GitLab Architecture: A Complete Guide

Bibin Wilson, Arun Lal
Oct 9, 2025 6 min read

ON THIS PAGE

In this blog, we will explore the GitLab architecture and the components that work together to build a comprehensive software development lifecycle.

By the end of this blog, you will have learned the following.

1. The GitLab architecture
2. The core components of GitLab
3. The CI/CD architecture of GitLab
4. The storage of GitLab
5. The high availability and scalability of GitLab
6. The authentication and authorization
7. The monitoring

We can start from the architecture of GitLab to understand the components.

What is Gitlab

GitLab is a popular DevOps tool that combines version control, CI/CD, and project management tools.

It is a end to end platform that can handle the complete software development lifecycle. It is a perfect fit for organisations that needs a unified platform instead of multiple tools.

Companies like Agoda, Airbus, CERN, Goldman Sachs, and NVIDIA use GitLab for their CI/CD and DevSecOps implementations.

Overall Gitlab offers the following.

- **Git repository management** - You can host and manage your code like Github.

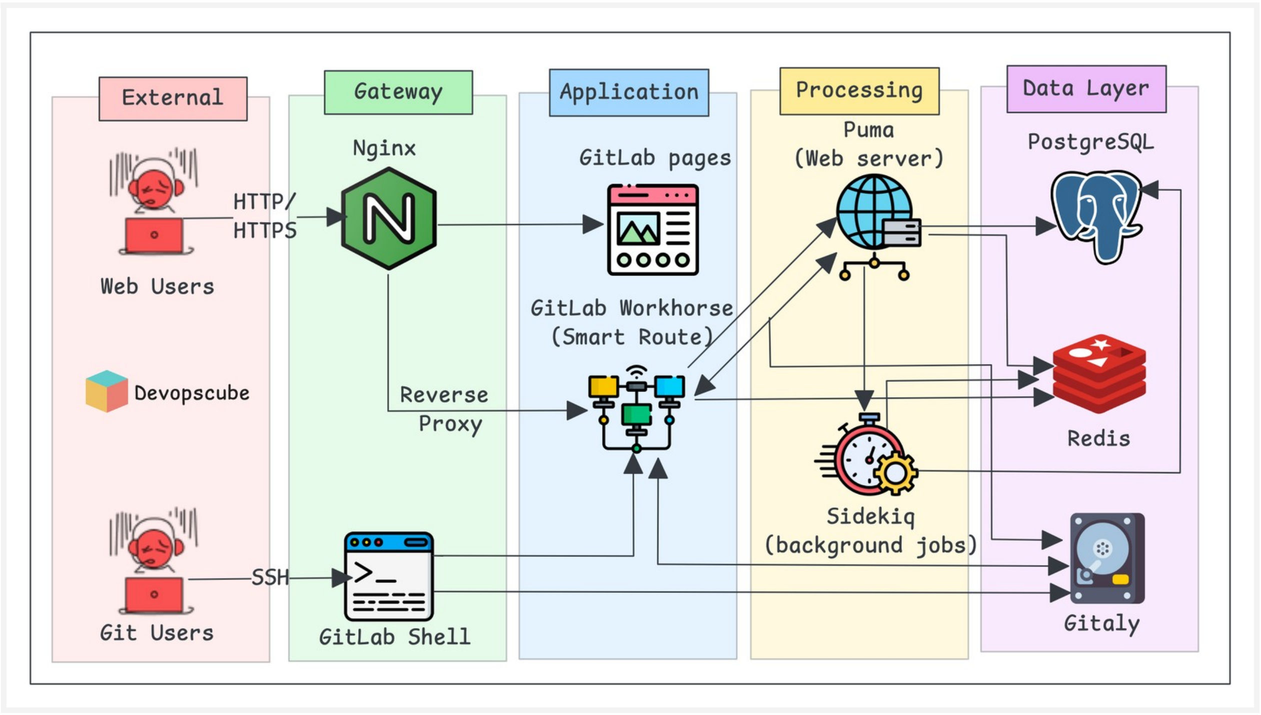
- **CI/CD pipelines:** It supports entire CI/CD workflow using simple yaml configs.
- **Issue tracking:** You can Plan and track work, bugs, and feature requests
- **Code review:** You can perform reviews, merge requests with approval workflows.
- **Container registry:** You can store and manage Docker images
- **Security scanning:** If offers Built-in security testing and vulnerability detection.

Overall, it has everything integrated in one place rather than managing multiple separate tools.

