SBIePay DevSecOps Implementation

Solution Document

December 2024

Document Management Information			
Document Title:	SBIePay: SBIePay DevSecOps Implementation		
Document Status:	Initial Draft		
Version Number:	1.0		
Release Date:	21-Decemeber-2024		
Revision Details			
Version No.	**Revision Date**	**Particulars**	**Approved by**
1.0	21-DEC-2024	Initial draft	Manoj R
Document Contact Detail s			
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

State Bank of India, the largest bank in India, has consistently been at the forefront of adopting new technologies and initiatives. The bank has decided to revamp the current SBIePay system to SBIePay 2.0, aiming to streamline the merchant on boarding process and introduce new features that are competitive in the current market. This comprehensive upgrade involves migrating both the front-end and back-end systems in terms of design and technical stack. We are transitioning from a monolithic architecture to a microservices-based architecture, which will be scalable and cloud-deployable. Additionally, DevSecOps practices will be implemented to ensure secure and efficient delivery.

2. Introduction

To deliver SBIePay 2.0, we will adopt the Agile SCRUM methodology in conjunction with DevSecOps practices. The SCRUM methodology will enable iterative development, early feedback, and incremental delivery, fostering enhanced collaboration between Business Owners and Development teams.

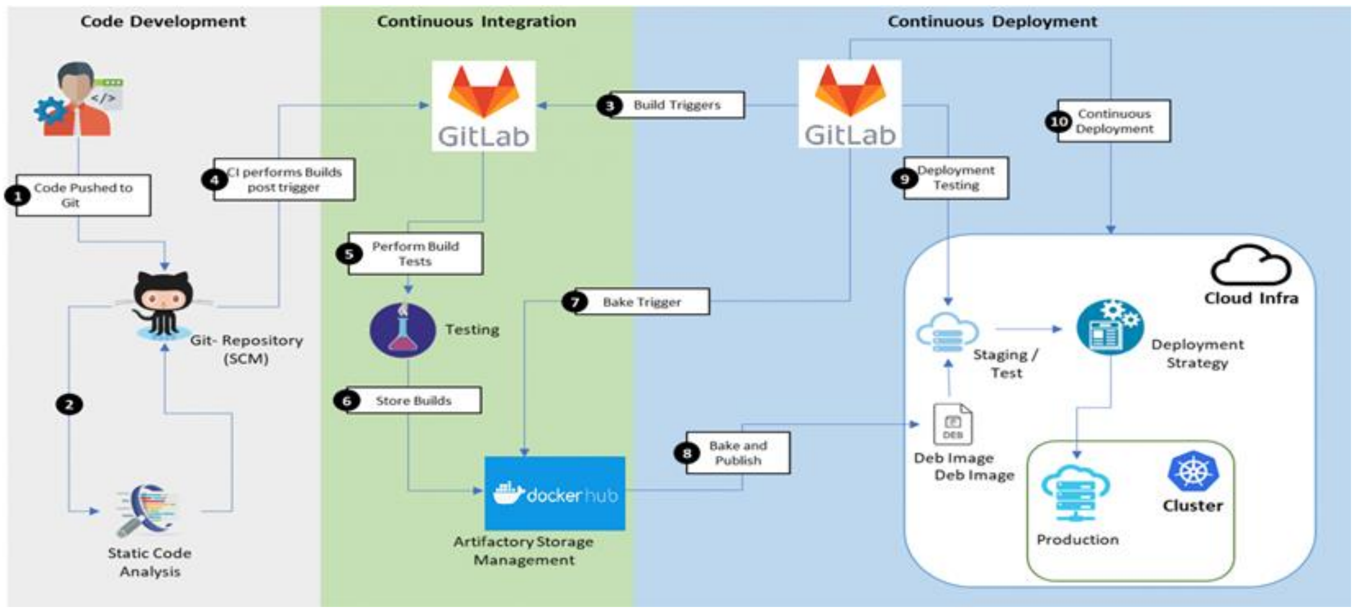
DevSecOps practices will ensure rapid development through automation at every stage of the development, testing, and deployment processes. This approach integrates quality and security practices early in the lifecycle, thereby accelerating the development, testing, and deployment of new functionalities and features.

Our goal is to achieve comprehensive end-to-end automation by leveraging the DevSecOps tool chain. This includes automated static code quality scans, continuous integration, automated functional and non-functional testing, security tests using a shift-left approach, and automated deployment. By implementing these practices, we aim to enhance efficiency, ensure robust security, and deliver a high-quality product.

