Practicing Continuous Integration and Continuous Delivery on AWS

Accelerating Software Delivery with DevOps

First Published June 1, 2017

Updated October 27, 2021

aws

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

This version has been archived.

For the latest version of this document, visit:

https://docs.aws.amazon.com/whitepapers/latest/ practicing-continuous-integration-continuous- delivery/welcome.html

Contents

Abstract

This paper explains the features and benefits of using continuous integration and continuous delivery (CI/CD) along with Amazon Web Services (AWS) tooling in your software development environment. Continuous integration and continuous delivery are best practices and a vital part of a DevOps initiative.

This version has been archived.

For the latest version of this document, visit:

**https://docs.aws.amazon.com/whitepapers/latest/
practicing-continuous-integration-continuous-
delivery/welcome.html**

The challenge of software delivery

Enterprises today face the challenges of rapidly changing competitive landscapes, evolving security requirements, and performance scalability. Enterprises must bridge the gap between operations stability and rapid feature development. Continuous integration and continuous delivery (CI/CD) are practices that enable rapid software changes while maintaining system stability and security.

Amazon realized early on that the business needs of delivering features for Amazon.com retail customers, Amazon subsidiaries and Amazon Web Services (AWS) would require new and innovative ways of delivering software. At the scale of a company like Amazon, thousands of independent software teams must be able to work in parallel to deliver software quickly, securely, reliably, and with zero tolerance for outages.

By learning how to deliver software at high velocity, Amazon and other forward-thinking organizations pioneered DevOps. DevOps is a combination of cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity. Using DevOps principles, organizations can evolve and improve products at a faster pace than organizations that use traditional software development and infrastructure management processes. This speed enables organizations to better serve their customers and compete more effectively in the market.

Some of these principles, such as two-pizza teams and microservices/service-oriented architecture (SOA), are out of the scope of this whitepaper. This whitepaper discusses the CI/CD capability that Amazon has built and continuously improved. CI/CD is key to delivering software features rapidly and reliably.

AWS now offers these CI/CD capabilities as a set of developer services: AWS CodeStar, AWS CodeCommit, AWS CodePipeline, AWS CodeBuild, AWS CodeDeploy, and AWS CodeArtifact. Developers and IT operations professionals practicing DevOps can use these services to rapidly, safely, and securely deliver software. Together they help you securely store and apply version control to your application's source code. You can use AWS CodePipeline to orchestrate an end-to-end software release workflow using these services. For an example of implementation, CodePipeline has the flexibility to integrate each service independently with your existing tools. These are highly available, easily integrated services that can be accessed through the AWS Management Console, AWS application programming interfaces (APIs), and AWS software development toolkits (SDKs) like any other AWS service.

What is continuous integration and continuous delivery/deployment?

This section discusses the practices of continuous integration and continuous delivery and explains the difference between continuous delivery and continuous deployment.

Continuous integration

Continuous integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. CI most often refers to the build or integration stage of the software release process and requires both an automation component (for example a CI or build service) and a cultural component (for example learning to integrate frequently). The key goals of CI are to find and address bugs more quickly, improve software quality, and reduce the time it takes to validate and release new software updates.

Continuous integration focuses on smaller commits and smaller code changes to integrate. A developer commits code at regular intervals, at minimum once a day. The developer pulls code from the code repository to ensure the code on the local host is merged before pushing to the build server. At this stage the build server runs the various tests and either accepts or rejects the code commit.

The basic challenges of implementing CI include more frequent commits to the common codebase, maintaining a single source code repository, automating builds, and automating testing. Additional challenges include testing in similar environments to production, providing visibility of the process to the team, and allowing developers to easily obtain any version of the application.

Continuous delivery and deployment

Continuous delivery (CD) is a software development practice where code changes are automatically built, tested, and prepared for production release. It expands on continuous integration by deploying all code changes to a testing environment, a production environment, or both after the build stage has been completed. Continuous delivery can be fully automated with a workflow process or partially automated with manual steps at critical points. When continuous delivery is properly implemented, developers always have a deployment-ready build artifact that has passed through a standardized test process.

With continuous deployment, revisions are deployed to a production environment automatically without explicit approval from a developer, making the entire software release process automated. This, in turn, allows for a continuous customer feedback loop early in the product lifecycle.

Continuous delivery is not continuous deployment

One misconception about continuous delivery is that it means every change committed is applied to production immediately after passing automated tests. However, the point of continuous delivery is not to apply every change to production immediately, but to ensure that every change is ready to go to production.

Before deploying a change to production, you can implement a decision process to ensure that the production deployment is authorized and audited. This decision can be made by a person and then executed by the tooling.

Using continuous delivery, the decision to go live becomes a business decision, not a technical one. The technical validation happens on every commit.

Rolling out a change to production is not a disruptive event. Deployment doesn't require the technical team to stop working on the next set of changes, and it doesn't need a project plan, handover documentation, or a maintenance window. Deployment becomes a repeatable process that has been carried out and proven multiple times in testing environments.

Benefits of continuous delivery

CD provides numerous benefits for your software development team including automating the process, improving developer productivity, improving code quality, and delivering updates to your customers faster.

Automate the software release process

CD provides a method for your team to check in code that is automatically built, tested, and prepared for release to production so that your software delivery is efficient, resilient, rapid, and secure.

Improve developer productivity

CD practices help your team's productivity by freeing developers from manual tasks, untangling complex dependencies, and returning focus to delivering new features in

software. Instead of integrating their code with other parts of the business and spending cycles on how to deploy this code to a platform, developers can focus on coding logic that delivers the features you need.

Improve code quality

CD can help you discover and address bugs early in the delivery process before they grow into larger problems later. Your team can easily perform additional types of code tests because the entire process has been automated. With the discipline of more testing more frequently, teams can iterate faster with immediate feedback on the impact of changes. This enables teams to drive quality code with a high assurance of stability and security. Developers will know through immediate feedback whether the new code works and whether any breaking changes or bugs were introduced. Mistakes caught early on in the development process are the easiest to fix.

Deliver updates faster

CD helps your team deliver updates to customers quickly and frequently. When CI/CD is implemented, the velocity of the entire team, including the release of features and bug fixes, is increased. Enterprises can respond faster to market changes, security challenges, customer needs, and cost pressures. For example, if a new security feature is required, your team can implement CI/CD with automated testing to introduce the fix quickly and reliably to production systems with high confidence. What used to take weeks and months can now be done in days or even hours.

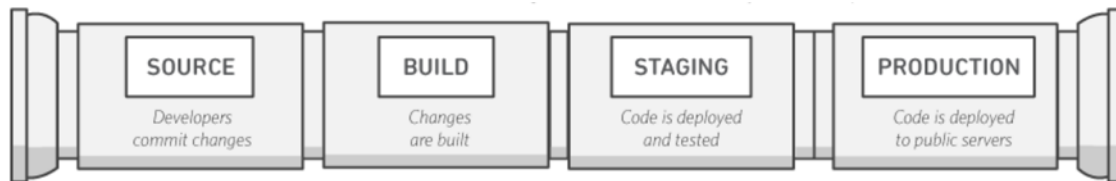# Implementing continuous integration and continuous delivery

This section discusses the ways in which you can begin to implement a CI/CD model in your organization. This whitepaper doesn't discuss how an organization with a mature DevOps and cloud transformation model builds and uses a CI/CD pipeline. To help you on your DevOps journey, AWS has a number of certified DevOps Partners who can provide resources and tools. For more information on migrating to the AWS Cloud, refer to the AWS Building a Cloud Operation Model.

