Fiskil Challenge

Challenge1: Solution

As an engineer I could think of two approaches to this problem.

Approach 1: If single tenancy architecture is not a requirement. We can break down approach 1 into five simple steps.

a) **Multi-tenancy architecture:** Since we are dealing with multiple clients, its important to design a multi-tenant architecture. This will allow us to isolate resources, data, and configurations for each client while sharing the same application instance.

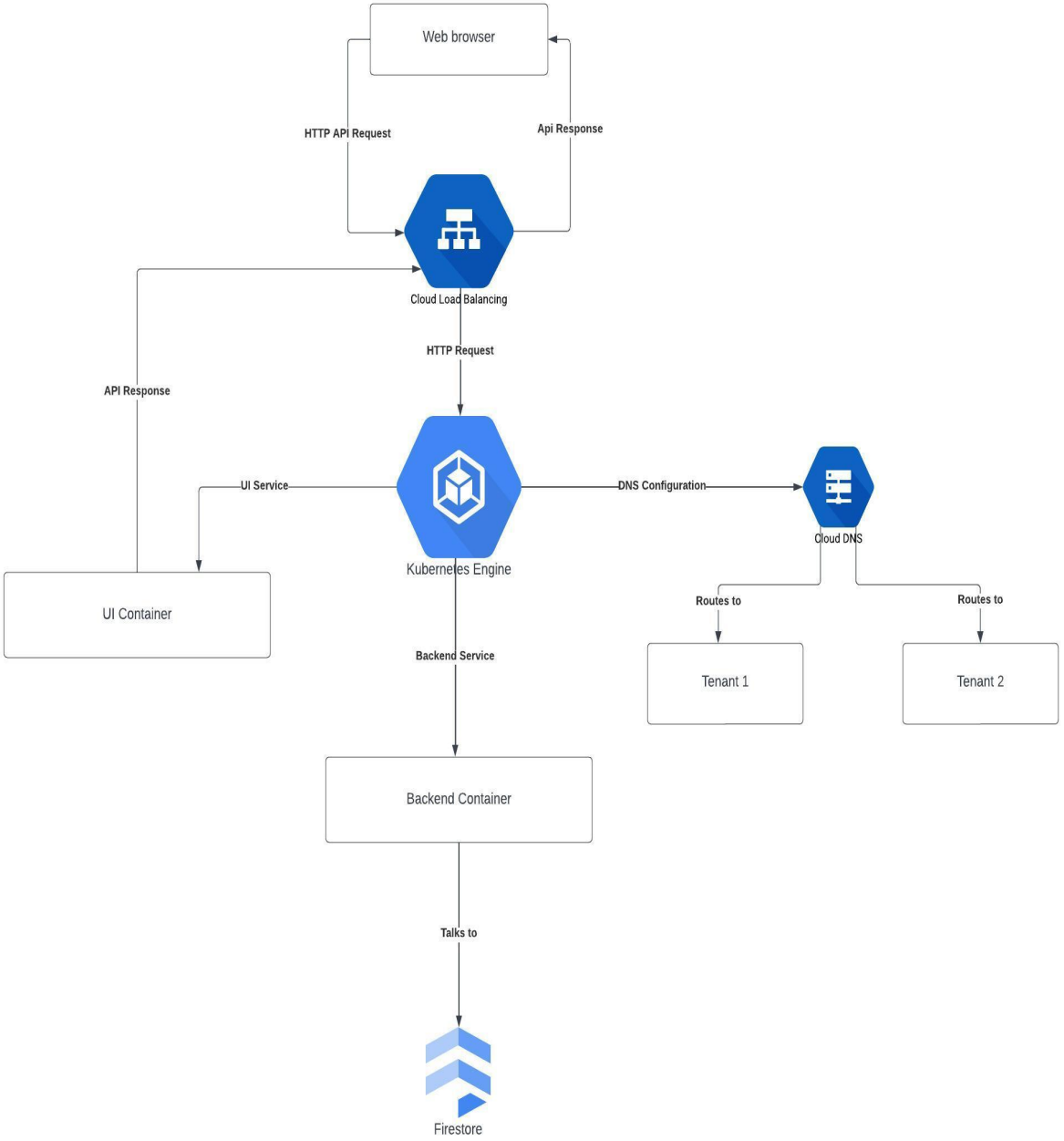
b) **Dockerization**: Dockerize the UI, backend service, and NoSQL DB. This will allow us to package our application with all its dependencies into a standardized unit for software development.

c) **Kubernetes:** Use Kubernetes for orchestration of our Docker containers. This will help us manage, scale, and maintain our containers efficiently.

d) **Branding:** For branding, we can use a configuration file that contains all the branding information for each client. This file can be loaded during the client's session and the UI can be rendered accordingly.

e) **DNS Configuration:** For DNS configuration, we can use a wildcard DNS entry that points to our application. The application can then use the subdomain to determine which client's configuration and branding to load.

