Documentation

Opensoft Competition (2022)

Team 222

IIT Kharagpur

Table of Contents

Problem Statement

To use modern cloud technologies and architecture like containers, microservices, and hybrid cloud in order to facilitate the technical operations of a leading nationwide fast-food chain (Anytime fast-food).

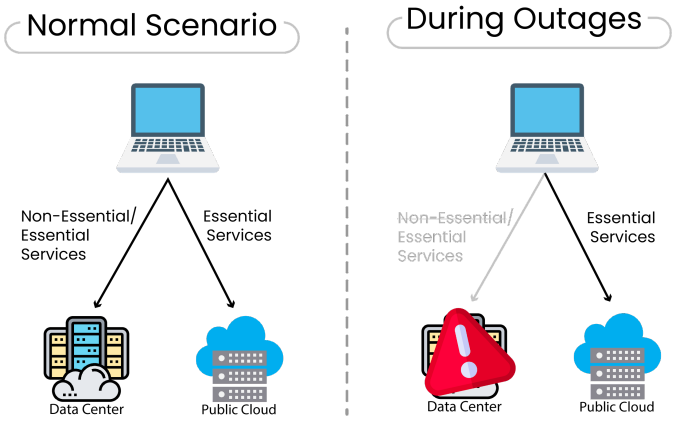
Context: AnytimeFood is a nationwide fast-food chain with over 7,000 restaurants that stores all consumer-related information such as food preferences, loyalty status, etc in a centralized IT core in their data center.

It has experienced disruptions at its centralized IT core resulting in down-times long delays for customers and leaving customers very unhappy. AnytimeFood made a huge loss during this outage and it anticipates the loss of many customers to their competitors as well.

So, the problem statement is to **design and build a modernized hybrid cloud architecture** for AnytimeFood using **new container and cloud-native technologies** to ensure that their critical operations such as food ordering and delivery can still function normally to provide an **acceptable consumer experience even during downtime** at its centralized IT core.

Requirements and Deliverables

- Essential micro-services to be deployed in both data-center and cloud.
- Non-essential micro-services to be deployed only in IT core (save the cost)
- Use container technologies to modernize the applications
- Use Microservices architecture
- Use Kubernetes to scale the architecture effectively
- Use a container management platform to ease the deployment
- Build a dashboard to view uptime/downtime of data-center and restaurant chains.
- Notification alerts for any outage.
- During outage - only essential requests should be entertained. (For other requests we shall get a down-time page)



Tech-Stack Used

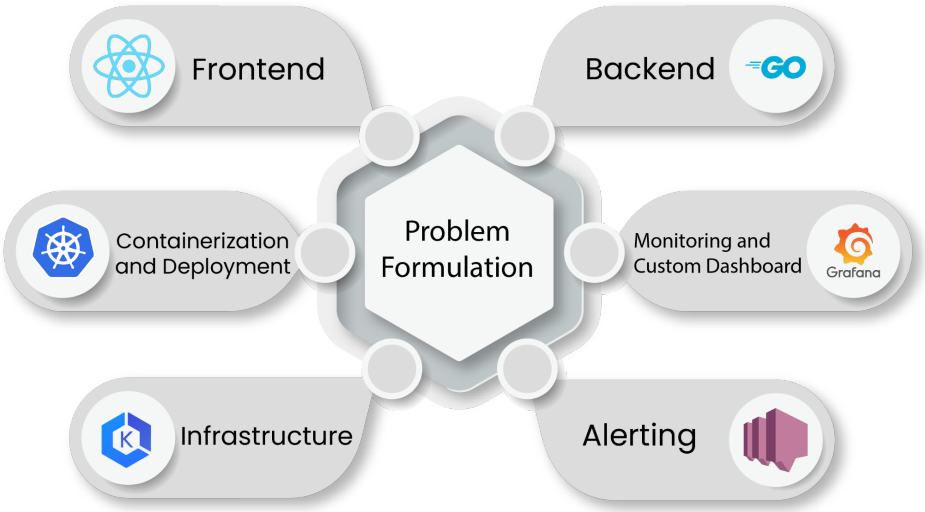
- **Frontend:**
  - **React:** JavaScript library to create component-based applications.
  - **Redux:** JavaScript library to efficiently manage and centralize application state
  - **Material UI**: React component library
- **Backend:**
  - **GoLang**
    - GORM: An ORM(object-relational mapping) library for Go.
    - Gorilla: Powerful HTTP router and URL matcher.
  - **PostgreSQL**: powerful, open-source object-relational database system.
- **Amazon Web Services:**
  - **EC2**: Elastic Compute Cloud service by Amazon, provides compute platform
  - **ECR**: Elastic Container Registry service offering high-performance hosting.
  - **EKS**: Elastic Kubernetes Service facilitates the running of Kubernetes on AWS
- **NGINX**
  - **Global Load balancer:** Distributes traffic across server resources.
  - **Ingress Controller**: Ingress controller for Kubernetes using NGINX as a reverse proxy and load balancer.
- **Prometheus:** An open-source monitoring system with a dimensional data model,
- **Grafana:** It is the open-source analytics & monitoring solution for databases.

