# Ten Commandments of GitOps Applied to Continuous Delivery

This lesson lists the ten commandments of GitOps principles and the reasoning behind each one.

**We'll cover the following**

Instead of listing someone else's rules, we'll try to deduce them ourselves. So far, we have only one, and that is the most important rule that is likely going to define the rest of the brainstorming and discussion.

## *1. Git is the only source of truth*

> The one rule to rule them all is that **Git is the only source of truth**.

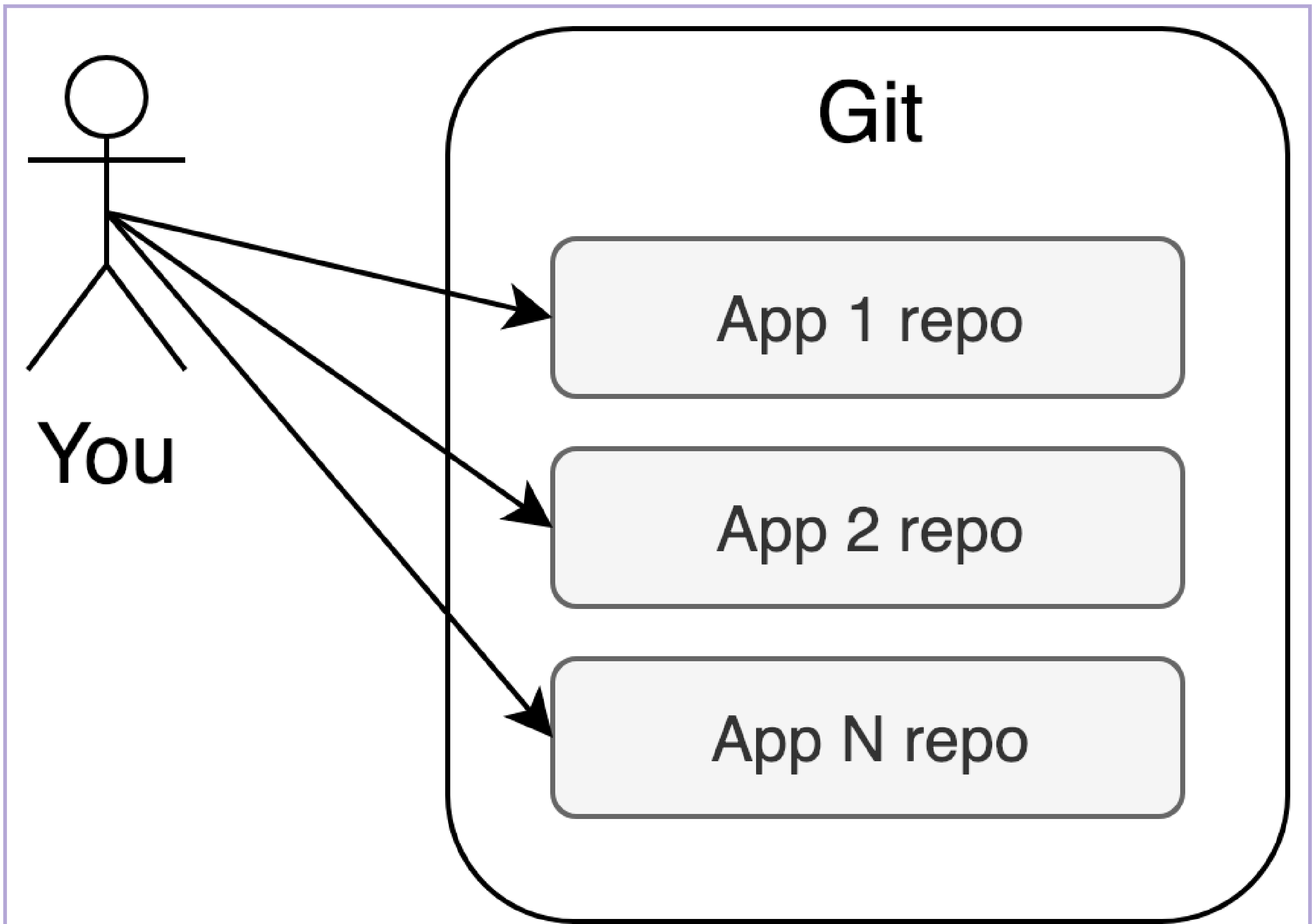
It is the first and most important commandment. All application-specific code in its raw format must be stored in Git. By code, I mean not only the code of your application, but also the tests, configuration, and everything else that is specific to that app or the system in general.

I intentionally said that it should be in **raw format** because there is no benefit of storing binaries in Git. That's not what it's designed for.

The real question is **why we want those things?** For one, good development practices should be followed. Even though we might disagree on which practices are good, and which aren't, they are all levitating around Git.

- If you're doing code reviews, you're doing it through Git.
- If you need to see the change history of a file, you'll see it through, Git.

