

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**Cloud Computing Report (25ECSC305)**

**Name: Chinmayi P Hosur  
USN: 01FE23BCS262  
Roll No: 153**

**Submitted To:**

**Mrs S. G. Kanakareddi,**

**Table of Contents**

1. **Introduction**
2. **BloodLink Application  
    2.1 Problem Statement  
    2.2 System Architecture Description  
    2.3 Interaction Flow  
    2.4 Deployment Platforms  
    2.5 Steps Used for Deployment**
3. **BloodLink Frontend Features  
    3.1 Homepage  
    3.2 User Authentication  
    3.3 Donor Dashboard  
    3.4 Hospital Dashboard  
    3.5 Admin Dashboard**
4. **BloodLink Backend Features**
5. **Micro-Recipe Application – Docker Deployment**
6. **Micro-Recipe Application – Kubernetes Deployment**
7. **Micro-Recipe Application – Microservices using Docker**
8. **Ansible Automation for Micro-Recipe Application**
9. **Final Conclusion**

**1. Introduction**

This report presents the design and deployment of two real-world applications—BloodLink and Micro-Recipe Community Board—using modern cloud computing and DevOps technologies. The work demonstrates both platform-managed (PaaS/SaaS) and infrastructure-managed (Docker, Kubernetes, Ansible) deployment strategies.

 **2.BLOODLINK – Online Blood Donation Request Availability System**

**2.1)Problem Statement:**

BloodLink is designed to solve the delays and confusion that often occur when hospitals try to find suitable blood donors. Many existing systems depend on old donor lists, manual calls, or broad location filters like city or pincode, which are not accurate enough during emergencies.

BloodLink replaces this with a cloud-based platform where hospitals can quickly create blood requests, mark them as normal or urgent, and automatically notify matching donors based on real-time geolocation and blood group compatibility.

Donors can update their availability, receive alerts instantly, and respond to requests through their dashboard. Hospitals can track which donors have accepted, monitor open requests, and manage their blood inventory in one place. Features like automated request expiry, geo-filtered matching and live maps make the entire process faster and more organized. Admin users have tools to verify accounts, view system activity, and maintain overall control.

Overall, BloodLink provides a simpler and more reliable way to connect hospitals and donors,  reduces response time and improves coordination during both routine and emergency situations.

**2.2)System Architecture Diagram Description**

The system architecture follows a **standard three-tier full-stack model** consisting of the **Frontend Layer**, **Backend/API Layer**, and **Database Layer**. Each layer handles a specific set of tasks and communicates with the others through well-defined interfaces.

**1. Frontend Layer (React + Vite)**

The frontend provides the user interface for all three roles — Donor, Hospital, and Admin. It sends HTTP requests to the backend API and displays data returned from the server. It handles session management (JWT tokens), form submissions, user navigation and visual presentation of dashboards.

**2. Backend/API Layer (Node.js + Express)**

The backend acts as the logical core of the system. It exposes REST endpoints for registration, login, donor management, request creation, inventory management, and admin functions. It validates inputs, runs the matching algorithm, updates MongoDB, and returns structured responses to the frontend. It also handles authentication, authorization, and optional notifications.

**3. Database Layer (MongoDB with Mongoose)**

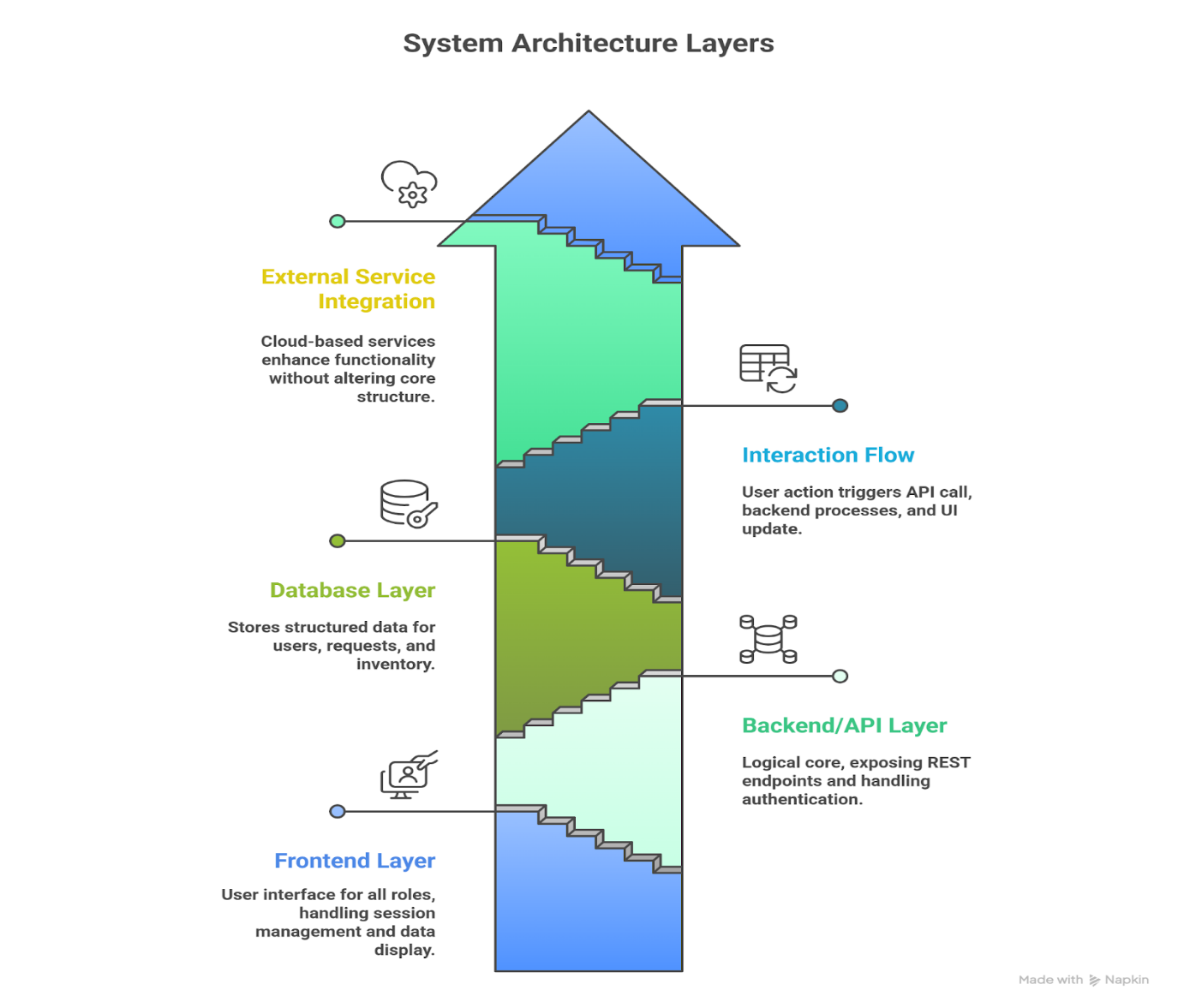
MongoDB stores structured collections for Users, Requests, and Inventory. The backend interacts with MongoDB using Mongoose models to perform CRUD operations. The database ensures persistent storage of user details, request data, donation pledges, and system logs.

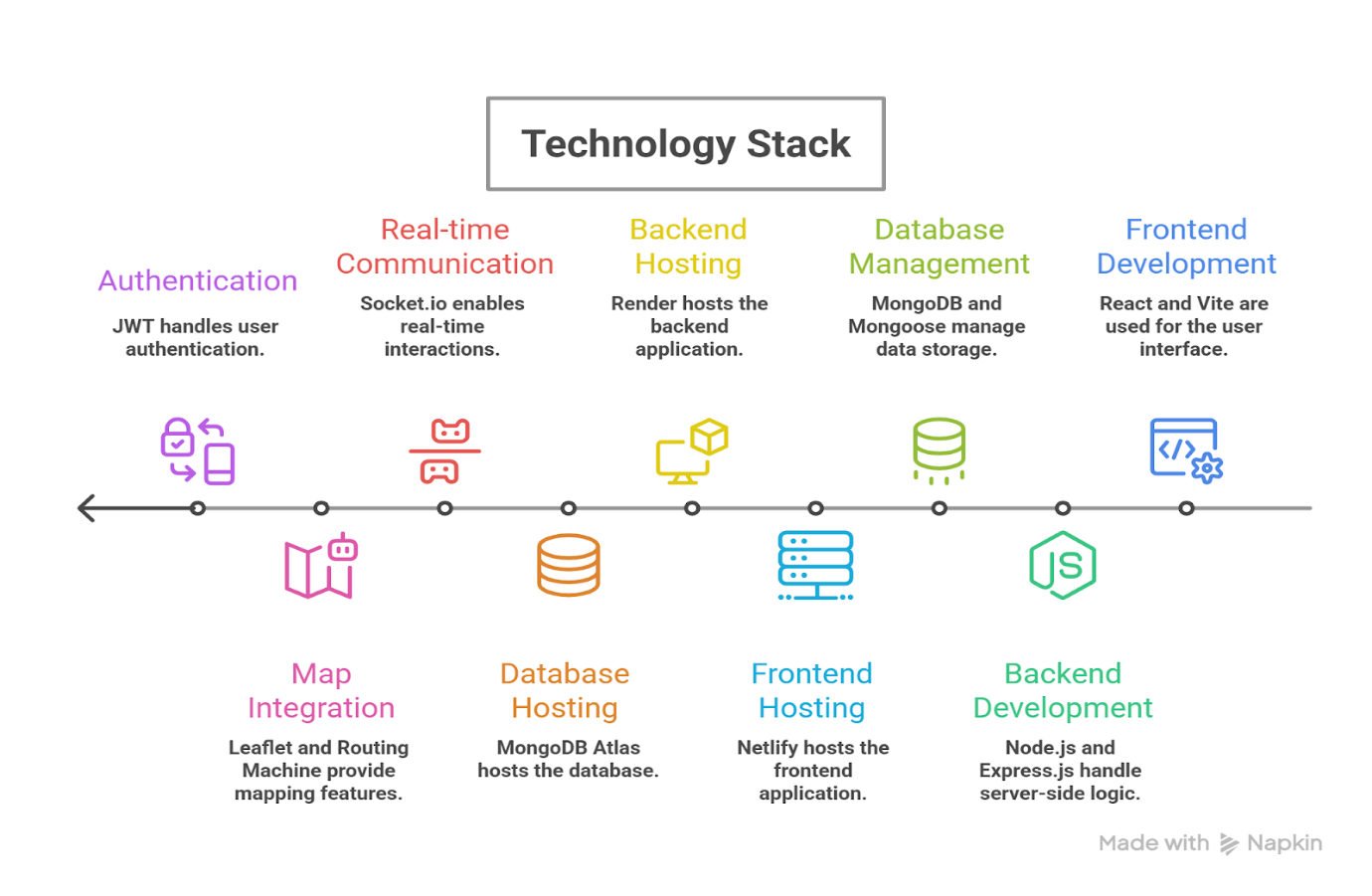
**Interaction Flow**

1. A user performs an action from the frontend (e.g., login, create request, toggle availability).
2. The frontend sends an API call to the backend.
3. The backend authenticates the request, runs business logic, interacts with MongoDB, and forms a JSON response.
4. The frontend receives the response and updates the UI accordingly.

**4. Optional External Service Integration**

The architecture can be extended with cloud-based services such as email/SMS APIs, cloud hosting platforms, geolocation services, and push notification systems. These external services plug into the backend layer without altering the frontend or database structure.





**2.4.Deployment Platforms**

1. **Backend – Render (PaaS)**

Render is a **Platform as a Service (PaaS)** used to deploy the backend API. It automatically builds and runs the Node.js server, manages the environment variables, and keeps the service online without manual server setup.

1. **Frontend – Netlify (PaaS)**

Netlify is a **PaaS** used to deploy the frontend React application.  
 It provides continuous deployment from GitHub, automatic builds, and a global CDN for fast loading.

1. **Database – MongoDB Atlas (DBaaS)**

MongoDB Atlas hosts the database in the cloud.  
 The backend connects to it using a secure connection string, allowing the application to store and retrieve data globally.

**2.5. Steps Used for Deployment**

1. **Backend (Render)**

* Push backend code to GitHub.
* Create a new Render Web Service.
* Connect the GitHub repository.
* Add environment variables (.env values).
* Select build command: npm install
* Select start command: npm start
* Deploy and verify API URL.

1. **Frontend (Netlify)**

* Push frontend code to GitHub.
* Create a new site in Netlify.
* Connect GitHub repository.
* Set build command: npm run build
* Set publish directory: dist/
* Deploy site and test frontend URL.
* Update frontend .env with Render backend API URL if needed.

1. **Database (MongoDB Atlas)**

* Create free cluster on MongoDB Atlas.
* Create a database user and password.
* Add IP access (0.0.0.0/0 for testing).
* Copy connection string from Atlas.
* Paste connection string into backend .env as MONGO\_URI.
* Backend connects automatically during deployment.

**3)Bloodlink Frontend Features**

* The interface is built using **React + Vite**, ensuring fast rendering and smooth navigation.
* State is managed using React hooks, and data is fetched through Axios-based API calls.
* The UI is intentionally minimalistic, making the platform clean, intuitive, and easy to demonstrate in the report.

