International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

Int. J. Sci. Res. Arch.
IJSRA
International Journal of Science and Research Archive
Research Journal Archive, INDIA

(Review Article)

Check for updates

Modernizing CI/CD Pipelines: Migrating from Legacy Tools to GitLab for Enterprise Applications

Naga Murali Krishna Koneru *

Accenture Solutions Private Limited, INDIA.

International Journal of Science and Research Archive, 2021, 01(02), 136-156

Publication history: Received on 27 December 2020; revised on 26 January 2021; accepted on 29 January 2021

Article DOI: https://doi.org/10.30574/ijsra.2021.1.2.0027

Abstract

In today's modern software development, CI and CD pipelines are essential tools for faster and more reliable delivery of high-quality software. Such legacy CI/CD tools perpetuate themselves, and many enterprises continue to cling to inefficient, janky, and constantly fragmented tools that default to introduce complexities in scaling and integrating with modern technologies. This paper advocates developing a modern CI/CD system with GitLab, a DevOps platform combining version control, CI/CD, monitoring, and security into a single unified platform. This paper studies the difficulties with the current CI/CD systems, what GitLab presents advantages for, and how to perform an enterprise migration of CI/CD. Scalability, automation features, and smooth integration with cloud platforms like Kubernetes and AWS are advantages of Google GitLab. A real-world case study of migration to GitLab shows empirically that it increased the deployment frequency, lead time, and operational efficiency. The results indicate that GitLab provides a good scale, efficient, and integrated solution in support of the requirements of the modern software architecture, which includes microservices and cloud-native applications. This research gives businesses a hands-on approach to their CI/CD pipelines, focusing on automated processes, toolchain convergence, and continuous improvement. Organizations adopting GitLab acquire speed in software delivery, lessen operational costs, and keep the market edge when technology becomes more complex.

Keywords: CI/CD Pipelines; GitLab; Legacy Tools; Automation; Scalability; DevOps

1 Introduction

In the territory of software development, no one's needs have been more urgent than the need for good, reliable, and scalable tools. Continuous Integration (CI) and Continuous Deployment (CD) pipelines are now involved in the software development process, making it faster and error-free according to the traditional approach. Modern development practices automate not only the code change integration, the automated testing and the application deployment but also significantly improve the speed and quality of software delivery. Nowadays, as businesses continuously try to stay up-to-date in the rapidly changing world of technology, CI/CD is an integral part of the process that aids software development. With a CI/CD pipeline, there are many benefits such as automation of repetitive work, reducing human error and reducing the time to market. CI/CD pipelines automate testing and deployment; this means testing and deploying earlier in the cycle helps identify issues earlier in the development cycle, producing higher-quality software. This also promotes a culture of collaboration where teams can work in parallel to different stages of the software development lifecycle, thus speeding up the process. While these advantages exist, many enterprises rely upon legacy CI/CD tools, which had their foundation laid long before the advent of the software application and are virtually unrecognizable from the monolithic architecture of the previous decade.

Most legacy CI/CD systems introduce significant obstacles that prevent meeting the expectations of developing software modernly. That is just one of the main reasons why toolchains are fragmented. These legacy tools are usually collections

* Corresponding author: Naga Murali Krishna Koneru

of multiple independent systems for version control, build automation, testing, and deployment in different stages of development. Due to the lack of integration between these tools, the data has to be synchronized manually across platforms, making the data prone to errors and inefficiencies. Having multiple tools to continue developing is cumbersome, and maintaining all of them can be a task, having to update, patch, and reconfigure the tools for each update to be compatible. A major limitation of existing CI/CD systems is that they cannot scale well as the complexity and scale of modern applications continue to grow. Legacy tools are no match for evolving software architecture, ill-suited microservices, cloud-native platforms, and distributed systems. These are tools for simpler cases, and they cannot handle the high needs of today's complex applications. As a result, experts have longer build times, higher failures, and less flexibility in deploying applications across various environments. Legacy CI/CD tools are a major hindrance to enterprises when confronted with the need to adopt modern software practices – such as containerization and cloud infrastructure-based software development.

Another major challenge is integration with the latest DevOps tools and cloud platforms. With the proliferation of technologies such as Kubernetes, AWS, and Azure, traditional CI/CD tools often cannot interact with these platforms as they are and do not natively support them. Doing so encourages developers to write custom scripts and build workarounds to build things together in a way that raises complexity and the possibility of errors. In addition, legacy tools force individuals to spend many resources on maintaining them since they are not updated automatically, and the products they support might be no longer by the original vendors. Ongoing maintenance, in addition to manual configuration changes of these systems, does add to the overall cost of operation, making them inefficient in a rapidly changing technological landscape. GitLab, a comprehensive DevOps platform, offers a robust solution to these challenges. GitLab provides a full suite of version control, CI/CD, monitoring, and security tools in a single platform. GitLab accomplishes this by consolidating these functions, which results in getting rid of, or at least reducing the need for, fragmented toolchains and making the way CI/CD pipelines are managed much easier. Its ability to scale efficiently with high levels of integration into modern technologies, i.e., Kubernetes and cloud platforms, makes it a great tool suitable for enterprises interested in modernizing their software delivery methodology. In addition to a highly supported strong community, GitLab's automated updates further decrease maintenance costs and allow teams to innovate rather than troubleshoot old systems.

This study aims to help understand the challenges of running CI/CD based on legacy systems, the benefits of going to GitLab, and the practical steps enterprises can take to modernize the CI/CD pipelines. This paper is a detailed analysis of how GitLab addresses GitLab limitations issues and how a company can migrate from the legacy tool to a more efficient, scalable, integrated solution. Based on real-world case studies and implementation strategies, this study provides an excellent basis for businesses to increase their software development workflows.

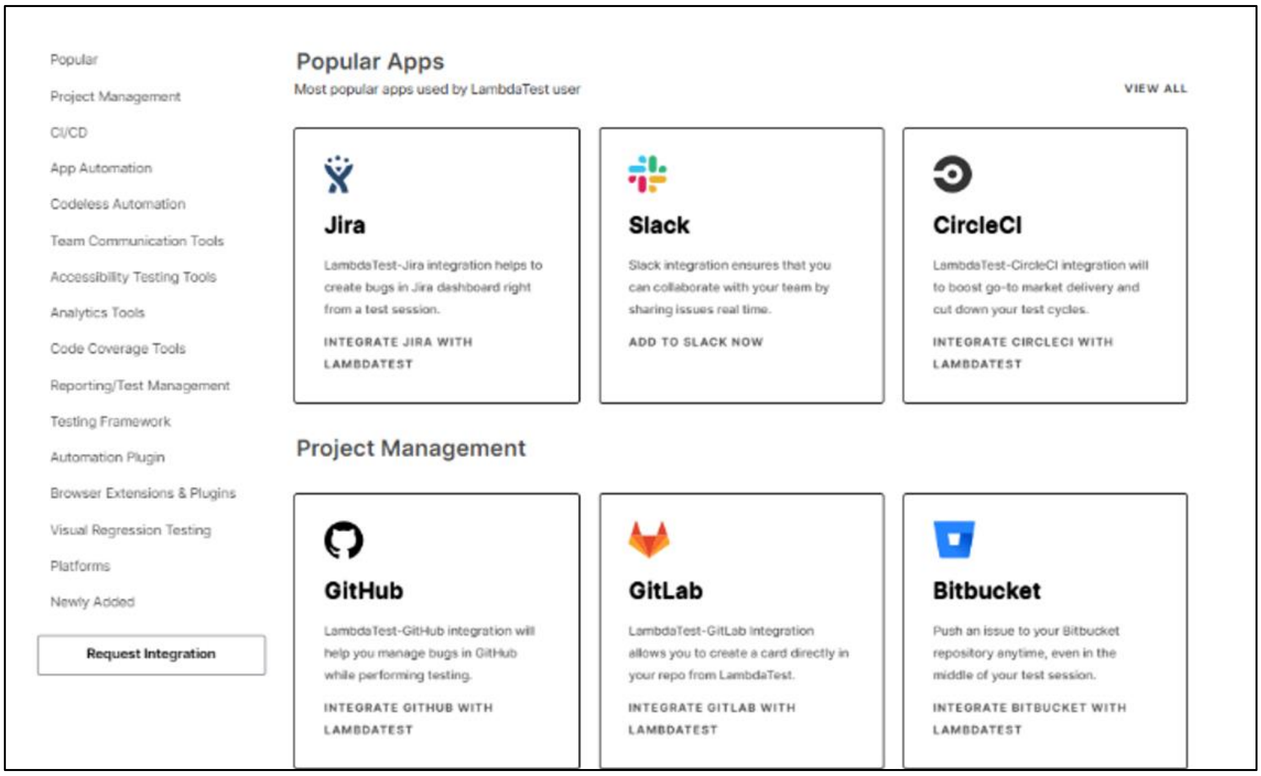
2 Challenges with Legacy CI/CD Tools

Legacy Continuous Integration and Continuous Deployment (CI/CD) tools were important for enterprise software development. They are hard to move with the times as their aid was to break centres of business, isolate them from the rest of the projects, and create a substantial process from scratch. Astware development processes evolve, these traditional tools change and often do not fit the requirements of contemporary DevOps workflows. Below are some of the critical challenges of legacy CI/CD tools.

2.1 Fragmented Toolchains

Legacy CI/CD tools are typically plagued by requiring pieces of the fragmented toolchain. Each stage of the software delivery process in many legacy pipelines is handled through different disconnected tools. GitHub can be responsible for version control, Jenkins for build automation, Selenium for testing, and Ansible for deployment. These are not seamless tools that integrate with each other; each has a specific purpose.

This fragmentation wastes much time in the development process. This means developers must manually sync the data from tools, which causes errors, delays, and additional overhead. For example, actions performed on the codebase in GitHub may also need to trigger builds by hand in Jenkins (Belmont, 2018). Configuring and maintaining configuring and maintaining each tool on its own can easily increase the system's complexity and complicate troubleshooting when issues occur. Such systems are disjointed; thus, it is possible to fail in one pipeline unit. That failure will affect the rest of the process, causing a slower development cycle and lower software quality. Besides, legacy tools have configuration languages, dependencies, and maintenance requirements. It is not easy to automate the workflow from one stage to another. Organization in silos causes a decrease in efficiency as a pipeline depends on multiple disparate tools, in which separate teams or specialists manage each tool.



Popular Apps

Most popular apps used by LambdaTest user

Jira

LambdaTest-Jira integration helps to create bugs in Jira dashboard right from a test session.

INTEGRATE JIRA WITH LAMBDATEST

Slack

Slack integration ensures that you can collaborate with your team by sharing issues real time.

ADD TO SLACK NOW

CircleCI

LambdaTest-CircleCI integration will to boost go-to market delivery and cut down your test cycles.

INTEGRATE CIRCLECI WITH LAMBDATEST

Project Management

GitHub

LambdaTest-GitHub integration will help you manage bugs in GitHub while performing testing.

