codefresh

Enterprise CI/CD Best Practices

2nd Edition

Table of Contents

Table of Contents

If you are trying to learn your way around Continuous Integration/Delivery/Deployment, you might notice that there are mostly two categories of resources:

1. High-level overviews of what CI/CD is and why you need it. These are great for when you are getting started but do not cover anything about day two operations or how to optimize an existing process.
2. Detailed tutorials that cover only a specific aspect of CI/CD (e.g., just unit testing or just deployment) using specific programming languages and tools.

We believe that there is a gap between those two extremes. We are missing a proper guide that sits between those two categories by talking about best practices, but not in an abstract way. If you always wanted to read a guide about CI/CD that explains not just the "why" but also the "how" to apply best practices, then this guide is for you.

We will describe all the basic foundations of effective CI/CD workflows, but instead of talking only in generic terms, we will explain all the technicalities behind each best practice and more importantly, how it can affect you if you don't adopt it.

Setting Priorities

Several companies try to jump on the DevOps bandwagon without having mastered the basics first. You will soon realize that several problems which appear during the CI/CD process are usually pre-existing process problems that only became visible when that company tried to follow best practices in CI/CD pipelines.

The table below summarizes the requirements discussed in the rest of the guide. We also split the requirements according to priority:

- **Critical** requirements are essential to have before adopting DevOps or picking a solution for CI/CD. You should address them first. If you don't, then they will block the process later down the road.
- Requirements with **High** priority are still important to address, but you can fix them while you are adopting a CI/CD platform
- Requirements with **Medium** priority can be addressed in the long run. Even though they will improve your deployment process, you can work around them until you find a proper solution.

Number	Best practice	Category	Importance
• **1**	All project assets are in source control	Artifacts	Critical
• **2**	A single artifact is produced for all environments	Artifacts	High
• **3**	Artifacts move within pipelines (and not source revisions)	Artifacts	High
• **4**	App development happens with short-lived branches (one per feature)	Build	High
• **5**	Builds can be performed in a single step	Build	High
• **6**	Builds are fast (less than 5 minutes)	Build	Medium
• **7**	Store your dependencies	Build	High
• **8**	Tests are automated	Testing	High
• **9**	Tests are fast	Testing	High
• **10**	Tests auto clean their side effects	Testing	High
• **11**	Multiple test suites exist	Testing	Medium
• **12**	Test environments on demand	Testing	Medium
• **13**	Running test suites concurrently	Testing	Medium
• **14**	Security scanning is part of the process	Quality and Audit	High
• **15**	Quality scanning/code reviews are part of the process	Quality and Audit	Medium
• **16**	Database updates have their lifecycle	Databases	High
• **17**	Database updates are automated	Databases	High
• **18**	Database updates are forward and backward compatible	Databases	High

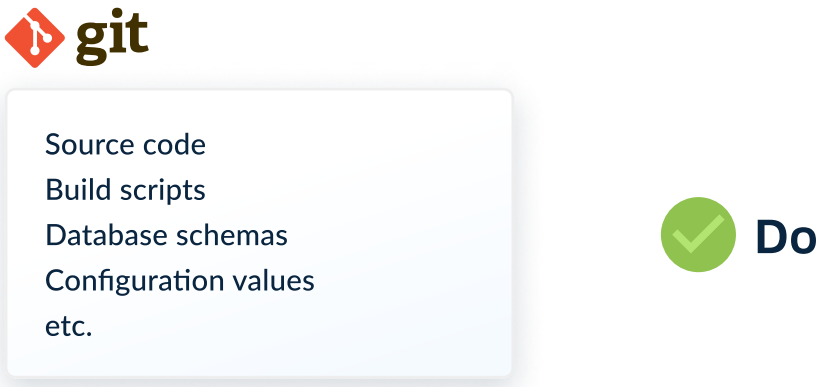
Number	Best practice	Category	Importance
• 19	Deployments happen via a single path (CI/CD server)	Deployments	Critical
• 20	Deployments happen gradually in stages	Deployments	High
• 21	Metrics and logs can detect a bad deployment	Deployments	High
• 22	Automatic rollbacks are in place	Deployments	Medium
• 23	Staging matches production	Deployments	Medium

Best Practice 1

All Project Assets Are in Source Control

Artifact management is perhaps the most important characteristic of a pipeline. At its most basic level, a pipeline creates binary/package artifacts from source code and deploys them to the appropriate infrastructure that powers the application that is being deployed.

The single most important rule to follow regarding assets and source code is the following:



All files that constitute an application should be managed using source control.

Unfortunately, even though this rule seems pretty basic, there are a lot of organizations out there that fail to follow it. Traditionally, developers are using version control systems only for the source code of an application but leave out other supporting files such as installation scripts, configuration values, or test data.

Everything that takes part in the application lifecycle should be checked into source control. This includes but is not limited to:

1. Source code
2. Build scripts
3. Pipeline definition
4. Configuration values
5. Tests and test data
6. Database schemas
7. Database update scripts
8. Infrastructure definition scripts
9. Cleanup/installation/purging scripts
10. Associated documentation

The end goal is that anybody can check out everything that relates to an application and can recreate it locally or in any other alternative environment.

A common anti-pattern we see is deployments happening with a special script that is available only on a specific machine or on the workstation of a specific team member, or even an attachment in a wiki page, and so on.

Version control also means that all these resources are audited and have a detailed history of all changes. If you want to see how the application looked 6 months ago, you can easily use the facilities of your version control system to obtain that information.

Note that even though all these resources should be versioned control, it doesn't have to be in the same repository. Whether you use multiple repositories or a single one, is a decision that needs careful consideration and has not a definitive answer. The important part however is to make sure that everything is indeed version controlled.

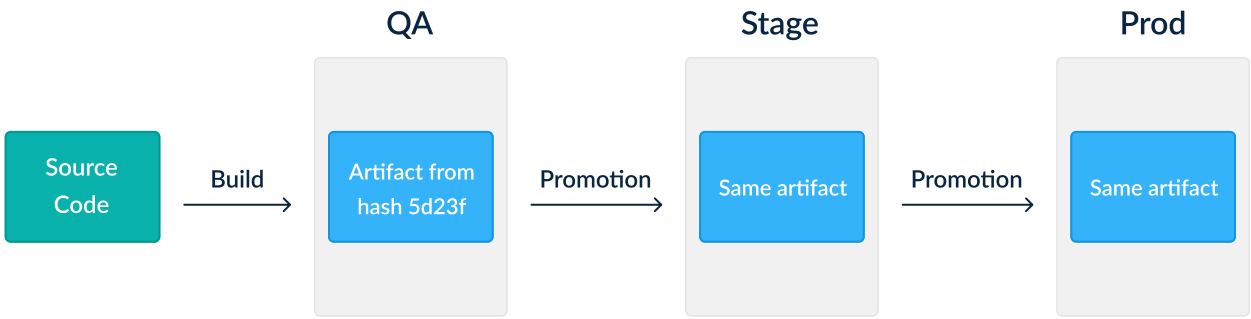
Even though GitOps is the emerging practice of using Git operations for promotions and deployments, you don't need to follow GitOps specifically to follow this best practice. Having historical and auditing information for your project assets is always a good thing, regardless of the actual software paradigm that you follow.

Best Practice 2

A Single Artifact Is Produced For All Environments

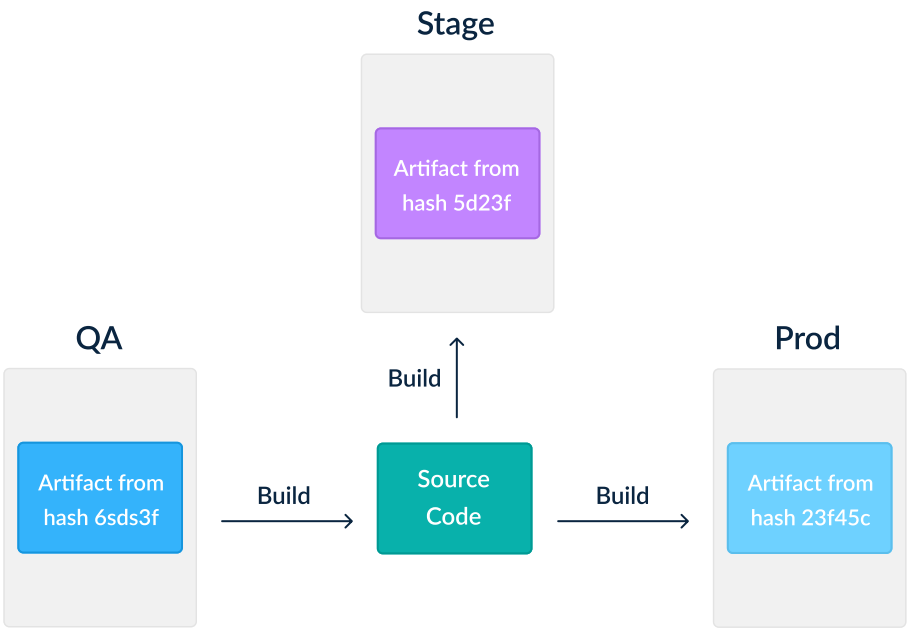
One of the main functionalities of a CI/CD pipeline is to verify that a new feature is fit for deployment to production. This happens gradually as every step in a pipeline is essentially performing additional checks for that feature.

For this paradigm to work, however, you need to make sure that what is being tested and prodded within a pipeline is also the same thing that gets deployed. In practice, this means that a feature/release should be packaged once and be deployed to all successive environments in the same manner.



