Architecture Project

System: GitLab
Date: 2025.03.14
Andrew Steiger, Greg Atamian, Reed Klaeser

Table of Contents

Introduction

In this section we introduce our system, describe its purpose, outline its relevant scope, and identify its audience.

Purpose

GitLab Community Edition (GCE) is an open-core end-to-end software development platform primarily developed by the company GitLab. GitLab Community Edition (GCE), hereafter GCE, aims to comprehensively address development needs by providing version control, issue tracking, code review, CI/CD, and more.

Scope

To ensure appropriate depth in the face of so many features we have chosen to focus this architecture review on the relevant components involved in tracking, building, testing, and deploying code in a traditional server environment. We also discuss issue tracking and code review features or explore the flexibility to deploy code to containers or serverless environments. Additionally, while a GitLab system could be configured with multiple GCEs for high availability, the system we discuss will have only one. Should we find that discussing other functionalities becomes necessary or informative we'll expand the scope in turn. An organization deploying code with the GitLab system, hereafter GitLab, will have GCE running on one machine and GitLab Runners on the machines being deployed to. Even though it is a different open-source application than GCE, we'll include GitLab Runners in our discussion for completeness. The GitLab system we have chosen to scope our project to orchestrates critical tasks over a network that ultimately allows developer changes to reach end-users. As we discuss later, the Quality Attributes and Requirements that must be met to make this happen include many of those we have discussed in class. We chose to investigate GitLab not only because it is large enough to support relevant discussions but also because it is a very well documented system.

Audience/Stakeholders

GCE's relevant audience includes developers whose changes are deployed through the system, technical leaders who manage these changes, technical administrators who manage the system, the quality assurance team responsible for validating the overall and/or individual quality of changes, and business administrators associated with the changes. Together these stakeholders represent the continued relevance of tracking software changes from ideation to ultimate delivery.

Glossary of Terms

- **A/B testing** - Exposing two versions to a customer and recording which they respond better to.

- **Batch processes** - Processing data in a single go rather than in real-time, often scheduled
- **CI Trace Chunk -** A detailed recording of a process occurring within the CI pipeline, with a defined start and stop time.
- **Code Review -** A process where software developers review each other's changes to code prior to those changes being deployed.
- **Container** - Operating system level or application-level virtualization environment
- **Continuous Integration and Continuous Deployment (CI/CD)** - A process of deploying code changes frequently and reliably. Continuous Integration refers to an automated process of introducing code changes to a code base, and continuous deployment refers to an automated process of deploying those changes to different environments.
- **DevOps** - A practice which marries software development and business operations into a single effort. When followed, practitioners hope to improve the performance, efficiency and security of their software development and delivery.
- **Feature flags** - Toggles that allow enabling or disabling features in code without changing the code. This allows behavior to be toggled in production without a new deployment.
- **HA high availability** - Elimination of single points of failures with the goal of operating with minimal downtime.
- **HTTP/HTTPS** - Protocol for sending data over the network. HTTPS is the encrypted version where the S stands for secure.
- **IPC** – Inter-process communication is a way for processes on a machine to communicate
- **Issue tracking** - The process of recording and managing issues such as bugs or features that must be developed
- **Kanban** - A work management system using boards to represent the progress of tasks through a workflow
- **Kubernetes** - A container orchestration platform that automates the deployment, scaling, and management of containerized applications.
- **Git** - A version control system
- **GitLab** - Used to refer to the system inclusive of CE and Runner. A tool for developers to implement code changes via CI/CD and DevOps
- **GitLab Community Edition (GCE) -** The GitLab management instance
- **Gitlab Runner** - Process that handles jobs within a pipeline
- **Pipeline** - Composed of various tasks that promote code changes through various testing environments and ultimately to the production environment.
- **Privileged User** - These users represent a minimal subset of the contributing business user base. They are expected to have a high understanding of the system and what is described and proposed as required in this document.
- **Ruby** - High-level general-purpose programming language
- **Ruby on Rails** - Server-side web application framework
- **Semantic Versioning** - Convention of assigning unique version numbers to software according to the significance of changes between versions

- **SMTP** - Protocol for email transport
- **systemd** - A software suite for Linux providing several system components, especially service configuration.
- **TCP/IP** - Provides reliable, ordered, error checked delivery of data over the network.
- **Virtual machine (VM)** - An operating system abstracted to run on a hypervisor as opposed to directly on hardware.
- **YAML** - A human readable serialization language used for configuration files

Overview of Requirements

The main function of GitLab is to provide a full environment for developer teams to collaborate on translating business requirements into deployed end-user applications. Although there are various plugins and modules available to accomplish this end goal, we consider in this documentation a base case for functionality. To this point, the requirements listed within revolve around standard code base management with Continuous Improvement pipelines, as well as deployment and testing via Continuous Deployment pipelines.

Functional Requirements

Organizational Process Requirements

Key GitLab Native Components: Rails/Puma, GitLab Runner, Sidekiq, Gitaly

OPR1 - The system shall provide methods for Continuous Improvement, where a developer or contributors to the code base are able to commit code changes to a respective repository. After peer review of code, these changes propagate the CI pipeline in a completely automated fashion and are verified in various testing environments before ultimately being prepared for deployment to a production environment.

OPR2 - The system shall provide methods for Continuous Deployment, where tested builds from the CI pipeline are able to be deployed to further testing and production environments, in a completely automated fashion.

OPR3 - The system will maintain work items used to describe business requirements, via Kanban board or similar manner. Assignment of work items will be to single sign on identity used in other aspects of the system.

OPR3 - The system will maintain code bases in repositories and utilize git standard commands to maintain this code base. Git authentication will synchronize with the single sign-on identity used in other aspects of the system.

Control Requirements

Key GitLab Native Components: Rails/Puma, GitLab Shell, Sentry

CR1 - Users of the system will be presented with recovery actions such as manually stopping, starting and retrying pipeline tasks. These users will be able to rollback versions of the deployables due to compilation or testing errors that resulted from merged code.