Problem Formulation

The problem required work on the following sub-categories:

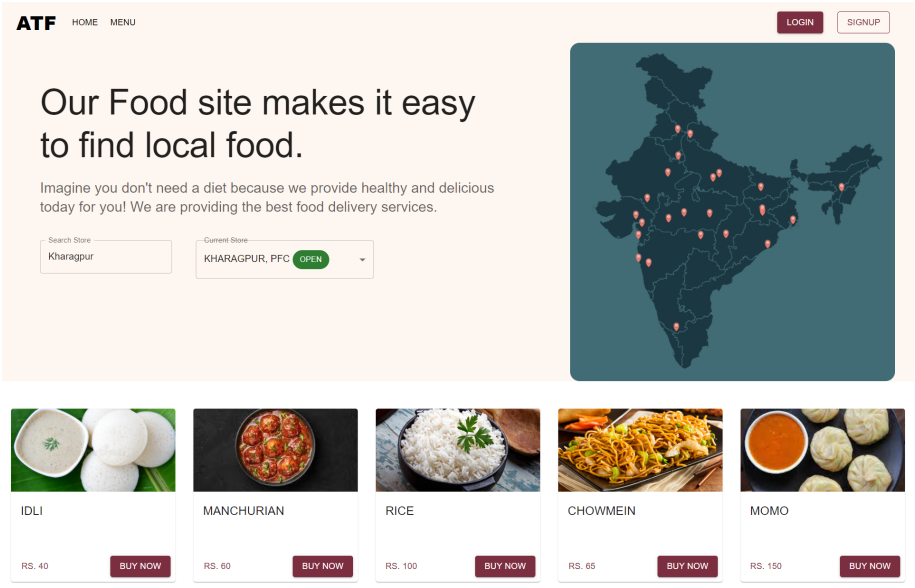
- **Frontend** - It was created for accessing the services majorly such as searching food items, order placing, vendor dashboard of AnyTimeFood through Graphical Interface.
- **Backend** - Built the REST APIs necessary for the frontend such as customer-related activities, vendor-related activities, and infrastructure-related activities.
- **Containerization and Deployment** - Created the docker images of all the applications and pushed them to the AWS ECR registry for ease of configuring them in the cloud. The application got deployed on AWS EC2 and EKS by pulling out the docker images.
- **Infrastructure -**
    - Created an EC2 instance that acts as a data center and an EKS Kubernetes cluster that acts as a public cloud on separate VPCs.
    - These two got connected using a load balancer for distribution and redirection of workloads (essential and non-essential services)
    - During the time of outages, all the essential services will be redirected to the public cloud as the Kubernetes cluster is capable of running multiple containerized applications with high availability (when something fails, it will replace it and restart over) and the non-essential services, the data center will be made unavailable to the public.
- **Monitoring and Custom Dashboard -**
    - Deployed Prometheus using helm charts for fetching out the metrics of the health checkup of the hybrid cloud.
    - Deployed Grafana using helm-charts and created custom dashboards of all the pods, restaurants status, etc by adding Prometheus, Postgres as data sources into Grafana for visualizing the metrics from time to time.
    - We installed the Grafana infinity plugin and visualized the data center health check endpoint from time to time.
- **Alerting -**
    - Used the AWS SNS service for alerting all the users through email during the downtime of the non-essential services
    - Created a lambda function using AWS Lambda service to trigger the SNS service and generated an URL that can be used as a webhook in Grafana.

- Custom alerts have been made based on the threshold level of the required metrics by placing the generated URL through a webhook contact point in Grafana.