Do

Unfortunately, a lot of organizations fall into the common trap of creating different artifacts for dev/staging/prod environments because they have not adopted a common infrastructure for configuration. This implies that they deploy a slightly different version of what was tested during the pipeline. Configuration discrepancies and last-minute changes are some of the biggest culprits when it comes to failed deployments, and having a different package per environment exacerbates this problem.



Don't

Instead of creating multiple versions per environment, the accepted practice is to have a single artifact that only changes configuration between different environments. With the appearance of containers and the ability to create a self-sufficient package of an application in the form of Docker images, there is no excuse for not following this practice.

Regarding configuration there are two approaches:

1. The binary artifact/container has all configurations embedded inside it and changes the active one according to the running environment (easy to start, but not very flexible. We don't recommend this approach)

2. The container has no configuration at all. It fetches needed configuration during runtime on demand using a discovery mechanism such as environmental variables passed by Kubernetes manifests, a key/value database, a filesystem volume, a service discovery mechanism, etc. (the recommended approach)

The result is the guarantee where the exact binary/package that is deployed in production is the same one that was tested in the pipeline.

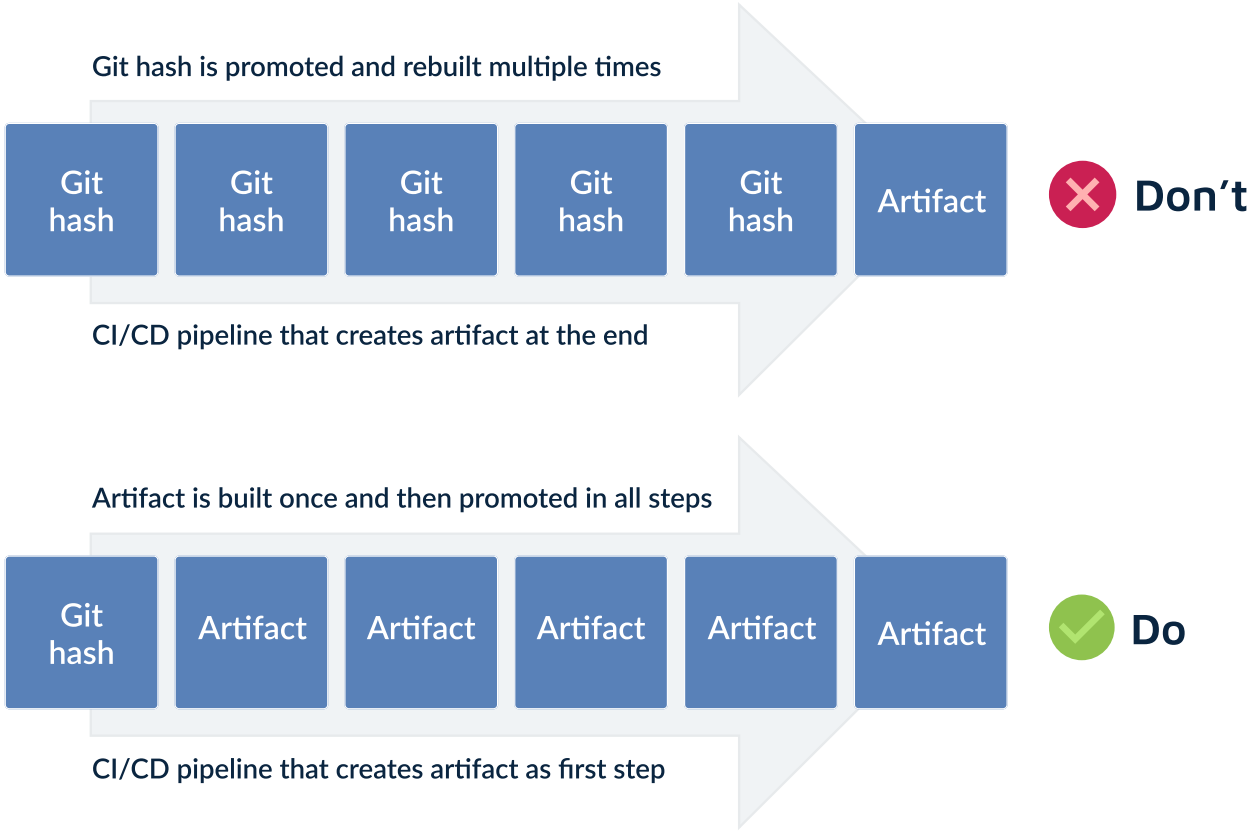
Best Practice 3

Artifacts Move Within Pipelines (and Not Source Revisions)

A corollary to the previous point (the same artifact/package should be deployed in all environments) is the fact that a deployment artifact **should be built only once**.

The whole concept around containers (and VM images in the past) is to have **immutable** artifacts. An application is built only once with the latest feature or features that will soon be released.

Once that artifact is built, it should move from each pipeline step to the next as an unchanged entity. Containers are the perfect vehicle for this immutability as they allow you to create an image only once (at the beginning of the pipeline) and promote it towards production with each successive pipeline step.



Unfortunately, the common anti-pattern seen here is companies promoting commits instead of container images. A source code commit is traveling in the pipeline stages and each step is being rebuilt by checking out the source code again and again.

In theory, containers rebuilt from the same commit should be identical but in practice sometimes minor variations may occur.

It is a bad practice for two other reasons. First of all, it makes the pipeline very slow as packaging and compiling software is a very lengthy process and repeating it at each step is a waste of time and resources.

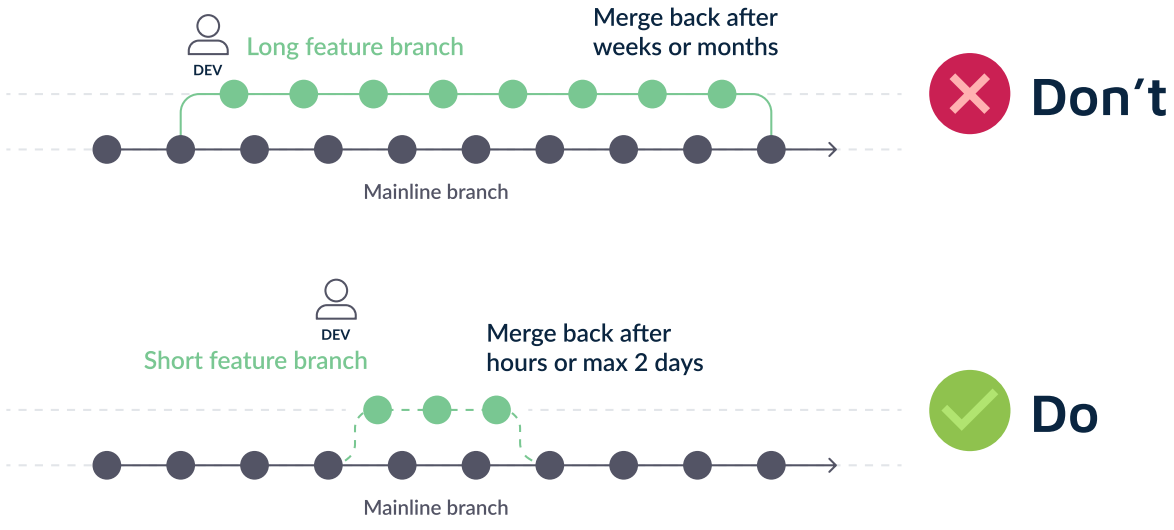
Secondly, it breaks the previous rule. Recompiling a code commit at every pipeline step leaves the window open for resulting in a different artifact than before. You lose the guarantee that what is deploying in production is the same thing that was tested in the pipeline.

Best Practice 4

App Development Happens With Short-Lived Branches (One Per Feature)

A sound pipeline has several quality gates (such as unit tests or security scans) that test the quality of a feature and its applicability to production deployments. In a development environment with a high velocity (and a big development team), not all features are expected to reach production right away. Some features may even clash with each other at their initial deployment version.

To allow for fine-grained quality gating between features, a pipeline should have the power to veto individual features and be able to select only a subset of them for production deployment. The easiest way to obtain this guarantee is following the feature-per-branch methodology where short-lived features (i.e. that can fit within a single development sprint) correspond to individual source control branches.



This makes the pipeline design very simple as everything revolves around individual features. Running test suites against a code branch tests only the new feature. Security scanning of a branch reveals problems with a new feature.

Project stakeholders are then able to deploy and rollback individual features or block complete branches from even being merged into the mainline code.

Unfortunately, there are still companies that have long-lived feature branches that collect multiple and unrelated features in a single batch. This not only makes merging a pain but also becomes problematic in case a single feature is found to have issues (as it is difficult to revert it individually).

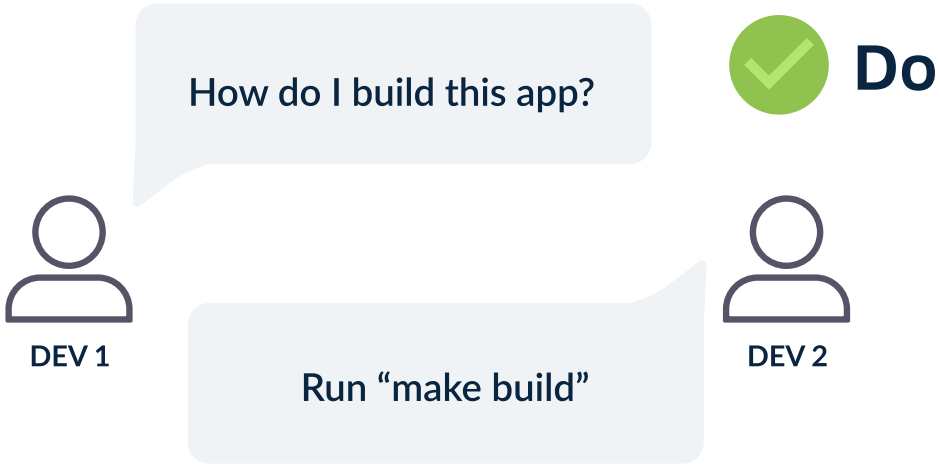
The evolution of short-lived branches is to follow trunk-based development and feature toggles. This can be your endgame but only if you have mastered short-lived branches first.