CR2 - The components will be configurable beyond the development stage, to incorporate changes at runtime, without the need for redeployment. A common abstracted interface will be provided with the executables to implement these changes.

CR3 - The system will be accessible remotely and securely by the code contributors. Communication between parties will be completely attainable through common components provided by the system.

Flexibility Requirements

Key GitLab Native Components: GitLab Runner, Sidekiq, PostgreSQL, Redis

FR1 - The technical components of the system will be flexible enough to allow for alternate platforms, such as database providers, or version control vendors.

FR2 - The system will allow the introduction or replacement of entire components, of varying size. This may include vendor or third-party components, new components required to satisfy new functionality (such as a new microservice, data source, UI). The system will be designed to grow its user base.

FR3 - Components of the system such as the GitLab Runner, or the Postgres Databases, will be easily distributed to alternate locations as needed. This will allow encapsulating the responsibilities of the underlying GitLab hardware.

Non-functional Requirements

Modifiability

Key GitLab Native Components: GitLab Runner, Sidekiq

NFR1 - During the Continuous Integration phase of code change, developers and code contributors must be able to merge code into shared branch repositories, where branch versioning prevents change collision, and exactly one new version tag is created with each single merge of a code branch.

NFR2 - Privileged users will be able to modify and configure the pipeline through the same mechanisms as described in NFR1 for Continuous Integration, to adjust the infrastructure. These changes encompass deployment strategies, scaling infrastructure, changing the location of components, adding and removing infrastructure etc. These changes will reflect in newly automated CI build tasks within 20 seconds of the changes being made.

Availability

Key GitLab Native Components: Rails/Puma, GitLab Shell, Sentry

NFR3 - The system will monitor and be aware of faults that occur in the pipelines. Users of the system must be able to quickly identify the health of the system, identify artifacts that are at fault, and compare cycle-time, with respect to all stages of the pipeline. These identified faults will be refreshed or made available to the UI within 2 seconds of detection.

NFR4 - Faults in the system will be clearly described and logged, defining at minimum the stage in the pipeline where it occurred, the time that it occurred, and the components directly affected. Appropriate parties will be notified via messaging such as email or notifications with these descriptions. The reported timestamp of the logs generated will deviate no more than 500ms from the time of the actual event.

Integrability

Key GitLab Native Components: GitLab Runner, Sidekiq, Gitaly

NFR5 - The components of the system will adopt a standard versioning mechanism to track changes to the code base over time. The version will be unique to the component, and testing of cross component version compatibility will be performed at the appropriate stages needed to ensure expected functionality. Semantic versioning will be enforced, and MINOR and PATCH versions will not exceed a value of 99 before a bump to the higher-level value is made.

Use Cases

CI/CD Deployment

This use case defines the most standard application of GitLab, to deploy or change an application via changes made by a code contributor.

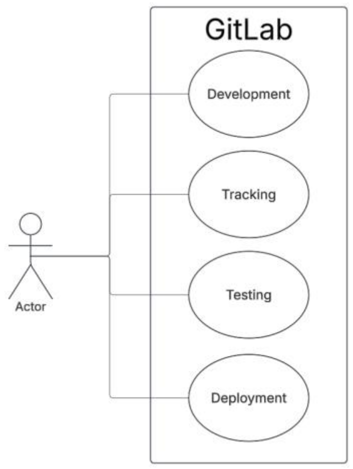


Figure 1 - Use Case Diagram

Contributors merging new code into the code base should automatically drive build and test processes in a development environment. Successful completion of tasks in the development environment should automatically trigger deployment into an integration environment and run integration and system tests. Successful completion of integration environment testing should result in an executable automatically deployed to a staging environment where tests are run with production-like data. Successful completion of staging environment tests should result in deployment to production.

Scalability

GitLab GCE can scale to meet the needs of organizations. For instance, though GCE ships configured to run one Redis instance configuration changes to the following files config/redis.yml, config/application.rb, config/sidekiq.yml can partition the Redis caches according to the type of state being stored. If a company found themselves deploying to one thousand instead of ten servers, the ability to scale Redis could prevent bottlenecks from high read/write traffic due to caching or job queues. Similarly, Gitaly (git store) or Sidekiq (background job processing) can be scaled with configuration files alone. In addition to these configuration changes, each new resource must be provisioned. We discuss this limitation and the limitation of relying on virtual machines rather than containers at length in the Limitations of Architecture section.

SSO Authentication

Applications, especially those like GitLab that contain intellectual property and are capable of breaking production systems, should be properly secured to effectively prevent unauthenticated access. Furthermore, they should give administrators the ability to choose the system that authenticates users, configure role-based access control, and create a paper trail of actions taken by authenticated users. The system should provide these features in a way that increases Usability.

- The user will authenticate once into the GitLab instance via Puma.

- This single identity will persist for all git operations and for tracking purposes within the application. In this way, code changes, pipeline task creation, and other system-wide actions will track whom the change was requested by.
- Usability will benefit from pairing fault tracking with the identity of the user(s) involved.
- Administrative users are designated to handle critical actions and configuration.
- A single source for authentication (Gitaly) will provision temporary access tokens for use with a user's session.
- Usability of the application will benefit from this single authentication method, which can be used for access to the entire application.
- When access tokens expire, the user should be redirected to the single sign-on page to reauthenticate.
- Elasticsearch will have access to all user data stored in a single Postgres Instance, to provide search results based on a user such as the individual's code changes, or work issues completed.