1. **HomePage:**

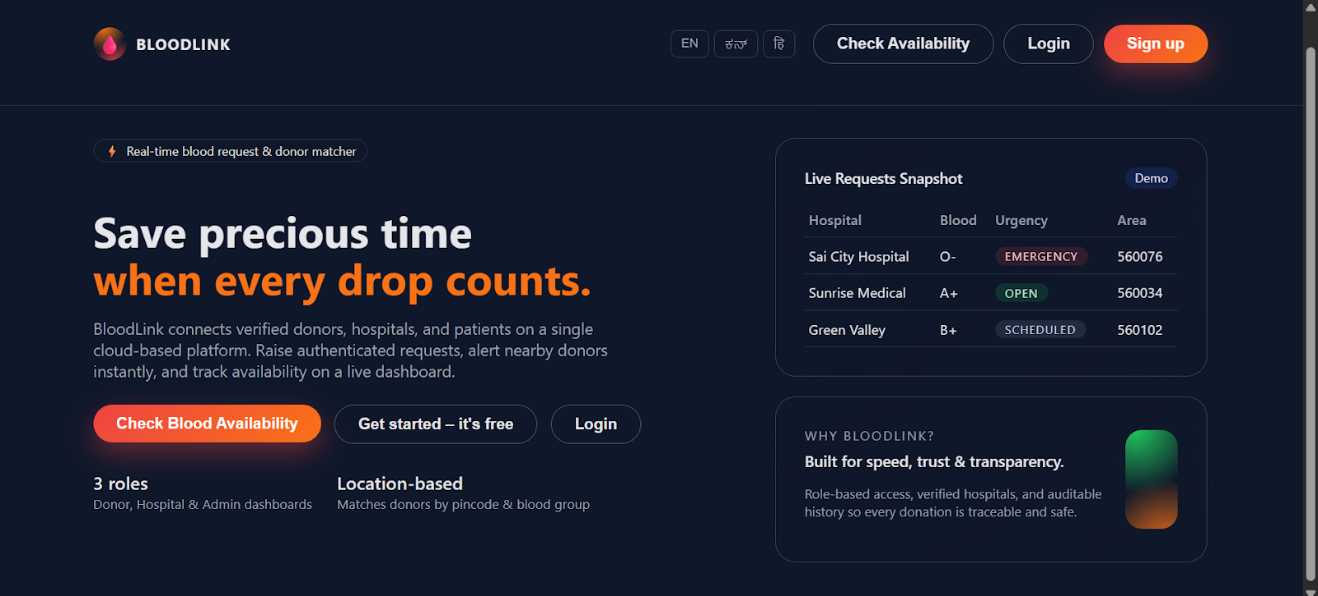


Fig. 1.1 Represents the Home Page of our BloodLink Website

1. **User Sign-Up and Login Interface**

The sign-up page allows users to create an account by selecting their role — Donor, Hospital, or Admin. Each role requires different details such as blood group for donors or hospital information for hospitals. The frontend validates all fields and sends the data to the backend authentication API.

The login page accepts registered email and password credentials and verifies them through the backend API. Based on the user’s role, the frontend automatically redirects them to the correct dashboard (Donor, Hospital, or Admin). JWT tokens are stored on the client side to maintain session state securely.

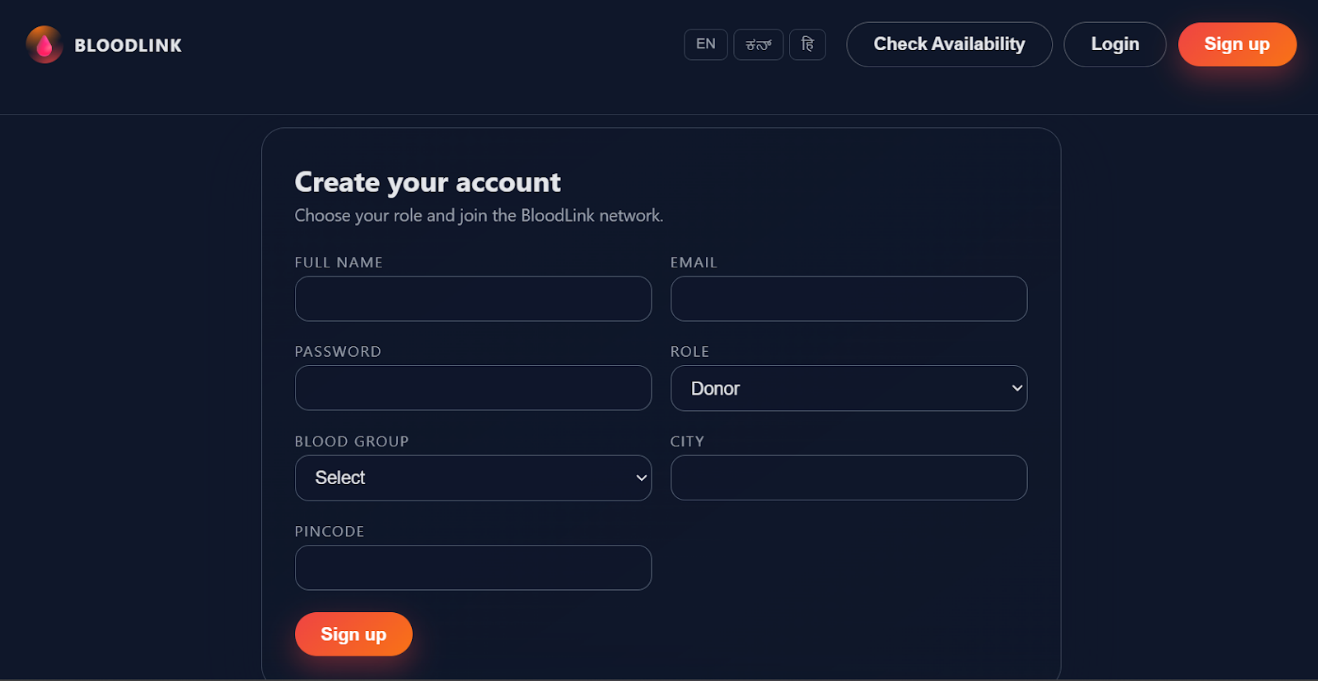


Fig. 1.2 represents the Sign up page.

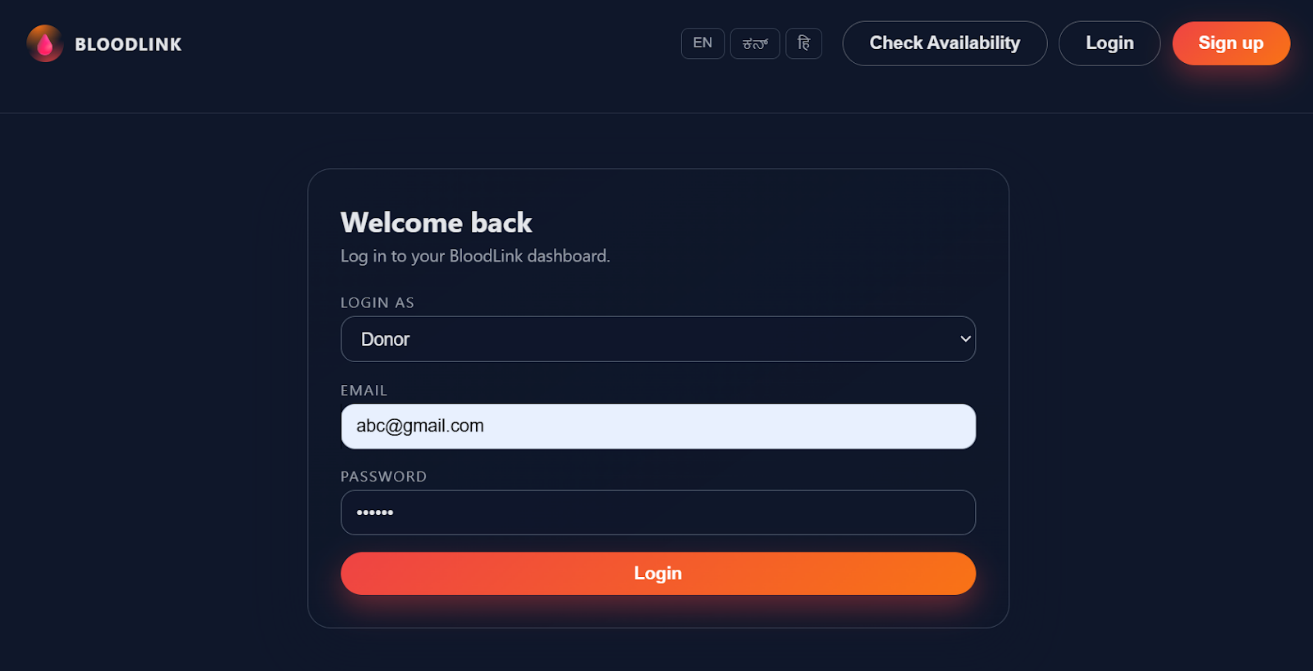


Fig. 1.3 represents the Log-in page.

1. **Check blood availability**

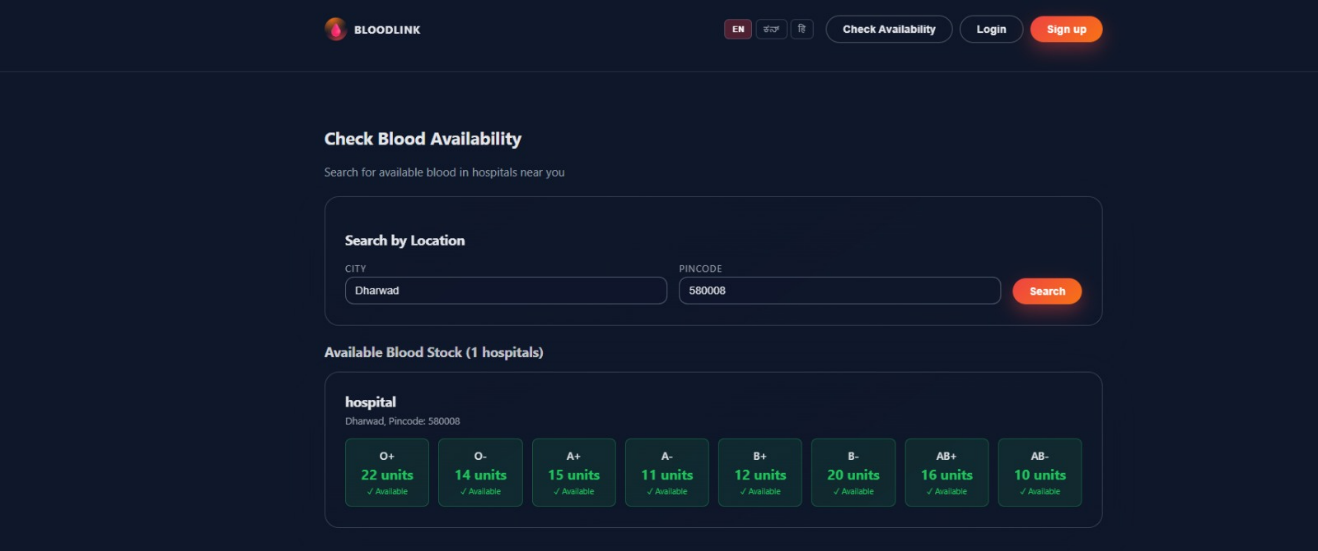


Fig. 1.4 shows the blood availability feature present in the home page.

1. **Donor Dashboard**

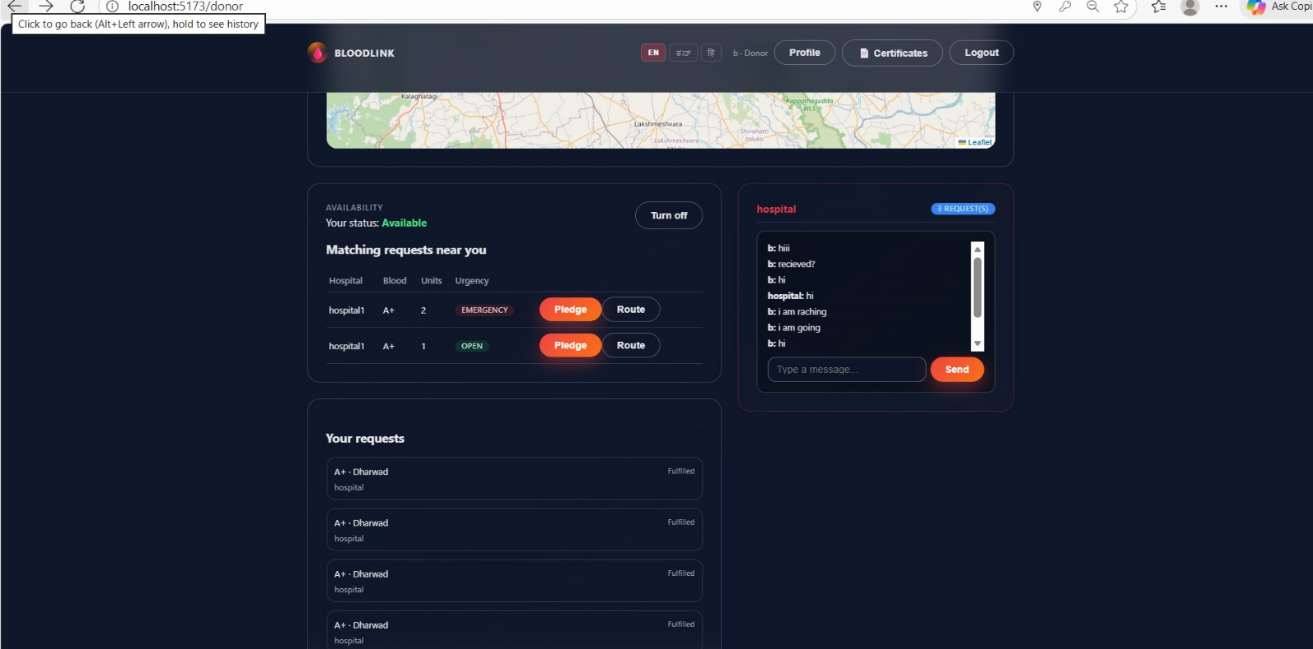


Fig. 1.5 shows the overall layout of the donor dashboard.

**1. Availability Toggle**

This toggle lets donors update their availability status instantly. When switched on, the frontend triggers an API call that marks the donor as “Available,” making them visible for matching with active blood requests.

**2. Matching Requests Display**

The dashboard shows blood requests that match the donor’s blood group and region (city/pincode). The frontend fetches these requests via API and displays them in a clean card layout for quick viewing.

**3. Pledge to Donate**

Each matching request includes a **“Pledge”** button. When clicked, the frontend sends a pledge confirmation to the backend and updates the UI to reflect that the donor has committed to the request.

**4. Matched Requests List**

Donors can also see all requests they have been matched to or pledged for. This interface gives donors a quick overview of ongoing and past commitments, improving transparency and tracking.

**5. Donors gets Certificate**

Donors get e-certificate for blood donation and also has reward points for every donation made.

