INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

Case Study: Implementing Application Monitoring and Log Analysis in DevOps

Course: DevOps and Cloud Computing

Name: Nasser Peiroti

Student ID: IU14103659

Date of Submission: 22.03.2025

Tutor's Name: Prof. Dr. Tianxiang, Lu

Table of Contents

1. Introduction

In the digital age, web applications that serve a high volume of users demand robust monitoring and effective log analysis. This report documents the implementation of a comprehensive monitoring and logging strategy for a newly developed Django-based web application named "Shahrzadvip". The platform is designed to support an online shopping experience for a dressmaking production line branded "Shahrzad". The website enables users to explore products, manage accounts, make purchases, and engage with content via an integrated blog system.

Given the dynamic nature of web traffic and user engagement, implementing an effective monitoring solution is crucial for identifying performance bottlenecks, ensuring system reliability, and maintaining scalability. Our monitoring solution leverages industry-standard tools such as Prometheus, Grafana, and Docker, while Locust was employed for performance testing.

1.1. Challenges and Motivations for Monitoring

The Shahrzadvip platform introduces several challenges that necessitate an effective monitoring and logging strategy:

1. Dynamic Traffic Patterns: As an e-commerce website, the platform expects fluctuating traffic, particularly during promotions or seasonal sales. Effective monitoring is required to detect sudden spikes and ensure uptime.
2. Complex Application Logic: The Django-based architecture integrates multiple modules such as cart management, user authentication, inventory management, and order processing. Ensuring these interdependent features function seamlessly under high demand requires continuous monitoring.
3. Performance Bottlenecks: Key performance indicators such as response time, CPU/memory usage, and request throughput must be monitored to proactively detect issues.
4. Error Tracking and Debugging: As the project is newly developed, unknown bugs and performance bottlenecks may arise. Monitoring and detailed log analysis are essential to diagnose and resolve such issues effectively.

5. Data Integrity and Security: E-commerce platforms handle sensitive data like customer information and transaction details. Monitoring must ensure secure practices and maintain user trust.

By implementing a robust monitoring strategy, we aim to ensure high performance, minimize downtime, and deliver a smooth shopping experience for Shahrzadvip customers.

1.2. Potential System Bottlenecks and Monitoring Focus

A comprehensive understanding of the application's architecture is crucial for identifying potential performance risks. Upon analyzing the views and models of the Shahrzadvip project, the following key concerns were identified:

- Cart App:
    - Challenge: The cart heavily relies on frequent database updates and concurrent read/write operations during peak user activity.
    - Monitoring Focus: Tracking database latency, query execution times, and detecting locking or blocking issues is essential.
- Store App:
    - Challenge: The product catalog involves dynamic filtering, sorting, and querying, which may introduce performance overhead as the product list grows.
    - Monitoring Focus: Emphasis on indexing strategies, cache efficiency, and response time for product searches.
- User Management:
    - Challenge: Features like user authentication, session management, and permission checks may strain the system if requests are not efficiently handled.
    - Monitoring Focus: Monitor active sessions, login attempt rates, and response times for key authentication endpoints.
- Blog App:
    - Challenge: Since blogs often have dynamic content generation and media uploads, there is a risk of slower response times when serving static files or content-heavy pages.
    - Monitoring Focus: Track response times for blog endpoints and ensure CDN integration for static file distribution.

- Database Design:
    - Challenge: Complex foreign key relationships in models like OrderItem, Product, and Discount could lead to inefficient joins or slow aggregation queries.
    - Monitoring Focus: Enable query performance monitoring, track slow queries, and analyze database locks.
- API & Third-Party Integrations:
    - Challenge: The application may use third-party APIs for payment gateways, shipment tracking, or external data sources. Network delays could impact user experience.
    - Monitoring Focus: Implement latency monitoring and track third-party API failures.

1.2.1. Identifying Key System Bottlenecks and Monitoring Priorities

Before implementing monitoring solutions, it’s critical to analyze the system's architecture and identify potential performance bottlenecks. By evaluating the application’s structure, data flow, and usage patterns, we can prioritize critical components that are most vulnerable to performance degradation.