INTEGRATE GITHUB WITH LAMBDATEST

GitLab

LambdaTest-GitLab Integration allows you to create a card directly in your repo from LambdaTest.

INTEGRATE GITLAB WITH LAMBDATEST

Bitbucket

Push an issue to your Bitbucket repository anytime, even in the middle of your test session.

INTEGRATE BITBUCKET WITH LAMBDATEST

Figure 1 CI/CD Pipeline Challenges With Resolution

2.2 Lack of Scalability

This is because businesses grow, both in terms of size and complexity of their architectures, adopting increasingly complex architectures more commonly. These legacy tools typically cannot keep up with the pace of modern software development as these were designed with monolithic applications in mind, which are orders of magnitude simpler in terms of dependencies and the size of the codebase from the distributed systems which are commonplace nowadays. Microservices are a complex solution that enables multiple independent services that may be deployed in different environments and overshadows legacy systems (Dattatreya, 2019). The biggest hurdle is that legacy tools cannot scale as demand increases. Centralized processes and their restricted capability for parallel implementation are common with legacy systems. Since development teams already started to move to cloud-based or containerized environments like Kubernetes, legacy tools are failing to scale in the same way to distribute applications efficiently.

For instance, when the number of microservices grows, it could result in prohibitive delivery of pipeline delays in the length of time for tools such as Jenkin,s. Specifically, these delays are triggered because Jenkins is not optimized for supporting large complex builds and does not work efficiently across multiple nodes. Legacy tools typically also cannot scale dynamically in response to workload changes. This creates resource bottlenecks, long queue times for the builds, and higher operational costs other than the need for human intervention to rebalance workloads over limited resources available (Nzanywayingoma & Yang, 2019). It also affects the scalability of testing and deployment. In the legacy world, testing is usually a bottleneck when many services need parallel testing or integration into different environments. Such a situation can lead to frequent build failures and longer testing cycles, increasing the time to market for applications.

2.3 Poor Integration

The other major challenge of legacy CI/CD tools is their poor integration with modern DevOps and cloud platforms. As a part of the modern software developing world, CI/CD integration with containers like Kubernetes, AWS, and others is vital for executing fast and trustful deployments (Bondarenko, 2020). Legacy tools do not typically integrate natively with these platforms, and plugins would require custom scripts and manual configurations to a bridge. For instance, developers typically need to craft custom scripts to automate interactions with cloud services or container orchestration platforms like Kubernetes using legacy tools like Jenkins. This extra layer of complexity increases the risk of errors during deployment since the custom scripts must be meticulously written, tested, and maintained. As the cloud platform advances, these scripts will also need constant updating to ensure compatibility with all the latest features in the platform, which would add to the overhead for development teams.

The lack of integration makes it unable to support the implementation of automated monitoring and security checks. Modern CI/CD pipelines should be built to integrate with any monitoring, security, and incident management tools to ensure risk-free flow from development to production (Podjarny, 2017). Legacy tools suffer from these integrations. As a result, teams often have to set up their own third-party security and monitoring tools by hand, which leads to inefficiencies and security vulnerabilities. Deploying these becomes cumbersome, and deployment issues and security vulnerabilities can go unnoticed. Besides, most CI/CD tools were designed before cloud-native architecture and containerization spread. This typically leaves them disabled from containers of containerized applications or web applications built as microservices, where the friction between the development and operations teams continues. Instead, these teams must spend time integrating legacy tools into their new practices instead of investing that time in improvements in the quality of the software.

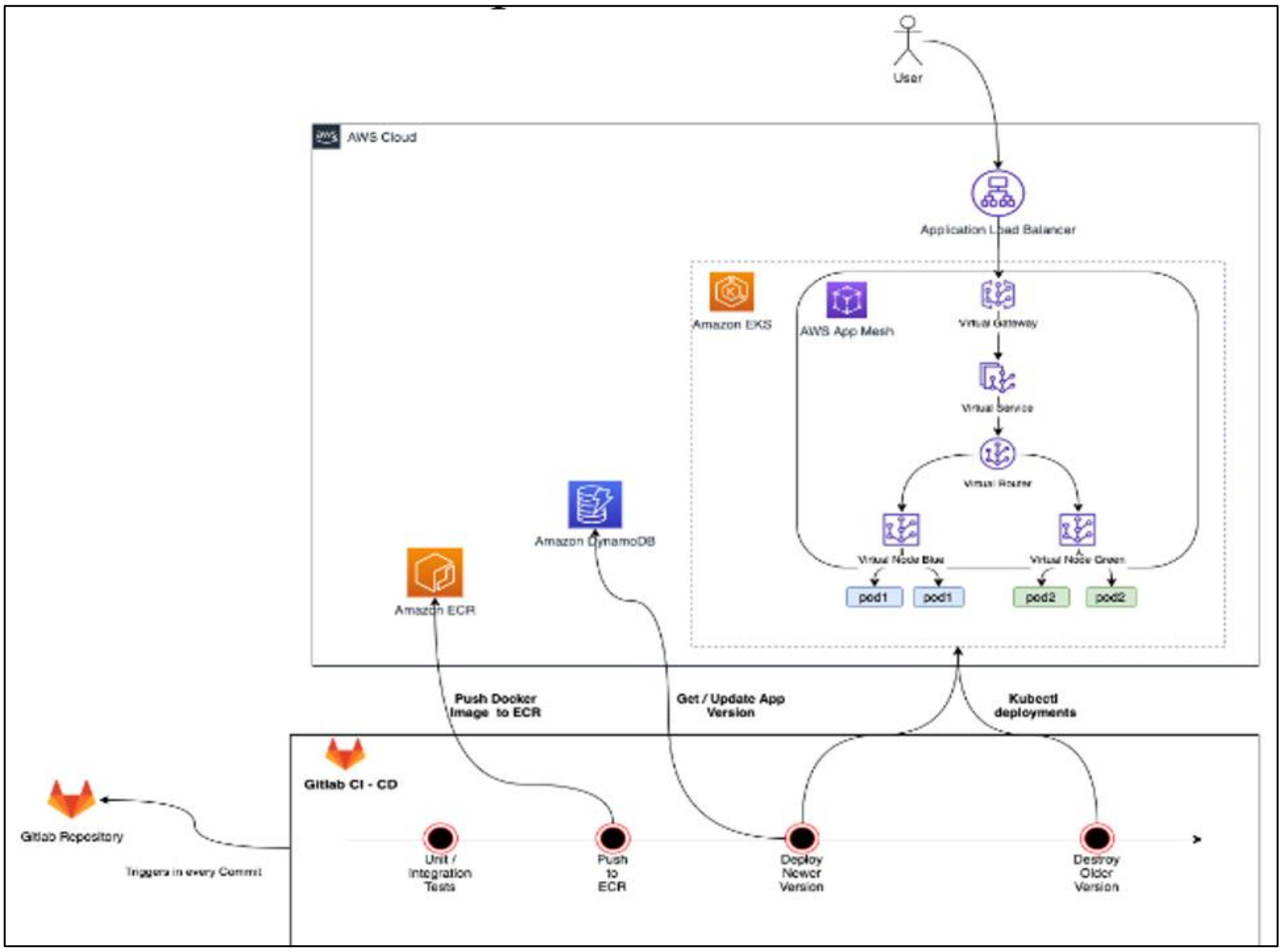


Figure 2 CI/CD with Amazon EKS using AWS App Mesh and Gitlab

2.4 High Maintenance Costs

Legacy CI/CD systems are usually one of the most costly undertakings for enterprises in terms of maintenance costs. Creating and maintaining tools for such legacy systems takes much time and resources. Legacy systems often do not have automatic updates and, therefore, have operators to track versions and patches for every software tool as needed, even for updating them to ensure they are always secure. As a result, the lack of automation in the maintenance process distracts the development teams from their main tasks, including updating configuration files, troubleshooting problems, and maintaining the infrastructure that supports the CI/CD pipeline. This implies higher operational costs, as most of the cost comes from maintaining legacy systems that need special know-how and dedicated resources. In contrast, other resources could be employed to improve the software delivery process (Nyati, 2018).

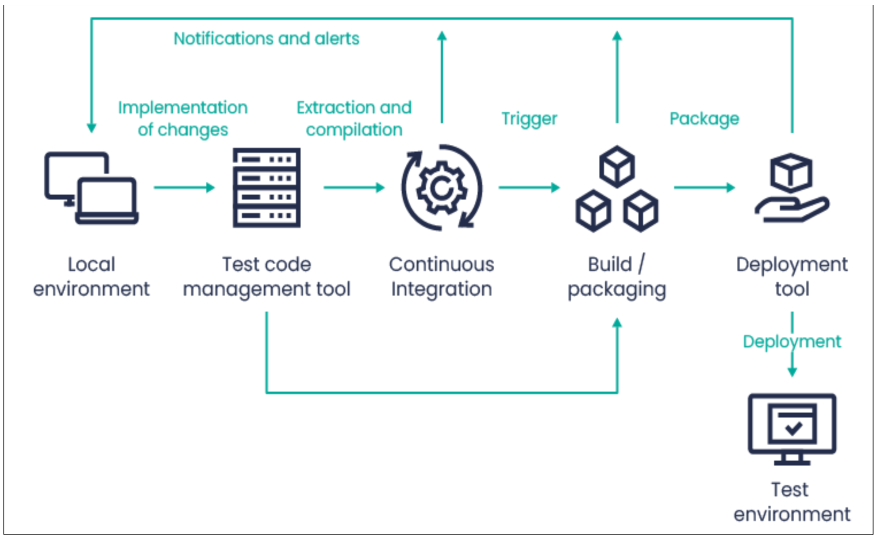


Figure 3 Automating CI/CD continuous integration & continuous delivery

The second challenge is the lack of vendor support for outdated tools. Most modern CI/CD systems are actively developed, and vendor support is ensured. Such organizational dependence on outdated and unsupported systems presents very serious risks, as these organizations are open to security breaches and other issues that should be easy to address but often are not unless the vendor helps out. As those tools age, it also becomes more difficult to find skilled personnel familiar with these old technologies, and the cost of goods is increasing. In turn, the cost of maintaining legacy CI/CD systems can be greater than they bring. These systems are constantly patched and pulled around in hopes they 'work', thus robbing resources from innovating or developing software (Williams, 2019). This also makes it difficult to automate all the maintenance tasks needed in the CI/CD pipeline, which slows down the development process and may even decrease the quality of the final product.

Legacy CI/CD tools are challenging enough, but with the ever-changing software industry, enterprises need to consider whether the software development landscape requires them to go with more modern, scalable, integrated solutions. That's why organizations must address these challenges if they want to find ways to streamline their workflows, enhance teamwork, and decrease turnaround time.

