**Set up a comprehensive CI/CD Pipeline**

1. Our backend is nHost (Hasura GraphQL engine + PostgreSQL), 2 Python microservices, and a NextJS server serving up Server-Side Rendering.
2. Set up a comprehensive CI/CD Pipeline with Gitpod with Cursor, GitHub Actions, AI, and Security.
3. Save all code to GitHub repo during development and configuration.
4. Provide complete documentation and appropriate comments.
5. Set up 2 FA (Biometric WebAuthn and Google Authenticator code) access to GitPod and all other resources mentioned here.

Tools to be integrated:

1. GitPod, GitHub Actions, AI, and Security.
2. Selenium, pytest, Playwright, Jest, Mocha, Appium
3. OSV-Scanner, SonarCube, TruffleHog, and Snyk
4. ESLint for JavaScript, Pylint for Python, TSLint for TypeScript
5. Prettier for JavaScript, Black for Python
6. Coverage.py for Python, Jest for JavaScript
7. OpenZiti Zero Trust Networking
8. Smoke tests, performance tests
9. Graylog, Prometheus with Grafana, Zabbix, Chart.js or D3.js
10. Ansible or Terraform,
11. Vultr.com hosting

Sequence of CI/CD Pipeline Processes

**Gitpod Workspace Configuration**

* Install Essential Extensions:
  + Remote - SSH: For seamless remote machine connections.
  + GitLens: For advanced Git features and insights.
  + Cursor: For intelligent code completion and suggestions.
  + Other relevant extensions: Tailor to your specific project needs.
* Configure Cursor:
  + Authenticate your Cursor account.
  + Customize settings to optimize the AI assistant's behavior for your coding style and preferences.
* Build and Test:
  + Checks out the code.
  + Sets up a Gitpod workspace.
    - Use the gitpod-io/action to trigger Gitpod workspaces.
    - Configure Gitpod to automatically open when a pull request is created or a new branch is pushed.
    - Clone your repository into the Gitpod workspace.

**GitHub Actions Workflow**

Workflow Trigger:

* Trigger the workflow on pushes to the main branch or pull requests.

1. Code Quality Checks:
   1. Automatically trigger builds and run full tests (Selenium, pytest, Playwright, Jest, Mocha, Appium).
2. Security and Static Code Analysis:
   1. Integrate with OSV-Scanner, SonarCube, TruffleHog, and Snyk to perform security scans and static code analysis.
3. Linting, Formatting, and Code Coverage:
   1. Perform automatic linting (ESLint for JavaScript, Pylint for Python, TSLint for TypeScript).
   2. Apply formatting (Prettier for JavaScript, Black for Python).
   3. Analyze code coverage (Coverage.py for Python, Jest for JavaScript).
4. OpenZiti Zero Trust Networking:
   1. Install OpenZiti CLI:
      1. Action: Add the OpenZiti CLI to the CI/CD pipeline environment.
      2. Purpose: Ensure the CLI is available for executing Ziti commands within the pipeline.
   2. Configure Ziti Controller and Edge Router:
      1. Action: Set up the Ziti Controller and Edge Router as part of the infrastructure.
      2. Purpose: Establish the core components required for OpenZiti’s zero trust networking.
   3. Create and Manage Ziti Identities:
      1. Action: Automate the creation and management of Ziti identities for secure access.
      2. Purpose: Ensure that each application and user has a unique, verifiable identity for secure access.
   4. Embed Ziti SDKs:
      1. Action: Verify that the necessary Ziti SDKs are included as part of code quality checks. Alert developers if missing.
      2. Purpose: Ensure the application code is properly integrating Ziti SDKs for zero trust networking.
      3. Note: Initial integration of SDKs is done by developers during development. The CI/CD pipeline checks for their presence.
   5. Deploy Tunneling Applications:
      1. Action: Automate the deployment of Ziti tunneling applications based on predefined configurations (see code examples below). Alert developers if not deployed.
      2. Purpose: Secure communications for applications that cannot be directly modified by creating secure tunnels.
   6. Define and Apply Policies:
      1. Action: Automate the definition and application of access policies using OpenZiti CLI or APIs.
      2. Purpose: Ensure that access controls are consistently applied and managed across the infrastructure.
5. Testing in Staging Environment:
   1. Deploy to staging environment.
   2. Run additional tests (smoke tests, performance tests).
6. Monitoring and Alerting: Make this a separate standalone module that can also be integrated into other applications. See notes below.
   1. Instrumentation setup into the CI/CD pipeline to automate the deployment of monitoring configurations to implement a robust monitoring and alerting system to track the health of the applications.
   2. Set up Graylog (Dependencies: MongoDB for data storage. Elasticsearch or Logstash for log data indexing and searching) for logging,
   3. Prometheus with Grafana for monitoring,
   4. Zabbix for real-time monitoring, user activity tracking and visualization of data, and integrate notifications and alerts.
   5. Use Chart.js or D3.js for graphical display and monitoring.
7. Deployment:
   1. Use Ansible, Terraform, or Kubernetes to automate the deployment to production.
8. Deploy to Vultr.com hosting.

