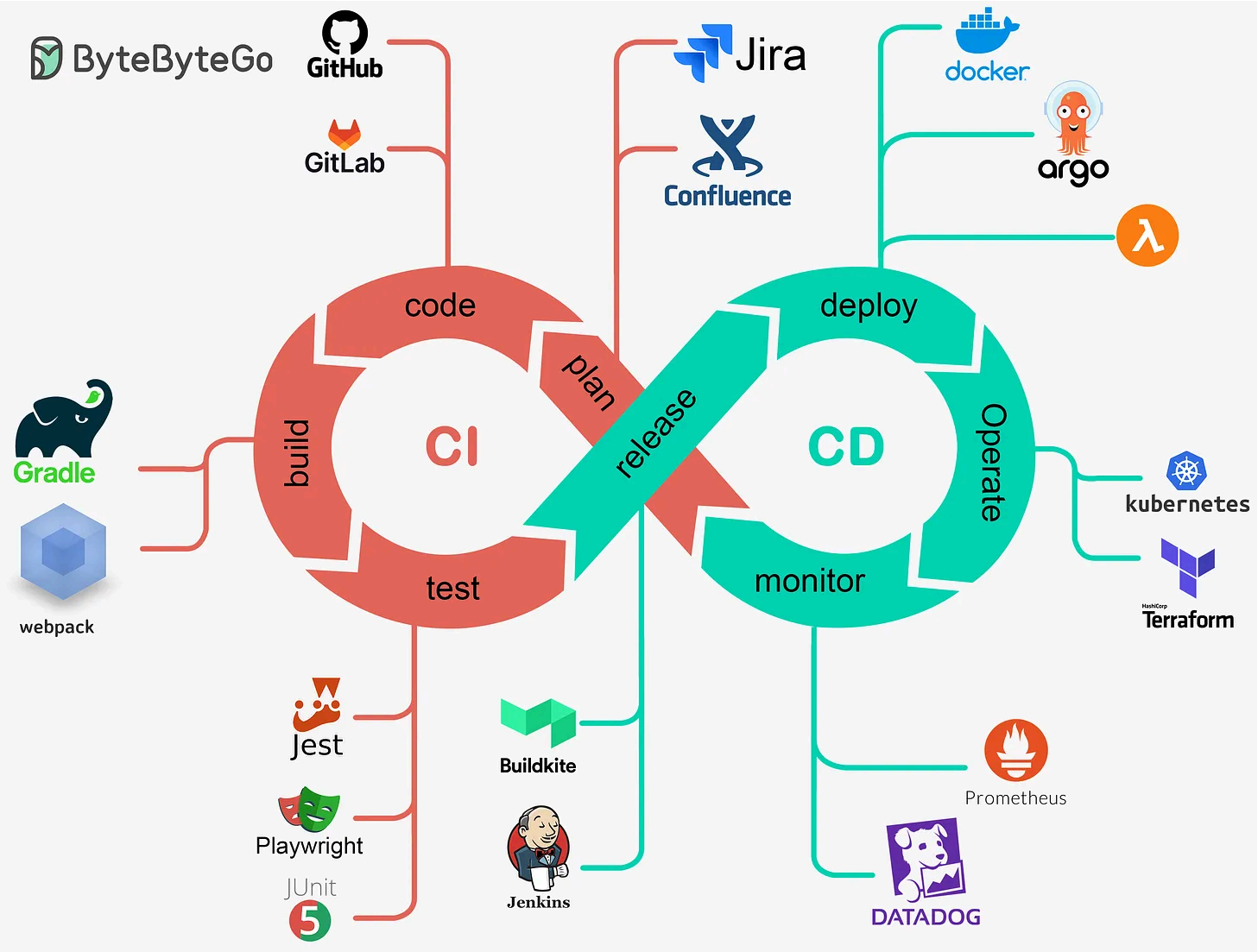
A Crash Course in CI/CD

BYTEBYTEGO

APR 04, 2024 · PAID

Introduction

What is CI/CD? How does it help us ship faster? Is it worth the hassle? In this issue, we will look into Continuous Integration and Continuous Deployment, or CI/CD for short. CI/CD helps automate the software development process from the initial code commit to deployment. It eliminates much of the manual human intervention traditionally required to ship code to production.



The CI/CD process builds, tests, and deploys code to production. The promise is that it enables software teams to deploy better-quality software faster. This all sounds very good, but does it work in real life? The answer is — it depends.

Let’s break up CI/CD into their parts and discuss them separately.

Continuous Integration (CI)

Continuous integration (CI) is a development practice that many people believe they're using in their work, but they don't fully get it.

Before CI came along, development teams typically operated in silos, with individual developers working independently on distinct features for extended periods. Their work would eventually need to be merged into a shared codebase, often resulting in complications such as merge conflicts and compatibility issues among the hundreds of files and contributors. This dilemma, often known as "**merge hell**," represented the difficulties faced in traditional development methods.

Avoid “merge hell”

Let's consider a scenario with two developers, Alice and Bob. Alice writes her code and shares it as soon as she has a functional version that doesn't cause any issues, even if it's not fully complete. She uploads her code to a central repository. Bob follows the same approach, always grabbing the latest version of the code before starting his work. As Alice continues to update her code, Bob does the same. If Bob makes changes, Alice incorporates them into her work without any problems. They collaborate smoothly, with a low chance of interfering each other because they always work off the latest code. If they encounter conflicts, it's usually on recent changes they've both made, so they can sit down together, resolve the issues, and move forward.

However, with so many people constantly contributing code, problems are inevitable. Things may not always run smoothly, and new errors can emerge. So, what's the solution?

Automation

The solution is automation. It acts like a vigilant watchdog, constantly monitoring the code. Whenever a change occurs, it springs into action, grabbing the code, building it, and running tests. If anything fails during this process, the team receives an alert, ensuring everyone is aware of the issue. With this safety net in place, continuous integration becomes a reality.

