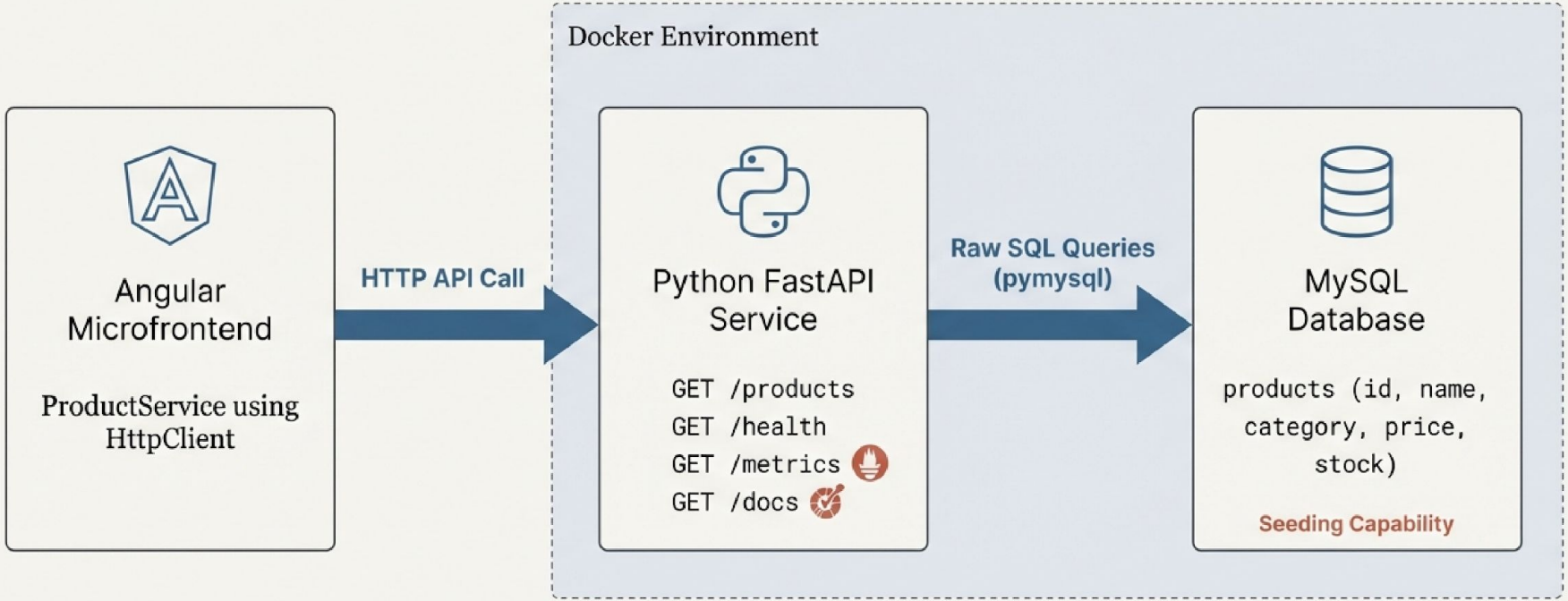
FINAL PROJECT

INFO 8985 - Monitoring and Logging

Nikhil Shankar Chirakkal Sivasankaran - 9026254
Deepak Tamizhalagan - 8983627
Cibi Sharan Cholarani - 9015927
Zafar Ahmed Shaik - 9027671
Richard Andrey Biscazzi - 8903530

Full-Stack Microservice Architecture: From UI to Database

A containerized, end-to-end data flow powering a dynamic product catalog.



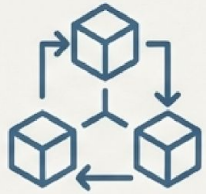
A complete, decoupled system where a Python API serves data from a relational database to an Angular frontend, all orchestrated with Docker.

Outcomes: A Scalable & Developer-Friendly Foundation

The integrated architecture delivers immediate value and is built for future expansion.

Dynamic Product Catalog

Successfully replaced a static, hardcoded product list with a dynamic system powered by a relational MySQL database. The complete data flow from database to UI is seamless and robust.

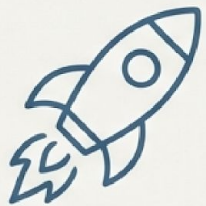


Scalable Microservices

A true microservices architecture with proper separation of concerns between the frontend (Angular), backend (FastAPI), and database. Services are independently containerized and scalable.

Built-in Observability

The API exposes a `/metrics` endpoint, providing out-of-the-box integration with monitoring tools like Prometheus for health checks and performance tracking.



Accelerated Development

With automated database seeding for easy setup and self-generating API documentation at the `/docs` endpoint, the system is designed for developer efficiency and rapid onboarding.

This project establishes a resilient foundation for all future CRUD (Create, Read, Update, Delete) operations and long-term data persistence.

Technology Stack: Deliberate Choices for Performance

Utilizing lightweight, modern tools to build an efficient and maintainable system.

Backend & Database

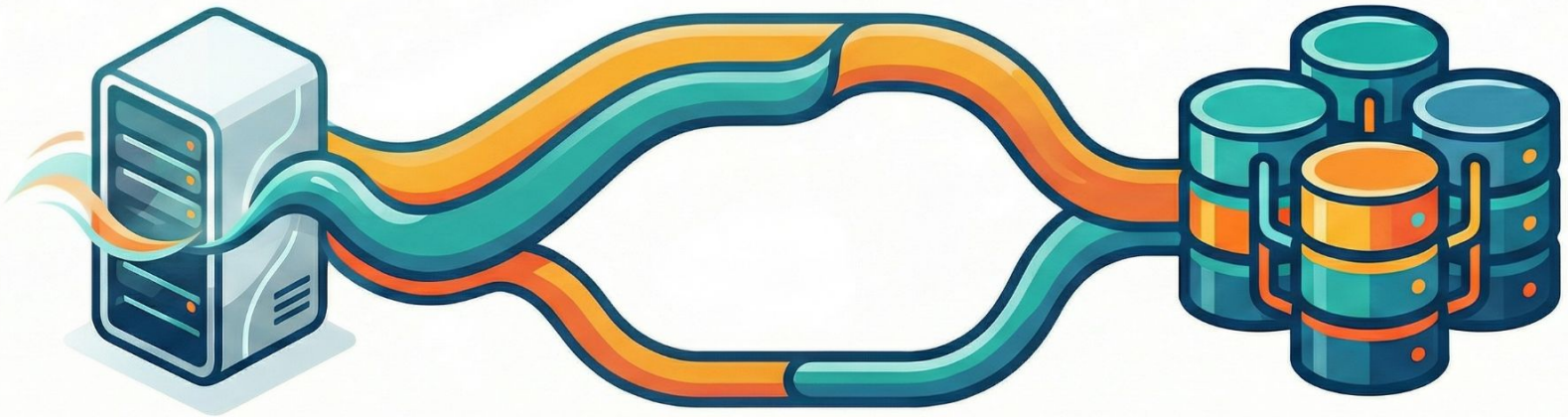
- **FastAPI Framework:** For high-performance, asynchronous API development and automatic OpenAPI documentation.
- **Pymysql Library:** Chosen for its lightweight, dependency-free nature for direct, efficient raw SQL operations.
- **RESTful API Design:** Clean, intuitive endpoints for product retrieval and standardized health checks.
- **Relational Schema:** A structured `products` table designed for data integrity and future query complexity.