3 Migration Strategy

3.1 Assessment and Planning

The first step towards a migration to GitLab is to carefully evaluate the current CI/CD environment. This involves looking at the existing pipeline, its inefficiencies, and how to use legacy tools to address the limitations. The first part evaluates the current toolchain, builds processes, deploys flows, and integrates multiple systems.

To determine the pain points of a current setup, it is necessary to inventory the tools in use in all contexts. Some common challenges include fragmented tool extraction, where various tools are used for version control, build automation, testing, and deployment. Examples may also include scalability issues when legacy tools cannot cope with the requirements of contemporary, cloud-native apps. Moreover, the organization should determine how integrated the current tools are and how much manual intervention is needed to keep the pipeline running smoothly, if it does.

After the existing CI/CD pipeline is fully assessed, it is time to build an elaborate migration plan. The specific goals of the migration, the timeline that is supposed to pass, the allocation of resources, and the risk mitigation strategies should be outlined in this plan (Kumar, 2019). With a well-defined roadmap, the migration process will stay on track, and all the known potential challenges will be dealt with proactively. The schedule must also consider any dependencies that might affect the migration, for example, some hardware, some programming, and some personnel restrictions.

3.2 Toolchain Consolidation

The next step in the migration process after the assessment phase is toolchain consolidation. One of the main reasons to move to GitLab is that it unifies several CI/CD processes into one platform. Legacy systems commonly employ various tools for respective development stages, like Jenkins for automation, GitHub for version control, and Ansible for deployment. The problem is that these tools generally work in isolation, and most need manual synchronization and coordination between teams.

The first in migrating to GitLab is to combine these disparate tools into a unifying workflow. The most important part would be migrating the version control repositories (located in a separate tool) to GitLab's version control system. This centralizes the code base, as all development work involves creating a single code base that acts as a single source of truth. At the same time, the existing CI/CD pipelines must be set up once again using GitLab's robust features for automation and continuous delivery. GitLab provides many integrations with other DevOps tools to keep the existing workflows if needed (Engwall & Roe, 2020). For example, GitLab can integrate with Kubernetes for container orchestration or link with integrated third-party tools for testing or monitoring. The purpose of this phase is to replace external tools with a single, simpler thing. The main principle of migration should be to simplify the overall setup at the cost of losing some functionality.

3.3 Pipeline Configuration

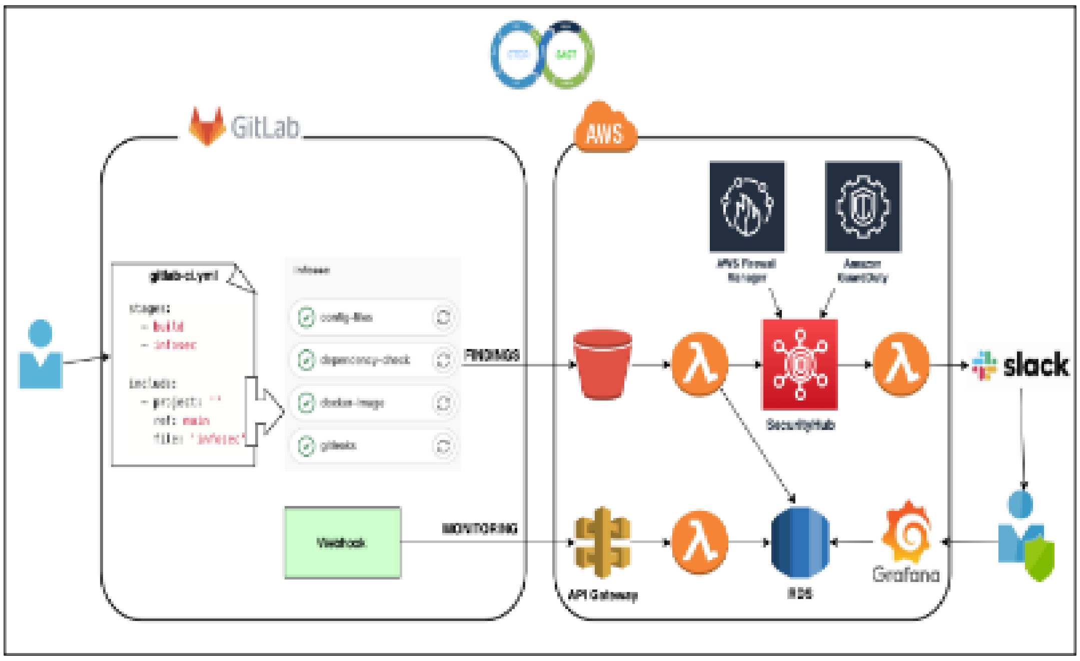
After consolidating the toolchain, the next step is configuring the CI/CD pipeline in GitLab. Defining it means defining the core stages of the pipeline, which in our case become the build, test, and deploy, and making them automated as much as possible. An advantage of GitLab is that it provides an integrated pipeline configuration that covers all software development expectations. This is when the migration team has to start thinking about the pipeline that will automate certain processes and fit into the organization's development and deployment practices. This includes creating build jobs that build code and creating corresponding artefacts like binaries or container images. Automated tests, including integration tests and security scans, are also run via test jobs, which must be defined. These tests keep the code quality unchanged from the initial testing to the end of the development cycle.

After the build and test jobs have been built out, the deployment process should also be automated. It usually sets up GitLab to deploy applications to multiple environments (staging, production, or test). The continuous delivery capabilities of GitLab facilitate the automation of deployments so that the application is always delivered to the right environment without human intervention (Chinamanagonda, 2020). Further, it is not limited to build, test, and deployment. GitLab provides tools for setting up automated triggers and notifications, which will send warnings to developers when the code is running a test or a deployment has succeeded or failed. It automates the process, making the team's communication and the development process easier, giving results much quicker to the developers.

3.4 Testing and Validation

Once configured, it is crucial to validate and test the new CI/CD process and then transition the entire build CI/CD process to GitLab before this is done. In this phase, the newly configured pipeline ensures that it is acceptable and works as expected, per the organization's requirements. The new pipeline must be tested by running integration tests to confirm that it fits together as expected with the current systems. This includes ensuring that GitLab can smoothly play with these other tools and services that you use in your workflow, such as databases, APIs, or external integrations. Testing the pipeline's functionality is important, but it should be easily scalable and reliable under different conditions.

Performance testing is also vital in addition to the integration tests. The GitLab pipeline should be run to see if it can handle the expected load, especially in a production environment. This can involve running stress tests to simulate high traffic or massive deployments. The pipeline should also be optimized for speed, and there should not be bottlenecks or delays in the build, test, and deployment stages. Testers need to validate the security of their apps. Security scans are an important part of ensuring the security of the codebase from its inception, as one can integrate security checks into the CI/CD pipeline early in GitLab's pipeline (Orazi et al., 2020). All code can be run through automated security tests to try and prevent security flaws from being introduced and ensure that all code has been evaluated regarding security before it is deployed. The pipeline has to be validated from the users' point of view. This includes an intuitive pipeline, clear workflows, easy navigation, and proper documentation. It will also test with real-world scenarios and use cases to make the system functional and easy to use.



```
stages:
  - build
  - infosec

include:
  - project: ''
    ref: main
    file: 'infosec'
```

Figure 4 Incorporating security in CI/CD pipelines

3.5 Training and Adoption

The last step in this journey is to enable and connect the development and operations team with relevant features and workflows using GitLab. Training is important to ensure an organization fully adopts the new CI/CD system and maximizes its impact. Hands-on workshops using GitLab should be part of training so that team members can practice using GitLab to perform common tasks, such as repository management, pipeline configuration, test running, and application deployment (Evertse, 2019). These workshops should be based on roles within the organization, such as developers and operations teams, and participants should understand how they can work with GitLab in their specific workflows. Besides workshops, continued adoption support calls for comprehensive documentation. The GitLab's documentation should provide step-by-step instructions for the most popular tasks, such as common troubleshooting and pipeline configuration, and it should be available as soon as possible. The process should also involve regular feedback sessions to address any issues the teams may bring to the training phase.

Any challenges encountered during the migration process should receive ongoing support to address those issues. This may include setting up a support team or a ticketing system where people report problems or ask for assistance. If supported continuously, organizations can be sure that there are no roadblocks during the adoption of GitLab and that any knowledge gaps can be filled. One can get a high adoption rate with proper training teams, solid documentation to support them, and robust ways to transition to GitLab's integrated CI/CD platform. It will lead to better collaboration, more efficient workflows and better software quality. From where one is to GitLab, there will be much planning, testing, and equally effective training (Bansal, 2015). Organizations can create a smooth transition of maximum benefit to GitLab's modern CI/CD capabilities by consolidating legacy tools, automated pipeline configuration, performance validation, and team training.

4 Case Study: Enterprise Migration to GitLab

4.1 Background

One of the world's most prominent financial services companies, with core banking and investment services offerings, utilized a legacy CI/CD pipeline that consisted of tools like Jenkins, GitHub, and Ansible. Though this legacy pipeline was functional in its initial years, it started to show an increase in inefficiencies as the company's infrastructure grew and its architectures became more complex. The company struggled with building and deploying workflows when moving from a monolith to a microservices-based environment.

The main issues were having a fragmented toolchain where Jenkins built automation, GitHub controlled our version, and Ansible handled our deployment automation. Although these tools worked alone, the development and operations teams performed tedious manual processes and escalated to using different tools. The second problem is that the tools could not deal with the growing scale and complexity of the company's expanding infrastructure, specifically in the cloud-native world.

The scale was one of the critical pain points. The existing legacy tools could not work efficiently due to the frequent changes and scalable deployment cycles required by microservices. Our build times were growing, deployment failures occurred more often, and the backlog of tasks crept up while our release cycle grew slower. There were also high operational costs due to the lack of a unified platform, and maintaining and upgrading the legacy pipeline required many resources. The company decided to modernize its CI / CD pipeline by migrating to GitLab in response to these challenges (Koopman, 2019). This full DevOps vendor solution provides a unified toolchain to manage the whole software development life cycle. The aim was to get to a more advanced, scalable, and integrated CI/CD system compatible with the ongoing expansion and transformation of the company's infrastructure.

