Mercari's

Large Scale System Design Hackathon

Team 5

Inter-IIT Tech Meet 10.0

mercari

Design Documentation

1. Requirement Specifications

1.1. Patient Registration
1.2. Hospital, Lab and Pharmacy Inventory Management
1.3. Doctor Appointment Scheduling.

2. Design Decisions

2.1. Assumptions and Trade-offs

- We are using Kubernetes for container orchestration, and docker as the underlying containerization software.
- We are making sure that the entire hospital chain is composed of decoupled microservices, which allows us to scale the services independently.
  - One of the drawbacks of the microservices approach is the service calls taken to serve a request have substantial latency.
- We are using normalized relational databases, for maintaining consistency and removing redundancy.
- We are using a centralized auth service, which makes sure all of the service calls are authenticated and authorized.
- The entire Kubernetes cluster is monitored for all the relevant metrics, making it easy for the developers to identify and track the issues with the service calls if needed.
- We are using Sentry for managing centralized error logging, making managing and tracking errors much simpler, which helps to audit our application with ease.
- We are using AWS RDS as the centralized database and AWS EKS for the Kubernetes Cluster. Using a managed service helped in cutting short of the development time while providing scalability and high availability out of the box.
- We chose AWS over GCP, due to the fine-grained control AWS provides. GCP has an opinionated approach, which helps to simplify the development, but we need more custom control in the Kubernetes deployment.

2.2. Tech & Infrastructure Stack

For the services implemented, we are using -

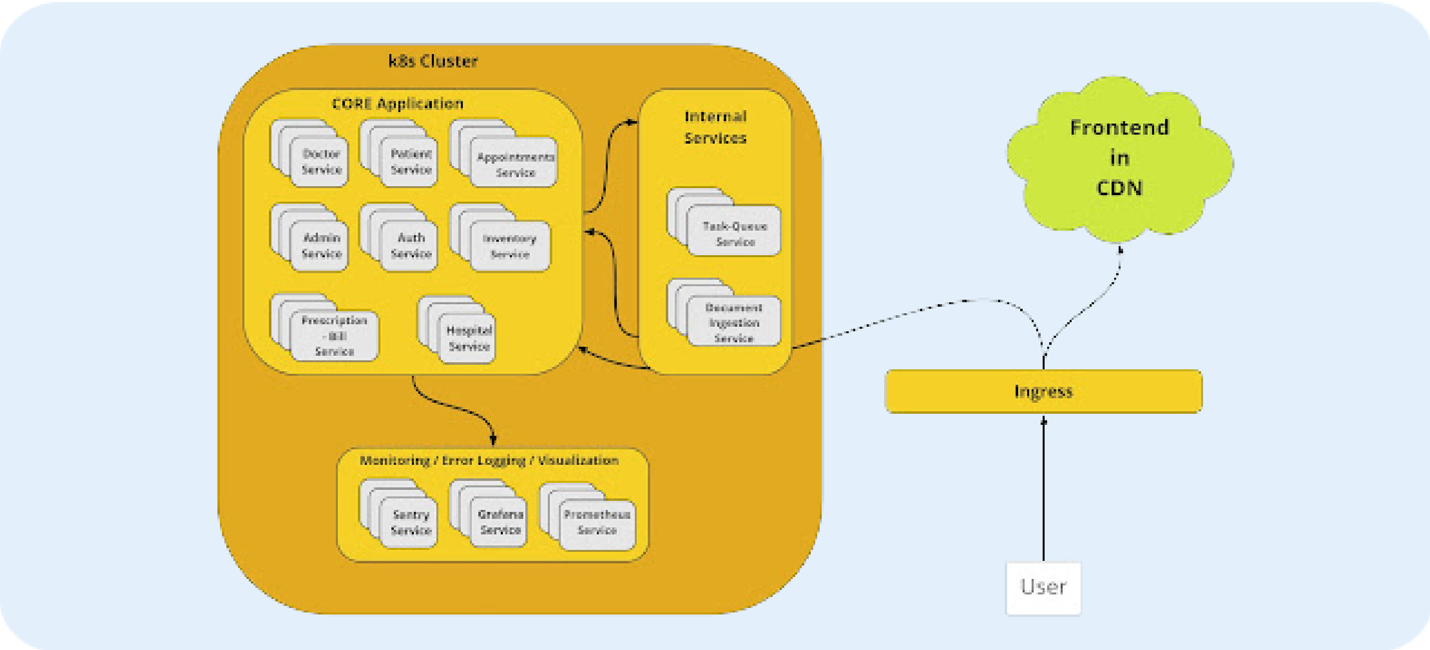
- NodeJS, ExpressJS for hospital and patient service.
- Python, Flask-RestX, and uWSGI/Gunicorn for the rest of the micro-services.
- MySQL as the RDBMS, and Redis as an in-memory database.