Frontend & DevOps

- **Angular Microfrontend:** A decoupled client application for a clean separation from the backend logic.
- **HttpClient Module:** The standard, robust method for consuming RESTful services in Angular.
- **Docker & Docker Compose:** Containerization of both the API and MySQL services, ensuring consistent environments and simple orchestration.
- **Database Health Checks:** Integrated into the Docker setup to ensure service dependencies are managed correctly on startup.

Every component was selected to optimize for performance, scalability, and an excellent developer experience.

Backend Architecture Flow



1. FastAPI Service (Port 5000)

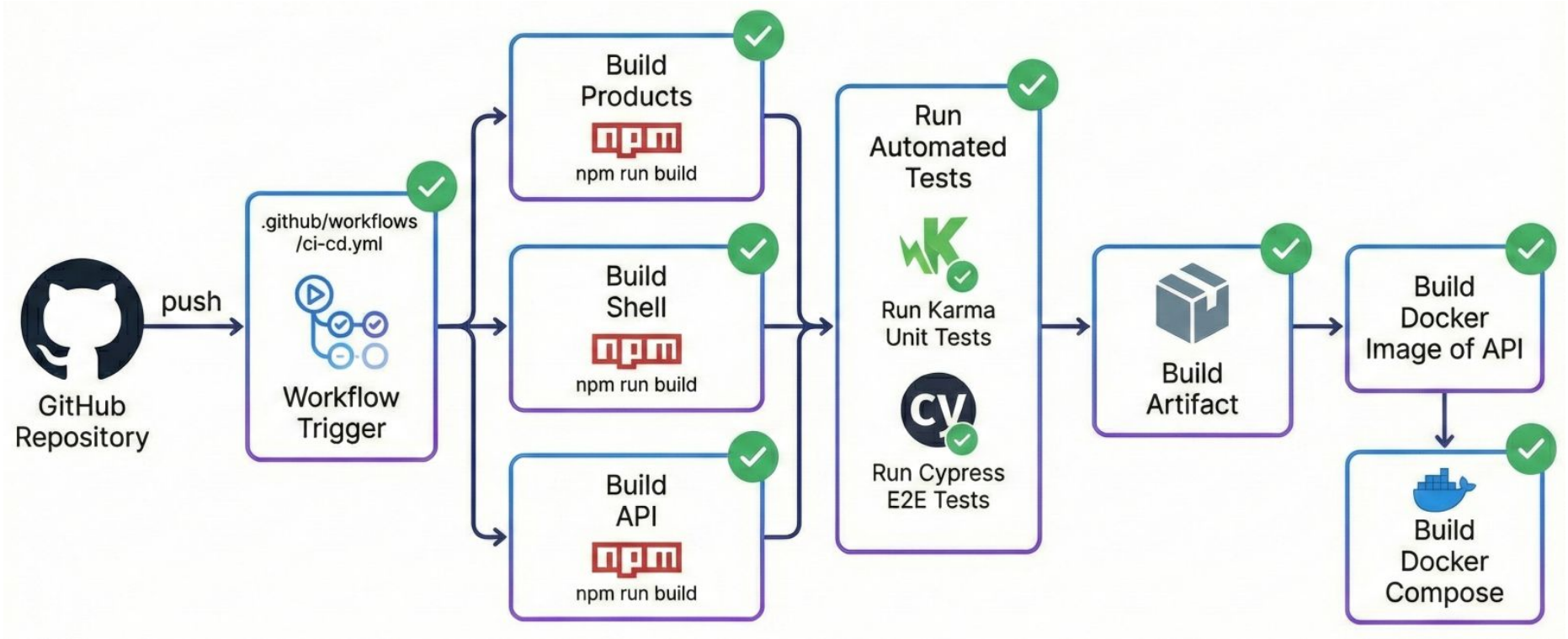
Provides a RESTful API with an /api/products endpoint for clients.

2. Direct Database Communication

The service queries the database directly using raw SQL commands.

3. MySQL Database (Port 8877)

Stores all product data, including ID, name, category, and price.



CI/CD Pipeline with GitHub Actions

- Automated workflow to build **Products** and **Shell** Angular applications
- Configured parallel job execution for faster build times
- Set up automated testing on push and pull request events
- Added Docker image building for Python API service

Key Outcomes

- Automated build verification run on code change
- Early detection of build failures and integration issues
- Consistent build environment across development and CI
- Ready for Docker image publishing to container registry
- Foundation laid for automated deployment to production

Testing Strategy for Micro-Frontend Application

Importance of testing in Microfrontend applications:

- Ensures each microfrontend works independently.
- Detects UI and functional errors earlier.
- Checks the integration between Shell and Products app
- Improves application reliability
- Prevents breaking changes during pipeline execution.

Testing Tools and Types of tests

Tools used here are:

- Jasmine - Unit testing framework
- Karma - Test runner for Angular unit tests
- Cypress - End-to-end testing tool

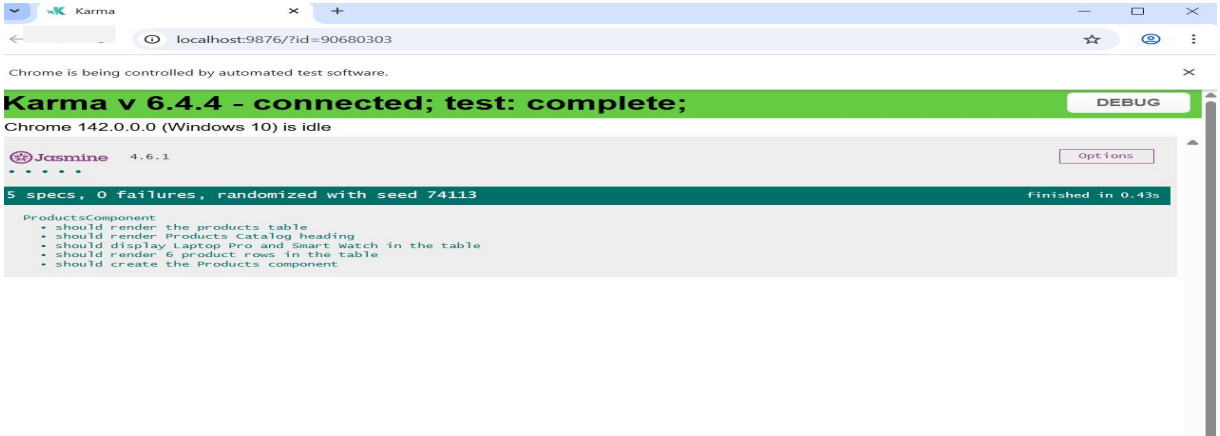
Types of tests performed:

- Component Unit test
- Module federation Integration test
- Cypress end-to-end test

Component Unit Testing

We tested the Shell and Products component using Jasmine and Karma. Tests are written for:

- Components creation test.
- Shell heading render and products UI table render test.
- Navigation link rendering test.