Key Analysis Process

Our decision to prioritize the shop_grid view for focused monitoring was informed by the following steps:

- Review of Core Application Components:
    - The application was divided into core functional modules: Cart**,** User**,** Store**,** and Blog. Among these, the Store module handles the primary product browsing experience — a critical user-facing component essential for driving sales and engagement.
- Database Query Analysis:
    - During our evaluation of the Django application’s views, the shop_grid view was identified as one of the most query-intensive endpoints**.** It involves:
        1. Multiple database joins (e.g., Product, Discount, Category).

2. Sorting logic across several fields (price, popularity, etc.).
3. Potential performance risks due to non-optimized pagination and dynamic query construction**.**

- Performance Risk Assessment:
  - Given the nature of an e-commerce site like Shahrzadvip**,** the shop_grid endpoint is expected to receive a high volume of concurrent requests, especially during promotional campaigns or seasonal sales.
  - Failure to efficiently serve this endpoint could result in delayed page loads, high server load, and a poor user experience — directly impacting sales and customer retention.
- Code Complexity and Scalability Concerns:
  - The complexity of dynamically combining multiple query filters and sorting logic makes this view prone to scalability issues as product data scales over time.

Based on these findings, we identified the shop_grid view as a prime candidate for intensive monitoring. By tracking its performance metrics closely, we aim to:

- Detect inefficient queries or missing database indexes.
- Identify memory and CPU spikes that may signal performance bottlenecks.
- Monitor the endpoint's throughput to ensure scalability under high traffic

2. Monitoring and Log Analysis Strategy

2.1 Assessing System Performance Risks and Monitoring Priorities

Given that Shahrzadvip is a newly developed platform, we initially faced a challenge of insufficient real-world data to effectively simulate realistic user behavior and system load. Without adequate data, meaningful insights from monitoring tools such as Prometheus and Grafana would be incomplete or misleading.

To address this, we utilized Faker, a powerful library in Django, to generate realistic sample data for the Product model. By populating the database with thousands of realistic entries, we successfully mimicked real-world product data, including product names, descriptions, prices, and stock levels.

Why Faker?

- Data Scarcity: Since Shahrzadvip was newly developed, no user interactions had yet populated the database.
- Simulated Load Conditions: By generating thousands of fake products, we ensured our application had realistic query loads during monitoring and load testing.
- Controlled Testing Environment: Faker allowed us to generate product data with varied prices, categories, and descriptions, closely resembling actual user activity on the site.

Code Snippet for Faker Data Generation (Product Model Example)

```
import random
from faker import Faker
from store.models import Product, Category

fake = Faker()

def create_fake_products(n=1000):
    categories = Category.objects.all()
    for _ in range(n):
        Product.objects.create(
            name=fake.company(),
            description=fake.text(),
            price=random.uniform(10.0, 500.0),
            stock=random.randint(1, 100),
            category=random.choice(categories)
        )

print("Fake products generated successfully!")
```

By using this approach, we ensured the shop_grid endpoint encountered realistic product data queries during performance testing and monitoring.

2.2 Monitoring Tools and Configuration

Our monitoring strategy combines various tools to ensure comprehensive performance tracking and issue detection. Each tool was selected to address specific aspects of our system:

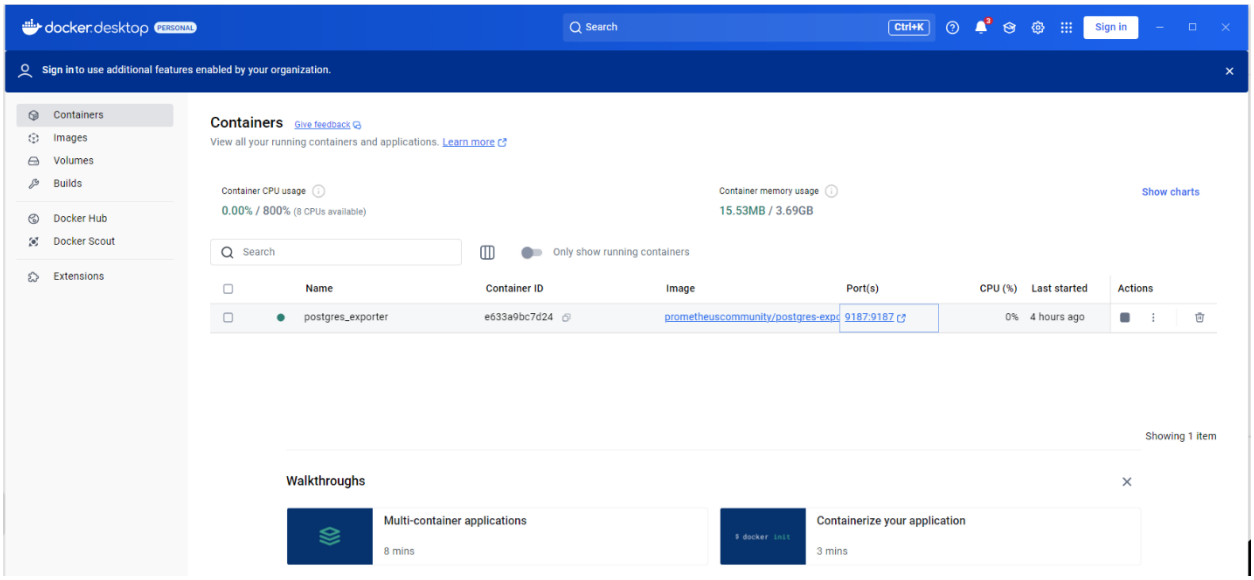


Figure 1_Docker

2.2.1 Prometheus Metrics for PostgreSQL Monitoring via Docker

Since PostgreSQL is the backbone of our data management system, integrating Prometheus with Docker was essential for collecting detailed database performance metrics. The following Prometheus metrics were configured to achieve granular visibility into database performance:

- Database Connection Tracking: Using Prometheus to track the number of active, idle, and long-running connections, helping us identify connection pool issues.
- Query Latency Measurement: PromQL queries were designed to monitor query execution times, with particular emphasis on the shop_grid view.
- Cache Efficiency Analysis: We monitored the effectiveness of caching by tracking buffer hits versus direct disk reads.

- Database Throughput Monitoring: Metrics related to the number of transactions, query counts, and write/read rates were collected to observe PostgreSQL's overall load.

Sample PromQL Queries for PostgreSQL Monitoring via Docker:

```
# Track active database connections
pg_stat_activity_count{state="active"}

# Monitor slow queries (e.g., exceeding 500ms)
pg_stat_activity_max_tx_duration{state="active"} > 0.5

# Cache hit ratio
rate(pg_stat_database_blks_hit{datname="shahrzad_db"}[5m])
/
(rate(pg_stat_database_blks_hit{datname="shahrzad_db"}[5m]) +
rate(pg_stat_database_blks_read{datname="shahrzad_db"}[5m]))

# Transaction rate
rate(pg_stat_database_xact_commit{datname="shahrzad_db"}[5m]
```