- Celery with a distributed RabbitMQ deployment as a message broker for a task queue to carry out asynchronous tasks.

→ For the Infra side of things, we are using AWS EKS, Prometheus, Sentry, Grafana, AWS S3, AWS RDS.

→ The S3 ingestor service is public cloud-agnostic, we can plugin any object storage as we wish. We have added the S3 plugin as one of the cloud services.

2.3. System Architecture Overview



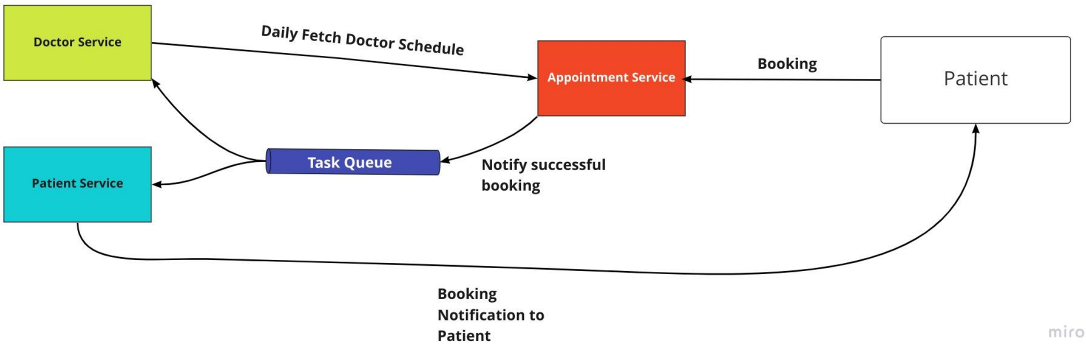
- We have broken down the entire project cluster into multiple microservices, which helps us in decoupling individual services and scaling them independently based on our requirements
- We have an Open Source **Integrated Monitoring/Error Logging/Visualization Software Suite**, which helps us in ***identifying/tracking down/visualization*** the metrics associated with the project
  - Having Open Source software dependency helps us in preventing vendor-lock-in, and ability to get latest updates/feature additions for the software.
- The entire setup is based on Kubernetes, so we can easily scale horizontally.
- We have custom dashboards for visualizing the cluster, and the metrics associated. This helps us in identifying the health of the cluster, and provision more Nodes if needed.

2.4. Important Flows

→ Auth Flow

- Given the requirement of compatibility with mobile apps, JWT is the best authentication mechanism to use.

- Using JWT helps us to get rid of Load Balancer affinities that affect traditional Cookie-based systems that rely on the login data stored in the RAM of the server.
- We use a global secret that stores the JWT Secret key. Hence, the secret key is shared across multiple microservices.
- A separate Auth service handles login, registration, and refreshing of tokens. It also handles the blacklisting of tokens and provides APIs for checking this blacklist through inter-service API calls.



➔ Appointment Saga

- We notice that the number of users catered by the doctor service and the appointment service vary by an order of magnitude of 50-100, hence, they need to be scaled separately. For example, in bigger cities, we have a lot of patients that come during weekends.
- The appointment service is configured to pull the schedule of each doctor on a daily basis and then book patients in the time slot based on this pulled view of the schedule.
- As patients book slots, the booking event is pushed to a Task Queue that asynchronously messages the doctors and patient regarding the confirmation of booking.
- This allows us to work with an **eventually consistent** appointment scheduling system, which is perfect for our use case.