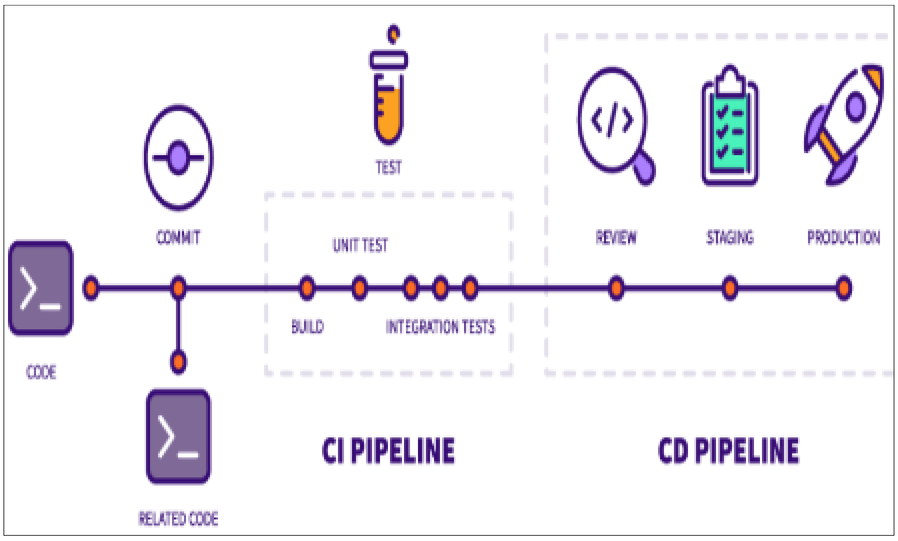


Figure 5 Modernizing CI/CD using GITLAB CI

4.2 Migration Process

The phased migration to GitLab was done in sequence following a structured plan to minimize disruption, and the new pipeline will fulfill company needs. To begin with, an analysis was done on the present CI/CD pipeline. The company faced challenges, such as toolchain fragmentation and manual process inefficiency. Also, the migration team thoroughly covered the company's development and deployment workflows to ensure that the new GitLab-based pipeline solves those pain points.

4.2.1 Toolchain Consolidation

Our first step of the migration was to bring together the company's disparate toolchain in GitLab. GitHub repositories were migrated to Gitlab along with Jenkins build jobs to Gitlab's share of CI/CD. The rewrite of Ansible scripts for deployment prepared for GitLab's built-in deployment features. This consolidation allowed to remove the need for multiple, disjointed tools. It allowed for a closer group of tools adopted by the development and operations teams to manage this load.

4.2.2 Pipeline Configuration

After the toolchain was consolidated, the migration team worked to set up the GitLab CI/CD pipeline to enable automatic build, testing, and deployment. The GitLab CI/CD pipelines they developed were customized to the company's fine-grained microservices architecture (Bogner et al., 2029). The build jobs were built to execute in parallel to reduce build times, and the testing in the pipeline was integrated à la test early and test often so that code quality was maintained.

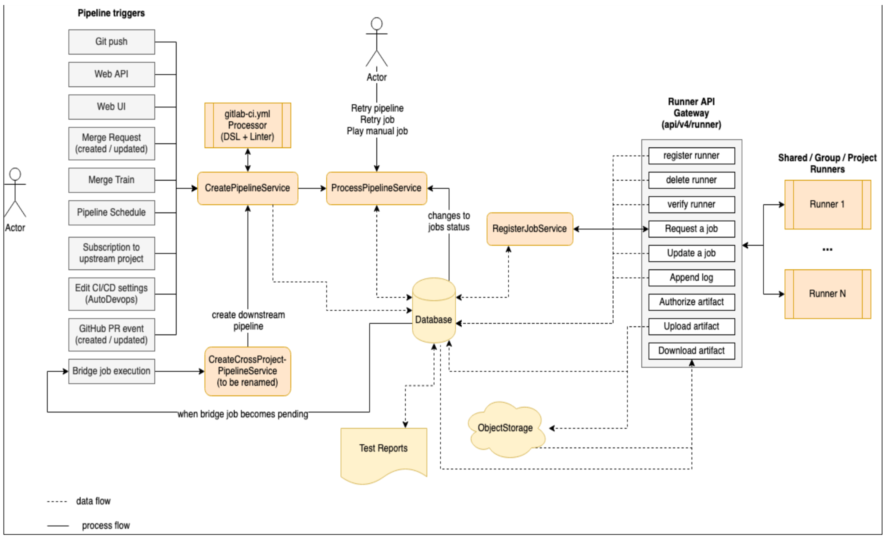


Figure 6 CI/CD development guidelines

The deployment process was also automated using GitLab's features, including Kubernetes integration for container orchestration. The company eliminated manual intervention via continuous integration, continuous testing, and continuous deployment via GitLab support while greatly increasing the reliability of the deployment process.

4.2.3 Testing and Validation

The company did extensive testing before it fully switched to GitLab to ensure that the new pipeline performed as it should for performance, security, and reliability. Integration tests were run to ensure the migrated pipeline could successfully handle the same workloads as the legacy system. The new pipeline was also put to performance tests to ensure performance with the increasing demands of the company's infrastructure.

In the GitLab pipeline, security scans were also introduced to find vulnerabilities early in the development cycle and guarantee security in the CI/CD process (Rangnau et al., 2020). During these tests, the company could fix any potential problem before it became a real problem and ensure that the pipeline would be sufficient for the volume of production workloads.

4.3 Results

The move to GitLab greatly helped the company pipeline, contributing to the commits to make operations more efficient, scalable, and fast to deploy.

4.3.1 Deployment Frequency

One of the best improvements was an increase in deployment frequency. The company's deployment process was slow and manual; deployments could only be done once a week. This allowed them to deploy many times a day and drastically cut down the gap between code changes and release into production. This improvement enabled the company to react more promptly to the customer's demands and the changes in the market.

4.3.2 Lead Time

The commitment to code to production deployment was dramatically decreased. In fact, during the migration, the company builds, tests, and deploys processes through lead times of up to two weeks. By migrating to GitLab and optimizing the build pipelines, setting up automated testing, and reducing the build and deploy workflows, lead time was reduced to two days (Karamitsos et al., 2020). Not only was this able to reduce operational efficiency, but it also increased the company's speed in marketing new features and updates to customers.

4.3.3 Operational Efficiency

After the migration, operational efficiency improved significantly. Maintenance effort by the company was reduced by 50% due to GitLab's combined platform and automation capabilities. Previously, they had to handRESULT parties that involved hand manipulation for toolchain management, troubleshooting issues, and upgrading the toolchain. The company's efficiency was advanced by the seamless integration between GitLab's version control, CI/CD, and monitoring tools that made little differentiation across the development lifecycle.

The successful migration to GitLab changed the company's CI/CD pipeline, allowing faster and more reliable software delivery. Seasoning of a move from a fragmented, legacy toolchain to an integrated, automated system allowed for significant deployment frequency, lead time, and operational efficiency. GitLab adoption allowed the company to modernize its software delivery processes and maintain speed with the increasing service needs inside a complex cloud-native infrastructure.

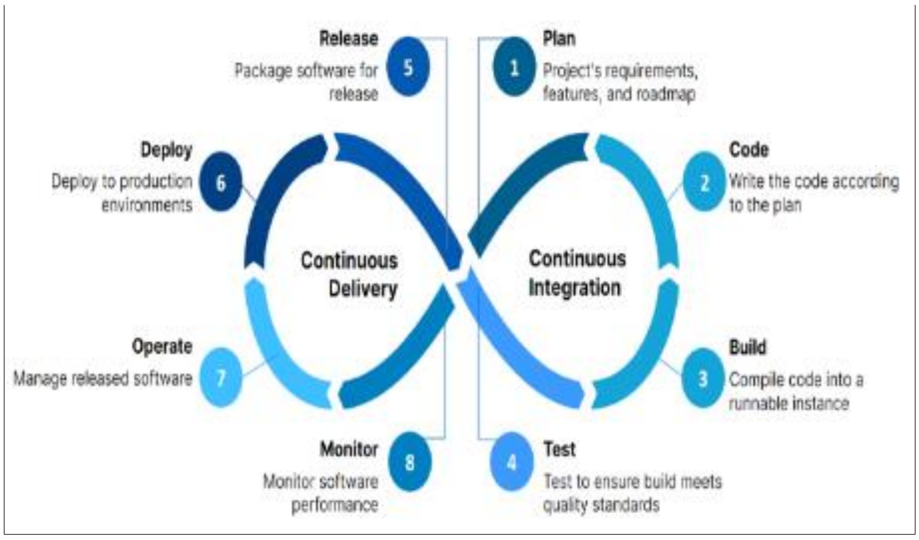

Release
Package software for release
5
Plan
1
Project's requirements, features, and roadmap
Deploy
Deploy to production environments
6
Code
2
Write the code according to the plan
Continuous Delivery
Continuous Integration
Operate
Manage released software
7
Build
3
Compile code into a runnable instance
Monitor
Monitor software performance
8
Test
4
Test to ensure build meets quality standards

Figure 7 The Role of the CI/CD Pipeline in Cloud Computing

5 Implementation Code Examples

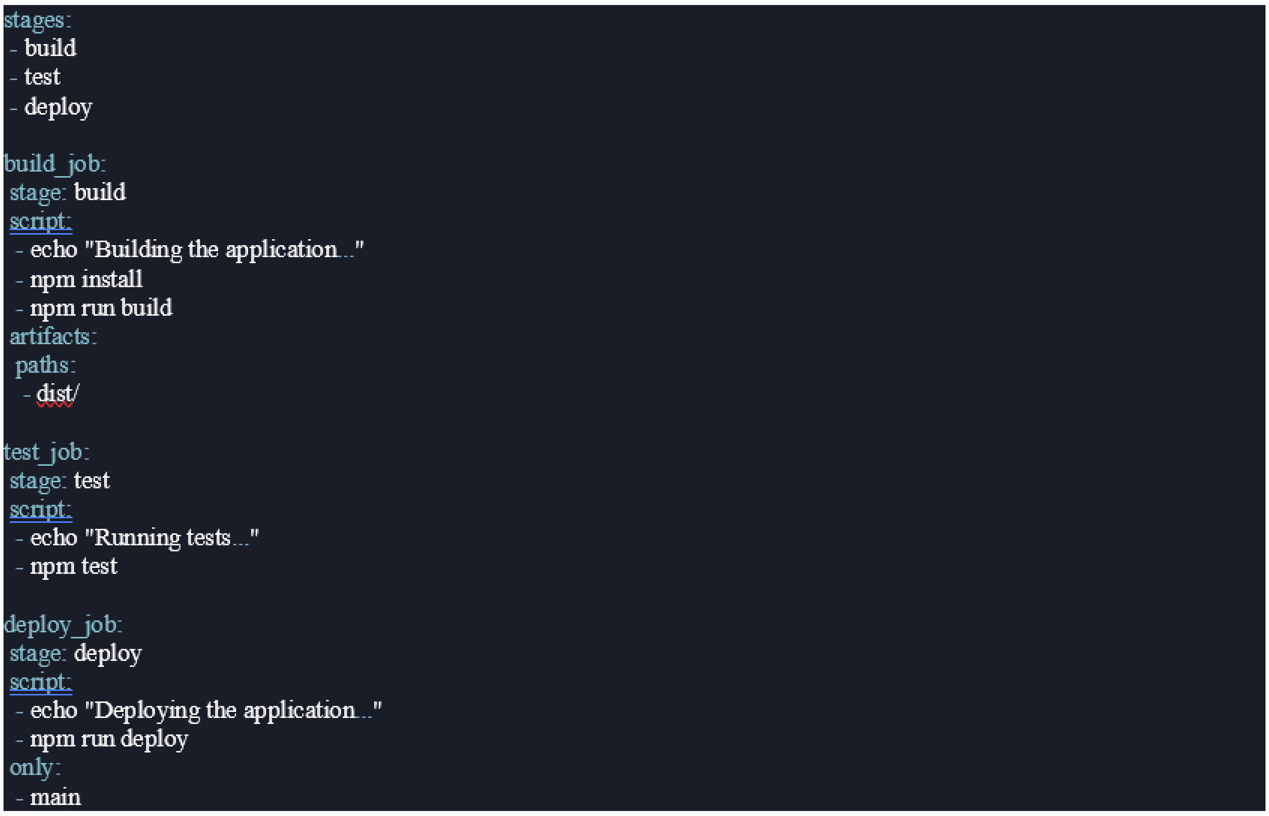
The Implementation Code Examples section demonstrates the real-world configurations used to produce the catalogs and helps better understand how to configure GitLab CI/CD pipelines depending on the type of development workflow. This section presents three distinct pipeline scenarios: a basic Node.js pipeline, a Docker pipeline, and an auto-scaling with Kubernetes. Each pipeline illustrates key GitLab features and tools. In these examples, these foundations are templates for teams to utilize when they have determined to modernize and automate their software development process.

