Bilkent University

Department of Computer Engineering

Snippy.me: URL Shortener

CS443 Cloud Computing & Mobile Applications

Design Report

Team Members: Rahmiye Büşra Büyükgebiz, Ozan Aydın, Doğaç Eldenk

Design Report
March 31, 2021

Table of Contents

Project Definition **2**

Site Reliability Engineering **3**
- SLA’s, SLI’s and SLO’s 3
- Assumptions 4

Implementation Decisions **4**
- Architecture 4
- Tech Stack Decisions 4
- Trade-off Analysis 7
  - Utilization vs. Availability 7
  - Complexity vs. Scalability 7
  - Memory vs. Speed 7
- Security 7
  - Google Cloud Armor 7
  - reCaptcha 8
- Mobile Application Design 9

Testing Strategies **10**
- Unit Testing 10
- Cross-Functional Testing 10

Forecast/Estimations **10**
- Cost Estimations 10
- Capacity Estimations 11

Metrics **11**
- SLA Monitoring 11
- Compute Infrastructure 11

Resources **11**

1. Project Definition

Snippy.me is a cloud based URL shortener cross platform mobile application. The main purpose of our system is to provide our users a platform where they can shorten their links and share them with others. The urls can be generated automatically or users can enter their unique short links. Our system exposes its API to its B2B customers using the REST interface.

2. Site Reliability Engineering

2.1. SLA's, SLI's and SLO's

The table below indicates our SLA's, SLI's and SLO's. For simplicity, in the first column we provide the SLA's without the objectives and penalties. We then provide the metrics for this SLA, and the measurements, which is our SLO's for this SLA. We then add a penalty column to indicate the consequences for not meeting our SLA standards.

Penalty Type	**Action**	**Applies To**
Type-S	Full refund during the inquiry with coverage for the losses of the company if they have insurance. B2C and Free users can be throttled to match the KPI.	All B2B customers
Type-A	Full refund during the inquiry and possible termination.	All B2B customers
Type-B	Partial refund during the inquiry and possible termination.	All B2B customers and paid B2C customers
Type-Z	Technical support will get in touch with the user.	B2B customer, premium and freemium B2C customers.

Functionality (SLA)	**Performance Metric (SLI)**	**Measurement (SLO)**	**Penalty**
Active customers can shorten their links without encountering any errors 99.99% of the time.	Downtime of the server.	Downtime per year must be <= 52 minutes 36 seconds.	Type-S
The shortened links will redirect to the target URL with latency smaller than 500 ms in Europe and 1 s for the rest of the world.	Latency of the successful HTTP(s) responses.	The latency of the responses must be < 500ms for Europe and < 1s for rest of the world.	Type-B

System admins can access the health and performance metrics of the instances without any error 99.9% of the time.	Uptime of the server.	Monitoring service downtime must be <= 8 hrs 45 mins.	Type-B
The REST API will provide 20 requests per second to the B2B customers	Requests bandwidth per second.	REST API must handle 20 requests per second.	Type-Z
The shortened links should be not easy to predict for security.	TestU01 Randomness Test	$p < 1e-10$	Type-S
The mobile application will be available to the users.	Availability of the app.	> Android v4.1, > iOS 8.0	Type-A

2.2. Assumptions

- Our system will handle around 1,000,000 redirect requests per day.
- The maximum incoming requests per second will not exceed 1,000.
- We will have around 100,000 registered users who will generate custom links per day.
- We will have around 20,000 active users per month.

3. Implementation Decisions

3.1. Architecture

Our application will consist of three modules: Authentication service, Analytics service and the App service. The authentication and analytics service's roles are straight forward. App service acts as a bridge between those services and the application. Also the shortened links will be resolved in the App service. We have chosen to use the app service as our link resolver without any micro services between for decreasing the latency.

We have splitted our database into two fragments: Redis for link resolution and Firestore for users and analytic data. The link resolutions will be done via Redis with minimum latency.

Our services will run under a Kubernetes cluster. Each service will be deployed in a docker container per pod for scalability. We will be using horizontal pod scalers for scaling horizontally. Each service will have a replica set. We have chosen two pods per auth and analytics services and three for app service. We have agreed on a minimum of two pods to match our availability constraints. So if a pod fails, the other pod will support the incoming connection until the other pods are re-created. The reason we have chosen three pods for the app is that the availability of the links are much more important than our other services. The services will be designed stateless as REST and cloud-native approach proposes. Because they are stateless, a load balancer can scale and distribute the load between them very easily.

