WHAT IS A CONTAINER PIPELINE?

Containerization has revolutionized the way applications are developed and deployed, leading to isolated, dependency-managed, and immutable software applications that can be deployed anywhere. All these advances coupled with reduced resource footprint helps to minimize operational expenditure and management overhead.

In this rapidly evolving technology landscape, software developments need to move at a breakneck speed in order to:

- Meet end-user demands
- Differentiate market conditions

That's why organizations have started to rely on more and more automated software development lifecycles (SDLCs): to cope with this rapid development pace with enough agility to deal with unexpected issues.

The most important aspect of a good SDLC? A delivery pipeline that covers everything—development, testing, deployment, and maintenance—under a single overarching process. In this article, we will see how combining containers and automated delivery pipelines can help create container pipelines that leverage the advantages of both technologies.

(This article is part of our DevOps Guide. Use the right-hand menu to navigate.)

Basics of SDLC pipelines

A pipeline is a workflow that creates an integrated end-to-end process encompassing all the stages of SDLC to develop software efficiently in a timely manner.

At the most basic level, a pipeline will consist of these stages:

- **Development stage**. Application is developed and committed to the code repository
- **Code review/acceptance stage.** Manually review code or run automated code reviews.
- **Build stage.** Build and package the code.
- **Testing stage.** Comprehensive testing stage which can include both automated and manual testing.
- **Deployment stage.** Deploy the tested software to the production environment.

Each stage consists of inbuilt validations and verifications to ensure that there are no errors in the code or package. When an error is encountered, it is quickly notified to the relevant teams so that they can quickly remedy the error. With most tasks automated, the feedback loops are shorter—yet still robust enough to cover all the aspects of the application.

Importantly, these pipelines are not limited to software development. They also cover other areas, such as:

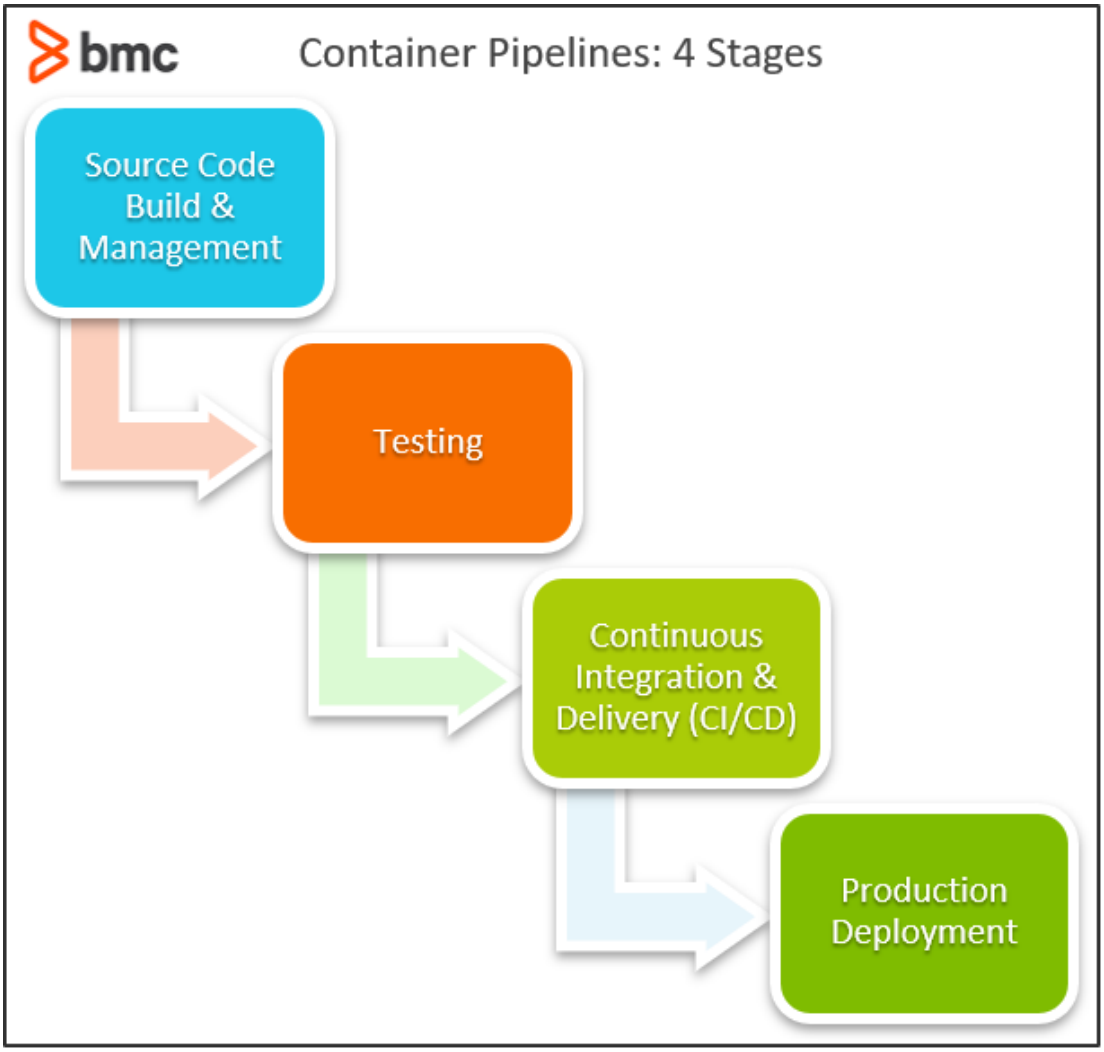
- Database management
- Infrastructure management

