

Solution Document FOR DEVSECOPS IMPLEMENTATION

APRIL 2025

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Management Information** | | | |
| Document Title: | SBIePay: SBIePay DevSecOps Implementation | | |
| Document Status: | Initial Draft | | |
| Version Number: | 1.0 | | |
| Release Date: | 03-January-2024 | | |
| Revision Details | | | |
| Version No. | Revision Date | Particulars | Approved by |
| 1.0 | 03-Jan-2024 | Initial draft | Manoj R |
| 1.1 | 28-Apr-2025 | Added additional information |  |
| **Document Contact Details** | | | |
| Role | Name | Designation | |
| Author | Sourabh Dutta | Assistant Manager Systems | |
| Inspected by/Reviewed by | Manoj R | Chief Manager Systems | |
| Approved By | Manoj R | Chief Manager Systems | |
| Distribution List | | | |

**Contents**

[1. Executive Summary 5](#_Toc196759841)

[2. Introduction 5](#_Toc196759842)

[3. Document Purpose 6](#_Toc196759843)

[4. Version Control System 7](#_Toc196759844)

[5. Comprehensive Scope of DevSecOps Solution Implementation 8](#_Toc196759845)

[6. Target Audience and Stakeholder Engagement 8](#_Toc196759846)

[7. Current Operational Landscape and Identified Gaps 8](#_Toc196759847)

[8. Strategic Goals, Objectives, and Justification for the Proposed DevSecOps System 9](#_Toc196759848)

[8.1. Strategic Intent and Necessity for DevSecOps Adoption 9](#_Toc196759849)

[8.2. Core Goals and Measurable Objectives of the Implementation 9](#_Toc196759850)

[9. Optimized Branching Model for Agile Development and Releases 10](#_Toc196759851)

[9.1. Branches in a Git Strategy 10](#_Toc196759852)

[9.2. Feature Branch 10](#_Toc196759853)

[9.3. Develop branch 10](#_Toc196759854)

[9.4. Release branch 10](#_Toc196759855)

[9.4.1. Main branch 10](#_Toc196759856)

[9.4.1.1. Branch vs Deployment Environment Diagram 11](#_Toc196759857)

[10. Critical Factors Shaping the DevSecOps Solution Design 13](#_Toc196759858)

[11. Continuous Integration and Pipelines 13](#_Toc196759859)

[11.1.1. Pipelines 15](#_Toc196759860)

[11.2. Stages 15](#_Toc196759861)

[11.2.1. Lint 15](#_Toc196759862)

[11.2.2. Build 15](#_Toc196759863)

[11.2.3. Test 15](#_Toc196759864)

[11.2.4. SAST 15](#_Toc196759865)

[11.2.5. Fortify Scan 16](#_Toc196759866)

[11.2.5.1. Packaging 16](#_Toc196759867)

[12. Continuous Deployment and Environment Management 16](#_Toc196759868)

[12.1. Continuous Deployment 16](#_Toc196759869)

[12.1.1. Helm Charts and GitOps Deployment Automation 16](#_Toc196759870)

[12.2. Environments 17](#_Toc196759871)

[12.2.1. Development Environment (DEV) 17](#_Toc196759872)

[12.2.1.1. Access 17](#_Toc196759873)

[12.2.2. SIT environment (SIT) 17](#_Toc196759874)

[12.2.2.1. Access 18](#_Toc196759875)

[12.2.2.2. Testing Environment (UAT) 18](#_Toc196759876)

[12.2.2.3. Access 18](#_Toc196759877)

[12.2.3. Pre-Prod Environment 18](#_Toc196759878)

[12.2.3.1. Access 18](#_Toc196759879)

[12.2.4. Production Environment 18](#_Toc196759880)

[12.2.4.1. Access 18](#_Toc196759881)

[13. Operational Requirements and User Roles 19](#_Toc196759882)

[13.1. Developer 20](#_Toc196759883)

[13.2. Lead Developer 20](#_Toc196759884)

[13.3. Release Manager / Architect / Bank Officials 20](#_Toc196759885)

[13.4. Stakeholders (Product Owner/Business Analyst) 21](#_Toc196759886)

[14. Tools Utilized in DevSecOps Pipeline 21](#_Toc196759887)

[15. Monitoring, Observability, and Reporting Framework 21](#_Toc196759888)

# Executive Summary

The evolution of SBIePay 2.0 is centered on embracing modern, resilient, and secure software development practices. Through the synergistic integration of Agile SCRUM methodologies and DevSecOps principles, the project aims to overhaul traditional delivery mechanisms and establish a dynamic, responsive software pipeline.

The strategic vision for SBIePay 2.0 is not merely incremental improvement; it is transformative. By embedding security controls ("DevSec") and automated quality gates early in the software lifecycle ("shift-left" principle), SBIePay 2.0 will significantly reduce the risk of vulnerabilities reaching production environments. Continuous Integration (CI) and Continuous Deployment (CD) pipelines will accelerate time-to-market, while robust monitoring and feedback loops will ensure that quality and performance standards remain uncompromised.

The overarching goal is to foster a culture of collaboration between Development, Security, and Operations teams, breaking down traditional silos to create a seamless, transparent, and accountable software delivery environment. With automation at its core and security as a foundation, SBIePay 2.0 aims to achieve enhanced agility, resilience, operational efficiency, and superior user satisfaction.

