GitLab

A beginners guide to

- An introduction to GitOps
- The benefits of infrastructure automation
- GitOps best practices

Table of contents

Introduction

As software applications become more sophisticated, the demands on infrastructure increase. Infrastructure teams need to support complex deployments at immense scale and speed. While much of application development has been automated, infrastructure has remained largely a manual process requiring specialized teams. Instead of manual processes, is there a repeatable and reliable way to design, change, and deploy software environments? Infrastructure-as-code (IaC) tools like Ansible and Terraform are a good start, but they don't solve the entire problem. Teams need a prescriptive workflow that puts IaC into action automatically.

This eBook will introduce the infrastructure automation process of GitOps and how it offers an end-to-end solution for designing, changing, and deploying infrastructure. In this eBook, you'll also learn:

- ☐ How GitOps works with processes you already use in application development
- ☐ The three components teams need to get started with GitOps
- ☐ GitOps best practices and workflows

Organizations with a **mature DevOps culture** can deploy code to production hundreds of times per day. While the software development lifecycle has been automated, rolling out infrastructure is still largely a manual process. IT teams struggling to keep up with more frequent deployments is not a new problem.

When physical hardware was required, infrastructure automation was practically impossible. With virtualization, things got a bit easier. It wasn't until the cloud went public that large infrastructures could be completely automated with relative ease. The cloud doesn't require hardware and, unlike "traditional" servers and Virtual Machines (VMs), cloud native services can be created and managed independently without having to provision a VM or Operating System (OS).

By using scripting languages like PowerShell and Bash, IT teams are able to deploy various services to the cloud. Automated scaling is often included in cloud services, like serverless offerings. When scaling is not automatic, being able to deploy another instance of your service instantly is important.

Just because these services are available doesn't mean teams are able to use them effectively. AWS alone has over 200 services, and many companies rely on dozens of them. These services often have many settings. Using the AWS portal to deploy all services manually is time-consuming, error prone, and not realistic for large organizations.

GitOps offers a way to automate and manage infrastructure, and it does this by using DevOps best practices that many organizations already use, such as version control, code review, and CI/CD pipelines. Having infrastructure described as code allows you to deploy the same service over and over. By using parameterization, it's possible to deploy the same service, but to different environments and with different names and settings.