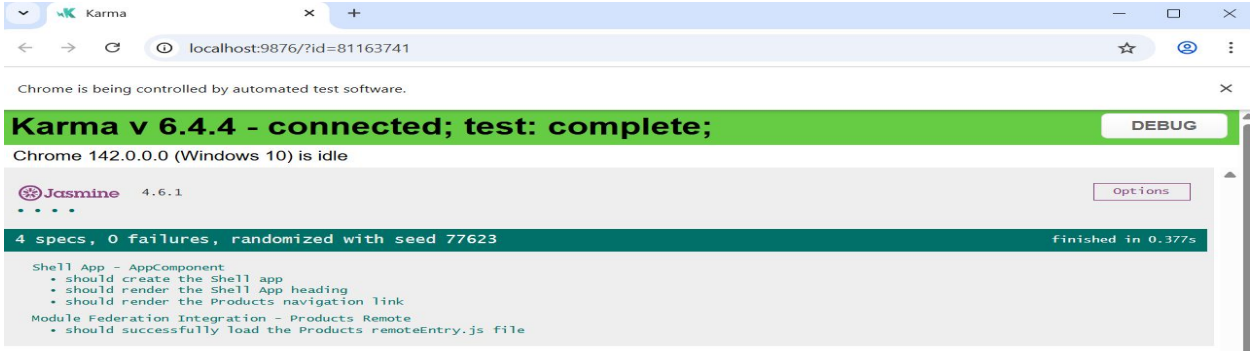


Module Federation Integration Testing

We did the module federation Integration testing to make sure that Shell loads the Products frontend remotely using remoteEntry.js

It checks if:

- The Products remoteEntry.js is fetched.
- Confirms that the remote microfrontend is accessible.
- Ensures Webpack module federation is properly working.



Monitoring & Observability Implementation

To ensure the reliability and performance of the backend API, a complete monitoring and observability stack was implemented using **Prometheus** and **Grafana**. This setup provides real-time insights into service health, request traffic, and resource consumption.

1. Prometheus Setup

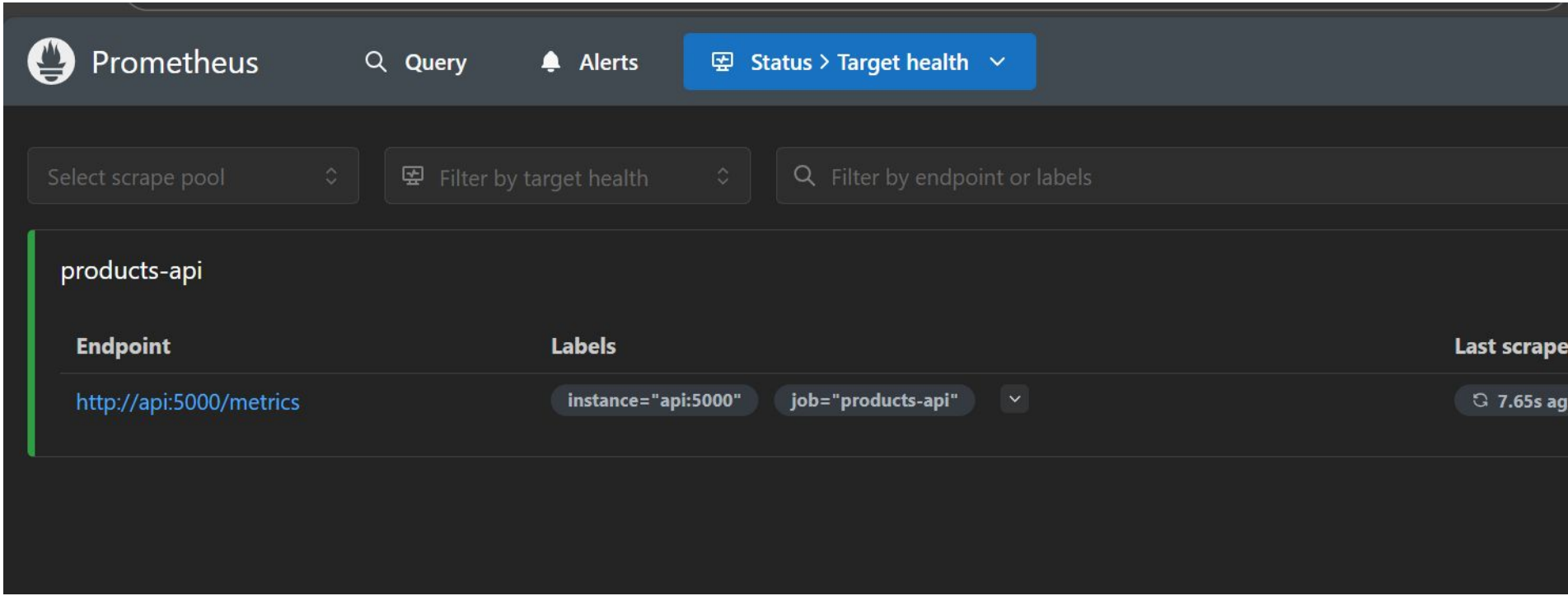
Prometheus was configured to scrape application metrics exposed from the FastAPI service at:

http://api:5000/metrics

A custom job named **products-api** was added inside `prometheus.yml` to collect the following key indicators:

Metric Name	Purpose
`Http_requests_total`	Measures number of API calls
`Up`	Determines API availability/health
`Process_resident_memory_bytes`	Tracks memory utilization

Prometheus Target Health Screenshot shows the API as:
✔ Status: **UP**
✔ Scrape frequency: Every 15s

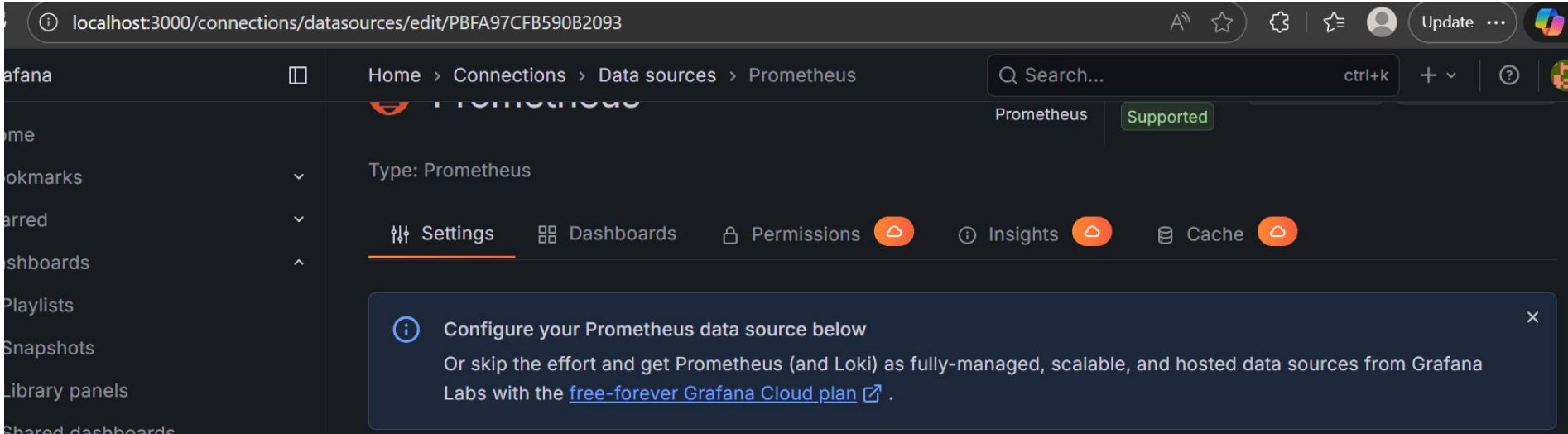


2. Grafana Data Source Configuration

Grafana was connected to Prometheus as its main data source:

Setting	Value
Type	Prometheus
Default	Yes
URL	`http://prometheus:9090`

This allows Grafana to query Prometheus and visualize application metrics in dashboard panels.