If you find a developer doubting whether the code should be in Git (or some other code repository), please make sure that they're isolated from the rest of the world because you just found a specimen of endangered species. There are only a few left, and they are bound to be extinct.



Application-specific Repositories

## 2. *Everything must be tracked, actions should be reproducible, and idempotent*

While there is no doubt amongst developers about where to store the files they create, that's not necessarily true for other types of experts. I see *testers*, *operators*, and people in other roles that are still not convinced that's the way to go and whether absolutely everything should be documented and stored in Git.

As an example, I still meet operators who run **ad-hoc commands** on their servers. As we all know, ad-hoc commands executed inside servers are not reliably reproducible, they are often not documented, and the result of their execution is often not idempotent.

> So, let's create a second rule: **everything must be tracked, every action must be reproducible, and everything must be idempotent**.

- If you only run a command instead of creating a script, your activities are not documented.
- If you did not store it in Git, others will not be able to reproduce your actions.

Finally, that script must be able to produce the same result no matter how many times we execute it. Today, the easiest way to accomplish that is through **declarative syntax**. More often than note, that would be `YAML` or `JSON` files that describe the desired outcome, instead of imperative scripts.

Let's take installation as an example. If it's imperative (install something), it will fail if that something is already installed; it won't be idempotent.

Every change must be recorded (tracked). The most reliable and easiest way to accomplish that is by allowing people only to push changes to Git. **That and only that is the acceptable human action!**

What that means:

- If we want our application to have a new feature, we need to write code and push it to Git.
- If we want it to be tested, we write tests and push them to Git, preferably at

the same time as the code of the application.

- If we need to change a configuration, we update a file and push it to Git.
- If we need to install or upgrade OS, we make changes to files of whichever tool we're using to manage our infrastructure, and we push them to Git.

It all boils down to the fact that you should *push it to Git*. What is more interesting is what we should **NOT** do.

> **You are not allowed to add a feature of an application by changing the code directly inside the production servers.**

It doesn't matter how big or small the change is, it cannot be done by you, because you cannot provide a guarantee that the change will be *documented*, *reproducible*, and *tracked*. Machines are much more reliable when performing actions inside your production systems. You are their overlord, you're not one of them. Your job is to express the desired state, not to change the system to comply with it.

## 3. Communication between processes must be asynchronous

The real challenge is to decide:

- **How will that communication be performed?**
- **How do we express our desires in a way that machines can execute actions that will result in convergence of the actual state into the desired one?**

### An analogy

We can think of us as an *aristocracy* and the machines as *servants*. The good thing about an aristocracy is that there is no need to do much work. As a matter of fact, not doing any work is the main benefit of being a king, a queen, or an heir to the throne. Who would want to be a king if that means working as a car mechanic? No girl dreams of becoming a princess if that would mean working in a supermarket. Therefore, if being an aristocrat means not doing much work, we still need someone else to do it for us. Otherwise, how will our desires become reality? That's why aristocracy needs servants. Their job is to do their bidding.

Given that human servitude is forbidden in most of the world, we need to look for servants outside the human race. Today, servants are bytes that are converted into processes running inside machines. We (humans) are the overlords and machines are our slaves. However, since it is not legal to have slaves, nor is it politically correct to call them that, we will refer to them as agents. So, we (humans) are overlords of agents (machines).

If we are true overlords that trust the machines to do our biddings, there is no need for that communication to be synchronous. When we trust someone to do our bidding, we do not need to wait until our desires are fulfilled.

Let's imagine that you are in a restaurant and you tell a waiter *"I'd like a burger with cheese and fries."*

**What do you do next?**

- Do you get up, go outside the restaurant, purchase some land, and build a farm?
- Are you going to grow animals and potatoes?
- Will you wait until they are mature enough and take them back to the restaurant. Will you start frying potatoes and meat?

To be clear, it's completely OK if you like owning land and if you are a farmer. There's nothing wrong with liking to cook. But, if you went to a restaurant, you did that precisely because you did not want to do those things. The idea behind an expression like "I'd like a burger with cheese and fries" is that we want to do something else, like chatting with friends and eating food. We know that a cook will prepare the meal and that our job is not to grow crops, to feed animals, or to cook. We want to be able to do other things before eating. We are like aristocracy and, in that context, farmers, cooks, and everyone else involved in the burger industry are our agents.