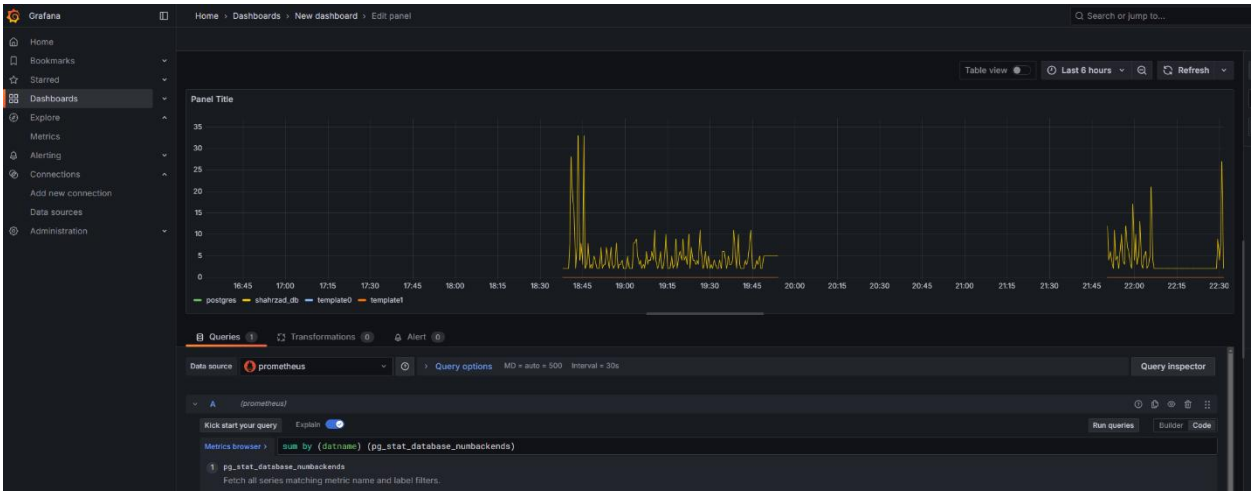


Figure 2_Grafana

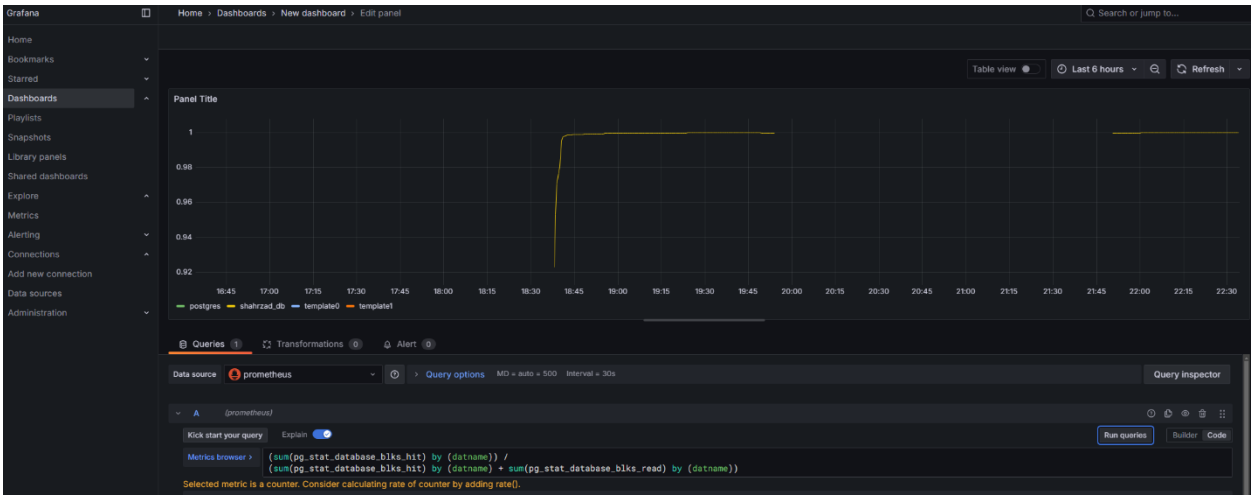
2.2.2 Grafana Dashboards for Visualization

We employed Grafana for its powerful visualization capabilities. The following key panels were configured to enhance our PostgreSQL insights:

- Real-time Connection Usage Panel: Displays active, idle, and waiting connections.
- Query Performance Panel: Visualizes slow queries in the shop_grid view to identify bottlenecks.
- Database Load Panel: Tracks CPU and memory consumption by the PostgreSQL container in Docker**.**

Key Alert Configurations in Grafana:

- Trigger alerts when the shop_grid view response time exceeds 500ms.
- Trigger alerts if PostgreSQL's cache miss rate surpasses 30%.



2.2.3 Load Testing with Locust

To evaluate system resilience under stress conditions, we employed Locust to simulate user behavior. The following testing parameters were defined:

- Simulated 100-200 concurrent users accessing the shop_grid view to assess its response time, throughput, and error rate.
- Monitored PostgreSQL's load during simulated peaks to identify query delays or connection saturation.

