**Assignment Activity Unit 4**

by

Anonymous User

An Assignment submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Information Technology

MIST 5250-01 – Foundations of Software Engineering

Term 5 **-** 2025

University of The People

July 15, 2025

Assignment: Project-Based Learning, Debugging, and Performance Optimization

Effective team collaboration on a software project hosted on GitHub necessitates a well-defined branching strategy to streamline development processes, reduce conflicts, and boost overall productivity. Implementing a structured approach to version control is crucial for maintaining code quality and ensuring smooth integration of various development efforts. The Gitflow model organizes the development workflow into several distinct branches, each serving a specific purpose. This structure facilitates parallel development, controlled releases, and efficient bug fixes, thereby enhancing the team's ability to deliver high-quality software consistently.

**Scenario 1**

**Designing a Git Branching Strategy**

Efficient team collaboration on a software project hosted on GitHub requires a structured branching strategy that minimizes conflicts and enhances productivity. For our web application, I would adopt the **Gitflow** branching model, which is widely recommended for collaborative software development (Driessen, 2010). Gitflow organizes work into several branches:

* **Main (or Master)**: This branch always contains production-ready code.
* **Develop**: The primary integration branch where features and fixes are merged before being released.
* **Feature branches**: Created from develop, each for a specific feature, allowing developers to work independently without disrupting the main codebase.
* **Release branches**: Created from develop when preparing for production deployment, used for final testing and minor fixes.
* **Hotfix branches**: Created from main to quickly fix critical production issues, then merged back into both main and develop.

Using this strategy, the team can develop multiple features in parallel, minimize merger conflicts, and maintain a stable main branch for production deployments. It also facilitates code reviews and incremental integration, crucial for ensuring software quality.

**Implementing Continuous Integration (CI) with GitHub Actions**

To automate testing and deployment, I would implement **Continuous Integration (CI)** pipelines using GitHub Actions. GitHub Actions allows us to create workflows triggered on specific Git events, such as pushes or pull requests.

A typical CI pipeline for our web application would include:

* **Automated Testing**: Each commit triggers unit tests, integration tests, and end-to-end tests to ensure new code does not break existing functionality.
* **Code Quality Checks**: Tools like ESLint for JavaScript and flake8 for Python backend can analyze code quality and enforce consistent style (Spinellis, 2020).
* **Build Process**: Compiling front-end assets and preparing backend services.
* **Deployment**: Deploying successful builds to staging or production environments automatically.

For example, a .github/workflows/ci.yml workflow could define steps such as:

1. name: CI Pipeline

2. on:

3. push:

4. branches:

5. - develop

6. - main

7. pull\_request:

8. branches:

9. - develop

10.

11. jobs:

12. build-and-test:

13. runs-on: ubuntu-latest

14. steps:

15. - uses: actions/checkout@v3

16. - name: Set up Node.js

17. uses: actions/setup-node@v3

18. with:

19. node-version: '18'

20. - run: npm install

21. - run: npm run test

22.

GitHub Actions also supports secrets, which can be used for securely storing API keys or deployment credentials.

**Benefits for Collaboration and Quality**

Implementing Gitflow and CI brings multiple advantages:

* **Improved Collaboration**: Developers can work on isolated branches, reducing conflicts and enabling parallel development. Code reviews on pull requests improve code quality and knowledge sharing (Rigby & Storey, 2011).
* **Avoiding Merge Conflicts**: Smaller, frequent merges into the development branch help avoid large, complex conflicts.
* **Consistency and Reliability**: CI ensures every change is tested before merging, reducing the risk of introducing bugs.
* **Faster Feedback**: Automated tests provide quick feedback on new commits, helping teams catch errors early in development.
* **Enhanced Project-Based Learning**: Working in a professional Git environment simulates real-world software engineering practices, connecting theory to practical skills, as recommended in project-based learning methodologies (Krajcik & Blumenfeld, 2006).

Thus, using Gitflow and GitHub Actions would significantly improve our team’s efficiency, collaboration, and product quality.

**Scenario 2**

**Analyzing Causes of Slow Database Performance**

The e-commerce platform’s slow performance during product searches, filtering, and checkout can stem from multiple factors:

1. **Lack of Proper Indexing**: If indexes are missing on frequently searched columns (e.g., product name, category), the database must scan entire tables, increasing query times. (*MySQL :: MySQL 8.4 Reference Manual :: 15.8.2 EXPLAIN Statement*, n.d.)
2. **Complex SQL Queries**: Joins on large tables or subqueries can be resource-intensive if not optimized (Silberschatz et al., 2019).
3. **High Concurrency**: Many simultaneous transactions during peak hours can lead to lock contention, slowing down queries.
4. **Hardware Constraints**: Limited CPU, memory, or disk I/O can bottleneck database operations.
5. **Inefficient Data Design**: Poor schema design, like lack of normalization or excessive denormalization, can cause redundancy and bloated data retrieval.