3.2. Tech Stack Decisions

- **Spring Boot**

We will use Spring Boot in order to create microservices which require minimum setup configurations. Spring Boot will shorten our development time with its auto configuration feature. In addition, it has a great documentation on the web and the team is somewhat experienced with using it. It also makes it easier to test Java applications by providing a default setup for unit and integration tests. Spring also enables connection with Redis and can be containerized very easily.

- **Gradle [2]**

Gradle	**Maven**
Shorter build time with incremental and parallel running subtasks	Build time is 7 to 85 times lesser than Gradle
Team is somewhat experienced.	Team is somewhat experienced.
A compiler daemon that makes compiling a lot faster	Good compile time, but fails when compared to Gradle
XML code is much shorter and more understandable	XML code is long and hard to understand

- **Flutter**

Flutter	**React Native**	**Native**	**Web**
Write once run anywhere	Write once run anywhere	Need to develop two different applications.	Write once run anywhere
Very short development time	Reasonably short development time	Very long development time	Very short development time
Seamless mobile experience with native performance	Seamless mobile experience with native performance	Seamless mobile experience with native performance	Limited functionality and poor user experience
Team is somewhat experienced.	Team is somewhat experienced.	Low team experience, especially in iOS	Team is highly experienced
Most of the beautiful native looking UI components come OOB	Requires plugins and extensions or too much effort to create beautiful UI components	Native components	Requires frameworks to create responsive and good looking websites.

We will be using Flutter for our mobile cross-platform app development. The other alternatives were creating a *Native App, Web application*, *React Native*, *Ionic* or *Xamarin*. We have eliminated Ionic and Xamarin because we didn't have much experience with them and their popularity was less than others [6].

From the table above we can see that Flutter fits our needs the best. The closest competitor was React Native but we have chosen Flutter because of the rapid development time and out-of-the-box beautiful UI design.

- **GCP**

In the past, we had experience with multiple cloud providers such as Google Cloud Platform or Amazon Web Services [4].

GCP	**AWS**
Easier to use, more intuitive UI	Difficult to learn, lots of services, messy UI
Team is somewhat experienced	Team is somewhat experienced
Better documentation and customer support	Good documentation, because the business is big, customer support is not as responsive to the small customers
Low Prices	Very Low Prices

Both platforms were very close in when we compared them. We have chosen to work with GCP even though it is a little more costly because of its intuitive UI and documentation. We think time is a big factor in our decisions so GCP is a better fit in our situation. But AWS might be more efficient in the future when the business grows. Also we will be using GKE for our kubernetes engine backend for container orchestration, load balancing and scaling.

- **GCP Endpoints for OpenAPI**

We will be using GCP Endpoints used internally by Google for exposing our REST API to the customers and the users. GCP Endpoints enables us to secure, monitor and set quotas for our endpoints. We have used OpenAPI with Swagger in the past for creating documentation for our REST API and it creates a well organized documentation for our customers and the internal teams with a very little effort. Therefore we are planning to take our API documentation to the next level by integrating the GCP Endpoints to our application.

- **GCP Cloud Firestore and Redis** [5]

We will be sharding our database into segments for increasing our performance. The shortened URLs need to be accessed as fast as possible. Therefore a low latency key-value store such as Redis is a very good fit in our case
.

Key-Value Store	**Document Store**	**Relational Database**
Extremely fast write and read speeds.	Fast write, good query times.	Fast write, good query times.
Simple complex data	Flexible complex data	Predefined Schema
Highly Scalable	Highly Scalable	Only vertically scalable.

For our analytics and user authentication data, we will be using the GCP Firebase for two main purposes: seamless Flutter integration and flexibility. Using a document store will be a naive approach to our growing application for storing multiple data types in our database.

3.3. Trade-off Analysis

3.3.1. Utilization vs. Availability

Our application is highly available with the help of the GKE. We will horizontally scale when there is high traffic. But this means that in terms of utilization, our system will not always be optimized as there usually be duplicate, unused pods waiting for traffic to keep our availability high.

3.3.2. Complexity vs. Scalability

We aim to achieve scalability by making use of compartmentalization via Docker and Kubernetes. Building each microservice requires extra effort and maintaining multiple microservices increases our platform and development complexity.

3.3.3. Memory vs. Speed

Since Redis is an in-memory database, we have another tradeoff such that we can achieve faster read and write but Redis uses a portion of our memory [3], also our data set size cannot be larger than our memory. Therefore we need to vertically scale our instances.

3.4. Security

We will use two tools in order to maintain security of our application and to prevent database overload.

3.4.1. Google Cloud Armor

Google Cloud armor is a GCP service that works with the load balancer in order to prevent DDoS and Web Application Firewall attacks. Cloud Armor will scale up based on our traffic, we can also ban/permit specific IP addresses from sending requests, and write custom protection codes if need be.