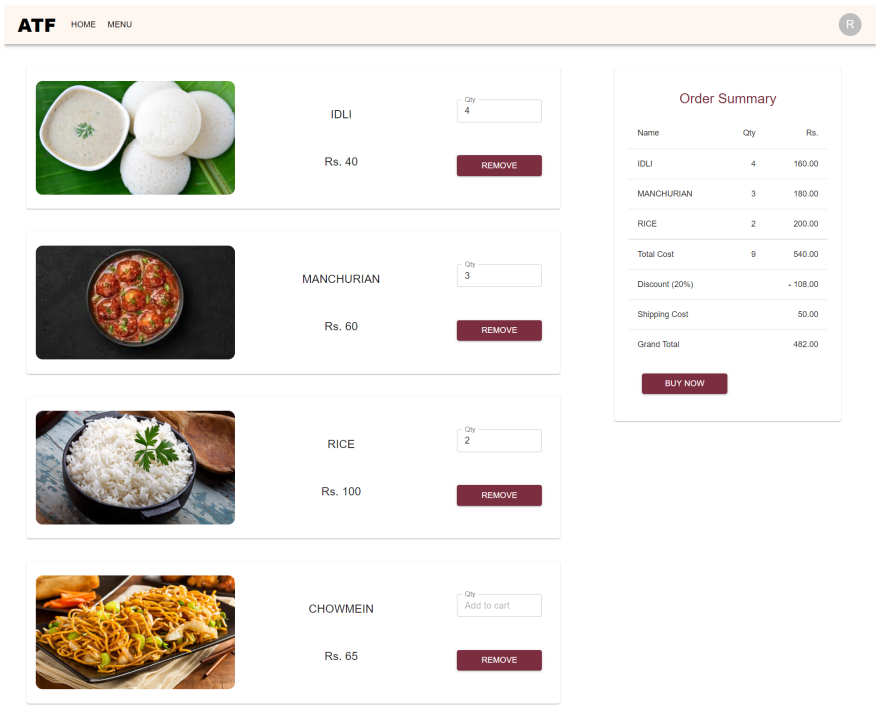


Frontend

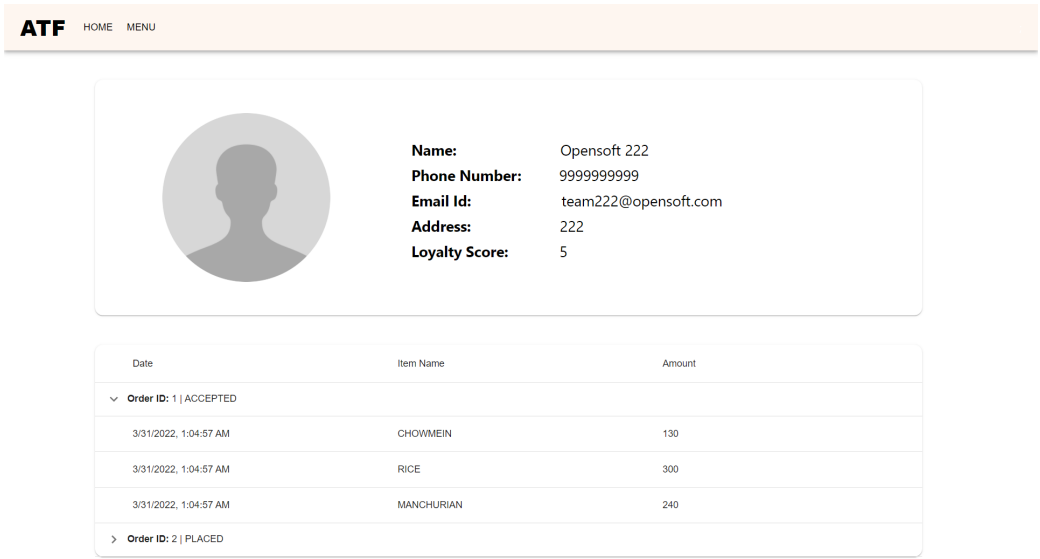
- **Home Page**: It is the landing page of the website and consists of the following features:



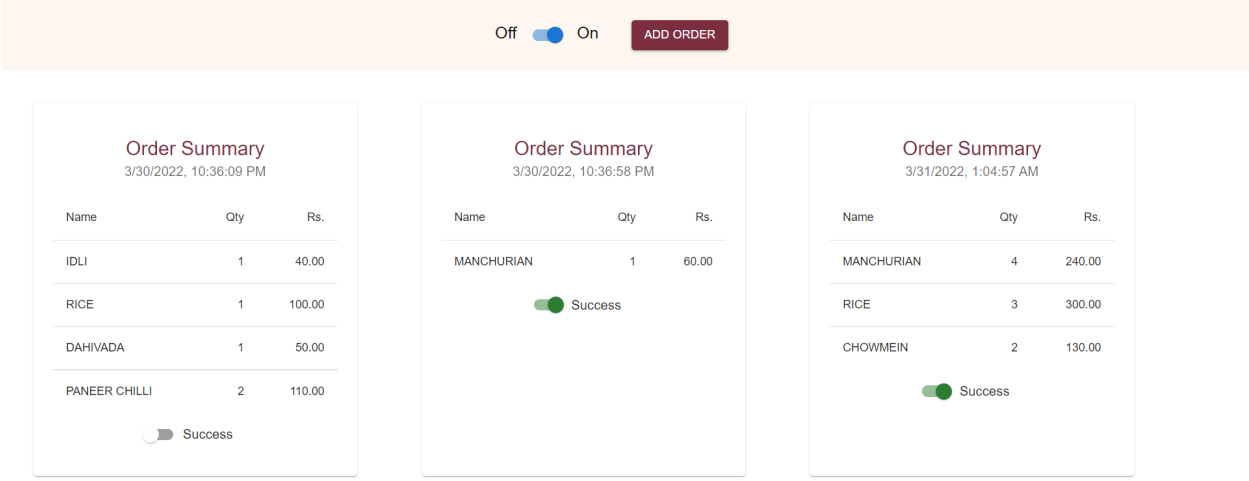
  - **Navbar:** The navbar up top routes to different pages depending upon the links clicked such as the login/signup pages, Menu Page, and back to Home page.
  - **Search stores:** We can search stores based on city, The API on it searches by regex matching. Every time the Search bar data changes the API is called and based on a regex match all the stores are sent to the frontend. The search drop-down also shows the status of the store (OPEN / CLOSE)
  - **Best Sellers:** Through the endpoint of the recommendations API, we fetch the five topmost items and display it in the best seller's menu. It is a non-Essential service. This will not be shown when non-Essential services are down.
- **Signup/Login:** The data is taken from the form filled and is taken to the database to display the profile page based on the login status of the user.
- **Menu:** Cards of items with quantity and buy options. Also, we give you a brief order summary along with the total billing.