A pathway to continuous integration/continuous delivery

CI/CD can be pictured as a pipeline (refer to the following figure), where new code is submitted on one end, tested over a series of stages (source, build, staging, and production), and then published as production-ready code. If your organization is new to CI/CD it can approach this pipeline in an iterative fashion. This means that you should start small, and iterate at each stage so that you can understand and develop your code in a way that will help your organization grow.

| SOURCE	BUILD	STAGING	PRODUCTION
Developers commit changes	Changes are built	Code is deployed and tested	Code is deployed to public servers

CI/CD pipeline

Each stage of the CI/CD pipeline is structured as a logical unit in the delivery process. In addition, each stage acts as a gate that vets a certain aspect of the code. As the code progresses through the pipeline, the assumption is that the quality of the code is higher in the later stages because more aspects of it continue to be verified. Problems uncovered in an early stage stop the code from progressing through the pipeline. Results from the tests are immediately sent to the team, and all further builds and releases are stopped if software does not pass the stage.

These stages are suggestions. You can adapt the stages based on your business need. Some stages can be repeated for multiple types of testing, security, and performance. Depending on the complexity of your project and the structure of your teams, some stages can be repeated several times at different levels. For example, the end product of one team can become a dependency in the artifact of the next team. This means that the first team's end product is subsequently staged as an artifact in the next team's project.

The presence of a CI/CD pipeline will have a large impact on maturing the capabilities of your organization. The organization should start with small steps and not try to build a fully mature pipeline, with multiple environments, many testing phases, and automation in all stages at the start. Keep in mind that even organizations that have highly mature CI/CD environments still need to continuously improve their pipelines.

Building a CI/CD-enabled organization is a journey, and there are many destinations along the way. The next section discusses a possible pathway that your organization could take, starting with continuous integration through the levels of continuous delivery.

## Continuous integration
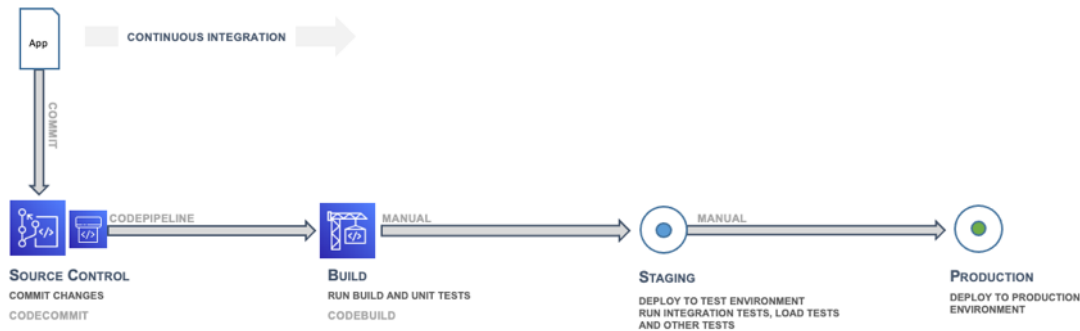
Continuous integration—source and build

The first phase in the CI/CD journey is to develop maturity in continuous integration. You should make sure that all of the developers regularly commit their code to a central repository (such as one hosted in CodeCommit or GitHub) and merge all changes to a release branch for the application. No developer should be holding code in isolation. If a feature branch is needed for a certain period of time, it should be kept up to date by merging from upstream as often as possible. Frequent commits and merges with complete units of work are recommended for the team to develop discipline and are encouraged by the process. A developer who merges code early and often, will likely have fewer integration issues down the road.

You should also encourage developers to create unit tests as early as possible for their applications and to run these tests before pushing the code to the central repository. Errors caught early in the development process are the simplest and easiest to fix.

When the code is pushed to a branch in a source code repository, a workflow engine monitoring that branch will send a command to a builder tool to build the code and run the unit tests in a controlled environment. The build process should be sized appropriately to handle all activities, including pushes and tests that might happen during the commit stage, for fast feedback. Other quality checks, such as unit test coverage, style check, and static analysis, can happen at this stage as well. Finally, the builder tool creates one or more binary builds and other artifacts, like images, stylesheets, and documents for the application.
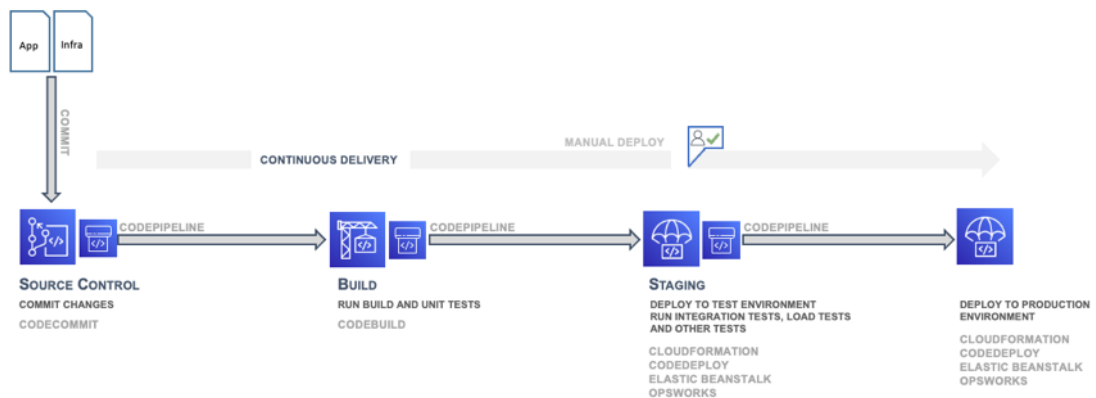
Continuous delivery: creating a staging environment
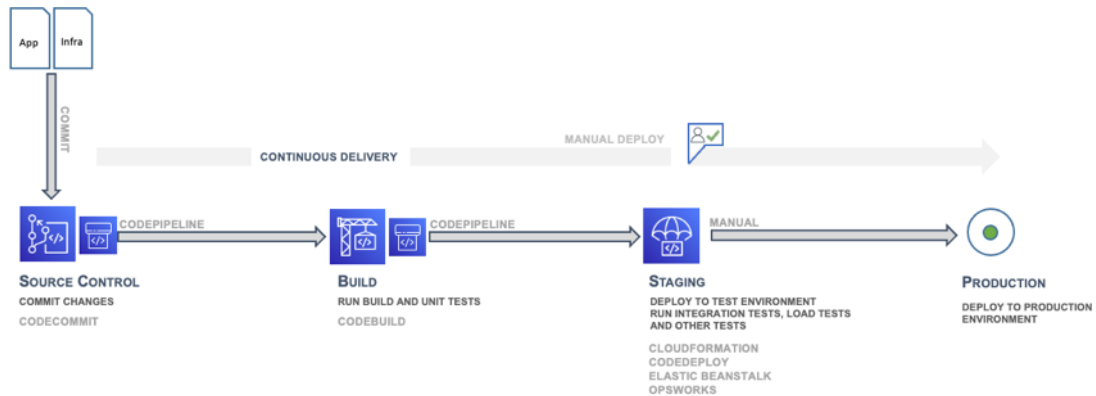
Continuous delivery—staging

Continuous delivery (CD) is the next phase and entails deploying the application code in a staging environment, which is a replica of the production stack, and running more functional tests. The staging environment could be a static environment premade for testing, or you could provision and configure a dynamic environment with committed infrastructure and configuration code for testing and deploying the application code.