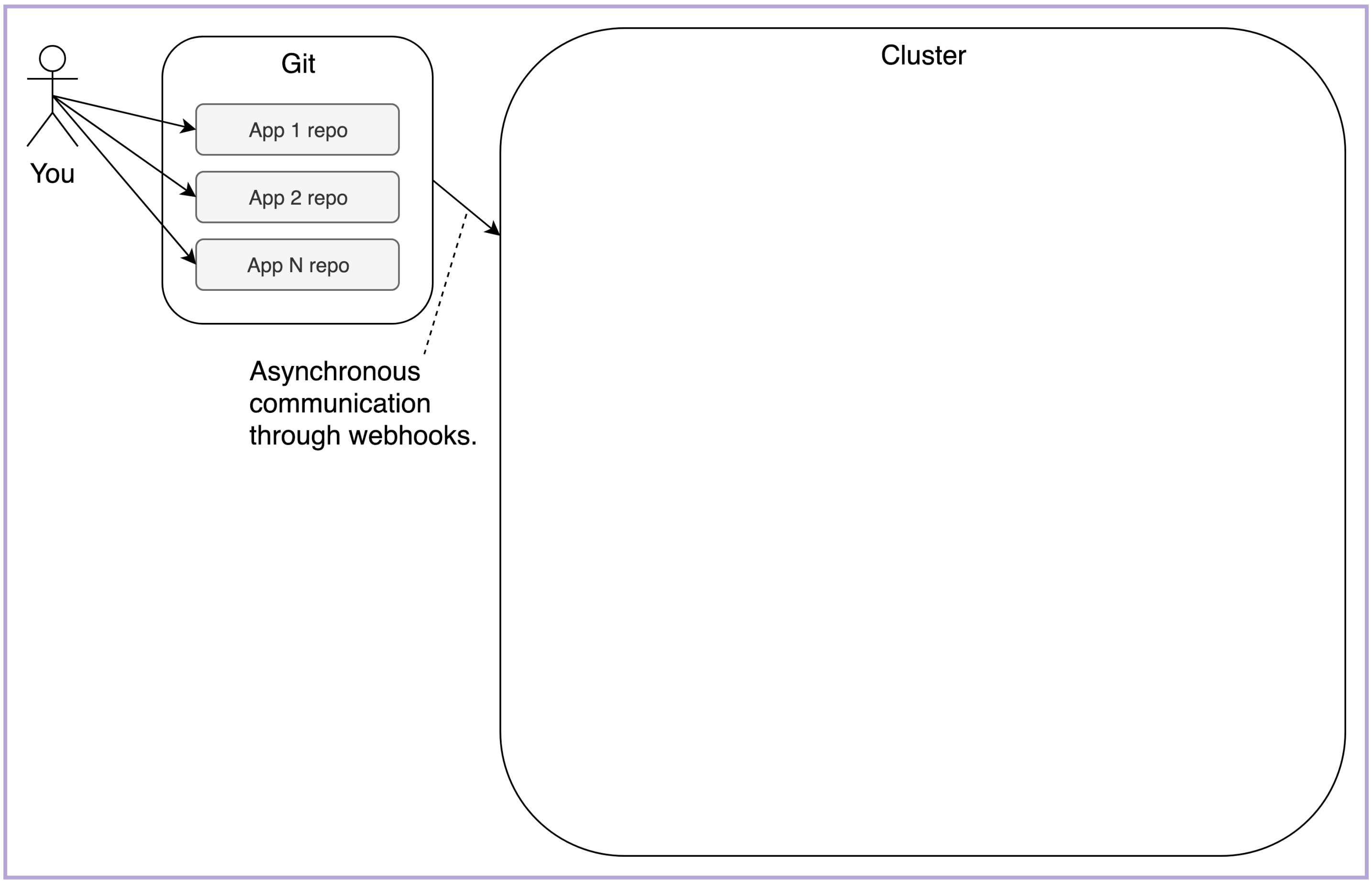
So, when we request something, all we need is an **acknowledegment**. If the response to *"I'd like a burger with cheese and fries"* is *"consider it done"*. We got the *ack* we need, and we can do other things while the process of creating the burger is executing. Farming, cooking, and eating can be **parallel processes**. For them to operate concurrently, the communication must be asynchronous. We request something, we receive an acknowledgment, and we move back to whatever we were doing.

> So, the third rule is: **communication between processes must be asynchronous** if operations are to be executed in parallel.

## Webhooks

If we already agreed that the only source of truth is Git (that's where all the information is), then the logical choice for asynchronous communication is **webhooks**. Whenever we push a change to any of the repositories, a webhook can be triggered to the system. As a result, the new desire expressed through code (or config files), can be propagated to the system which, in turn, should delegate tasks to different processes.

We have yet to design such a system. For now, think of it as one or more entities inside our cluster. If we apply the principle of having everything defined as code and stored in Git, there is no reason why those webhooks wouldn't be the only operational entry point to the system. There is no excuse to allow SSH access to anyone. If you define everything in Git, what additional value can you add if you're inside one of the nodes of the cluster?



