COMPANY OPERATIONS SOP

January 2024

Prepared by

Tamanna Saini

Version

V.2024.04

Table of Contents

Company Overview

Vision

We envision a future where technology empowers businesses to achieve their full potential through innovative, scalable, and secure software solutions. Our goal is to revolutionize digital transformation by delivering intelligent, user-friendly applications that streamline operations, enhance efficiency, and foster sustainable growth. Through continuous research, cutting-edge development, and a customer-centric approach, we aim to be a global leader in software innovation, setting new benchmarks in quality, security, and reliability.

Mission

We strive to deliver cutting-edge software solutions with a commitment to innovation, quality, and customer satisfaction. Our operational processes ensure seamless software development, deployment, and maintenance while adhering to industry best practices and compliance standards.

Scope

This SOP outlines the complete workflow of software company operations, covering:

- Software Development Lifecycle (SDLC)
- Software Testing & Quality Assurance (QA)
- Deployment & Release Management
- Customer Support & Issue Resolution
- Security & Compliance
- System Maintenance & Updates
- Incident Management & Bug Tracking

Software Development Lifecycle (SDLC)

1. Planning & Requirement Gathering

- Conduct stakeholder meetings to define project scope and objectives.
- Gather business requirements from clients and internal teams.
- Conduct feasibility analysis and risk assessment.
- Document functional and non-functional requirements in detail.
- Identify the technology stack, system architecture, and required integrations.

2. Design Phase

- Develop wireframes, prototypes, and UI/UX mockups. This phase typically involves several key steps, starting with user research. This research focuses on understanding the needs of the target audience, identifying the problem the product aims to solve, and determining why the product is the most effective solution. It also includes analyzing competing products to see how they compare. After this, the process moves on to sketching the layout and features, which is essentially about organizing the information in a clear, intuitive way to ensure the product meets both user needs and functional requirements.
- Create detailed database architecture, including schema design and data flow. The company use tools like Figma and Sketch to create wireframes.
- Define API endpoints, authentication mechanisms, and security protocols. Mapping out specific URLs or paths that represent functions will be the first step in defining API endpoints. A detailed explanation about defining APIs, contact company's head developer. After defining, configure multiple API keys for data security and user privacy using token based system OAuth.
- Finalize software design specifications, including front-end and back-end structures.

3. Development & Coding

- Follow Agile methodologies such as Scrum for iterative development. It uses a cyclical approach which includes Sprint planning, Daily Scrum, Sprint Review and Sprint Retrospective.
- Maintain Product Backlogs and Sprint Backlogs on a daily basis.
- Use version control systems like Git to manage source code efficiently which allows the development team to back up and archive the source code.
- Implement coding standards and best practices to ensure code quality. The process involves 3 main steps: Defining your coding standards, communicate and train, and enforce the standards. This improves code reliability and simplifies code maintainace, and increase code wuality.
- Conduct daily stand-up meetings to discuss progress, blockers, and next steps.
- Write modular and reusable code to enhance maintainability and scalability.

Software Testing & Quality Assurance (QA)

1. Unit Testing

- Developers write unit tests for individual components using frameworks like JUnit (Java), Mocha (JavaScript), or PyTest (Python).
- Automate unit testing using continuous integration (CI) pipelines.
- 1.3 Ensure at least 80% test coverage for critical modules.

2. Integration Testing

2.1 Define the Scope of Integration Testing

- Identify the components or systems that need to be tested. This could include APIs, microservices, databases, and any other systems interacting with your application.

2.2 Create a Test Plan

- Test Cases: Develop test cases for each integration point (APIs, microservices, databases).
- Test Data: Prepare data that will be used to test these integrations (this could include sample API payloads, mock responses, and database records).
- Environment Setup: Set up a test environment that closely mirrors the production environment (e.g., staging servers, mock services for external APIs).

2.3 Verify API Integration

- End-to-End API Flow: Ensure that APIs communicate with each other correctly.
- Response Validation: Validate that the response from the API is correct, in terms of status codes (e.g., 200, 404), response time, and the structure of the returned data.
- Error Handling: Test how the API handles various types of errors (e.g., invalid requests, server errors).
- Authentication and Authorization: If applicable, verify that the API's security features (like OAuth or JWT) work as expected.
- Testing Tools: Use tools like Postman, Swagger, or custom scripts to simulate API calls and monitor responses.