**One-Time Setup When Setting Up the Repository**

1. **Include Ziti Configuration in Repository**:
   * **Objective**: Ensure that the OpenZiti configuration is versioned and accessible.
   * **Action**: Add configuration files (e.g., ziti-config.json) to the repository.
   * **Example**: Storing OpenZiti configuration alongside application configuration files.

{

"controller": "https://ziti-controller.example.com",

"edgeRouter": "https://ziti-edge-router.example.com",

"identities": [

{

"name": "my-device",

"jwtFile": "./my-device.jwt"

}

]

}

1. **Include Monitoring Configuration in Repository**:
   * **Objective**: Ensure that monitoring configuration is versioned and accessible.
   * **Action**: Add monitoring configuration files (e.g., prometheus.yml, grafana-dashboard.json) to the repository.
   * **Example**: Storing Prometheus and Grafana configuration alongside application configuration files.
   * Example prometheus.yml:

yaml

scrape\_configs:

- job\_name: "my-app"

scrape\_interval: 15s

metrics\_path: "/metrics"

static\_configs:

- targets: ["localhost:3000"]

**Crafting a Comprehensive CI/CD Pipeline with Gitpod, AI, and Security**

1. Gitpod Workspace Configuration

* Install Essential Extensions:
  + Remote - SSH: For seamless remote machine connections.
  + GitLens: For advanced Git features and insights.
  + Cursor: For intelligent code completion and suggestions.
  + Other relevant extensions: Tailor to your specific project needs.
* Configure Cursor:
  + Authenticate your Cursor account.
  + Customize settings to optimize the AI assistant's behavior for your coding style and preferences.

2. GitHub Actions Workflow

Workflow Trigger:

* Trigger the workflow on pushes to the main branch or pull requests.

Explanation of the Workflow:

1. Build and Test:
   * Checks out the code. Automatically trigger builds and run full tests.
   * Sets up a Gitpod workspace.
     + Use the gitpod-io/action to trigger Gitpod workspaces.
     + Configure Gitpod to automatically open when a pull request is created or a new branch is pushed.
     + Clone your repository into the Gitpod workspace.
   * Sets up the Python environment.
   * Installs dependencies.
2. Run tests: Execute unit, integration, and end-to-end tests using tools like Jest, Mocha, JUnit, Pytest, Playwright, and Selenium.
3. Scans for vulnerabilities, code analysis and security scanning using Snyk, OSV-Scanner, SonarQube, and TruffleHog.
4. Add linting, formatting, and code coverage analysis.
5. OpenZiti Zero Trust Networking: Install CLI, configure components, manage identities, verify SDKs, deploy tunneling applications, and apply policies.
6. Deploy to Staging:
   * Deploys the code to the staging environment.
   * Run tests: Run additional tests in the staging environment (e.g., smoke tests, performance tests).
7. Monitoring and Alerting: Deploy monitoring configurations and set up tools (standalone module).
8. Deploy to Production:
   * Manual approval: Require manual approval before deploying to production.
   * Deploys the code to the production environment.

a. Create a .github/workflows/ci.yml file:

YAML

name: CI/CD Pipeline with OpenZiti

on: [push, pull\_request]

jobs:

# Step 1: Code Quality Checks

code-quality-checks:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set Up Gitpod

uses: gitpod-io/action@v2

with:

gitpod-url: https://gitpod.io/#https://github.com/your-username/your-repo

- name: Set Up Python

uses: actions/setup-python@v4

with:

python-version: '3.11'