Creating a robust pipeline depends on the requirements of your users and the technologies you use.

Container pipeline vs traditional pipeline

Container pipelines differ from traditional pipelines.

The main difference is that container pipelines focus on creating containers and deploying them in an orchestration platform like Kubernetes. There, all the stages mentioned above will remain the same, with noticeable changes coming from the build and deployment stages as they will be focused on containers.



Stages of container pipelines

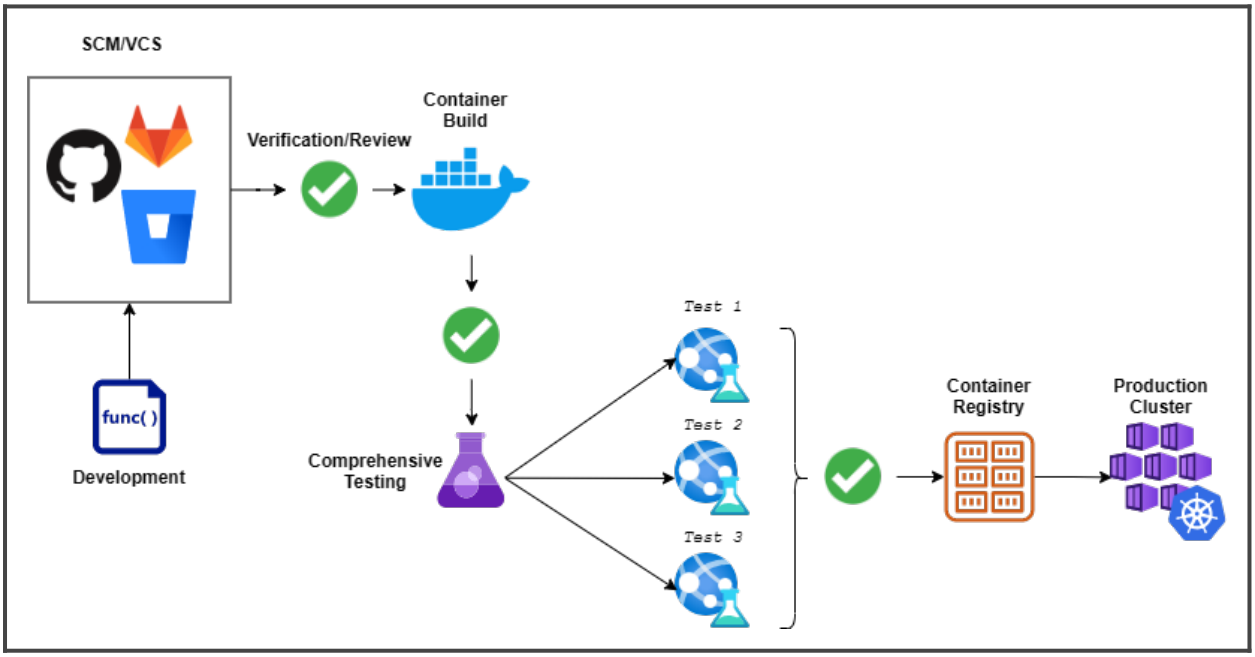
When developing container pipelines, we have to focus on four main areas:

1. Centralized source code management & build
2. Testing
3. CI/CD
4. Production deployment

The complexity and the scope of the pipeline will depend on many factors:

- Application architecture
- Requirements
- Deployment strategy
- The technologies in use

Because containers are the key component, the technologies and architecture will revolve around other technologies and platforms common to containers: Docker, Kubernetes, and Rancher, and similar offerings.



Stage 1. Centralized source code management & build

This is the beginning of any pipeline where all individual developers commit their code to a centralized code repository. This will span the development, code review, and build stages of a pipeline. A pull request will act as the starting point of the automated pipeline workflow in the code repo.

The first task is to run a battery of tests to ensure the functionality of the new code. This will include tests such as:

- Unit tests, which cover the core functionality of code.
- A/B tests, which can be used to compare the performance of two components.