- **Checkout:** Upon checkout also keep storing the data about total items, card details, and the total price for easy access upon return.
- **Profile/Dashboard:** Shows data about a certain user such as his name, email address, and other kinds of information with his order history and order status in a collapsable table which is grouped by the Order ID.



- **Vendor Dashboard:** Vendor Dashboard is a detailed dashboard designed for the store end. In this the functionalities are: Vendor can Open / Close the store using the ON / OFF button. If the vendor closes his store orders will not be accepted for his store. The details of the order he has received which he can change the status to success after completing it.



- **Vendor Checkout:** Vendor Checkout is the functionality given to the vendor where he can order for his customer who physically visits the store with the help of the customer id.

Backend

The backend is developed completely in Go language assuring a simple language syntax yet powerful implementation, supported by a massive community. PostgreSQL is used for database deployment in AWS RDS. Various go libraries have been used such as :

1. **Gorm** is used for CRUD operations, for the initial migration, and the creation of the database schema.
2. **net/HTTP** is used to start the server.
3. **Gorilla** is used to make the handler functions configure the PermissiveCors.

All APIs have been divided into 2 parts **essentials** and **non-essentials.**

- **Essentials**

  This contains all the APIs that are necessary to execute basic operations to keep the website running. The APIs are :

  - **For customers:**
    - AddCustomer
    - Login
    - GetOrders
    - PlaceOrders
    - GetItemById
    - GetItemByName
  - **For vendors:**
    - AddVendor
    - LoginVendor
    - VendorGetOrders
    - VendorPlaceOrders
    - GetItemById
    - GetItemByName
    - VendorGetItems
    - VendorStoreStatus
    - OrderTweakStatus
  - **For Infrastructure:**
    - HealthCheck
    - Infra

- **Non-essentials**

  This contains all the APIs that are somewhat extra and act as complimentary services. The APIs are :

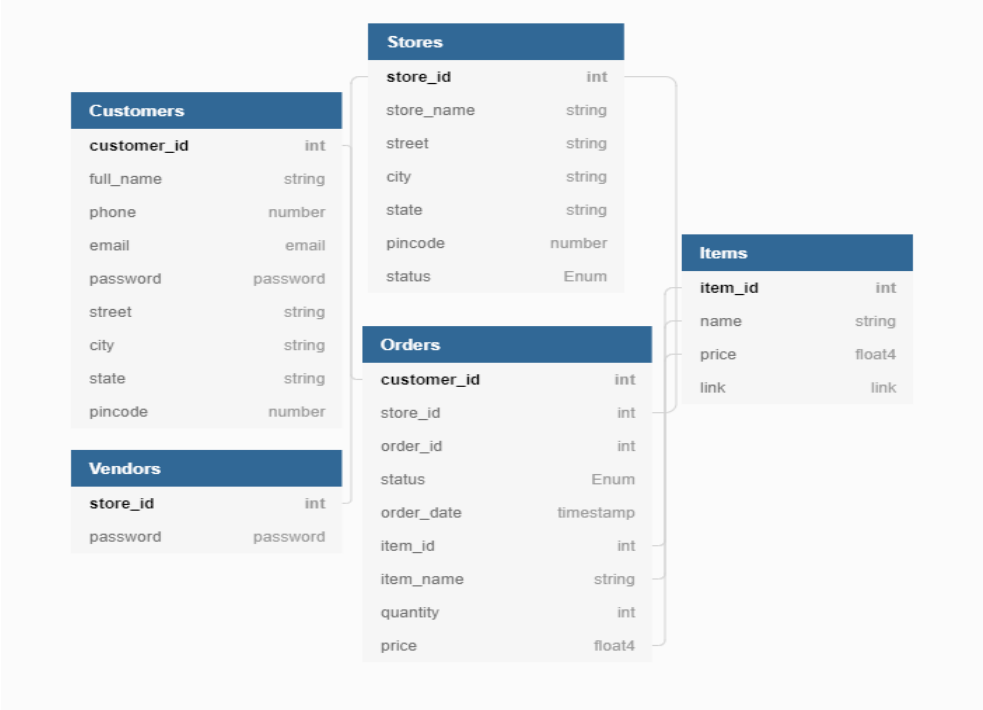
  - Loyalty
  - Recommend

  All the non-essential APIs stop working whenever the data center is down.