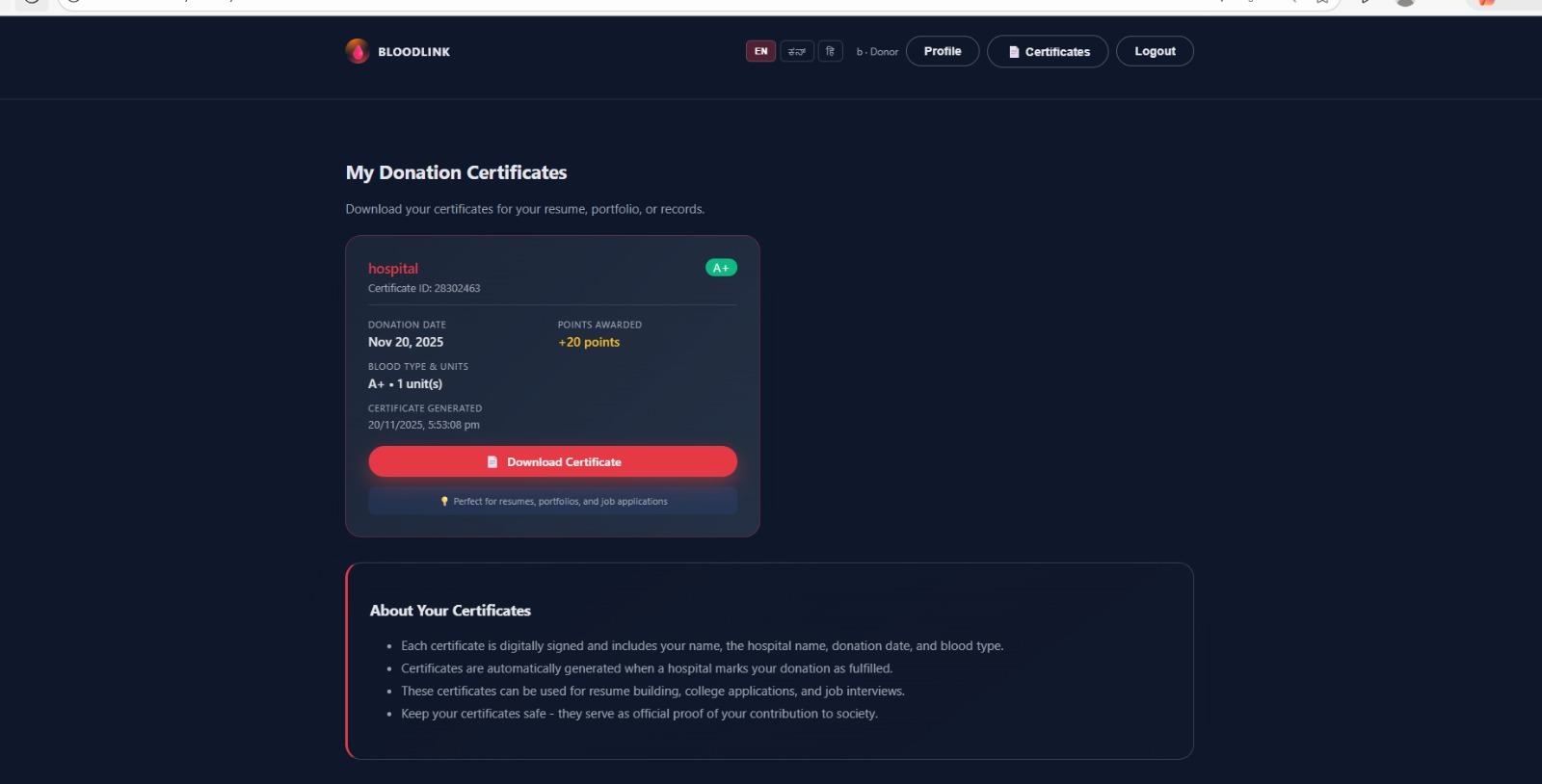


Fig. 1.6 represents a sample certificate a donor receives from the hospital for donating their blood.

**6. Live Mapping**

Displays real-time locations of donors and active hospital requests on an interactive map for faster and more accurate decision-making.

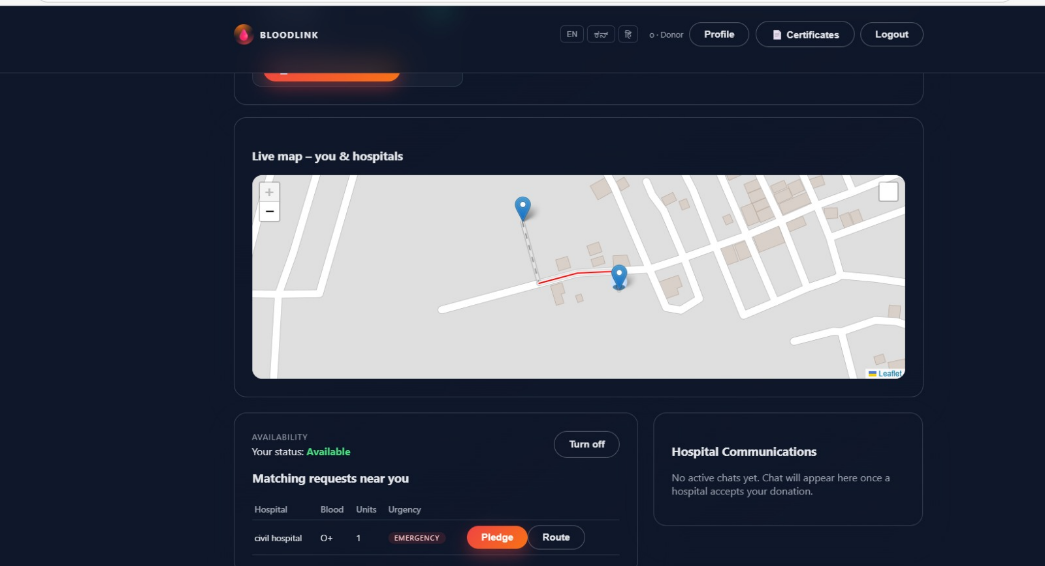


Fig 1.7 shows the live mapping feature between the donor’s and hospital’s location and the shortest distance between them.

1. **Hospital Dashboard**

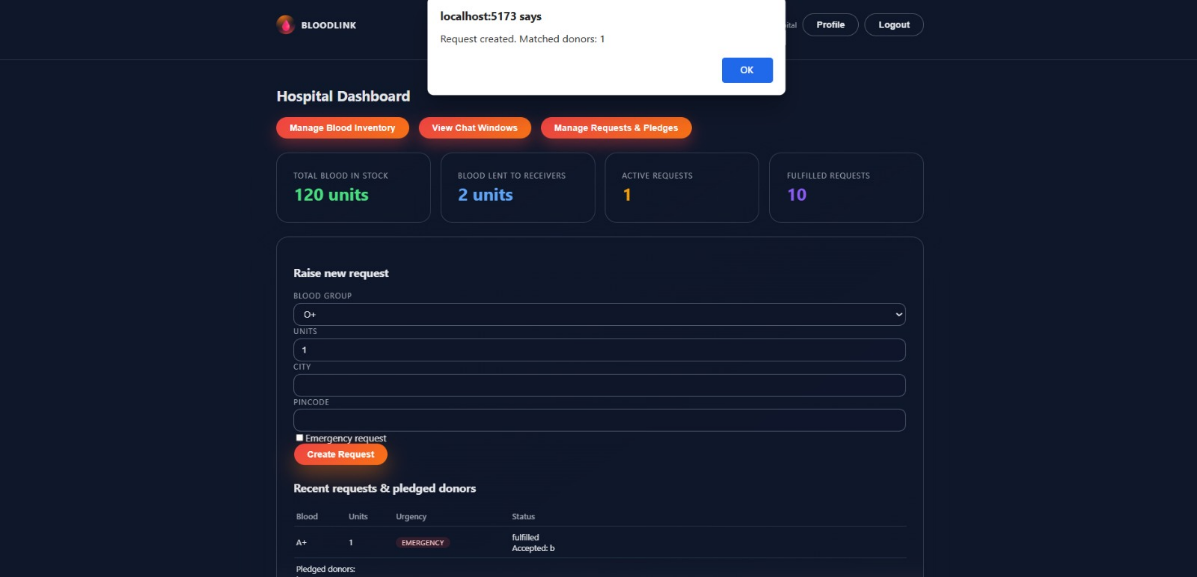


Fig. 1.8 shows the overall layout of the Hospital dashboard

**1. Create New Blood Request Form**

Hospitals can raise new blood requests through a structured form that captures details such as blood group, units required, location, and emergency status. The frontend validates user input and sends the request to the backend API.

**2. Request Status Management**

All requests created by the hospital are listed on the dashboard with status indicators (open, fulfilled, cancelled). Hospitals can update the status using simple UI actions, and the frontend instantly reflects changes.

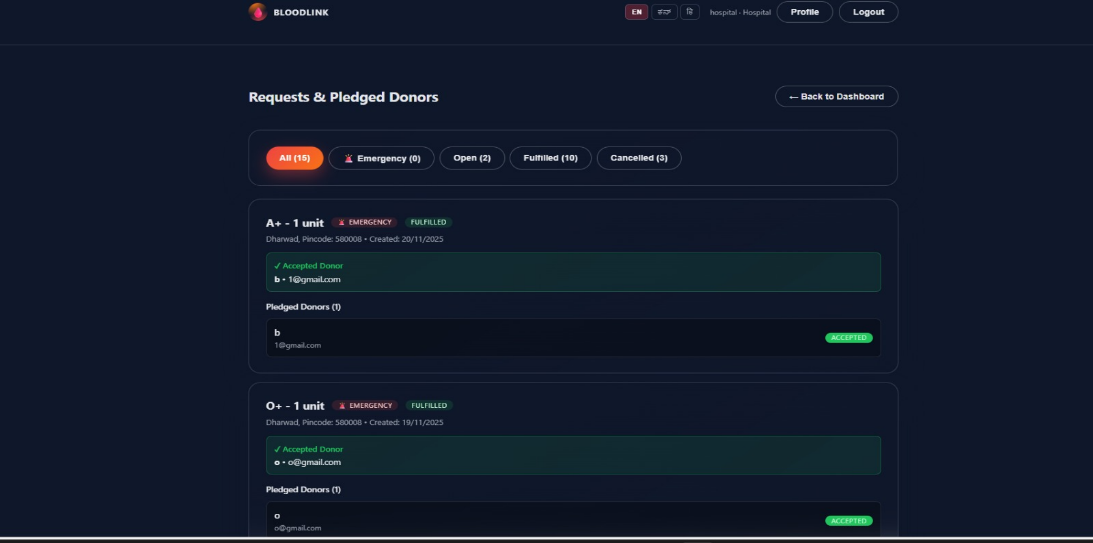


Fig 1.9 shows the Request and Pledged donors page

**3. Blood Inventory Management**

A dedicated section lets hospitals update and track available blood stock by blood group. The frontend provides a simple interface to add or adjust inventory values and sends updates to the backend in real time.

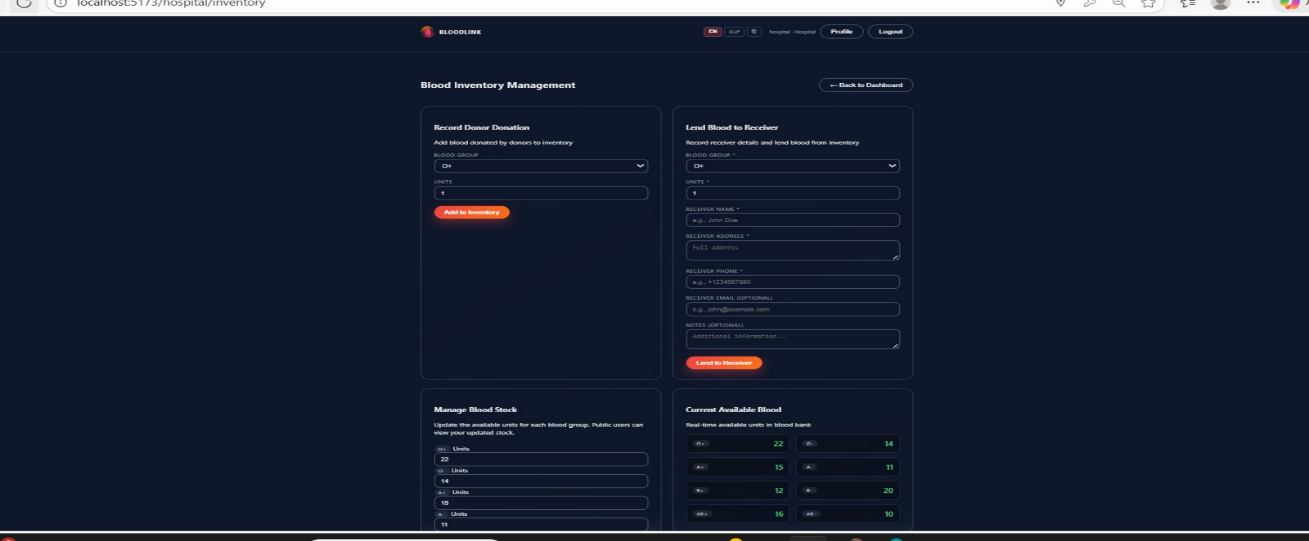


Fig 1.10 shows the Blood Inventory Management page in the Hospital Dashboard.

**4.Chat window**

Hospitals can talk to the donors by texting them, this uses socket programming to connect between hospital and donors.This includes real time texting between them . There is a separate chat window for unique donors.

**C. Admin Dashboard:**The Admin Dashboard is accessible only through a separate login ID, secret key, and secret password, which are configured beforehand. This ensures that only authorized administrators can access system-wide controls, manage users, and monitor platform activity.

**1. System Overview Statistics**

The admin dashboard displays summary metrics such as total donors, hospitals, active requests, and completed donations. These statistics are fetched from the backend and shown in an easy-to-read card layout.

**2. User Management Table**

Admins can view all registered donors and hospitals in a unified table. The frontend provides a verification toggle that allows admins to approve or revoke user verification with a single interaction.