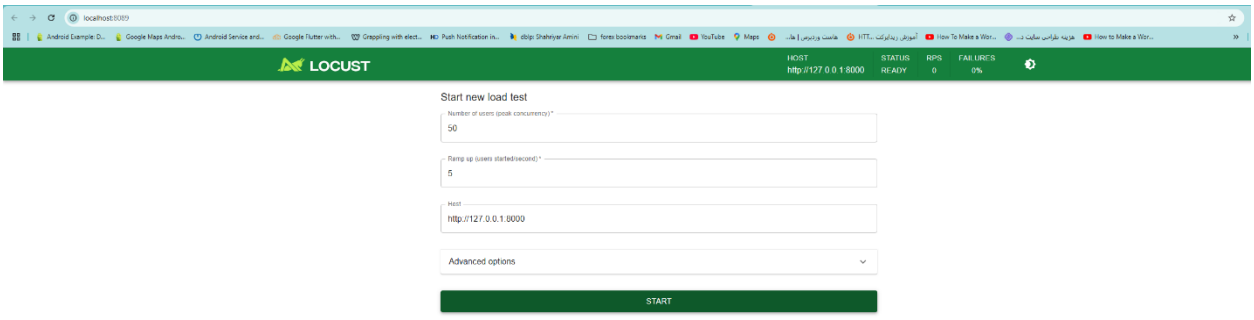


Figure 3_Locust

2.2.4 Centralized Logging with Docker and Prometheus

For comprehensive log analysis, we configured Docker to route container logs directly into Prometheus, ensuring that we capture critical database events such as:

- Query failures
- Slow transaction warnings
- Potential deadlock situations
- Connection saturation issues

This centralized logging configuration enabled us to proactively detect anomalies and correlate performance degradation patterns with specific database activities.

3. Deployment Plan

To ensure a stable and reliable deployment process for the monitoring system, a carefully staged deployment strategy was adopted. This strategy allowed us to minimize risks, validate configurations, and ensure optimal functionality without impacting the live production environment.

Figure 4_Django locust running

3.1 Local Environment Setup

Since the shahrzadvip web application is already hosted live, we deliberately conducted our monitoring and testing in a local development environment to avoid disruptions to the production system. This approach allowed us to simulate realistic conditions while maintaining full control over the monitoring configurations.

3.2 Step 1: Preparing the Local Environment

1. Database Preparation with Faker:
    - Since shahrzadvip is a newly developed application without substantial data, we populated the PostgreSQL database with sample data using the Faker library in Django.
    - Faker was run before initiating monitoring tools to ensure the database contained realistic data patterns resembling production usage.

2. Running the Django Application on Localhost:
    - The Django application was launched locally to emulate typical web interactions and traffic patterns.
    - This provided a controlled environment for monitoring without impacting the live website.

3.3 Step 2: Traffic Simulation with Locust

To simulate various traffic conditions and assess system performance under realistic scenarios, we utilized Locust as our load-testing tool. Traffic levels were designed to reflect typical user behaviors during low, medium, and high usage periods:

- Low Traffic Scenario: Simulated 50 users with a spawn rate of 5 users/second.
- Medium Traffic Scenario: Simulated 100 users with a spawn rate of 10 users/second.
- High Traffic Scenario: Simulated 200 users with a spawn rate of 20 users/second.

Each scenario targeted key endpoints such as the shop_grid view to evaluate query performance, page load times, and database efficiency.

3.4 Step 3: Deploying the PostgreSQL Database in Docker

To enable detailed database monitoring, the PostgreSQL instance was containerized using Docker. This setup allowed us to seamlessly integrate PostgreSQL metrics into Prometheus for enhanced visibility.

- The Docker container was configured to expose PostgreSQL metrics via the PostgreSQL Exporter, ensuring Prometheus could scrape key metrics directly.
- Metrics such as active connections, slow queries, cache hit ratios, and transaction throughput were collected.