- **DATABASE**

  The database contains various tables such as :

  - Customers
  - Vendors
  - Stores
  - Items
  - Orders



Infrastructure

- **Requirements:**
  - Divide the traffic between public and private cloud
  - Essential services to run even during downtime
  - Use microservices architecture for fault tolerance
  - Services need to be auto-scalable
- **Approach:**
  - The infrastructure diagram of our network is as follows:
  - **Datacenter:** We mimic the behavior of the data center by deploying it in an EC2 machine in another center (i.e. outside VPC of Public Cloud)
  - **Global Load Balancer:** We use an NGINX load balancer deployed on a t3.large EC2 instance. This load balancer divides the load between the data center (private cloud) and the host endpoint of the Kubernetes cluster.
    i. Essential Requests and frontend are distributed amongst both the data center and public cloud.
    ii. Non - Essential Requests are routed only to the Datacenter.
    iii. Why NGINX global load balancer: - It is very easy to configure and very versatile in its working. It has multiple functions and can be enriched with modules.
  - **Database:**
    i. The database used is PostgreSQL in the backend.
    ii. The problem statement requires our application to run on both public and private clouds parallelly. In such a scenario, the consistency of the database becomes an issue of concern.
    Also, there will be multiple instances of the same application running in different containers across different nodes. These instances perform read and write operations both.
    iii. So, for a highly available and consistent database solution, we used **AWS RDS.**
    AWS RDS is a fully managed database service provided by AWS which is highly scalable and highly available also.
  - **Container Management:**
    i. Applications running in containers can be deployed easily to multiple different operating systems and hardware platforms.

ii. Containers require fewer system resources than traditional or hardware virtual machine environments because they don't include operating system images.

iii. The entire architecture was containerized using Docker and AWS ECR for ease of maintenance and configuring in the cloud.

iv. Microservices are used for easy debugging and deployment.

- **Public Cloud Deployment**

i. We require a deployment solution in the public cloud that is highly available, auto-scalable, and fault-tolerant.

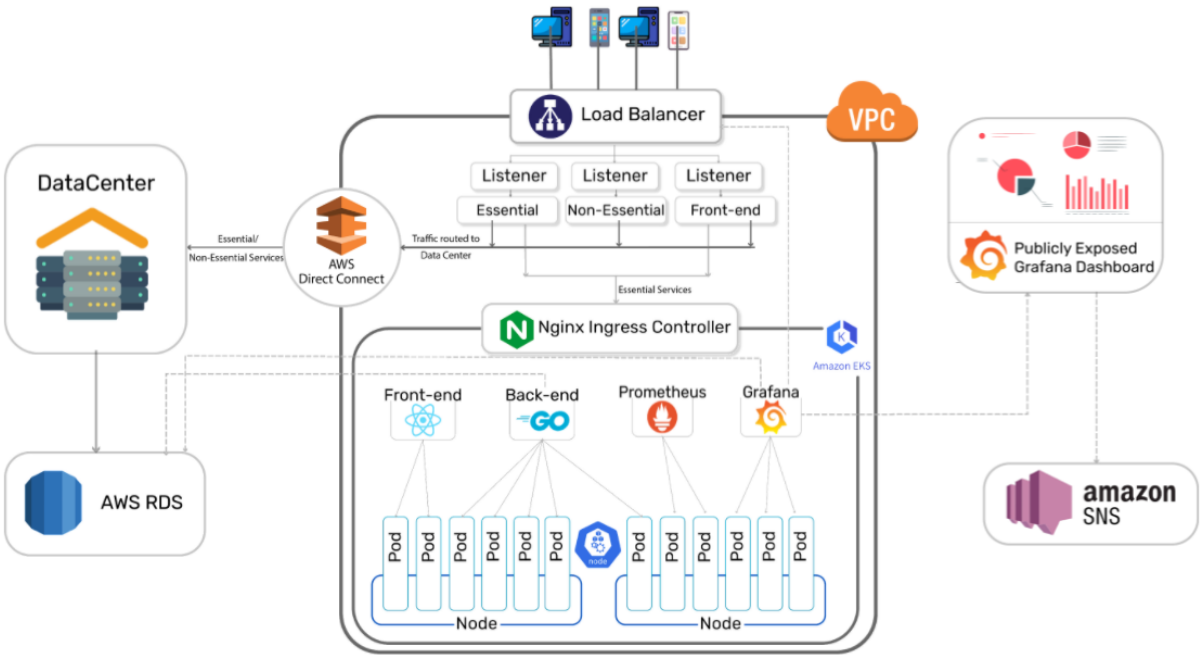
ii. So we used Kubernetes for the deployment of our Application which provides us with the above Functionalities.

iii. It is a Kubernetes cluster constructed and hosted on AWS through the EKS service which acts as a cloud platform for the traffic during normal periods.

iv. When there is huge incoming traffic or downtime, we only redirect the essential services to the public cloud and the data center, non-essential services will be made unavailable to the public.

v. The reason for redirecting the essential public cloud is that the Kubernetes cluster is capable of running multiple containerized applications with high availability(when something fails, it will replace it and restart over).