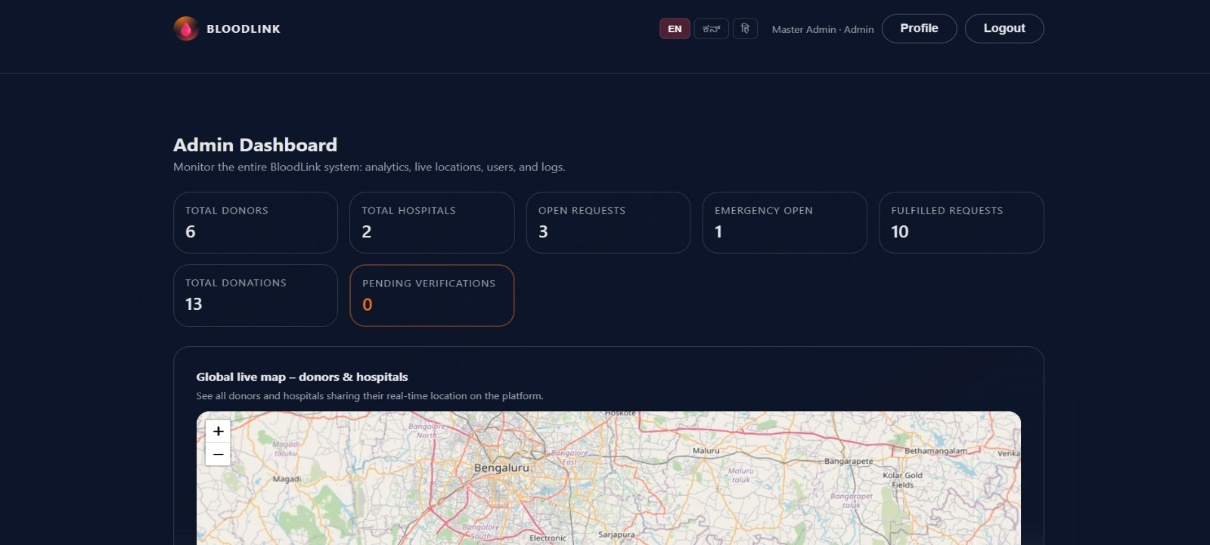


Fig 1.11 shows the overall layout of the Admin Dashboard.

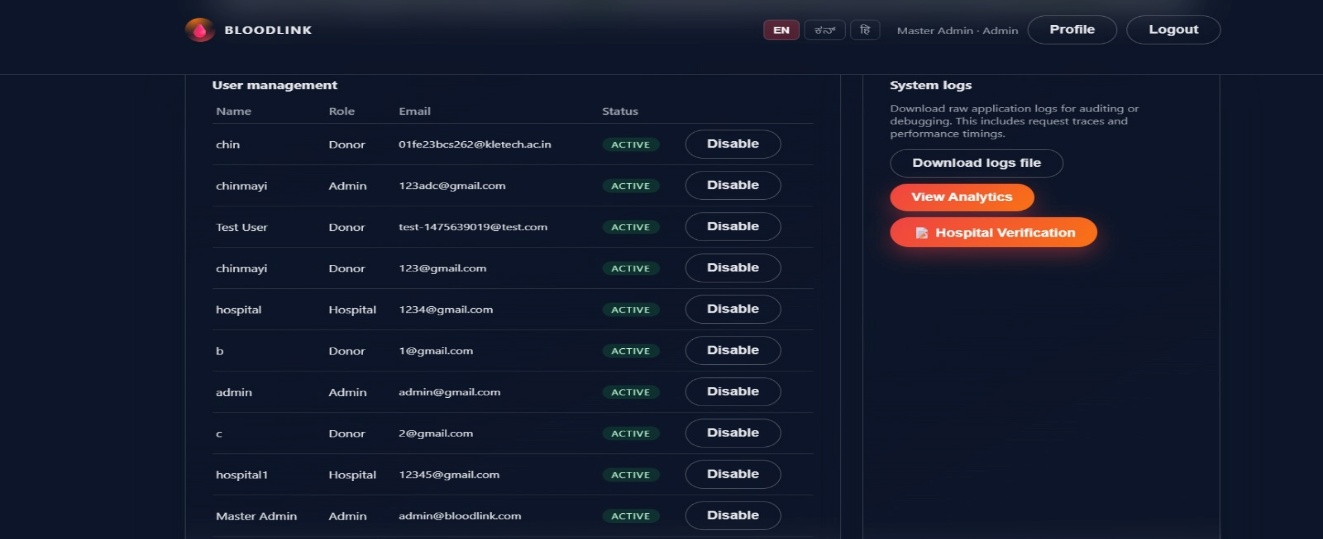


Fig 1.12 shows the user management and system logs feature in the Admin Dashboard.

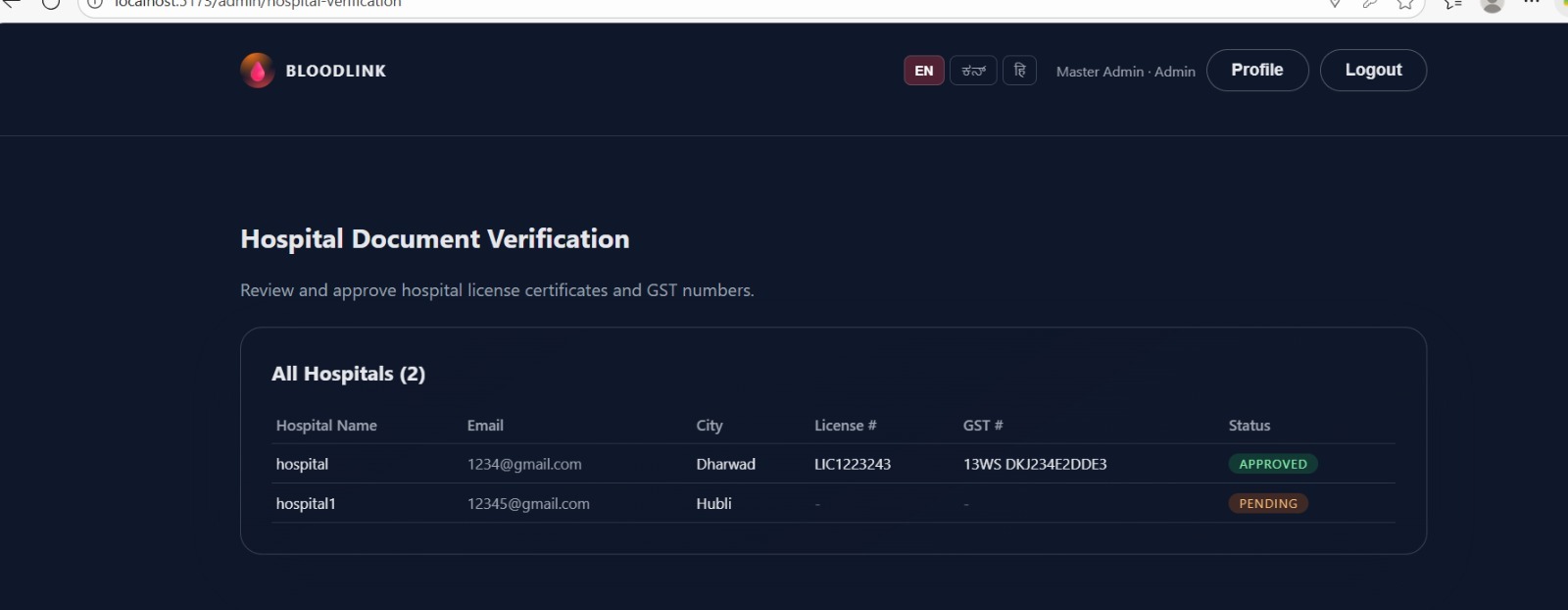


Fig 1.13 shows the Hospital Verification feature in the Admin Dashboard.

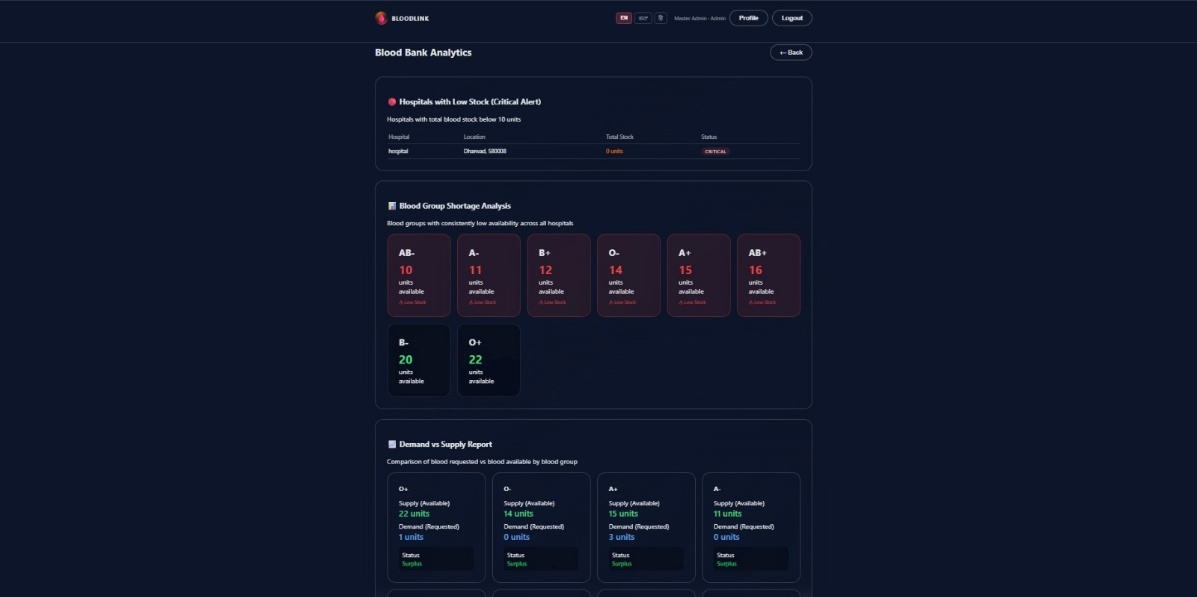


Fig 1.14 shows the Blood Bank Analysis feature in the Admin Dashboard.

**4)Bloodlink Backend Features**

The backend of BloodLink is built using **Node.js, Express, and MongoDB (Mongoose)**, providing a structured REST API that handles authentication, user management, request creation, and matching logic. It is responsible for all role-based operations and ensures secure, validated communication with the frontend. The backend also manages persistent data storage, real-time request processing, and business logic execution.

**1. Role-Based Authentication**

The backend includes a secure login and registration flow using JWT (JSON Web Tokens). Each user — Donor, Hospital, or Admin — receives a role-specific token, enabling protected access to their respective dashboards and preventing unauthorized route access.

**2. Donor Management API**

The backend stores donor details such as blood group, location, availability status, and pledge history. Donors can update their availability or location, and the backend uses this information to filter and match them with hospital requests efficiently.

**3. Hospital Request Management**

Hospitals can create new blood requests through a dedicated API that records blood group, units needed, emergency status, and location. The backend updates request status (open, fulfilled, cancelled) and ensures that request data stays synchronized across all dashboards.

**4. Matching Algorithm (Blood Group + Region)**

The backend runs a matching function that finds donors based on blood group and pincode (or geolocation in the enhanced version). When a new request is created, matching donors are fetched instantly and attached to the request record for the hospital to review.

**5. Inventory Management API**

Hospitals can maintain a blood inventory by updating available units for each blood group. The backend stores this information in the database and returns accurate, real-time inventory data to the frontend dashboard.

**6. Admin Control Endpoints**

The admin API enables system-level functions such as verifying donors or hospitals, viewing total stats, and listing all system users. These endpoints allow administrators to maintain platform integrity and ensure data accuracy.

**7. Notifications & Logging**

Whenever a hospital creates a request, the backend logs the matched donors and can be extended to trigger email/SMS notifications through external services like Twilio, SendGrid, or Nodemailer. This ensures timely alerts for urgent requests.

**8. Database Operations & Validation**

All input is validated through backend logic before being saved to MongoDB. The database models enforce schema consistency for Users, Requests, and other entities, ensuring reliable and structured data storage.

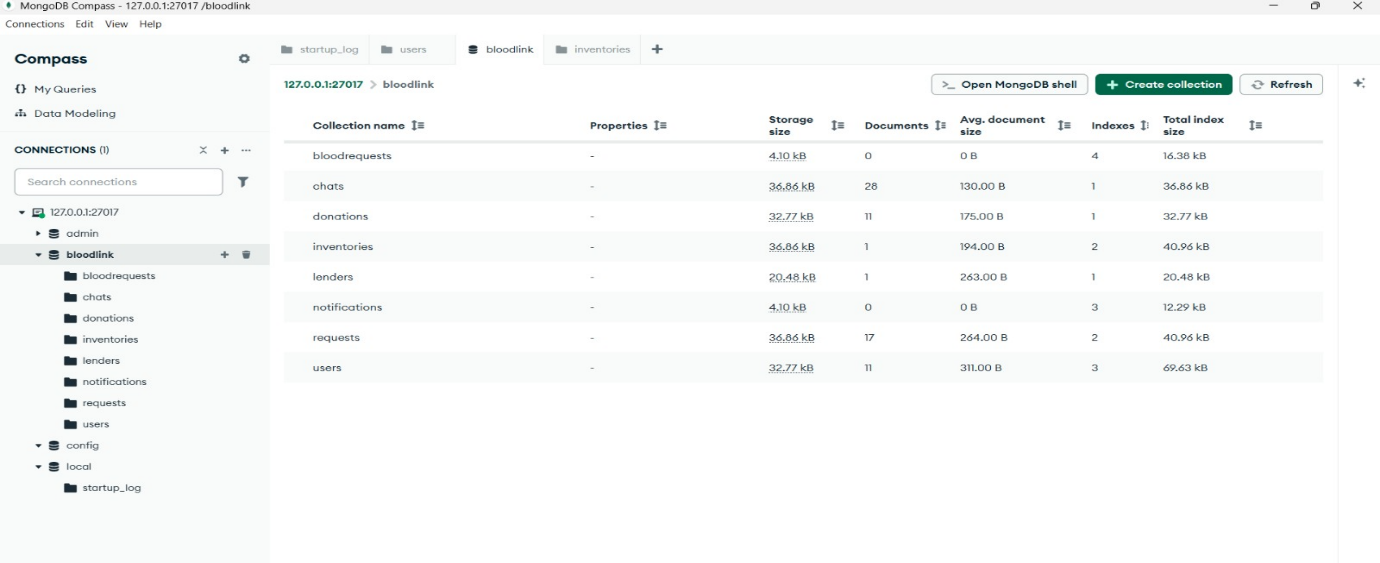


Fig 1.15 shows the database connected to our Bloolink website.

**5.Micro-Recipe Application – Docker Deployment**

**5.1. Overview**

This document describes how to deploy the Micro-Recipe Application using Docker and Docker Compose. The application follows a microservices-style setup with separate containers for frontend, backend, database, and monitoring.