Integrating code changes, testing, and deployment without human intervention is an essential feature of DevOps practices called CI/CD. GitLab covers these processes and many others as a unified DevOps platform, which supports them using many configurable options that can be customized for specific application environments. All the configuration examples presented here are conceived to create a minimum overhead for implementation while offering flexibility to add additional steps in the workflow and use different workflow elements. The example of a basic pipeline in Node.js shows a sequence of steps in building, testing, and deploying the Node.js application. In contrast, the pipeline based on Docker handles the Docker images and the automated containerized flow (Mehtonen, 2019). It also shows how the GitLab Runners can be auto-scaled in the Kubernetes-based configuration, scaling with less resource utilization and making the pipeline execution quick in large-scale environments. Going through these concrete examples of GitLab lets teams internalize its capabilities and how to apply them to improve efficiency, eliminate manual errors, and guarantee consistent delivery. They are made with these examples to make it easy to modify to particular organizational needs and technical requirements.

5.1 Example 1: Basic CI/CD Pipeline in GitLab

A typical CI/CD pipeline for a Node.js application consists of building, testing, and deploying. Below is an example of a .gitlab-ci.yml, a GitLab CI/CD pipeline configuration file for a Node.js application. The first section of this configuration shows the usage of multiple stages, an important concept of continuous integration and deployment with multiple artifacts.

yaml

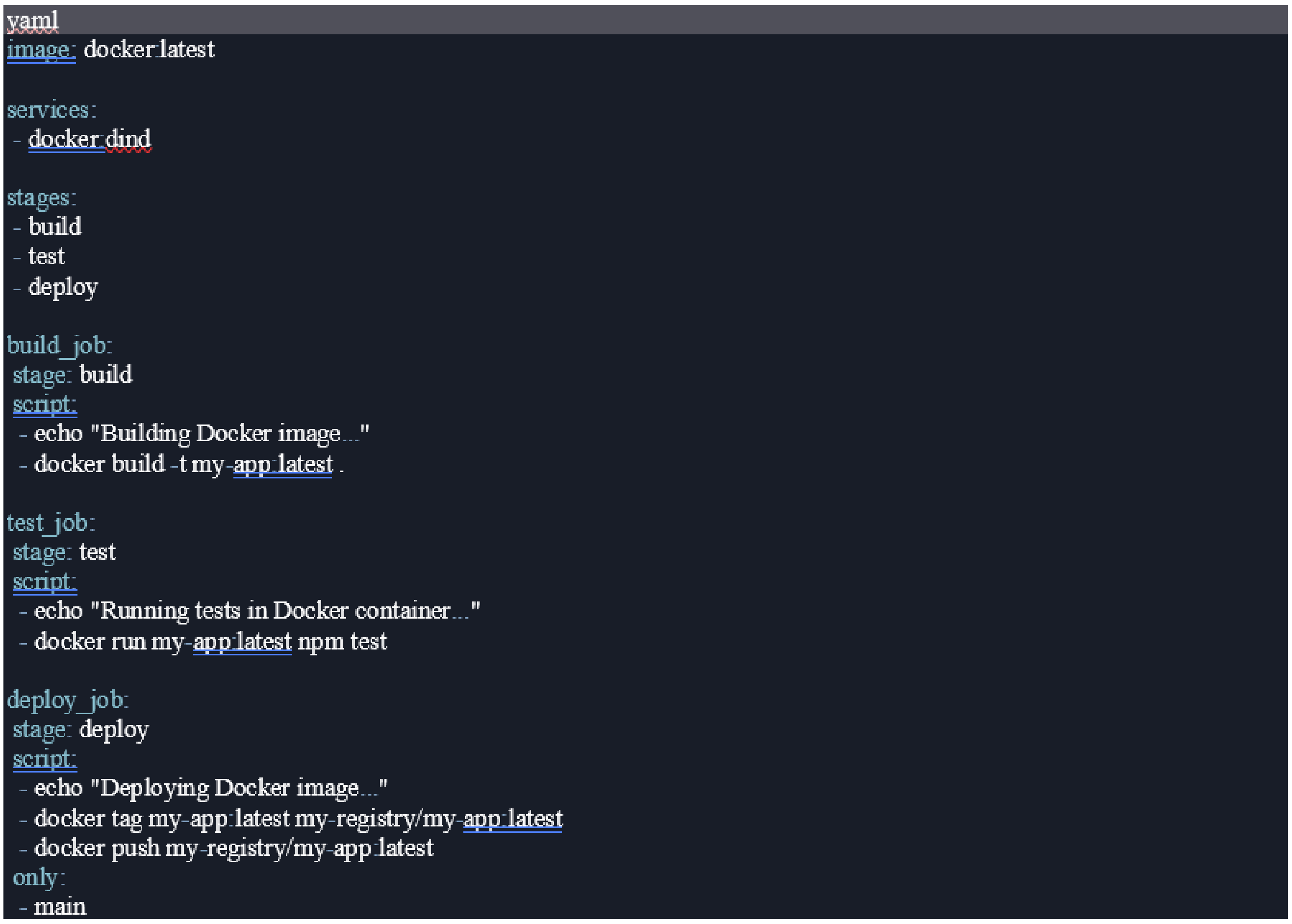


```
stages:
 - build
 - test
 - deploy

build_job:
 stage: build
 script:
  - echo "Building the application..."
  - npm install
  - npm run build
 artifacts:
  paths:
   - dist/

test_job:
 stage: test
 script:
  - echo "Running tests..."
  - npm test

deploy_job:
 stage: deploy
 script:
  - echo "Deploying the application..."
  - npm run deploy
 only:
  - main
```

This pipeline defines three stages: build, test, and deploy. This is the CI/CD process stage. It calls the build_job to install the necessary dependencies (npm install) and then build the app (npm run build). The dist/ directory will be captured using the artifacts directive, which holds the build artifacts, such as the built application code. The application is verified by unit tests using the npm test command on the test_job. The final stage, the deploy_job, will deploy and will be triggered only during the main branch (subject of the only keyword), ensuring the app will only be deployed after a successful test and build (Tegeler et al., 2029). For simple Node.js applications, this pipeline is ideal and is also a good fundamental starting point from which to build and deploy code on GitLab CI/CD.

5.2 Example 2: Docker-Based CI/CD Pipeline

Docker containers are usually used in modern application deployment because they offer portability and consistency. The example below shows how to create a Docker-based GitLab CI/CD pipeline to automate the build, test, and deployment within a Docker container. GitLab supports Docker and is further supported by Docker in Docker (Docker: did), which allows the running of Docker commands within the CI/CD pipeline.

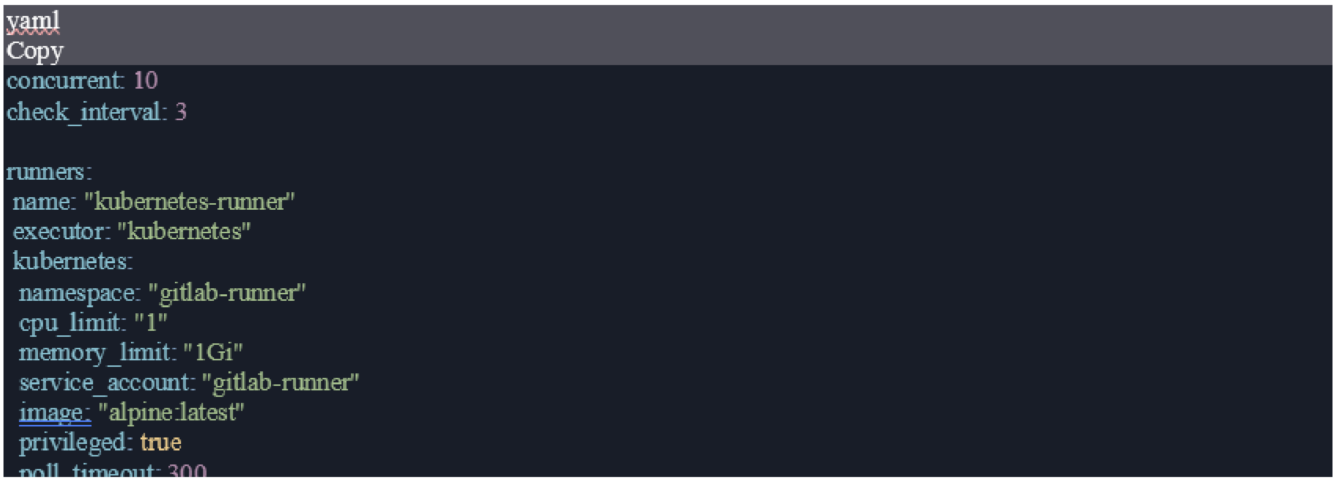


```
yaml
image: docker:latest

services:
 - docker:dind

stages:
 - build
 - test
 - deploy

build_job:
 stage: build
 script:
  - echo "Building Docker image..."
  - docker build -t my-app:latest .

test_job:
 stage: test
 script:
  - echo "Running tests in Docker container..."
  - docker run my-app:latest npm test

deploy_job:
 stage: deploy
 script:
  - echo "Deploying Docker image..."
  - docker tag my-app:latest my-registry/my-app:latest
  - docker push my-registry/my-app:latest
 only:
  - main
```

This pipeline defines three stages: build, test, and deploy. This is the CI/CD process stage. It calls the build_job to install the necessary dependencies (npm install) and then build the app (npm run build). The dist/ directory will be captured using the artifacts directive, which holds the build artifacts, such as the built application code. The application is verified by unit tests using the npm test command on the test_job. The final stage, the deploy_job, will deploy and will be triggered only during the main branch (subject of the only keyword), ensuring the app will only be deployed after a successful test and build (Tegeler, et al., 2019). For simple Node.js applications, this pipeline is ideal and is also a good fundamental starting point from which to build and deploy code on GitLab CI/CD.