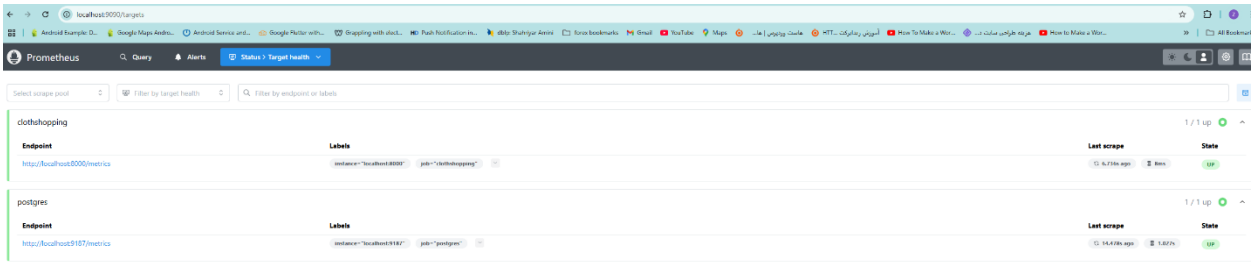


Figure 5_prometheus

3.5 Step 4: Prometheus Configuration and Monitoring

Prometheus was configured to collect comprehensive application and database performance metrics. The key steps included:

- Configuring Prometheus to scrape:
  - PostgreSQL Exporter for database performance data.
  - Django Application Metrics for request latency, error rates, and cache efficiency.
- PromQL queries were written to generate insightful metrics, especially for the shop_grid view.

Sample PromQL Query for Monitoring the shop_grid View:

promql
CopyEdit
Monitoring shop_grid response time exceeding 500ms
pg_stat_activity_max_tx_duration{datname="shahrzad_db", state="active"} > 0.5

3.6 Step 5: Visualization with Grafana

The final deployment stage involved integrating Prometheus data into Grafana dashboards to visualize performance trends, identify bottlenecks, and set up proactive alerts. Key panels in Grafana included:

- Database Load Monitoring: Tracking CPU, memory consumption, and database transactions.
- Cache Efficiency Panel: Displaying cache hit rates for critical views like shop_grid.
- Alert System: Configured alerts to notify developers if:
    - The shop_grid response time exceeded 500ms.
    - Cache miss rates surpassed 30%.
    - The number of active PostgreSQL connections exceeded predefined thresholds.

3.7 Step 6: Ensuring Stability and Validating Monitoring Tools

Before scaling the monitoring setup, we:

- Observed real-time metrics in Grafana to verify correct data collection.
- Evaluated database behavior under varying traffic conditions to ensure Prometheus effectively identified performance patterns.
- Conducted repeated tests to validate the consistency and reliability of alerts and performance data.

3.8 Final Deployment Strategy Justification

The chosen deployment strategy ensured:

- Minimal Production Risk: By performing all configurations and simulations locally, we avoided any impact on the live shahrzadvip platform.
- Robust Data Collection: Using Docker, Prometheus, and Grafana ensured granular monitoring at both the application and database levels.
- Proactive Issue Resolution: With alerts configured for critical metrics, potential issues could be identified and resolved before impacting user experience.

This staged deployment strategy allowed us to deliver a comprehensive and reliable monitoring solution while preserving system stability and minimizing risks.

4. Security and Privacy in Monitoring

Given that our monitoring environment was run locally to avoid interfering with the live shahrzadvip website, we adopted targeted security measures to ensure data integrity and privacy during testing.

4.1 Role-Based Access Control (RBAC)

- RBAC was configured for Prometheus, Grafana, and Docker to restrict access based on roles:
    - Developers could access monitoring dashboards and logs for debugging.
    - Administrators managed configuration settings and alerts.
    - Access to sensitive data was strictly limited to essential users.

4.2 Data Encryption for Secure Communication

- Since Prometheus scrapes metrics directly from PostgreSQL Exporter and the Django application, TLS encryption was enabled to secure data during transmission.

Example TLS Configuration in Prometheus (prometheus.yml)

```
scrape_configs:
  - job_name: 'postgresql_metrics'
    scheme: https
    tls_config:
      ca_file: /etc/prometheus/ssl/ca.crt
      cert_file: /etc/prometheus/ssl/client.crt
      key_file: /etc/prometheus/ssl/client.key
```

4.3 Log Anonymization and Data Masking

- As the shahrzadvip application handles customer orders, logs were filtered to remove or mask sensitive data such as usernames, contact details, and financial data.