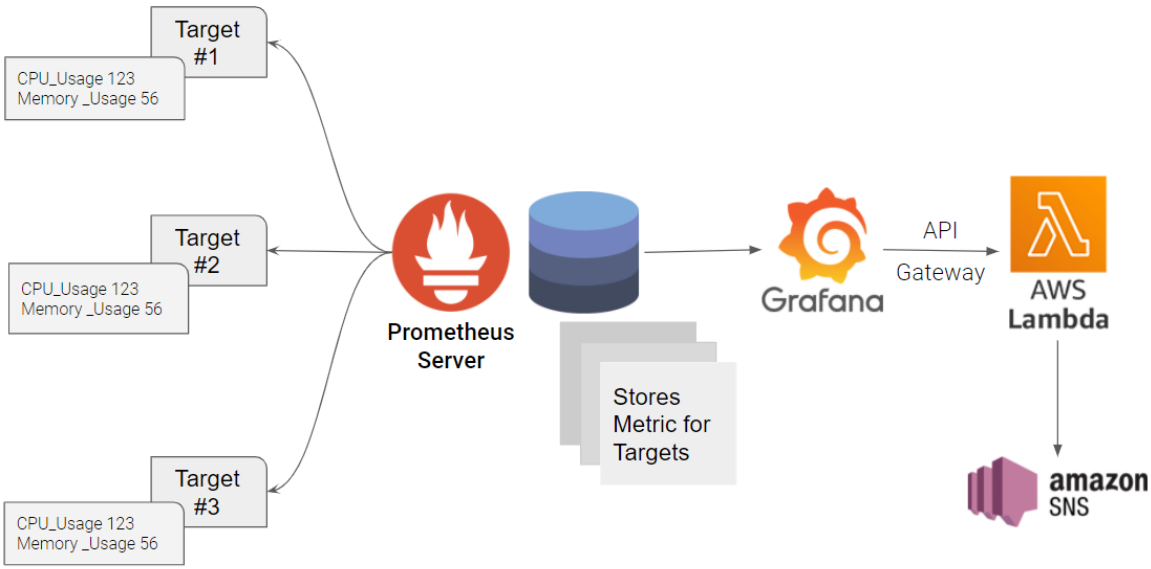
vi. The hybrid cloud architecture will get monitored and alerted for the outages of the services using Prometheus - for the collection of the metrics and Grafana - for visualizing the metrics and alerting the system



- **Actual Infrastructure**
    - Global Load Balancer: EC2 t3.large mimics the NGINX Global Load Balancer
    - Datacenter: EC2 t2.large mimics the Datacenter
    - The cluster details are as follows:
        i. **Nginx Ingress Controller**: Manages the frontend and backend services and acts as an endpoint for the Kubernetes Cluster
        ii. **Worker Nodes**:
            1. T3.medium machine used
            2. Minimum number of nodes: 5 & Max number: 8
        iii. Applications/Services Deployed:
            1. Frontend Application
                a. Min Replicas: 2, Max: 100
            2. Backend Application
                a. Min Replicas: 3, Max: 100
            3. Prometheus
            4. Grafana
            5. AWS Auto-Scaler: To auto-scale the worker nodes as per the requirement.
                a. Trigger**:** New pods are pending and existing worker nodes do not have enough resources to accommodate them.
            6. Metric Server: To get the CPU usage of pods used to autoscale pods
            7. Horizontal Pod Autoscaler (HPA): to auto-scale the pods of front and backend as per the needs.
                a. Trigger**:** 80% of the maximum CPU limit