## Continuous delivery: creating a production environment

Continuous delivery—production

https://docs.aws.amazon.com/whitepapers/latest/ practicing-continuous-integration-continuous- delivery/welcome.html

In the deployment/delivery pipeline sequence, after the staging environment, is the production environment, which is also built using infrastructure as code (IaC).

Continuous deployment
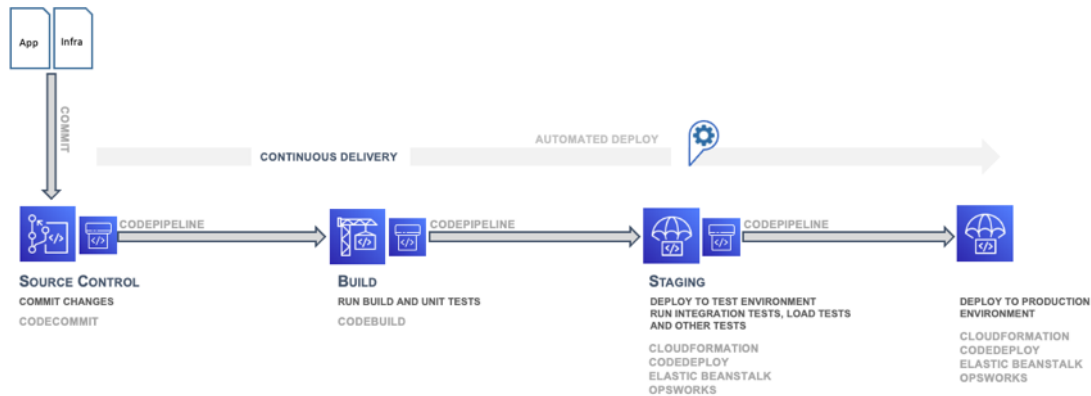
Continuous deployment

The final phase in the CI/CD deployment pipeline is continuous deployment, which may include full automation of the entire software release process including deployment to the production environment. In a fully mature CI/CD environment, the path to the production environment is fully automated, which allows code to be deployed with high confidence.

Maturity and beyond

As your organization matures, it will continue to develop the CI/CD model to include more of the following improvements:

- More staging environments for specific performance, compliance, security, and user interface (UI) tests
- Unit tests of infrastructure and configuration code along with the application code
- Integration with other systems and processes such as code review, issue tracking, and event notification
- Integration with database schema migration (if applicable)
- Additional steps for auditing and business approval

Even the most mature organizations that have integrated multi-environment CI/CD pipelines continue to look for improvements. DevOps is a journey, not a destination. Feedback about the pipeline is continuously collected and improvements in speed, scale, security, and reliability are achieved as a collaboration between the different parts of the development teams.

Teams

AWS recommends organizing three developer teams for implementing a CI/CD environment: an application team, an infrastructure team, and a tools team (refer to the following figure). This organization represents a set of best practices that have been developed and applied in fast-moving startups, large enterprise organizations, and in Amazon itself. The teams should be no larger than groups that two pizzas can feed, or about 10-12 people. This follows the communication rule that meaningful conversations hit limits as group sizes increase and lines of communication multiply.
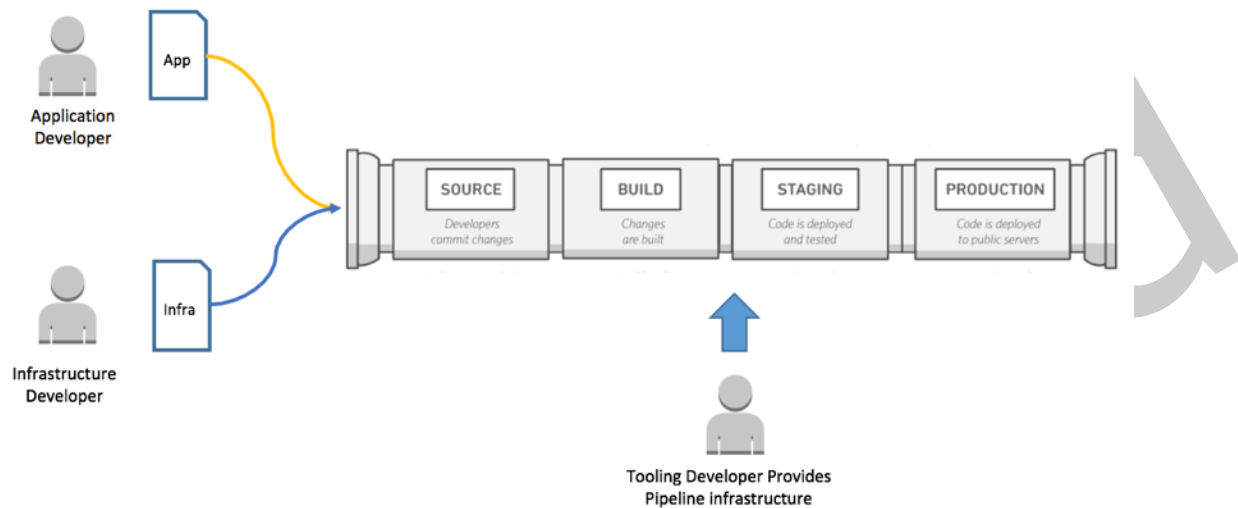
Application, infrastructure, and tools teams

Application team

This version has been archived.

The application team creates the application. Application developers own the backlog, stories, and unit tests, and they develop features based on a specified application target. This team's organizational overhead is minimized, and team members spend on non-core application tasks.

For the latest version of this document, visit:

In addition to having functional programming skills in the application language, the application team should have platform skills and an understanding of system configuration. This will enable them to focus solely on developing features and hardening the application.

Infrastructure team

The infrastructure team writes the code that both creates and configures the infrastructure needed to run the application. This team might use native AWS tools, such as AWS CloudFormation, or generic tools, such as Chef, Puppet, or Ansible. The infrastructure team is responsible for specifying what resources are needed, and it works closely with the application team. The infrastructure team might consist of only one or two people for a small application.

The team should have skills in infrastructure provisioning methods, such as AWS CloudFormation or HashiCorp Terraform. The team should also develop configuration automation skills with tools such as Chef, Ansible, Puppet, or Salt.

Tools team

The tools team builds and manages the CI/CD pipeline. They are responsible for the infrastructure and tools that make up the pipeline. They are not part of the two-pizza team; however, they create a tool that is used by the application and infrastructure teams in the organization. The organization needs to continuously mature its tools team, so that the tools team stays one step ahead of the maturing application and infrastructure teams.

The tools team must be skilled in building and integrating all parts of the CI/CD pipeline. This includes building source control repositories, workflow engines, build environments, testing frameworks, and artifact repositories. This team may choose to implement software such as AWS CodeStar, AWS CodePipeline, AWS CodeCommit, AWS CodeDeploy, AWS CodeBuild and AWS CodeArtifact, along with Jenkins, GitHub, Artifactory, TeamCity, and other similar tools. Some organizations might call this a DevOps team, but AWS discourages this and instead encourages thinking of DevOps as the sum of the people, processes, and tools in software delivery.

Testing stages in continuous integration and continuous delivery

The three CI/CD teams should incorporate testing into the software development lifecycle at the different stages of the CI/CD pipeline. Overall, testing should start as early as possible. The following testing pyramid is a concept provided by Mike Cohn in the book *Succeeding with Agile*. It shows the various software tests in relation to their cost and the speed at which they run.