- name: Install Dependencies

run: |

pip install -r requirements.txt

- name: Run Unit Tests (pytest)

run: pytest

- name: Run Integration Tests (Playwright)

run: playwright test

- name: Run Selenium Tests

run: selenium\_test\_runner.py

- name: Run JavaScript Tests (Jest, Mocha)

run: npm test

# Step 2: Security and Static Code Analysis

security-code-analysis:

runs-on: ubuntu-latest

needs: code-quality-checks

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Code Analysis with SonarQube

uses: SonarSource/sonar-scanner-action@latest

with:

sonar.organization: your-organization

sonar.projectKey: your-project

sonar.host.url: https://sonarcloud.io

sonar.login: your-token

- name: Security Scanning with Snyk

uses: snyk/actions/upgrade-lockfile@v1

with:

orgToken: your-snyk-token

- name: Secret Scanning with TruffleHog

uses: truffleHog/truffleHog-action@v1

with:

search\_path: .

exclude: .git,node\_modules

- name: Vulnerability Scanning with OSV-Scanner

uses: osv-scanner/action-scanner@v1

with:

token: ${{ secrets.OSV\_TOKEN }}

# Step 3: Linting, Formatting, and Code Coverage

linting-formatting-coverage:

runs-on: ubuntu-latest

needs: security-code-analysis

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Lint JavaScript with ESLint

run: npm run lint

- name: Lint Python with Pylint

run: pylint \*\*/\*.py

- name: Format JavaScript with Prettier

run: npm run format

- name: Format Python with Black

run: black .

- name: Code Coverage Analysis with Coverage.py

run: coverage run -m pytest && coverage report

- name: Code Coverage Analysis with Jest

run: npm run coverage

# Step 4: OpenZiti Zero Trust Networking

openziti-integration:

runs-on: ubuntu-latest

needs: linting-formatting-coverage

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Install OpenZiti CLI

run: |

curl -s https://raw.githubusercontent.com/openziti/ziti/release-next/quickstart/docker-compose/get-ziti.sh | sudo bash

- name: Configure Ziti Controller and Edge Router

run: |

ziti controller start

ziti edge router start

- name: Create and Manage Ziti Identities

run: |

ziti edge create identity device my-device

ziti edge create enrollment -o my-device.jwt --identity my-device

- name: Verify Ziti SDKs

run: |

python3 -c "import ziti; print('Ziti SDK is present!')" || echo "Ziti SDK is missing" && exit 1

- name: Deploy Tunneling Applications

run: |

ziti edge deploy --config ziti-config.yaml || echo "Tunneling application deployment failed" && exit 1

- name: Define and Apply Policies

run: |

ziti edge create policy EdgeRouterPolicy edge-router-policy --service-roles '#' --edge-router-roles '#'

ziti edge create policy ServicePolicy service-policy --service-roles '#' --identity-roles '#'

# Step 5: Testing in Staging Environment

testing-staging:

runs-on: ubuntu-latest

needs: openziti-integration

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set Up Python

uses: actions/setup-python@v4

with:

python-version: '3.11'

- name: Install Dependencies

run: |

pip install -r requirements.txt

- name: Deploy to Staging

run: ansible-playbook deploy-staging.yml

- name: Run Smoke Tests

run: npm run smoke-test

- name: Run Performance Tests

run: npm run performance-test

# Step 6: Monitoring and Alerting (Standalone Module)

monitoring-alerting:

runs-on: ubuntu-latest

needs: testing-staging

steps:

- name: Deploy Monitoring Module

run: |

ansible-playbook deploy-monitoring.yml

terraform apply

- name: Set up Graylog

run: |

docker-compose up -d graylog

- name: Set up Prometheus and Grafana

run: |

docker-compose up -d prometheus grafana

- name: Set up Zabbix

run: |

docker-compose up -d zabbix

# Step 7: Deployment to Production

deploy-production:

runs-on: ubuntu-latest

needs: monitoring-alerting

steps:

- name: Manual Approval

uses: actions/github-script@v4

with:

script: |

const {Octokit} = require("@octokit/rest");

const github = new Octokit({auth: process.env.GITHUB\_TOKEN});

await github.actions.createWorkflowDispatch({

owner: context.repo.owner,

repo: context.repo.repo,

workflow\_id: 'ci.yml',

ref: context.ref,

inputs: {

approve: 'true'

}

});

- name: Checkout code

uses: actions/checkout@v3

- name: Set Up Python

uses: actions/setup-python@v4

with:

python-version: '3.11'

- name: Install Dependencies

run: |

pip install -r requirements.txt

- name: Deploy to Production

run: ansible-playbook deploy-production.yml

b. Configure Secrets: Protect sensitive information - API keys and tokens - using GitHub Secrets.