So, what exactly is continuous integration (CI)?

Definition

Continuous integration involves automating builds, executing tests, and merging code from individual developers into a shared repository. The primary goal of continuous integration is to efficiently integrate source code into shared repositories. Once changes are committed to the version control system, automated builds and test cases are executed to ensure the functionality and validity of the code. These processes validate how the source code compiles and how test cases perform during execution.

Tools

What are some common tools used in CI? A robust source code management system is the foundation. **GitHub** is a popular example. It holds everything needed to build the software, including source code, test scripts, and scripts to build the software applications.

Many tools are available to manage the CI process itself. **GitHub Actions** and **Buildkite** are modern examples, while **Jenkins**, **CircleCI**, and **TravisCI** are also widely used. These tools manage the build and test tasks in CI.

Numerous test tools exist for writing and running tests. These tools are usually language and ecosystem-specific. For example, in Javascript, **Jest** is a unit testing framework, while **Playwright** and **Cypress** are common integration testing frameworks for web applications.

Build tools are even more diverse and ecosystem-specific. **Gradle** is a powerful build tool for Java. The Javascript build ecosystem is fragmented and challenging to keep track of. **Webpack** is the standard, but many new build tools claim to be much faster, although they are not yet as extensible as Webpack.

Benefits of continuous integration

Continuous integration holds significant importance for several reasons. The table below presents some of the major advantages of CI.

Benefits of CI	
Avoid merge conflicts	CI mitigates merge conflicts by ensuring frequent synchronization of source code between individual developers' systems and the shared repository.
Reduce code review effort	It reduces the effort required for extensive code reviews. The frequent collaboration streamlines the process, diminishing the need for manual review interventions.
Speeds up development process	CI accelerates the development pace by facilitating efficient collaboration among developers. This collaborative environment fosters smoother integration of source code, enhancing the overall development process.
Readily testable build	In CI, we have a build that's up-to-date and ready to go. It is guaranteed to compile, and it contains the latest updates. So, we always have a version of the software that's ready for testing.
Reduces project backlog	CI contributes to reducing product backlogs. The frequent implementation of changes into the shared repository ensures that pending tasks are continually addressed and the backlog is reduced.

Continuous Deployment (CD)

Continuous deployment (CD) is the next step after CI in the CI/CD pipeline. CD is the practice of automatically deploying every code change that passes the automated testing phase to production.

While true continuous deployment is challenging and not as widely adopted as CI, a more common practice is continuous delivery, which is similar but has a subtle difference, as explained below.

Continuous delivery

Continuous delivery focuses on the rapid deployment of code changes into production environments. Its roots can be traced back to the Agile Manifesto, which emphasizes *"early and continuous delivery of valuable software"* to satisfy customers.

The objective of continuous delivery is to efficiently transition valuable code changes into production. The initial step involves transforming the code into deployable software through a build process. Once the software is ready, the next logical step might seem to be deploying it directly into production. However, the real practice involves rigorous testing to ensure that only stable software enters the production environment.

Typically, organizations maintain multiple test environments, such as "QA," "Performance," or "Staging." These environments serve as checkpoints for validating the software before it reaches production. The software undergoes testing in each environment to ensure its readiness for deployment.

In essence, the journey to production in continuous delivery involves transitioning software through various testing environments before deployment into the production environment.

A key aspect of continuous delivery is ensuring that the code remains deployable at all times. Once the delivery process is completed, the code is ready for deployment to any desired

environment. This end-to-end process includes building the source code, executing test cases, generating artifacts such as WAR or JAR files, and delivering them to specific environments.

Automatic deployment

Coming back to continuous deployment (CD), it involves the automatic deployment of code changes to the production environment. Essentially, CD represents the final stage in the development pipeline. In this phase, not only are artifacts prepared and test cases executed, but the process extends further to deploying the artifacts to the production environment. Continuous deployment ensures that any changes made to the code are promptly deployed to the production environment without human intervention.

Continuous deployment vs. continuous delivery

Continuous deployment and continuous delivery are related concepts, but they have distinct differences. Here, we list some of the differences:

Aspect	Continuous Delivery	Continuous Deployment
Definition	Code changes are automatically built, tested, and prepared for release to production.	Every change that passes automated tests is released to production automatically without human intervention.
Goal	Ensure code can be released to production at any time, but deployment may not be immediate upon passing tests.	Fully automate the deployment process, reducing the time from writing code to making it available to users in production.
Deployment to Production	Manual process triggered by humans; code sits in a staging environment awaiting approval.	Automated process; changes are deployed to production automatically without human intervention.
Automation and Testing	Emphasizes automation and rigorous testing to prepare code for deployment.	Requires a high degree of automation, rigorous testing, and monitoring to ensure automated deployment doesn't adversely affect the production environment.

While continuous deployment may be suitable for some organizations, continuous delivery is the approach that many are striving to achieve, as it offers a cautious yet automated approach to software delivery.

Tools

The tools we mentioned earlier, like GitHub Actions, Buildkite, and Jenkins, are commonly used to handle CD tasks. Infrastructure-specific tools also make CD easier to maintain. For example, **ArgoCD** is popular on Kubernetes.

CI/CD is a powerful software development practice that can help teams ship better-quality software faster. However, it's not a one-size-fits-all solution, and its implementation may vary depending on the complexity of the system.

Benefits of continuous deployment

Continuous deployment offers numerous benefits to organizations. Here, we list some of them.