3. API Performance Dashboard

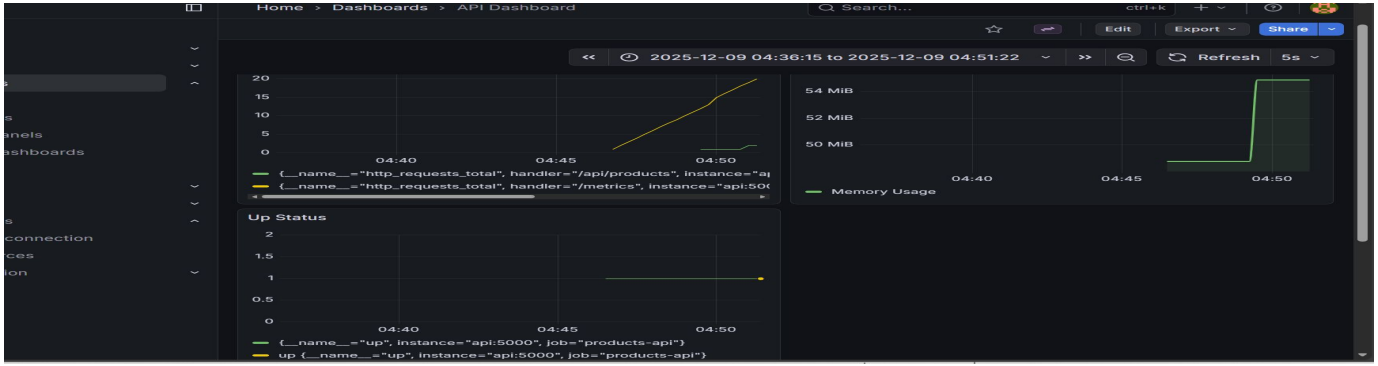
A custom **API Dashboard** was created with panels visualizing:

Panel	Description
API Requests Over Time	Real-time graph of incoming product API requests
Memory Usage	Shows API container memory growth during load
Up Status	Displays service health (1 = UP, 0 = DOWN)

To generate metrics, repeated calls were made to:

http://localhost:5000/api/products

Dashboard auto-refresh is set to 5 seconds to capture Live metrics.



🔹 4. Validation Through Stress Requests

Using a burst of traffic tests, dashboard results confirmed:

- ✔ `http_requests_total` increased with each API hit
- ✔ Memory usage spiked during stress
- ✔ Service remained continuously **UP** (no failures)

This validates system stability and responsiveness under real-time usage.

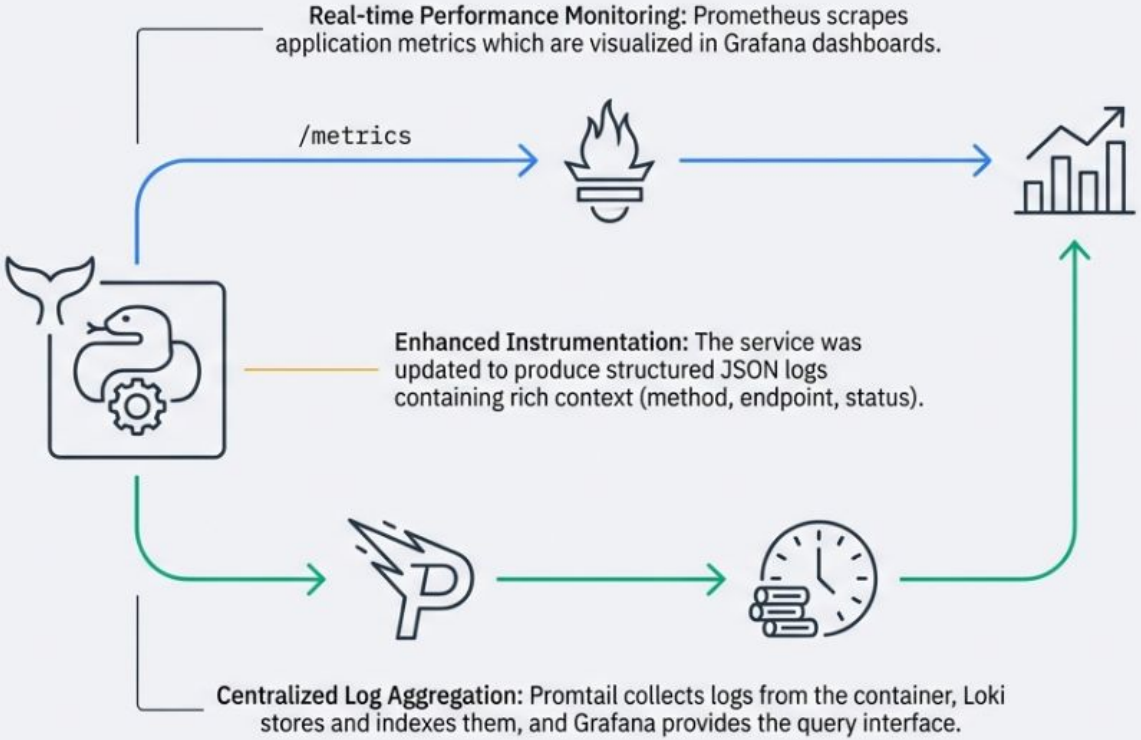
📌 Conclusion

The monitoring architecture successfully provides:

- Continuous visibility into API operations
- Early detection of failures
- Performance trend analysis for future optimizations

This ensures high availability, resilience, and better operational management of the deployed microservice.

Architecting Full Observability for a FastAPI Service



The final design provides a unified observability plane, combining metrics and logs into a single interface for comprehensive analysis.

Live System Visibility: From Metrics to Actionable Logs

Structured JSON Log: Includes critical context like `http_method`, `endpoint`, and `client_address` for powerful filtering and analysis.

Error Capture Confirmed: The `/force-error` endpoint successfully generated this log, proving the system's effectiveness for debugging and error tracking.

Grafana "Explore" Log View

Grafana Metrics Dashboard

Metrics in Action: API calls are directly reflected in Prometheus metrics, confirming the health and performance monitoring setup.

This implementation delivers full transparency into application behavior, significantly improving our ability to debug issues and understand operational performance.