- SonarQube Token: Obtain a token from SonarQube and add it as a secret in your GitHub repository.

- Snyk Token: Obtain a token from Snyk and add it as a secret in your GitHub repository.

- OSV-Scanner Token: Obtain a token from OSV-Scanner and add it as a secret in your GitHub repository.

3. Deployment to Staging and Production

- Leverage Deployment Tools - Ansible, Terraform, or Kubernetes - to automate the deployment process.

- Integrate with Deployment Platform: Vultr.com

- Trigger Deployments from GitHub Actions using the deployment job.

4. Additional information

* Optimize the Workflow:
  + Employ caching, parallel execution to run tests concurrently, and other techniques to accelerate build times.
  + Use container registry like Docker Hub or GitHub Container Registry to store the container images.
  + Implement a robust monitoring and alerting system to track the health of the applications.
  + Configure the project's environment variables (e.g., API keys, database credentials).
* Continuous Improvement:
  + Regularly review and refine the CI/CD pipeline to enhance its efficiency and effectiveness.

**Manual Approval for Production Deployment**

To ensure deployments to production require manual approval, we can use the workflow\_run event with workflow\_dispatch for manual triggers.

**Using CI/CD Pipelines for Instrumentation**

**Purpose**:

* CI/CD integration ensures that the setup, deployment, and configuration of monitoring tools happen automatically, reducing manual effort and ensuring consistency across environments. Whenever developers develop APIs or integrate functions, the instrumentation is already in place.

**When to Use**:

* **Automation**: Ideal for automating the instrumentation setup across all environments.
* **Consistency**: Ensures that every deployment has consistent monitoring and logging configurations.
* **Scalability**: Suitable for large teams and projects where manual setup would be tedious.

**How It Works**:

* The CI/CD pipeline automatically installs the necessary libraries and inserts instrumentation code during the build/deployment process.
* Developers write their code, and when they push changes, the CI/CD system ensures the instrumentation is in place before deploying the application.

**Install Graylog**:

* **Docker Installation**: This is the simplest method.

sh

docker run --name mongo -d mongo:3

docker run --name elasticsearch -p 9200:9200 -p 9300:9300 \

-e ES\_JAVA\_OPTS="Xms2g -Xmx4g" -e discovery.type=single-node \

-e xpack.security.enabled=false -e bootstrap.memory\_lock=true \

-d docker.elastic.co/elasticsearch/elasticsearch:5.6.11

docker run --name graylog --link mongo --link elasticsearch \

-p 9000:9000 -p 12201:12201 -p 514:514 -p 5555:5555 \

-e GRAYLOG\_WEB\_ENDPOINT\_URI="http://127.0.0.1:9000/api" \

-d graylog/graylog:2.4.6-1

* **Access Graylog**: Open your browser and go to http://localhost:9000/. The default username and password are both admin.

**Configure Graylog Inputs**:

* Go to **System** > **Inputs** and launch a new input.
* Select **Syslog UDP** and configure it to collect logs from your applications.

**Setting Up Prometheus with Grafana for Monitoring**

**Install Prometheus and Node Exporter**:

* **Node Exporter**: Collects hardware and OS-level metrics.

sh

docker run -d --name node-exporter -p 9100:9100 prom/node-exporter

* **Prometheus**: The monitoring system.

sh

docker run -d --name prometheus -p 9090:9090 \

-v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml \

prom/prometheus

**Configure Prometheus**:

* Create a prometheus.yml file with the following content:

yaml

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: 'node'

static\_configs:

- targets: ['localhost:9100']

* Place this file in the same directory as your Prometheus container.