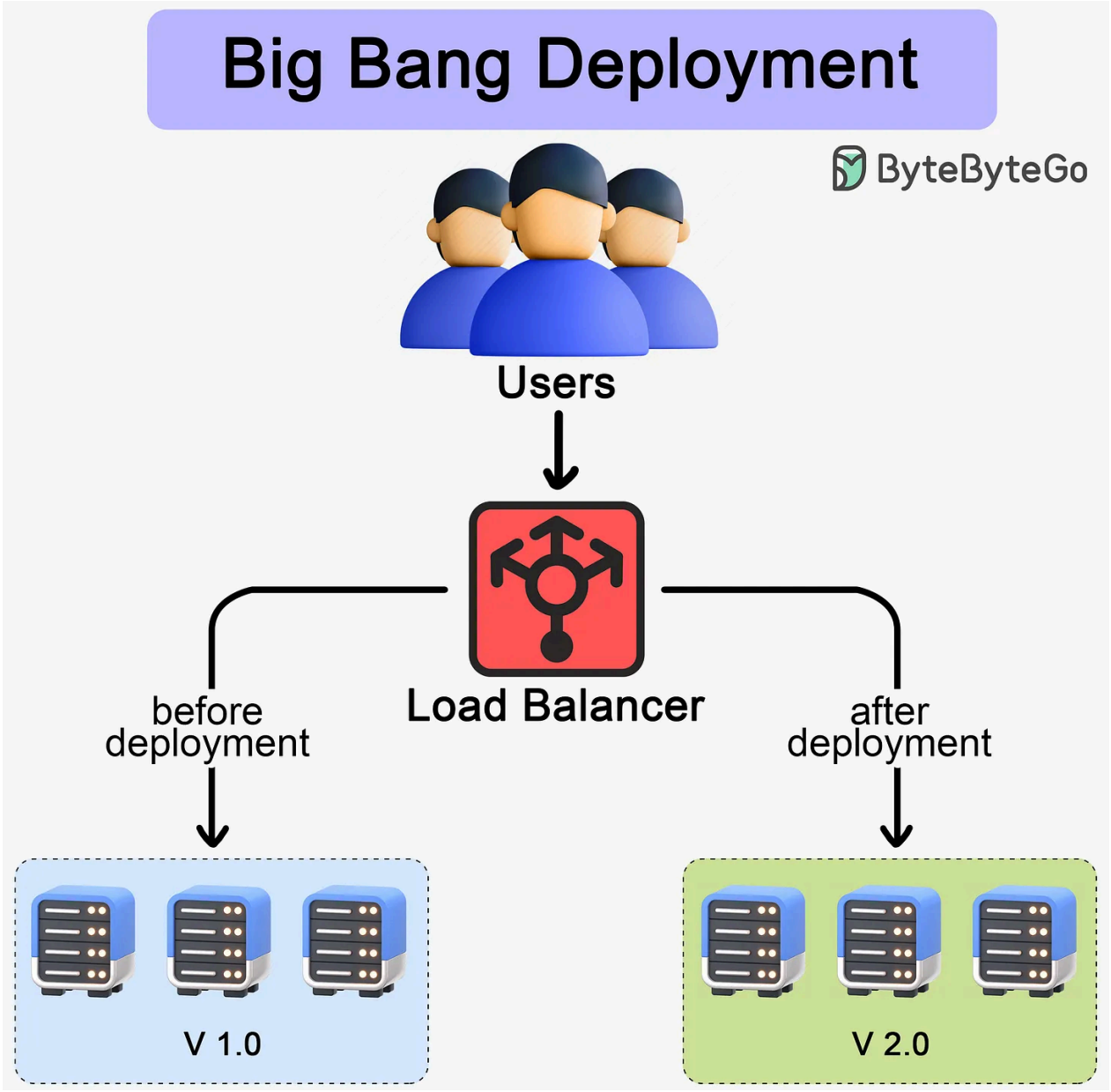
Benefits of CD	
Faster development	It facilitates faster product development and delivery by streamlining the automation process. This results in quicker product development cycles, a significant advantage of CD.
Less risky releases and easier problem resolution	CD leads to less risky releases, as automation reduces the likelihood of manual errors during deployment. This aspect contributes to a smoother deployment process with fewer chances of errors. Additionally, if any issues arise, they can be identified and fixed more quickly.
Continuous quality improvement	CD supports continuous improvement in product quality. By automating the deployment process, organizations can consistently enhance the quality of their products over time through frequent incremental improvements.

Deployment Strategies

Nothing beats the satisfaction of seeing our code go live to millions of users. It is always thrilling to see. But getting there is not always easy. Let’s explore some common deployment strategies.

Big bang deployment

One of the earliest methods of deploying changes to production is the **Big Bang Deployment**. Picture it like ripping off a bandage. We push all our changes at once, causing a bit of downtime as we have to shut down the old system to switch on the new one. The downtime is usually short, but be careful - it can sting if things don't go as planned. Preparation and testing are key. If things go wrong, we roll back to the previous version. However, rolling back is not always pain-free. We might still disrupt users, and there could be data implications. We need to have a solid rollback plan.