Context Diagram

We're interested in the GitLab system responsible for tracking, building, testing, and deploying code. This system allows changes made by developers to manifest into newly deployed executable code for customers to interact with in Production. Code changes are made by developers who use a git client to check in and out code stored on the management server. Depending on the organization, developers might also make configuration changes to GitLab or those may be left to administrators. GCE is responsible for managing these functions and for orchestrating the performance of relevant build and deployment work. Interfacing with the GitLab Runners on build, test, and production servers, runs what are called CI/CD jobs that perform the work of building and migrating code. GCE is a Ruby on Rails application that is typically configured to run on its own management server. GitLab Runner, hereafter Runner, is a Go application that runs CI/CD jobs and sends the results back to CE. Developers or administrators may view the results of CI/CD jobs as reported by GCE through their clients. GCE serves the web pages that developers and administrators view using a Puma application server. Frequently code is deployed and tested in test environments prior to being deployed to production. Testing strategies will vary but CI/CD jobs can be arranged sequentially so that changes are verified in test environments prior to being deployed to production. Once code is deployed to production users will be served the new code. It should be noted that production can be anything, an ecommerce website or even an internal backend system.

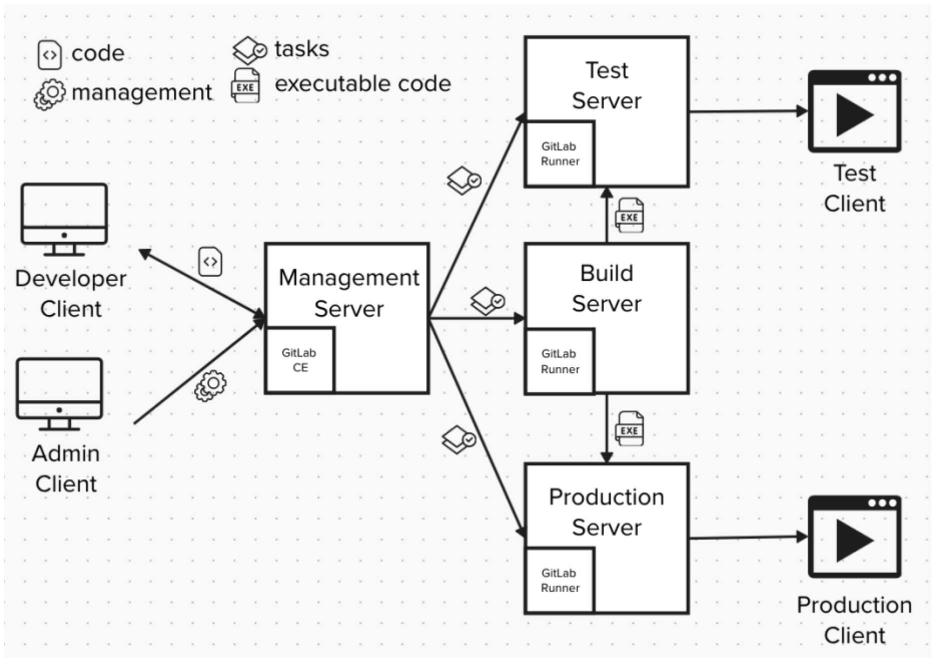


Figure 2 - Context Diagram

System Quality Attributes

Modifiability

A key to success for the system is its modifiability, as the system is designed to manage frequent code changes to multiple applications through continuous development. The technologies that development teams use is dynamic in nature, as is the architecture design that determines the components and connectors within the system. Transitively, this requires that the platform for managing code change, testing and deployment (GitLab), be just as flexible if not more. Among the requirements that result in the need for modifiability include changing entire components and structures and changing entire frameworks and platforms.

Availability

GitLab as a system will generally accommodate many users as well as types of users, who all work concurrently on multiple projects. The actions of these users will often initialize automated scripts to run, create new downstream tasks, or run entire workflow pipelines to a deployed environment. With a potential multitude of users, tasks and applications, a system with high Availability is imperative. Requirements that illustrate this consider the monitoring needed for a system of this variety, the administrative actions that can be taken on these work streams, and the detection of faults that may occur in any of the mentioned processes.

Integrability

The system needs to be able to easily incorporate change not only within the applications that are developed with the use of the system, but also for the system itself. Gitlab is composed of various open-source modules, developed by different teams, and often not exclusively for use with GitLab. The orchestration of updates, for these internal elements as well as system elements, needs to be clear and indefinitely possible. Standard Connector components should be in place to facilitate this change as well as options for configuration, to ensure that the system is correctly collaborating.

Scalability

Closely tied to the Availability and Modifiability of the system, the ability to increase or reduce resources is deterministic to the systems success. To handle frequent requests for change from the business, it must be considered that the infrastructure should be flexible enough to handle large changes with ease. It should be able to handle several new services, applications, storage devices and users. Conversely, the system should just as easily be able to de-register unneeded resources on command.

Distributability/Deployability

The nature of a Continuous Improvement/Deployment management system requires in the modern age that portability and distributional transparency be incorporated. The dynamic stakeholders or users of the system, as well as the dynamic components of the system, should have transparent communication regardless of location or even knowledge of a specific host or platform. Each element and cohesive concept should be encapsulated to achieve this. An example of this within the system would be the ability to update database versions, change a database location, change the underlying platform, or increase the database redundancy, without users noticing change. Satisfaction of this quality is arguably the core of satisfying other quality attributes previously mentioned.

Quality Attribute Scenarios

Modifiability Requirement (NFR2)

Privileged users will be able to modify and configure the pipeline through the same mechanisms as described in NFR1 for Continuous Integration, to adjust the infrastructure. These changes encompass deployment strategies, scaling infrastructure, changing the location of components, adding and removing infrastructure etc. These changes will reflect in newly automated CI/CD build tasks within 20 seconds of the changes being made.

Source

System Administrators, Special access users.

Stimulus

Starting/stopping the pipelines. Adding/modifying/removing pipeline tasks, pipeline order of operations, pipeline infrastructure locations.

Artifacts

The literal pipelines and tasks performed as part of a run. The configuration of pipelines and their tasks. Larger categories of tasks including those used for all types of test cases, those used for dependency management and verification, those used for building executables, those used for notification of pipeline status etc.

Environment

Changes can be made at runtime or downtime of a pipeline. In any logical environment starting with the CI Develop environment, but will likely apply to Integration/QA, Staging, and Production.