Ref: Mike Cohn, Succeeding with Agile
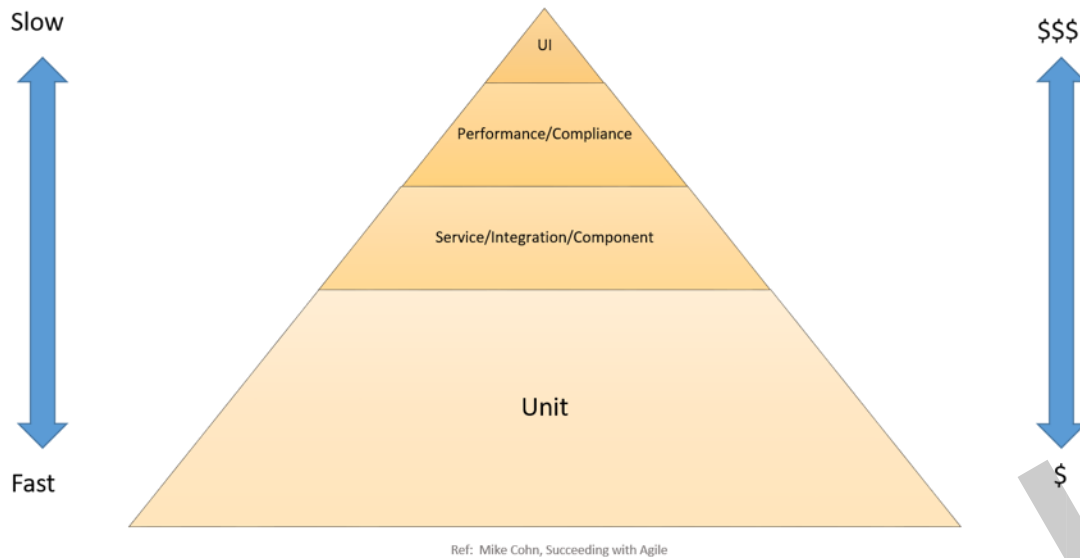
CI/CD testing pyramid

Unit tests are on the bottom of the pyramid. They are both the fastest to run and the least expensive. Therefore, unit tests should make up the bulk of your testing strategy. A good rule of thumb is about 70 percent. Unit tests should have near-complete code coverage because bugs caught in this phase can be fixed quickly and cheaply.

Service, component, and integration tests are above unit tests on the pyramid. These tests require detailed environments and therefore, are more costly in infrastructure requirements and slower to run. Performance and compliance tests are the next level. They require production-quality environments and are more expensive yet. UI and user acceptance tests are at the top of the pyramid and require production-quality environments as well.

All of these tests are part of a complete strategy to assure high-quality software. However, for speed of development, emphasis is on the number of tests and the coverage in the bottom half of the pyramid.

The following sections discuss the CI/CD stages.

Setting up the source

At the beginning of the project, it's essential to set up a source where you can store your raw code and configuration and schema changes. In the source stage, choose a source code repository such as one hosted in GitHub or AWS CodeCommit.

Setting up and running builds

Build automation is essential to the CI process. When setting up build automation, the first task is to choose the right build tool. There are many build tools, such as:

- Ant, Maven, and Gradle for Java

- Make for C/C++

- Grunt for JavaScript

- Rake for Ruby

The build tool that will work best for you depends on the programming language of your project and the skill set of your team. After you choose the build tool, all the dependencies need to be clearly defined in the build scripts, along with the build steps. It's also a best practice to version the final build artifacts, which makes it easier to deploy and to keep track of issues.

Building

In the build stage, the build tools will take as input any change to the source code repository, build the software, and run the following types of tests:

Unit testing – Tests a specific section of code to ensure the code does what it is expected to do. The unit testing is performed by software developers during the development phase. At this stage, a static code analysis, data flow analysis, code coverage, and other software verification processes can be applied.

Static code analysis – This test is performed without actually executing the application after the build and unit testing. This analysis can help to find coding errors and security holes, and it also can ensure conformance to coding guidelines.

Staging

In the staging phase, full environments are created that mirror the eventual production environment. The following tests are performed:

Integration testing – Verifies the interfaces between components against software design. Integration testing is an iterative process and facilitates building robust interfaces and system integrity.

Component testing – Tests message passing between various components and their outcomes. A key goal of this testing could be idempotency in component testing. Tests can include extremely large data volumes, or edge situations and abnormal inputs.

System testing – Tests the system end-to-end and verifies if the software satisfies the business requirement. This might include testing the user interface (UI), API, backend logic, and end state.

Performance testing – Determines the responsiveness and stability of a system as it performs under a particular workload. Performance testing also is used to investigate, measure, validate, or verify other quality attributes of the system, such as scalability, reliability, and resource usage. Types of performance tests might include load tests, stress tests, and spike tests. Performance tests are used for benchmarking against predefined criteria.

Compliance testing – Checks whether the code change complies with the requirements of a nonfunctional specification and/or regulations. It determines if you are implementing and meeting the defined standards.

User acceptance testing – Validates the end-to-end business flow. This testing is executed by an end user in a staging environment and confirms whether the system meets the requirements of the requirement specification. Typically, customers employ alpha and beta testing methodologies at this stage.

Production

Finally, after passing the previous tests, the staging phase is repeated in a production environment. In this phase, a final canary test can be completed by deploying the new code only on a small subset of servers or even one server, or one AWS Region before deploying code to the entire production environment. Specifics on how to safely deploy to production are covered in the Deployment Methods section.

The next section discusses building the pipeline to incorporate these stages and tests.
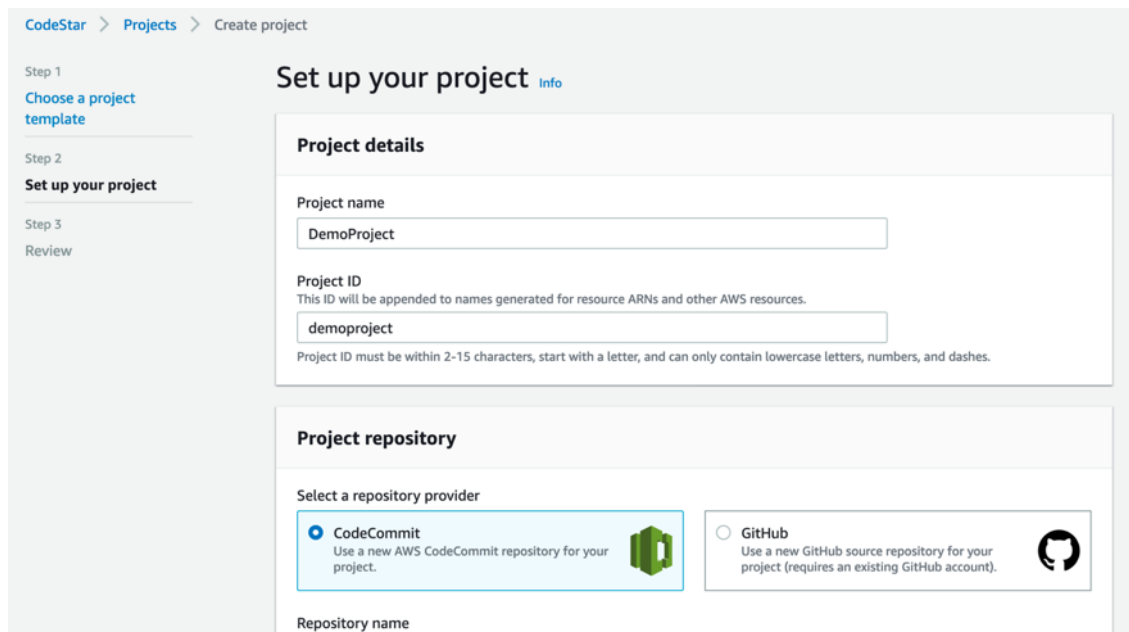
Building the pipeline

This section discusses building the pipeline. Start by establishing a pipeline with just the components needed for CI and iteratively will create a continuous delivery pipeline with more components and stages. This section also discusses how you can consider using AWS Lambda functions and manual approvals for large projects, plan for multiple teams, branches, and AWS Regions.