Monitoring and Alerting

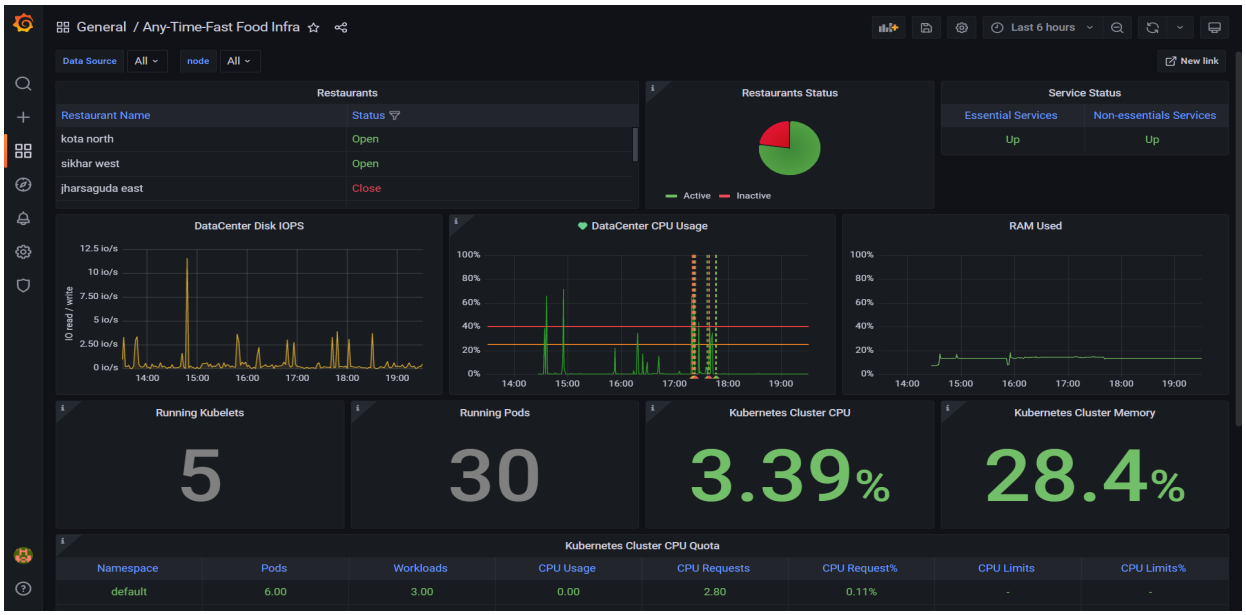
- **Technologies used:** Prometheus, Grafana, AWS SNS, AWS Lambda
- **Infrastructure:** We have used open-source software called Prometheus for service monitoring and Grafana for visualization. AWS Simple Notification Services(SNS) is used for sending SMS and E-mail alerts to customers. AWS Lambda is used to connect AWS SNS to Grafana.



- **Prometheus:** Prometheus records real-time metrics of every component of the infrastructure(including itself) in a time-series database. It also provides flexible querying on the stored data using its query language named PromQL. We have deployed the Prometheus stack in our Kubernetes cluster using Helm Charts. Prometheus stack consisted of Prometheus server, Grafana, and a few other essential services. We have exposed the deployed Grafana to the localhost. Grafana can now be accessed on the local machine.
- **Node Exporter:** We have sourced all the metrics scraped by the Prometheus server in the Kubernetes cluster to Grafana for its processing and visualization. Deployed Prometheus server monitors the health of all the nodes, pods, and services in the cluster including its health. At this point, two essential targets namely - Nginx Ingress Controller

and Private Cloud remain unmonitored. To monitor the health and collect metrics of the same, we have deployed two Prometheus node exporters(specific to each target) in an Amazon EC2 instance as containers.

- **Grafana:** We then configured our Prometheus server to collect the metrics exposed by node exporters and sourced this matrix to our locally hosted Grafana. This completes our task of real-time monitoring every component of our infrastructure. Installation using helm charts also provided tens of pre-built dashboards in Grafana for efficient monitoring.



- **Alerting:** To obtain alerting, we have integrated Grafana, AWS Lambda, and AWS SNS. A threshold of 40% is set on the CPU utilization of the Private Cloud. Thresholds on RAM usage and every other metric are also possible simultaneously but we decided to keep it only on the CPU for the simplicity of demonstration. If CPU utilization of the Private cloud rises above the threshold and stays there for longer than the set time(which in our case is set to 15 sec), Grafana triggers an alert and releases a post request to our AWS Lambda function. AWS Lambda when triggered is set to perform the following operations:

  1. Turns off our private cloud/data center.
  2. Publish a message in our AWS SNS topic. This topic is subscribed by all the customers of AnyTimeFood.

  [**Note**: Customers may choose to opt out of this notification service.]

Cost Analysis

Type	Cost/Hour	Quantity	Total Cost/Hour
Cluster	$0.10	1	0.10$
Nat Gateway	$0.06	1	0.06$
Load Balancer	$0.03	2	0.06$

EC2 Instance	Cost/Hour (instance cost + EBS Volume Cost)	Quantity	Total Cost/Hour
t3.medium(Node Group)	0.04$ + 0.0088$	5*	0.2440$ *
t3.large(nginx)	0.08$ + 0.0035$	1	0.0835$
t3.large(datacenter)	0.09$ + 0.0035$	1	0.0935$