# Introduction

SBIePay 2.0 represents a paradigm shift in how software is conceived, built, tested, secured, and delivered. This transformation journey moves away from traditional, linear, and fragmented approaches towards a holistic, iterative, and collaborative model that places automation and security at the forefront.

The DevSecOps model fosters an environment where code is continuously integrated, tested, and secured, significantly reducing bottlenecks and vulnerabilities. Instead of treating security as a post-development checkbox, it becomes an intrinsic element of the development process. Agile SCRUM practices ensure that project milestones are met through iterative sprints, enabling continuous feedback from stakeholders and users.

In this new operating model, the integration of automation tools — including static code analysis, dynamic security testing, automated deployments, and real-time monitoring — ensures that SBIePay can rapidly adapt to business changes, regulatory requirements, and security threats without sacrificing quality or compliance. This introduction sets the context for the comprehensive strategies that will follow in the document.

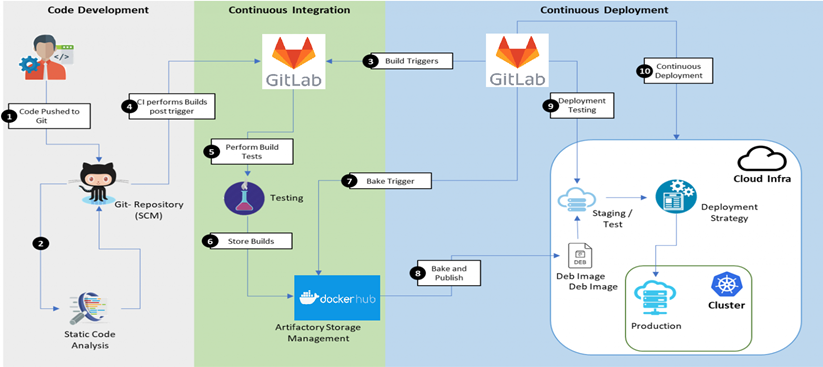
# Document Purpose

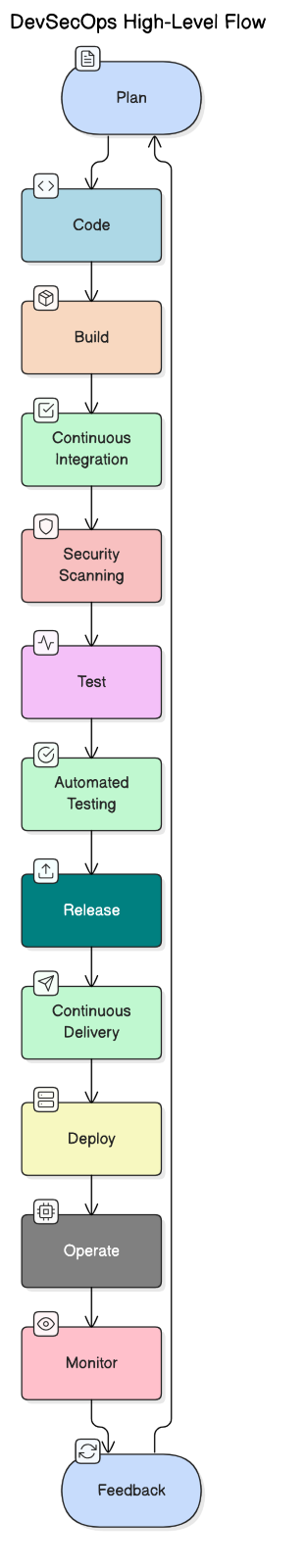
This document serves as a comprehensive guide for the design, implementation, and management of DevSecOps practices within the SBIePay 2.0 program. Its primary objective is to define and standardize how security, development, operations, and quality assurance will integrate to deliver a high-performing, resilient, and secure software platform.

The document outlines the foundational principles, methodologies, tooling architecture, operational procedures, environment structures, and governance models that will drive the success of SBIePay 2.0. It targets a broad audience — from developers and QA engineers to release managers, product owners, and compliance officers — ensuring that each stakeholder understands their responsibilities, interfaces, and critical deliverables.

Furthermore, this document provides guidance on establishing continuous integration and continuous deployment pipelines, implementing security at every stage of the software lifecycle, and building an ecosystem of tools that support scalable and maintainable development practices. It will serve as the reference manual for current and future project members to align efforts, maintain consistency, and drive continuous improvement.

The following is a Reference diagram:





# Version Control System

A solid version control strategy is the backbone of any modern software development initiative. For SBIePay 2.0, GitLab serves as the central platform for managing all source code repositories, enabling distributed development, code collaboration, and seamless integration with CI/CD workflows.

The strategic framework for version control is designed to promote structured, scalable, and secure code management. Developers will work within isolated feature branches, promoting cleaner integration processes and reducing the risk of conflicts. Merge requests (MRs) with mandatory peer reviews and automated quality checks will be the standard mechanism for integrating changes into the main development branches, ensuring that only thoroughly validated code advances through the pipeline.