5.3 Example 2: Docker-Based CI/CD Pipeline

Docker containers are usually used in modern application deployment because they offer portability and consistency. The example below shows how a Docker-based Docker GitLab CI/CD pipeline can be created to automate the build, test, and deployment within a Docker container. GitLab supports Docker and is further supported by Docker in Docker (Docker: did), which allows the running of Docker commands within the CI/CD pipeline.



```
yaml
Copy
concurrent: 10
check_interval: 3

runners:
 name: "kubernetes-runner"
 executor: "kubernetes"
 kubernetes:
  namespace: "gitlab-runner"
  cpu_limit: "1"
  memory_limit: "1Gi"
  service_account: "gitlab-runner"
  image: "alpine:latest"
  privileged: true
  poll_timeout: 300
```

In this configuration, the CI/CD pipeline operates in this manner since the concurrent: 10 directive allows 10 jobs to run concurrently so that the available resources do not exceed. With the check interval: 3 settings, the GitLab Runner runs an inspection at the 3-second interval. The runner's section defines the auto-scaling runner using the executor "Kubernetes" to scale the workload. To prevent resource overutilization, the Kubernetes configuration contains the resource limits for CPU (cpu_limit: "1") and memory (memory_limit: "1Gi"). The Kubernetes namespace in which the GitLab Runner runs is specified in that namespace: "GitLab."

The privileged: true directive also ensures that the runner has permission to run Docker commands in Utah Kubernetes. All poll_timeout: 300 means the number of seconds to wait to get job health status from Kubernetes. In this setup, a high volume of CI/CD jobs is launched by teams, and it is optimized for resource management so that GitLab Runners can scale dynamically and based on infrastructure usage. With implementation code examples, the basic gitlab ci/cd examples are use cases of gitlab ci/cd that can be implemented for building, testing, and deploying an application using basic pipeline configurations and auto scaling-based pipeline configurations. With these examples, it is clear that GitLab is flexible and covers modern DevOps best practices to let teams run their development processes with ease and high performance to scale easily. It can also be used as a base for several projects and may incorporate different organizational requirements.

6 Best Practices for Migrating to GitLab

When moving from legacy CI/CD tools to GitLab, following some important best practices is the best way to benefit from migration. By instituting these practices, the adoption will be smooth, the pipeline efficiency will be available, and the integration with the overall software development lifecycle process will be better (Shahin et al., 2017). Not only that, migration to CI/CD is a success if it optimizes the CI/CD process itself and leads to better collaboration and shorter time to market.

6.1 Thorough Assessment before Migration

The CI/CD pipeline is assessed before any migration process. Here, the enterprise makes sure it knows what it currently does, what toolchain it is using, and what pains need to be solved. The legacy system is assessed comprehensively, including identifying inefficiencies, bottlenecks, and failures. Another important assessment is the current toolset: version control, build automation, testing frameworks, and delivery strategies. Knowing these factors helps an organization choose the features and functions it wants to maintain or enhance that will operate in the new pipeline.

Beyond technical aspects, it is important to evaluate the organizational challenges to pave the way for a migration to GitLab. This includes determining which areas of teamwork lack and understanding any security protocol or automated process with a gap. Then, a detailed audit should be conducted, which could rely on a manual and metrics from the legacy system. This assessment will provide a roadmap of the migration plan so that the final solution's result aligns with the organization's needs and improves overall performance.

6.2 Prioritize Toolchain Simplification

Migrating to GitLab is a great incentive for consolidating numerous instruments on a solitary, unified stage. Most CI/CD legacy systems are built to be responsible for various tools for version control, testing, deployment, and monitoring.

Such fragmentation causes complexity and inefficiency in the workflow. Migrating to GitLab is a natural opportunity because it will streamline and simplify the toolchains, improving maintainability and reducing cost.

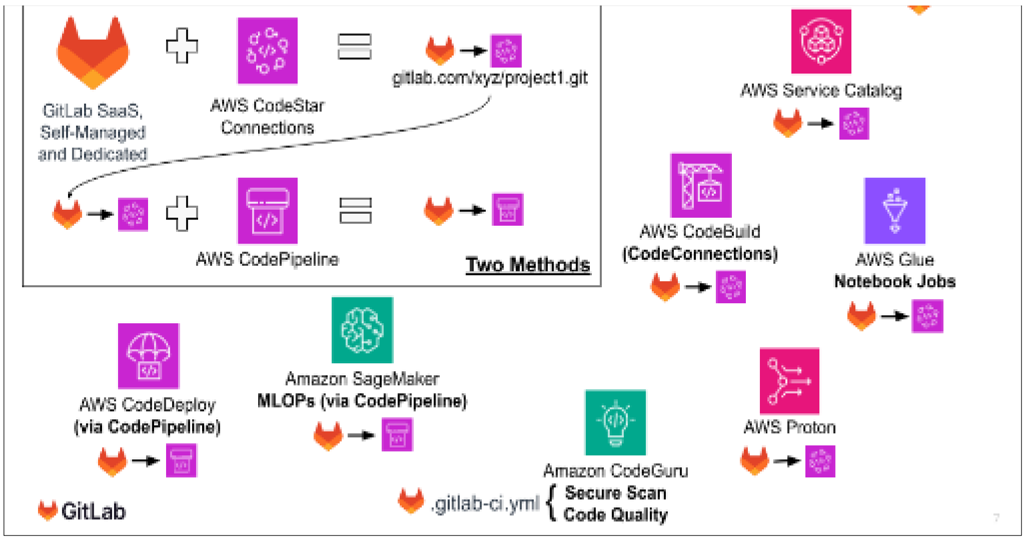

GitLab SaaS, Self-Managed and Dedicated
AWS CodeStar Connections
gitlab.com/xyz/project1.git
AWS CodePipeline
Two Methods
AWS Service Catalog
AWS CodeBuild (CodeConnections)
AWS Glue Notebook Jobs
AWS CodeDeploy (via CodePipeline)
Amazon SageMaker MLOPs (via CodePipeline)
AWS Proton
Amazon CodeGuru
.gitlab-ci.yml
Secure Scan
Code Quality
GitLab

Figure 8 Ultimate guide to migrating from AWS CodeCommit to GitLab

There is a need to avoid transferring the extra complexity of the legacy system when planning the migration. Enterprises should leverage GilLab's integrated version control, CI/CD, security, and monitoring tools. Consolidating all these functions into one such platform helps the enterprise lower error risks, lower its tool maintenance overhead (reduce the total number of tools to use), and improve the overall productivity of the pipeline. Moreover, the toolchain simplification facilitates scheduling and coordination across teams, lowers the effort of its configuration management, and eliminates tool incompatibilities.

6.3 Maintain Continuous Collaboration

CI/CD pipeline migration is complicated, and collaboration is especially essential. Transitioning to GitLab involves active development, operations, security, and other team involvement, and therefore, it is very important. Keeping these teams in constant contact renders cooperation easy as they can identify potential problems sooner and solve them immediately (Mattessich & Johnson, 2018). When the teams work together, regular meetings and feedback loops are important to keep everyone on the same page and match their respective goals with the objective of the migration.

After the migration phase, continuous collaboration should continue and be based exclusively on GitLab. Helping the adoption by making sure GitLab's features are used properly throughout teams and addressing any issues that come up quickly will go a long way towards making this adoption go as smoothly as possible. This environment encourages the sharing of best practices, expedites the settlement of any difficulties, and prevents migration from being a one-time event and becoming a constant evolution into a more expeditious development and deployment cycle.

6.4 Leverage GitLab's Automation Features

Another reason GitLab is popular is its automation abilities, which assist with the efficiency of the CI/CD pipeline. Automation decreases the requirement for manual intervention, shortens the time to market, and reduces the chance of human error (Sebok & Wickens, 2017). As an enterprise, one should deploy all the automation that comes with GitLab, such as automated testing, deployment, infrastructure provisioning, monitoring, etc.

Continuous automated testing mitigates defects, making it to production by validating that code changes are tested. Enterprises can build and test pipelines automatically with integrated CI/CD tools and get real-time feedback from the pipeline results to developers. Automated deployments mitigate the possibility of errors, at least through the release process of code deployment in various environments. Infrastructure provisioning can also be automated, as well as dynamic scaling and best usage of resources.

The more an enterprise automates in GitLab, the more reliable and consistent its pipeline will become. Faster iteration cycles are good for teams that want to test and deploy software without being bored by manual tasks, and they can also be automated. Thus, playing GitLab's automation cards is key to sustainable development speed.

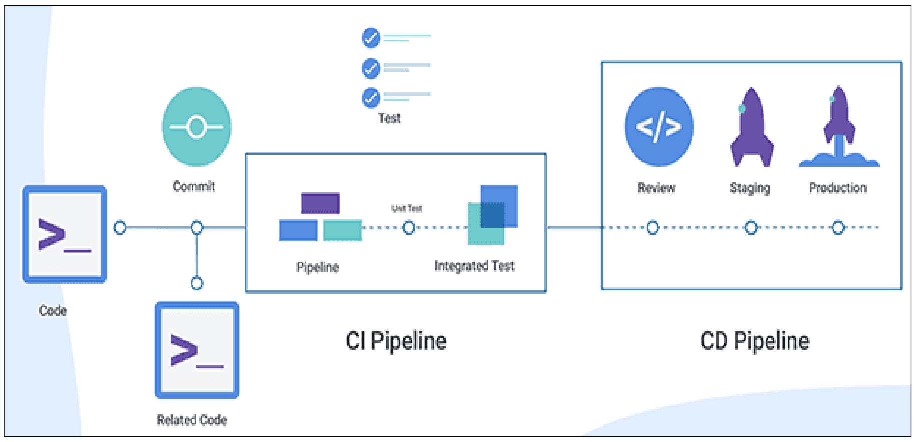


Figure 9 Automated Testing Pipeline with GitLab CI

6.5 Invest in Training and Support

If the teams are helping with a successful migration to GitLab, they cannot be forced to do so without investing in their training and support. Operations teams and developers can no longer be unfamiliar with how GitLab works—having a comprehensive set of hands-on training will enable teams to use the new platform effectively.

Enterprises must provide formal training programs, workshops, and self-paced learning resources. GitLab's official documentation, community support, and online forums are valuable sources for helping teams resolve issues and learn new features. Having internal GitLab champions or experts to help onboard and troubleshoot during the early stages of adoption also makes such a transition much easier (Hsu & Patterson, 2020). Ongoing support cannot be stressed enough. Inevitably, a team will begin to utilize GitLab, where questions and challenges are bound to emerge. Experts will be available to support them as the operation progresses, and check-ins will be regular. By updating training materials with the new features and best practices of GitLab, teams will also be able to be more productive and gain more confidence in using GitLab.