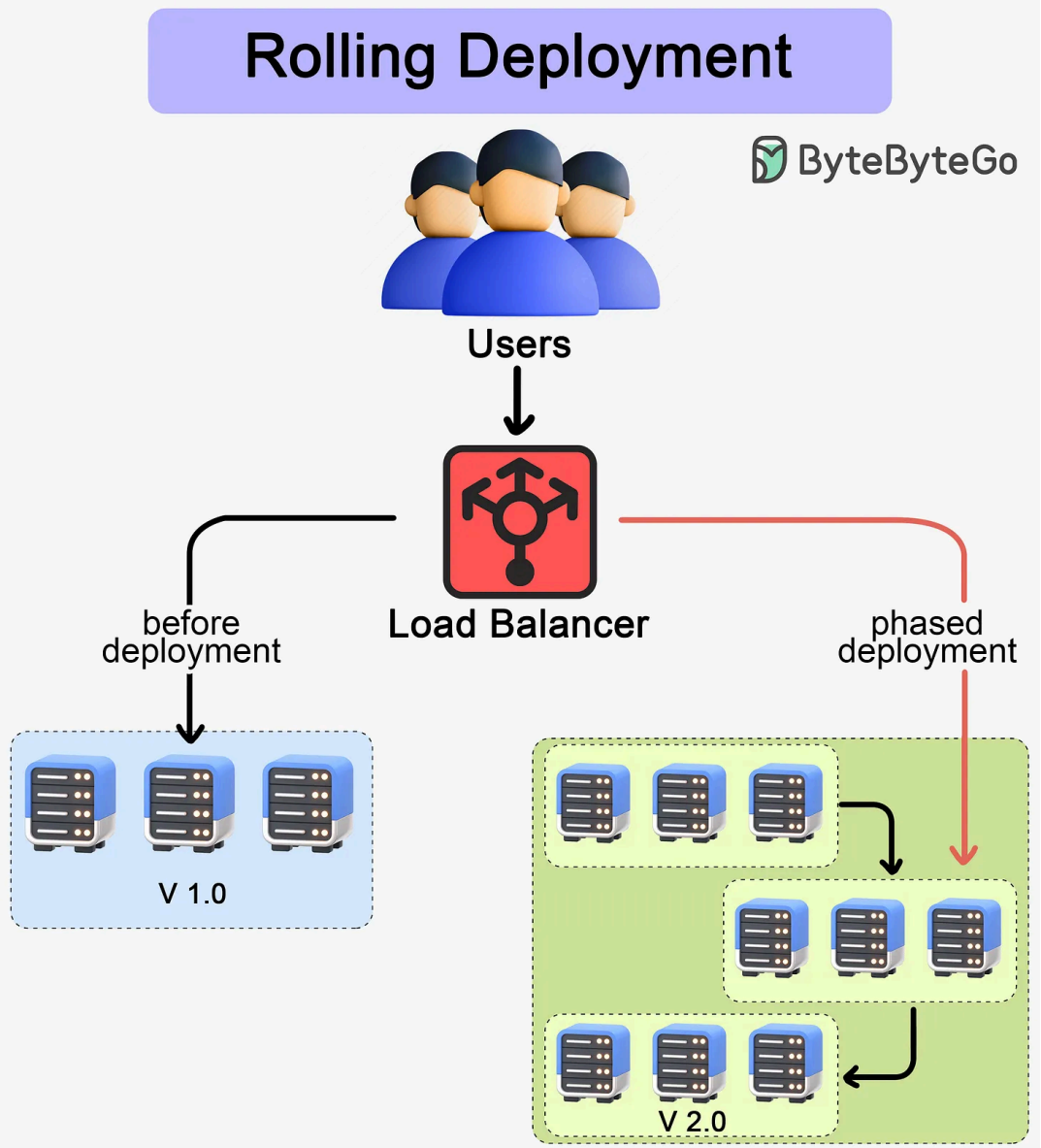

Big Bang Deployment
ByteByteGo
Users
Load Balancer
before deployment
after deployment
V 1.0
V 2.0

Big bang deployment is sometimes the only choice, for example, when an intricate database upgrade is involved.

Rolling deployment

This method lets us incrementally update different parts of the system over time. Rolling deployment is more like a marathon than a sprint. It's a staged rollout where we gradually deploy the new version of the application to the production environment.

Here's an example of how it might work. Imagine we have 10 servers running our application. In a rolling deployment, we might take down the first server, deploy the new version of our application there, and bring it back online. Once we've confirmed everything is working as expected, we'll move on to the second server, and so on. This approach allows the new version to gradually replace the old one, server by server, until the entire system is updated.



One big advantage of rolling deployment is that it usually prevents downtime. While we're updating one server, the others are still up and running, serving our users. Another advantage is

that we can spot and mitigate any issues early during the rollout, reducing the risk of widespread problems. We're only ever exposing a small part of our system to the new version at any one time.

However, rolling deployment is typically a slower process. While it reduces the risk of system-wide issues, it doesn't entirely eliminate it. If an issue slips past our initial checks, it might still propagate as we update more servers.

Additionally, this strategy doesn't support targeted rollouts. We can't control which users get the new version during the rollout. All users will gradually see the new version as we update the servers, and we can't direct the new version to specific users based on criteria like location, device type, etc.

Rolling deployment is a popular choice for many teams. It balances risk and user impact in a controlled, methodical way.