Starting with a minimum viable pipeline for continuous integration

Your organization's journey toward continuous delivery begins with a minimum viable pipeline (MVP). As discussed in Implementing continuous integration and continuous delivery, teams can start with a very simple process, such as implementing a pipeline that performs a code style check or a single unit test without deployment.

A key component is a continuous delivery orchestration tool. To help you build this pipeline, Amazon developed AWS CodeStar.
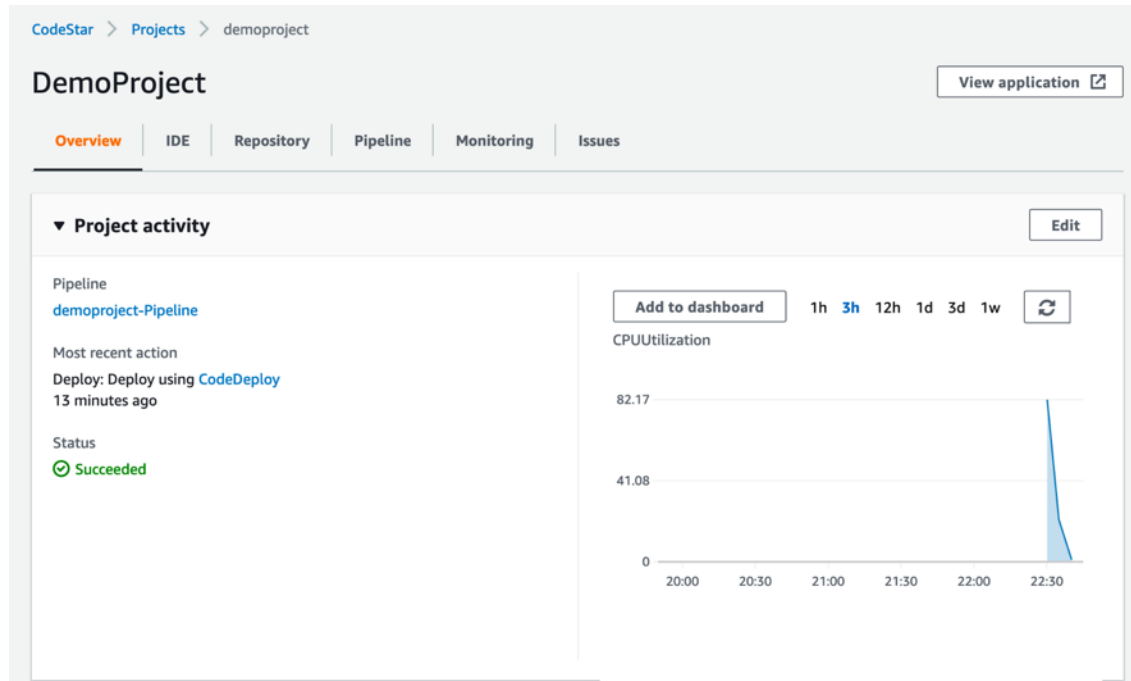

AWS CodeStar setup page

AWS CodeStar uses AWS CodePipeline, AWS CodeBuild, AWS CodeCommit, and AWS CodeDeploy with an integrated setup process, tools, templates, and dashboard. AWS CodeStar provides everything you need to quickly develop, build, and deploy applications on AWS. This allows you to start releasing code faster. Customers who are already familiar with the AWS Management Console and seek a higher level of control can manually configure their developer tools of choice and can provision individual AWS services as needed.

For the latest version of this document, visit:

**https://docs.aws.amazon.com/whitepapers/latest/
practicing-continuous-integration-continuous-
delivery/welcome.html**

AWS CodeStar dashboard

AWS CodePipeline is a CI/CD service that can be used through AWS CodeStar or through the AWS Management Console for fast and reliable application and infrastructure updates. AWS CodePipeline builds, tests, and deploys your code every time there is a code change, based on the release process models you define. This enables you to rapidly and reliably deliver features and updates. You can easily build out an end-to-end solution by using our pre-built plugins for popular third-party services like GitHub or by integrating your own custom plugins into any stage of your release process. With AWS CodePipeline you only pay for what you use. There are no upfront fees or long-term commitments.

The steps of AWS CodeStar and AWS CodePipeline map directly to the source, build, staging, and production CI/CD stages. While continuous delivery is desirable, you could start out with continuous integration, using a pipeline that performs a build action:
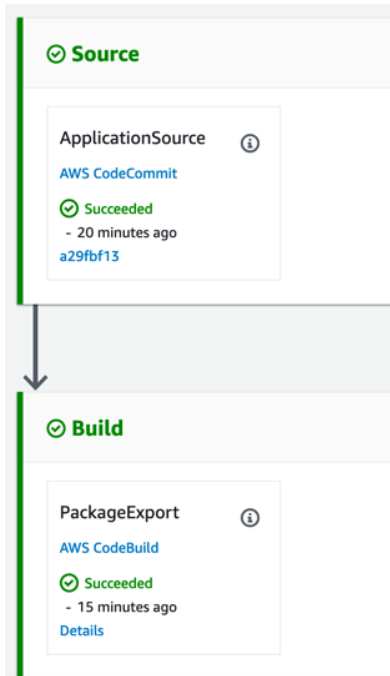
AWS CodePipeline - source and build stages

For AWS CodePipeline, the source stage can accept inputs from GitHub, AWS CodeCommit, and Amazon Simple Storage Service (Amazon S3). Automating the build process is a critical first step for implementing continuous delivery and moving toward continuous deployment. Eliminating human involvement in producing build artifacts removes the burden from your team, minimizes errors introduced by manual packaging, and allows you to start packaging consumable artifacts more often.

AWS CodePipeline works seamlessly with AWS CodeBuild, a fully managed build service, to make it easier to set up a build step within your pipeline that packages your code and runs unit tests. With AWS CodeBuild, you don't need to provision, manage, or scale your own build servers. AWS CodeBuild scales continuously and processes multiple builds concurrently so your builds are not left waiting in a queue. AWS CodePipeline also integrates with build servers such as Jenkins, Solano CI, and TeamCity.

For example, if you need static analysis of your code, linting, code checks, and code metrics, establishing a run in parallel using AWS CodeBuild, those steps can be added as new projects without any further effort in building or installing build servers to handle the load.

CodePipeline — build functionality

The source and build stages shown in the figure *AWS CodePipeline—source and build stages*, along with supporting processes and automation, support your team's transition toward a continuous integration. At this level of maturity, developers need to regularly pay attention to build and test results. They need to grow and maintain a healthy unit test base as well. This, in turn, bolsters the entire team's confidence in the CI/CD pipeline and furthers its adoption.