📌 **GitLab** = Code hosting + CI/CD + Issue tracking + DevOps tools

GitLab Architecture

The following outlines the architecture of GitLab and how its components interact with one another.



GitLab Architecture

Here is how all the components work together, as shown in the above arcitecture diagram.

- When a **web user** accesses GitLab, the HTTP(S) request first goes to **Nginx**, which acts as a **reverse proxy**.

as a reverse **proxy**.

- Nginx routes the request to the **GitLab Workhorse**, which decides whether to handle it itself or send it to **Puma**.
- **Puma** is a **web server** that handles time-sensitive requests, such as retrieving project, issue, and user data from **PostgreSQL**, or caching or session information from **Redis**.
- If there are background tasks, such as sending notifications or executing CI/CD jobs, Puma sends them to **Sidekiq**, which manages the jobs using **Redis** and stores and retrieves data in **PostgreSQL** and **Gitaly**.
- Gitaly is a storage where all the Git repository data is stored.
- When users perform the Git operations, they directly access the Gitaly through the **GitLab** shell over SSH.

All these components need to work with GitLab properly. We can explore them in the next section.

Core Components of GitLab

Once of the key requirement of DevOps engineers is to understand the core architecture of Gitlab. Becuase,you cant optimize or fix what you dont understand.

By understanding the core architecture, you can trohbleshoot, scale, perform upgrades, integrations and more.

GitLab consists of many components. We broke down each component and explained what each component does. Lets get started.

Proxy/HTTP Server (Nginx)

GitLab uses the Nginx web server as the **reverse proxy** for user requests.

This proxy even handles the SSL/TLS termination to encrypt the traffic between the user and GitLab.

GitLab Web UI / API

We can interact with GitLab in two ways: directly from a web browser using the User

Interface (Web UI), and through API calls, primarily for programs to interact with.

> 💡 GitLab offers two API endpoints, which are **REST** and **GraphQL**.
>
> **GraphQL** is efficient for complex queries, modern integrations, UI applications

Controllers / Rails / Puma

GitLab core is built on the Ruby on Rails web application framework.

When you perform an action in Gitlab, several components work together at the backend to handle the requests.

It starts with **Puma**. It is a **web server** of GitLab that listens to HTTP requests. It forwards the requests to the Rails application.

Rails follows the **Model-View-Controller (MVC)** pattern. When Puma passes the request to Rails, it routes to the **appropriate Controller** to tell what action should be performed.

GitLab Workhorse

Workhorse is a **reverse proxy** that sits in front of Puma.

It **handles slow and large HTTP requests** (e.g., Big uploads/downloads). Once it picks the larger requests, it forwards the remaining time-sensitive requests to Puma.

GitLab Shell

We can access the shell of GitLab over the SSH protocol to download the code and upload the changes from the local to the remote.

We use the standard authentication methods to securely perform these Git operations on GitLab.

The storage and access of Git in GitLab is handled by **Gitaly**.

Gitaly

The storage of the GitLab for the **code we store** is managed by a service called **Gitaly**.

The requests come from Rails and Shell for the Git operations, such as push, commit, MR, etc. These components talk to Gitaly through the **gRPC** protocol.

Sidekiq

Sidekiq is a **background job processing** unit for long running tasks.

For example, a task needs to send hundreds of emails to users, and Rails creates a job queue in Redis.

The Sidekiq workers then pick them and execute. So the main requests will not be affected.

Redis

Redis server is an **in memory store** of GitLab to store the temporary data for the following key operations.

- Queue system for the Sidekiq jobs
- A caching system for the frequently accessed data.
- Store temporary information of the applications, like the state of a CI/CD job.

All the long lived data will be stored in the PostgreSQL database.

PostgreSQL

PostgreSQL is a relational database that **stores long lived data** like user account information, project information, permissions, issues, and MR, pipeline metadata, registry metadata, etc.

GitLab Bare Repository

In Gitlab, every project has a **bare repository** on the server side. It is a special type of Git repository that does not have a working directory repository that does not have a working directory.

Normally, in a local Git repository, the `.git` directory stores all the data commits, branches, tags, and history.

In GitLab, only this `.git` data is stored on the server. That is what forms the **bare repository**. This repository exists mainly for **synchronization.**

In short, the bare repository in GitLab exists mainly to implement core Git functionality (git push, git pull etc)

Also, all GitLab components, such as GitLab Rails, GitLab Shell, and Gitaly, work together to handle these Git operations, whether they come through HTTP or SSH.

GitLab Storage and Data Persistence

We have already looked at the storage components of GitLab individually.

Overall, here is how storage works in Gitlab.