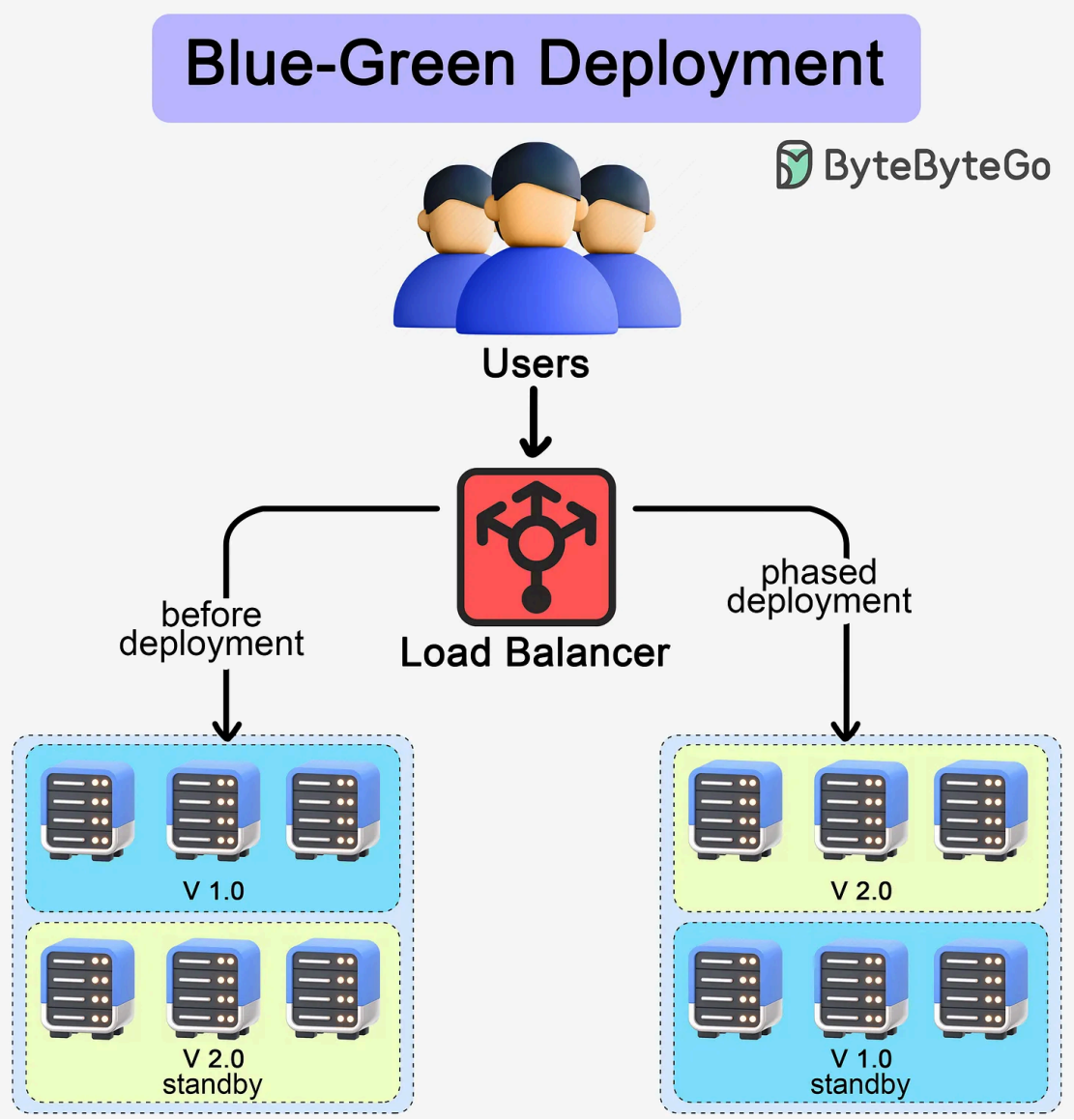
Blue-Green deployment

Now let's talk about the Blue-Green deployment. Here, we maintain two identical production environments, cleverly named blue and green. At any given time, one side is active and visible to users, and the other is idle. The active environment (say, blue) serves the current live version of the application to the users. The idle one (green) is our playground where we can safely deploy and test the new version.

When we have a new version ready to go, we deploy it to the green environment. While this is happening, the blue environment is still live and serving the current version of the application to users. Our QA team then tests the new version in the green environment. This gives us the

chance to catch and fix any bugs or issues before they reach our users. Once the new version in the green environment is deemed ready, we simply switch the load balancer to redirect traffic from the blue environment to the green one. Users are seamlessly transitioned to the new version of the application with zero downtime.

Now the blue environment becomes idle and serves as our safety net. If we encounter any issues with the new version, we can quickly switch back to it, effectively rolling back to the previous version.



While blue-green deployment allows for seamless transitions and easy rollbacks, there's a catch. Just like with the rolling deployment, we can't direct the new version to specific users. The switch from blue to green happens for all users at once. It is also resource-intensive. Maintaining

two identical production environments doubles the infrastructure and resource needs. We could spin down the idle environment between deployments, but this introduces complexity. Managing two parallel production environments and ensuring seamless data synchronization can add significant complexity to the deployment process. It requires sophisticated infrastructure management and tooling.

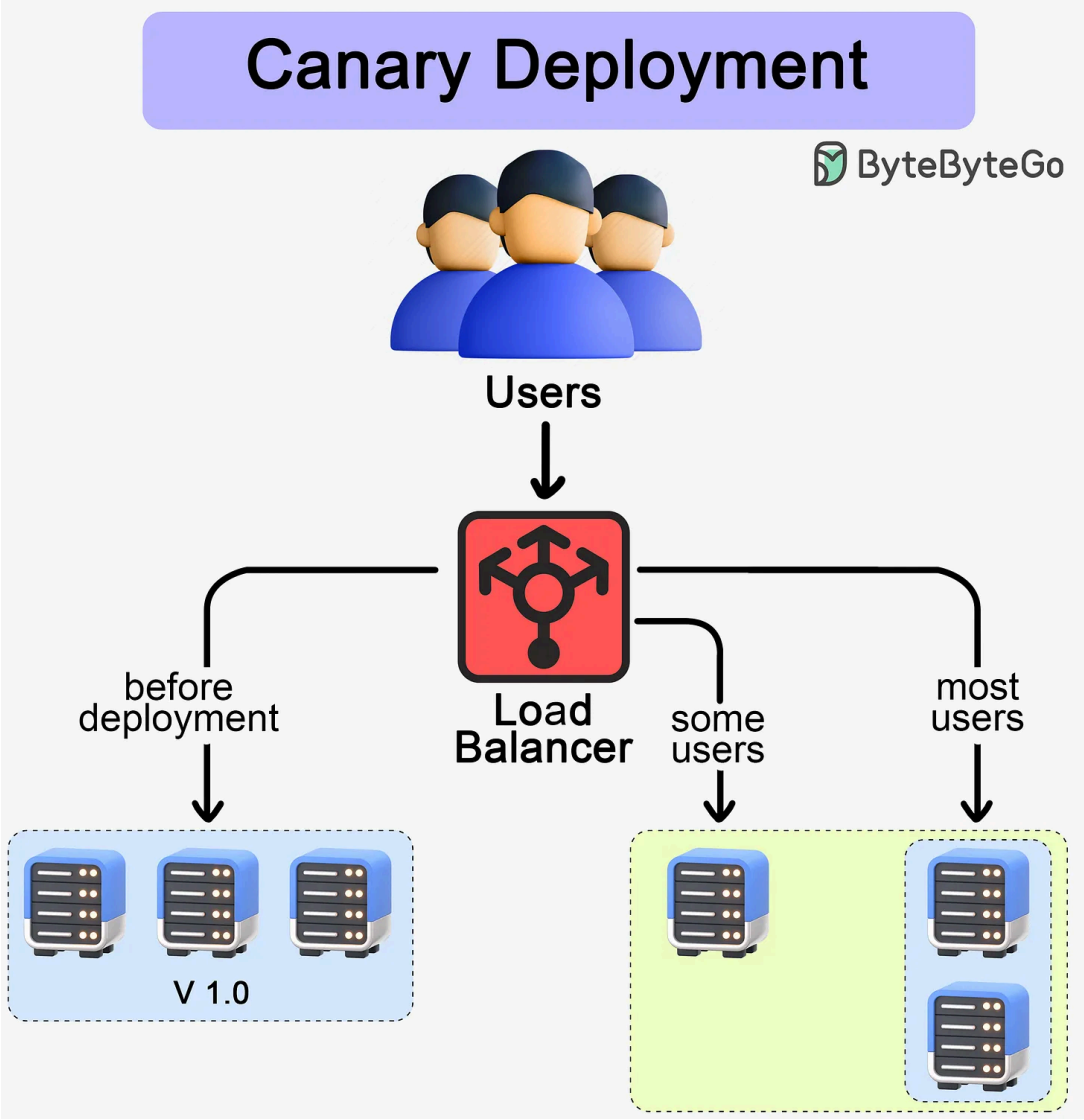
However, with its high level of control and minimized risk, blue-green deployment remains a popular strategy for smooth user experience and reliable rollbacks.