➔ Storage Access Control

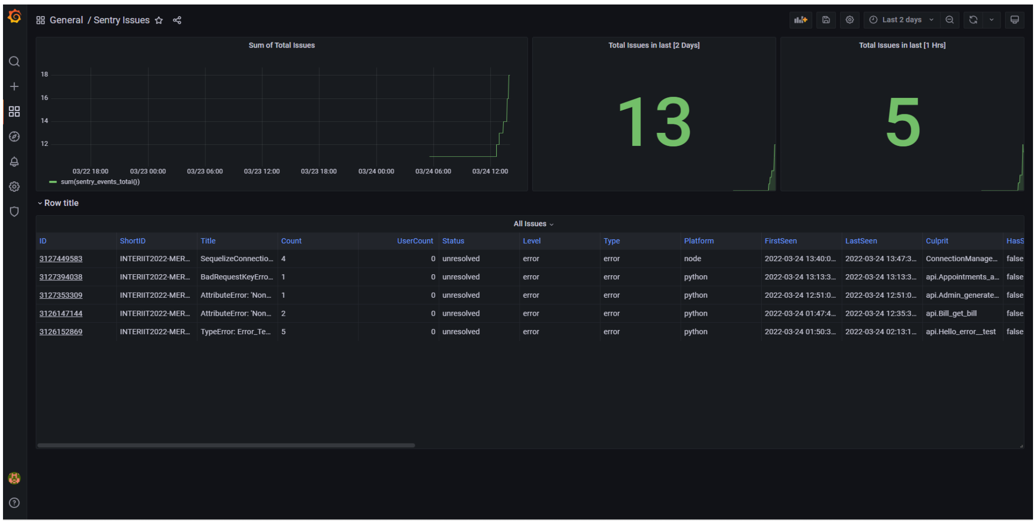
- We implement Role-Based Access Control (RBAC) for the prescriptions, lab reports, and medical history files uploaded by the users of the system.
- We implement this by creating a thin wrapper over S3 which checks the ownership of files using metadata stored in a Redis database service.
- This storage wrapper can be easily extended to any other object storage system as well.

➔ Monitoring, Error Logging and Visualization

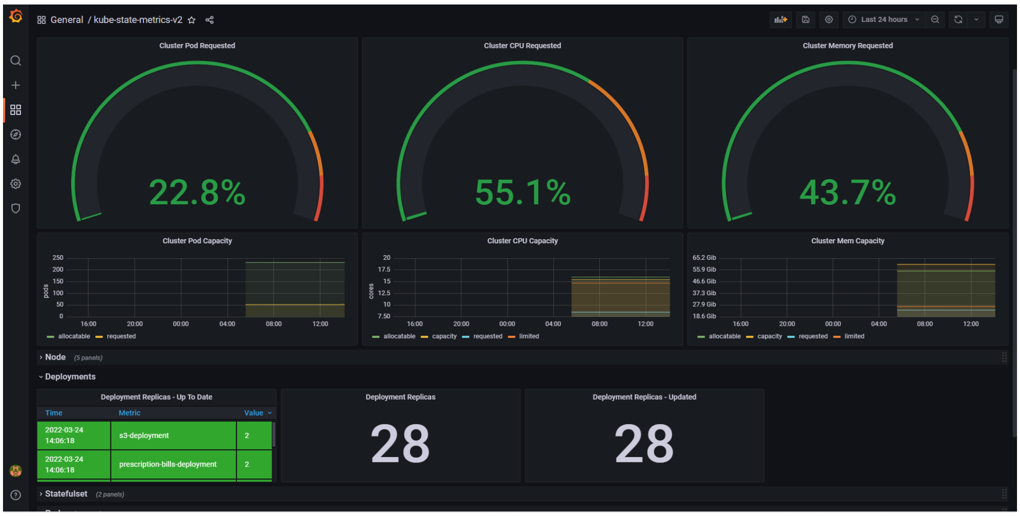
- *We understand that monitoring and visualization are an important part of running large-scale distributed systems. Having a transparent overview of the health of the application is really important.*
- We are using the Grafana Dashboard for rich visualizations of the cluster metrics.
- We have Service Level Metrics for showing the 2xx/3xx/5xx errors for each of the requests handled.
- We have also added p95/p99 latency for getting an idea of the latency for each service request.

- We have another dashboard for getting relevant metrics for the cluster, which helps us in provisioning more Nodes, in case the requested percentage(%) of resources crosses a threshold.
- We have an uptime dashboard for getting the percentage(%) of available replicas for the service.

Service Level Metrics



Centralized Error Logging Dashboard



Kube-state-metrics: Kubernetes Cluster Metrics

3. Scalability and Portability

- The use of Kubernetes allows us to cater to the national population easily, scaling up to as much as India's population, while maintaining a 99.99% of availability. However, for a growing user base, our modular design allows for easy swapping of components.
- In particular, we can replace the managed MySQL instances we use for our database with more resilient CockroachDB for geo-distributed scaling.
- We have taken special care to build our solution solely based on Open Source components with very minimal vendor lock-in. Our solution can be deployed in any popular Kubernetes service (EKS, GKE, AKS) or on-premises through OpenStack.

