Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2021, 8(10):104-108

Research Article ISSN: 2394 - 658X

Continuous Integration and Continuous Deployment (CI/CD) in Full-Stack Development

Prathyusha Kosuru

Project Delivery Manager

ABSTRACT

CI/CD is one of the fundamental processes in the modern full-stack development that enables teams to build and release robust applications much faster, with minimal errors. CI/CD pipelines involve the automation of build, test, and deploy processes; it minimizes human intervention in the deployment process, leading to the improvement of efficiency. This paper aims at covering topics such as understanding CI/CD, the tools that can be used to implement it, common use cases/considerations and ways that demonstrate its usage (El Khalyly et al., 2020).

Keywords: Continuous Integration (CI), Continuous Deployment (CD), CI/CD pipelines, Full-stack development automation, Build automation tools, Automated testing in CI/CD, Deployment automation, Jenkins.

INTRODUCTION

CI/CD has become a crucial practice in the software industry due to the demand for more enhanced and faster cycles in the developmental process. Such practices allow development teams to identify problems before they escalate, streamline time-consuming processes, and release new versions of software more frequently. Especially in the context of full-stack development wherein front-end and back-end code need to be integrated, having a properly functioning CI/CD pipeline is even more crucial. By using these techniques, developers will gain valuable insights into strengthening their applications, ensuring they are robust, adaptable, and able to meet the evolving demands of the market (MAESTRONI, 2020).

INTRODUCTION TO CI/CD IN FULL-STACK DEVELOPMENT

CI/CD are some of the fundamental concepts used in current full-stack development to improve the delivery of quality solutions to customers. CI consists of updating codes from various developers or contributors in a shared repository and identifying problems through successive builds and tests. CD takes this a step further by automating the way code is actually pushed up to production to the point where updates are happening frequently and there is not much manual work involved in making it so. CI/CD processes are used in full-stack applications to ensure the smooth integration of newly developed front-end and back-end requirements, repeated cycles, regularity, and better generalization of the application (Mattila, 2018).

KEY CONCEPTS OF CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT

Continuous Integration (CI) is built around the process of integrating code changes into a central repository frequently and verifying those changes through tests. This approach helps to prevent integration issues from being worse should they occur. Continuous Integration (CI) is further expanded by Continuous Deployment (CD) which deploys the code to production as soon as it is tested and passes all tests, thus enabling frequent and accurate changes. CI and CD share common features that include automation, consistency, and the ability to provide fast feedback, which are crucial elements in both methods that enhance the quality of code and speed up development. When these practices are incorporated, it becomes possible to enhance the way work is done in the team to yield efficient and quality deliverables (Pratama & Kusumo, 2021).

BENEFITS OF IMPLEMENTING CI/CD PIPELINES IN FULL-STACK PROJECTS

Using CI/CD pipelines in full-stack projects has a lot of advantages. It improves code quality through building and testing automation which allows for early bug detection. Continuous release processes are another attribute which offers rapid updates and frequent feature delivery. CI/CD pipelines also enhance communication between the front-end and the back-end teams since the code changes are integrated constantly to avoid combining two conflicting solutions at the end. Furthermore, they reduce the Configuration variation between different environments due to the use of automated configuration and orchestrated deployment. As a whole, the CI/CD pipeline makes the processes more effective and consistent and allows the teams to concentrate on their creativity and product development while being confident in the quality and stability of the software being released.

POPULAR CI/CD TOOLS FOR FULL-STACK DEVELOPMENT (JENKINS, GITLAB CI, CIRCLECI)

When it comes to the creation of full-stack CI/CD pipelines, there is a vast number of tools to choose from and they all come equipped with certain functionalities and compatibility options. Jenkins is an open-source automation server that is widely used for implementing CI/CD pipelines. It is very flexible and is compatible with a vast array of plugins throughout the pipeline process. It is important to mention that Jenkins can be utilized for numerous large-scale projects where flexibility and adjustability are needed.

It’s worth to note that GitLab CI is the CI/CD solution integrated directly into the GitLab environment. It can be easily integrated with git repositories and provides functionalities to automate, test and deploy the application. GitLab CI is a good value if a team chooses GitLab as a version control tool because it offers a full set of tools in one platform for managing a software development process.

CircleCI is another CI/CD tool that provides many features similar to other providers, though it especially emphasizes speed and convenience. CircleCI also goes well with the containerization platforms such as Docker which can also be used to make the applications portable and runnable in various environments (Rangnau et al., 2020).

AUTOMATING BUILD PROCESSES IN CI/CD PIPELINES

Continuous Integration and Continuous Delivery are crucial features of full-stack development, where among the most important tasks is the automation of the build process. The build phase compiles the code and repackages it into deployable form for testing and / or distribution. Automating this phase eliminates manual builds by developers since this process is lengthy and prone to errors. In full-stack development, building involves compiling front-end Javascript frameworks such as React and Angular as well as dealing with the networks and server aspects which can be built using Network and Server technologies such as Node. js, Java, or Python. These procedures are intended to deliver a smooth interaction in between the front and back end of the application. Jenkins and GitLab CI are CI/CD tools that each time code is pushed to the repository it triggers builds that will validate each commit.