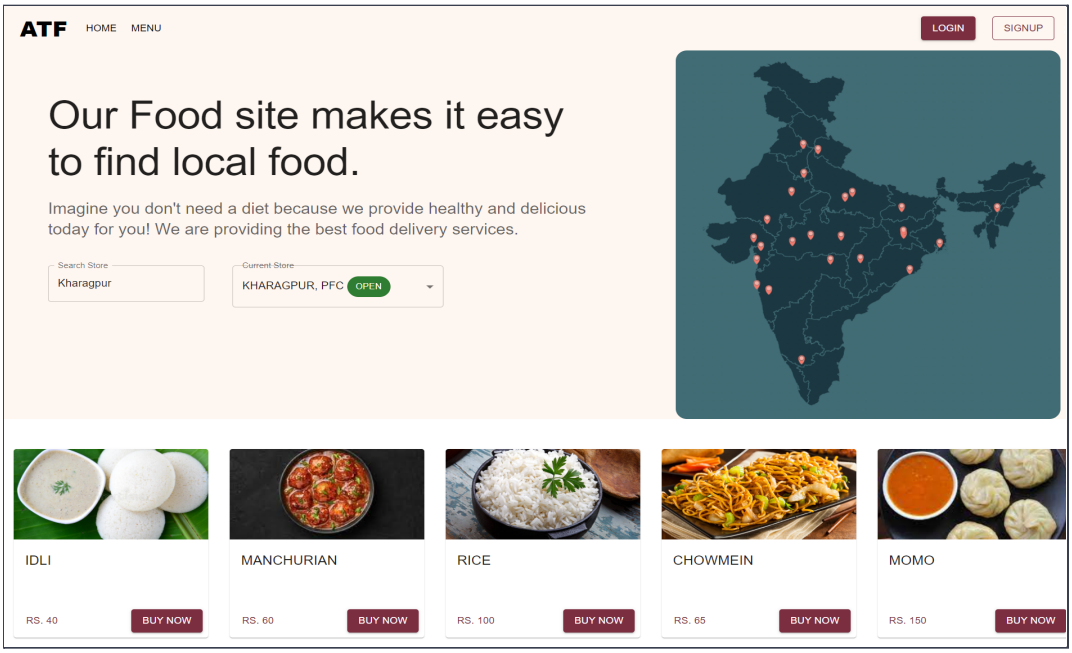
Total Cost = 0.641$/Hour , 67.86$/month*

*The cost depends on the number of t3.medium instances of nodegroup which may change as per the load.

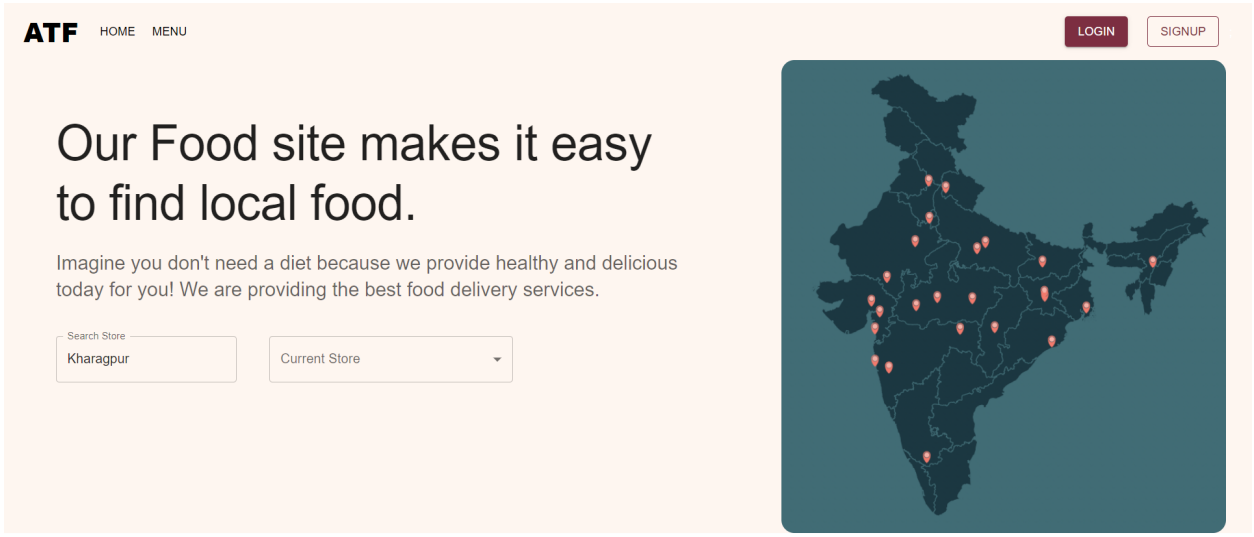
Test Cases

- **During normal periods**:
    - All the essential service workloads will be made available & get distributed among the data center and the public cloud

- All the non-essential services will be made openly available through the datacenter



- **During outages:**
  - All the non-essential services will be made unavailable to the public
  - All the essential services will be redirected only to the public cloud



Non-Essential Services are down. Sorry for inconvinience.