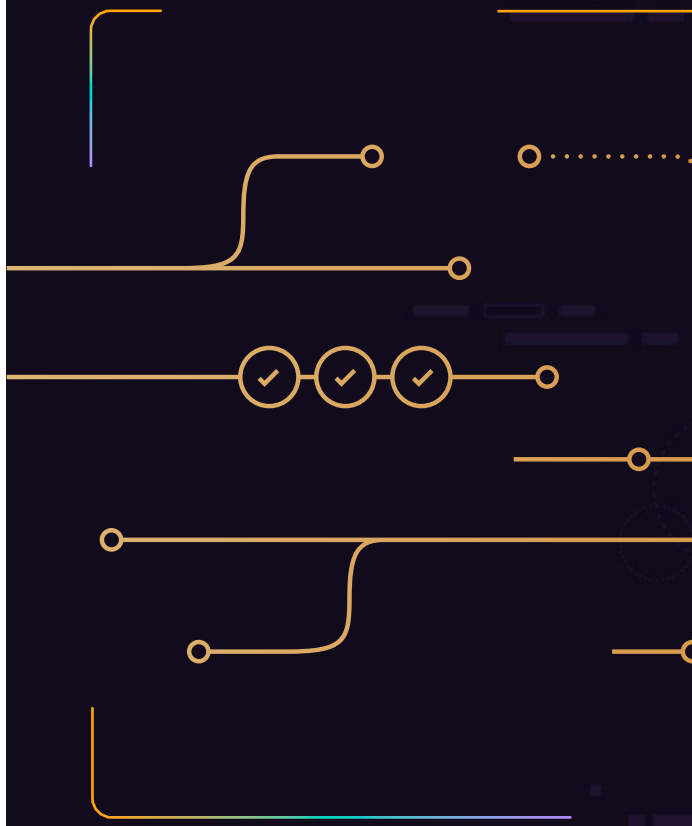
The road to GitOps

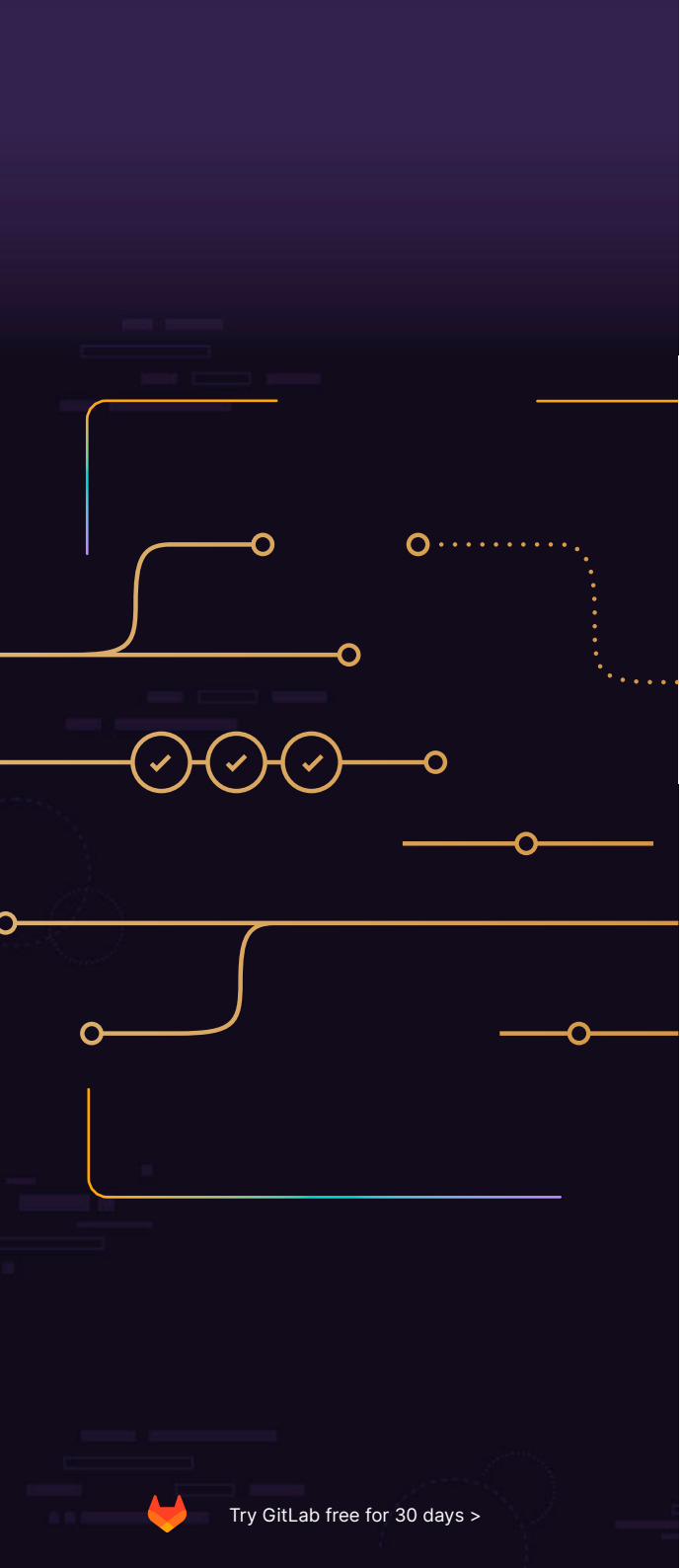
AWS has been publicly available since 2006, but even before that time, on-premises infrastructure management could be a daunting task for IT teams. Various servers ran several applications and services, and scaling up required IT to manually set up an entire server and reinstall the same applications with the same settings. Luckily, tools were developed to make this task a little easier.

The first generation of configuration management (CM) tools, like **Puppet** and **Chef**, made it easy to set up existing servers. IT could spin up a server or VM, install the Puppet or Chef agent, and let the tool establish everything needed to run applications on the server. These tools ran on on-premises servers, as well as on cloud servers.

First-generation CM tools were an efficient way to replicate all the steps to set up new production servers. With these steps now automated, setting up new servers became a lot easier. However, they still didn't provision new VMs and didn't work well with cloud native infrastructure.

Next came second-generation CM tools like **Ansible** and **SaltStack**. These tools can install software on individual servers, just like the first-gen CM tools, but can also provision VMs before setting them up. For example, they can create ten EC2 instances, then install all the needed software on each of these instances.