Asynchronous communication through webhooks from Git to the system

Asynchronous communication through webhooks from Git to the system

Depending on the desired state, the actor that should converge the system can be **Kubernetes**, **Helm**, **Istio**, a cloud, or an on-prem provider, or one of many other tools. More often than not, multiple processes need to perform some actions in parallel. That would pose a problem if we'd rely only on webhooks. By their nature, they are not good at deciding who should do what. If we draw another parallel between aristocracy and servants (agents), we would quickly see how it could be inconvenient for royalty to interact directly with their staff. Having one servant is not the same as having tens or hundreds. For that, royalty came to the idea to employ a butler. He is the chief manservant of a house (or a court). His job is to organize servants so that our desires are always fulfilled. He knows when you like to have lunch when you'd want to have a cup of tea or a glass of Gin&Tonic, and he's always there when you need something unpredictable.

Given that our webhooks (requests for a change) are dumb and incapable of transmitting our desires to the individual components of the system, we need something equivalent to a butler. We need someone (or something) to make decisions and make sure that each desire is converted into a set of actions and assigned to different actors (processes). That butler is a component in the Jenkins X bundle. Which one it is, depends on our needs or, to be more precise, whether the butler should be static or serverless. Jenkins X supports both and makes those technical details transparent.

Every change to Git triggers a webhook request to a component in the Jenkins X bundle. It, in turn, responds only with an acknowledgment (ACK) letting Git know that it received a request. Think of *ack* as a subtle nod followed with the butler exiting the room and starting the process right away. He might call a cook, a person in charge of cleaning, or even an external service if your desire cannot be fulfilled with the internal staff. In our case, the staff (servants, slaves) are different tools and processes running inside the cluster. Just as a court has servants with different skillsets, our cluster has them as well. The question is how to organize that staff so that they are as efficient as possible. After all, even aristocracy cannot have unlimited manpower at their disposal.