Best Practice 5

Builds Can Be Performed in a Single Step

> CI/CD pipelines are all about automation. It is very easy to automate something that already was very easy to run in the first place.

Ideally, a simple build of a project should be a single command. That command usually calls the build system or a script (e.g., bash, PowerShell) that is responsible for taking the source code, running some basic tests, and packaging the final artifact/container.

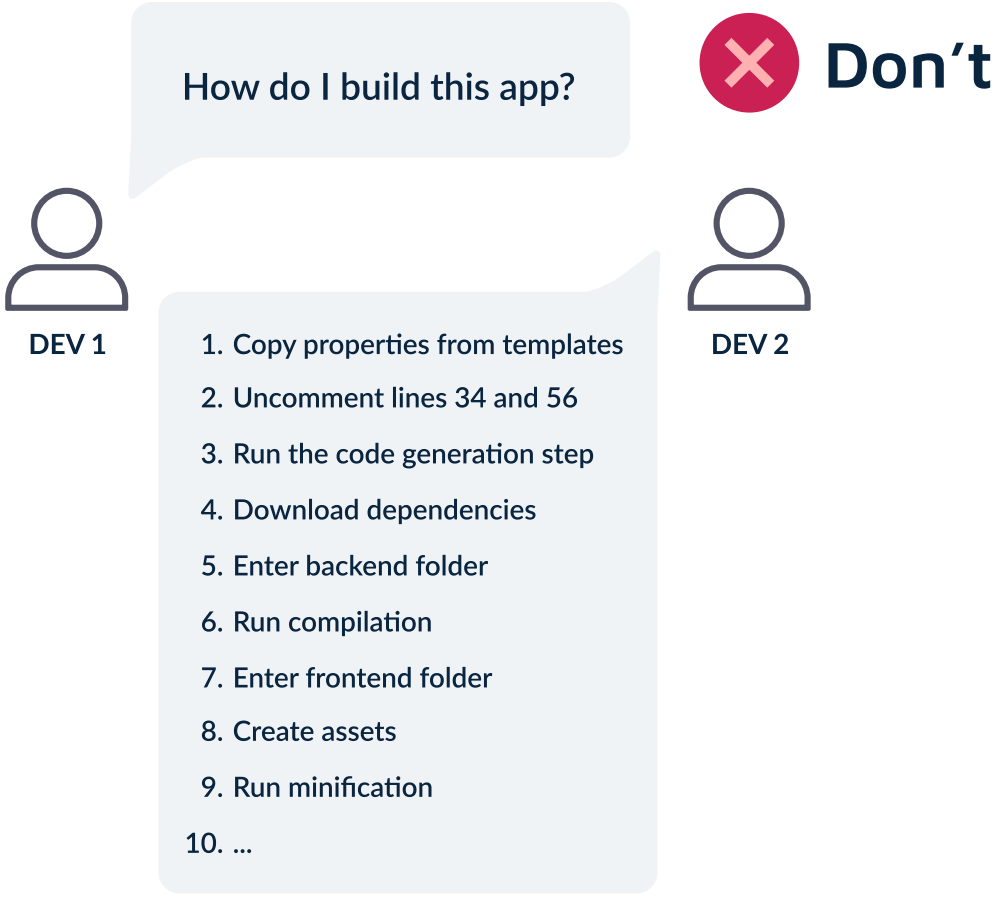


It is ok if more advanced checks (such as load testing) need additional steps. The basic build, however (that results in a deployable artifact) should only involve a single command. A new developer should be able to check out a brand new copy of the source code, execute this single command and get immediately a deployable artifact.

To meet this best practice most teams choose to use either a make file (good) or a dockerfile (better). Because a well constructed dockerfile (or other container standard) is essentially self-contained, it eliminates potential variations on individual user machines.

The same approach is true for deployments (deployments should happen with a single command). Then if you need to create any pipeline you can simply insert that single step in any part of the pipeline.

Unfortunately, there are still companies that suffer from many manual steps to get a basic build running. Downloading extra files, changing properties, and in general having big checklists that need to be followed are steps that should be automated within that very same script.



If a new hire in your development team needs more than 15 minutes for the basic build (after checking out the code in their workstation) then you almost certainly suffer from this problem.

A well-built CI/CD pipeline just repeats what is already possible on the local workstation. The basic build and deploy process should be already well oiled before being moved into a CI/CD platform.

Best Practice 6

Builds Are Fast (Less Than 5 Minutes)

> Having a fast build is a big advantage for both developers and operators/sysadmins.
>
> Developers are happy when the feedback loop between a commit and its side effects is as short as possible. It is very easy to fix a bug in the code that you just committed as it is very fresh on your mind. Having to wait for one hour before developers can detect failed builds is a very frustrating experience.

Builds should be fast both in the CI platform and in the local station. At any given point in time, multiple features are trying to enter the code mainline. The CI server can be easily overwhelmed if building them takes a lot of time.

Operators also gain huge benefits from fast builds. Pushing hot fixes in production or rolling back to previous releases is always a stressful experience. The shorter this experience is, the better. Rollbacks that take 30 minutes are much more difficult to work with than those that take three minutes.

In summary, a basic build should be really fast. Ideally less than five minutes. If it takes more than 10 minutes, your team should investigate the causes and shorten that time. Modern build systems have great caching mechanisms.

- Library dependencies should be fetched from an internal proxy repository instead of the internet
- Avoid the use of code generators unless otherwise needed
- Split your unit (fast) and integration tests (slow) and only use unit tests for the basic build
- Fine-tune your container images to take full advantage of the Docker layer caching

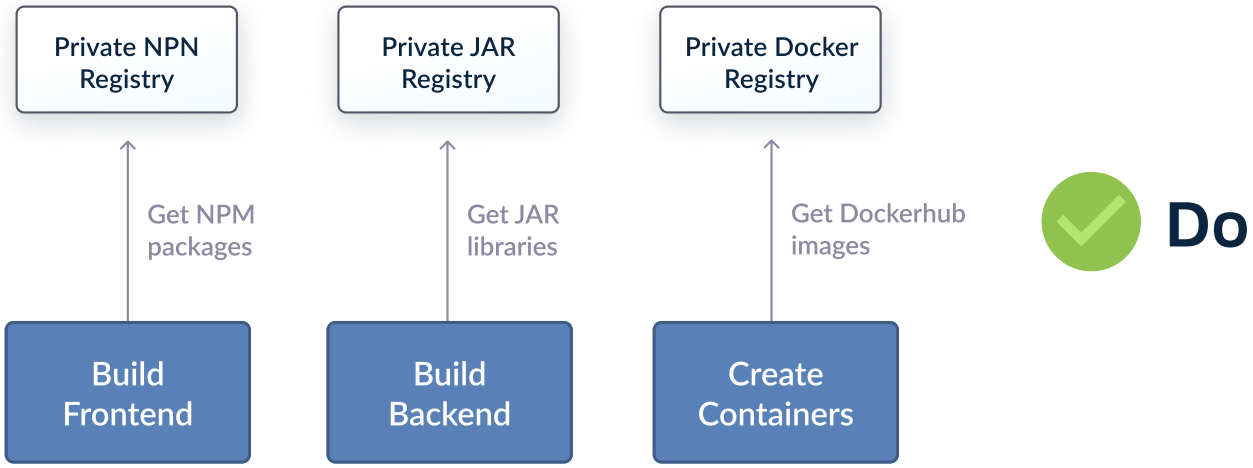
Getting faster builds is also one of the reasons that you should explore if you are moving to microservices.