The High-level component diagram for the proposed solution is shown below:



For Continues integration and continuous delivery I propose the following:

a) **Version control**: Use a version control system like Git. This will allow us to track changes, work on different features in isolation, and integrate them later.

b) **Automated Testing and Test driven development** : Write unit tests, integration tests, and end-to-end tests. These tests should be run every time code is pushed to the repository

c) **Build:** Use a build tool to compile the code and create a Docker image

d) **Continuous Integration:** Use a CI server like Jenkins or CircleCI. The CI server will pull the code, run the tests, and build the Docker image automatically whenever code is pushed.

e) **Continuous Deployment:** Use a CD tool to deploy the Docker image to the Kubernetes cluster. This can be done automatically for every change that passes the tests, or it can be done manually.

f) **Monitoring and Logging:** Implement monitoring and logging to track the application's performance and errors. This will help us identify and fix issues quickly.

As an engineer my advice would be as follows:

a) Architecture: To Prefer Multi Tenancy Architecture over Single Tenancy Architecture for the following reasons:

1) Cost: Single-tenant systems are generally more expensive to maintain because each tenant requires their own separate instance of the software and infrastructure. This can lead to higher costs for hardware, software licenses, and maintenance as the tenants/customer increase over time.

2) Scalability: Single-tenant systems can be less scalable than multi-tenant systems. In a multi-tenant system, adding a new tenant often just means adding a new set of data to the database, whereas in a single-tenant system, it can require provisioning and setting up a whole new instance of the application.

3) Backup and Disaster management: In a single-tenant architecture, each tenant's data is stored separately, which means that backup and disaster recovery processes must be performed individually for each tenant. This can be a time-consuming and resource-intensive process, especially for organizations with a large number of tenants. In contrast, in a multi-tenant architecture, data from all tenants is typically stored in a unified manner, which can simplify and streamline backup and disaster recovery operations.

Other advices would be:

b) **Infrastructure as Code**: Use tools like Terraform to manage your infrastructure. This will allow to version control the infrastructure and keep it consistent across different environments.

c) **Configuration Management:** Use a configuration management tool to automate the setup and configuration of servers.

d) **Scalability:** Design applications to be stateless. This will allow to scale application horizontally by adding more instances.

e) **Security:** Implement security best practices like using HTTPS,and implementing rate limiters, etc.

f) **Backup and Disaster Recovery:** Regularly backup the data and have a disaster recovery plan.

Approach 2:

If a single-tenant architecture is a requirement, the approach would change slightly as follows:

a) **Single-tenant Architecture:** In this case, each client would have their own instance of the application, including the UI, backend service, and NoSQL DB. This can provide better isolation and security, but it can also increase costs and complexity.

b) **Dockerization:** The same as before, we would Dockerize the UI, backend service, and NoSQL DB.

c) **Kubernetes Namespaces:** We can use Kubernetes namespaces to isolate each client's resources. Each namespace would have its own set of services, deployments, and pods.

d) **Branding:** The same as before, we can use a configuration file that contains all the branding information for each client.

e) **DNS Configuration:** Instead of using a wildcard DNS entry, we would need to create a separate DNS entry for each client. This could be automated using a DNS API.

f) **Scaling:** Since each client has their own instance, we can scale each client's resources independently based on their needs.

g) **Data Isolation:** Since each client has their own database, data isolation is ensured

The reststs of CI/CD process would remain the same as multi tenant architecture. The main challenge with a single-tenant architecture is managing the increased number of resources. We would need to ensure that we have sufficient monitoring and logging in place to manage all the instances. We would also need to ensure that we

have a robust backup and disaster recovery plan in place, as each client's data would be stored separately