## 4. *Processes should run for as long as needed, but not longer*

Let's go big and declare ourselves the royalty of a wealthy country like the United

Kingdom (UK). We'd live in Buckingham Palace. It's an impressive place with 775

rooms. Of those, 188 are staff rooms. We might draw the conclusion that the staff counts 188 as well, but the real number is much bigger. Some people live and work there, while others come only to perform their services. The number of servants (staff, employees) varies. You can say that it is elastic. Whether people sleep in Buckingham Palace or somewhere else depends on what they do. Cleaning, for example, is happening all the time.

Given that royalty might be a bit spoiled, they need people to be available almost instantly. "Look at that. I just broke a glass, and a minute later, a new one materialized next to me, and the pieces of the broken glass disappeared." Since that is Buckingham Palace and not Hogwarts School of Witchcraft and Wizardry, the new glass did not materialize by magic, but by a butler that called a servant specialized in fixing the mess princesses and princes keep doing over and over again. Sometimes a single person can fix the mess (broken glass), and at other times a whole team is required (a royal ball turned into alcohol-induced shenanigans).

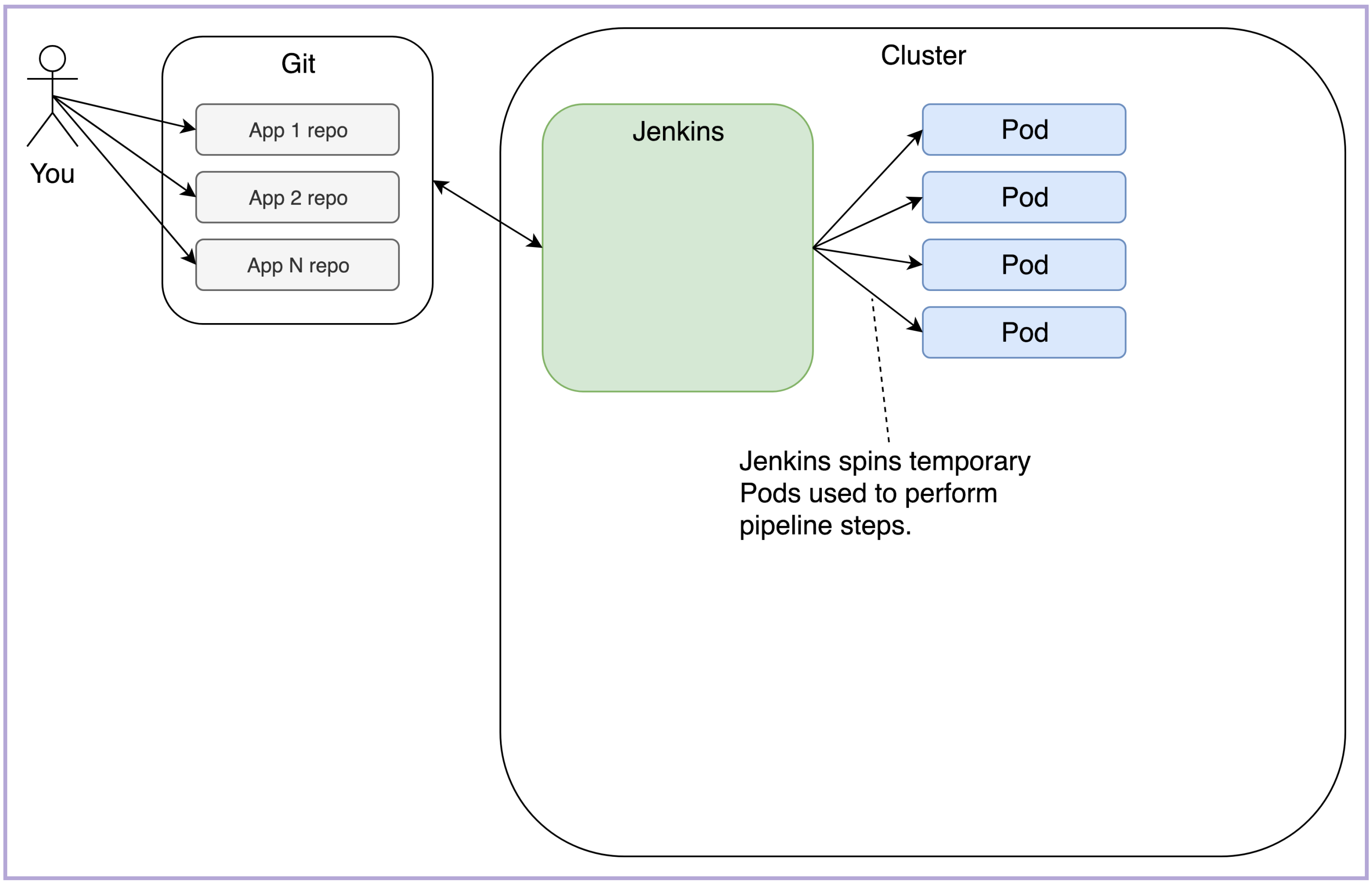
Given that the needs can vary greatly, servants are often idle. That's why they have their own rooms. Most are called when needed, so only a fraction is doing something at any given moment. They need to be available at any time, but they also need to rest when their services are not required. They are like Schrodinger's cats that are both alive and dead. Therefore, when there is no work, a servant is idle (but still alive). In our case, making something dead or alive on a moment's notice is not an issue since our agents are not humans, but bytes converted into processes. That's what containers give us, and that's what serverless is aiming for.