Best Practice 7

Store Your Dependencies

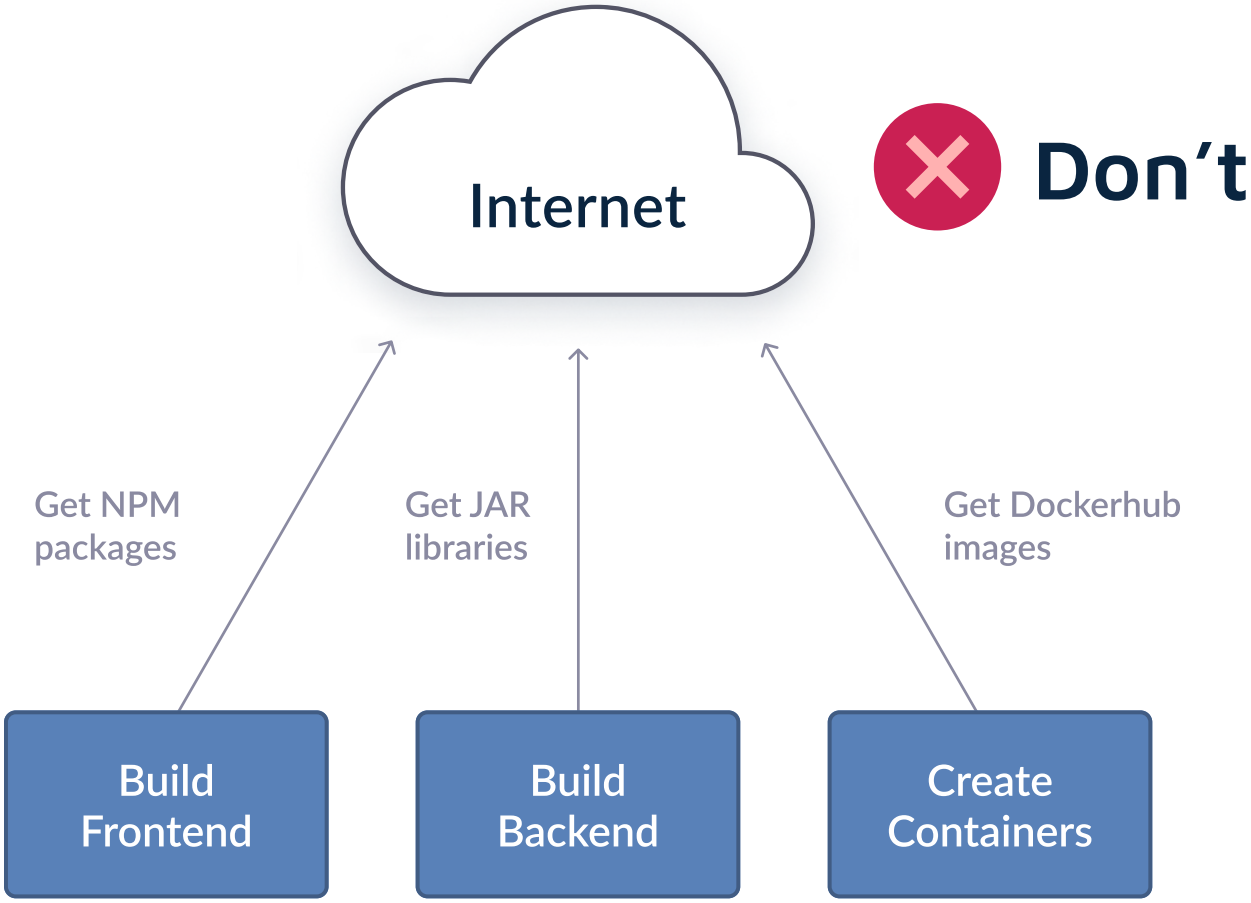
The left-pad incident and the dependency confusion hack famously demonstrate the need to have control over dependencies. While both incidents have great security implications, the truth is that storing your dependencies is also a very important tenet that is fundamental to the stability of your builds.

Every considerable piece of code uses external dependencies in the form of libraries or associated tools. Your code should of course be always stored in Git. But all external libraries should be also stored by you in some sort of artifact repository.



Spend some time to collect our dependencies and understand where they are coming from. Apart from code libraries, other not-so-obvious moving parts are needed to a complete build, such as your base docker images or any command-line utilities.

The best way to test your build for stability is to completely cut off internet access in your build servers (essentially simulating an air-gapped environment). Try to kick off a pipeline build where all your internal services (git, databases, artifact storage, container registry) are available, but nothing else from the public internet is accessible, and see what happens.



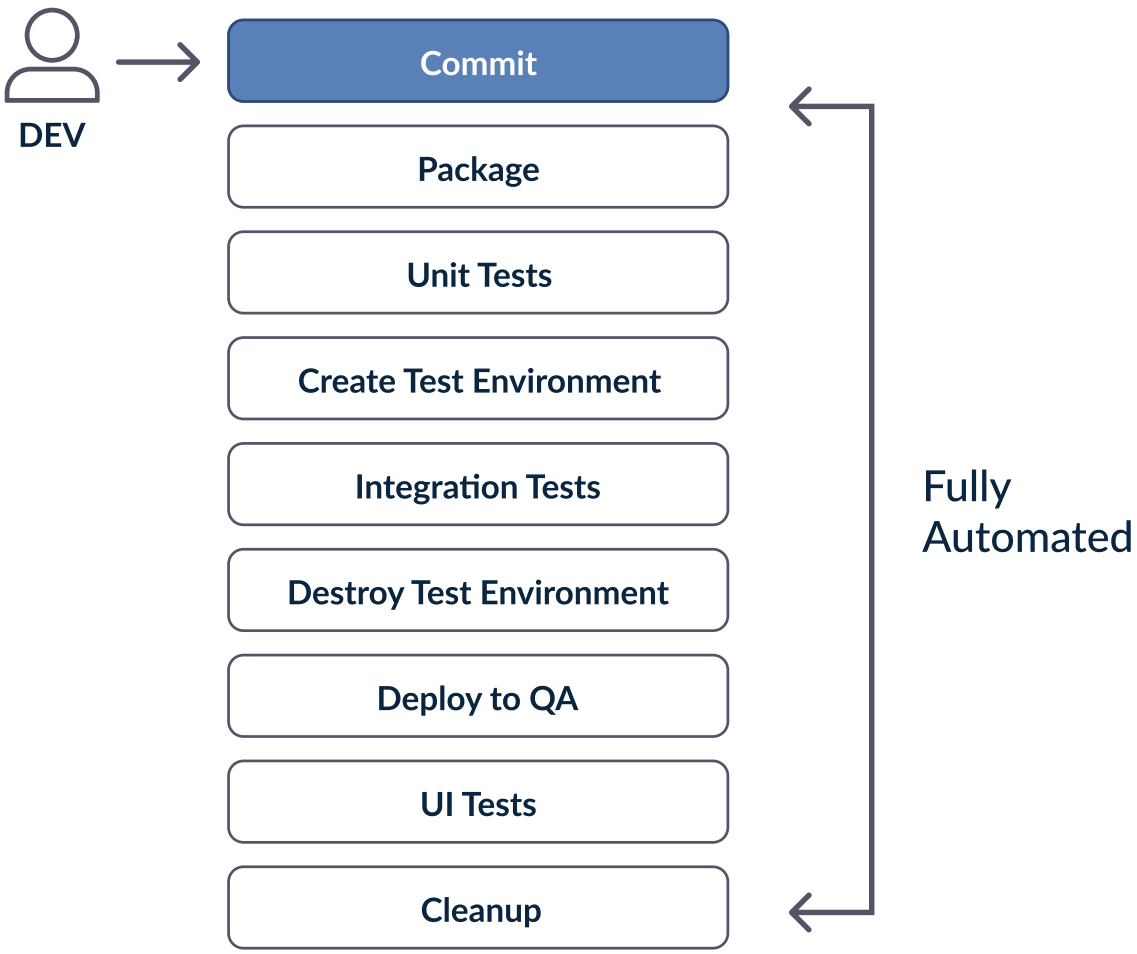
If your build complains about a missing dependency, imagine that the same thing will happen in a real incident if that particular external resource is also down.

Best Practice 8

Tests Are Automated

> The main goal of unit/integration/functional tests is to increase the confidence in each new release that gets deployed. In theory, a comprehensive amount of tests will guarantee that there are no regressions on each new feature that gets published.

To achieve this goal, tests should be fully automated and managed by the CI/CD platform. Tests should be run not only before each deployment but also after a pull request is created. The only way to achieve the level of automation is for the test suite to be runnable in a single step.



Unfortunately, several companies are still creating tests the old-fashioned way, where an army of test engineers is tasked with the manual execution of various test suites. This blocks all new releases as the testing velocity essentially becomes the deployment velocity.

Test engineers should only write new tests. They should never execute tests themselves as this makes the feedback loop of new features vastly longer. Tests are always executed automatically by the CI/CD platform in various workflows and pipelines.

It is ok if a small number of tests are manually run by people as a way to smoke test a release. But this should only happen for a handful of tests. All other main test suites should be fully automated