6.6 Start Small and Scale Gradually

Starting with the small pilot project is one of the best ways to migrate to GitLab, as this is a first step that can scale over the entire organization. The enterprise can try out its GitLab capabilities in a safe environment to discover initial challenges and workarounds before adoption. Picking a project representative of typical development processes is important because there is nothing else to lean on to know how GitLab behaves under real-world conditions.

Adapting to the new system can be done small, implementing new ways and addressing issues the team can tackle independently before moving on to bigger projects. It also offers the chance to gain real-world feedback from teams to reflect on training materials and processes. When the pilot project works, the migration can be expanded to other projects and the company. Slowly scaling allows the team to cope and walk away with a small win for each failure, covering issues without disposing of teams to death in transitions.

6.7 Monitor and Continuously Improve

After the migration, to evaluate the CI/CD pipeline, one needs to continuously monitor its functioning to see how well it is optimized. With powerful monitoring and analytics features, GitLab enables teams to track metrics like the total minutes spent building the application, how often the application was deployed, the error rates, and the total resources utilized (Eraslan et al., 2020). These are important insights to determine where the pipeline is performing well and where it may require improvement to meet performance goals.

This should be accompanied by ongoing feedback loops, allowing teams to give some input on how the system is performing and what difficulties they encounter. The organization's culture in the company, embedded with the CI/CD pipeline, should continue to improve and change as the company evolves. Refinements of workflows, automation, and new security might occur throughout iterative improvement (Argyropoulos et al., 2020). The migration to GitLab has many benefits, but its success depends on how it is migrated. With thorough assessment, toolchain simplification, continuous collaboration, leveraging automation, investing in training, starting small, and continuous improvement, enterprises can overcome the challenges of going from their existing process to GitLab and reap greater speed, reliability, and developer productivity.

7 Future and Emerging Trends in CI/CD

CI/CD pipelines are also changing as software development practices evolve. The future of CI/CD will depend on integrating the latest technologies and methodologies to enhance inference quality, automation, and overall software efficiency.

7.1 AI and Machine Learning in CI/CD

Artificial Intelligence (AI/ML) is being added to nearly every CI/CD pipeline to automate many manual processes and integrate speed in software development cycles. Traditional CI/CD workflows entail manually running many tasks, such as code analysis, testing, and bug detection, resulting in errors and delays (Boda, 2019). With AI, many of these processes can be automated, meaning there is much less human intervention speeding the process down.

The fastest way to detect bugs is to use large volumes of code and data, which some of the AI and ML algorithms can do. Through historical data and patterns, machine learning models can predict areas of code that are more likely to have defects (including the likelihood of having defects) so that the testers can focus their time on areas of code that are less likely to be defective in order to reduce the time spent coding review manually. AI-driven tools improve code quality by identifying subtle issues that regular testing methods cannot notice.

AI can also be incorporated into the overall software development lifecycle (SDLC) for streamlining their workflows. Given the lack of inventory on continuous deployment, AI-driven CI/CD tools provide predictive capabilities to know about future bottlenecks or delays in the pipeline and provide suggestions for improvements (Felstaine & Hermoni, 2018). It enables development teams to take action proactively to minimize deployments, increase lead time, and increase productivity. To continue accelerating software delivery at scale, it is increasingly valuable to integrate AI and machine learning into CI/CD pipelines to automate complex tasks and improve software quality at scale.

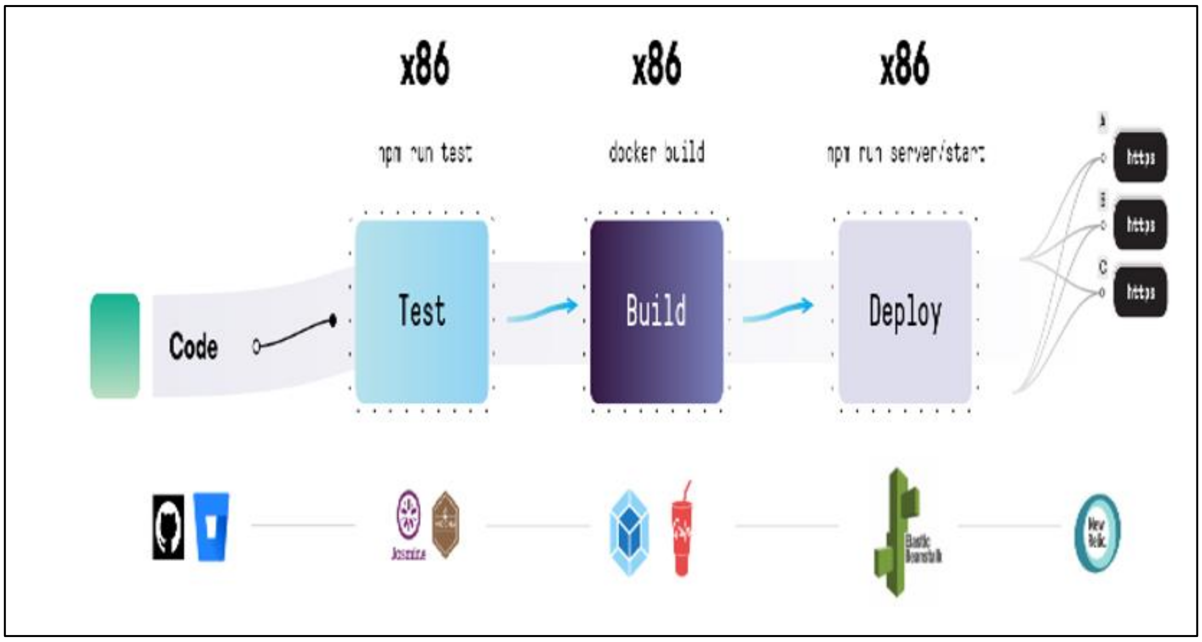


Figure 10 CI/CD for Machine Learning & AI

7.2 GitOps and Infrastructure as Code (IaC)

The next wave of CI/CD pipeline automation is GitOps and Infrastructure as Code (IaC). It will eliminate the pain of managing infrastructure and deployments. In GitOps, application code and infrastructure configurations are the Source of Truth for the single Git repository. Users can use this approach to version control and manage all of their infrastructure in the same way they manage application code.

When enterprises adopt GitOps, infrastructure deployments will be consistent, repeatable, and auditable. In the past, all infrastructure changes frequently occurred, often unnoticed. Git repositories became the central hub for all team changes, and the ability to track and roll back to the previous configuration was just a Git operation. It diminishes the possibility of human error but also assures that the infrastructure is always consistent with the application code, even during deployments.

Infrastructure as Code (IaC) continues this idea by allowing infrastructure to be defined and managed from code instead of directly specifying it in the infrastructure code (Chinamanagonda, 2019). Organizations can use IaC tools like Terraform, Ansible, or CloudFormation to declaratively provision and manage the infrastructure. Consequently, this accelerates the deployment process, improves the interactions of developers and operations teams, and facilitates the management of cloud environments. Combining GitOps and IaC into CI/CD pipelines increases the scalability and flexibility of deployments, including multi-cloud and hybrid cloud infrastructure.

7.3 Continuous Testing and DevSecOps

Continuous testing regularly influences the formation of modern CI/CD pipelines, which provide faster and more reliable software delivery. In the past, testing was treated as a separate phase (of the software development life cycle) after completion. With CI/CD coming in, testing must be incorporated into each pipeline stage so that the defects are caught early, and quality is guaranteed during the development project (Jokinen, 2020). Automated testing is the most important aspect of allowing continuous testing in CI/CD pipelines. From the beginning of the development process to commits of code, builds, and deployments, automated tests are run to ensure that the code being produced is valid and working, fit for purpose, and carried out efficiently. Continuous testing provides rapid feedback and promptly catches bugs without a large barrier so bugs do not slip through to production.

DevSecOps is also an integral part of CI/CD pipelines. The DevSecOps approach incorporates security into the CI/CD pipeline, and project security concerns are addressed not only after development but throughout the whole software development lifecycle. The proactive security approach, in this case, prevents security vulnerabilities from arising at all or at least from being discovered until late in the development phase, which minimizes the risk of a security breach and data leak. Security testing must also be incorporated into the CI/CD pipeline by automating security scans, vulnerability assessments, and compliance checks. Organizations 'security testing tools, like static application security testing (SAST) and dynamic application security testing (DAST), help them discover security vulnerabilities in code and infrastructure (Mateo et al., 2020). A shift left in the security approach helps the entire security posture of applications and makes security an integral part of the DevOps culture.

7.4 Cloud-Native CI/CD

As more enterprises embrace cloud-native architectures, they require the capability to leverage the full scale of cloud environments. In that context, CI/CD pipelines need to benefit. Cloud-native CI/CD pipelines are built to take advantage of the availability of cloud resources, such as elastic computing, storage, and container management programs, such as Kubernetes (Karslioglu, 2020). This is where organizations can come and adopt cloud-native CI/CD solutions to ensure their software delivery process can be scaled, resilient, and optimized for the cloud.

A very important part of cloud-native CI/CD is the usage of containers such as Docker and the orchestration of containers using platforms such as Kubernetes. Developers can package their application along with its dependencies in a container, allowing for consistency in software that will run anywhere, from development to staging to production. This consistency guarantees that software behaves as expected, regardless of where it is deployed, reduces the risk of deployment issues, and improves reliability.

Cloud-native CI/CD pipelines also support faster release cycles by leveraging the elasticity of cloud resources (Laszewski et al., 2018). These pipelines have auto-scaling capabilities and can cope with sudden changes in demand, completing builds and tests quickly even with high traffic. Since this allows teams to scale their resources up or down on demand, one can keep the application performance consistent and reduce the application's time to market.

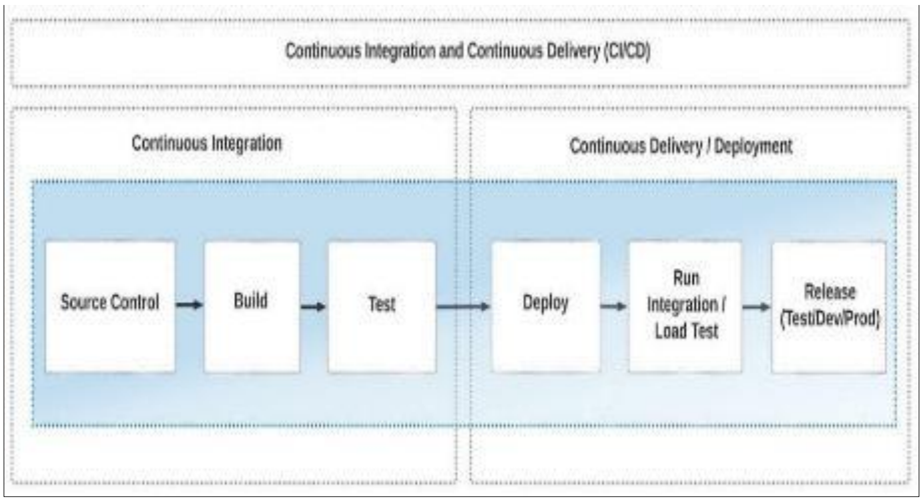

Continuous Integration and Continuous Delivery (CI/CD)
Continuous Integration
Continuous Delivery / Deployment
Source Control
Build
Test
Deploy
Run Integration / Load Test
Release (Test/Dev/Prod)

Figure 11 CI/CD for Cloud-Native Applications

7.5 Blockchain and CI/CD

For the compliance and security-driven industries, blockchain technology is entering the picture to secure and verify CI/CDs. Although there are several approaches to enhance transparency, integrity, and traceability in the software development lifecycle, its distributed ledger technology has a special characteristic to provide blockchain transparency, traceability, and integrity. Blockchain can track and verify each step in the software delivery process, from changes to code to builds, tests, and deployments, as it is used in CI/CD (Saarenpää, 2020). By recording these steps using blockchain, organizations can establish an immutable, auditable history of every action taken in the pipeline. This second approach offers great transparency to trace the issue or breach to its source.