By being able to create as many processes as needed, and by not having processes that we do not use, we can make our systems scalable, fault-tolerant, and efficient.

> So, the next rule we'll define is that **processes should run for as long as needed, but not longer**.

That can be containers that scale down from something to zero and back again. You can call it serverless. The names do not matter that much. What does matter is that everything idle must be killed, and all those alive should have all the resources they need. That way, our butler (Jenkins, prow, something else) can organize tasks as efficiently as possible. He has an unlimited number of servants

(agents, Pods) at his disposal, and they are doing something only until the task is done. Today, containers (in the form of Pods) allow us just that. We can start any process we want, it will run only while it's doing something useful (while it's alive), and we can have as many of them as we need if our infrastructure is scalable. A typical set of tasks our butler might assign can be building an application through Go (or whichever language we prefer), packaging it as a container image and as a Helm chart, running a set of tests, and (maybe) deploying the application to the staging environment.



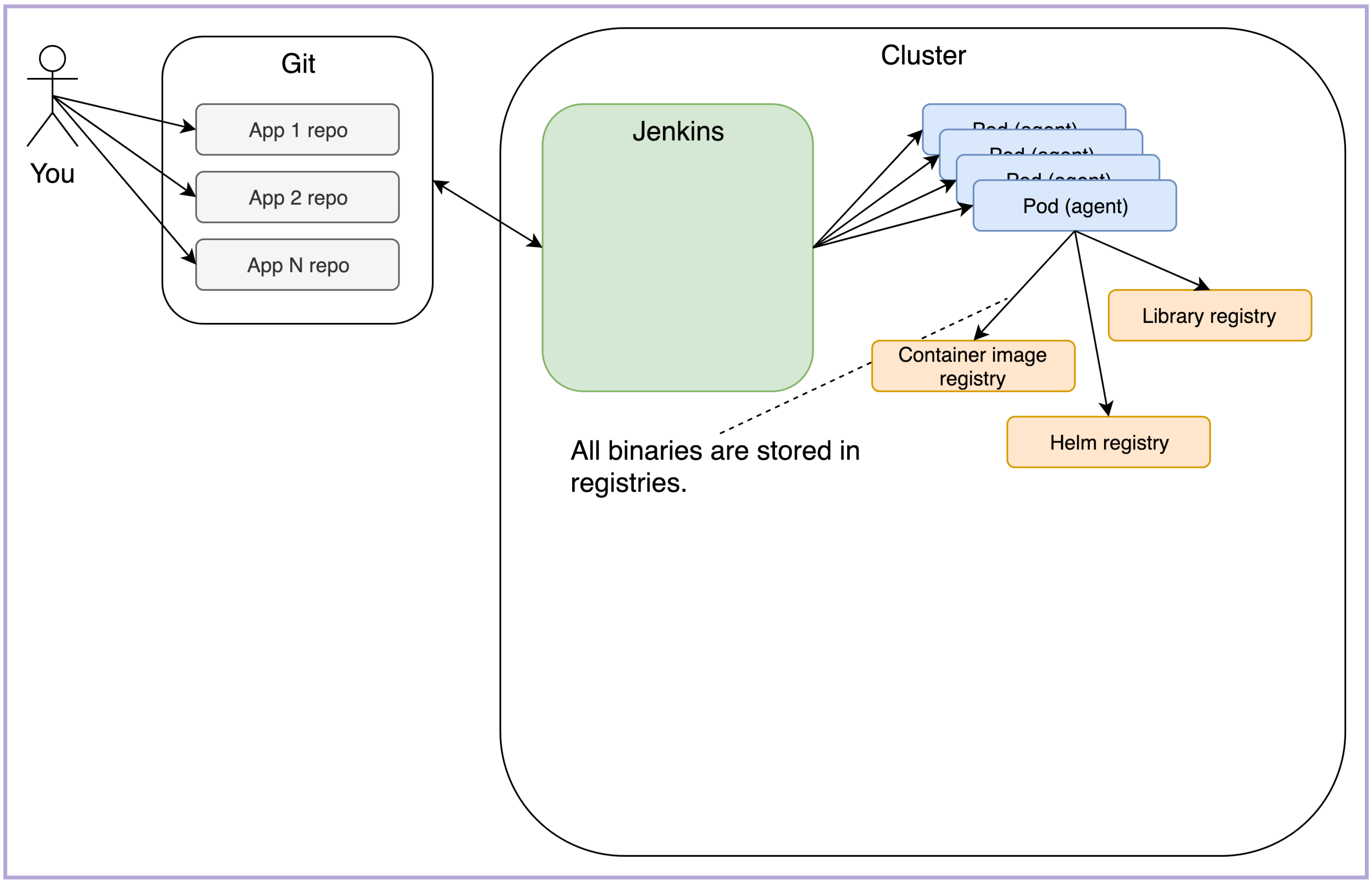
Jenkins spinning temporary Pods used to perform pipeline steps

## 5. *All binaries must be stored in registries*

In most cases, our pipelines will generate some binaries. Those can be libraries, container images, **Helm** packages, and many others. Some of those might be temporary and needed only for the duration of a build. A good example could be a binary of an application. We need it to generate a container image. Afterward, we can just as well remove it since that image is all we need to deploy the application. Since we're running the steps inside a container, there is no need to remove anything, because the Pods and the containers they contain are removed once builds are finished. However, not all binaries are temporary. We do need to store

container images somewhere. Otherwise, we won't be able to run them inside the cluster. The same is true for **Helm** charts, libraries (those used as dependencies), and many others. For that, we have different applications like **Docker** registry (container images), **ChartMuseum** (Helm charts), **Nexus** (libraries), and so on. It's important to understand, that we only store binaries in those registries only and not code, configurations, and other raw-text files. Those must go to Git because that's where we track changes, that's where we do code reviews, and that's where we expect them to be. Now, in some cases, it makes sense to keep raw files in registries as well. They might be an easier way of distributing them to some groups. Nevertheless, Git is the single source of truth, and it must be treated as such.

> This leads us to yet another rule that states that **all binaries must be stored in registries** and that raw files can be there only if that facilitates distribution while understanding that those are not the sources of truth.



All binaries are stored in registries

## 6. Information about all the releases must be stored

## *6. Information about all the releases must be stored in environment-specific repositories or branches*

We already established that all code and configurations (excluding secrets) must be stored in Git and that Git is the only entity that should trigger pipelines. We also argued that any change must be recorded. A typical example is a new release. It is way too common to deploy a new release, but not to store that information in Git. Tags do not count because we cannot recreate a whole environment from them. We'd need to go from tag to tag to do that.

The same is true for release notes. While they are very useful and we should create them, we cannot diff them, nor can we use them to recreate an environment. What we need is a place that defines a full environment. It also needs to allow us to:

- track changes,
- review them,
- approve them,
- and so on.

In other words, what we need from an environment definition is not conceptually different from what we expect from an application. We need to store it in a Git repository. There is very little doubt about that. What is less clear is which repository should have the information about an environment.