This version has been archived.
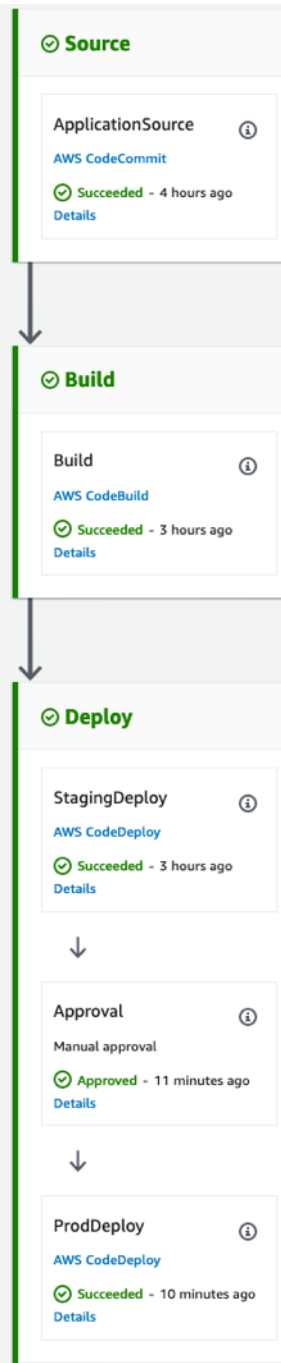
For the latest version of this document, visit:

https://docs.aws.amazon.com/whitepapers/latest/ practicing-continuous-integration-continuous- delivery/welcome.html

AWS CodePipeline stages

This vers... ...rchived.

For the latest v... ...document, visit:

**https://docs.aws.a... ...itepapers/latest/
practicing-conti... ...on-continuous-
delivery/welcome.html**

Continuous delivery pipeline

After the continuous integration pipeline has been implemented and supporting processes have been established, your teams can start transitioning toward the continuous delivery pipeline. This transition requires teams to automate both building and deploying applications.

A continuous delivery pipeline is characterized by the presence of staging and production steps, where the production step is performed after a manual approval.

In the same manner the continuous integration pipeline was built, your teams can gradually start building a continuous delivery pipeline by writing their deployment scripts.

Depending on the needs of an application, some of the deployment steps can be abstracted by existing AWS services. For example, AWS CodePipeline directly integrates with AWS CodeDeploy, a service that automates code deployments to Amazon EC2 instances and instances running on-premises, AWS OpsWorks, a configuration management service that helps you operate applications using Chef, and to AWS Elastic Beanstalk, a service for deploying and scaling web applications and services.

AWS has detailed documentation on how to implement and integrate AWS CodeDeploy with your infrastructure and pipeline.

After your team successfully automates the deployment of the application, deployment stages can be expanded with various tests. For example, you can add other out-of-the-box integrations with services like Ghost Inspector, Runscope, and others as shown in the following figure.

This version has been archived.

For the latest version of this document, visit:

**https://docs.aws.amazon.com/whitepapers/latest/
practicing-continuous-integration-continuous-
delivery/welcome.html**

BetaDeploy ⓘ			
AWS CodeDeploy			
⊘ Succeeded - Just now			
Details			

↓

VerifyDeployment ⓘ	**UITesting** ⓘ	**APITest** ⓘ
AWS Lambda ↗	**Ghost Inspector UI Testing** ↗	**Runscope API Monitoring** ↗
⊙ In progress - Just now	⊘ Succeeded - Just now	⊘ Succeeded - Just now
Details ↗	**Details** ↗	**Details** ↗
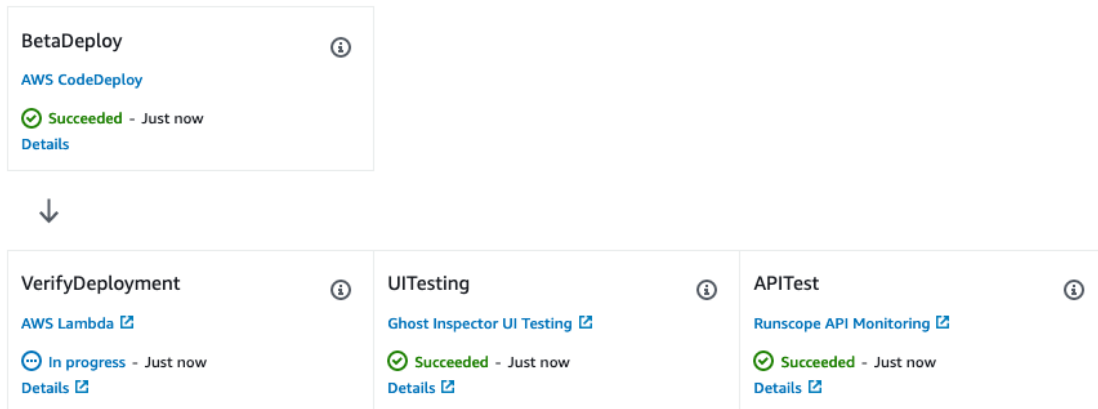
AWS CodePipeline—code tests in deployment stages

Adding Lambda actions

AWS CodeStar and AWS CodePipeline support integration with AWS Lambda. This integration enables implementation of a broad set of tasks, such as creating custom resources in your environment, integrating with third-party systems (such as Slack), and performing checks on your newly deployed environment.

Lambda functions can be used in CI/CD pipelines to do the following tasks:

- Roll out changes to your environment by applying or updating an AWS CloudFormation template.

- Create resources on demand in one stage of a pipeline using AWS CloudFormation and delete them in another stage.

- Deploy application versions with zero downtime in AWS Elastic Beanstalk with a Lambda function that swaps Canonical Name record (CNAME) values.

- Deploy to Amazon EC2 Container Service (ECS) Docker instances.

- Back up resources before building or deploying by creating an AMI snapshot.

- Add integration with third-party products to your pipeline, such as posting messages to an Internet Relay Chate (IRC) client.

Manual approvals

Add an approval action to a stage in a pipeline at the point where you want the pipeline processing to stop so that someone with the required AWS Identity and Access Management (IAM) permissions can approve or reject the action.

If the action is approved, the pipeline processing resumes. If the action is rejected—or if no one approves or rejects the action within seven days of the pipeline reaching the action and stopping—the result is the same as an action failing, and the pipeline processing does not continue.

AWS CodePipeline manual approval action
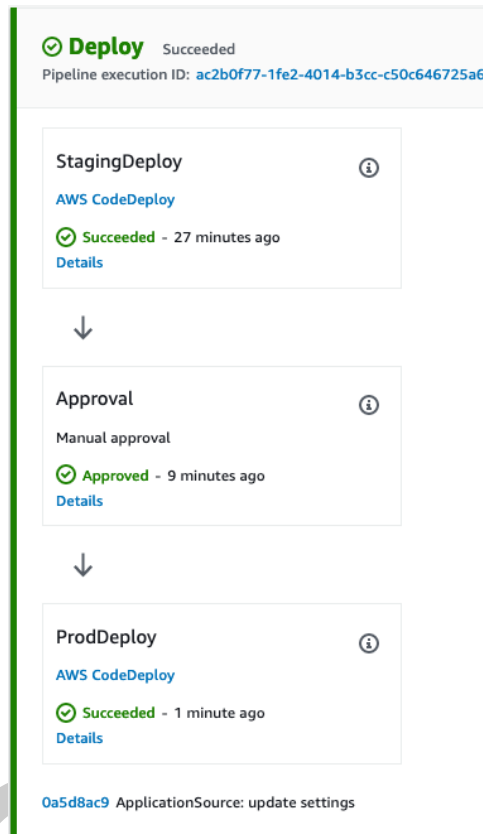
Deploying infrastructure code changes in a CI/CD pipeline

AWS CodePipeline lets you select AWS CloudFormation as a deployment action in any stage of your pipeline. You can then choose the specific action you would like AWS CloudFormation to perform, such as creating or deleting stacks and creating or executing change sets. A stack is an AWS CloudFormation concept and represents a group of related AWS resources. While there are many ways to provision Infrastructure as Code, AWS CloudFormation is a good choice and is recommended by AWS as a scalable, complete solution. It provisions the largest and most comprehensive set of AWS resources as code. AWS recommends using AWS CloudFormation in an AWS CodePipeline project to track infrastructure changes and tests.