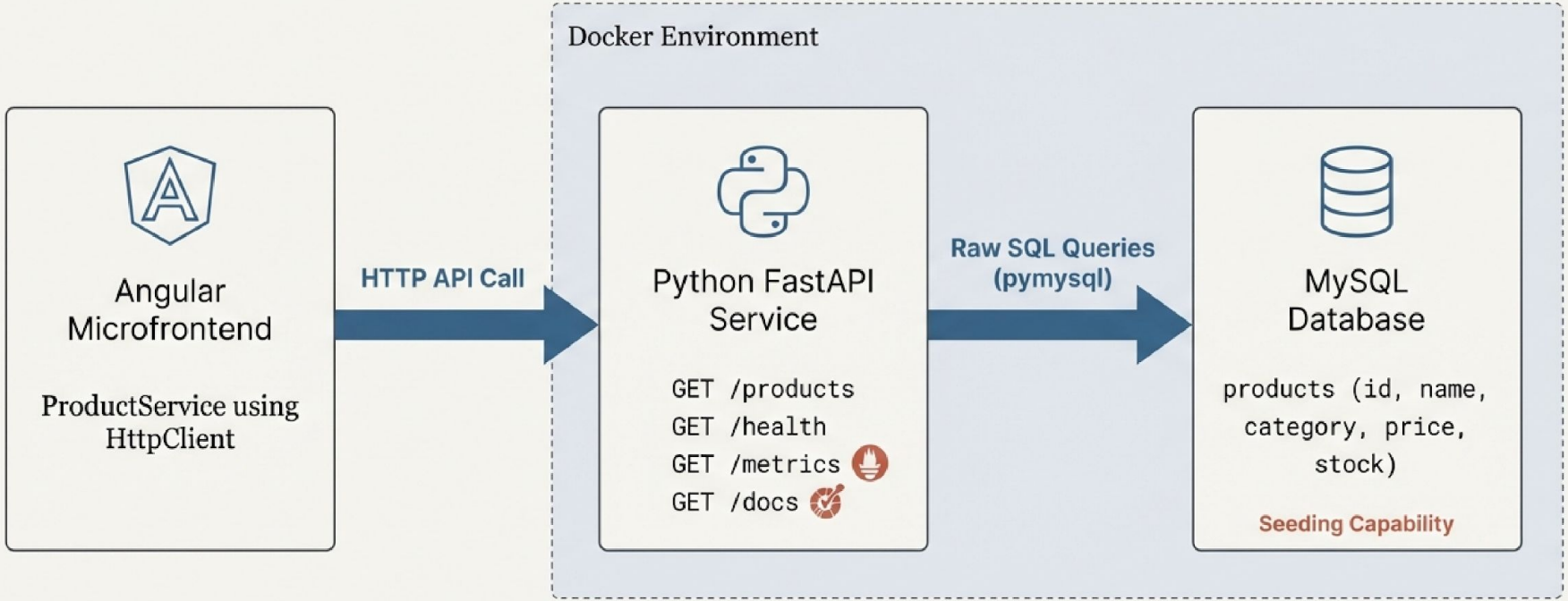
FINAL PROJECT

INFO 8985 - Monitoring and Logging

Nikhil Shankar Chirakkal Sivasankaran - 9026254
Deepak Tamizhalagan - 8983627
Cibi Sharan Cholarani -
Zafar Ahmed Shaik -
Richard Andrey Biscazzi - 8903530

Full-Stack Microservice Architecture: From UI to Database

A containerized, end-to-end data flow powering a dynamic product catalog.



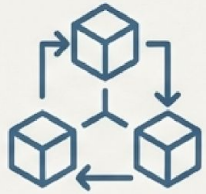
A complete, decoupled system where a Python API serves data from a relational database to an Angular frontend, all orchestrated with Docker.

Outcomes: A Scalable & Developer-Friendly Foundation

The integrated architecture delivers immediate value and is built for future expansion.

Dynamic Product Catalog

Successfully replaced a static, hardcoded product list with a dynamic system powered by a relational MySQL database. The complete data flow from database to UI is seamless and robust.

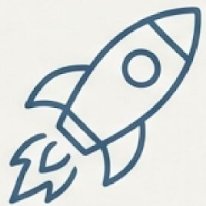


Scalable Microservices

A true microservices architecture with proper separation of concerns between the frontend (Angular), backend (FastAPI), and database. Services are independently containerized and scalable.

Built-in Observability

The API exposes a `/metrics` endpoint, providing out-of-the-box integration with monitoring tools like Prometheus for health checks and performance tracking.



Accelerated Development

With automated database seeding for easy setup and self-generating API documentation at the `/docs` endpoint, the system is designed for developer efficiency and rapid onboarding.

This project establishes a resilient foundation for all future CRUD (Create, Read, Update, Delete) operations and long-term data persistence.

Technology Stack: Deliberate Choices for Performance

Utilizing lightweight, modern tools to build an efficient and maintainable system.

Backend & Database

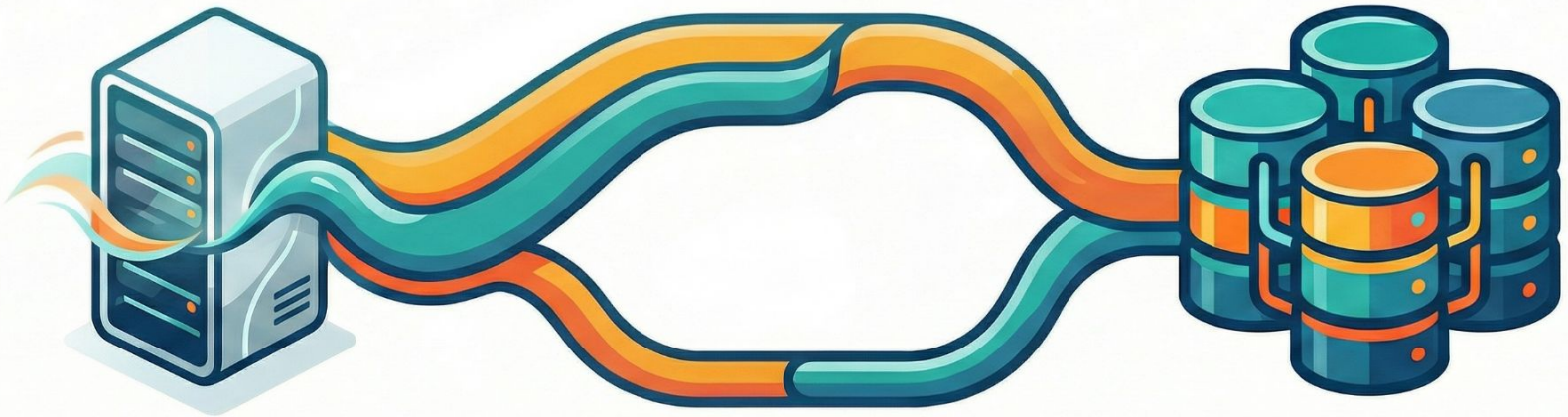
- **FastAPI Framework:** For high-performance, asynchronous API development and automatic OpenAPI documentation.
- **Pymysql Library:** Chosen for its lightweight, dependency-free nature for direct, efficient raw SQL operations.
- **RESTful API Design:** Clean, intuitive endpoints for product retrieval and standardized health checks.
- **Relational Schema:** A structured `products` table designed for data integrity and future query complexity.