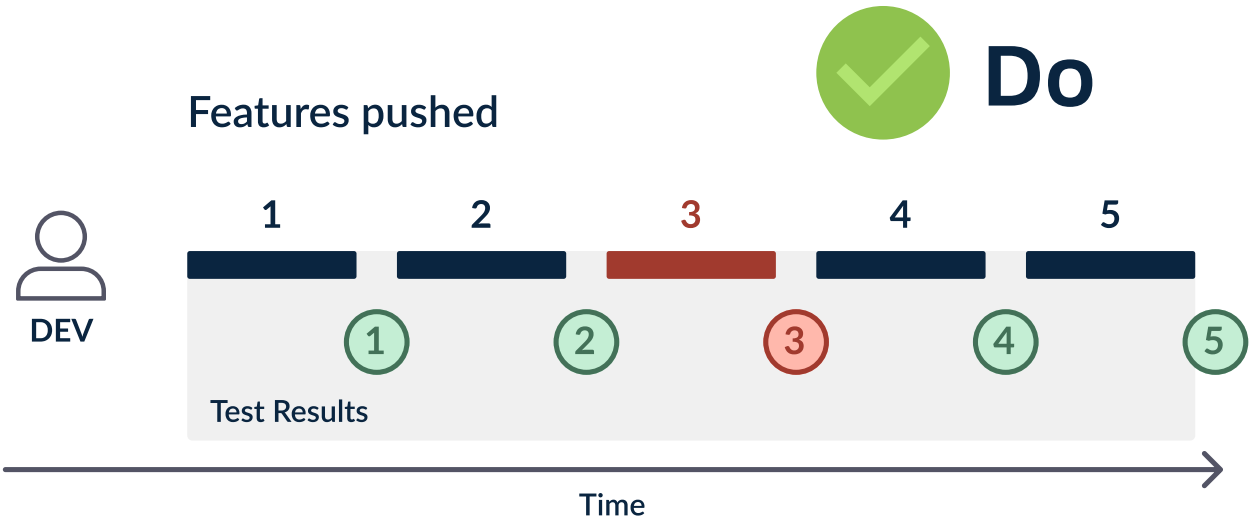
Best Practice 9

Tests Are Fast

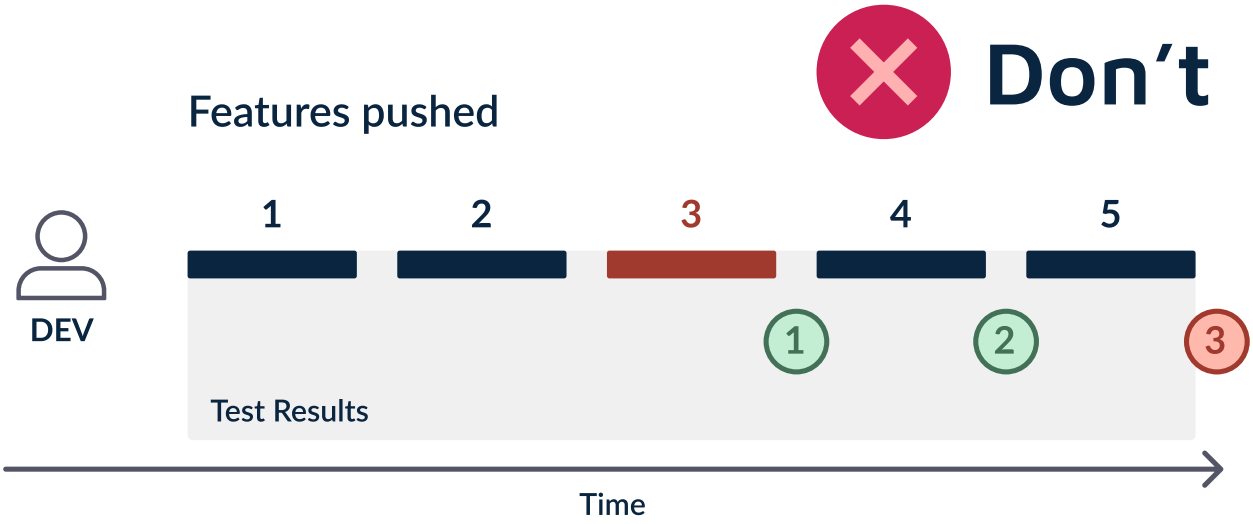
A corollary of the previous section is also the quick execution of tests. If test suites are to be integrated into delivery pipelines, they should be really fast. Ideally, the test time should not be bigger than the packaging/compilation time, which means that tests should finish after five minutes, and no more than 15.

To achieve this goal, tests should be fully automated and managed by the CI/CD platform. Tests should be run not only before each deployment but also after a pull request is created. The only way to achieve the level of automation is for the test suite to be runnable in a single step.

The quick test execution gives confidence to developers that the feature they just committed has no regressions and can be safely promoted to the next workflow stage. A running time of two hours is disastrous for developers as they cannot possibly wait for that amount of time after committing a feature.



If the testing period is that large, developers just move to their next task and change their mind context. Once the test results do arrive, it is much more difficult to fix issues on a feature that you are not actively working on.



Unfortunately, the majority of time waiting for tests steps from ineffective test practices and lack of optimizations. The usual factor of a slow test is code that "sleeps" or "waits" for an event to happen, making the test run longer than it should run. All these sleep statements should be removed and the test should follow an event-driven architecture (i.e., responding to events instead of waiting for things to happen)

Test data creation is also another area where tests are spending most of their data. Test data creation code should be centralized and re-used. If a test has a long setup phase, maybe it is testing too many things or needs some mocking in unrelated services.

In summary, test suites should be fast (5-10 minutes) and huge tests that need hours should be refactored and redesigned.

Best Practice 10

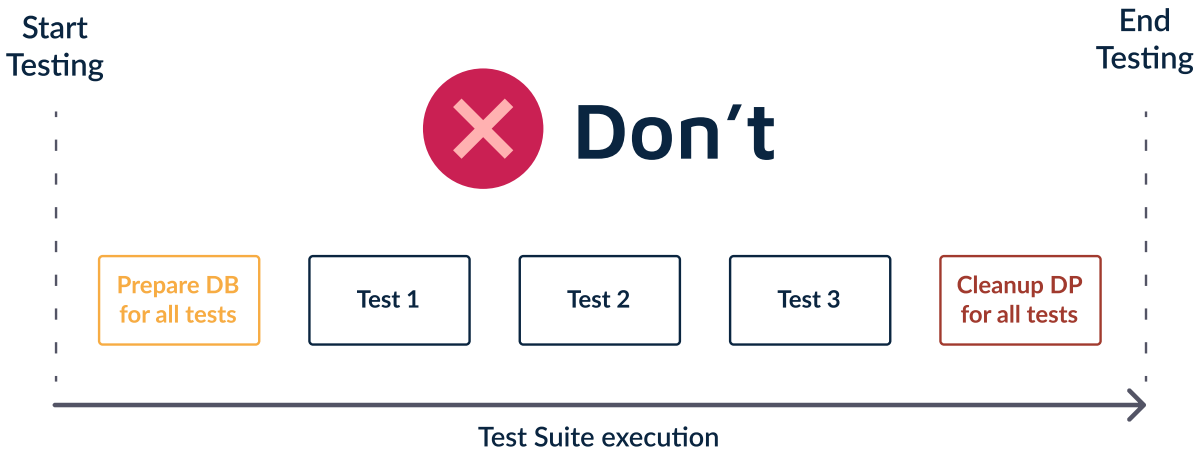
Tests Auto Clean Their Side Effects

Generally speaking, you can split your unit tests is two more categories (apart from unit/integration or slow and fast) and this has to do with their side effects:

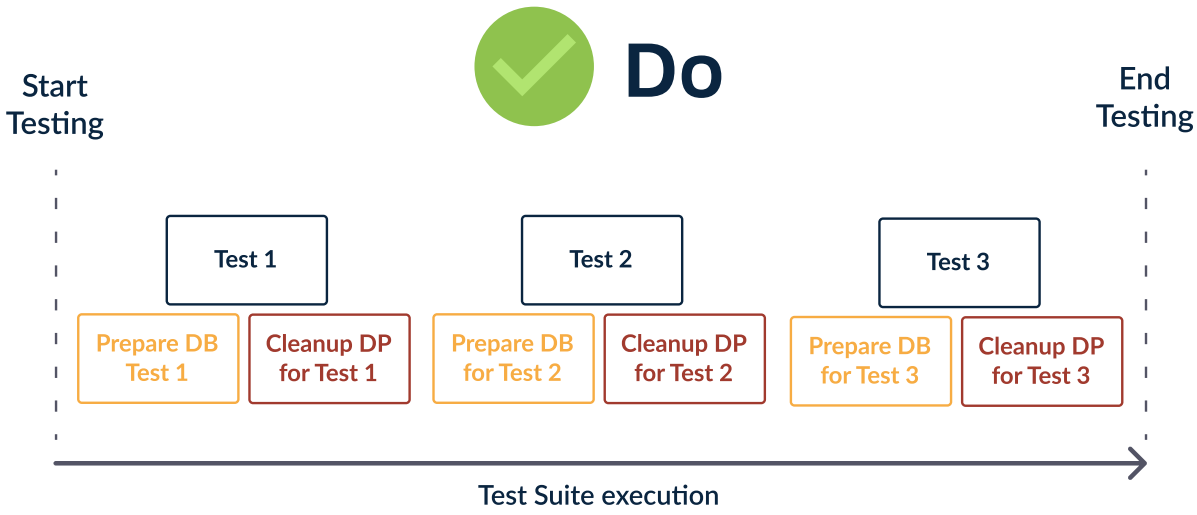
1. Tests that have no side effects. They read only information from external sources, never modify anything and can be run as many times as you want (or even in parallel) without any complications.
2. Tests that have side effects. These are the tests that write stuff to your database, commit data to external systems, perform output operations on your dependencies, and so on.

The first category (read-only tests) is easy to handle since they need no special maintenance. But the second category (read/write tests) is more complex to maintain as you need to make sure that you clean up their actions as soon as the tests finish. There are two approaches to this:

1. Let all the tests run and then clean up the actions of all of them at the end of the test suit
2. Have each test clean-up by itself after it runs (the recommended approach)



Having each test clean up its side-effects is a better approach because it means that you can run all your tests in parallel or any times that you wish individually (i.e., run a single test from your suite and then run it again a second or third time).



Being able to execute tests in parallel is a prerequisite for using dynamic test environments as we will see later in this guide.

Best Practice 11

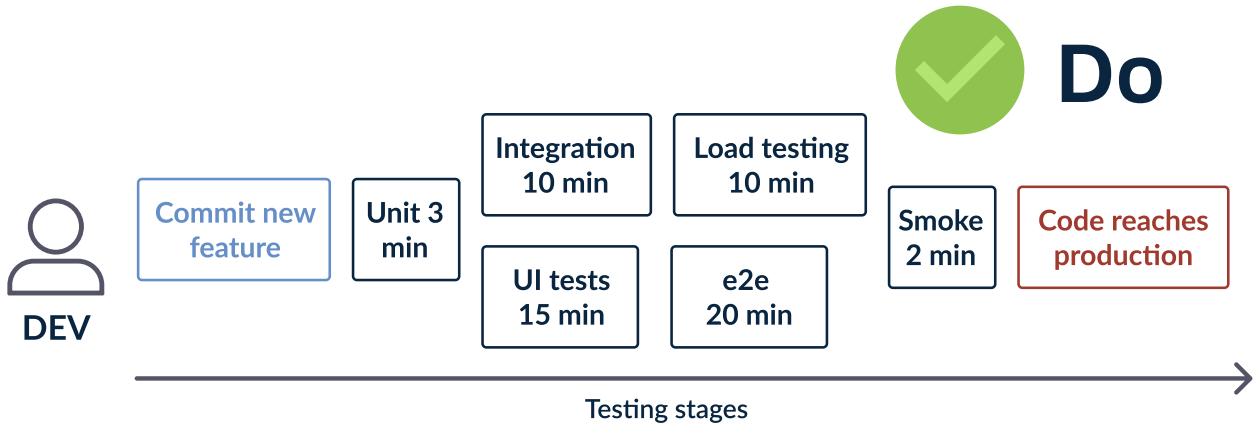
Use Multiple Test Suites Exist

Testing is not something that happens only in a single step inside a CI/CD pipeline. Testing is a continuous process that touches all phases of a pipeline.