CI/CD for serverless applications

You can also use AWS CodeStar, AWS CodePipeline, AWS CodeBuild, and AWS CloudFormation to build CI/CD pipelines for serverless applications. Serverless applications integrate managed services such as Amazon Cognito, Amazon S3, and Amazon DynamoDB with event-driven service, and AWS Lambda to deploy applications in a manner which doesn't require managing servers. If you are a serverless application developer, you can use the combination of AWS CodePipeline, AWS CodeBuild, and AWS CloudFormation to automate the building, testing, and deployment of serverless applications that are expressed in templates built with the AWS Serverless Application Model (SAM). For more information, refer to the AWS Lambda documentation for Automating Deployment of Lambda-based Applications.

You can also create secure CI/CD pipelines that follow your organization's best practices with AWS Serverless Application Model Pipelines (AWS SAM Pipelines). AWS SAM Pipelines are a new feature of AWS SAM CLI that give you access to benefits of CI/CD in minutes, such as accelerating deployment frequency, shortening lead time for changes, and reducing deployment errors. AWS SAM Pipelines come with a set of default pipeline templates for AWS CodeBuild/CodePipeline that follow AWS deployment best practices. For more information and to view the tutorial, refer to the blog Introducing AWS SAM Pipelines.

Pipelines for multiple teams, branches, and AWS Regions

For a large project, it's not uncommon for multiple project teams to work on different components. If multiple teams use a single code repository, it can be mapped so that each team has its own branch. There should also be an integration or release branch for the final merge of the project. If a service-oriented or microservice architecture is used, each team could have its own code repository.

In the first scenario, if a single pipeline is used it's possible that one team could affect the other teams' progress by blocking the pipeline. AWS recommends that you create specific pipelines for team branches and another release pipeline for the final product delivery.
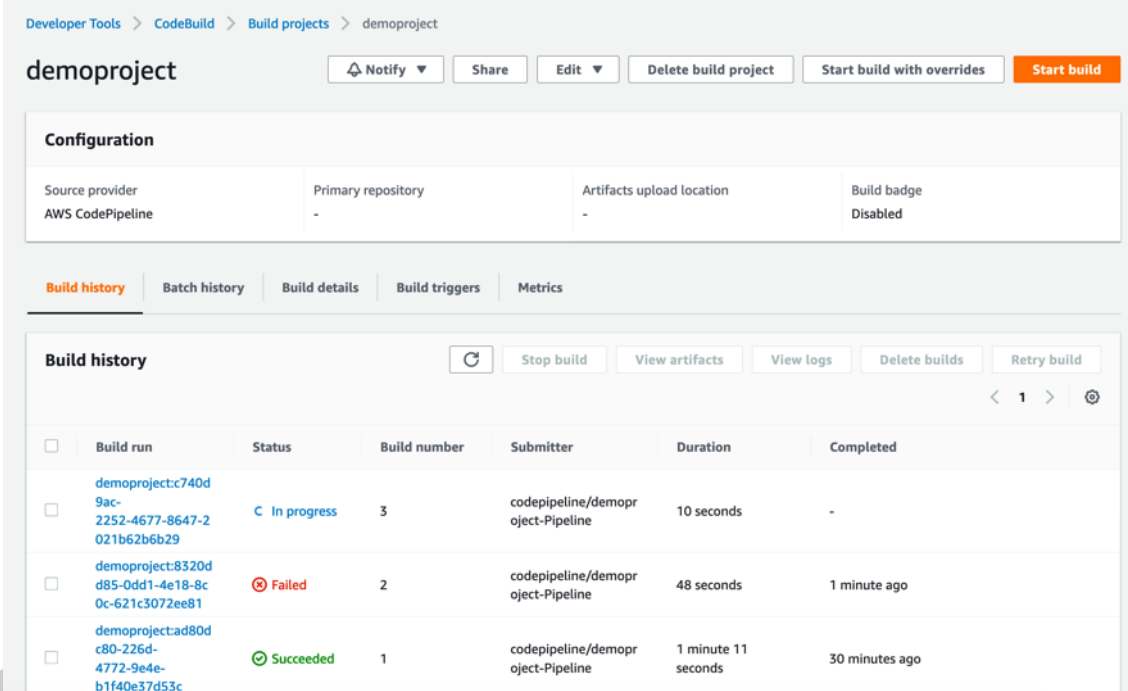
Pipeline integration with AWS CodeBuild

AWS CodeBuild is designed to enable your organization to build a highly available build process with almost unlimited scale. AWS CodeBuild provides quickstart environments for a number of popular languages plus the ability to run any Docker container that you specify.

With the advantages of tight integration with AWS CodeCommit, AWS CodePipeline, and AWS CodeDeploy, as well as Git and CodePipeline Lambda actions, the CodeBuild tool is highly flexible.

Software can be built through the inclusion of a `buildspec.yml` file that identifies each of the build steps, including pre- and post- build actions, or specified actions through the CodeBuild tool.

You can view detailed history of each build using the CodeBuild dashboard. Events are stored as Amazon CloudWatch Logs log files.

CloudWatch Logs log files in AWS CodeBuild

Pipeline integration with Jenkins

For the latest version of this document, visit:

You can use the Jenkins build tool to [create delivery pipelines](#). These pipelines use standard jobs that define steps for implementing continuous delivery stages. However, this approach might not be optimal for larger projects because the current state of the pipeline doesn't persist between Jenkins restarts, implementing manual approval is not straightforward, and tracking the state of a complex pipeline can be complicated.

**https://docs.aws.amazon.com/whitepapers/latest/
practicing-continuous-integration-continuous-
delivery/welcome.html**

Instead, AWS recommends that you implement continuous delivery with Jenkins by using the [AWS Code Pipeline plugin](#). This plugin allows complex workflows to be described using Groovy-like domain-specific language and can be used to orchestrate

complex pipelines. The AWS Code Pipeline plugin's functionality can be enhanced by the use of satellite plugins such as the Pipeline Stage View Plugin, which visualizes the current progress of stages defined in a pipeline, or Pipeline Multibranch Plugin, which groups builds from different branches.

AWS recommends that you store your pipeline configuration in *Jenkinsfile* and have it checked into a source code repository. This allows for tracking changes to pipeline code and becomes even more important when working with the Pipeline Multibranch Plugin. AWS also recommends that you divide your pipeline into stages. This logically groups the pipeline steps and also enables the Pipeline Stage View Plugin to visualize the current state of the pipeline.

The following figure shows a sample Jenkins pipeline, with four defined stages visualized by the Pipeline Stage View Plugin.

Main - Stage View

	Dev	QA	Staging	Production
Average stage times:	26s	1min 8s	1s	1s
#2 Oct 27 09:02 No Changes				
almost complete				
#1 Oct 27 08:51 No Changes	27s	1min 8s	1s (paused for 44s)	1s

Defined stages of Jenkins pipeline visualized by the
Pipeline Stage View Plugin

Deployment methods

You can consider multiple deployment strategies and variations for rolling out new versions of software in a continuous delivery process. This section discusses the most common deployment methods: all at once (deploy in place), rolling, immutable, and blue/green. AWS indicates which of these methods are supported by AWS CodeDeploy and AWS Elastic Beanstalk.