3. Document Purpose

This document outlines the DevSecOps practices that will be implemented to ensure the secure and efficient delivery of SBIePay 2.0.

The purpose of this document is to outline the design for CI & CD. This will include a view of the high-level approach and the breakdown of the internal various DevOps processes. Below is a reference diagram:



4. Version Control System

GitLab is a web-based DevOps platform that provides a range of features for source code management (SCM) using Git, and it goes far beyond that to support the entire software development lifecycle. It's built around Git as its version control system (VCS), which is a distributed version control tool that tracks changes to source code over time.

GitLab and its capabilities in version control areas below:

4.1.Git as the Core Version Control System

- **Distributed Version Control**: GitLab relies on Git, a distributed version control system. Git allows developers to work on code locally (in their own copies or "repositories") and then sync their changes with a central server when ready.
- **Branching & Merging**: Git provides powerful branching and merging capabilities, allowing multiple developers to work on separate features, fixes, or experiments in parallel and merge them later.
- **Commit History**: Git tracks every change made to the codebase. GitLab offers a graphical interface to view commits, branches, tags, and diffs.

4.2.GitLab Features for Version Control and Collaboration

- **Repositories**: In GitLab, a repository (repo) holds all your project's files and the entire history of changes. Developers can clone repositories, create branches, make changes, and push updates back to the repository.

- **Merge Requests (MR)**: GitLab has a strong focus on **merge requests** (similar to GitHub's pull requests), where changes made in a feature branch can be reviewed before being merged into the main codebase. This is a key feature for collaboration and ensuring code quality.
- **Code Review**: With GitLab’s MR functionality, code review is facilitated. Developers can comment on specific lines of code, approve, or request changes to the code being merged.
- **Issue Tracking**: GitLab integrates issue tracking with its version control system, so you can associate commits and MRs with specific issues, bugs, or feature requests.

4.3.CI/CD Integration

- GitLab’s integration with **Continuous Integration (CI)** and **Continuous Deployment (CD)** is a big strength. When you push changes to a repository, GitLab can automatically trigger pipelines that build, test, and deploy your code.
- **GitLab CI/CD** allows you to define workflows in a .gitlab-ci.yml file, where you can specify scripts and commands to be run at different stages of the pipeline (build, test, deploy, etc.).

4.4.Collaboration & Visibility

- **Collaborative Coding**: GitLab enables teams to collaborate on code by sharing repositories, reviewing MRs, and tracking progress through issues and milestones.
- **Visibility and Permissions**: GitLab supports configurable user roles and permissions. You can grant access to users based on their role (e.g., Developer, Maintainer, Guest).
- **Activity & Audit Logs**: GitLab keeps track of all activity, so you can monitor changes, commits, merges, and comments.

4.5.Advanced Version Control Features

- **Tagging**: GitLab supports Git tags, allowing you to mark specific points in the repository’s history (e.g., releases or milestones).
- **Forking & Collaboration**: GitLab allows users to fork repositories, create their own branches, make changes, and submit those changes as merge requests to the original repository.
- **Conflict Resolution**: When merging branches, GitLab provides tools to help resolve any merge conflicts that may arise.

4.6.Self-Hosted

- **Self-Hosted GitLab**: GitLab can be installed on your own servers, providing full control over the repository and project data.

4.7.Integration with Other Tools

- GitLab integrates with a variety of third-party tools for project management, issue tracking, monitoring, and more. It can be connected with services like Slack, JIRA, Kubernetes, Docker, etc.

4.8.Security and Compliance

- GitLab includes built-in security scanning tools (SAST, DAST, dependency scanning) as part of the CI/CD pipelines, which help identify vulnerabilities in your code and dependencies.
- **Code Quality Reports**: It generates code quality reports that can be integrated into the pipeline to ensure the highest standards of code hygiene.

4.9.GitLab for Enterprises

- GitLab is scalable and suitable for both small teams and large enterprises. With advanced features such as multi-project pipelines, multiple group-level repositories, and enhanced security features, it's designed to support complex, large-scale development workflows.

5. Document Scope

This document includes the DevSecOps practices and the tools that will be implemented in SBIePay 2.0 program.

6. Document Audience

The intended audience of the document are Project Management team, Business Owners, Development team, QA/Testing team, DevOps teams, Development Leads, ISD team.