1. All the GitLab repository information is stored in the **bare repository** (like a database of repositories) and the data is stored in the file system by the **Gitaly** server.
2. All the **metadata** (users, projects, issues, CI/CD builds, labels, etc) of GitLab is stored in the **PostgreSQL** database.
3. GitLab uses a multi layer **cache** using **Redis, In-memory and HTTP** for quick response, also uses for the job queues.

💡 To store the **large binary data** like CI/CD artifacts, package registries, attachments, etc, we can use **object storage** like AWS S3, Azure Blob, or MinIO.

GitLab Scaling and High Availability

When implementing Gitlab in DevOps projects, scaling & avaialbilty]becomes a key part when the projects grow. For example, if a central platform team provides GitLab as a service, many teams may onboard to the same GitLab instance.

In such cases, you dont need to scale the entire GitLab system at once. Instead, you can scale individual components based on the workload and requirements.

For example, PostgreSQL, Redis, and Gitaly are the **stateful components** and these handle heavy I/O and background operations. So instead of running them a single instances, you can deploy them a clusters.

The replicas in the clusters should **span multiple zones** for high availability. Also, periodic backups will help in disaster recovery scenarios.

GitLab Authentication

GitLab by default, provides the standard login using username and password. Additionally you can enable Two-Factor authentication to improve security.

For enterprise setups, you can use the **external authentication providers** such as LDAP, SAML, or OAuth2 for the user authentication.

To securely access the GitLab shell for the Git operation, we can use the Secure Shell (SSH) method.

For API based authentication, you can use the Personal Access Tokens (PATs) or Bearer Token.

GitLab Authorization

We can control access to Gitlab by assigning permissions to users or applications, specifying what actions they are allowed to perform and on which resources.

GitLab follows a **hierarchical permission model**, where access can be granted at either the group level or the project level.

The following are the roles that we can assign.

1. Guest (view only)
2. Reporter (Read all & Issues management)

3. Developer (Read & Write)
4. Maintainer (Project Admin - Almost full control)
5. Owner (Full control)

GitLab Monitoring with Prometheus & Grafana

A Self-hosted GitLab insrtance can be monitored using Prometheus and Grafana. We can track all the performance, system health and issues using its internal metrics and visualize them on dashbaords.

GitLab also stores logs for all its components in a central location at `/var/log/gitlab/`. You can use log collectors such as Fluent Bit to gather these logs and send them to a central logging or monitoring system for further analysis.

Conclusion

Thats a wrap!

In this blog, we have looked at the Gitlab architecture and how all the key components interact behind the scenes.

We also discussed important topics such as storage, scalability, authentication, and authorization, which are very important in enterprise environments.

To know more about how real organizations use GitLab for their project, you can refer to this official documentation.

In the upcoming blogs, we will cover the setup and configurations of GitLab.

devops CI/CD GitLab

ABOUT THE AUTHOR

Bibin Wilson

Bibin Wilson (authored over 300 tech tutorials) is a cloud and DevOps consultant with over 12+ years of IT experience. He has extensive hands-on experience with public cloud platforms and Kubernetes.

Featured

Kubernetes Tutorial For Beginners: 72 Comprehensive Guides
19 Nov 2025

Kubernetes Built-in AI/ML Features You Should Know
05 Jun 2025

Python For DevOps: Guide for DevOps Engineers
17 May 2025

Jenkins Tutorial For Beginners: 21+ Practical Guides
01 Apr 2023

Latest

Setting up Istio Ingress With Kubernetes Gateway API
26 Nov 2025

Sandboxed Containers: What They Are and How They Isolate

Workloads

25 Nov 2025

Setup Rancher on Kubernetes Cluster

21 Nov 2025

How to Set up Istio on Kubernetes Cluster? [Step-by-Step]

17 Nov 2025

READ NEXT

Setting up Istio Ingress With Kubernetes Gateway API

Sandboxed Containers: What They Are and How They Isolate Workloads

Setup Rancher on Kubernetes Cluster

Kubernetes Tutorial For Beginners: 72 Comprehensive Guides

How to Set up Istio on Kubernetes Cluster? [Step-by-Step]

Navigation

📚 Tutorials

Kubernetes Tutorial

ArgoCD Tutorial

Jenkins Tutorial

💻 Roadmaps

DevOps Engineer Roadmap

Python For DevOps

Golang For DevOps

Git For DevOps

Shell Scripting For DevOps

📕 Certification Guides

CKA Exam Guide

CKAD Exam Guide

CKS Exam Guide

KCNA Exam Guide

Resources