Software Testing & Quality Assurance (QA)

2.4 Verify Microservices Integration

- Service Communication: Verify that microservices can communicate with each other through their respective APIs (e.g., REST, gRPC).
- Data Flow Between Services: Ensure that data flows correctly between services. For example, if one service makes a request to another service, the correct response should be returned and processed by the consuming service.
- Fault Tolerance: Test how the microservices behave when one of the services is down or unavailable.
- Service Discovery: Ensure that services can find and communicate with each other dynamically, especially in environments like Kubernetes.
- Testing Tools: Consider using tools like Docker Compose or Kubernetes to orchestrate the microservices and simulate real-world traffic.

2.5 Verify Database Integration

- Database Connections: Ensure that all services or APIs that rely on the database can connect properly.
- Data Integrity: Verify that data is correctly written to and read from the database.
- Transactions: Ensure that transactions are handled correctly across services. For example, if multiple services are involved in a transaction, they should either all succeed or fail together (e.g., using two-phase commit).
- Database Queries: Test that the queries used by the application are optimized and return the expected results.
- Database Failover: Verify that the application behaves correctly in case of database connection failures or failover scenarios.
- Testing Tools: Use database testing tools (e.g., DBUnit, Testcontainers) and SQL queries to test data handling.

2.6 Mock External Services (if needed)

- For any third-party APIs or external systems your software integrates with, use mocking techniques to simulate their behavior.
- Tools like WireMock or Mockito can help create mock responses for external services.

.

Software Testing & Quality Assurance (QA)

2.7 Execute Integration Tests

- Automated Testing: If possible, automate the integration tests to speed up the process. Use frameworks like JUnit, TestNG, or Cypress for automated integration tests.
- Manual Testing: For complex workflows or edge cases, manual testing might be needed to ensure that the integration points work as expected.

2.8 Monitor Logs and Metrics

- During testing, ensure that the application's logs and metrics are monitored to detect any issues or abnormalities.
- Check for any failed database queries, errors in API calls, or unexpected service downtime.
- Tools like Prometheus, Grafana, or ELK stack (Elasticsearch, Logstash, Kibana) can help with real-time monitoring.

2.9 Handle Edge Cases

- Consider scenarios like network failures, timeouts, or invalid input, and ensure that the system behaves gracefully (e.g., retry mechanisms, meaningful error messages).

2.10 Test Performance and Scalability (Optional)

- Ensure that the system can handle the expected load by performing stress and load testing on the APIs, microservices, and database.
- Use tools like JMeter, Gatling, or Artillery to simulate traffic and monitor system behavior under load.

2.11 Review and Analyze Results

- Analyze the test results to identify any failures or discrepancies.
- If a failure occurs, trace it back to the source, whether it's an API, microservice, or database-related issue.

2.12 Regression Testing

- After fixing issues, rerun the integration tests to confirm that the fix didn't break any other part of the system (regression testing).

Software Testing & Quality Assurance (QA)

2.13 Continuous Integration (CI) Setup

Integrate your integration tests into a CI/CD pipeline (using Jenkins, GitLab CI, or similar) to run tests automatically with each code change.

3. User Acceptance Testing (UAT)

- Deploy the application in a staging environment for client testing.
- Gather feedback from stakeholders and make necessary improvements.
- Document test cases and validation criteria for future reference.

Deployment & Release Management

1. Staging Environment Testing

- Deploy software in a controlled staging environment identical to production.
- Conduct load testing, security testing, and regression testing.
- Verify all third-party integrations and data migrations.

2. Production Deployment

- Use CI/CD pipelines for automated deployment to production environments.
- Implement rollback strategies in case of failed deployments.
- Monitor deployment logs and fix issues in real time.

3. Post-Deployment Monitoring

3.1 Define Key Metrics and KPIs

- Identify which metrics are critical to monitor in real-time. This could include:
  - System Performance: CPU usage, memory usage, disk space, network latency.
  - Application Health: Response times, error rates, transaction success/failure rates.
  - User Experience: Load times, uptime, user interactions.
  - Database Performance: Query execution times, connection counts, slow queries.
- Set up Key Performance Indicators (KPIs) based on business goals (e.g., conversion rates, user engagement).