Canary deployment

Canary deployment, named after the age-old practice of using canaries in coal mines to detect dangerous gasses, is a strategy used to ‘test the air’ before a full-scale rollout. If the canary was in distress, miners knew it was time to evacuate. Similarly, in software deployment, instead of deploying the new version to all servers or users, we choose a small subset, our 'canaries.' It could be a percentage of servers or a group of users, often selected based on certain criteria.

For example, we might start by deploying to a single server, a small cluster, or even a certain geographical location. This allows us to monitor the performance of the new version under real-world conditions, but on a much smaller scale. If everything goes well and our new version performs as expected, we can gradually roll it out to the rest of the servers or users. But if something goes wrong, we've got a safety net. We can halt the deployment, fix the issues, and try again, all without impacting the majority of our user base. This incremental approach offers us both safety and control. Canary deployment also gives us the power of targeted rollouts, allowing us to direct our canaries based on user-specific criteria, like geographical location or device type.

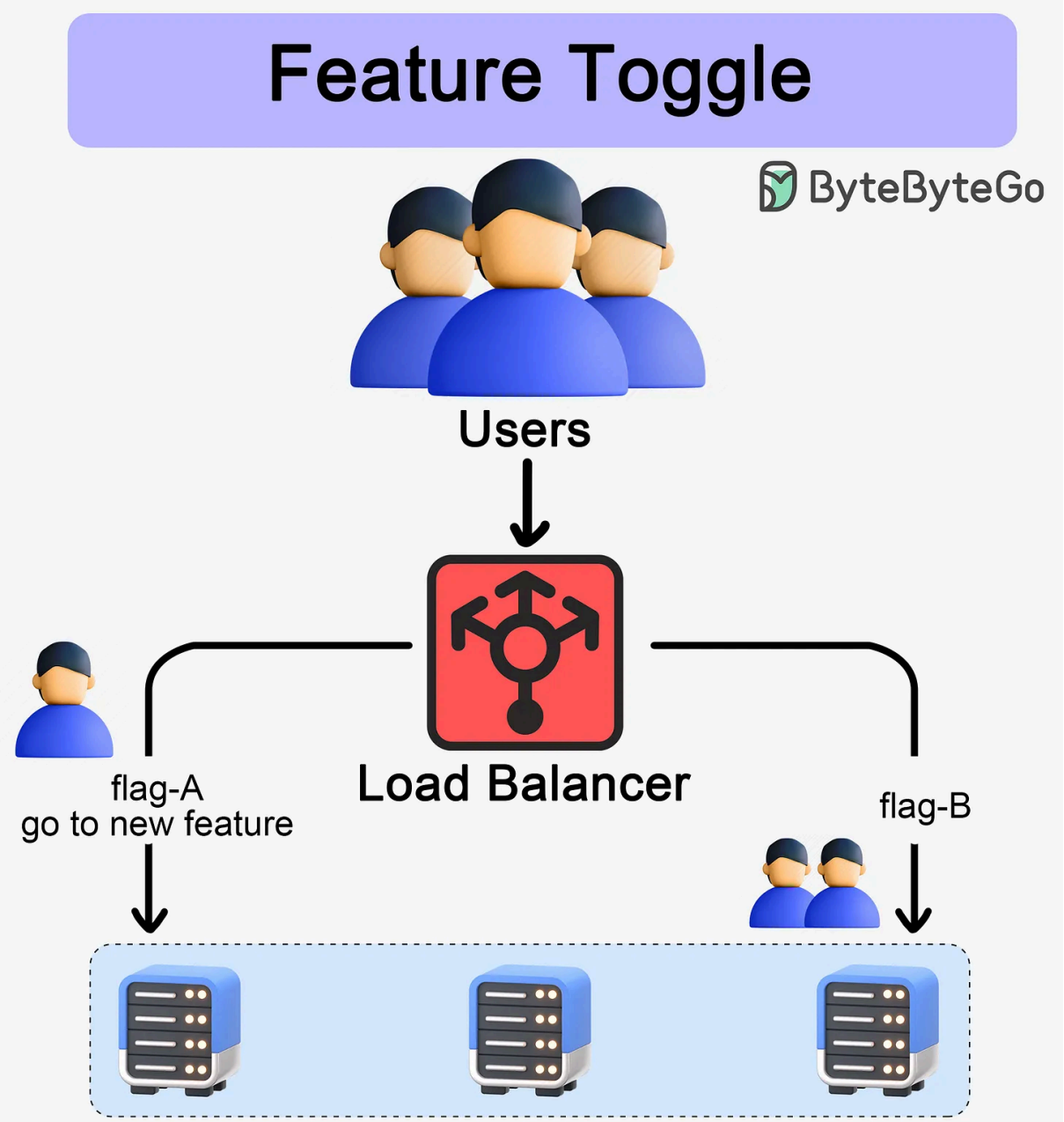
However, canary deployment does come with its own set of challenges. It requires careful monitoring and automated testing for the canaries. It requires somewhat complicated infrastructure tooling to ramp up or halt the deployment as needed. The strategy can be complex to implement and manage, especially when dealing with database schema changes or API compatibility issues. Canary deployment is usually not a standalone strategy. It's often combined with rolling deployment to create an approach that brings together the best of both worlds.