This means that multiple test types should exist in any well-designed application. Some of the most common examples are:

- Really quick unit tests that look at major regressions and finish very fast
- Longer integrations tests that look for more complex scenarios (such as transactions or security)
- Stress and load testing
- Contract testing for API changes of external services used
- Smoke tests that can be run on production to verify a release
- UI tests that test the user experience

This is just a sample of different test types. Each company might have several more categories. The idea behind these categories is that developers and operators can pick and choose different testing types for the specific pipeline they create.

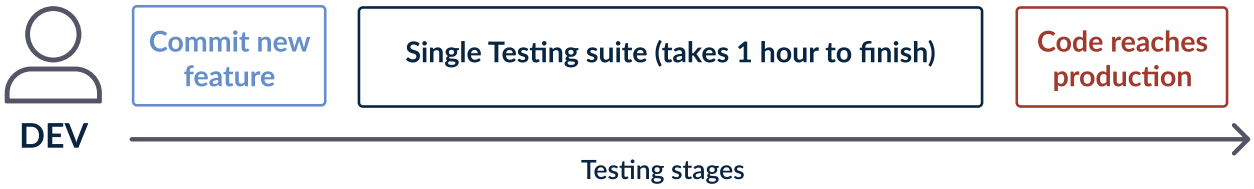


As an example, a pipeline for pull requests might not include stress and load testing phases because they are only needed before a production release. Creating a pull request will only run the fast unit tests and maybe the contact testing suite.

Then after the Pull Request is approved, the rest of the tests (such as smoke tests in production) will run to verify the expected behavior.

Some test suits might be very slow, that running them on demand for every Pull Request is too difficult. Running stress and load testing is usually something that happens right before a release (perhaps grouping multiple pull requests) or in a scheduled manner (a.k.a. Nightly builds)
The exact workflow is not important as each organization has different processes. What is important is the capability to isolate each testing suite and be able to select one or more for each phase in the software lifecycle.

Don't

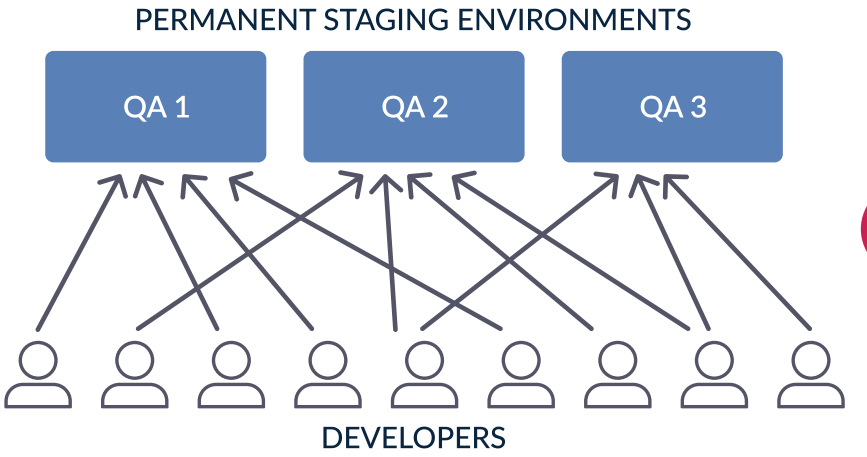


Having a single test suite for everything is cumbersome and will force developers to skip tests locally. Ideally, as a developer, I should be able to select any possible number of test suites to run against my feature branch allowing me to be flexible on how I test my feature.

Best Practice 12

Test Environments On Demand

The traditional way of testing an application right before going into production is with a staging environment. Having only one staging environment is a big disadvantage because it means that developers must either test all their features at once or they have to enter a queue and "book" the staging environment only for their feature.

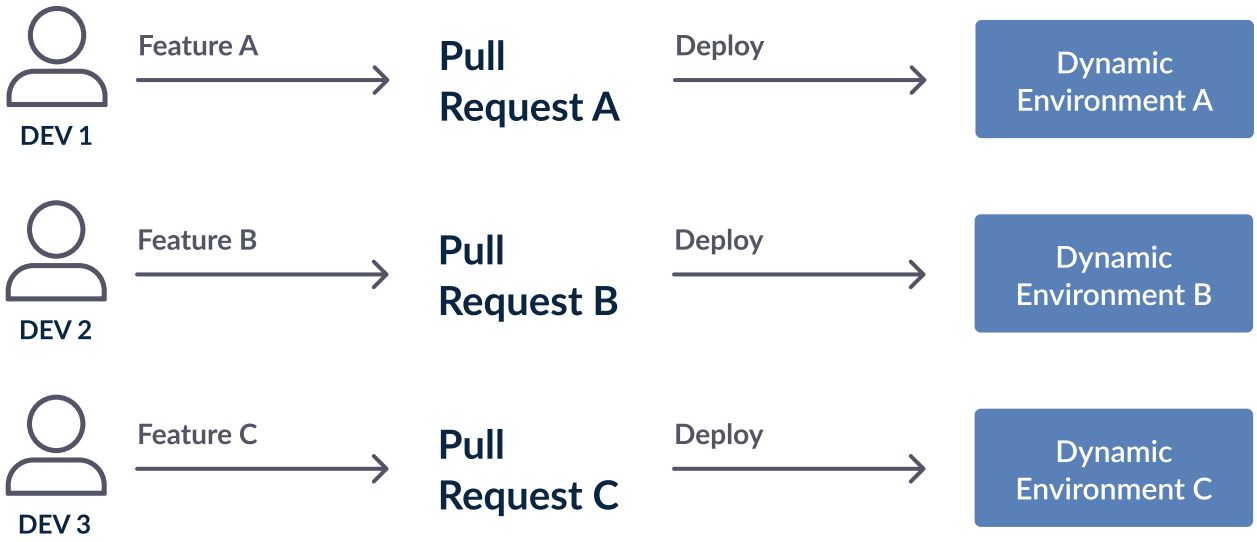


Don't

This forces a lot of organizations to create a fleet of test environments (e.g., QA1, QA2, QA3) so that multiple developers can test their features in parallel. This technique is still not ideal because:

- A maximum of N developers can test their feature (same as the number of environments) in parallel.
- Testing environments use resources all the time (even when they are not used)
- The static character of environments means that they have to be cleaned up and updated as well. This adds extra maintenance effort to the team responsible for test environments

With a cloud-based architecture, it is now much easier to create test environments on-demand. Instead of having a predefined number of static environments, you should modify your pipeline workflow so that each time a Pull Request is created by a developer, then a dedicated test environment is also created with the contents of that particular Pull Request.



Do Each feature is tested on its own

The advantages of dynamic test environments cannot be overstated:

1. Each developer can test in isolation without any conflicts with what other developers are doing
2. You pay for the resources of test environments only while you use them
3. Since the test environments are discarded at the end there is nothing to maintain or clean up

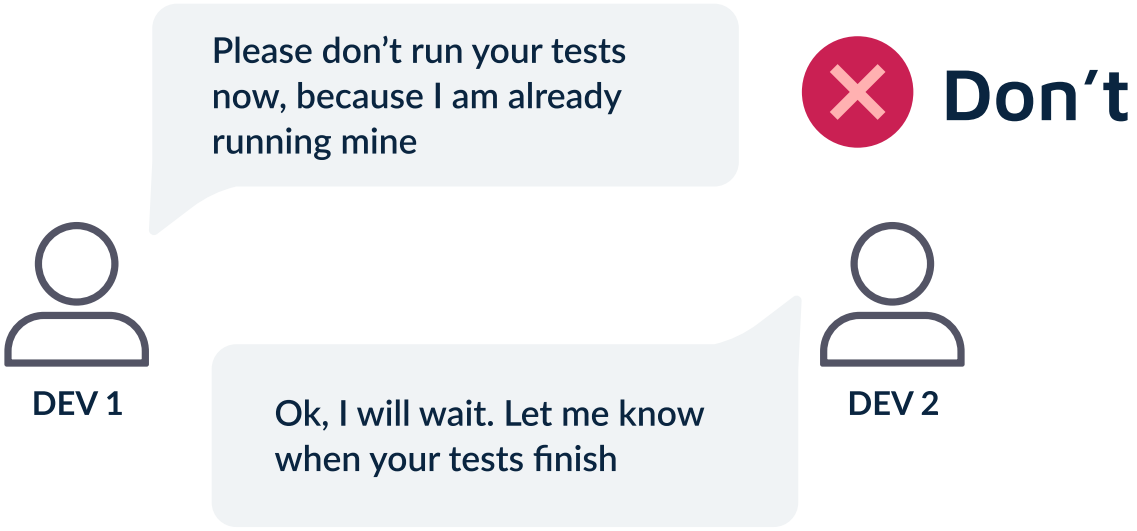
Dynamic test environments can shine for teams that have an irregular development schedule (e.g., having too many features in flight at the end of a sprint).

Best Practice 13

Running Test Suites Concurrently

This is a corollary of the previous best practice. If your development process has dynamic test environments, it means that different test suites can run at any point in time for any number of those environments even at the same time.