7. Current state / System

This is not applicable as this is a completely new DevSecOps implementation.

The goal of adopting DevSecOps practices is to facilitate a more iterative, rapid and secure delivery process.

8. Goals, Objectives and Rationale for Proposed System

8.1. Purpose of proposed system

The implementation of DevSecOps practices aims to facilitate iterative, rapid, and secure delivery.

8.2. System Goals and Objectives

The implementation of DevSecOps practices aims to facilitate iterative, rapid, and secure delivery. The solution is designed to facilitate the regular delivery of small, incremental functionalities rather than large batches at extended intervals. It integrates security at every stage of the development cycle. The proposed DevSecOps toolset aims to support and automate the entire delivery cycle, including:

8.2.1. Automated code quality scans

8.2.2. Automated unit and system testing

8.2.3. Continuous integration

8.2.4. Automated deployment

8.2.5. Automated security testing (SAST in Fortify & DAST via Gitlab)

8.2.6. Performance testing

8.2.7. Monitoring

By implementing these practices, the solution ensures a streamlined, secure, and efficient development process.

8.3. System Scope

8.3.1. In Scope:

The DevSecOps implementation for SBIePay 2.0 encompasses the following activities:

a) Establishing the DevSecOps Tool chain: Setting up a comprehensive DevSecOps tool chain on the Bank's provided infrastructure to support the seamless delivery of SBIePay 2.0.

b) Tool Implementation: Deploying essential tools for product planning, development, testing, and deployment to ensure a streamlined workflow.

c) Pipeline Configuration: Building and configuring robust integration, testing, and deployment pipelines to facilitate continuous delivery and integration.

d) Automated Security Testing: Implementing automated security testing using Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools to identify and mitigate vulnerabilities early in the development cycle.

e) Near-Zero Downtime Deployments: Ensuring deployments are executed with minimal to no downtime, enhancing the reliability and availability of the system.

8.3.2. Out of scope:

a. Migration of SBIePay 1.0 Test Automation scripts

8.4. Business Process Supported

The DevSecOps tools set supports the software development lifecycle.

8.5. High level Functional Requirements

The DevSecOps solution facilitates iterative development and delivery of the product. The DevSecOps toolset automates various software delivery activities, including:

1. Application Lifecycle Management: Streamlining the entire lifecycle from planning to deployment.
2. Source Code Management: Efficiently managing and versioning source code.
3. Static Code Quality Scans: Ensuring code quality and security through automated scans.
4. Functional Testing: Automating tests to verify the functionality of the application.
5. Continuous Integration: Integrating code changes continuously to detect issues early.
6. Artifact Management: Managing build artifacts efficiently via Gitlab.
7. Browser and Device Testing: Ensuring compatibility across different browsers and devices.
8. Security Testing: Conducting comprehensive security tests, including SAST, DAST using Gitlab and container image scans.
9. Performance Testing: Assessing the application's performance under various conditions.
10. Deployment: Automating the deployment process for consistency and reliability.
11. Log Monitoring: Continuously monitoring logs for anomalies and issues.
12. Application and Infrastructure Monitoring: Keeping track of application and infrastructure health.
13. Reports & Dashboards: Providing insights and visibility through detailed reports and dashboards.

9. Branching Strategy

Git is a branching model that involves the use of multiple branches to move code from development to production. Git works well for teams that have scheduled release cycles and a need to define a collection of features as a release. Development is completed in individual feature branches that are merged, with approval, into a development branch, which is used for integration. The features in this branch are considered ready for production. When all planned features have accumulated in the develop branch, a release branch is created for deployments to upper environments.

9.1. Branches in a Git strategy

A Git branching strategy commonly has the following branches:

9.1.1. Feature branch

Feature branches are short-term branches where you 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

9.1.2. 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 two developer approvals. To prevent deletion, enable branch protection on the develop branch.

Naming convention: **Develop**

9.1.3. Release branch

In Git, release branches are short-term branches. These branches are special because you can deploy them to multiple environments, embracing the build-once, deploy-many methodology. Release branches can target the testing, Pre-Prod, or production environments. After a development team has decided to promote features into higher environments, they create a new release branch and use increment the version number from the previous release. At gates in each environment, deployments require manual approvals to proceed. Release branches should require a merge request to be changed.

After the release branch has deployed to production, it should be merged back into the develop and main branches to make sure that any bugfixes or hotfixes are merged back into future development efforts.