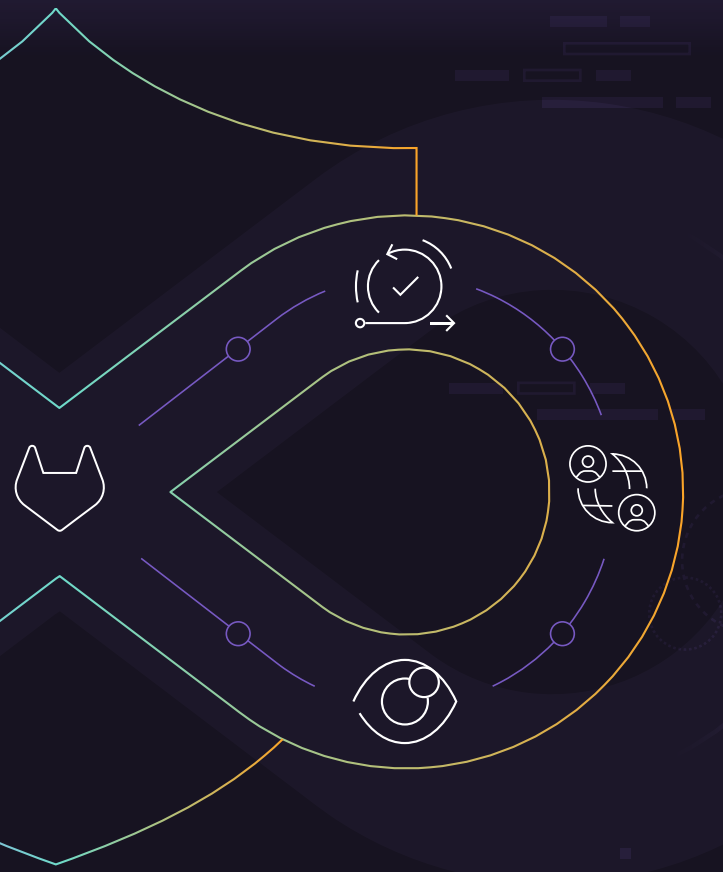
One important drawback of these CM tools is that they only provisioned and set up servers and VMs. They don't offer solutions for cloud native services.

Amazon CloudFormation appeared around the same as the second-gen CM tools. It doesn't handle server setup, but offers the ability to use declarative code to provision an entire AWS application architecture. There was no longer the need to click through the management console to manually create resources. You could simply describe your infrastructure as JSON or YAML and deploy it using the **AWS** Management Console, the Command-Line Interface (CLI), or the AWS SDK. But, as an Amazon service, it only works on AWS.

Microsoft Azure offers a similar tool, the Azure Resource Manager (ARM), which allows you to describe your infrastructure in JSON templates. But much like Amazon CloudFormation and AWS, ARM only works with Azure services.

When private clouds and other public clouds, like Azure and **Google Cloud**, gained traction, many enterprises switched to another cloud or went multicloud in order to not depend on a single cloud platform. To address this new requirement, **multicloud** CM tools appeared, such as **Terraform**. Simply describe your services and deploy them to multiple clouds/providers/cloud services.

An upside to these tools is that they unlock the ability to do things like version control, code review, and **continuous integration/continuous delivery (CI/CD)** on infrastructure code.



How GitOps works

GitOps takes tried-and-true DevOps processes and applies them to infrastructure code. As the name suggests, it combines Git and operations, or resource management. **Git is an open source version control system** that tracks code management changes. Like DevOps, the goal of GitOps is to use CI/CD to automatically deploy your resources by using code stored in your Git repositories.

With GitOps, your infrastructure definition code, defined as JSON or YAML and stored in a .git folder in a project, lives in a Git repository that serves as a single source of truth. Using Git's features makes it possible to see the complete change history for the organization's infrastructure code, and teams can roll back to an earlier version if necessary.

Git also makes it possible to do code reviews on your infrastructure. Code review is a key DevOps practice used to ensure that bad application code doesn't make it into production. This is just as important for infrastructure code. Bad infrastructure code can accidentally spin up expensive cloud infrastructure and cost the company thousands of dollars per hour. Likewise, a bad script could take down your application, resulting in downtime for your services. Code reviews prevent these mistakes by ensuring multiple people see every change before it's approved.