Critical elements of this framework include:

* **Branching Models**: Feature, develop, release, and main branches with strict role-based access controls.
* **Commit Hygiene**: Clear, descriptive commit messages linked to issue trackers for traceability.
* **Tagging and Versioning**: Standardized tagging practices to mark production-ready releases.
* Code Reviews and Approvals: Enforced code review policies to ensure high-quality, secure code.

In addition, GitLab’s integration with security scanning, artifact management, and monitoring tools ensures that the version control system is not just a place for code storage but a living system that supports secure, automated, and transparent development processes.

# Comprehensive Scope of DevSecOps Solution Implementation

The solution encompasses:

* Setting up CI/CD pipelines integrated with security scanning tools.
* Automating functional, performance, and security testing.
* Implementing monitoring and alerting frameworks.

Deploying applications to multiple environments: DEV, SIT, UAT, Pre-Prod, Prod.

# Target Audience and Stakeholder Engagement

The document is intended for:

* Project Management Teams
* Business Owners
* Development and QA Teams
* DevOps and Security Teams
* Release Managers
* Compliance and Regulatory Authorities

By catering to a wide audience, it ensures that strategic goals, technical implementations, and operational responsibilities are clearly understood and actionable.

# Current Operational Landscape and Identified Gaps

The existing SBIePay system suffers from manual processes, siloed teams, delayed deployments, and minimal proactive security practices. This increases the risk of late defect detection, security vulnerabilities, and unstable releases.

Gaps identified include:

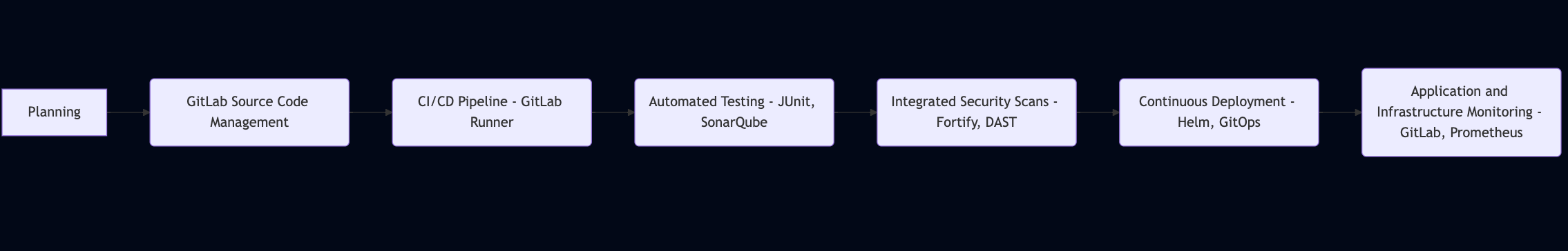
* No early-stage security testing
* Manual deployment errors
* Lack of automated quality checks
* Limited system observability

The proposed DevSecOps adoption aims to eliminate these issues, creating a secure, efficient, and agile operational environment.

# Strategic Goals, Objectives, and Justification for the Proposed DevSecOps System

The DevSecOps implementation seeks to achieve:

* Seamless integration of security across the SDLC.
* Continuous integration, testing, and deployment automation.
* Faster, more reliable, and secure software releases.
* Improved system observability and proactive issue detection.



This ensures SBIePay 2.0 aligns with modern regulatory requirements, market agility demands, and cybersecurity best practices.

## Strategic Intent and Necessity for DevSecOps Adoption

Embedding security early and automating the entire software lifecycle enables a "security-first" culture. This drastically reduces vulnerabilities and ensures compliance from inception, making software more robust, reliable, and trustworthy.

Given the evolving cyber-threat landscape and regulatory frameworks, adopting DevSecOps is a strategic necessity for any mission-critical financial application like SBIePay.

## Core Goals and Measurable Objectives of the Implementation

Key initiatives include:

* Automated Code Quality Scans: With Semgrep, Gitlab
* Automated Unit/System Testing: Using JUnit
* Continuous Integration: Via GitLab CI
* Continuous Deployment: Via GitLab CD
* Automated Security Testing: SAST with Fortify; DAST via GitLab
* Monitoring & Alerting: Leveraging GitLab observability features

Success will be measured through KPIs like build success rate, defect escape rate, security vulnerability counts, and deployment frequency.

# Optimized Branching Model for Agile Development and Releases

The project adopts a Git branching model that utilizes multiple branches to manage the flow of code from development through to production. This strategy is particularly well-suited for teams operating with scheduled release cycles, allowing features to be grouped into defined releases. Development occurs in isolated feature branches, which are then integrated into a central develop branch after review and approval. This develop branch serves as the integration point and represents code being prepared for release. Once all features planned for a release are stable in the develop branch, a release branch is created to manage deployments to higher environments like SIT, UAT, and Pre-prod, production.

## Branches in a Git Strategy

A Git branching strategy commonly has the following branches:

## Feature Branch

Feature branches are short-lived branches where developers develop features. The feature branch is created by branching off of the develop branch. Developers iterate, commit, and test code in the feature branch. When the feature is complete, the developer promotes the feature. There are only one path forward from a feature branch:  
Creating a merge request into the develop branch

|  |  |
| --- | --- |
| **Naming convention** | Feature/<repo initials>-<story number>\_\_<descriptor> |
| **Naming convention example** | Feature/JS-123456\_FeatureA |

## Develop branch

The develop branch is a long-lived branch where features are integrated, built, validated, and deployed to the development environment. All feature branches are merged into the develop branch. Merges into the develop branch are completed through a merge request that requires a successful build and developers’ approvals.