If your tests have special dependencies (e.g., they must be launched in a specific order, or they expect specific data before they can function) then having a dynamic number of test environments will further exacerbate the pre-run and post-run functions that you have for your tests.



The solution is to embrace best practice 10 and have each test prepare its state and clean up after itself. Tests that are read-only (i.e., don't have any side-effects) can run in parallel by definitions.

Tests that write/read information need to be self-sufficient. For example, if a test writes an entity in a database and then reads it back, you should not use a hardcoded primary key because that would mean that if two test suites with this test run at the same time, the second one will fail because of database constraints.

While most developers think that test parallelism is only a way to speed up your tests, in practice it is also a way to have correct tests without any uncontrolled side effects.

Best Practice 14

Security Scanning is Part of the Process

A lot of organizations still follow the traditional waterfall model of software development. And in most cases, the security analysis comes at the end. The software is produced and then a security scan (or even penetration test) is performed on the source code. The results are published and developers scramble to fix all the issues.

Putting security scanning at the end of a release is a lost cause. Some major architectural decisions affect how vulnerabilities are detected and knowing them in advance is a must not only for developers but also all project stakeholders.

Security is an ongoing process. An application should be checked for vulnerabilities at the same time as it is developed. This means that security scanning should be part of the pre-merge process (i.e as one of the checks of a Pull Request). Solving security issues in a finished software package is much harder than while it is in development.

Security scans should also have the appropriate depth. You need to check at the very least:

1. Your application source code
2. The container or underlying runtime where the application is running on
3. The computing node and the Operating System that will host the application

A lot of companies focus only on two (or even one) of these areas and forget the security works exactly like a chain (the weakest link is responsible for the overall security)

If you also want to be proactive with security, it is best to enforce it on the Pull Request level. Instead of simply scanning your source code and then reporting its vulnerabilities, it is better to prevent merges from happening in the first place if a certain security threshold is not passed.

Best Practice 15

Quality Scanning/Code Reviews Are Part of the Process

Similar to security scans, code scans should be part of the day-to-day developer operations. This includes:

- Static analysis of code for company-approved style/formatting
- Static analysis of code for security problems, hidden bugs
- Runtime analysis of code for errors and other issues

While there are existing tools that handle the analysis part, not all organizations execute those tools in an automated way. A very common pattern we see is enthusiastic software teams vowing to use these tools (e.g., Sonarqube) for the next software project, only to forget about them after some time or completely ignoring the warning and errors presented in the analysis reports.

In the same manner as security scans, code quality scanning should be part of the Pull Request process. Instead of simply reporting the final results to developers, you should enforce good quality practices by preventing merges if a certain amount of warning is present.

Best Practice 16

Database Updates Have Their Lifecycle

> As more and more companies adopt continuous delivery we see an alarming trend of treating databases as an external entity that exists outside of the delivery process. This could not be further from the truth.

Databases (and other supporting systems such as message queues, caches, service discovery solutions, etc.) should be handled like any other software project. This means:

- **Their configuration and contents should be stored in version control**
- **All associated scripts, maintenance actions, and upgrade/downgrade instructions should also be in version control**
- **Configuration changes should be approved like any other software change (passing from automated analysis, pull request review, security scanning, unit testing, etc.)**
- **Dedicated pipelines should be responsible for installing/upgrading/rolling back each new version of the database**

The last point is especially important. There are a lot of programming frameworks (e.g., rails migrations, Java Liquibase, ORM migrations) that allow the application itself to handle DB migrations. Usually the first time the application startup it can also upgrade the associate database to the correct schema. While convenient, this practice makes rollbacks very difficult and is best avoided.

Database migration should be handled like an isolated software upgrade. You should have automated pipelines that deal only with the database, and the application pipelines should not touch the database in any way. This will give you the maximum flexibility to handle database upgrades and rollbacks by controlling exactly when and how a database upgrade takes place.

Best Practice 17

Database Updates are Automated

> Several organizations have stellar pipelines for the application code, but pay very little attention to automation for database updates. Handling databases should be given the same importance (if not more) as with the application itself.

This means that you should similarly automate databases to application code:

- Store database changesets in source control
- Create pipelines that automatically update your database when a new changeset is created
- Have dynamic temporary environments for databases where changesets are reviewed before being merged to mainly
- Have code reviews and other quality checks on database changesets
- Have a strategy for doing rollbacks after a failed database upgrade

It also helps if you automate the transformation of production data to test data that can be used in your test environments for your application code. In most cases, it is inefficient (or even impossible due to security constraints) to keep a copy of all production data in test environments. It is better to have a small subset of data that is anonymized/simplified so that it can be handled more efficiently.

Best Practice 18

Database Updates Are Forward and Backward Compatible

> Application rollbacks are well understood and we are now at the point where we have dedicated tools that perform rollbacks after a failed application deployment. And with progressively delivery techniques such as canaries and blue/green deployments, we can minimize the downtime even further.

Progressive delivery techniques do not work on databases (because of the inherent state), but we can plan the database upgrades and adopt evolutionary database design principles.

By following an evolutionary design you can make all your database changesets backward and forwards compatible allowing you to rollback application and database changes at any time without any ill effects.

As an example, if you want to rename a column, instead of simply creating a changeset the renames the column and performing a single database upgrade, you instead follow a schedule of gradual updates as below:

1. Database changeset that only adds a new column with the new name (and copies existing data from the old column). The application code is still writing/reading from the old column
2. Application upgrade where the application code now writes to both columns but reads from the new column
3. Application upgrade where the application code writes/reads only to the new column
4. Database upgrade that removes the old column

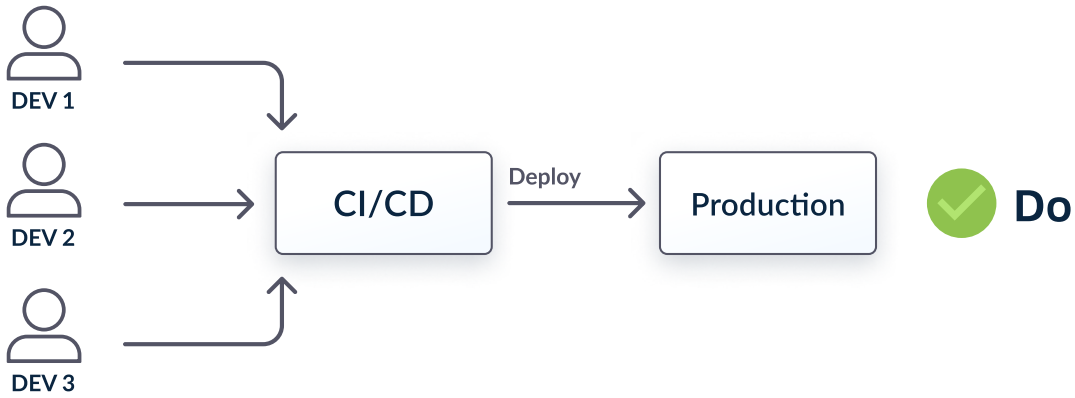
The process needs a well-disciplined team as it makes each database change span over several deployments. But the advantages of this process cannot be overstated. At any stage in this process, you can go back to the previous version without losing data and without the need for downtime.

For the full list of techniques see the database refactoring website.

Best Practice 19

Deployments Happen Via a Single Path (CI/CD Server)

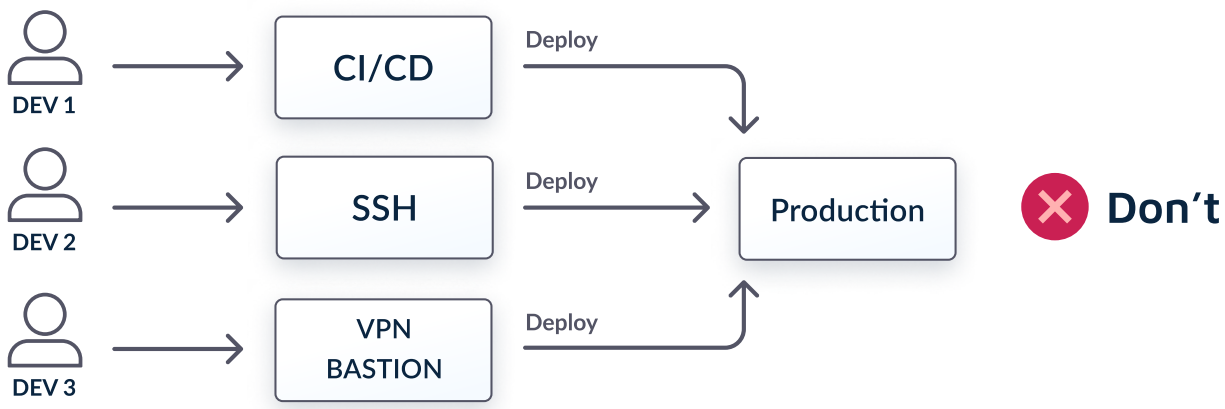
Continuing the theme of immutable artifacts and deployments that send to production what was deployed, we must also make sure the pipelines themselves are the only single path to production.



The main way to use CI/CD pipelines as intended is to make sure that the CI/CD platform is the **only** application that can deploy to production. This practice guarantees that production environments are running what they are expected to be running (i.e., the last artifact that was deployed).

Unfortunately, several organizations either allow developers to deploy directly from their workstations, or even to "inject" their artifacts in a pipeline at various stages.

This is a very dangerous practice as it breaks the traceability and monitoring offered by a proper CI/CD platform. It allows developers to deploy to production features that might not be committed in source control in the first place. A lot of failed deployments stem from a missing file that was present on a developer workstation and not in source control.



In summary, there is only a single critical path for deployments, and this path is strictly handled by the CI/CD platform. Deploying production code from developer workstations should be prohibited at the network/access/hardware level.