The following table summarizes the characteristics of each deployment method.

Table 1 - Characteristics of deployment methods

Method	Impact of failed deployment	Deploy time	Zero downtime	No DNS change	Rollback process	Code deployed to
Deploy in place	Downtime	🕐	X	✓	Re-deploy	Existing instances
Rolling	Single batch out of service. Any successful batches prior to failure running new application version.	🕐🕐 †	✓	✓	Re-deploy	Existing instances
Rolling with additional batch (beanstalk)	Minimal if first batch fails, otherwise similar to rolling.	🕐🕐🕐 †	✓	✓	Re-deploy	New and existing instances
Immutable	Minimal	🕐🕐🕐🕐	✓	✓	Re-deploy	New instances
Traffic splitting	Minimal	🕐🕐🕐🕐	✓	✓	Re-route traffic and terminate new instances	New instances
Blue/green	Minimal	🕐🕐🕐🕐	✓	X	Switch back to old environment	New instances

† *Varies depending on batch size*

All at once (in-place deployment)

All at once (in-place deployment), is a method you can use to roll out new application code to an existing fleet of servers. This method replaces all the code in one deployment action. It requires downtime because all servers in the fleet are updated at once. There is no need to update existing DNS records. In case of a failed deployment, the only way to restore operations is to redeploy the code on all servers again.

In AWS Elastic Beanstalk, this deployment is called *all at once*, and is available for single and load-balanced applications. In AWS CodeDeploy this deployment method is called *in-place deployment* with a deployment configuration of AllAtOnce.

Rolling deployment

With rolling deployment, the fleet is divided into portions so that all of the fleet isn't upgraded at once. During the deployment process two software versions, new and old, are running on the same fleet. This method allows a zero-downtime update. If the deployment fails, only the updated portion of the fleet will be affected.

A variation of the rolling deployment method, called canary release, involves deployment of the new software version on a very small percentage of servers at first. This way, you can observe how the software behaves in production on a few servers, while minimizing the impact of breaking changes. If there is an elevated rate of errors from a canary deployment, the software is rolled back. Otherwise, the percentage of servers with the new version is gradually increased.

AWS Elastic Beanstalk has followed the rolling deployment pattern with two deployment options, rolling and rolling with additional batch. These options allow the application to first scale up before taking servers out of service, preserving full capability during the deployment. AWS CodeDeploy accomplishes this pattern as a variation of an in-place deployment with patterns like OneAtATime and HalfAtATime.

Immutable and blue/green deployments

The immutable pattern specifies a deployment of application code by starting an entirely new set of servers with a new configuration or version of application code. This pattern leverages the cloud capability that new server resources are created with simple API calls.

The blue/green deployment strategy is a type of immutable deployment which also requires creation of another environment. Once the new environment is up and passed

all tests, traffic is shifted to this new deployment. Crucially the old environment, that is the "blue" environment, is kept idle in case a rollback is needed.

AWS Elastic Beanstalk supports immutable and blue/green deployment patterns. AWS CodeDeploy also supports the blue/green pattern. For more information on how AWS services accomplish these immutable patterns, refer to the Blue/Green Deployments on AWS whitepaper.

Database schema changes

It's common for modern software to have a database layer. Typically, a relational database is used, which stores both data and the structure of the data. It's often necessary to modify the database in the continuous delivery process. Handling changes in a relational database requires special consideration, and it offers other challenges than the ones present when deploying application binaries. Usually, when you upgrade an application binary you stop the application, upgrade it, and then start it again. You don't really bother about the application state, which is handled outside of the application.

When upgrading databases, you do need to consider state because a database contains much state but comparatively little logic and structure.

The database schema before and after a change is applied should be considered different versions of the database. You could use tools such as Liquibase and Flyway to manage the versions.

In general, those tools employ some variation on these workflows:

- Add a table to the database where a database version is stored.

- Keep track of database change commands and bunch them together in versioned change sets. In the case of Liquibase, these changes are stored in XML files. Flyway employs a slightly different method where the change sets are handled as separate SQL files or occasionally as separate Java classes for more complex transitions.

- When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which change sets to run in order to bring the database up-to-date with the latest version.

Summary of best practices

The following are some best practice dos and don'ts for CI/CD.

Do:

- Treat your infrastructure as code.

    o Use version control for your infrastructure code.

    o Make use of bug tracking/ticketing systems.

    o Have peers review changes before applying them.

    o Establish infrastructure code patterns/designs.

    o Test infrastructure changes like code changes.

- Put developers into integrated teams of no more than 12 self-sustaining members.

- Have all developers commit code to the main trunk frequently, with no long-running feature branches.

- Consistently adopt a build system such as Maven or Gradle across your organization and standardize builds.

- Have developers build unit tests toward 100% coverage of the code base.

- Ensure that unit tests are 70% of the overall testing in duration, number, and scope.

This version has been archived.

- Ensure that unit tests are up-to-date and not neglected. Unit test failures should be fixed, not bypassed.

- Treat you **For the latest version of this document, visit:**

- Establish role-based security controls (that is, who can do what and when).

    o Monitor/track every resource possible.

    https://docs.aws.amazon.com/whitepapers/latest/
    o Alert on services, availability, and response times.
    practicing-continuous-integration-continuous-
    o Capture, learn, and improve. **delivery/welcome.html**

    o Share access with everyone on the team.

    o Plan metrics and monitoring into the lifecycle.

- Keep and track standard metrics.

  - Number of builds.

  - Number of deployments.

  - Average time for changes to reach production.

  - Average time from first pipeline stage to each stage.

  - Number of changes reaching production.

  - Average build time.

- Use multiple distinct pipelines for each branch and team.

Don't:

- Have long-running branches with large complicated merges.

- Have manual tests.

- Have manual approval processes, gates, code reviews, and security reviews.

Conclusion

Continuous integration and continuous delivery provide an ideal scenario for your organization's application teams. Your developers simply push code to a repository. This code will be integrated, tested, deployed, tested again, merged with infrastructure, go through security and quality reviews, and be ready to deploy with extremely high confidence.

This version has been archived.

When CI/CD is used, code quality is improved and software updates are delivered quickly and with high confidence that there will be no breaking changes. The impact of any release can be correlated with data from production and operations. It can be used for planning the next cycle, too—a vital DevOps practice in your organization's cloud transformation.

For the latest version of this document, visit:

https://docs.aws.amazon.com/whitepapers/latest/ practicing-continuous-integration-continuous- delivery/welcome.html

Further reading

For more information on the topics discussed in this whitepaper, refer to the following AWS whitepapers:

- [Overview of Deployment Options on AWS](#)

- [Blue/Green Deployments on AWS](#)

- [Setting up CI/CD pipeline by integrating Jenkins with AWS CodeBuild and AWS CodeDeploy](#)

- [Implementing Microservices on AWS](#)

- [Docker on AWS: Running Containers in the Cloud](#)

Contributors

The following individuals and organizations contributed to this document:

- Amrish Thakkar, Principal Solutions Architect, AWS

- David Stacy, Senior Consultant - DevOps, AWS Professional Services

- Asif Khan, Solutions Architect, AWS

- Xiang Shen, Senior Solutions Architect, AWS

Document revisions

Date	Description
October 27, 2021	Updated content.
June 1, 2017	First publication

This version has been archived.

For the latest version of this document, visit:

https://docs.aws.amazon.com/whitepapers/latest/ practicing-continuous-integration-continuous- delivery/welcome.html