3.4.2. reCaptcha

Preventing DDoS attack completely is not possible, but in order to secure our databases and keep our services alive, we wanted to add an extra layer of protection in reCaptcha. Users will have to tick a “I’m not a robot” checkbox after they click the shorten button. This feature will be available only if there are more than 1 request per second from the same client.

3.5. Mobile Application Design

In the figure given in the appendix, you can see the mockup of our mobile application design. Our mobile application will have a basic design that will have only 6 pages. When a user opens the application, they will see the home page which includes a text field that inputs the URL to be shortened and also two buttons that redirect the user to login or create an account. The designs of Sign in and Sign up pages are trivial. Another page that home page redirects is the page that shows the shortened link where the user can copy their link, use the button which directs the user to the homepage or use Sign in and Sign up buttons. If a user decides to create an account or login, they are redirected to the fifth page, where the signed-in user can view the report of their shortened link, go to the home page to shorten more links or change settings. Change settings will be a basic popup that looks like the Sign in page where users can enter new passwords or update their email, therefore we did not include it in the main structure of our mockups. The last page, reports page, includes some statistics about the signed-in users’ shortened links that we explained in other sections. This page is available only to the signed-in users.

4. Testing Strategies

4.1. Unit Testing

The first step in our testing strategy is unit testing. We will verify the functionality of our code by testing methods one by one with unit tests and ensuring that the behaviour is as expected. We will use JUnit framework for unit testing since it can be integrated with Gradle easily, it is a simple framework for writing automated tests and it provides efficiency in terms of reporting time.

4.2. Cross-Functional Testing

We will test all of our SLA’s such that we never fail the standards. In order to perform load testing, we will use Locust, among other contenders like JMeter and Gatling. We chose Locust as its biggest contender, JMeter was too heavy and dependent on GUI. Also, even though JMeter is older, Locust is currently very popular and has a very active user base.

- To test “99.99% uptime guarantee for our customers”, we have to do a load test in order to understand the maximum requests per second that our server can handle and we will compare it with our assumptions. We do this by sending batches of requests to our servers and calculate the success rate of the requests.

- In order to test “latency constraints”, we have to seperate the load testing into different modules where in each module, we set up concurrency equal to our estimated active users and send different rates of requests and calculate the median latency of each request sent.

5. Forecast/Estimations

5.1. Cost Estimations

We are assuming a 1 year commitment on the GKE platform with a minimum of 7 pods for the bare metal minimum. If we choose to use the shared CPU model, we can get as low as $27 per month [1]. But if our traffic increases, we need to scale our pods vertically first later horizontally. Using an e1 machine would cost us $70 and n1 would cost us 170$. Realistically we want to start with a shared cpus and scale up as we build our customers.

5.2. Capacity Estimations

If we assume an average size of 100byte per link, we can store up to 20,000,000 links in our database just by using 2GB of RAM for our redis server. For our users and customers, We would be storing in average 1Mb of data per customer and 1Mb of data per day analytic per link. Therefore we will require more than 100GB of space when we match our expected amount of usage.

6. Metrics

In the section above, we will consider different types of metrics and how and why we will monitor them. We will use Cloud Monitoring, a service of GCP, in order to monitor our program in a robust and efficient way.

6.1. SLA Monitoring

We will monitor all the SLA’s we provided in the above section. Cloud Monitoring allows us to monitor failure ratio by keeping track of every request. We will integrate monitoring tools to both our cloud infrastructure, backend and frontend.

6.2. Compute Infrastructure

Cloud Monitoring is able to CPU usage, system memory, database utilization and so on. We will monitor database utilization, system memory utilization and CPU usage in order to be able to understand if our system is under a heavy load, even though this will increase the computational overhead and client side work, it is important to keep consistent data on when, how and why there is a congestion.

Resources

[1] “Google Cloud Platform Kubernetes Engine Pricing” https://cloud.google.com/kubernetes-engine/pricing.

[2] “Gradle vs Maven Comparison” https://gradle.org/maven-vs-gradle.

[3] “FAQ” https://redis.io/topics/faq.

[4] “Google Cloud vs AWS: Difference Between AWS and GCP” https://www.guru99.com/google-cloud-vs-aws.html.

[5] “Google Cloud Firestore and Redis” https://www.g2.com/compare/google-cloud-firestore-vs-redis.

[6] “Choosing the Right Mobile Application Framework..” https://betterprogramming.pub/choosing-the-right-mobile-app-development-framework-for-your-next-project-8159c8bbd5bc.