**Naming convention**: develop

## Release branch

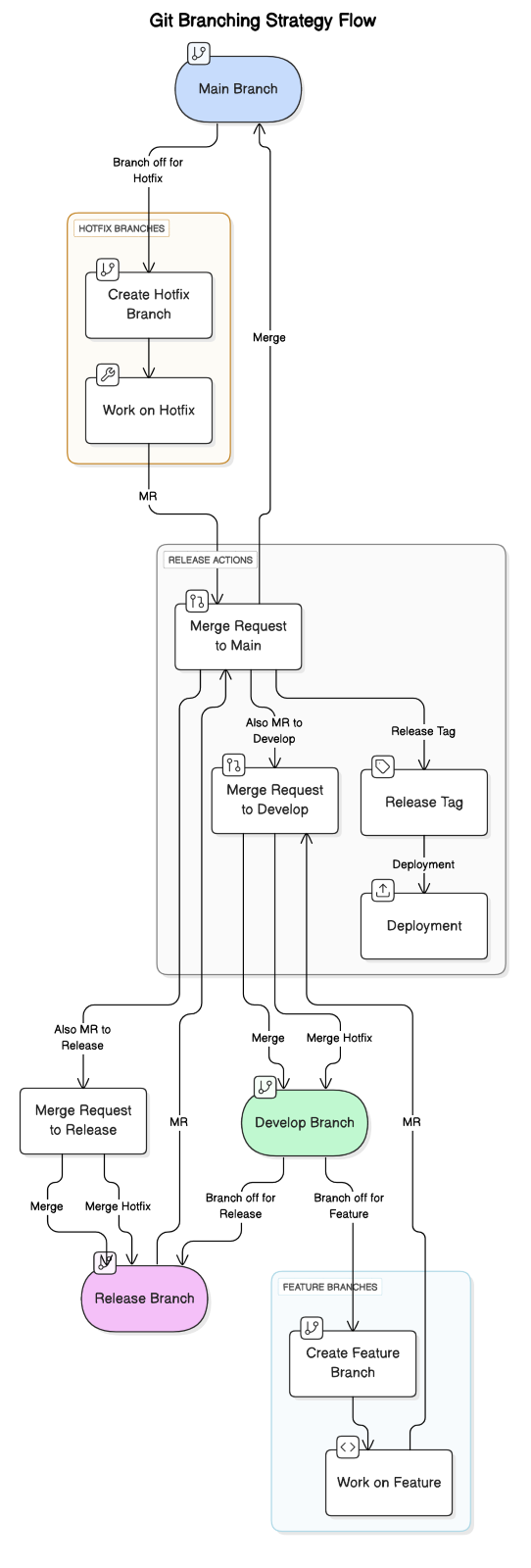
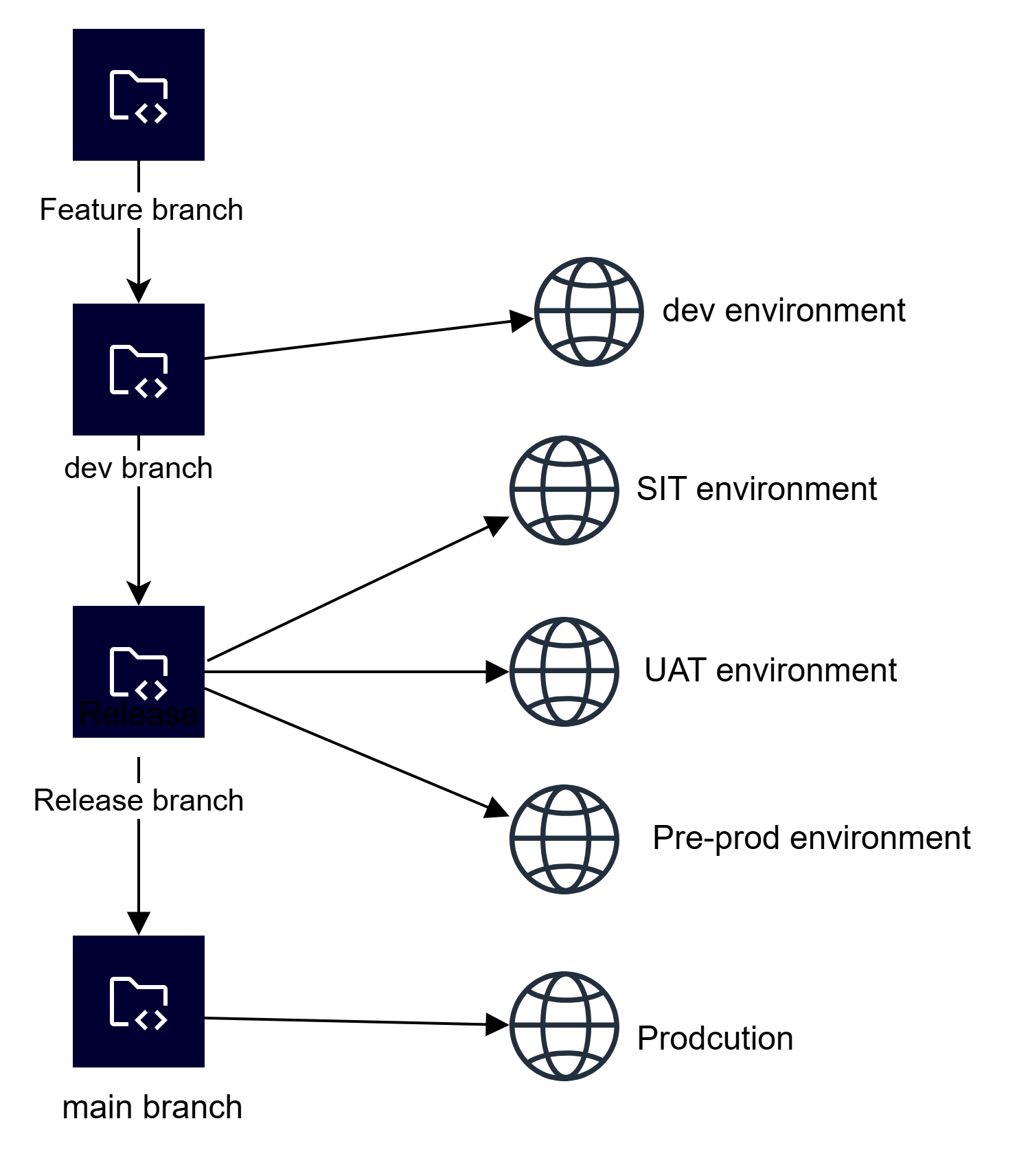
In GIT, release is long lived branch. Once code is qualified in development branch, code can be promoted to release branch. And release branch code is deployed in SIT, UAT and Pre-prod.