Example Anonymized Log Entry

```
INFO 2025-03-15 12:03:45 - User [ID: 12345] placed an order for Product_ID: XXX
```

4.4 Secure Local Container Configuration

- While the deployment occurred locally, we ensured secure Docker practices by:
    - Running the PostgreSQL container with non-root privileges.
    - Using strong credentials for local PostgreSQL access.
    - Avoiding unnecessary exposure of services by binding them strictly to localhost.

Docker Compose Configuration for Local Isolation

```
services:
  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: securepassword
    ports:
      - "5432:5432"  # Exposed locally only
```

4.5 Secure Access to Grafana

- Grafana was configured with OAuth 2.0 authentication to prevent unauthorized dashboard access. Since Grafana was locally hosted, this step ensured only authenticated users could access performance metrics.

4.6 Summary of Security Practices

Security Measure	Purpose
Role-Based Access Control (RBAC)	Minimize unauthorized data exposure
TLS Encryption	Secure data during transmission
Log Anonymization	Protect sensitive user data in logs
Local Docker Isolation	Prevent accidental external exposure of containers
OAuth 2.0 Authentication	Ensure secure dashboard access

5. Log Analysis and Issue Resolution

To identify and resolve performance issues, a centralized log analysis strategy was implemented.

- Prometheus was configured with alert rules to trigger notifications in Grafana when error rates or response times exceeded predefined thresholds.
- Locust was used to simulate different traffic scenarios — low (50 users), medium (100 users), and high (200 users) revealing performance bottlenecks.

During testing, Grafana metrics showed that the shop_grid view experienced performance degradation during load spikes, with response times exceeding acceptable thresholds. This finding was further validated by PostgreSQL metrics, which indicated high query execution times during these spikes.

These insights enabled developers to identify inefficient database queries in the shop_grid view, prompting optimizations that improved overall performance and scalability.

This combination of alerting, visualization, and load testing proved highly effective in guiding performance enhancements.

6. Conclusion

This case study highlights the importance of a structured monitoring and log analysis strategy in ensuring the reliability and scalability of a newly developed web application like ShahrzadVIP. Given the startup's limited data availability, the use of Faker effectively populated the database to simulate realistic user interactions.

By integrating Prometheus**,** Grafana**,** and Docker into our monitoring setup, we achieved granular visibility into application and database performance. Connecting Docker PostgreSQL allowed for detailed tracking of query execution times, cache efficiency, and overall database load.

The results from Locust load testing — under low, medium, and high traffic scenarios — provided valuable insights into system behavior under varying loads. Notably, performance degradation in the shop_grid view during traffic spikes was identified, with metrics confirming that database queries were the primary cause. These insights empowered the development team to optimize queries and improve database efficiency.

Through proactive alerting in Grafana, combined with detailed performance analysis from Prometheus and Locust, the strategy successfully enhanced system reliability, scalability, and user experience. This comprehensive monitoring solution ensures that ShahrzadVIP can adapt to future growth while maintaining optimal performance.

Bibliography

Docker, Inc. (n.d.). Docker documentation. Retrieved March 23, 2025, from https://docs.docker.com/

Prometheus Authors. (n.d.). Prometheus documentation. Retrieved March 23, 2025, from https://prometheus.io/docs/introduction/overview/

PyPI. (n.d.). django-faker. Retrieved March 23, 2025, from https://pypi.org/project/django-faker/

Locust. (n.d.). Locust documentation. Retrieved March 23, 2025, from https://docs.locust.io/en/stable/

Django Software Foundation. (n.d.). Django documentation. Retrieved March 23, 2025, from https://docs.djangoproject.com/

Python Software Foundation. (n.d.). Python documentation. Retrieved March 23, 2025, from https://www.python.org/doc/

PostgreSQL Global Development Group. (n.d.). PostgreSQL documentation. Retrieved March 23, 2025, from https://www.postgresql.org/docs/