We should be able to respond to the following questions:

- *"Which release of an application is running in production?"*
- *"What is production?"*
- *"What are the releases of all the applications running there?"*

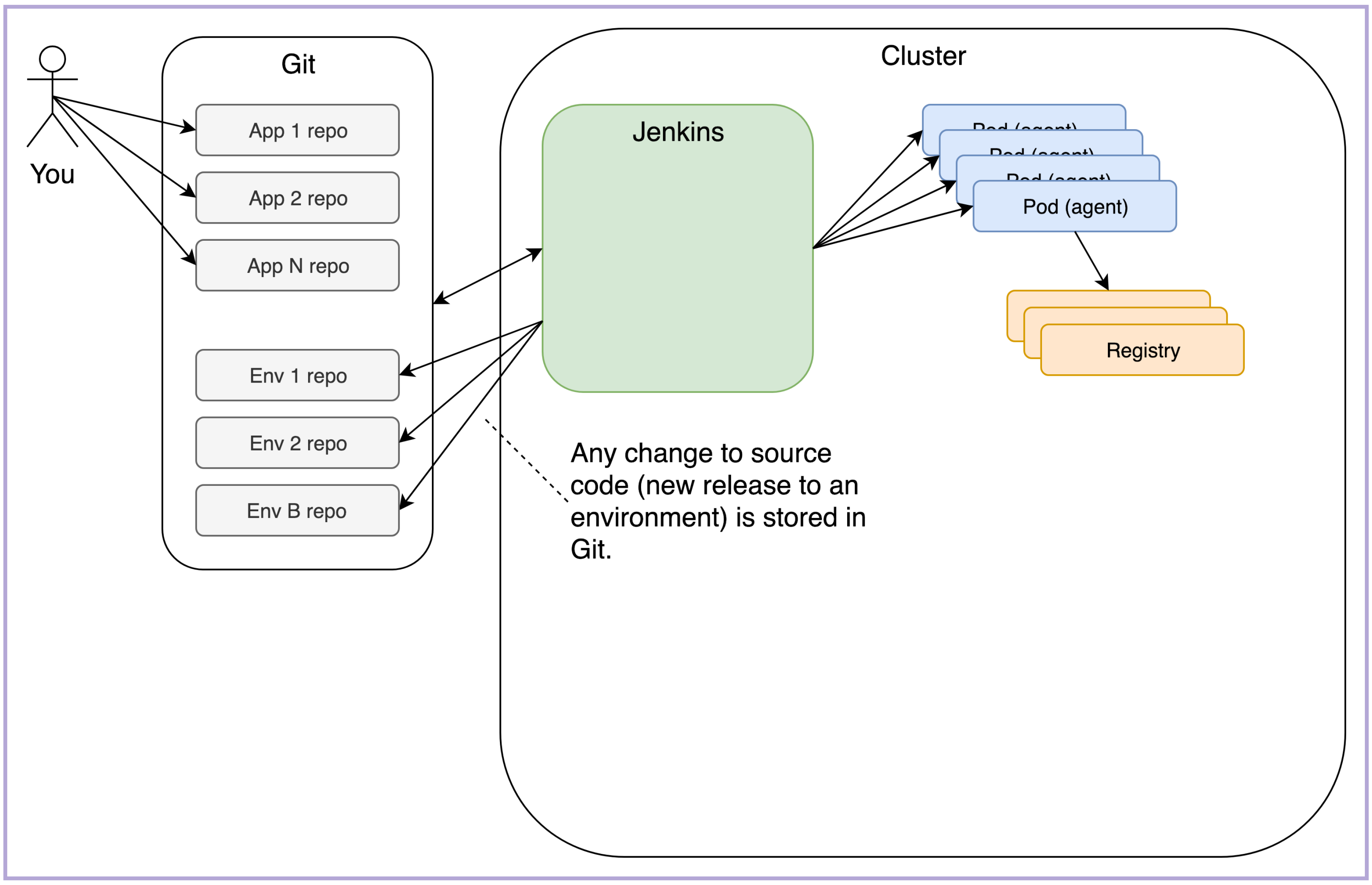
If we store information about a release in the repository of the application we just deployed, we would be able to answer only the first question. We would know which release of our app is in an environment. What we couldn't easily answer is the same question but in reference to the whole environment, not just to one application. Or, to be more precise, we could not do that easily. We'd need to go from one repository to another.

Another important thing we need to keep in mind is the ability to recreate an environment (e.g., staging or production). That can't be done easily if the information about the releases is spread across many repositories.

All those requirements lead us to only one solution. Our environments need to be in separate repositories or, at least, in different branches within the same repository. Given that we agreed that information is first pushed in Git which, in turn, triggers processes that do something with it, we cannot deploy a release to an environment directly from a build of an application. Such a build would need to push a change to the repository dedicated to an environment. In turn, such a push would trigger a webhook that would result in yet another build of a pipeline.

This gives us the following rule:

> **Information about all the releases must be stored in environment-specific repositories or branches**



Any change to source code (new release to an environment

## 7. Everything must follow the same coding practices

When we write new code, we tend not to push directly to the master branch, but to create pull requests. Even if we don't need approval from others (e.g., code review) and plan to push it to the master branch directly, having a pull request is still very useful. It provides an easy way to track changes and intentions behind them. Now, that does not mean that I am against pushing directly to master, quite the contrary. But, such practice requires discipline and technical and process mastery that is still out of reach to many. So, I will suppose that you do work with pull requests.