The deployment uses:

* Docker & Docker Compose
* NGINX (reverse proxy)
* Node.js backend
* Frontend (Vite/React or similar)
* MongoDB
* Prometheus & Grafana (monitoring)

**5.2. System Architecture**

**Containers involved:**

* **frontend** – Web UI (Community Recipe Board)
* **backend** – API server
* **mongo** – MongoDB database
* **nginx** – Reverse proxy for frontend & backend
* **prometheus** – Metrics collection
* **node-exporter** – System metrics
* **grafana** – Monitoring dashboards

**Exposed Ports (example):**

* Frontend: 8080
* Backend API: 5000
* MongoDB: 27017
* Grafana: 3000
* Prometheus: 9090

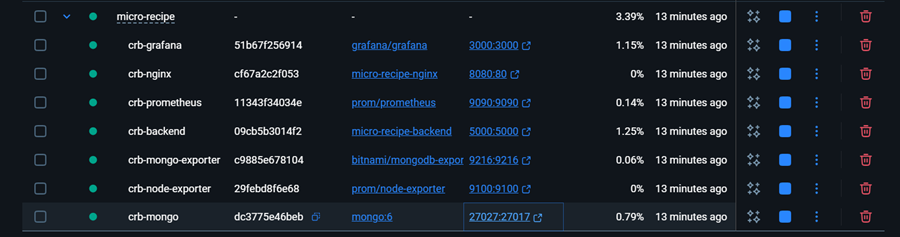


Figure : Running Docker containers for the Micro-Recipe application, including backend, frontend, MongoDB, Prometheus, and Grafana.

**5.3. Dockerfile Configuration**

**docker-compose.yml**

services:

frontend:

build: ./frontend

container\_name: micro-recipe-frontend

ports:

- "8080:80"

depends\_on:

- backend

backend:

build: ./backend

container\_name: micro-recipe-backend

ports:

- "5000:5000"

environment:

- MONGO\_URL=mongodb://mongo:27017/recipes

depends\_on:

- mongo

mongo:

image: mongo:6

container\_name: micro-recipe-mongo

ports:

- "27017:27017"

volumes:

- mongo\_data:/data/db

prometheus:

image: prom/prometheus

container\_name: micro-recipe-prometheus

ports:

- "9090:9090"

volumes:

- ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

node-exporter:

image: prom/node-exporter

container\_name: micro-recipe-node-exporter

ports:

- "9100:9100"

grafana:

image: grafana/grafana

container\_name: micro-recipe-grafana

ports:

- "3000:3000"

depends\_on:

- prometheus

volumes:

mongo\_data:

**5.4. Deployment Steps**

Step 1: Build and Start Containers  
  
bash:

cd micro-recipe

docker-compose up -d –build

Step 2: Verify Running Containers

bash:

docker ps

**5.5. Accessing the Application**

* Frontend UI: http://localhost:8080
* Backend API: http://localhost:5000
* Grafana: http://localhost:3000
  + Default login: admin / admin
* Prometheus: http://localhost:9090

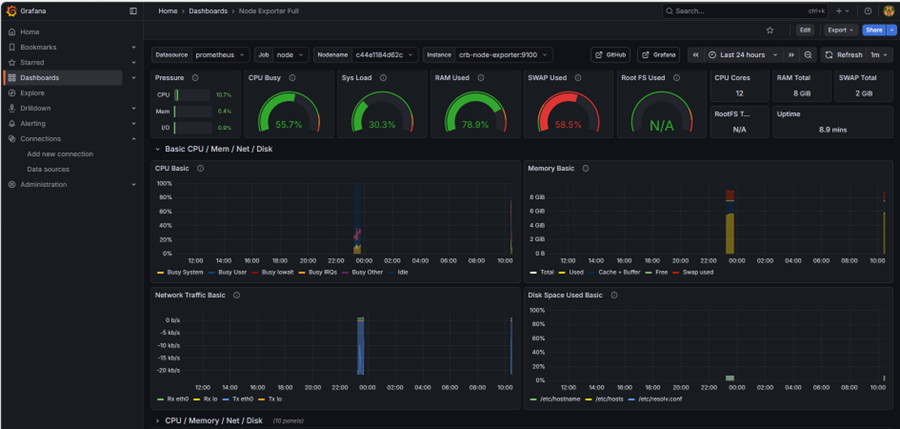
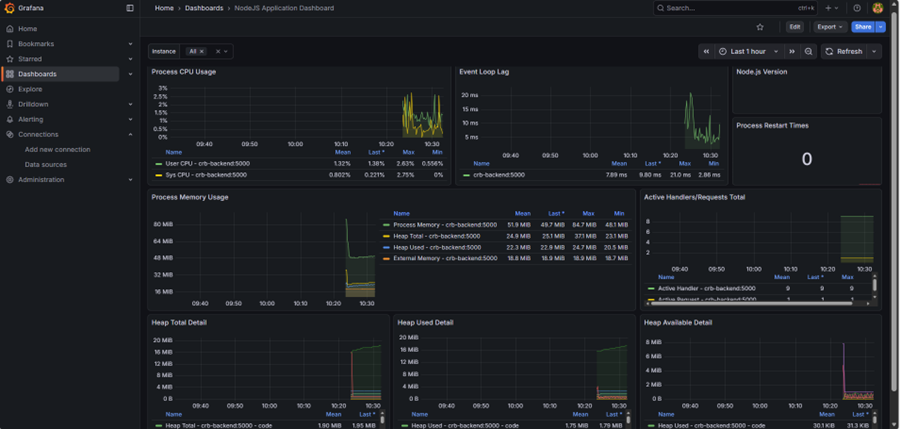


Figure : Grafana Node Exporter dashboard showing CPU, memory, disk, and system metrics collected via Prometheus.

Figure : Grafana Node.js Application dashboard visualizing backend CPU usage, memory consumption, and event loop latency.

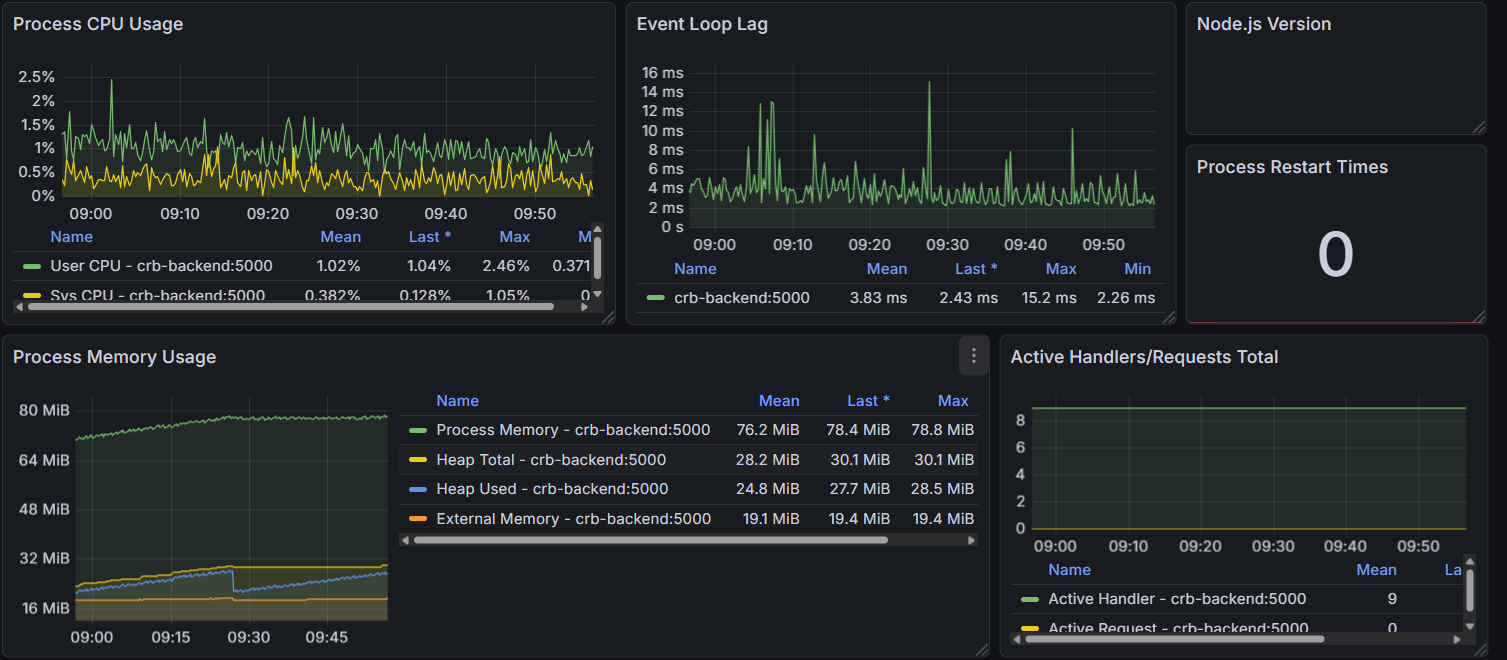


Fig) Graphs of cpu,process memory,event loop lag,active handlers.



Figure : Micro-Recipe Community Recipe Board frontend interface served via Docker and accessed through NGINX

**6.Micro-Recipe Kubernetes Deployment**

**6.1.INTRODUCTION**

The *Micro-Recipe Application* is deployed using **Docker Desktop’s Kubernetes cluster**, enabling a fully containerized, scalable, and production-ready environment on a local machine. The system consists of multiple microservices—Frontend, Backend (Node.js), MongoDB, Nginx, Prometheus, Grafana, Node Exporter, and Mongo Exporter—each running inside its own Kubernetes Pod.

The system is composed of the following services:

* **Frontend**: A simple web interface served using Nginx
* **Backend (Node.js)**: Handles API requests, database operations, and business logic
* **MongoDB**: Persistent database for storing recipes
* **Nginx**: Acts as a reverse proxy for routing frontend + backend API calls
* **Prometheus**: Scrapes metrics from all services
* **Grafana**: Visualizes metrics with dashboards
* **Node Exporter**: Collects OS-level metrics (CPU, RAM, Disk, Network)
* **Mongo Exporter**: Collects MongoDB performance metrics

**6.2. System Architecture**

The application uses a **Microservices Architecture** where each component runs in its own isolated environment (Pod) but works together as a single system.

**Component Breakdown:**

* **Frontend (Nginx):** \* **Role:** Serves the static HTML/JS/CSS files.
  + **Networking:** Exposed on Port 80 via a LoadBalancer service, making it accessible directly at http://localhost.
  + **Configuration:** Configured to proxy API requests (/api/) internally to the Backend Service.
* **Backend (Node.js/Express):**
  + **Role:** Handles business logic and API requests.
  + **Networking:** Accessible internally at http://backend-service:5000. It connects to the database via the hostname mongo.
* **Database (MongoDB):**
  + **Role:** Stores user and recipe data.
  + **Networking:** Protected inside the cluster. Accessible only by the backend or via a secure Port Forward tunnel (localhost:27018).
* **Monitoring Stack:**
  + **Prometheus:** Scrapes metrics from the Backend and Mongo Exporter.
  + **Grafana:** Visualizes these metrics in Dashboards (accessible via localhost:3001).

**6.3.Monitoring stack in detail**

**6.3.1 Prometheus**

Prometheus scrapes metrics every 5–15 seconds from:

|  |  |  |
| --- | --- | --- |
| Target | Port | Metrics Type |
| backend | 5000 | API metrics |
| node-exporter | 9100 | Node CPU/Mem/Disk |
| mongo-exporter | 9216 | DB metrics |
| prometheus | 9090 | Prometheus internal |

Prometheus is deployed as:

* Deployment
* ConfigMap containing prometheus.yml
* ClusterIP service

**6.3.2 Grafana**

Grafana visualizes Prometheus metrics through dashboards.

Dashboards imported:

* **1860** – Node Exporter Full
* **3662** – Prometheus Stats
* Custom DB dashboard
* Custom API latency dashboard

Grafana admin credentials:

admin / admin

Grafana connects to:

http://prometheus-service:9090

**6.3.3 Node Exporter**

Exports OS-level metrics:

* CPU cycles
* RAM usage
* Disk IO
* Network bytes sent/received

Deployed as a **DaemonSet**, ensuring one per node.

**6.3.4 Mongo Exporter**

Exports MongoDB metrics:

* Cache usage
* Operation counters
* Collection stats
* Query performance

Scraped via:

mongodb://mongo:27017



Fig 1) All the deployed and pods of the project

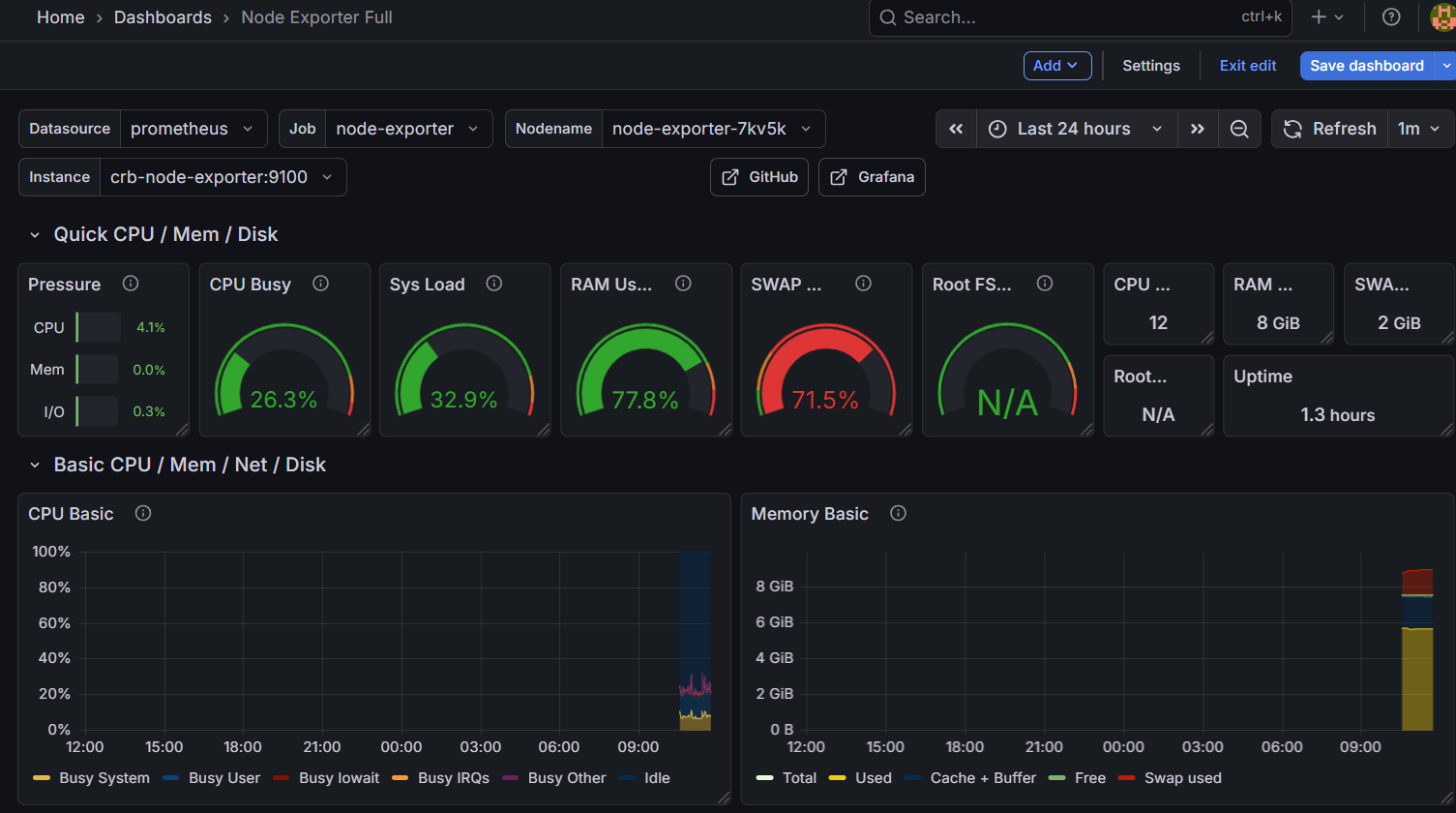


Fig 2)CPU,Sytem load,memory, ram used graphs in grafana

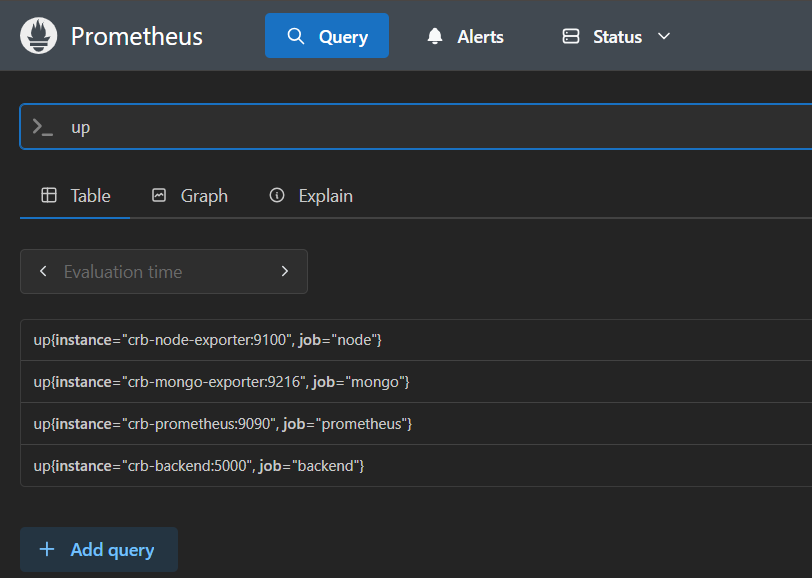
****

Fig3)Prometheus queries which are up

**6.4. Operational Quick Reference**

Save this section to manage your project in the future.

**Start / Update the Project**

Bash

# Apply all configurations in the k8s folder

kubectl apply -f k8s/

**Check Health**

Bash

# Watch pods start up in real-time

kubectl get pods -w

**Access Database (MongoDB Compass)** *Running this opens a tunnel so you can see the data inside K8s.*

Bash

kubectl port-forward service/mongo 27018:27017

# Connection String: mongodb://localhost:27018

**Access Monitoring (Grafana)** *Running this allows you to view your dashboards.*

Bash

kubectl port-forward service/grafana-fresh-service 3001:3000

# URL: http://localhost:3001

# Login: admin / admin

Prometheus scrapes:

* Backend metrics (if exposed)
* Node exporter at :9100
* Mongo exporter at :9216
* Prometheus self-metrics at :9090

Grafana connects to Prometheus via:

http://prometheus-service:9090

Dashboards used:

* Node Exporter Full (1860)
* Prometheus Stats (3662)
* Custom Backend Metrics Dashboard
* Custom MongoDB Dashboard

**6.5.EXPLANATION OF KEY YAML FILES**

k8s folder contains:

**backend.yaml**

* Deployment (2 replicas)
* Service (ClusterIP)
* ENV: MONGO\_URI

**frontend.yaml**

* Deployment (2 replicas)
* LoadBalancer service (public UI)

**mongo.yaml**

* Deployment
* Service
* PVC for storage

**grafana.yaml**

* Deployment + LoadBalancer
* Default credentials set via ENV

**prometheus.yaml + prometheus-config.yaml**

* Prometheus Deployment
* ConfigMap containing scrape configs
* ClusterIP service

**node-exporter.yaml**

* DaemonSet (runs on every node)
* Service exposing metrics

**mongo-exporter.yaml**

* Deployment + Service scraping MongoDB

**CONCLUSION**

This project demonstrates a complete microservices deployment workflow using Kubernetes. By integrating monitoring tools like Prometheus and Grafana, the system achieves high observability, enabling developers to track performance, resource usage, and cluster health in real time.

**7.Micro-Recipe Application – Microservices Deployment using Docker**

**7.1. Introduction**

This document explains the deployment of the **Micro-Recipe Application** using a **microservices architecture** powered by **Docker**. Each core functionality of the application is deployed as an independent service, enabling scalability, fault isolation, and easier maintenance.

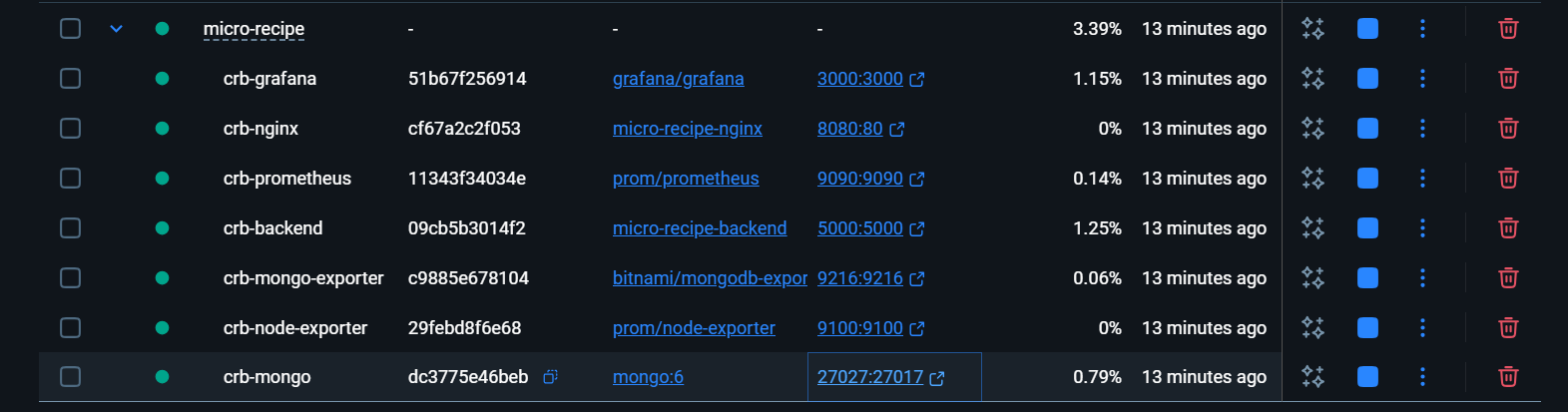
The system is containerized using Docker and orchestrated using **Docker Compose**.

**7.2. Microservices Architecture Overview**

In a microservices approach, the application is broken down into smaller, loosely coupled services. Each service:

* Runs in its own container
* Has a single responsibility
* Can be deployed and scaled independently

**Microservices Used**

Figure: Screenshot from Micro-Recipe application showing deployment and monitoring.

|  |  |
| --- | --- |
| **Service** | **Responsibility** |
| Frontend Service | User Interface (Community Recipe Board) |
| Backend API Service | Business logic & REST APIs |
| Database Service | Persistent data storage (MongoDB) |
| NGINX Gateway | Reverse proxy & request routing |
| Monitoring Service | Metrics collection (Prometheus) |
| Visualization Service | Metrics visualization (Grafana) |
| Node Exporter | Host-level system metrics |

**7.3. Logical Architecture Flow**

User

↓

Frontend Service (UI)

↓

NGINX API Gateway

↓

Backend API Service

↓

MongoDB Service

Monitoring services run independently and observe all services.

**7.4. Benefits of Microservices with Docker**

* **Loose Coupling** – Services can be updated independently
* **Scalability** – Individual services can be scaled
* **Fault Isolation** – Failure in one service doesn’t crash the system
* **Technology Flexibility** – Each service can use different tech stacks
* **Faster Deployment** – Smaller builds and quicker rollouts

**7.5. Microservice-wise Docker Configuration**

**7.5.1 Frontend Microservice**

**Responsibility:** Handles UI rendering and user interaction

FROM node:18-alpine AS build

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/dist /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

**7.5.2 Backend API Microservice**

**Responsibility:** Authentication, recipes, likes, comments, and saves

FROM node:18-alpine

WORKDIR /app

COPY package\*.json ./

RUN npm install --production

COPY . .

EXPOSE 5000

CMD ["node", "server.js"]

**7.5.3 Database Microservice (MongoDB)**

**Responsibility:** Stores users, recipes, comments, and likes

image: mongo:6

ports:

- "27017:27017"

volumes:

- mongo\_data:/data/db

**7.5.4 API Gateway Microservice (NGINX)**

**Responsibility:** Routes incoming requests to appropriate services

server {

listen 80;

location / {

proxy\_pass http://frontend;

}

location /api/ {

proxy\_pass http://backend:5000;

}

}

**7.5.5 Monitoring Microservices**

* **Prometheus** – Collects metrics
* **Node Exporter** – Exposes system metrics
* **Grafana** – Visualizes metrics

These services are independent and observe application health.

**7.6. Docker Compose – Microservices Orchestration**

services:

frontend:

build: ./frontend

container\_name: crb-frontend

backend:

build: ./backend

container\_name: crb-backend

environment:

- MONGO\_URL=mongodb://mongo:27017/recipes

mongo:

image: mongo:6

container\_name: crb-mongo

volumes:

- mongo\_data:/data/db

nginx:

image: nginx:alpine

container\_name: crb-nginx

ports:

- "8080:80"

volumes:

- ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf

depends\_on:

- frontend

- backend

prometheus:

image: prom/prometheus

container\_name: crb-prometheus

ports:

- "9090:9090"

node-exporter:

image: prom/node-exporter

**7.7. Deployment Steps**

**Step 1: Build and Deploy Microservices**

docker-compose up -d --build

**Step 2: Verify Services**

docker ps

Each microservice runs in an isolated container.

**7.8. Access Points**

|  |  |
| --- | --- |
| Service | URL |
| Application UI | http://localhost:8080 |
| Backend API | http://localhost:8080/api |
| Grafana | http://localhost:3000 |
| Prometheus | <http://localhost:9090> |