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

9.1.4. 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**

9.1.5. Bugfix branch

The bugfix branch is a short-term branch that is used to fix issues in release branches that haven't been released to production. A bugfix branch should only be used to promote fixes in release branches to the testing, Pre-Prod, or production environments. A bugfix branch is always branched off of a release branch.
After the bugfix is tested, it can be promoted to the release branch through a merge request.
Then you can push the release branch forward by following the standard release process.

Naming convention:	bugfix/ <repo initials>_<story number>__<descriptor>
Naming convention example:	Bugfix/JS_123456_Fix_Problem_A

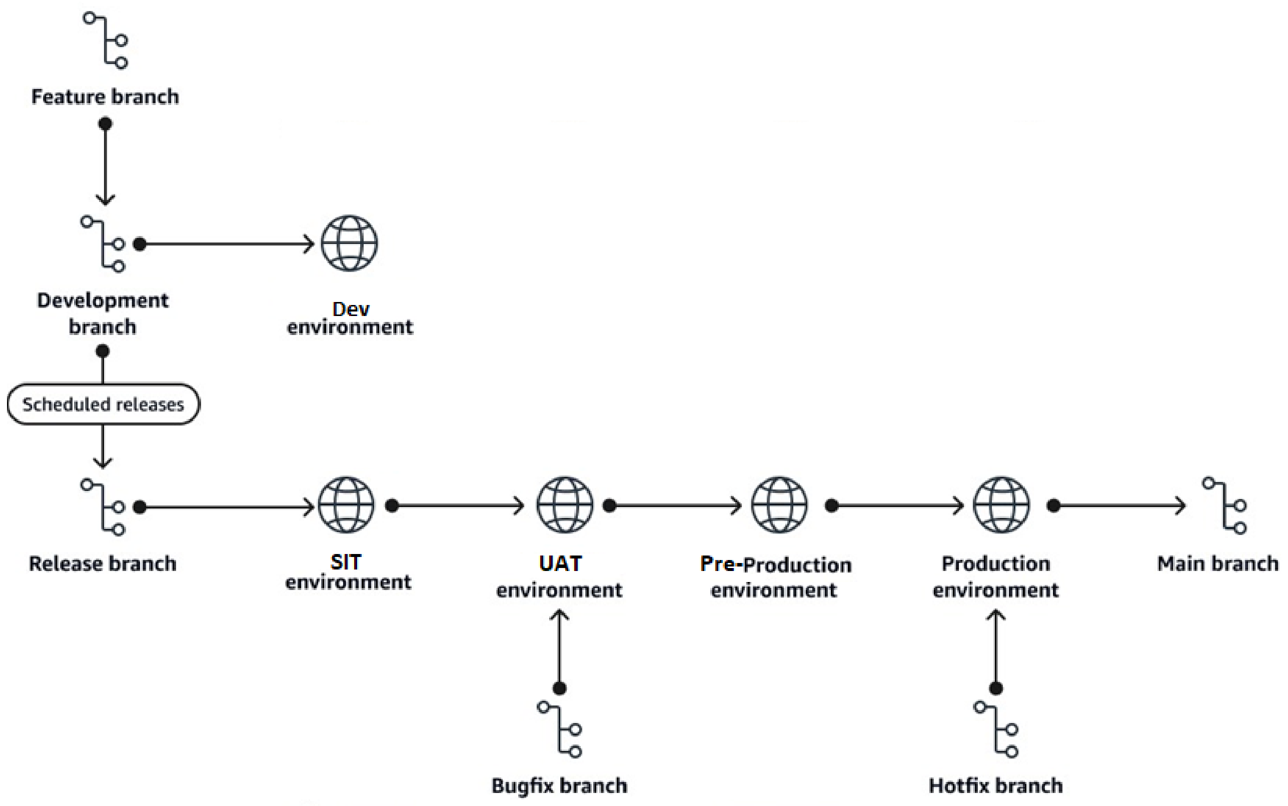
9.1.6. Hotfix branch

The hotfix branch is a short-term branch that is used to fix issues in production. It only be used to promote fixes that must be expedited to reach the production environment. A hotfix branch is always branched from main.

After the hotfix is tested, you can promote it to production through a merge request into the release branch that was created from main. For testing, you can then push the release branch forward by following the standard release process.