Response

Changes to pipelines and/or tasks are immediately visible for the next run, or current run is modified in the expected way such as skipping a task or stopping the pipeline.

Response Measure

The Response is apparent to the user within 20 seconds of the directive being issued.

Tactics to Satisfy Modifiability

Component Replacement

This is implemented at compile time or build time, early in the module lifecycle. The pipelines and their associated tasks are defined completely by scripts written in YAML[i]. The various files reference commands used to perform the tasks and are declarative in that a high-level file may reference a task to run which exists in another file. The YAML files contain global keywords (variables) that can be used within any YAML file of a pipeline, which can be passed to downstream pipeline file sets with "*include:<type>:<reference>*" syntax. These files are stored in a repository or branch just as code is stored. A privileged user committing changes to these files triggers reactive updates in the UI when valid changes are made.

Configuration-Time Binding

This binding occurs typically at deployment, startup time, or initialization time, late in the module lifecycle. These parameters are used within the applications themselves to specific key values such as topics the application will publish to, or conversely subscriptions the application will listen to. They may contain configuration for feature flags[ii] to promote A/B testing or rollback of code paths as needed. Additional configurations that would be applied at startup would include relevant hostnames, upper and lower limiting parameters, etc. If needed, a centralized system could be implemented to issue these values so that changes could be made at runtime and implemented via backdoor. This late module configuration is often the most effective, but with a tradeoff to additional upfront development time.

Availability Requirement (NFR3)

The system will monitor and be aware of faults that occur in the pipelines. Users of the system must be able to quickly identify the health of the system, identify artifacts that are at fault, and compare cycle-time, with respect to all stages of the pipeline. These identified faults will be refreshed or made available to the UI within 2 seconds of detection. *Key GitLab Native Components: Prometheus, Grafana, GitLab Exporter, Node Exporter, Redis Exporter, Alertmanager*

Source

Software and software infrastructure, physical infrastructure.

Stimulus

An incorrect response from a pipeline task, incorrect timing of an expected task.

Artifacts

The pipeline tasks and the subsequent actions that result from the task completion.

Environment

Normal operation, all stages of the pipeline run.

Response

Notification of faults is handled to predetermined users, based on configured relevance. This may include logs, UI notifications, email etc.

Response Measure

No more than 2 seconds elapses from when a fault is detected or a threshold is breached, before notification of the relevant parties is completed.

Tactics to Satisfy Availability

Detect Faults - Monitor, Heartbeat, Exception Detection

The broad coverage of detecting faults is available in the system. Monitors are assigned to deployed containers and storage systems. Heartbeat messages are sent to a monitor from deployed instances, even if they are deployed in a separate environment such as Kubernetes, via defined interfaces and protocols[iii]. System exception detection is built into code to determine if timeouts have occurred, or abnormal instructions have been received. The various methods of detecting faults should result in immediate notification via logs, or a message sent to an operator, based on relevance.

Recover from Faults - Redundant Spare, Rollback, Exception Handling, Retry

Mechanisms are in place to create redundant clones of elements. In the case of a fault in an application or overload of its resources, load balancing should route requests to a spare. If startup of an application with new changes or versioning occurs, this redundant spare mechanism should allow for rollback to a previous version that is known to function correctly. Exception handling within messaging such as HTTP requests, will contain exception classes to identify pre-meditated error codes that occur such as resource Not

Found, or Unavailable. Transient faults will implement retry logic within reason and should report when this limit has been exceeded[iv].

Integrability Requirement (NFR5)

The components of the system will adopt a standard versioning mechanism to track changes to the code base over time. The version will be unique to the component, and testing of cross component version compatibility will be performed at the appropriate stages needed to ensure expected functionality. Semantic versioning will be enforced, and MINOR and PATCH versions will not exceed a value of 99 before a bump to the higher-level value is made.
Key GitLab Native Components: Gitaly, GitLab Shell, GitLab Workhorse, GitLab Registry, GitLab Rails, GitLab Runner

Source

GitLab Release Manager

Stimulus

A GitLab Release Manager begins the monthly release process for an independently versioned component of GCE by releasing a new version to the first test environment.

Artifacts

The code environment is the artifact which is acted on by receiving the latest version.

Environment

The status of other components is a relevant environment attribute. For instance, a new version of another component may end up having unexpected interactions with the component in question. How production-like the test environments are is also relevant. For instance, test code environments frequently experience less load than production environments which could cause performance issues to be missed.

Response

Automated integration tests based on version numbers of each component run, creating the regressions needed to understand integration issues.

Response Measure

No more than 30 minutes of human intervention is needed to run and interpret the results of automated integration testing between GCE components.

Tactics to Satisfy Integrability

Not all GCE components are developed by Gitlab. Those that are (mentioned above in *Key GitLab Native Components),* are independently versioned and are released on the 22nd of each month in the manner described in the Quality Attribute Scenario above. We are not sure of the exact Response Measure used within GitLab, but it is clear from their release documentation[v] that the process of testing new versions is highly automated and approximates the process of little human intervention we described. GitLab employs the following tactics, all of which work towards this Quality Attribute Scenario and the overall goal of integrability.

Limit Dependencies - Encapsulate, Use an Intermediary, Adhere to Standards

GCE components are encapsulated using private public protections. Take for instance GitLab Shell, which is written in Go, a programming language where lowercase functions are treated as private and uppercase functions as Public. Public shell functions like New()[vi] are exposed and private shell internals like validate()[vii] are protected.

GCE does use an intermediary in the form of GitLab Workhorse, a reverse proxy, but its intention is more to relieve load from the Puma application server than to limit dependencies. It does, nonetheless, have the effect of limiting dependencies since requests like those from GitLab Shell or client web requests are funneled to Workhorse before being passed to Puma, Redis, or Gitaly.

Adhering to standards like those for semantic versioning described in NFR5 helps both automated and manual processes identify the correct software versions. This is essential in deploying the right code or identifying problematic changes. See the release documentation link above for a description of GitLab's standards policies.