Feature toggle

Feature toggle is a strategy that stands apart from the others we've discussed. It's not about deploying a new version of the entire application, but rather about managing specific new

features within that application. With feature toggle, we introduce a 'toggle' or switch in the code for new features. This allows us to turn the feature on or off for certain users or circumstances. Think of it as a gate that we can open or close. It controls who gets to see the new feature.


Feature Toggle
ByteByteGo
Users
Load Balancer
flag-A
go to new feature
flag-B

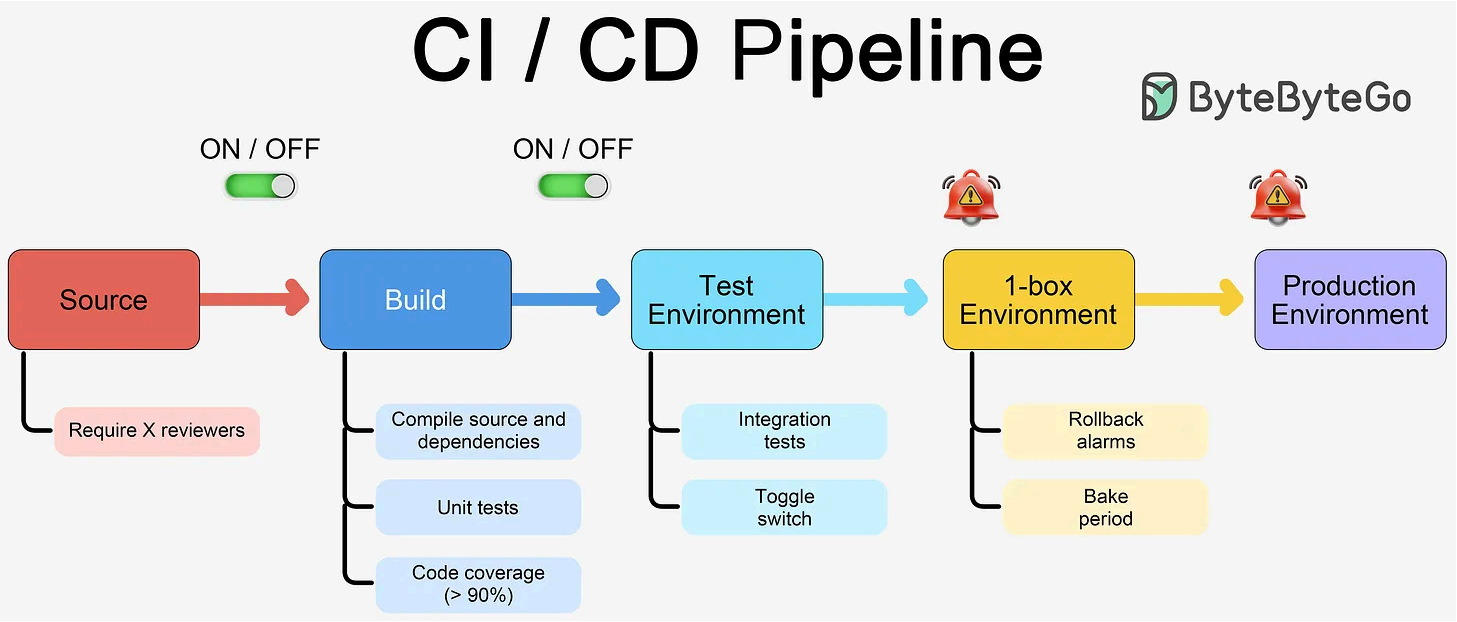
Feature toggle can be used in combination with any of the deployment strategies we've discussed. For example, if we're doing a canary deployment, we can turn on the feature toggle for just the canary users, letting them test out the new feature while the rest of the user base carries on with the current version. Feature toggle offers excellent control over new features and allows for targeted user testing. It's great for A/B testing or gradually rolling out a feature to see how it performs.

However, feature toggle has its downsides. If not managed properly, toggles can add complexity to the codebase and make testing more difficult. Old or obsolete toggles need to be cleaned up to prevent 'toggle debt,' which can make the system increasingly hard to maintain.

A Typical CI/CD Pipeline

We will outline the blueprint of a typical CI/CD pipeline. The motivation behind this is the observation that many people face challenges in configuring their software application's environment.

We won't dive into specific technologies or cloud concepts. Instead, we will focus on general principles and best practices that can be adapted to any technology stack. Whether you opt for cloud platforms like AWS or prefer custom tools, the concepts outlined here are universally applicable.



Let's jump straight into it.

Source

The first and rather apparent step in a typical CI/CD pipeline is source code management. This is where we'll store all our packages and dependencies for our application. One crucial aspect at the source control level is the need for a mechanism mandating a certain level of review for project changes. It's imperative that we can't merge code into our source control randomly; instead, we should require approval from designated reviewers via pull requests. While this can be set up using various technologies, it's definitely a necessary step. At a minimum, one reviewer should be required, but ideally, having at least two reviewers would be safer.