Frontend & DevOps

- **Angular Microfrontend:** A decoupled client application for a clean separation from the backend logic.
- **HttpClient Module:** The standard, robust method for consuming RESTful services in Angular.
- **Docker & Docker Compose:** Containerization of both the API and MySQL services, ensuring consistent environments and simple orchestration.
- **Database Health Checks:** Integrated into the Docker setup to ensure service dependencies are managed correctly on startup.

Every component was selected to optimize for performance, scalability, and an excellent developer experience.

Backend Architecture Flow



1. FastAPI Service (Port 5000)

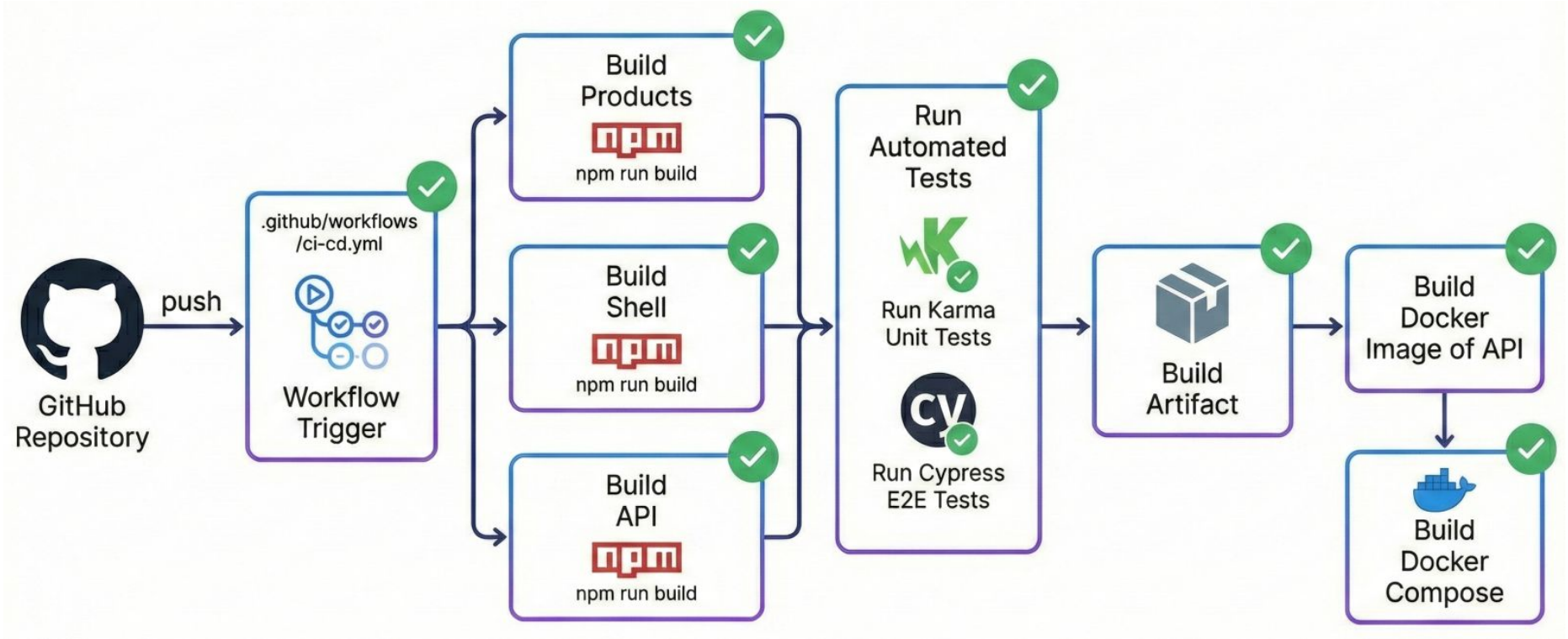
Provides a RESTful API with an /api/products endpoint for clients.

2. Direct Database Communication

The service queries the database directly using raw SQL commands.

3. MySQL Database (Port 8877)

Stores all product data, including ID, name, category, and price.



CI/CD Pipeline with GitHub Actions

- Automated workflow to build **Products** and **Shell** Angular applications
- Configured parallel job execution for faster build times
- Set up automated testing on push and pull request events
- Added Docker image building for Python API service

Key Outcomes

- Automated build verification run on code change
- Early detection of build failures and integration issues
- Consistent build environment across development and CI
- Ready for Docker image publishing to container registry
- Foundation laid for automated deployment to production

Testing Strategy for Micro-Frontend Application

Importance of testing in Microfrontend applications:

- Ensures each microfrontend works independently.
- Detects UI and functional errors earlier.
- Checks the integration between Shell and Products app
- Improves application reliability
- Prevents breaking changes during pipeline execution.

Testing Tools and Types of tests

Tools used here are:

- Jasmine - Unit testing framework
- Karma - Test runner for Angular unit tests
- Cypress - End-to-end testing tool

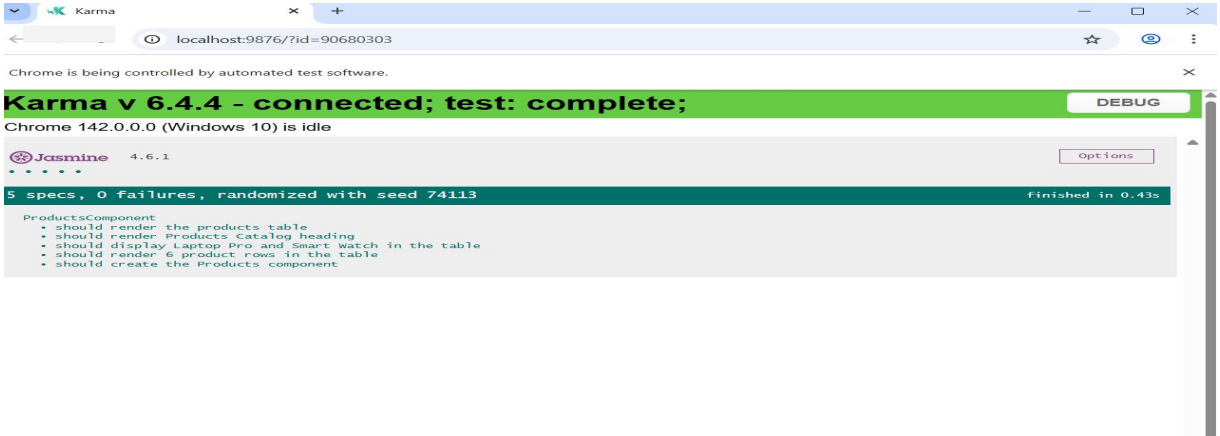
Types of tests performed:

- Component Unit test
- Module federation Integration test
- Cypress end-to-end test

Component Unit Testing

We tested the Shell and Products component using Jasmine and Karma. Tests are written for:

- Components creation test.
- Shell heading render and products UI table render test.
- Navigation link rendering test.

