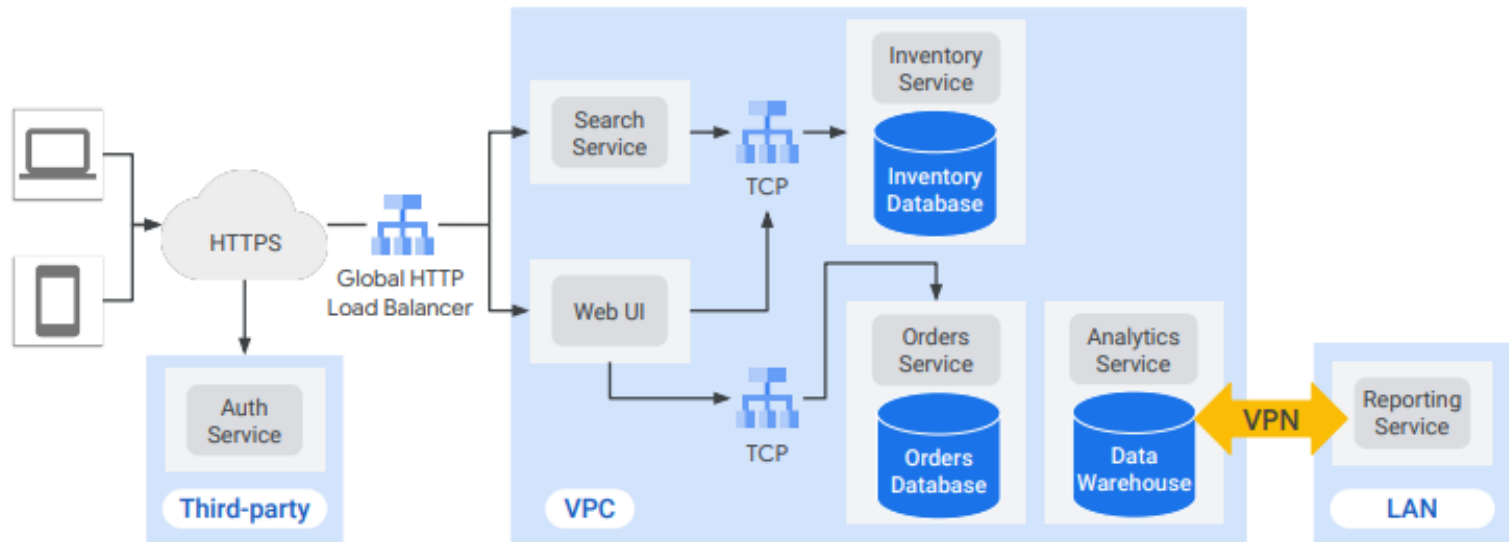
load balancers for the services

Service	 HTTP	 TCP	 UDP
Search	X		
Inventory		X	
Analytics	X		
Web UI	X		
Orders		X	

Based on those network characteristics, the global HTTP load balancer was chosen for the public-facing services and the internal TCP load balancer for internal-facing services

Network diagram

This depicts how the services will communicate over the network. Include regions, zones, load balancers, CDN, and DNS if applicable.



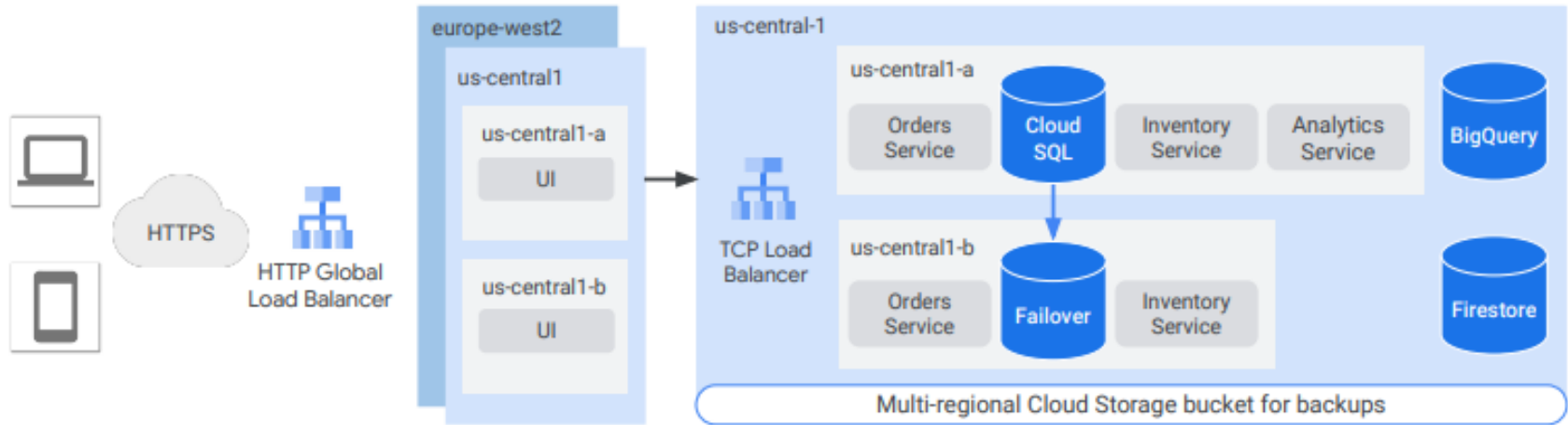
Note:

- User traffic from mobile and web will first be authenticated using a third-party service.
- Then a Global HTTP Load Balancer directs traffic to our public facing Search and web UI services.
- From there, regional TCP load balancers direct traffic to the internal inventory and orders services.
- The analytics service could leverage BigQuery as the data warehouse with an on-prem reporting service that accesses the analytics service over a VPN.

This might be good enough to start, and could be refined once the implementation start

Designing reliable, scalable application

Even if some services are down, we want the web frontend of our application to be available nearly all the time. We also want the website to be fast with very low latency to users all over the world.



Note:

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

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

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

Note:

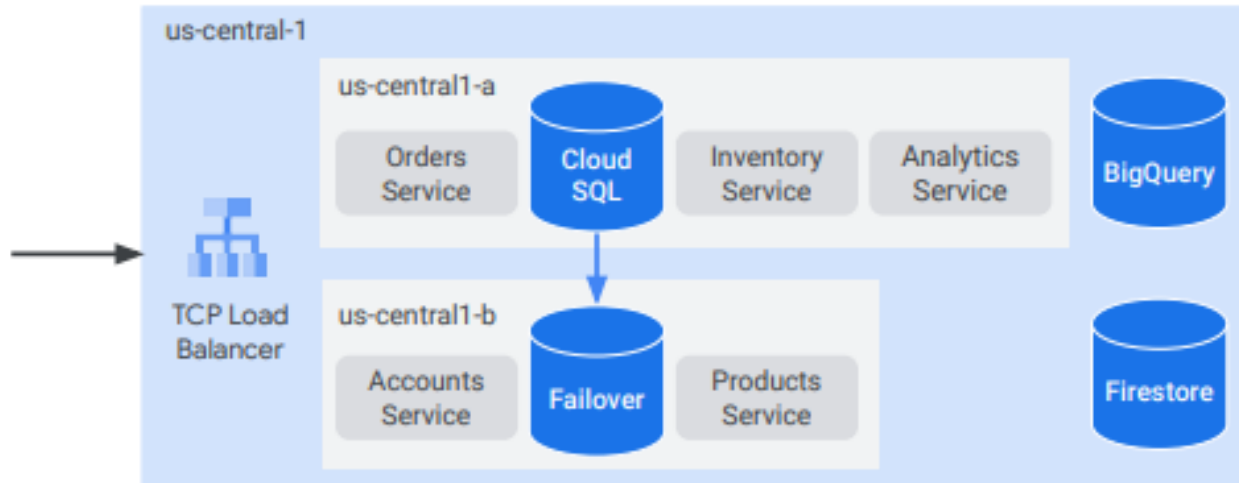
To ensure high availability, I've decided to deploy the Orders and Inventory services to multiple zones. Because the Analytics service is not customer-facing, I can save some money by deploying it to a single zone. I again have a failover Cloud SQL database, and the Firestore database and BigQuery data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

The Cloud SQL database needs a failover for high availability. Because BigQuery and Firestore are managed, we don't have to worry about that.