High Level Architecture

Key Components

AlertManager (Prometheus Tool)

This tool handles alerts from various client applications, including Prometheus, to deduplicate, group and route these alerts to the respectively configured entity. This may include an email, or countless additional implementations which are made possible by a webhook receiver. Grouping alerts will in practice achieve results such as taking multiple alerts from identical instances of a service and grouping this as a single alert with references to each instance. It contains inhibition logic to repress alerts that are deemed unnecessary if other alerts are already firing. Silencing of alerts is also configurable through the web UI for periods of time[viii][ix].

Gitaly

Is a core data remote procedure call service used by Gitlab to read and write Git data. Implements client-server architecture. A Gitaly client is any node that runs a process that makes requests of the Gitaly server. Gitaly clients are also known as Gitaly consumers and include the Gitlab rails application, Gitlab shell, and Gitlab Workhorse to name a few.

GitLab Shell

Responsible for handling Git SSH sessions for Gitlab and modifies the list of authorized keys. Gitlab Shell provides a limited set of predefined Git commands. Example use case is shown below[x].

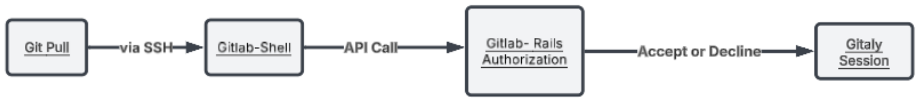


Figure 3 – Authentication and Authorization Flow

GitLab Workhorse

GitLab Workhorse is a program designed at GitLab to help alleviate pressure from Puma. It acts as a reverse proxy used between GitLab Shell or the web client, to funnel requests to Puma, Redis, or Gitaly[xi].

Grafana (Prometheus Output)

Grafana is, at its core, a web UI for displaying real time data in graphic visuals within dashboards. It is highly customizable and integrates with a multitude of data sources. Within GitLab, it handles display of metrics collected by Prometheus which is centered around performance monitoring of the system[xii].

Jaeger (Prometheus Input)

Jaeger was initially developed by Uber Technologies but is now open source. It provides a distributed tracing pattern within the GitLab system by using additional patterns, such as pairing a request ID with a log, to aid in monitoring and collection of important metrics such as message latency[xiii].

NGINX

Webserver that routes requests via HTTP for distributed systems. Additionally, it acts as a reverse proxy, content cache, load balancer, and SMTP mail server among other responsibilities. It connects various functionalities such as authentication, deployment, logging, and configurations for request distribution[xiv].

PostgreSQL

Used for both authentication of users from user information storage, as well as for metadata storage. Leveraged when using Gitlab shell for authentication and allowing users access to the subset of Git commands that Gitlab offers.

Prometheus

Prometheus provides a server which is capable of handling alerts from various configured sources, as well as system monitoring. From the official documentation, "It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts when specific conditions are observed." An interesting aspect of Prometheus is that it uses a multi-dimensional data model, meaning that it joins the concept of a time-series database with key-value pairs or relations from a relational database. It acts as a central authority for alert collection and feeds these alerts to dashboarding or UI tools such as Grafana[xv].

Puma (Gitlab Rails)

Puma is a Ruby application server that is used to run the core Rails Application that provides the user facing features in GitLab. Often this displays in process output as bundle or config.ru depending on the GitLab version[xvi].

Redis

Used for Caching (mostly via Rails.cache), as a job processing queue with Sidekiq, to manage the shared application state, to store CI trace chunks, as a Pub/Sub queue backend for ActionCable, rate limiting state storage, and storing session information. Every application process is configured to use the same Redis server and can be used for inter-process communication when PostgreSQL isn't as appropriate[xvii].

Runner

GitLab CI/CD is the open-source continuous integration service included with GitLab that coordinates the testing[xviii].

Sentry (Prometheus Input)

Sentry is used to collect application-level logs, with the primary goal of aiding developers to detect issues. It is applicable to a multitude of programming languages and has API interfaces to collect programmatic logs in real-time[xix].

Sidekiq

Sidekiq is a Ruby background job processor that pulls jobs from the Redis queue and processes them. Background jobs allow GitLab to provide a faster request/response cycle by moving work into the background[xx]

GitLab High-Level Architecture Diagram

GitLab is a highly configurable system but has a consistent set of core contexts. A typical user interaction is through web browser access, or via SSH. Web browser access allows the user to perform the common tasks of creating "Issue" cards, modifying git repositories, and a multitude of pipeline management functions. SSH communication allows users to authenticate to the system to perform similar functionality, via GitLab CLI. Below is a high-level representation of the system architecture, and the typical usage flow through the application. Further detail of how the various components is connected and their use is provided in reference XXIV. It should be noted that GitLab is a vast and dynamically incorporated system, that fits several use cases. In the various depictions of the architecture or sub-architecture that are shown, variations always exist based on the goals of the implementers.