Moreover, containerization platforms such as Docker also contribute to the automation of the build process through packaging of the application and its dependencies into containers. This makes it possible to have a uniform approach to application development across various environments such as local development, test environments, and production environments. Docker also reduces the complexity of the build pipeline through guaranteeing the homogeneity of the application behavior regardless of the target environment (Shahin et al., 2017).

IMPLEMENTING AUTOMATED TESTING IN FULL-STACK CI/CD WORKFLOWS

In any CI/CD pipeline, it is critical to incorporate testers to check the quality and stability of the code. Testing covers all the layers of projects in full-stack projects and implies not only the testing of front-end and back-end systems but also their integration. Then, integration tests are meant to check if the smaller units combined together work in the required manner. For instance, someone can test API endpoints on the back end, and make sure the UI components are fine on the front end to ensure the integration is strong.

End-to-end tests ensure that all sub components – front-end and back-end – function as a whole and that data moves between the front-end and the server in the correct manner. They cover the entire functionality of the application as it is from the user interface point of view. Selenium or Cypress are examples of the popular testing tools used for end-to-end testing in full-stack applications.

Including automated testing into the CI/CD process makes it possible for every build to be tested before going for deployment. This process ensures that certain bugs do not reach production and frees developers to isolate problems and solve them before they snowball. Sometimes, test coverage tools may be run in the pipeline to monitor the test coverage percentage which serves as a reminder to developers to keep their code testable (Singh et al., 2019).

ENSURING CODE QUALITY WITH STATIC CODE ANALYSIS AND LINTING

Code review and the use of linters are critical activities for ensuring high levels of code quality within the CI/CD process. It is the approach of analyzing a piece of code for several factors such as errors, bugs, code smells as well as coup’s violation without actually running the code. Other examples of automation tools for code review include

SonarQube for detecting security flaws and code complexity, Checkstyle for opinionated code quality, and ESLint for code style checks. Static analysis as a part of the CI process helps to identify the issues to avoid getting them to the production.
Linting works in conjunction with static analysis by targeting coding style and ensuring compliance with specific standards. For instance, code formatters such as Prettier for JavaScript or linting tools like Pylint for Python to make sure all the code is formatted and written in a consistent manner (El Khalyly et al., 2020).

BEST PRACTICES FOR SETTING UP CI/CD ENVIRONMENTS

CI/CD pipeline establishment is an activity that should be planned properly to guarantee its effectiveness. Divide the stages of the business pipeline into build, test, and deploy stages to ensure a logical progression of work. Integrate version control systems such as Git to handle code changes effectively and also create seamless pipeline executions upon commits or pull requests.
Separate teams for each stage of the pipeline. For example, it is appropriate to use different hosting or operating environments for development, staging, and production to prevent intermixing and achieve stability. There are tools such as Docker, which ensures that these environments are made standard since the same environment can be established several times.
Minimize manual intervention by automating tasks wherever feasible. This increases the dependency on pipelines to include tools that help manage dependencies, builds, and deployment, as well as features for logging and monitoring for performance and diagnostics. Periodically assess pipeline configurations and make any necessary modifications based on changes in project needs and improvements in technology (MAESTRONI, 2020).

MANAGING ENVIRONMENT VARIABLES AND SECRETS IN CI/CD

Environmental variables and secretes management is important for securities and configurations in CI/CD pipeline. Environment variables hold state that can be different based on the environment, like a database URL or an API key. Employ third-party tools such as HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, etc., for the storage and protection of secrets.
Do not directly embed secrets into the source code or into the configuration data. However, introduce secrets at runtime during the pipeline execution with help of secure functions available in CI/CD systems. The environment variables and secrets should not be changed by anyone but some specific group of users, thus access controls should be implemented (Mattila, 2018).

DEPLOYING FULL-STACK APPLICATIONS ACROSS MULTIPLE ENVIRONMENTS

It is vital to have a strategy on how best to roll out full-stack applications across different environments. With CI/CD solutions, it is possible to automate the process of deploying to development, staging or even a production environment. Use blue-green deployments or canary releases in order to prevent disruptions and reduce the chance of deploying a problematic code.
Make sure that each environment is properly set up with environment specific configurations and dependencies. Utilize IAC tools, such as Terraform, or Ansible to ensure that infrastructure is provisioned similarly across the various environments.
Always perform frequent releases in staging environments that are slightly similar to production to detect problems before reaching the users. Continuously initiate performance and availability checks on all deployments and have procedures for reversal in case the deployment was unfruitful (Pratama & Kusumo, 2021).