|  |  |
| --- | --- |
| **Naming convention** | release/v{major}.{minor} |
| **Naming convention example** | release/v1.0 |

### Main branch

The main branch is a long-lived branch that always represents the code that is running in production. Code is merged into the main branch automatically from a release branch after a successful deployment from the release pipeline. To prevent deletion, enable branch protection on the main branch.  
**Naming convention**: main

#### Branch vs Deployment Environment Diagram



# Critical Factors Shaping the DevSecOps Solution Design

Key factors influencing the solution design:

* Adherence to Bank IT Security and Compliance Policies
* Compatibility with the SBIePay 2.0 technology stack (Java, React, Openshift)
* Need for full automation across build, test, deploy cycles
* Scalability for future feature and user base growth
* Cost-effectiveness through optimal tool selections

Each decision — from tool adoption to process design — is mapped against these strategic factors to ensure long-term project sustainability.

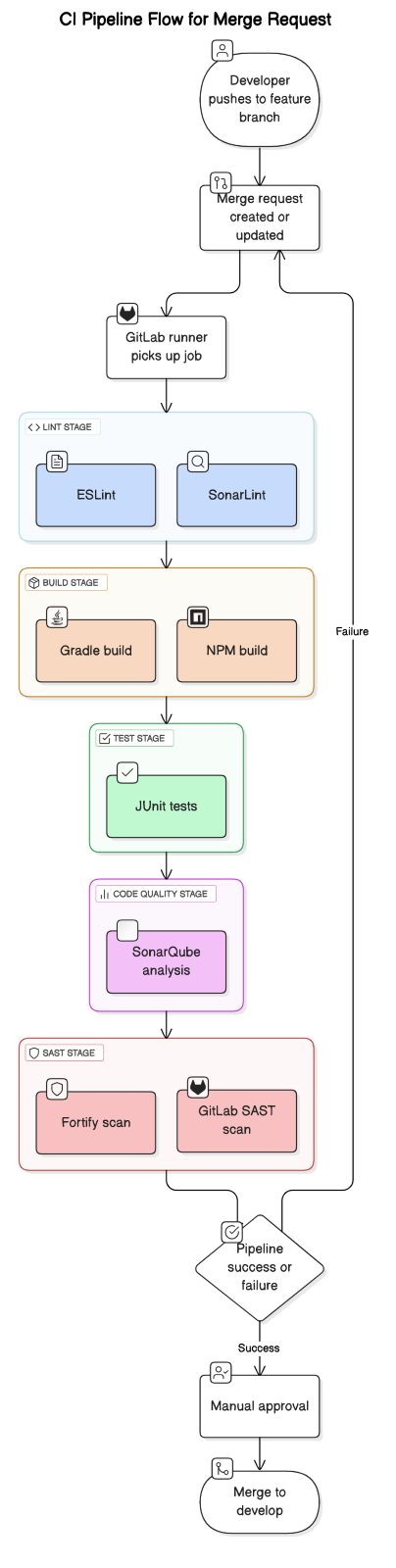
# Continuous Integration and Pipelines

Continuous Integration (CI) involves automatically building, testing, and integrating code changes into a shared repository several times a day. For SBIePay 2.0, GitLab CI pipelines orchestrate builds, unit tests, linting, SAST scans, and artifact packaging.

**Stages in CI Pipeline:**

* **Linting**: Code syntax verification (ESLint, YAML Linter)
* **Building**: Compiling Java (Gradle) and ReactJS (NPM)
* **Testing**: Running JUnit and automated system tests
* **SAST**: Code security scanning with GitLab Advanced SAST and Fortify
* **Packaging**: Docker image creation and version tagging

Every commit triggers the CI pipeline ensuring rapid feedback loops and early defect detection.