The biggest benefit to using Git and IaC is that you can now use continuous integration and deployment. With tools like GitLab CI/CD, you can automatically deploy (updated) infrastructure code and automatically apply it to your cloud environment. Resources that have been added to the infrastructure code are provisioned automatically and made ready for use. Resources that were changed are updated in your cloud environment and resources that are removed from the infrastructure code are automatically spun down and deleted. This allows you to write code, commit it to your Git repository, and take full advantage of all the benefits of the DevOps process, but for your infrastructure.

GitOps = IaC + MRs + CI/CD

- **IaC** – GitOps uses a Git repository as the single source of truth for infrastructure definitions. A Git repository is a .git folder in a project that tracks all changes made to files in a project over time. Infrastructure as code (IaC) is the practice of keeping all infrastructure configuration stored as code. The actual desired state may or may not be not stored as code (e.g., number of replicas, pods).

- **MRs** – GitOps uses **merge requests** (MRs) as the change mechanism for all infrastructure updates. The MR is where teams can collaborate via reviews and comments and where formal approvals take place. A merge commits to your master (or trunk) branch and serves as a changelog for auditing and troubleshooting.

- **CI/CD** – GitOps automates infrastructure updates using a Git workflow with continuous integration and continuous delivery (CI/CD). When new code is merged, the CI/CD pipeline enacts the change in the environment. Any configuration drift, such as manual changes or errors, is overwritten by GitOps automation so the environment converges on the desired state defined in Git. GitLab uses CI/CD pipelines to manage and implement GitOps automation, but other forms of automation such as definitions operators can be used as well.

The benefits of GitOps

A GitOps framework makes infrastructure automation possible, and while automation has value in itself, it's not the only advantage to GitOps. Organizations that adopt GitOps enjoy other benefits that can make a long term impact.

Collaboration on infrastructure changes. Since every change will go through the same change/merge request/ review/approval process, senior engineers can focus on other areas beyond the critical infrastructure management.

Faster time to market. Execution via code is faster than manual point and click. Test cases are automated and repeatable, so stable environments can be delivered rapidly.

Simplified auditing. When infrastructure changes are conducted manually across a set of multiple interfaces it can make auditing complex and time consuming. Data needs to be pulled from multiple places and normalized in order to conduct the audit. Using GitOps, all changes to environments are stored in the git log making audits simple.

Less risk. All changes to infrastructure are tracked through merge requests, and changes can be rolled back to a previous state.

Less error prone. Infrastructure definition is codified and repeatable, making it less prone to human error. With code reviews and collaboration in merge requests, errors can be identified and corrected before they ever make it to production.

Reduced costs and downtime. Automation of infrastructure definition and testing eliminates manual tasks, improves productivity, and reduces downtime due to built-in revert/ rollback capability. Automation also allows for infrastructure teams to better manage cloud resources, which can also improve cloud costs.

Improved access control. There's no need to give credentials to all infrastructure components since changes are automated (only CI/CD needs access).

Collaboration with compliance. In heavily regulated contexts policy often dictates that the number of people who can enact changes to a product environment remains as small as possible. With GitOps almost anyone can propose a change via merge request opening the scope of collaboration broadly while keeping the ability to strictly limit the number of people with the ability to merge to the production branch to maintain compliance.

Common GitOps tools

What makes GitOps unique is that it's not a single product, plugin, or platform. GitOps is a framework that helps teams manage IT infrastructure through processes they already use in application development. Popular tools are Ansible, Terraform, and Kubernetes, but the GitOps process is largely technology agnostic (save for Git, of course).

GitOps is suited for a variety of scenarios. GitOps and Kubernetes is a particularly good fit, for example. Kubernetes works on all major cloud platforms and uses stateless and immutable containers. Since containerized apps running in Kubernetes are self-contained, you don't need to provision and configure servers for each app. Provision Kubernetes clusters and other needed infrastructure, like databases and networking, using Terraform.

Deploying stateful applications requires some additional consideration for persisting data to external services, like an Amazon Aurora database instance or a Redis cache. While Kubernetes lends itself well to a GitOps framework, it's not a requirement for doing GitOps. You can use it with traditional cloud infrastructure like VMs, too. In this case, you'd provision with Terraform and then use a CM tool like Ansible to configure new VMs.