3.2 Set Up Real-Time Alerts

- Error Alerts: Set up notifications for critical errors (e.g., 500 server errors, failed transactions, database connection failures).
- Performance Thresholds: Set thresholds for important metrics (e.g., CPU usage over 80%, response time above a certain value). Alerts can be triggered when these thresholds are exceeded.
- User Behavior: Set up alerts for unusual user behavior, such as a high number of failed login attempts or traffic spikes.
- Slack, Email, or SMS: Configure the alerts to be sent to the right teams or individuals via Slack, email, or SMS for quick action.

3.3 Enable Log Management

- Collect logs from all components of your system (e.g., application, database, server logs).
- Centralize the logs in a tool like ELK Stack or Splunk to make it easier to search and analyze in real time.
- Monitor for issues such as error messages, crashes, or performance bottlenecks.

Customer Support & Issue Resolution

1. Ticketing System

- Use platforms like Jira, Zendesk, or Freshdesk for tracking and managing customer issues.
- Categorize issues based on severity: Critical, High, Medium, and Low.

2. Tiered Support Model

- Tier 1: Basic troubleshooting such as password resets, login issues, and connectivity problems.
- Tier 2: In-depth technical support involving system settings, API configurations, and database queries.
- Tier 3: Escalation to the development team for software bugs and feature requests.

Security & Compliance

1. Data Protection

- Implement AES-256 encryption for sensitive data storage.
- Enforce multi-factor authentication (MFA) for all user logins.
- Conduct regular security audits to identify vulnerabilities.

2. Compliance Standards

- Ensure compliance with GDPR, HIPAA, ISO 27001, and other industry regulations.
- Maintain logs of user activity for auditing and security purposes.
- Regularly update privacy policies and data protection measures.

System Maintenance & Updates

1. Scheduled Maintenance

- Perform weekly and monthly system maintenance to ensure uptime.
- Update software libraries, dependencies, and security patches.
- Conduct performance optimizations to improve system efficiency.

2. Emergency Patches

- Release hotfixes for security vulnerabilities and critical bugs.
- Notify stakeholders of emergency maintenance windows in advance.
- Document patch releases and their impact on system stability.

Incident Management & Bug Tracking

1. Incident Response Plan

- Define incident severity levels and assign response teams.
- Set up incident communication channels via Slack, Microsoft Teams, or email.
- Perform root cause analysis (RCA) for major incidents.

2. Bug Tracking System

2.1 Bug Identification

- Users or IT staff report issues through the designated bug tracking tool (e.g., Jira, Bugzilla, Trello).
- The report should include:
  - Issue title
  - Description with steps to reproduce
  - Screenshots or error logs (if applicable)
  - Severity level (Critical, High, Medium, Low)

2.2 Bug Logging

- The support team logs the bug in the tracking system.
- Assigns a unique bug ID for tracking purposes.
- Classifies the issue based on:
  - Functional or non-functional bug
  - Affected system module
  - Expected vs. actual behavior

2.3 Bug Triage and Prioritization

- Project Manager and QA team assess the severity and impact of each bug.
- Priority levels:
  - Critical: Causes system failure or data loss (Immediate fix required).
  - High: Major functionality impacted (Fix within 24-48 hours).
  - Medium: Affects usability but has a workaround (Fix within a week).
  - Low: Minor UI or cosmetic issue (Fix in next scheduled update).

2.4 Bug Assignment and Fixing

- Developers are assigned bugs based on expertise.
- Root cause analysis is performed.
- Code changes and patches are developed and tested locally.

Incident Management & Bug Tracking

2.5 Code Review and Testing

- The fix undergoes peer code review.
- QA team tests the fix in a staging environment.
- Regression testing is conducted to ensure no new issues are introduced.

2.6 Deployment and Verification

- Approved bug fixes are deployed to production.
- Post-deployment monitoring ensures issue resolution.
- The bug is marked as "Resolved" once confirmed fixed.

2.7 Documentation and Closure

- All resolved issues are documented in the bug tracking tool.
- Lessons learned are recorded to prevent similar issues.
- The issue status is updated to "Closed."