MONITORING AND LOGGING IN CI/CD FOR FULL-STACK APPLICATIONS

Real-time continuous monitoring and logging are essential for sustaining the integrity and efficiency of full-stack applications from delivery to production during all stages. Monitoring is the process of observing how applications and infrastructure are performing in real-time as well as their operational characteristics. Prometheus, Grafana, and Datadog are examples of tools that give visibility into the system state and user activity to identify problems and slow performance.
Logging goes hand-in-hand with the monitoring process through the method of recording occurrences in the application and pipeline execution. Software like the Elasticsearch Logstash Kibana (ELK) system or Splunk compile logs from a variety of sources, allowing for the review and resolution of problems. These platforms offer real-time data and help organizations to detect and mitigate issues faster. Logging aids in identifying problems, comprehending user activity, and addressing deficiencies in the pipeline or application to avoid future problems. The incorporation of monitoring and logging into the CI/CD workflows ensure potential problems in operations are identified and addressed quickly (Rangnau et al., 2020).

TROUBLESHOOTING AND DEBUGGING CI/CD PIPELINES

Some of the issues that require attention in the CI/CD pipelines include glitches that are noticed in stages like build, test, and deployment. This problem-solving process helps maintain the pipeline integrity allowing the flow of software integration and delivery to proceed without hindrance. Some of the most frequent issues are build and test failures, as well as deployment issues. Check the various logs in the pipeline to determine the precise moment that a problem emerged. It brings out logs that contain error messages and stack traces that can aid in solving the issue.

Debugging is more effective when the problematic behavior is reproduced in a sandbox environment. This may entail rerunning the failing tests on the developer's local machine or debugging the build process. Look for problems like pipeline settings that are wrongly specified, dependencies that are no longer updated, or contrasting changes to the code that can have an impact on the build or deployment cycle.

Incorporate effective error handling and notifications in the pipeline in order to promptly let the respective teams know about the failure. Some steps like with the use of Slack integrations or email notifications are practical approaches to ensure that some individuals get information at once. Application of lessons learnt in improving the pipeline after the occurrence of some problems also have the effect of improving the reliability of the pipeline and its efficiency (Shahin et al., 2017).

CASE STUDY: IMPLEMENTING A CI/CD PIPELINE FOR A FULL-STACK E-COMMERCE APPLICATION

A case study of an operational use case is establishing the CI/CD pipeline for a full-stack e-commerce application. The pipeline begins with source code management, where code changes are committed to a Git repository by a developer. CI tools such as Jenkins are used to build applications where front-end scripts are compiled and back-end services are built. Automated tests are run in order to prevent regressions with new code which causes tests to fail.

Continuous Deployment is organized through utilities such as GitLab CI that automatically deploy code updates to staging and production environments. The deployment includes packaging the application with Docker, infrastructure with Terraform, and updates with Kubernetes. Application performance tracking and logging like Datadog and ELK Stack is employed to keep an eye on the application and its logs. The pipeline also has rollback mechanisms for handling deployment failures in an orderly manner (Singh et al., 2019).

CONCLUSION

CI/CD also brings numerous advantages, especially in full-stack projects where the quality of the code increases, the rate and efficiency of deployment rises, and collaboration happens among team members. Many of these advantages enable the development process to be less cumbersome, leading to improved project implementation and completion. Through using CI/CD tools, automating build and deploy tasks, and maintaining and following the best practices for environments, developers make the development life cycle more efficient. Supervising and record keeping are vital in ensuring the well-being of applications whereas proper diagnosing and debugging is vital in solving problems. It shows how the principles above can be applied in the real-world, illustrating the advantage of a properly constructed CI/CD pipeline and its positive impact on the quality and uptime of software systems. CI/CD practices are critical for any development team that seeks to adopt new strategies to improve the projects constantly that are being developed (Zampetti et al., 2017).

REFERENCES

[1]. El Khalyly, B., Belangour, A., Banane, M., & Erraissi, A. (2020, December). A new metamodel approach of CI/CD applied to Internet of Things Ecosystem. In 2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS) (pp. 1-6). IEEE.

[2]. MAESTRONI, M. (2020). Full-stack development of a smartphone and web application of RespirHò: a wearable device for continuous respiratory and activity monitoring.

[3]. Mattila, T. (2018). Building a complete full-stack software development environment. Petersson, K. (2020). Test automation in a CI/CD workflow.

[4]. Pratama, M. R., & Kusumo, D. S. (2021, August). Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing. In 2021 9th International Conference on Information and Communication Technology (ICoICT) (pp. 230-235). IEEE.

[5]. 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.

[6]. 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.

[7]. Singh, C., Gaba, N. S., Kaur, M., & Kaur, B. (2019, January). Comparison of different CI/CD tools integrated with cloud platform. In 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence) (pp. 7-12). IEEE.

[8]. Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., & Di Penta, M. (2017, May). How open source projects use static code analysis tools in continuous integration pipelines. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (pp. 334-344). IEEE.