**Install and Configure Grafana**:

* **Docker Installation**:

sh

docker run -d --name grafana -p 3000:3000 grafana/grafana

* + **Access Grafana**: Open your browser and go to http://localhost:3000/. Log in with the default credentials (username: admin, password: admin).
* **Add Prometheus as a Data Source in Grafana**:
  + Go to **Configuration** > **Data Sources** and click **Add data source**.
  + Select **Prometheus** and set the URL to http://prometheus:9090.
  + Click **Save & Test** to verify the connection.
* **Create Dashboards**:
  + Import pre-made dashboards or create your own to visualize the metrics collected by Prometheus.

**Suitable Standard Libraries**

**For Node.js (JavaScript/TypeScript)**:

* **Prometheus Client**: prom-client for metrics.
* **OpenTelemetry**: @opentelemetry/api, @opentelemetry/node, @opentelemetry/tracing for distributed tracing.
* **APM Tools**: New Relic, Datadog APM, or Sentry for application performance monitoring.

**For Python**:

* **Prometheus Client**: prometheus\_client for metrics.
* **OpenTelemetry**: opentelemetry-api, opentelemetry-sdk, opentelemetry-instrumentation for distributed tracing.
* **APM Tools**: New Relic, Datadog APM, or Sentry for application performance monitoring.

**Example**: In a GitHub Actions CI/CD pipeline, you might have steps to install monitoring libraries and run scripts to add instrumentation:

yaml

- name: Add Instrumentation

run: |

echo "Adding OpenTelemetry Instrumentation"

npm install @opentelemetry/api @opentelemetry/node @opentelemetry/tracing prom-client

# Insert additional instrumentation code here if needed

**4. Store and Monitor the Data**

Use Prometheus to scrape and store the metrics, and visualize them in Grafana. For comprehensive analysis, you can also store the logs in a centralized logging system like Elasticsearch or Logstash.

**Example Prometheus Metric Collection**:

yaml

- job\_name: "apisix"

scrape\_interval: 15s

metrics\_path: "/apisix/prometheus/metrics"

static\_configs:

- targets: ["apisix-quickstart:9091"]

**Correlate Data for Analysis**

To correlate the data and identify problematic services and accounts:

1. **Set Up Dashboards**: Create Grafana dashboards to visualize the metrics and logs. Use the transaction UUID and user/client information to filter and analyze the data.
2. **Analyze Logs**: Use a centralized logging system to analyze the logs. Search for specific transaction UUIDs to trace the full path of a transaction.
3. **Alerting**: Set up alerts in Grafana or Zabbix to notify you of issues, such as high error rates or slow response times.

**Example Grafana Dashboard Configuration**

Create a Grafana dashboard to display metrics like:

* **API Latency**: Time taken for each API call.
* **Error Rates**: Percentage of failed transactions.
* **Retries**: Number of retries for each transaction.
* **User Activity**: Number of transactions per user/client.

**Example Alert Rule**

Set up an alert rule in Grafana for high error rates:

yaml

alert:

name: HighErrorRate

condition: avg(5m, rate(http\_request\_total{status!~"2.."}[1m])) > 0.05

for: 5m

labels:

severity: "critical"

annotations:

summary: "High Error Rate"

description: "The error rate has been above 5% for the past 5 minutes."

**Integration with Vultr**

1. **Checkmk**: Specifically recommended by Vultr for monitoring servers in a Virtual Private Cloud (VPC). [How to Monitor Servers in a Vultr Virtual Private Cloud (VPC) with Checkmk | Vultr Docs](https://docs.vultr.com/how-to-monitor-servers-in-a-vultr-virtual-private-cloud-vpc-with-checkmk)
2. **Glances**: A lightweight monitoring tool that provides real-time system monitoring and can be used directly on Vultr servers. [Monitoring Your Server With Glances | Vultr Docs](https://docs.vultr.com/monitoring-your-server-with-glances)

Build a robust **module for integrating logging, monitoring, and notification systems with external and internal APIs in the nHost environment**.

Detailed plan and example code to help achieve this:

**1. Set Up Your Environment**

* Ensure you have an nHost project set up.
* Install necessary packages for logging, monitoring, and notifications.