Naming convention:	hotfix/ <repo initials>_<story number>__<descriptor>
Naming convention example:	Hotfix/JS_123456_Fix_Problem_A

9.1.7. Please find the branch strategy diagram of application:-



10. Factors Influencing Solution Design

10.1. Standards

The DevSecOps solution is meticulously designed to comply with the Bank's IT security policies and standards, as well as the overarching IT policies and standards. It adheres to the OWASP DevSecOps guidelines, ensuring robust security integration throughout the development lifecycle.

Benefits include:

- Enhanced Security: Continuous security integration helps identify and mitigate vulnerabilities early in the development process.
- Compliance Assurance: Adherence to industry standards and guidelines ensures compliance with regulatory requirements.
- Improved Efficiency: Automation of security checks and processes reduces manual effort and accelerates development timelines.
- Risk Reduction: Proactive risk management minimizes the potential for security breaches and data loss.
- Cost Savings: Early detection and resolution of security issues reduce the costs associated with post-deployment fixes.

10.2. Assumptions:

- DevSecOps tools will be installed as per bank's security policies.

10.3. Constraints

- Hardware or software environment

10.4. Design Goals

- The DevSecOps solution is designed to achieve the following goals:

1. Comprehensive Tool Chain: The selected tool chain effectively covers the entire end-to-end delivery lifecycle, ensuring seamless integration and operation.
2. Industry-Recognized Tools: The tools utilized are well-regarded and widely recognized within the industry, providing reliability and trustworthiness.
3. Technology Compatibility: The tools are fully compatible with the technology stack used for the
4. SBIePay 2.0 application, ensuring smooth and efficient operations.
5. Integrated Security Practices: Security practices are seamlessly integrated into the Software Development Life Cycle (SDLC), enhancing the overall security posture.
6. Accelerated Delivery: The tools facilitate a faster delivery cycle, enabling quicker time-to-market and improved responsiveness to business needs.

11. Operational Requirement

User Groups: The below table includes the user groups of the DevSecOps solution.

This table provides a clear overview of roles, their access to specific branches, and the tasks they are responsible for.

User Role	Git Lab Role	Branch Access	Responsibilities	Work Breakdown
Developer	Developer	- feature/* - develop - bugfix - hotfix	– 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/* - release	– Review and approve feature branches and mergers to develop.	1. Review feature merges. 2. Ensure code quality. 3. Assist in resolving conflicts during

		-develop -bugfix - hotfix	– Coordinate with UAT for bug fixes.	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.
Release Manager / Lead / Architect/Bank Officials	Owner or Maintainer	- main - develop - release/* -main/*	– Oversee merges into master and release branches. – Create release tags.	1. Merge UAT branch into main for release. 2. Create release tags. 3. Coordinate hotfixes when needed.
DevOps Engineer	Maintainer	- main - release/* - hotfix/* -bugfix/*	– Monitor production releases. – Apply hotfixes if necessary.	1. Deploy main to production. 2. Create hotfix branches for critical fixes. 3. Merge hotfixes into develop and master.

11.1. 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 (PR)**: Once the development is done, the developer creates a **Pull Request (PR)**. 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.

11.2. Lead Developer/QA Engineer

- **Role**: Oversees code reviews, ensures integration quality, and manages UAT processes.
- **Flow**:
    - **Review Pull 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 **UAT** 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.

11.3. 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.

11.4. Production Support/Hotfix Team

- **Role**: Handles urgent production issues or bugs and ensures that quick fixes do not affect the stability of the codebase.
- **Flow**:
    - **Cloning from Release Tag**: If a production bug is identified, the team clones the **Release Tag** and creates a **HotFix** branch.
    - **Production Bug Fixing**: The bug is fixed within the **HotFix** branch without disturbing the ongoing development in other branches.
    - **Merging Hotfix**: After the hotfix is tested and verified, it is merged back into the **Release** branch for production and into the **Master** branch for future stability.
    - **Deploy to Production**: The fix is deployed to the live production environment after approval.

11.5. 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.
  - **UAT Feedback**: After code is moved to **UAT**, stakeholders perform acceptance testing to validate that the developed features meet the business needs and requirements.
  - **UAT Approval**: They give final approval for merging to **Master** if the code meets the defined requirements.

12. Continuous Integration

Continuous Integration (CI) is a key practice in modern DevOps and software development workflows that involves automating the process of testing and integrating code changes into a shared repository frequently. GitLab provides built-in support for CI, making it a powerful tool for automating testing, builds, and deployments.

12.1. 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.

12.2. Stages

12.2.1. 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

12.2.2. 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

12.2.3. 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.