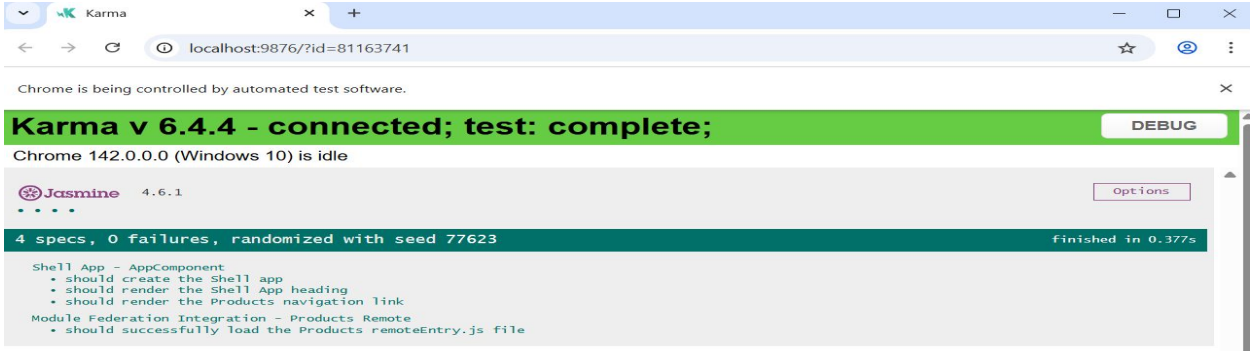


Module Federation Integration Testing

We did the module federation Integration testing to make sure that Shell loads the Products frontend remotely using remoteEntry.js

It checks if:

- The Products remoteEntry.js is fetched.
- Confirms that the remote microfrontend is accessible.
- Ensures Webpack module federation is properly working.



Monitoring & Observability Implementation

To ensure the reliability and performance of the backend API, a complete monitoring and observability stack was implemented using **Prometheus** and **Grafana**. This setup provides real-time insights into service health, request traffic, and resource consumption.

1. Prometheus Setup

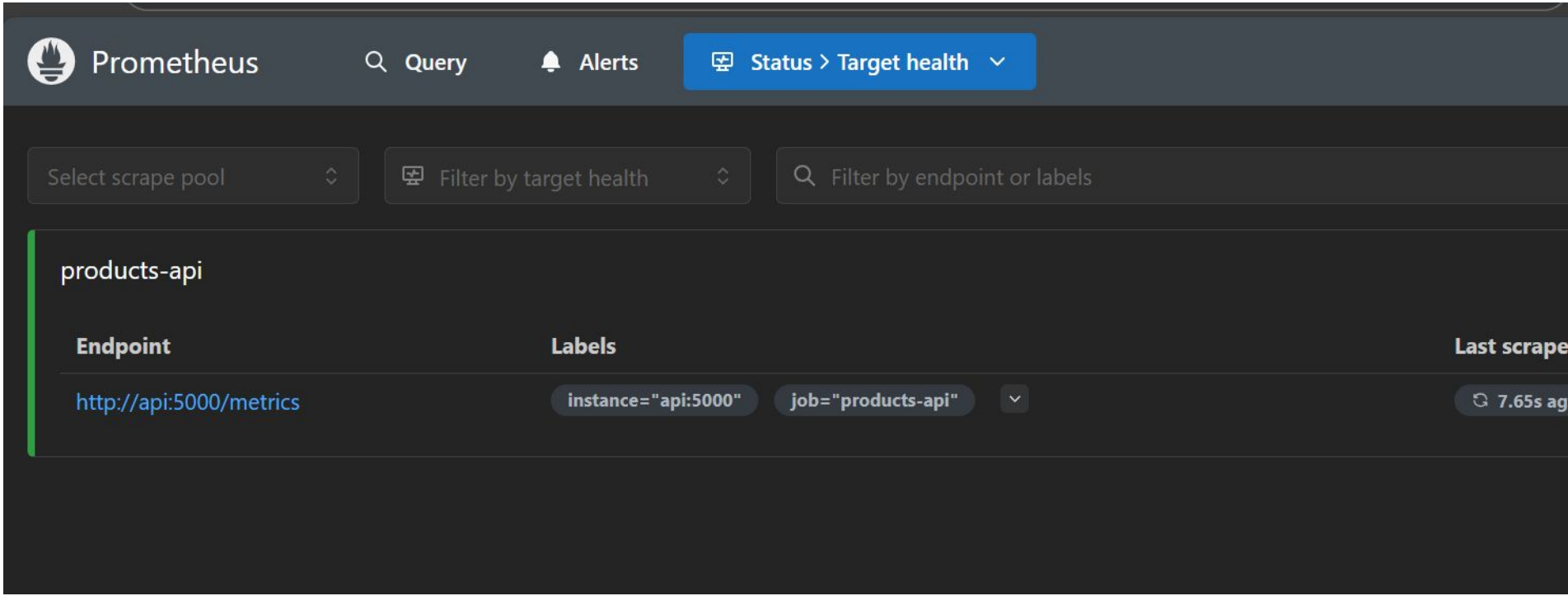
Prometheus was configured to scrape application metrics exposed from the FastAPI service at:

http://api:5000/metrics

A custom job named **products-api** was added inside `prometheus.yml` to collect the following key indicators:

Metric Name	Purpose
`Http_requests_total`	Measures number of API calls
`Up`	Determines API availability/health
`Process_resident_memory_bytes`	Tracks memory utilization

Prometheus Target Health Screenshot shows the API as:
✔ Status: **UP**
✔ Scrape frequency: Every 15s

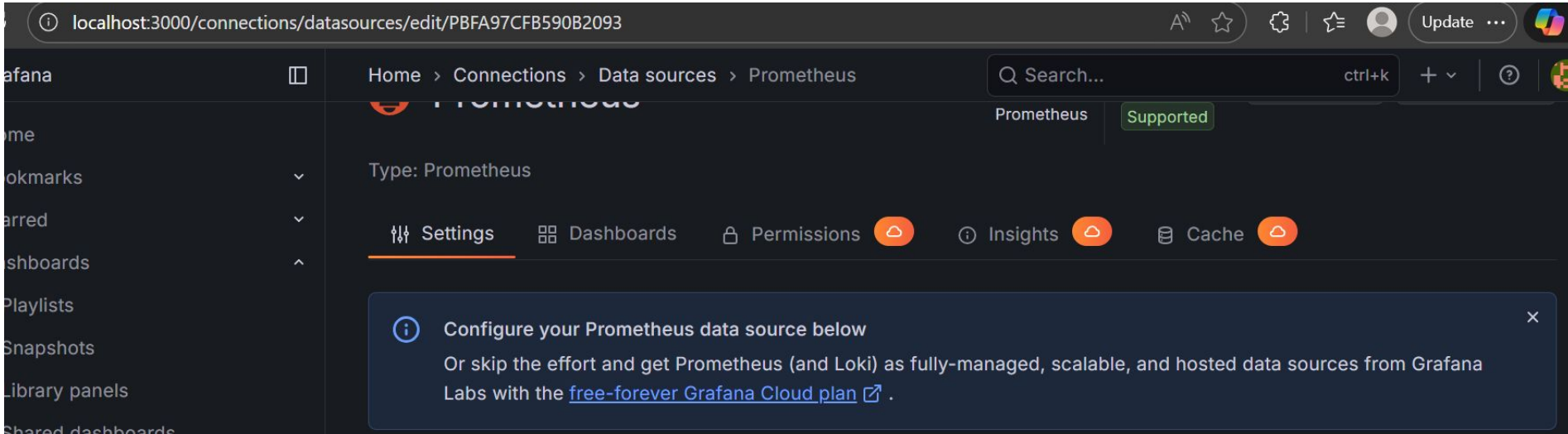


2. Grafana Data Source Configuration

Grafana was connected to Prometheus as its main data source:

Setting	Value
Type	Prometheus
Default	Yes
URL	`http://prometheus:9090`

This allows Grafana to query Prometheus and visualize application metrics in dashboard panels.