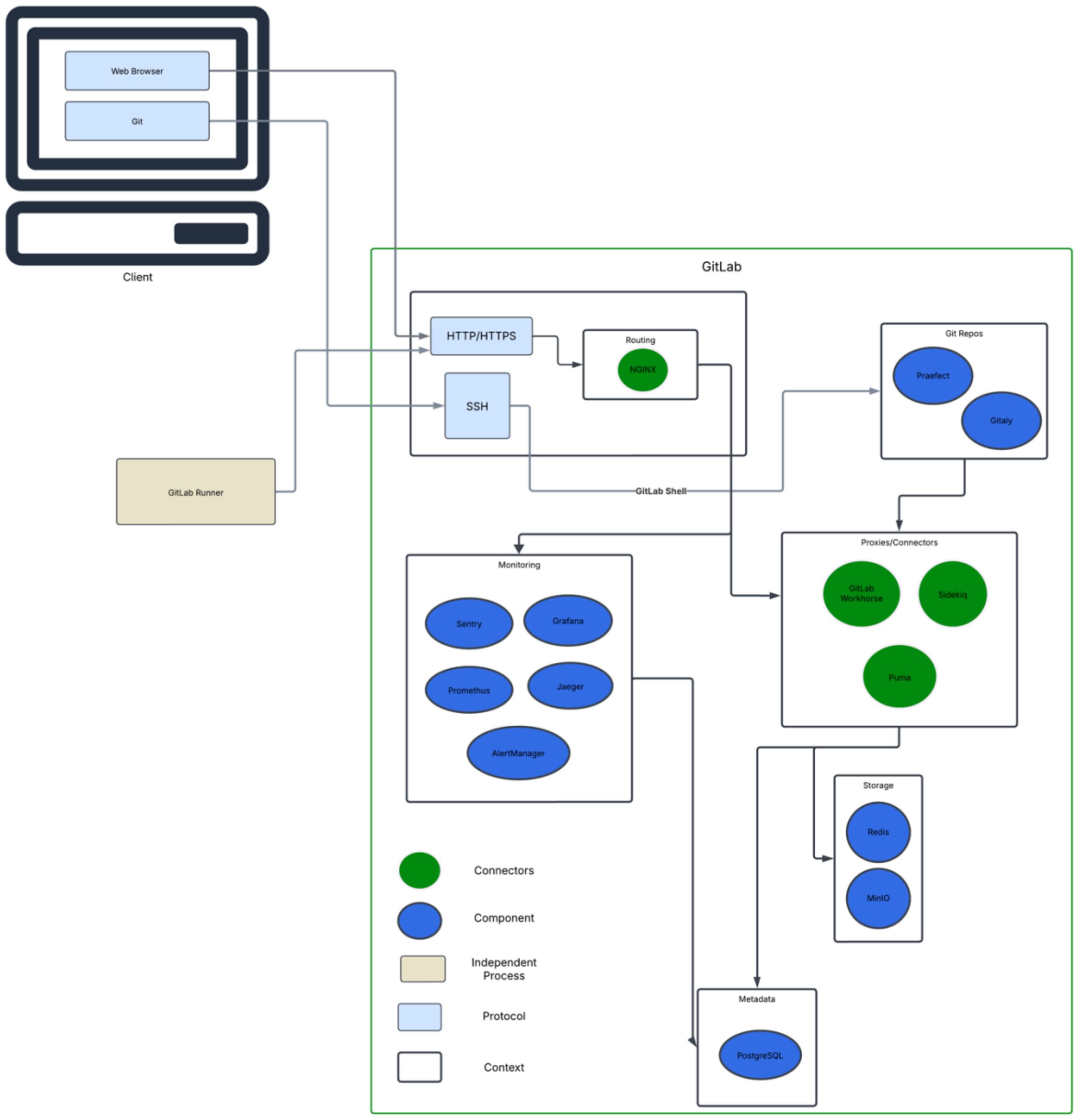


Figure 4 - High Level Architecture

Identification of Dominant Architectural Styles

GitLab provides a highly configurable open-source solution for CI/CD. The core functionality of GitLab can be optionally enhanced by independent plug-ins, to provide not only replication of services, but also net-new functionality. In other words, a typical microkernel architecture.

The microkernel architecture, with a modular core system, is the general concept that GitLab most exhibits. There exist multiple core systems that typically remain statically defined, and various (often optional) plug-in components. The domain functionalities revolve around Monitoring, Internal Data Management, the Code Repositories, and Data Storage. Within the details of these components exist a multitude of architectural patterns that's bounded only by the available open-source options available. This highlights one of the key benefits of GitLab, that updates can be made, and components can be added and removed, affecting isolated core sectors of the system. Note that components shown in the proceeding figures connected to domain components do not represent the domain capabilities in their entirety. A selection was made to illustrate primary capabilities. GitLab can be modeled in various methods for discussion, and a complete component diagram is available via the official documentation in reference XXIV.

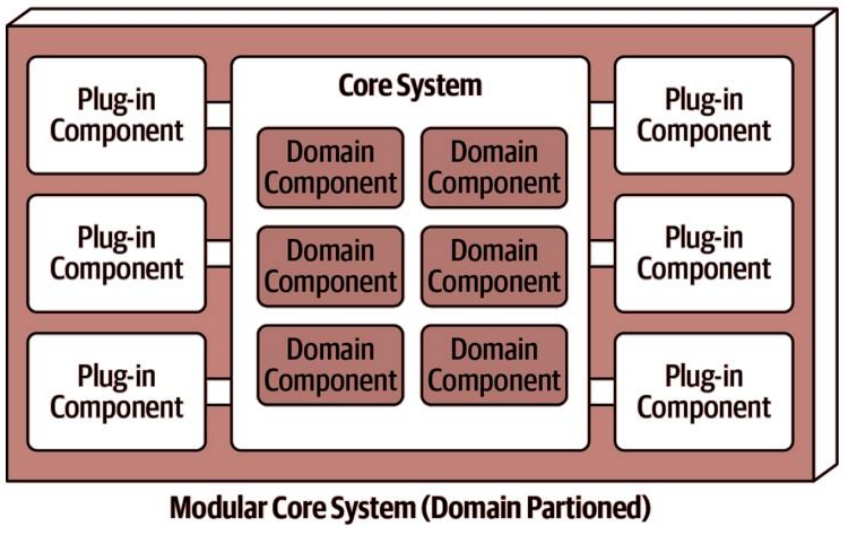


Figure 5 - Architecture Style

Core Component - Gitaly

The Gitaly server is the single source of access to the git repositories, which make up the development process of the resultant applications. Git authentication is handled by Gitaly, and uses LDAP to authenticate the user once, into the entire GitLab system. Once the user is authenticated, their identity is used to authorize access to various functionalities. There exists normally admin and standard user types. As mentioned, access from either users or services within GitLab to the git repositories goes only through Gitaly. Gitaly uses Remote Procedure Call (RPC) communication to provide access to the repositories to these various clients. The repositories themselves are only modified by Gitaly, which references each by their unique path within the system. Overall, Gitaly provides standardized interfaces to allow for the addition of new users and services within the system. Likewise, the network location of repositories is a generally abstract interface