If we are supposed to create pull requests of things we want to push to master branches of our applications, there is no reason why we shouldn't treat environments the same. What that means is not only that our application builds should push releases to environment-specific branches, but that they should do that by making pull requests.

Taking all that into account the next rule should state that:

> **Everything must follow the same coding practices** (environments included).

## 8. *All deployments must be idempotent* #

The correct way to execute the flow while adhering to the rules we mentioned so far would be to have as many pipelines as there are applications, plus a pipeline for deployment to each of the environments. A push to the application repository should initiate a pipeline that builds, tests, and packages the application. It should end by pushing a change to the repository that defines a whole environment (e.g., staging, production, etc.). In turn, that should initiate a different pipeline that (re)deploys the entire environment. That way, we always have a single source of truth. Nothing is done without pushing code to a code repository.

Always deploying the whole environment would not work without idempotency. Fortunately, Kubernetes, as well as Helm, already provide that. Even though we always deploy all the applications and the releases that constitute an environment, only the pieces that changed will be updated.

> That brings us to a new rule: **all deployments must be idempotent**.

## 9. *Git webhooks are the only ones allowed to initiate*

## a change that will be applied to the system #

Having everything defined in code and stored in Git is not enough. We need those definitions and that code to be used reliably. Reproducibility is one of the key features we're looking for. Unfortunately, we (humans) are not good at performing reproducible actions. We make mistakes, and we are incapable of doing exactly the same thing twice. We are not reliable; machines are. If conditions do not change, a script will do exactly the same thing every time we run it. While scripts provide repetition, the declarative approach gives us idempotency.

But why do we want to use a declarative syntax to describe our systems? The main reason is in idempotency provided through our expression of a desire, instead of imperative statements. If we have a script that, for example, creates ten servers, we might end up with fifteen if there are already five nodes running. On the other hand, if we declaratively express that there should be ten servers, we can have a system that will check how many do we already have, and increase or decrease the number to comply with our desire.

Where we do excel is creativity. We are good at writing scripts and configurations, but not at running them. Ideally, every single action performed anywhere inside our systems should be executed by a machine, not by us. We accomplish that by storing the code in a repository and letting all the actions execute as a result of a webhook firing an event on every push of a change. Given that we already agreed that Git is the only source of truth and that we need to push a change to see it reflected in the system.

> We can define the rule that: **git webhooks are the only ones allowed to initiate a change that will be applied to the system**.

That might result in many changes in the way we operate. It means that no one is allowed to execute a script from a laptop that will, for example, increase the number of nodes. There is no need to have SSH access to the servers if we are not allowed to do anything without pushing something to Git first.

Similarly, there should be no need to even have admin permissions to access Kubernetes API through `kubectl`. Those privileges should be delegated to machines, and our (human) job should be to create or update code, configurations, and definitions, to push the changes to Git, and to let the machines do the rest.

and definitions, to push the changes to Git, and to let the machines do the rest.
That is hard to do, and we might require considerable investment to accomplish

that. But, even if it takes a while to get there, we should still strive for such a process and delegation of tasks. Our designs and our processes should be created with that goal in mind, no matter whether we can accomplish them today, tomorrow, or next year.

## 10. *All the tools must be able to speak with each other through APIs* #

Finally, there is one more thing we're missing. Automation relies on APIs and CLIs (they are extensions of APIs), not on UIs and editors. While I do not think that the usage of APIs is mandatory for humans, they certainly are for automation. The tools must be designed to be API first, UI (and everything else) second. Without APIs, there is no reliable automation, and without us knowing how to write scripts, we cannot provide the things the machines need.

> That leads us to the last rule: **all the tools must be able to speak with each other through APIs**.

---

## Which rules did we define? #

1. Git is the only source of truth.
2. Everything must be tracked, every action must be reproducible, and everything must be idempotent.
3. Communication between processes must be asynchronous.
4. Processes should run for as long as needed, but not longer.
5. All binaries must be stored in registries.
6. Information about all the releases must be stored in environment-specific repositories or branches.
7. Everything must follow the same coding practices.
8. All deployments must be idempotent.
9. Git webhooks are the only ones allowed to initiate a change that will be applied to the system.
10. All the tools must be able to speak with each other through APIs.

The rules are not like those we can choose to follow or to ignore. They are all important. Without any of them, everything will fall apart. They are the commandments that must be obeyed both in our processes as well as in the architecture of our applications. They shape our culture, and they define our processes. We will not change those rules, they will change us, at least until we come up with a better way to deliver software.

> - Were all those rules (commandments) confusing?
> - Are you wondering whether they make sense and, if they do, how do we implement them?
>
> **Worry not.** Our next mission is to put GitOps into practice and use practical examples to explain the principles and implementation.

---

We might not be able to explore everything in this chapter, but we should be able to get a good base that we can extend later. However, as in the previous chapters, we need to create the cluster first.