Build

Moving on to the next phase, the build process happens after we commit changes to our source code. During this phase, the code goes through compilation, including all dependencies specified in our package.json, pom file, or any other language-specific configuration files. Additionally, we'll need to run all unit tests associated with our project.

Extensive unit test coverage is essential for a reliable CI/CD pipeline. Testing helps catch issues early and ensures the application works as expected.

Code coverage

Many testing frameworks, such as **TestNG** or **JUnit**, calculate code coverage for specific commits. While there's no universal target, it's generally a good practice to maintain high code coverage. The goal should be to steadily increase this percentage over time as the codebase evolves. However, achieving 100% coverage is often impractical and not necessarily required for a robust CI/CD pipeline.

Test environment

The next stage is the testing environment. A key activity in the testing environment is running integration tests. Integration tests validate the behavior of our APIs, ensuring they function correctly and adhere to expected business rules. These tests encompass scenarios such as API interactions and the fulfillment of business logic requirements.

For example, with an API like "create order," we should have an integration test that calls this API with valid input data and verifies that a new order is created successfully. After creating the order, the test should retrieve the order details and confirm that all order information matches the initial input. If the retrieved data doesn't match what was sent, the test has failed.

Integration tests also validate that the application adheres to defined business rules. For instance, if there's a rule stating "Orders cannot exceed $10,000 in value," we need a test specifically for this scenario. The test would attempt to create an order with a value greater than $10,000 and ensure that the application prevents or rejects such an order, in compliance with the business rule.

So, integration tests validate the core functionality of our APIs to see if they're doing what is expected.

Toggle switch

We should be able to enable or disable these stages at any given point. This capability allows us to control the promotion of changes to the next phase of our deployment pipeline, ensuring flexibility and control over the process.

Consider dependencies

Furthermore, when running integration tests, it's essential to consider dependencies. Suppose our APIs rely on external services, such as customer profile management or address storage. In that case, we should also test these dependencies to ensure our system's end-to-end functionality. This comprehensive testing approach instills confidence in core APIs and the underlying logic of our application, ensuring they perform as expected.

Overall, the testing environment serves as our sandbox, allowing us to experiment and validate changes thoroughly before proceeding further in the deployment pipeline.

1-box environment

The 1-box deployment is a way to test changes with a small portion of production traffic first. The idea is to mitigate the risk of deploying faulty changes to the entire production environment where all customer traffic goes. Here's how it works: if there are 10 hosts in production, one host is designated as the "1-box" host. This 1-box host handles around 10% or less of the total production traffic. This setup provides a safety net. If any issues arise, like increased errors or slowness, they only impact the 1-box host. Rolling back a problematic change is much simpler and quicker on just one host, compared to rolling back across all production.

Rollback alarms

When deploying from testing to production, several issues could pop up like increased error rate, slower responses, or deviations in important metrics. To catch these issues, we need rollback alarms. Rollback alarms continuously monitor things like error rates, latency, and key business metrics. If any of these go beyond set limits, indicating a potential problem, the alarms automatically trigger a rollback to the previous stable version. This automated rollback enhances the CI/CD pipeline's resilience by ensuring a quick revert if something goes wrong.

Bake period

Assuming no rollback alarms trigger during deployment, the next step is the "bake period." The bake period allows the changes to stabilize in the production environment before full deployment. It acts as a buffer to catch any delayed issues or abnormal behavior that may not show up right away. Once the bake period passes smoothly, you can have more confidence in the changes before progressing further.

Production environment

If everything goes smoothly up to this point, we're ready for the full production environment. The same safeguards like rollback alarms are also implemented here. After completing these steps, we can proceed confidently to production, assured that everything is working correctly.

Wrap Up

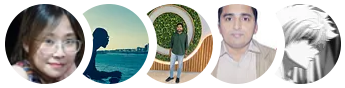
In this issue, we've explored how to make software development smoother and faster with the help of Continuous Integration and Continuous Deployment (CI/CD).

We started by understanding CI/CD, which helps automate the process from writing code to releasing it to users. We focused on Continuous Integration (CI), emphasizing its significance in mitigating merge conflicts and fostering collaboration among developers. Through automation, CI ensures that code integrates smoothly into shared repositories, improving efficiency and code quality. Then, we looked at Continuous Deployment (CD), which helps put code into production quickly and safely.

We learned about different deployment strategies, such as Big Bang, Rolling, Blue-Green, Canary, and Feature Toggle, and explained their benefits and challenges in managing deployment complexities while ensuring minimal user disruption.

Finally, we outlined a step-by-step plan for setting up a smooth CI/CD process. This includes reviewing code before it is added, testing it thoroughly, and monitoring it once it's live.

Overall, CI/CD helps teams work better together, release new features faster, and make sure everything runs smoothly for users. It's all about making software development easier and more reliable.



523 Likes · 35 Restacks

4 Comments

Write a comment...

BN Sarma Aug 10

Excellent article, very good coverage of CI/CD. Qudos to the authors.!

DevOps/DevSecOps Framework and how it is coupled (subset) to CI/CD would be an additional topic to cover someday?

I have seen a couple of use cases where CI/CD(DevOps) got abused in a distributed database setup using Multi Master Replication.

Developers over a period of time lose focus on the backend Database and infrastructure design/setup.

Odd/Even Sequences (Surrogate Key- A Crash Course on Relational Database Design) for Primary Key is a standard practice for conflict avoidance. Developers not aware of this setup and pushing Database Code Changes through CI/CD Pipeline can lead to data integrity issues.

LIKE REPLY SHARE

Hoang Tran May 10

Super detailed on CI/CD! I have clearer sense of how CI/CD all works together and different deployment strategies which were confusing to me.

LIKE REPLY SHARE

2 more comments...