### Pipelines

The core of GitLab’s CI/CD pipeline is the .gitlab-ci.yml file, which is placed at the root of the repository. This file defines the CI pipeline configuration, specifying the jobs, stages, and runners involved in the CI process.  
Key Elements of .gitlab-ci.yml:

* Stages: Defines the order of the pipeline. Stages run sequentially by default (e.g., build, test, deploy).
* Job: Each job corresponds to a unit of work, typically a single task such as compiling, testing, or deploying. Jobs are defined by a name (e.g., build\_job, test\_job).
* Script: This specifies the commands that are executed in each job. For example, npm install, npm test, etc.

## Stages

### Lint

A Lint Stage in CI refers to a stage in your .gitlab-ci.yml file (or other CI configuration files) that runs a linter against your source code to analyze it for problems. Linting can be done for various languages and tools, such as JavaScript, Python, YAML, CSS, etc.

* JavaScript/TypeScript: ESLint
* YAML: YAML linter
* JAVA: Sonarlint

### Build

The Build Stage is responsible for transforming your source code into a deployable artifact. During the build stage, the code is compiled, packaged, and prepared for deployment or further testing. This stage typically follows the linting and testing stages (if applicable) and is one of the most critical stages to ensure that your code can be successfully compiled and run in the target environment.

* Java: Gradle
* JavaScript/TypeScript: NPM
* Container Image: Podman

### Test

The Test Stage in a pipeline is a crucial part of the Continuous Integration (CI) process, where automated tests are executed to verify that the code behaves as expected. This stage helps catch bugs, regressions, and unexpected behavior early in the development cycle, ensuring that only high-quality code progresses through the pipeline.

### SAST

The SAST (Static Application Security Testing) stage in a CI/CD pipeline is crucial for identifying security vulnerabilities in the source code during the development process. SAST tools analyze the application's source code, bytecode, or binaries without executing it, to detect potential security issues like SQL injection, cross-site scripting (XSS), and buffer overflows, among others. This helps ensure that security vulnerabilities are identified early, before the code reaches production.

* Gitlab SAST using semgrep & Advanced Gitlab SAST

### Fortify Scan

#### Packaging

* Versioning
* Tagging with version – on commit version id is created.
* v0.X.X minor version increment for develop branch

e.g., v0.0.70

v.X.X major version increment for release

* Packaging
  + Build Docker Image, version number will be used to for image tag.
* Push to Non-Prod Container Registry for develop or release
* Deploy - Trigger CD Pipeline through merge request approved
* Development environment for develop branch.

SIT / UAT / Pre-prod environment for release branch.

# Continuous Deployment and Environment Management

Continuous Deployment (CD) automates the delivery of software across different environments (DEV, SIT, UAT, PRE-PROD) following successful CI validations.

Environments:

* Development (DEV): Daily integration builds and unit testing
* System Integration Testing (SIT): End-to-end functional validation
* User Acceptance Testing (UAT): Stakeholder validation
* Pre-Production (PRE-PROD): Performance, load testing, final approvals

Each environment has a controlled promotion mechanism ensuring stability and compliance before progressing.

## Continuous Deployment

Continuous Deployment (CD) is an extension of Continuous Integration (CI) where every change that passes automated tests is deployed (Manually) to Dev / SIT / UAT / Pre-prod. In GitLab, Continuous Deployment integrates seamlessly with GitLab CI/CD, allowing teams to automate the entire pipeline from development through to production.

### Helm Charts and GitOps Deployment Automation

Helm charts are used to package Kubernetes applications, simplifying deployments. Deployment manifests are version-controlled in Git repositories. Using GitOps practices:

* Code merge into the infrastructure repo automatically triggers Helm updates.
* Git becomes the single source of truth for deployments.
* Manual approval gates are configured for production releases ensuring final compliance validation.

This method guarantees deployment consistency, rollback capabilities, and operational transparency.

## Environments

SBIePay 2.0 has the following four common environments that span our development pipeline.

1. Development (DEV) – An environment where developers integrate their code to confirm that it all works as a single, cohesive application.
2. Development (SIT) – An environment where stable code will be deployed from release branch.
3. Testing (UAT) – An environment where BANK UAT teams or acceptance testing takes place. Teams often do performance or integration testing in this environment.
4. Pre-prod – A pre-production environment where you validate that the code and infrastructure perform as expected under production-equivalent circumstances. This environment is configured to be as similar as possible to the production environment.

This section describes each environment in detail. It also describes the deployment steps and exit criteria for each environment so that you can promote deployment to the next environment. The following image shows these environments in sequence.

### Development Environment (DEV)

The development environment is where developers integrate their code together to ensure it all works as one cohesive application. In Git, the development environment contains the latest features included by merge request and are ready for release. The development environment is considered to be a testing environment (Unit Test), and the code base might be unstable and unsuitable for deployment to production.

#### Access

Assign permissions according to the principle of least privilege. Least privilege is the security best practice of granting the minimum permissions required to perform a task. Developers have full access to the development environment.