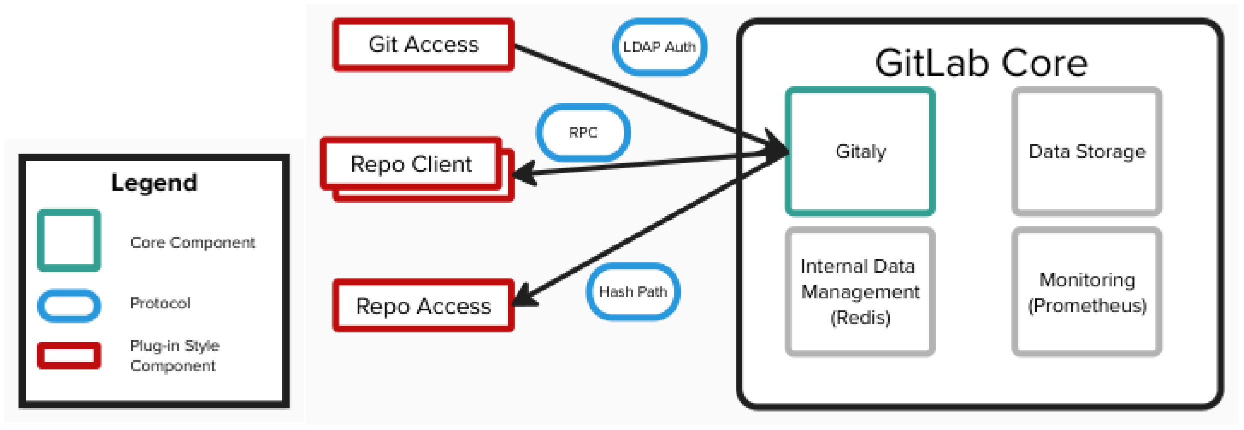


Figure 6 - Core Component: Gitaly

Core Component - Internal Data Management

Considering internal data management as core component, Redis is the primary manager. A single PostgreSQL database is used within this core component for data storage internal to GitLab. The Redis server uses TCP sockets with a generic server-client protocol, which allows clients to modify or query data structures in a shared way. The memory used for database operations as well as the data structures themselves, exist on the Redis server. GitLab primary use cases for Redis include session storage, temporary cache, and background job queues. It handles background job queues for the multi-threaded processor Sidekiq, storing them until they are dequeued. With simple TCP/IP connect ability, it stores shared session state with various stateful services. For ActionCable, the Rails backend and frontend WebSocket manager, it publishes updates with shared data changes to clients on the receiving end of the WebSockets.

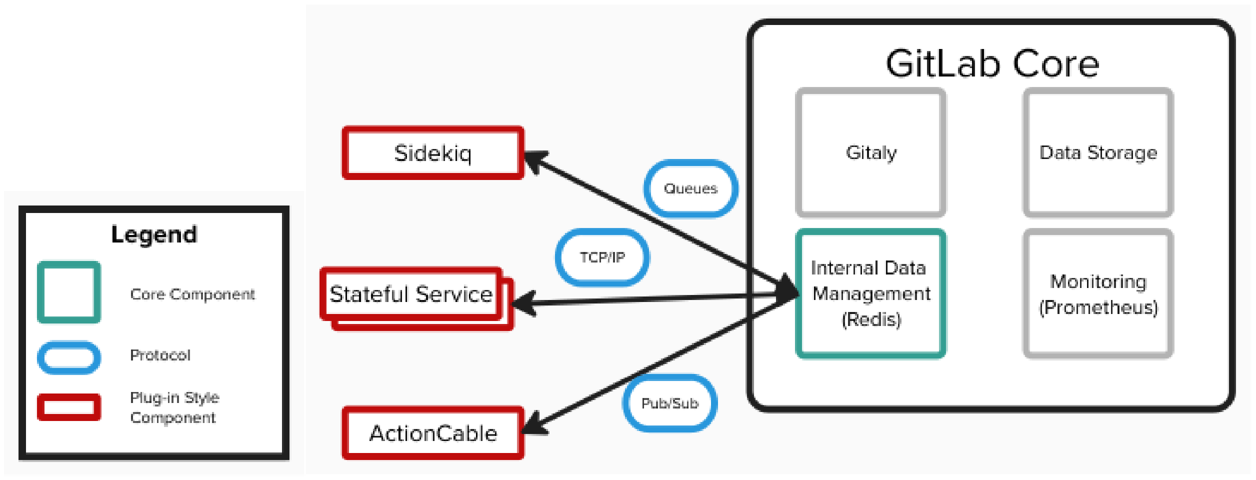


Figure 7 - Core Component: Redis

Core Component - Data Storage

The data storage module of GitLab, utilizes very standard protocols to connect various data stores to the system. The context of these data stores is files, images, repositories and other potentially large resources of the system. MinIO is an object storage server that provides high performance buckets with various additional features[xxi]. GitLab can also use NFS to store persistent data on any number of storage devices. Elasticsearch is a distributed search and analytics engine that integrates with the storage systems within GitLab[xxii].