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

Disaster recovery scenario

I've deployed for high availability by replicating resources in multiple zones. However, to meet regulatory requirements, I created a plan to bring up the application in another region if the main region is down



Service disaster recovery scenarios

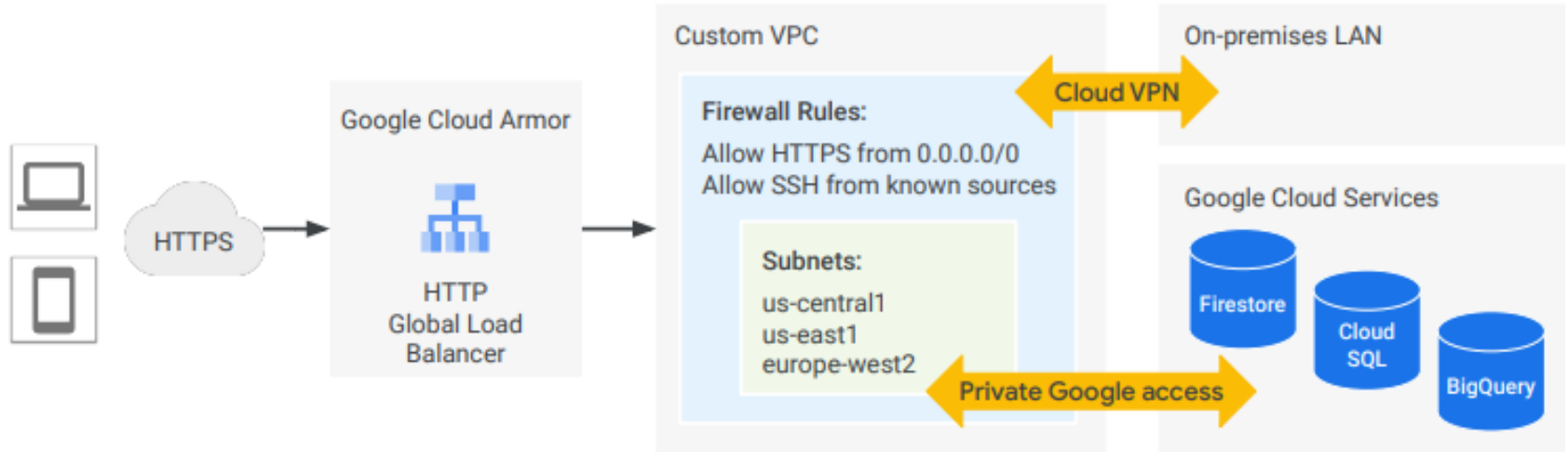
high-level list of possible scenarios. Each services uses different database services and has different objectives and priorities, all of which affects how to design the disaster recovery plans

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

Resource disaster recovery plans

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

Secure Google Cloud services Model



Note:

- First, I configured Google Cloud Armor on a global HTTP Load Balancer to block any denied IP addresses.
- My custom VPC network has subnets in us-central1 for the American customers, and a backup subnet in us-east1 and a subnet in europe-west2 for the European customers.
- My firewall rules only allow SSH from known sources, and although I allow HTTPS from anywhere, I can always deny IP addresses with Google Cloud Armor at the edge of Google Cloud's network.
- I also configured Cloud VPN tunnels to securely communicate with an on-premises network for the reporting service.

Note:

Now, while the load balancer needs a public IP address, I secured the backend services by creating them without external IP addresses. In order for those instances to communicate with the Google Cloud database services, I enable Private Google Access. This enables the inventory, orders, and analytics services' traffic to remain private, while reducing the networking costs

Cost estimating and planning

Service name	Google Cloud Resource	Cost
Orders	Cloud SQL	\$1,264.44
Inventory	Firestore	\$215.41
Inventory	Cloud Storage	\$1,801.00
Analytics	BigQuery	\$214.72

Note:

Here's a rough estimate for the database applications of the online travel portal, Treggs.

I adjusted the orders database to include a failover replica for high availability and came up with some high-level estimates for the other services.

The inventory service uses Cloud Storage to store JSON data stored in text files.

Because this is the most expensive service, I might want to reconsider the storage class or configure object lifecycle management.

Note:

Cloud SQL based on 1 instance of PostgreSQL in Iowa, 30 GB SSD storage Backup 1,000 GB, 12 cores, 16 GB RAM

Firestore based on US multi regional, 200,000 reads per day 10000 writes 0 deletes and 1,000 GB stored

Cloud Storage Iowa - us central 50 TB storage 200,000 class A operations and 200,000 class B operations.

BigQuery pricing based on location US multi-regional, 10 TB storage, 100 GB per month streaming, 2 TB queries



“Customer centric and reliable infrastructure design”.

THANK YOU