12.2.4. Code Quality

The **Code Quality Stage** in a **CI/CD pipeline** focuses on ensuring that the code adheres to best practices, is maintainable, and is free from common pitfalls, code smells, and inefficiencies. This stage can include various automated checks such as **static code analysis**, **style checks**, **complexity analysis**, **duplicate code detection**, and **code formatting** to ensure the codebase is healthy and maintainable.

- Gitlab Code Quality

12.2.5. 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

12.3. Merge Request Pipeline

12.3.1. Trigger Condition

- Rules
    - On MR Creation
    - Destination branch is develop or release

12.3.2. Stages

- Lint
- Build
- Test

- Code Quality
- SAST
- Vulnerability Assessment (HP Fortify)
- Notification

12.3.3. On Merged Pipeline

12.3.3.1. Trigger Condition

- Rules
    - On MR Approved
    - Source branch is develop or release or main

12.3.3.2. Stages

- Lint
- Build
- Test
- Code Quality
- SAST
- Packaging
- Container
    - Build Docker Image
    - Tagging with SHA/custom-identifier as tag
        - v1.X minor version increment for develop branch
        - vX.X major version increment for release
- Libraries
    - Tag the libraries with SHA/custom-identifier as tag
        - v1.X minor version increment for develop branch
        - vX.X major version increment for release
- Publish
- Container
    - Push to Registry
        - Non-Prod Container Registry for develop or release
        - Prod Container Registry for main
    - Trigger Environment CD Pipeline through Push
        - Development environment for develop branch
        - SIT environment for release branch
    - DAST
- Libraries
    - Push to Gitlab Package Registry
- Notify to specific DL group

13. Continuous Deployment

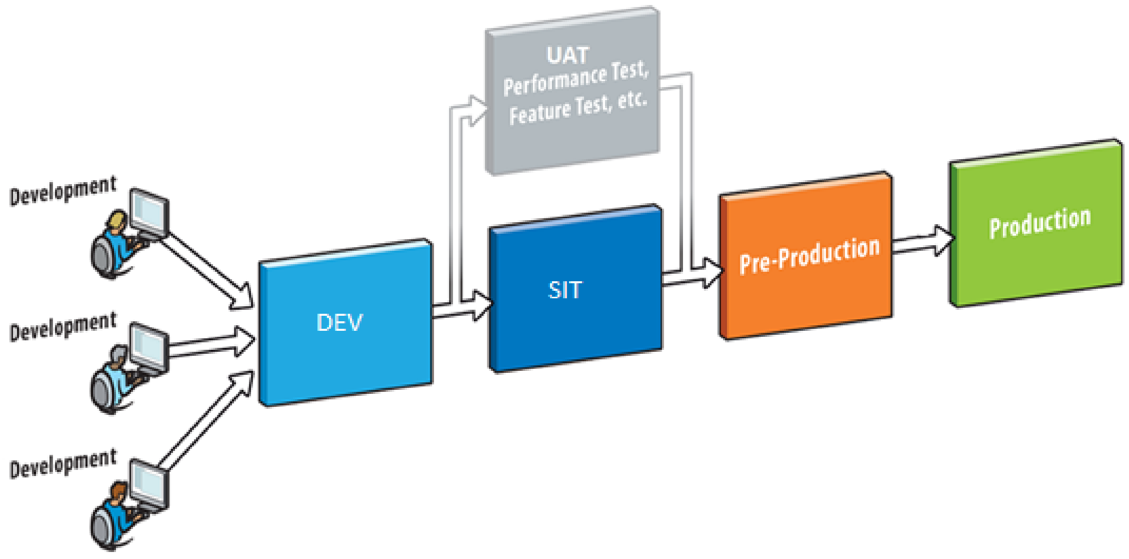
Continuous Deployment (CD) is an extension of Continuous Integration (CI) where every change that passes automated tests is automatically deployed to production without requiring manual intervention. In GitLab, Continuous Deployment integrates seamlessly with GitLab CI/CD, allowing teams to automate the entire pipeline from development through to production.

13.1. Environments

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

- Development (DEV) – An environment where developers integrate their code to confirm that it all works as a single, cohesive application.
- Development (SIT) – An environment where stable code will release from dev environment.
- Testing (UAT) – An environment where BANK UAT teams or acceptance testing takes place. Teams often do performance or integration testing in this environment.
- Pre-Prod – A preproduction 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.
- Production – An environment that handles traffic from your end users and customers.