Expectations before moving to the development environment

* Perform static code analysis

### SIT environment (SIT)

An environment where stable code will release from dev environment. In Git, the development environment contains the latest features included by merge request. The development environment is considered to be a testing environment (Internal Testing), and the code base might be stable and suitable for deployment to UAT.

#### Access

Assign permissions according to the principle of least privilege. Least privilege is the security best practice of granting the minimum permissions required to perform a task. Developers have full access to the development environment.

Expectations before moving to the development environment

* Perform static code analysis

#### Testing Environment (UAT)

Quality assurance (QA) personnel use the testing environment to validate features. They approve the changes after they finish testing. When they approve, the branch moves on to the next.

#### Access

Assign permissions according to the principle of least privilege. Developers should have less access to the testing environment than they have to the development environment. QA personnel require sufficient permissions to test the feature.

Expectations before moving to the Pre-Prod environment

1. The development and QA teams have performed sufficient testing to satisfy Business requirements.
2. The development team has resolved any discovered bugs.

### Pre-Prod Environment

The Pre-Prod environment is configured to be the same as the production environment~~.~~ For example, the data setup should be similar in scope and size. Use the Pre-Prod environment to verify that code and infrastructure operate as expected. This environment is also the preferred choice for business use cases, such as previews or customer demonstrations.

#### Access

Assign permissions according to the principle of least privilege. Developers should have the same access to the Pre-Prod environment as they do the production environment.

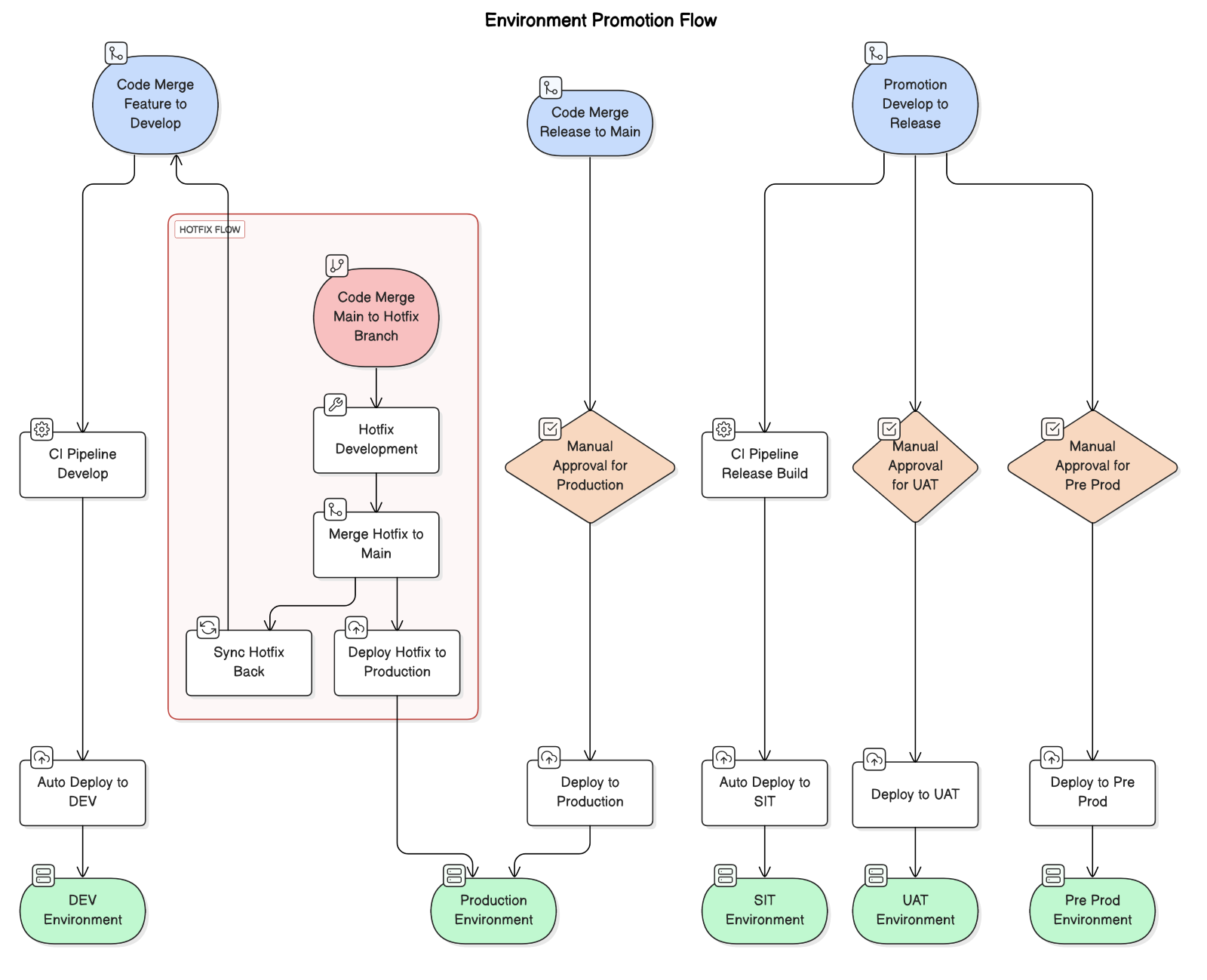
Expectations before moving to the production environment.