Best Practice 20

Deployments Happen Gradually in Stages

> We already talked about database deployments in best practice 18 and how each database upgrade should be forwards and backward compatible. This pattern goes hand-in-hand with progressive delivery patterns on the application side.

Traditional deployments follow an all-or-nothing approach where all application instances move forward to the next version of the software. This is a very simple deployment approach but makes rollbacks a challenging process.

You should instead look at:

1. Blue/Green deployments that deploy a whole new set of instances of the new version, but still keep the old one for easy rollbacks
2. Canary releases where only a subset of the application instances move to the new version. Most users are still routed to the previous version

If you couple these techniques with gradual database deployments, you can minimize the amount of downtime involved when a new deployment happens. Rollbacks also become a trivial process as in both cases you simply change your load balancer/service mesh to the previous configuration and all users are routed back to the original version of the application.

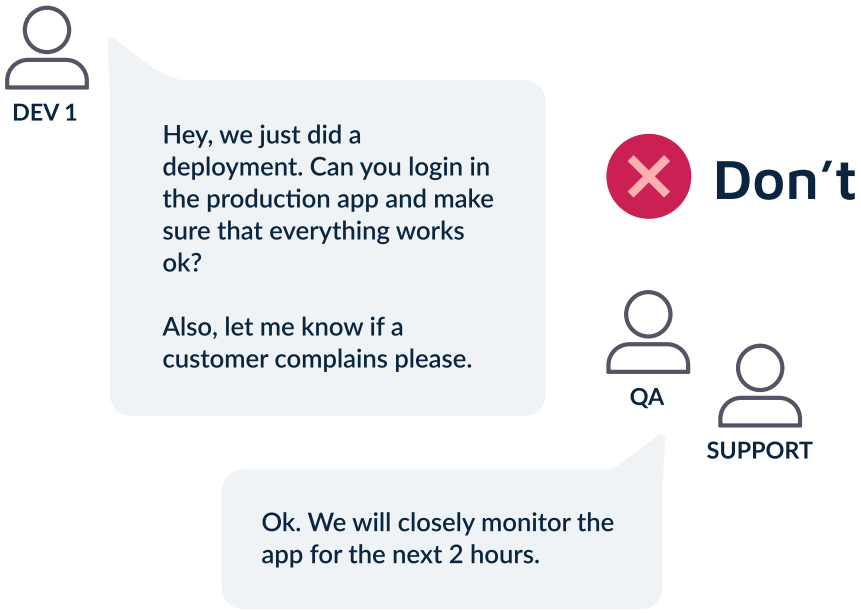
Make sure to also look at involving your metrics (see best practices 21 and 22) in the deployment process for fully automated rollbacks.

Best Practice 21

Metrics and Logs Can Detect a Bad Deployment

> Having a pipeline that deploys your application (even when you use progressive delivery) is not enough if you want to know what is the real result of the deployment. Deployments that look "successful" at first glance, but soon prove to introduce regressions is a very common occurrence in large software projects.

A lot of development teams simply perform a visual check/smoke test after a deployment has finished and call it a day if everything "looks" good. But this practice is not enough and can quickly lead to the introduction of subtle bugs or performance issues.



The correct approach is the adoption of application (and infrastructure) metrics. This includes:

- Detailed logs for application events
- Metrics that count and monitor key features of the application
- Tracing information that can provide an in-depth understanding of what a single request is doing

Once these metrics are in place, the effects of deployment should be judged according to a before/after comparison of these metrics. This means that metrics should not be simply a debugging mechanism (post-incident), but should act instead as an early warning measure against failed deployments.

Choosing what events to monitor and where to place logs is a complex process. For large applications, it is best to follow a gradual redefinition of key metrics according to past deployments. The suggested workflow is the following:

1. Place logs and metrics on events that you guess will show a failed deployment
2. Perform several deployments and see if your metrics can detect the failed ones
3. If you see a failed deployment that wasn't detected in your metrics, it means that they are not enough. Fine-tune your metrics accordingly so that the next time a deployment fails in the same manner you actually know it in advance

Too many times, development teams focus on "vanity" metrics, i.e., metrics that look good on paper but say nothing about a failed deployment.

Best Practice 22

Automatic Rollbacks Are in Place

This is a continuation of the previous best practice. If you already have good metrics in place (that can verify the success of a deployment) you can take them to the next level by having automated rollbacks that depend on them.

A lot of organizations have great metrics in place, but only manually use them:

1. A developer looks at some key metrics before deployment
2. Deployment is triggered
3. The developer looks at the metrics in an ad-hoc manner to see what happened with the deployment

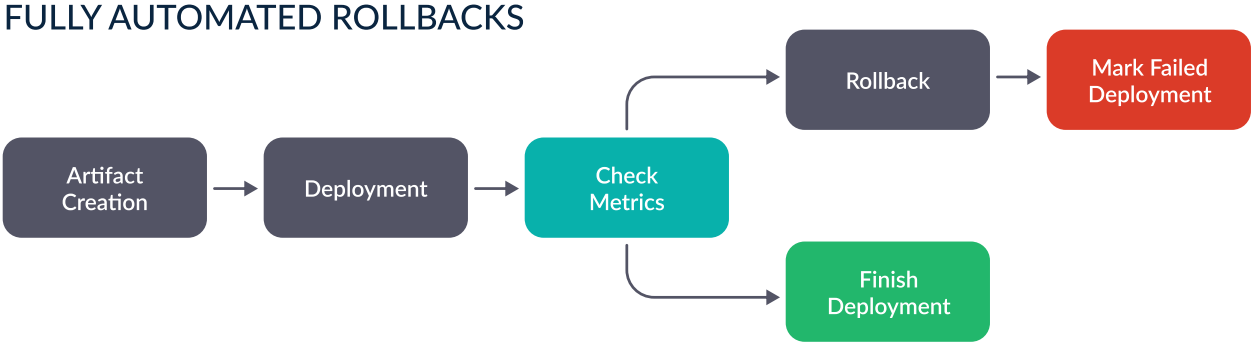
While this technique is very popular, it is far from effective. Depending on the complexity of the application, the time spent watching metrics can be 1-2 hours so that the effects of the deployment have time to become visible.

It is not uncommon for deployments to be marked as "failed" after 6-24 hours either because nobody paid attention to the correct metrics or because people simply disregarded warnings and errors thinking that was not a result of the deployment.

Several organizations are also forced to only deploy during working hours because only at that time there are enough human eyes to look at metrics.

Metrics should become part of the deployment process. The deployment pipeline should automatically consult metrics after a deployment happens and compare them against a known threshold or their previous state. And then in a fully automated manner, the deployment should either be marked as finished or even rolled back.

FULLY AUTOMATED ROLLBACKS



This is the holy grail of deployments as it completely removes the human factor out of the equation and is a step towards Continuous Deployment (instead of Continuous Delivery). With this approach:

1. You can perform deployments at any point in time knowing that metrics will be examined with the same attention even if the time is 3 am
2. You can catch early regressions with pinpoint accuracy
3. Rollbacks (usually a stressful action) are now handled by the deployment platform giving easier access to the deployment process by non-technical people

The result is that a developer can deploy at 5 pm on Friday and immediately go home. Either the change will be approved (and it will be still there on Monday) or it will be rolled back automatically without any ill effects (and without any downtime if you also follow best practice 20 for progressive delivery).

We recommend following GitOps principles with every deployment being clearly initiated by a git commit and the state fully-represented to be rolled back via a git commit.

Best Practice 23

Staging Matches Production

> We explained in best practice 12 that you should employ dynamic environments for testing individual features for developers. This gives you the confidence that each feature is correct on its own before you deploy it in production.

It is also customary to have a single staging environment (a.k.a. pre-production) that acts as the last gateway before production. This particular environment should be as close to production as possible so that any configuration errors can and mismatches can be quickly discovered before pushing the application deployment to the real production environment.

Unfortunately, most organizations treat the staging environment in a different way than the production one. Having a staging environment that is separate from production is a cumbersome practice as it means that you have to manually maintain it and make sure that it also gets any updates that reach production (not only in application terms but also any configuration changes).

Two more effective ways of using a staging environment are the following:

1. **Create a staging environment on-demand each time you deploy by cloning the production environment**
2. **Use as staging a special part of production (sometimes called shadow production)**

The first approach is great for small/medium applications and involves cloning the production environment right before a deployment happens in a similar (but possibly smaller) configuration. This means that you can also get a subset of the database and a lower number of replicas/instances that serve traffic. The important point here is that this staging environment only exists during a release. You create it just before a release and destroy it once a release has been marked as "successful".

The main benefit of course is that cloning your production right before deployment guarantees that you have the same configuration between staging and production. Also, there is nothing to maintain or keep up-to-date because you always discard the staging environment once the deployment has finished.

This approach however is not realistic for large applications with many microservices or large external resources (e.g., databases and message queues). In those cases, it is much easier to use staging as a part of the production. The important point here is that the segment of production that you use does NOT get any user traffic, so in case of a failed deployment, your users will not be affected. The advantage again is that since this is part of the production you have the same guarantee that the configuration is the most recent one and what you are testing will behave in the same way as "real" production.

Applying These Best Practices to Your Organization

We hope that now you have some ideas on how to improve your CI/CD process. Remember however that it is better to take gradual steps and not try to change everything at once.

Consult the first section of this guide where we talked about priorities. Focus first on the best practices that are marked as "critical" and as soon as you have conquered them move to those with "high" importance.

We believe that if you adopt the majority of practices that we have described in this guide, your development teams will be able to focus on shipping features instead of dealing with failed deployments and missing configuration issues.