  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.



13.1.1. 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.

13.1.1.1. 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 code coverage analysis
- Perform static code analysis

13.1.2. Development 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.

13.1.2.1. 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 code coverage analysis
- Perform static code analysis

13.1.3. 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

13.1.3.1. 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

- The development and QA teams have performed sufficient testing to satisfy Business requirements.
- The development team has resolved any discovered bugs through a bugfix branch.

13.1.4. 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 to production workloads. 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.

13.1.4.1. 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

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

13.1.5. Production environment

The *production environment* supports the released product, handling real data by real clients. This is a protected environment that is assigned access by least privilege and elevated access should only be allowed through an audited exception process for a limited period of time.

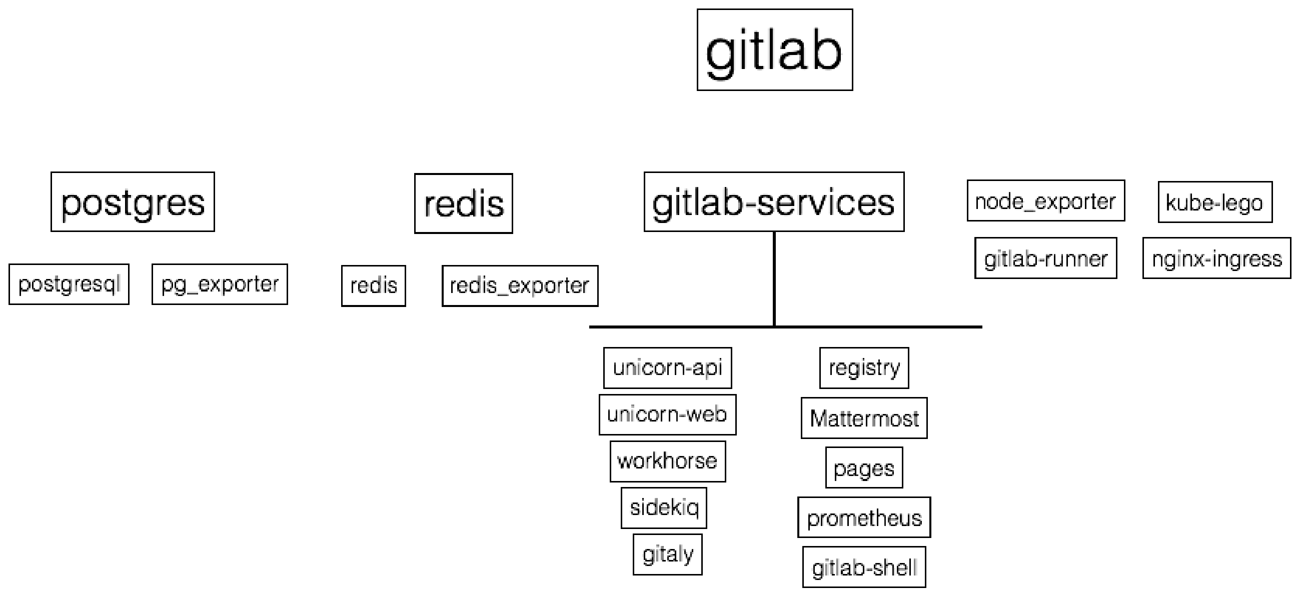
13.1.5.1. Access

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.

13.1.6. Helm charts

A Helm chart will be created to manage the deployment of each GitLab specific container/service. We will then also include bundled charts to make the overall deployment easier. This is particularly important for this effort, as there will be significantly more complexity in the Docker and Kubernetes layers than the all-in-one Omnibus-based solutions. Helm can help to manage this complexity, and provide an easy top level interface to manage settings via the **values.yaml** file.

Helm charts: -



14. Tools used

Tools	Version	Purpose

Gradle	8.9	Java Build Tool
NPM	10.9.0	React js Build tool
Eslint	9.11.1	JS Syntac checker
Fortify	23.1.0	Code Quality checker
Redhat Openshift Container Platform	4.14.29	Orchestration Platform
Podman	4.9.4	Container Platform
Gitlab Runner	17.2.1	Open source code repository
Gitlab Server	17.2.2-ee	Open source code repository
Gitman Semgrep Analyzer	5.19.0	Code Vulnerability checker
Gitlab Advanced SAST Analyzer	1.0.21	Code Vulnerability checker