Git code repository	git		
Git management tool	GitLab	GitHub	Bitbucket
Continuous Inegration tool	GitLab	Jenkins	circleci
Continuous Delivery tool	GitLab	Spinnaker	flux
Container Registry	GitLab	docker hub	
Configuration manager	ANSIBLE	CHEF	puppet
Infrastructure provisioning	Terraform	pulumi	AWS CloudFormation
Container orchestration	kubernetes		

Best practices for getting started with GitOps

For teams that are used to making small, manual changes to infrastructure, adopting a process like GitOps can be a big adjustment. GitOps is a framework that requires infrastructure teams to adopt new habits, and lose old habits. This can take some time and may not come naturally to every team. Having best practices that are referenced frequently will be helpful in committing to the long term strategy of GitOps.

Define all infrastructure as config files

Firstly, make sure all the infrastructure you want to manage via GitOps is described in IaC config files. Ideally, these files should be written in declarative code. This means you describe the end state of what you want rather than instructions of how to get there.

For example, use a JSON file with properties describing how you want to set up your services, rather than a JavaScript file where you instruct a provider to create services for you. While many cloud provider and CM tools allow for declarative syntax, it's best to choose tools that are designed to be used declaratively.

For the best results, describe **all** of your infrastructure. You may be tempted to leave out that one service that only uses default settings and takes only a minute to set up. The fact is, that manual action is easy to forget when you're spinning up a new environment. Others will likely not even know they have to deploy this one service manually. Omissions like this are a form of **technical debt** that can build over time and sabotage your GitOps strategy.

If you're already using IaC and you want to automate it with GitOps, start by adding your infrastructure code to the Git repository you plan to use for GitOps.

If you're not using IaC, defining your existing infrastructure using config code will take somework. AWS makes it easy to **create CloudFormation config files from existing resources**. Terraform can import existing infrastructure from various providers, but it doesn't do 100% of the work for you.

The goal, of course, is to have all infrastructure described as code, but that is a journey that takes time. Don't get overwhelmed with trying to automate all your infrastructure at once. Eat the elephant one bite at a time. If you work iteratively, more of your infrastructure will move to GitOps workflows over time.

Document what you can’t automate

It’s not always possible to automate everything. For example, Azure has some (usually newer) settings that are not yet added to ARM templates. A common workaround is to use PowerShell.

Another example is when working with third-party providers. Imagine working with a supplier who needs to manually approve-list your IP addresses. An approve-list request can only be supplied by a manager. The manual action for every new service and environment is to look up the IP address, pass it on to your manager, and have them email it to the supplier. Make sure such processes are very well-documented.

In all likelihood, you'll always have some legacy environments that need manual attention. Document these instances so that they’re accounted for.

Outline a code review and merge request process

It’s important to familiarize GitOps teams with Git and code reviews. Some teams already use a Git repository as a place to store config code, but don’t use features like merge requests. As a starting point, take a look at the code review guidelines for the **GitLab open source project**. This project can give you a sense for the types of information you'll want to eventually add to your code review guidelines.

Before approving a merge request, set a **minimum number of reviewers** so all code is reviewed by at least a few members of the team.

For teams new to GitOps, another option is to set up "optional reviews" rather than set up "required blocking reviews". Since this a new process, take some time to get used to doing code reviews and develop a good cadence. Once teams are familiar with the toolset and practices, implement mandatory reviews to ensure code reviews happen every time.

As we've recommended elsewhere – start small and simple. If you start out with a complex set of guidelines, no one will want to adopt your process. Focus more on adoption than on doing what's best. Over time, iterate on code review guidelines to make them more robust and complete.

Consider multiple environments

It's good practice to have multiple environments. One example you might follow is the DTAP environments: Development, Test, Acceptance, and Production. Code can be rolled out to the Development or Test environment, after which you can test whether the services are still available and working as expected. If they are, you can further roll out your changes to the next environments.

After you have rolled out your code into your environments, it's important to keep your code in sync with your running services. Once you know there's a difference between your system and your configuration, you can fix either one. A solution to this problem is to use immutable images, such as containers, so that it's less likely to have differences.

Tools like **Chef**, **Puppet**, and **Ansible** have features like "diff alert," which notifies you when your services differ from your configuration code. Using tools such as **Kubediff** and **Terradiff** give you the same features for Kubernetes and Terraform.

Make CI/CD the access point to resources

One practice that encourages a GitOps workflow, and reduces manual changes to cloud infrastructure, is to make your CI/CD tooling the access point for cloud resources.