1. A production-equivalent release has been deployed successfully to the Pre-Prod environment
2. (Optional) Integration and load testing were successful

### Production Environment

#### Access

In the production environment, developers should have limited, read-only access in the Management Console. For example, developers should be able to access log data for day-to-day operations. All releases to production should be gated by an approval step prior to deployment.



# Operational Requirements and User Roles

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User Role** | **Git Lab Role** | **Branch Access** | **Responsibilities** | **Work Breakdown** |
| Developer | Developer | * feature/\* | |  | | --- | | * Create and work on feature branches * Commit code regularly * Ensure feature completion * Address urgent issues in hotfix | | 1. Create feature branch from develop. 2. Implement feature. 3. Push code to feature branch. 4. Fix critical issues and merge into hotfix |
| Team Lead | Maintainer | * Feature/\* * Develop * Release | * Review and approve feature branches and mergers to develop * Coordinate with UAT for bug fixes | 1. Review feature merges. 2. Ensure code quality. 3. Assist in resolving conflicts during merges. |
| QA/UAT Tester | Reporter | * Develop * Release | |  | | --- | | * Test features in the develop / UAT branch * Report bugs and issues | | 1. Validate features against requirements. 2. File issues. 3. Retest after bug fixes. |
| Compliance Team | Guest or Reporter | Develop | * Validate compliance and regulatory requirements * Approve UAT releases | 1. Test compliance-specific requirements. 2. Report non-compliance issues. 3. Approve once all checks pass. |
| DevOps Engineer | Maintainer | * Main * Release * Develop | * Monitor production releases. * Apply hotfixes, if necessary. | 1. Deploy main to production. 2. Merge hotfixes into develop and release. |

## Developer

* + Role: Responsible for developing new features, fixing bugs, and improving code.
  + Flow:
    - **Clone from Develop Branch**: Developers start by cloning the Develop branch to create their own Feature branch.
    - **Feature Development**: They develop features or make changes in the Feature branch in isolation.
    - **Daily Sync with Develop**: Developers pull the latest updates from the Develop branch to ensure their feature branch stays in sync with the ongoing work.
    - **Code Review (MR):** Once the development is done, the developer creates a Merge Request (MR). This PR goes through a two-level review process to ensure code quality and alignment with standards.
    - **Merge to Develop**: After approval, the feature branch is merged into the Develop branch.

## Lead Developer

* + **Role**: Oversees code reviews, ensures integration quality, and manages UAT processes.
  + **Flow**:
    - **Review Merge Requests**: Lead developers or designated reviewers perform code reviews at multiple levels before merging features into the Develop branch.
    - **Merge to UAT**: Once the Develop branch has accumulated multiple features or changes, the Lead Developer merges the Develop branch into the release branch for testing purposes.
    - **Coordinate UAT Testing**: The QA Engineer or Test Lead works closely with developers during the User Acceptance Testing (UAT) phase to ensure functionality meets business requirements.
    - **Fix UAT Bugs**: Any issues discovered during UAT are fixed in the UAT branch, with the fixes being validated and approved before proceeding.

## Release Manager / Architect / Bank Officials

* + **Role**: Manages the release process and ensures smooth transition from UAT to production.
  + **Flow**:
  + **Merging to Master**: Once the UAT phase is completed successfully and no critical bugs remain, the Release Manager merges the UAT code into the Master branch.
  + **Merging to Release**: Before an official release, the code from Master is merged into the Release branch.
  + **Create Release Build**: A release build is generated from the Release branch, which is then deployed to production.

## Stakeholders (Product Owner/Business Analyst)

* **Role:** Focus on defining the requirements and verifying them during UAT.
  + **Flow**:
  + **Feature Requests**: They initiate new feature development by working with developers and defining business requirements.
  + **Develop**: Different features are merged to develop branch. Once build is success, QA validate features to validate developed features meets requirement.
  + **Release**: Validated code is merged to release branch from develop branch. Code in release branch is promoted to SIT / UAT / Pre-prod environment.
  + **UAT Feedback**: After code is moved to UAT, stakeholders perform acceptance testing to validate that the developed features meet the business needs and requirements.

# Tools Utilized in DevSecOps Pipeline

|  |  |  |
| --- | --- | --- |
| **Tool** | **Purpose** | **Version** |
| GitLab Server & Runner | Source Code Management & Pipeline Execution | 17.2.x |
| Gradle | Java Build Tool | 8.9 |
| NPM | ReactJS Build Tool | 10.9.0 |
| ESLint | JavaScript Syntax Checking | 9.11.1 |
| Fortify | Static Application Security Testing (SAST) | 23.1.0 |
| GitLab Advanced SAST/DAST | Security Vulnerability Scanning | 1.0.21 |
| OpenShift + Helm | Container Orchestration & Deployment | 4.14.29 |
| Podman | Container Image Management | 4.9.4 |

# Monitoring, Observability, and Reporting Framework

Monitoring is achieved through integration with GitLab, Prometheus, Openshift’s Logging monitoring, Grafana, and Loki Stack

The following are the capabilities:

* Real-time system and application health dashboards
* Alerts on anomalies or performance bottlenecks
* Logging with structured log management tools (e.g., Vector)
* Compliance and audit reporting

Automated dashboards provide transparency across development, QA, security, and operations teams.