The decentralized nature of the blockchain also secures the data because if no party controls all the data, it can be traced back. The ability to govern, store, and recall data according to data laws is a particularly valuable feature (particularly for finance, healthcare, and government-based industries).

7.6 Quantum Computing and CI/CD

Despite being in its early stages, this is where quantum computing could play a major role in the future of CI/CD with applications relating to optimization testing and performance. In contrast to classical computers, quantum computers use the principles of quantum mechanics to speed up the process of performing complex calculations, rendering them well suited to the problem of processing huge data and solving types of optimization problems that are currently computationally intractable. Quantum computing could be used to optimize test case selection in the context of CI/CD so that all critical tests are run first, which will accelerate the overall test time (Sivaraman, 2020). Quantum algorithms would also allow code to be optimized to improve application efficiency and performance. Because quantum computing speeds up performance bottleneck identification, it could also speed up iterations and more reliably build.

When quantum computing technology matures, it will profoundly affect how CI/CD pipelines are designed and operated. While there are still years to go from practical applications of quantum computing in CI/CD, the potential for improving the software development process is huge in any case where computational power is limited. Several emerging technologies are set to impact the future of CI/CD pipelines, each designed to improve software delivery, automation, and security with continuous testing, cloud-native infrastructure, quantum computing, AI, GitOps, and blockchain. The evolving needs of CI/CD strategies will be to adapt to the same and stay competitive in an increasingly complex software landscape.

8 Conclusion

Today, within the software development climate, enterprises are prioritizing modernizing continuous integration and deployment (CI/CD) pipelines. Legacy CI/CD tools that have served their purpose to some extent have become a cumbersome barrier to scalability, efficiency, and integration with modern development practices. Due to their often legacy nature, inflexibility, and lack of support for the complexity of cloud-native applications, these legacy systems cannot easily be extended to provide consistent, organic DevOps experiences supporting these projects. The cause is that businesses that wish to remain competitive are moving towards more advanced and unified solutions such as

GitLab to stay forward and scalable. This study has shed some light on the challenges faced by enterprises that run these poor CI/CD systems. Legacy tools include fragmented toolchains, lack of scalability, lack of integration with cloud platforms, and high maintenance costs, which are barriers to the speed and efficiency demanded by modern software delivery models. In addition, these legacy systems are no longer able to work with modern architecture, such as microservices, containerized environments, and so on.

A unified DevOps platform that solves these businesses' challenges is GitLab. Consequently, it combines version control, build automation, testing, deployment, and monitoring into one platform, alleviating the problems of disparate toolchains. Having scalability with AWS and native support for cloud-native technologies such as Kubernetes on GitLab makes it easier to deliver modern application needs to enterprises. The platform's automated updating further cuts down on maintenance so that teams can spend time innovating rather than trying to debug outdated systems. Enterprise migration to GitLab is shown to have tangible benefits, such as this modern CI/CD solution. The migration brought great benefits in terms of deployment frequency, lead time, and operational economy. Deployment frequency went from one time a week to as fast as per day, and lead time went from two weeks to two days. Maintenance was scaled down by 50% as well, as did the operational advantages of GitLab's integrated offering.

Future trends include the application of AI and ML in CI/CD, CI/CD with GitOps and IaC, continuous testing and DevSecOps integration, and increasing reliance on the cloud-native CI/CD pipelines, all of which will further improve the efficiency and reliability of the access to the software. Moreover, the future holds an exciting potential for development based on using quantum computing to optimize testing and performance in CI/CD workflows. In the coming years, blockchain will play a role in providing integrity and secure deployments. With the integration of more advanced automation tools, CI/CD will continue to change. Modernizing the CI/CD pipeline is important in the digital age when it intends to stay competitive as a company. GitLab migration offers a powerful, scalable, and integrated solution beyond the speed and quality of software delivery, allowing businesses to adjust to changing technology trends. This paper provides a framework with some valuable insides and strategies for organizations migrating to GitLab from Legacy tools because it provides the speed up and reliability of software delivery and keeps a leading edge in an increasingly dynamic software development environment

References

[1] Argyropoulos, N., Mouratidis, H., & Fish, A. (2020). Enhancing secure business process design with security process patterns. Software and Systems Modeling, 19(3), 555-577.

[2] Bansal, A. (2015). Energy conservation in mobile ad hoc networks using energy-efficient scheme and magnetic resonance. Journal of Networking, 3(Special Issue), 15. https://doi.org/10.11648/j.net.s.2015030301.15

[3] Belmont, J. M. (2018). Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI. Packt Publishing Ltd.

[4] Boda, V. V. R. (2019). CI/CD in FinTech: How Automation is Changing the Game. Journal of Innovative Technologies, 2(1).

[5] Bogner, J., Fritzsch, J., Wagner, S., & Zimmermann, A. (2019, March). Microservices in industry: insights into technologies, characteristics, and software quality. In 2019 IEEE international conference on software architecture companion (ICSA-C) (pp. 187-195). IEEE.

[6] Bondarenko, K. I. (2020). System of continuous software development using cloud technologies. https://dspace.nau.edu.ua/bitstream/NAU/47662/1/%D0%A4%D0%9A%D0%9A%D0%9F%D0%86_123_2020_%D0%91%D0%BE%D0%BD%D0%B4%D0%B0%D1%80%D0%B5%D0%BD%D0%BA%D0%BE%20%D0%9A.%D0%86.pdf

[7] Chinamanagonda, S. (2019). Automating Infrastructure with Infrastructure as Code (IaC). Available at SSRN 4986767.

[8] Chinamanagonda, S. (2020). Enhancing CI/CD Pipelines with Advanced Automation-Continuous integration and delivery becoming mainstream. Journal of Innovative Technologies, 3(1).

[9] Dattatreya Nadig, N. (2019). Testing resilience of envoy service proxy with microservices.

[10] Engwall, K., & Roe, M. (2020). Git and GitLab in library website change management workflows. code4lib Journal, (48).

[11] Eraslan, S., Kopec-Harding, K., Jay, C., Embury, S. M., Haines, R., Ríos, J. C. C., & Crowther, P. (2020). Integrating GitLab metrics into coursework consultation sessions in a software engineering course. Journal of Systems and Software, 167, 110613.

[12] Evertse, J. (2019). Mastering GitLab 12: Implement DevOps culture and repository management solutions. Packt Publishing Ltd.

[13] Felstaine, E., & Hermoni, O. (2018). Machine Learning, Containers, Cloud Natives, and Microservices. In Artificial Intelligence for Autonomous Networks (pp. 145-164). Chapman and Hall/CRC.

[14] Hsu, A., & Patterson, R. (2020). CASE STUDY OF SOFTWARE DEVELOPMENT IN THE DOD (Doctoral dissertation, Monterey, CA; Naval Postgraduate School).

[15] Jokinen, O. (2020). Software development using DevOps tools and CD pipelines, A case study. Helsingin yliopisto, 54.

[16] Karamitsos, I., Albarhami, S., & Apostolopoulos, C. (2020). Applying DevOps practices of continuous automation for machine learning. Information, 11(7), 363.

[17] Karslioglu, M. (2020). Kubernetes-A Complete DevOps Cookbook: Build and manage your applications, orchestrate containers, and deploy cloud-native services. Packt Publishing Ltd.

[18] Koopman, M. (2019). A framework for detecting and preventing security vulnerabilities in continuous integration/continuous delivery pipelines (Master's thesis, University of Twente).

[19] Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. International Journal of Computational Engineering and Management, 6(6), 118-142. Retrieved from https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf

[20] Laszewski, T., Arora, K., Farr, E., & Zonooz, P. (2018). Cloud Native Architectures: Design high-availability and cost-effective applications for the cloud. Packt Publishing Ltd.

[21] Mateo Tudela, F., Bermejo Higuera, J. R., Bermejo Higuera, J., Sicilia Montalvo, J. A., & Argyros, M. I. (2020). On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. Applied Sciences, 10(24), 9119.

[22] Mattessich, P. W., & Johnson, K. M. (2018). Collaboration: What makes it work.

[23] Mehtonen, V. (2019). Research on building containerized web backend applications from a point of view of a sample application for a medium sized business.

[24] Nyati, S. (2018). Transforming telematics in fleet management: Innovations in asset tracking, efficiency, and communication. International Journal of Science and Research (IJSR), 7(10), 1804-1810. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203184230

[25] Nzanywayingoma, F., & Yang, Y. (2019). Efficient resource management techniques in cloud computing environment: a review and discussion. International Journal of Computers and Applications, 41(3), 165-182.

[26] Orazi, G., Vallittu, K., Sainio, P., & Virtanen, S. (2020, September). Enhancing and integration of security testing in the development of a microservices environment.

[27] Podjarny, G. (2017). Securing Open Source Libraries. O'Reilly Media, Incorporated. https://go.snyk.io/rs/677-THP-415/images/OReilly%20Securing%20Open%20Source%20Libraries.pdf

[28] Rangnau, T., Buijtenen, R. V., Fransen, F., & Turkmen, F. (2020, October). Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC) (pp. 145-154). IEEE.

[29] Saarenpää, J. (2020). Creating an Azure CI/CD pipeline for a React web application.

[30] Sebok, A., & Wickens, C. D. (2017). Implementing lumberjacks and black swans into model-based tools to support human–automation interaction. Human factors, 59(2), 189-203.

[31] Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE access, 5, 3909-3943.

[32] Sivaraman, H. (2020). Machine Learning for Software Quality and Reliability: Transforming Software Engineering. Libertatem Media Private Limited.

[33] Tegeler, T., Gossen, F., & Steffen, B. (2019, January). A model-driven approach to continuous practices for modern cloud-based web applications. In 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence) (pp. 1-6). IEEE.

[34] Williams, L. (2019). Secure software lifecycle knowledge area issue. The National Cyber Security Center.