3. API Performance Dashboard

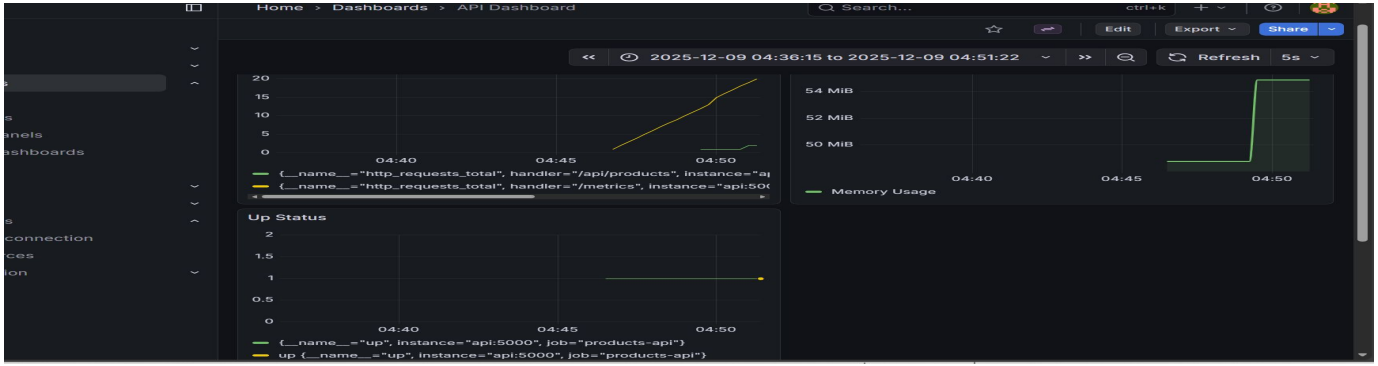
A custom **API Dashboard** was created with panels visualizing:

Panel	Description
API Requests Over Time	Real-time graph of incoming product API requests
Memory Usage	Shows API container memory growth during load
Up Status	Displays service health (1 = UP, 0 = DOWN)

To generate metrics, repeated calls were made to:

http://localhost:5000/api/products

Dashboard auto-refresh is set to 5 seconds to capture Live metrics.



🔹 4. Validation Through Stress Requests

Using a burst of traffic tests, dashboard results confirmed:

- ✔ `http_requests_total` increased with each API hit
- ✔ Memory usage spiked during stress
- ✔ Service remained continuously **UP** (no failures)

This validates system stability and responsiveness under real-time usage.

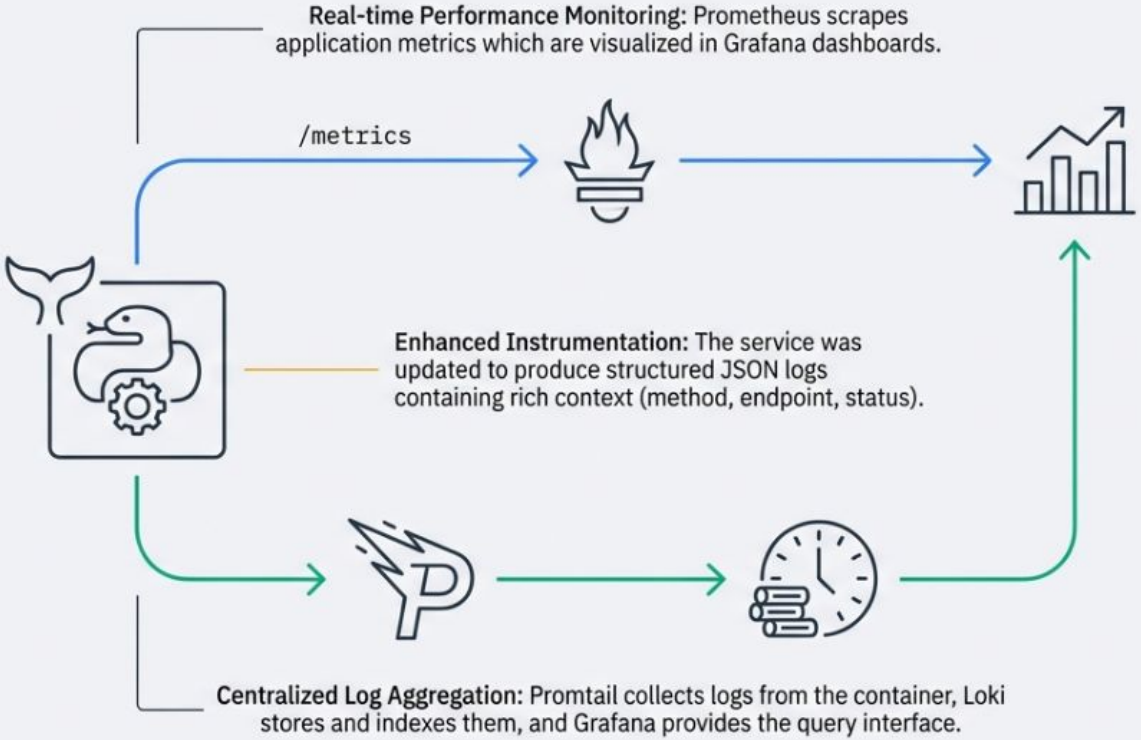
📌 Conclusion

The monitoring architecture successfully provides:

- Continuous visibility into API operations
- Early detection of failures
- Performance trend analysis for future optimizations

This ensures high availability, resilience, and better operational management of the deployed microservice.

Architecting Full Observability for a FastAPI Service



The final design provides a unified observability plane, combining metrics and logs into a single interface for comprehensive analysis.

Live System Visibility: From Metrics to Actionable Logs

Structured JSON Log: Includes critical context like `http_method`, `endpoint`, and `client_address` for powerful filtering and analysis.

Error Capture Confirmed: The `/force-error` endpoint successfully generated this log, proving the system's effectiveness for debugging and error tracking.

Grafana "Explore" Log View

Grafana Metrics Dashboard

Metrics in Action: API calls are directly reflected in Prometheus metrics, confirming the health and performance monitoring setup.

This implementation delivers full transparency into application behavior, significantly improving our ability to debug issues and understand operational performance.