Fig: Frontend part running on crb-frontend in microservice

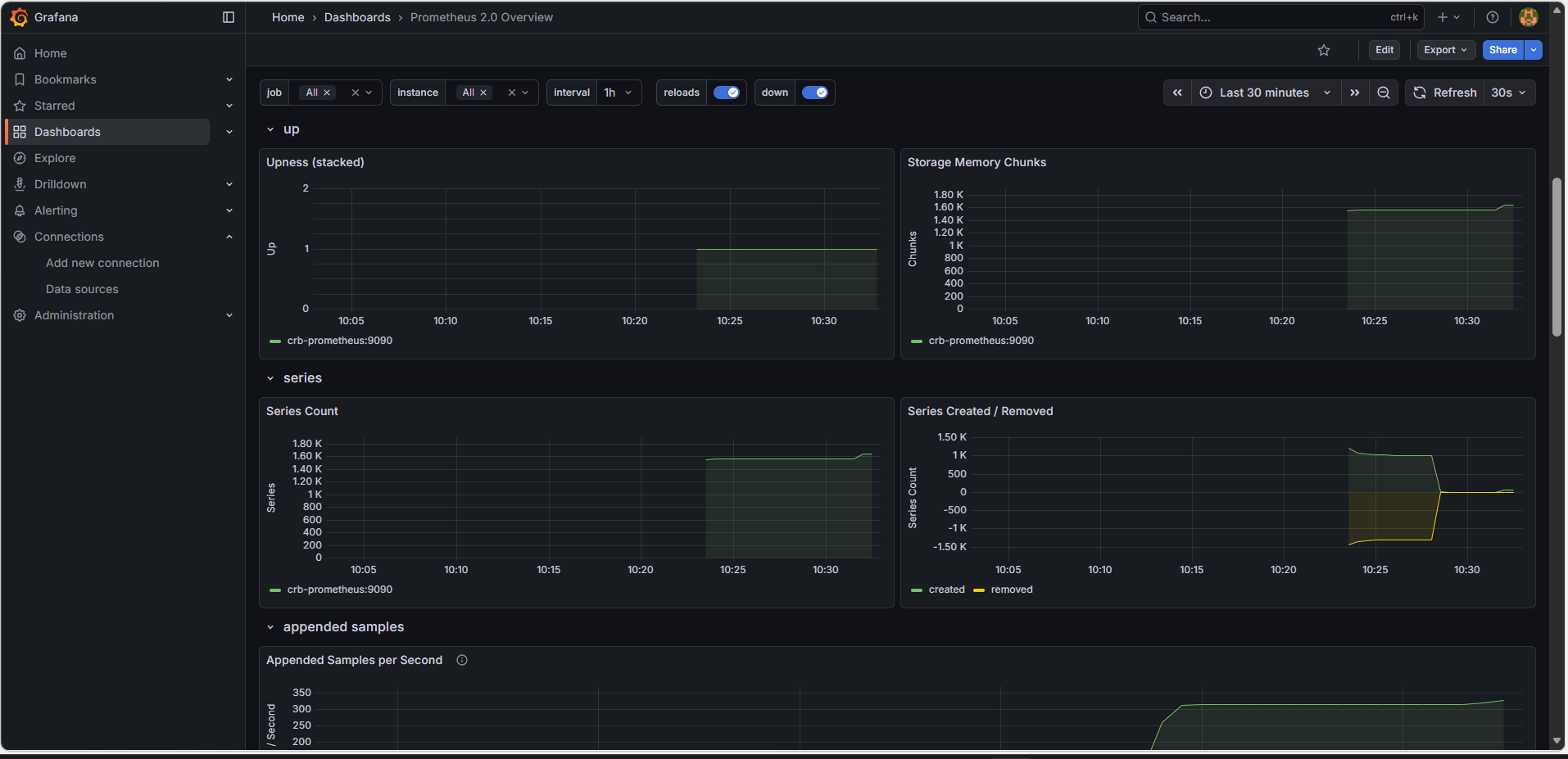


Fig:Monitering using Grafana

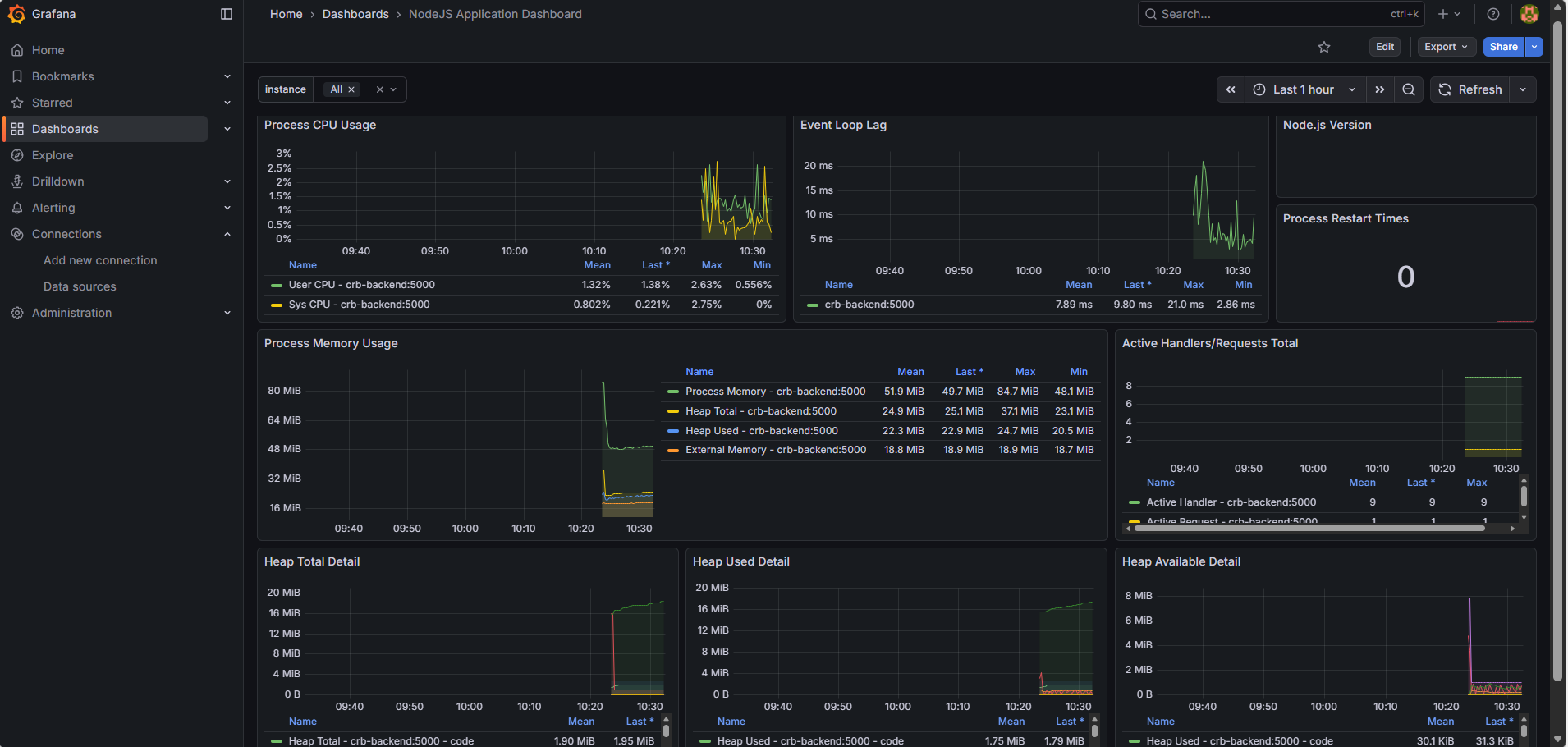


Fig:Node.js application dashboard monitoring in Grafana

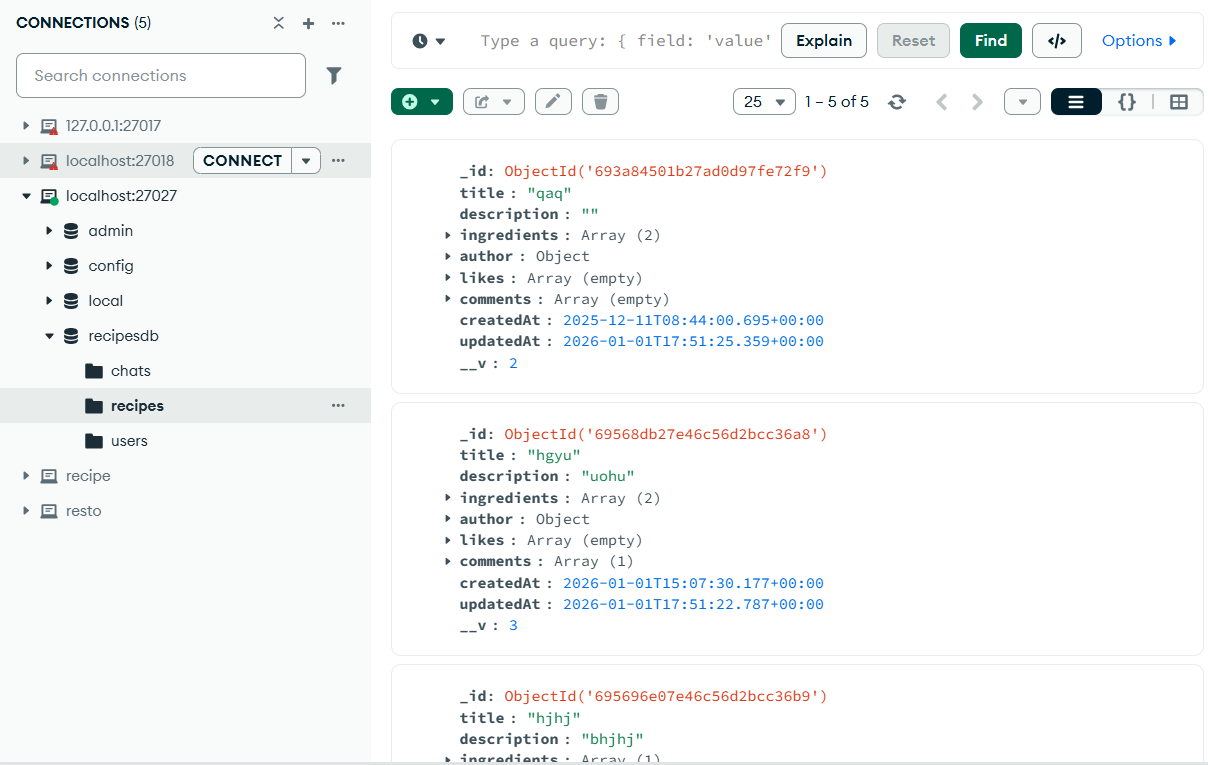


Figure: Screenshot from Micro-Recipe application showing deployment and monitoring.

**7.9. Conclusion**

Deploying the Micro-Recipe Application as Docker-based microservices improves scalability, resilience, and maintainability. Docker Compose provides lightweight orchestration, while Prometheus and Grafana deliver real-time observability, making the system production-ready.

**8.Ansible Automation Micro-Recipe Community Board**

**8.1. Objective**

The objective of this experiment is to automate the deployment, configuration, and monitoring of a Micro-Recipe Community Board application using Ansible and Docker. Ansible is used to ensure repeatable, consistent deployments with minimal manual effort.

**8.2. Technology Stack**

Frontend: HTML, CSS, JavaScript (Nginx)

Backend: Node.js, Express.js

Database: MongoDB

Containerization: Docker & Docker Compose

Monitoring: Prometheus, Grafana

Automation Tool: Ansible

Operating System: Ubuntu (WSL on Windows)

**8.3. Ansible Inventory**

[app]  
localhost ansible\_connection=local  
  
[micro\_recipe]  
localhost ansible\_connection=local

**8.4. Ansible Playbooks**

**8.4.1 main.yml**

This playbook ensures the application directory exists, copies application code, stops existing containers, and rebuilds and starts all services using Docker Compose.

---

- name: Ensure app directory exists

  file:

    path: /opt/app

    state: directory

- name: Copy application code

  synchronize:

    src: /mnt/d/micro-recipe/

    dest: /opt/app/

    delete: yes

- name: Stop existing containers

  shell: docker compose down || true

  args:

    chdir: /opt/app

- name: Build and start application

  shell: docker compose up -d --build

  args:

    chdir: /opt/app

  failed\_when: false

**8.4.2 deploy.yml**

This playbook deploys the application using Docker Compose by invoking the app role with privilege escalation.

---

- name: Deploy application using Docker Compose

  hosts: app

  become: true

  roles:

    - role: ../roles/app

**8.4.3 setup.yml**

This playbook prepares the server environment by installing Docker and required dependencies.

- name: Setup Micro-Recipe Server

  hosts: micro\_recipe

  become: true

  roles:

  - role: ../roles/docker

**8.5. Execution Command**

ansible-playbook -i inventory/hosts.ini playbooks/deploy.yml -K

**8.6. Results After Running Ansible Playbooks**

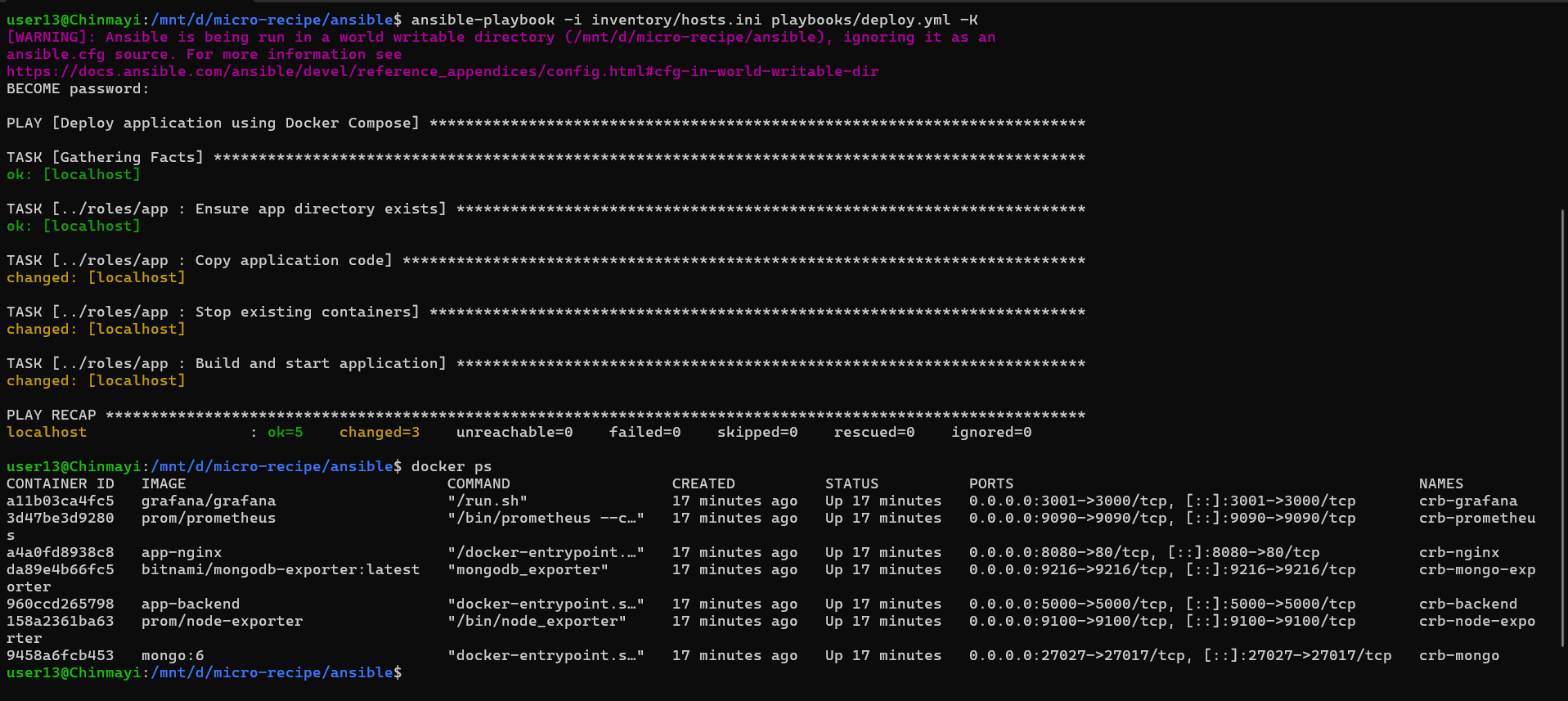


Fig 1)Docker containers running after Ansible deployment

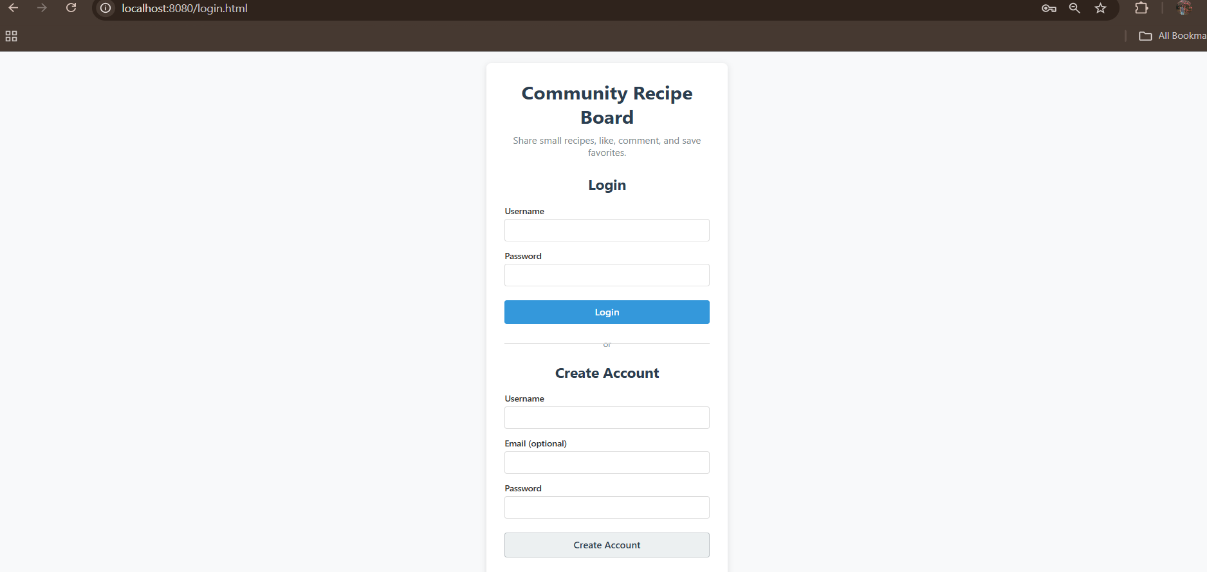


Fig 2)User login and account creation page

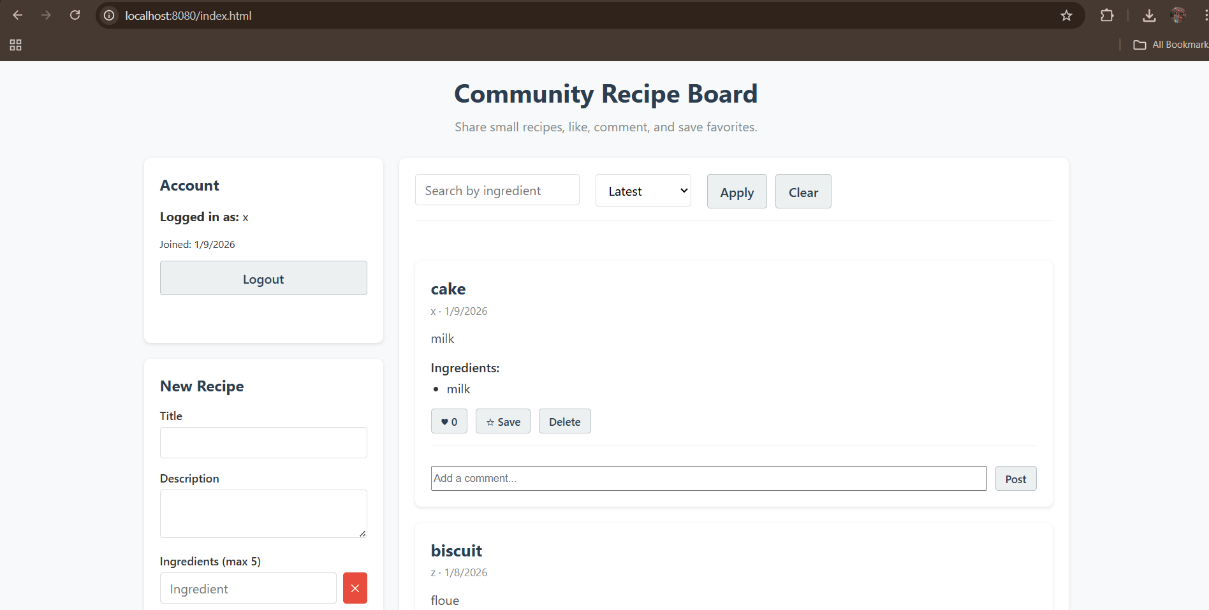


Fig 3)Frontend landing page – Community Recipe Board

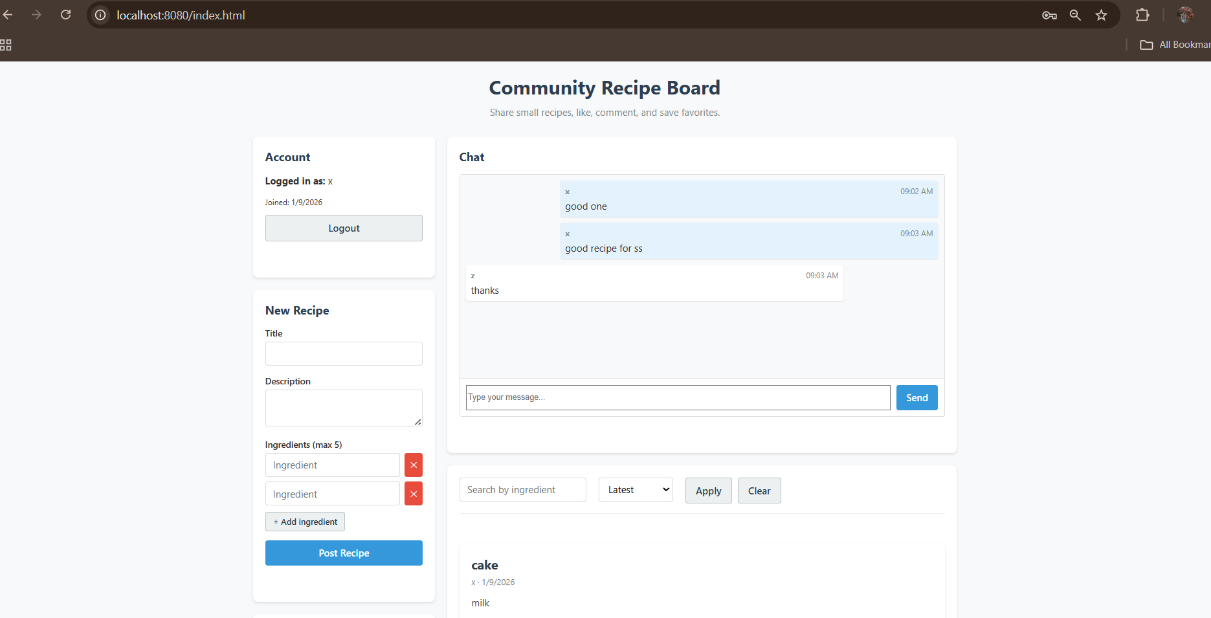


Fig 4)Chat and recipe interaction page

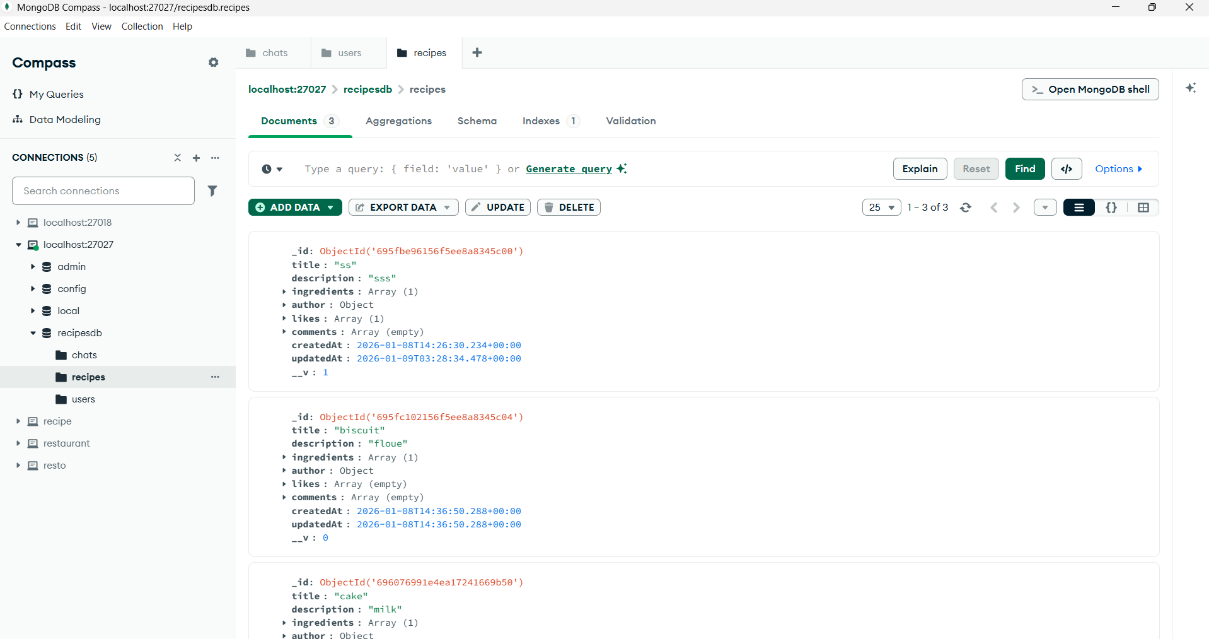


Fig 5)MongoDB Compass showing recipe data storage

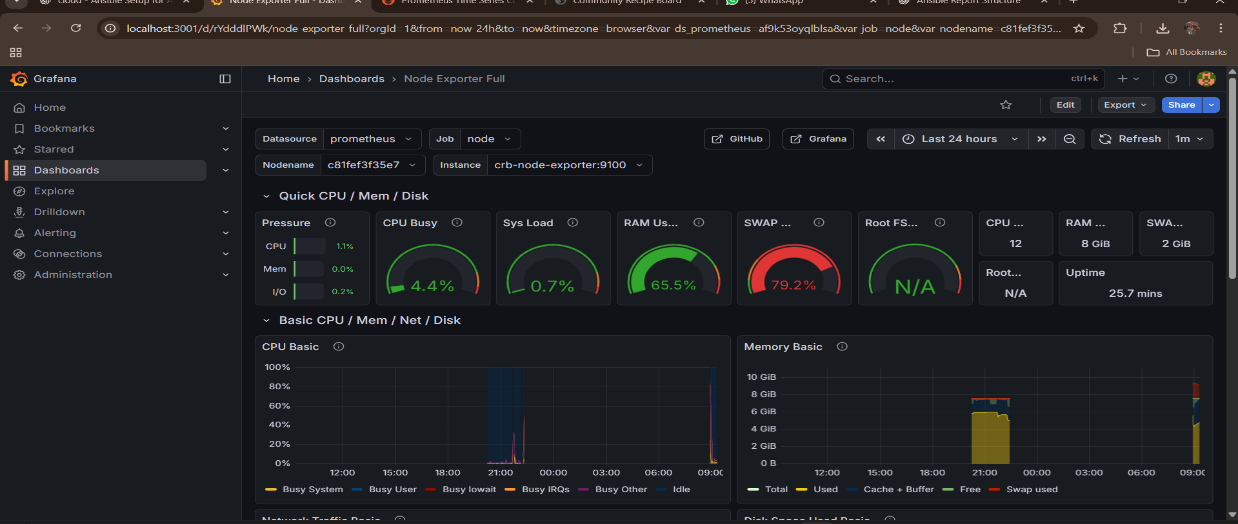
**8.7.Visualizing Performance Metrics via Grafana**

Fig 6)Grafana Node Exporter dashboard

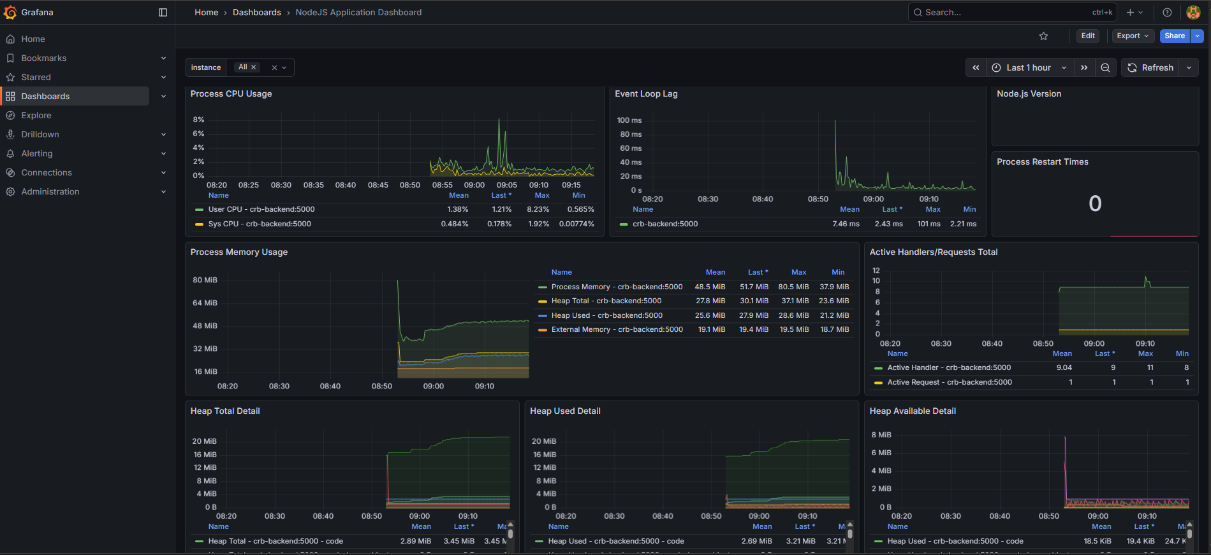


Fig 7)Grafana Node.js application dashboard



Fig 8)Prometheus monitoring dashboard



Fig 9) Prometheus 2.0 Overview Dashboard – Configuration Reload and Garbage Collection Metrics

Fig 10) Prometheus 2.0 Overview Dashboard – Scrape, Sync, and Target Monitoring

**8.8. Conclusion**

This experiment demonstrates how Ansible can be effectively used to automate deployment and monitoring of a containerized microservice application. The approach improves consistency, reduces manual errors, and simplifies application lifecycle management.

**9.CONCLUSION**

This project demonstrates two modern deployment approaches through the Micro-Recipe and Blood-Link applications, covering a wide spectrum of DevOps and cloud computing technologies.

The Micro-Recipe Application is deployed using Docker, microservices architecture, Ansible, and Kubernetes, representing an infrastructure-managed, cloud-native system. Docker ensures portability and consistency through containerization, while microservices enable independent development, scalability, and fault isolation. Ansible automates deployment and configuration, reducing manual effort and ensuring reliability. Kubernetes provides production-grade orchestration with features such as auto-scaling, self-healing, and high availability, making the application robust and scalable.

The Blood-Link Application, on the other hand, is deployed using Netlify (SaaS) for the frontend and Render (PaaS) for the backend. This platform-managed approach simplifies deployment by abstracting infrastructure concerns, enabling rapid development, easy scaling, and minimal operational overhead.

Together, these deployments highlight the contrast between full infrastructure control and platform simplicity. The project effectively showcases how different deployment models can be chosen based on application requirements, scalability needs, and operational complexity, providing strong practical exposure to modern DevOps and cloud-native practices.