4. Cost Analysis

We estimate that for a national level usage, we would need about 20 m5.large EKS nodes with 5 db.t4g.large MySQL RDS instances and 100GB/day of data egress and about 50TB data stored in s3. These are the major players in the costing landscape.

Since the budget is of the utmost priority while designing a system for government hospital chains, we have tried to make this as cost-efficient as possible.

Given these constraints, the daily cost of running the application on a national scale is as follows (as per current AWS rates in ap-south-1):

Resource	**Rate**	**Quantity**	**Total price per day**
m5.large instances	$0.101/hr	20	$48.48
db.t4g.large	$0.167/hr	5	$20.04
Egress	$0.1093/GB	100 GB/day	$10.93
NAT Gateway	$0.056/GB	100 GB/day	$5.60
S3	$0.025/GB	50 TB/month	$42.67
EKS Cluster charges	$0.1/hr	1	$2.4
		Total	$130.12 (~ ₹ 10,000)

5. Deliverables

We have exposed all our services to the external network for evaluation purposes. The links for accessing the Swagger UI for API testing are as follows :

admin	http://a9c092961e2974c9c9aa6d87d6ec23ad-1318982453.ap-south-1.elb.amazonaws.com
appointments	http://a2b9bef1490fa48d18825d04f399814b-1958880394.ap-south-1.elb.amazonaws.com

auth	http://a0f4b292f577243a5971d04ff7da5b94-987059869.ap-south-1.elb.amazonaws.com
doctor	http://a3d4df110319947cda7c3e950194e806-1836406126.ap-south-1.elb.amazonaws.com
hospital-crud	http://aaf792c0a22124ee59a68c5a90407979-1714793882.ap-south-1.elb.amazonaws.com
inventory	http://aee59aa5b31254a1785b8e39ddc1814a-1896607215.ap-south-1.elb.amazonaws.com
patient-crud	http://a5503b4300e3742af8d8b9197298b705-2124458015.ap-south-1.elb.amazonaws.com
prescription-bills	http://af0fdf42db6fb4bcc9b460c155384d86-289534987.ap-south-1.elb.amazonaws.com
phpmyadmin	http://a3b2ed2f68be0488f90562aec2f95970-54686662.ap-south-1.elb.amazonaws.com
ingestion (s3)	http://ae18803d8fef4493b984b56bb3c8b050-579456122.ap-south-1.elb.amazonaws.com
task-queue (tq)	http://a4daf9353217547d3a283965ab13367a-121713899.ap-south-1.elb.amazonaws.com
grafana	http://a35533eeebc134e2a8fd73739b48a532-1813051043.ap-south-1.elb.amazonaws.com
prometheus	http://a238d1c38a3d14b54bdb92481d53a12f-1330506167.ap-south-1.elb.amazonaws.com/

Grafana Username: admin
Grafana Password: QKrc9xbGrTliNLTV55zcZqrUOXv4BzGg1CJXJAZa

Frontend URL: http://mercari-interiit-website.s3-website.ap-south-1.amazonaws.com/

6. Future Enhancements

- We can integrate tracing for the entire Kubernetes setup -
  - Tracing provides us the full path of the service calls, and with inter-micro-services communication, it becomes difficult to debug and reproduce the errors.
  - Having a tracing infrastructure helps us in quickly identifying issues
- Having a centralized logging setup, like grafana Loki, would help in accessing/aggregating the logs for all of the micro-services in one place.
- Configuring Prometheus Alerts, in case of a node crash, unexpected rise in CPU/Memory Usage for every deployment/node/pod, unexpected increase in latency, unavailability of microservices.
- We can set up Error Budgets / Service Level Objectives / Service Level Agreements for each of the services, to make sure we have transparency with the clients about the real-time performance of the services.

7. References

1. https://microservices.io/patterns/data/saga.html
2. https://docs.celeryq.dev/en/stable/
3. https://prometheus.io/
4. https://grafana.com/
5. https://sentry.io/welcome/
6. https://aws.amazon.com/eks/
7. https://redis.io/
8. https://www.cockroachlabs.com/