Of course, having this access during initial development can really help teams write their code and you may need incidental access for various reasons. However, switching your mindset from "access unless" to "access because" can help in adopting and following the GitOps process.

Have a repository strategy

Think about how you want to set up your repositories. You may want to use a single repository for all your infrastructure. It makes sense to use a repository for your shared resources, but keep the code for service-specific resources with those services. Another approach is to keep resources for different projects in different repositories. This all depends on the structure of your organization, your services, and your personal preferences.

A few things to consider when deciding on a mono or multi-repo strategy:

- [ ] Do you frequently have contractors or individuals working on projects where it may not be secure for them to have access to all code?
- [ ] Do you have multiple dependencies to consider?
- [ ] Do you want your repository to serve as a single source of truth?

Google, one of the largest tech companies in the world, **uses a single repo for all code**. HashiCorp recommends that each repository containing Terraform code **be a manageable chunk of infrastructure**, such as an application, service, or specific type of infrastructure (like common networking infrastructure). The definition of "manageable" can vary, of course.

Consider a Git branching strategy, like **feature branching**, so that multiple people can work on the same repository simultaneously.

Keep changes small

Whatever you do, always be sure to keep your commits small. This allows for a fine-grained changelog where it's easier to rollback separate changes. At GitLab, we refer to this process as **iteration**, and it's what allows us to make changes quickly. If we take smaller steps and ship smaller, simpler features, we get feedback sooner.

CI/CD pipelines with GitOps and Terraform

Set up your CI/CD to first validate infrastructure code, such as by using the Terraform **validate** command or a linter for JSON files. Your infrastructure code should be handled as if it is production code. You want your production code to be clean and consistent.

When someone commits invalid code, make sure the build or validation fails and the team is immediately notified. This allows the team to quickly solve the issue by either applying a fix or rolling back the commit. If you're making small changes, that also makes it easier to find problems.

If the code is valid, the CI/CD should run any commands necessary to provision the infrastructure defined in the config code. For example, the Terraform **apply** command or **AWS update-stack** for CloudFormation.

To see a sample GitOps project that uses Terraform, CI/CD, and Kubernetes, you can visit our **GitOps-Demo** Group. From there, we've provided links to Terraform security recommendations, Terraform code to represent each configuration for three major cloud providers, and instructions for reproducing this demo within your own group.

Go to the GitOps-Demo Group

The future of infrastructure automation

GitOps isn't magic: It just takes IaC ops tools you already know and wraps them in a DevOps-style workflow. This allows for better revision tracking, fewer costly errors, and quick, automated infrastructure deployments that can be repeated for a **multi-environment or even multicloud setup**.

By adopting GitOps, organizations improve the developer experience because often-dreaded releases become fully automated, allowing developers to focus on just their code. Teams eliminate or minimize manual steps and make deployments repeatable and reliable.

Infrastructure maintenance often becomes a problem that takes up a lot of time. By fully automating this process, infrastructure can be elastic and keep up with frequent application deployments.

GitOps also improves security and standardization. By practicing GitOps, developers have no need to manually access cloud resources and additional security checks can be put in place at the code level in CI/CD pipelines.

GitLab can help you get started with a GitOps workflow. From GitLab, you can manage physical, virtual, and cloud native infrastructures (including Kubernetes and serverless technologies). GitLab also has tight integrations with industry-leading infrastructure automation tools like Terraform, AWS Cloud Formation, Ansible, Chef, Puppet, and others. In addition to a Git repository, GitLab offers CI/CD, merge requests, and single sign-on simplicity so that everyone can collaborate and deploy from one platform to any cloud provider.

About GitLab

GitLab is a DevOps platform built from the ground up as a single application for all stages of the DevOps lifecycle enabling Product, Development, QA, Security, and Operations teams to work concurrently on the same project.

GitLab provides teams a single data store, one user interface, and one permission model across the DevOps lifecycle allowing teams to collaborate and work on a project from a single conversation, significantly reducing cycle time and focus exclusively on building great software quickly.

Built on Open Source, GitLab leverages the community contributions of thousands of developers and millions of users to continuously deliver new DevOps innovations. More than 100,000 organizations from startups to global enterprise organizations, including Ticketmaster, Jaguar Land Rover, NASDAQ, Dish Network and Comcast trust GitLab to deliver great software at new speeds. GitLab is the world's largest all-remote company, with more than 1,200 team members in over 65 countries.


GitLab