These tests ensure that the end product (code) has optimal code without any bugs. Since this is the first stage of the pipeline, you can fix detected issues before you transfer them to the container.

Next comes the packaging of the application and creating the container image. All the dependencies and the application will be built at the container creation time on top of a predetermined image. Then this image can be pushed to an internal container registry which can be picked up by the pipeline to be deployed in a staging environment.

Stage 2. Testing

The testing stage will start once the container image is built, and the pipeline has deployed the application in a staging environment.

Extensive testing can be carried out to ensure the functionality, performance, and security of the container. (You might even roll these tests into continuous testing.) Some tests to carry out in this phase include:

- **Functional tests** which test functionality of the application, that is: does the application behavior match the requirement?
- **Regression tests** test the new containerized application with the previous versions to ensure the correct functionality.

- **Stress tests** sees how well the container handles stress. In other words, stress tests help determine the behavior of the container in less optimal conditions.
- **Security testing** tests for vulnerabilities and the security of the overall application and containers. (This will include penetration testing, vulnerability scanning, etc.)

Once you're finished testing with these test, you can move the containers to acceptance testing.

You can easily include end users as part of the acceptance testing phase, thanks to the flexibility containerization offers: simply deploy containers in a pre-production environment without any modifications to the application.

What if these containers are a part of a larger cluster or dependent on other containers?

In that case, the testing phase must include fault tolerance tests for cluster failures or communication issues to identify the behavior of the container and the orchestration platform. This ensures that new container versions will not affect the underlying orchestration platform.

Stage 3. Continuous integration & delivery (CI/CD)

Users will require a robust CI/CD tool that provides a platform to power this automated container pipeline and combine all the required functionality of the pipeline. Most CI/CD tools like the following can accommodate container pipelines:

- Jenkins
- CircleCI
- Travis CI
- Teamcity

Even cloud-based CI/CD tools such as AWS CodeBuild or Azure DevOps can power container pipelines. Besides that, CI/CD tools like Jenkins X and Flux are specialized tools with a greater emphasis on powering container pipelines with a GitOps approach and direct integrations with orchestration platforms like Kubernetes.

(Learn how to set up a CI/CI pipeline.)

When selecting a CI/CD tool, the only requirement is to check if the tool offers support for technologies and services used in the pipeline, such as:

- Version control systems
- Build tools
- Container platforms

The more native the support, the better the integration will be.

The primary concerns of the container pipeline CI/CD platform include native support for containerization like:

- Creating containerized images
- Interacting with container registries
- Integrating with orchestration tools

Sure, there may be workarounds for some of these requirements. But these will increase the complexity of the pipeline and might even require manual intervention.

Users also need to consider the expandability of the CI/CD tool. This is crucial if the CI/CD tool covers additional configurations such as database or infrastructure.

Finally, the pipeline should accommodate multiple simultaneous workflows—particularly if multiple teams work on different aspects of software and create multiple containers as in a microservices-based application.

(Compare containers & microservices.)

Stage 4. Production deployment

This is the logical end of a pipeline. The only thing left after the container has been tested and accepted is to deploy the container in the production environment.

As we are dealing with containers, the production release can be done seamlessly and directly to the production environment from the pre-release environment.

Most of the time, containerized deployments will be based on orchestration tools such as Kubernetes and Rancher. This further simplifies the container deployments as these tools offer native support to update and recreate containers without affecting their availability.

Container images can be pushed to a container registry and use the orchestration tool to update the existing containers or build new ones to power the application. Since orchestration tools are platform agnostic, users can utilize them to:

- Deploy to any supported clusters
- Facilitate multi-cloud deployments from a single pipeline

Best practices for container pipelines

Users can create streamlined container pipelines that are capable of delivering containerized applications at scale by focusing on the areas mentioned above and setting up proper workflows for each pipeline stage.

You can further extend these pipelines by integrating infrastructure management to manage all the staging and production infrastructure within the pipeline.

For example, suppose you need a new server or to change the load balancer configurations, and network routing is done from the pipeline through IaC tools (such as Terraform or AWS CloudFormation). In that case, it is paramount that users properly manage external resources such as firewalls, data stores, and backups, even in cloud-based managed Kubernetes clusters.

Including all these things in a container pipeline can cover all aspects of your deployment.

Increase development and flexibility

Container Pipelines are software delivery pipelines that are geared towards building and deploying containers. Combining containerization with automated pipelines using CI/CD tools offers more flexibility to software delivery teams while also speeding up the development process.

Related reading

- BMC DevOps Blog
- Container Sprawl: What It Is & How To Avoid It
- Managing Containers & Code for DevOps
- How To Run MongoDB as a Docker Container
- Automation In DevOps: Why & How To Automate DevOps Practices
- How & Why To Become a Software Factory