**2. Database Schema for Logging Requests**

sql

-- Create a table to log API requests

CREATE TABLE api\_requests (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

user\_id UUID REFERENCES users(id),

endpoint TEXT NOT NULL,

request\_time TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

response\_time TIMESTAMP WITH TIME ZONE,

status TEXT,

retries INTEGER DEFAULT 0,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP

);

-- Create a table for alerts

CREATE TABLE api\_alerts (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

request\_id UUID REFERENCES api\_requests(id),

alert\_type TEXT NOT NULL,

message TEXT,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP

);

**3. Module to Log API Requests and Responses**

* **Utility Functions**

javascript

// utils/apiUtils.js

import fetch from 'node-fetch';

import { nhost } from '../utils/nhost';

export const logApiRequest = async (userId, endpoint) => {

const { id: requestId } = await nhost.graphql.request(`

mutation ($userId: uuid!, $endpoint: String!) {

insert\_api\_requests\_one(object: {user\_id: $userId, endpoint: $endpoint}) {

id

}

}

`, { userId, endpoint });

return requestId;

};

export const logApiResponse = async (requestId, status, retries) => {

await nhost.graphql.request(`

mutation ($requestId: uuid!, $status: String!, $retries: Int!) {

update\_api\_requests\_by\_pk(pk\_columns: {id: $requestId}, \_set: {status: $status, retries: $retries, response\_time: "now()"}) {

id

}

}

`, { requestId, status, retries });

};

export const sendAlert = async (requestId, alertType, message) => {

await nhost.graphql.request(`

mutation ($requestId: uuid!, $alertType: String!, $message: String!) {

insert\_api\_alerts\_one(object: {request\_id: $requestId, alert\_type: $alertType, message: $message}) {

id

}

}

`, { requestId, alertType, message });

};

* **API Request Module**

javascript

// modules/apiRequestHandler.js

import { logApiRequest, logApiResponse, sendAlert } from '../utils/apiUtils';

const requestWithRetries = async (userId, endpoint, retries = 3) => {

const requestId = await logApiRequest(userId, endpoint);

let attempts = 0;

let response;

while (attempts < retries) {

attempts += 1;

try {

response = await fetch(endpoint);

if (response.ok) {

await logApiResponse(requestId, 'success', attempts);

return await response.json();

} else {

await logApiResponse(requestId, 'fail', attempts);

await sendAlert(requestId, 'error', `Request to ${endpoint} failed with status ${response.status}`);

}

} catch (error) {

await logApiResponse(requestId, 'error', attempts);

await sendAlert(requestId, 'error', `Request to ${endpoint} failed with error ${error.message}`);

}

}

throw new Error(`Failed to fetch from ${endpoint} after ${retries} attempts`);

};

export default requestWithRetries;

**4. Notifications and Alerts**

Use a service like Twilio or SendGrid to send notifications. Here’s an example using SendGrid:

javascript

// utils/notificationService.js

import sgMail from '@sendgrid/mail';

sgMail.setApiKey(process.env.SENDGRID\_API\_KEY);

export const sendNotification = async (email, subject, text) => {

const msg = {

to: email,

from: 'your-email@example.com',

subject,

text,

};

await sgMail.send(msg);

};

**5. Graphical Display and Monitoring**

Use a frontend library like Chart.js or D3.js to display the logs and alerts.

javascript

// components/ApiStatsChart.js

import React, { useEffect, useState } from 'react';

import { Line } from 'react-chartjs-2';

import { nhost } from '../utils/nhost';

const ApiStatsChart = () => {

const [data, setData] = useState([]);

useEffect(() => {

const fetchData = async () => {

const result = await nhost.graphql.request(`

query {

api\_requests(order\_by: {created\_at: desc}, limit: 50) {

request\_time

response\_time

status

}

}

`);

setData(result.data.api\_requests);

};

fetchData();

}, []);

const chartData = {

labels: data.map((req) => req.request\_time),

datasets: [

{

label: 'Response Time',

data: data.map((req) => new Date(req.response\_time).getTime() - new Date(req.request\_time).getTime()),

borderColor: 'rgba(75, 192, 192, 1)',

backgroundColor: 'rgba(75, 192, 192, 0.2)',

fill: false,

tension: 0.1,

},

],

};

return <Line data={chartData} />;

};

export default ApiStatsChart;

**6. Highlight Abnormal Behaviors**

Add logic to highlight abnormal behaviors based on thresholds.

javascript

// utils/anomalyDetection.js

export const detectAnomalies = (requests) => {

const threshold = 2000; // 2 seconds

return requests.filter(req => (new Date(req.response\_time).getTime() - new Date(req.request\_time).getTime()) > threshold);

};