These factors contribute to the observed lag in product searches, filtering, and checkout processes.

**Debugging and Troubleshooting Techniques**

To diagnose slow queries, I would follow systematic debugging steps:

1. **Query Logging and Analysis**: Enable the database’s slow query log to identify queries exceeding a specific execution time threshold. In MySQL, this can be done with:

1. SET GLOBAL slow\_query\_log = 'ON';

2. SET GLOBAL long\_query\_time = 1;

3.

1. **EXPLAIN Plans**: Use the EXPLAIN keyword to analyze query execution plans. It shows how tables are joined, which indexes are used, and how many rows are scanned. (*MySQL :: MySQL 8.4 Reference Manual :: 15.8.2 EXPLAIN Statement*, n.d.)

For example:

1. EXPLAIN SELECT \* FROM products WHERE category = 'Electronics' AND price < 500;

1. **Database Monitoring Tools**: Tools like **pgAdmin** for PostgreSQL, **MySQL Workbench**, or cloud-based services like **AWS RDS Performance Insights** can visualize metrics such as query latency, CPU utilization, and wait events.
2. **Query Profiling**: In some databases, commands like SHOW PROFILE can reveal time spent in parsing, execution, and sending results.
3. **Concurrency Testing**: Simulate high-load scenarios using tools like JMeter or Locust to identify bottlenecks under peak traffic (Dumas et al., 2013).

By applying these techniques, we can pinpoint problematic queries and design targeted optimizations.

**Optimization Strategy for Queries**

To improve performance, especially during peak periods, I would implement the following strategies:

**1. Index Optimization**

* **Single-Column Indexes**: Create indexes on frequently searched fields such as *category\_id, price*, and *product\_name.*
* **Composite Indexes**: For queries filtering on multiple columns, composite indexes can reduce the need for table scans. For example:

1. CREATE INDEX idx\_category\_price

2. ON products (category\_id, price);

3.

* **Covering Indexes**: Include all columns needed for a query in the index, so the database can fetch results directly from the index without accessing the table.

**2. Query Refactoring**

* **Simplify Joins**: Minimize unnecessary joins, especially with large tables.
* **Use LIMIT and Pagination**: Fetch only the required data, reducing memory usage and response times.
* **Avoid SELECT**: Explicitly specify needed columns to reduce data transfer overhead (Silberschatz et al., 2019).

**3. Caching Strategies**

* Implement catching popular product searches using Redis or Memcached to reduce database load during peak times.
* Cache query results that do not change frequently, such as product listings or categories.

**4. Database Partitioning**

* Partition large tables by category or date to reduce the amount of data scanned during queries.

**5. Connection Pooling**

* Use connection pools to manage database connections efficiently, reducing overhead during spikes in user activity.

**6. Hardware Scaling**

* Consider vertical scaling (upgrading hardware) or horizontal scaling (adding read replicas) to distribute load during busy hours.

Applying these optimizations would ensure the platform scales efficiently while maintaining fast response times, which is crucial for user experience in e-commerce applications. As Kumar (2013) highlight, “Both scheduling and load shedding (as QoS delivery mechanisms) minimize resource usage (e.g., queue size, tuple latency) and maximize performance and throughput.” (p. 321). This is the same as indexing database for high scaling.

**Conclusion**

This assignment illustrates the importance of combining theoretical knowledge with practical applications in software development. Effective branching strategies and CI/CD pipelines enhance collaboration and code quality, while systematic debugging and optimization techniques are vital for addressing real-world performance challenges. Employing project-based learning methodologies not only strengthens technical proficiency but also prepares developers for solving complex problems in professional environments (Krajcik & Blumenfeld, 2006). Through these approaches, we build the skills necessary for delivering scalable, high-quality software solutions.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**References**

* Driessen, V. (2010). A successful Git branching model. *nvie.com*. https://nvie.com/posts/a-successful-git-branching-model/
* Dumas, M., La Rosa, M., Mendling, J., & Reijers, H. A. (2013). *Fundamentals of business process management*. Springer.
* Kumar, Vijay. *Fundamentals of Pervasive Information Management Systems*, John Wiley & Sons, Incorporated, 2013.*ProQuest Ebook Central*, https://ebookcentral.proquest.com/lib/univ-people-ebooks/detail.action?docID=1245698.
* Krajcik, J. S., & Blumenfeld, P. C. (2006). Project-based learning. In R. K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (pp. 317–334). Cambridge University Press.
* Rigby, P. C., & Storey, M. A. (2011). Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 541–550). ACM.
* Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database system concepts* (7th ed.). McGraw-Hill Education.
* Spinellis, D. (2020). *Code quality: The open source perspective*. Addison-Wesley.
* *MySQL :: MySQL 8.4 Reference Manual :: 15.8.2 EXPLAIN Statement*. (n.d.). https://dev.mysql.com/doc/refman/8.4/en/explain.html