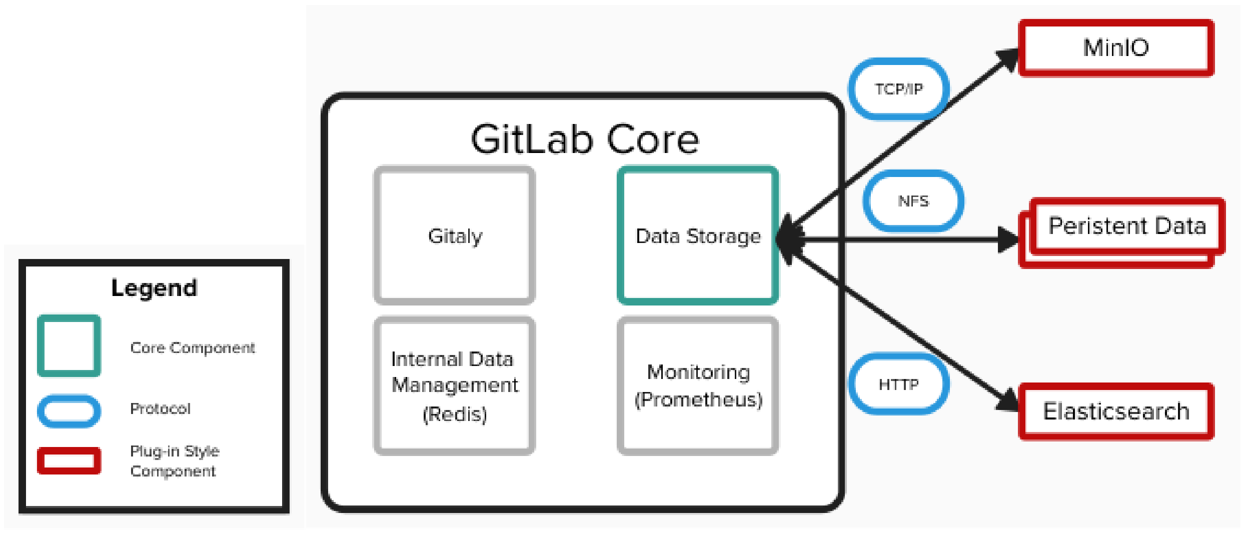


Figure 8 - Core Component: Storage

Core Component - Monitoring

The monitoring engine Prometheus utilizes abstract interfaces that allow various opportunities for logging and acts as a generic alert and metrics manager for various deployable containers. Not only are the input requirements genericized, but more impressively, the output is open for implementation. Typically, a GitLab instance is paired with Grafana for time-based graphical dashboards but, with a generic HTTP PULL model for querying the data, Prometheus as a core component is also configurable for various additional targets. The ability to push alerts based on met conditions is also widely used in GitLab with Alertmanager, which utilizes batch processes via an intermediary gateway. In perhaps the most generic sense, a TCP/IP listener can be connected to receive near real-time updates and display them in popular platforms such as Splunk.

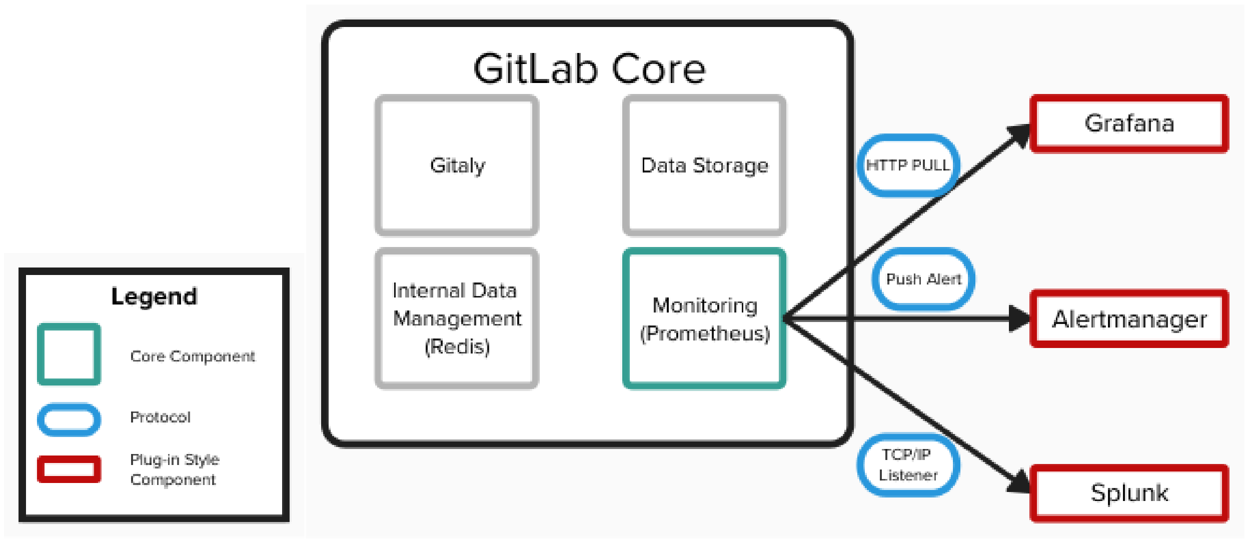


Figure 9 - Core Component: Monitoring

Explanation of Quality Attributes and Architectural Drivers Supported by the Architecture

The core components that make up the main functionality of GitLab are essentially required for the system to operate for its intended purpose, but the modifiability that it has been designed with give a large flexibility to the implementers to experiment with plug-in style components that are even unprescribed by the original architecture. Generic interface contracts are especially prevalent, common protocols such as TCP/IP, and various client-server communication standards such as publisher/subscriber. This presents a system that is adaptable to future needs and adaptable to present day requirements.

A typical microkernel architecture is packaged with a set of "plug-ins" and available as a single package. Though GitLab can be built directly from source code, it offers purchasable editions that come with various sets of extended features which are pre-configured out-of-the-box. For example, considering the components listed above, only a purchased edition from GitLab.com compared to a source installation will include an installed and pre-configured Alertmanager, Grafana, Prometheus, MinIO, Sidekiq, Runner, Redis, Puma, Gitaly and Elasticsearch[xxiii]. The minimum functionality to run the system is quite small.

A great benefit of this architecture style is that various teams can contribute work to solve gaps and vulnerabilities in the system. Being a completely open-source code system, this means countless minds can contribute to components without disrupting not only large contextual domains, but other inter-domain subsystems. Each plug-in component has its own repository and authors and is compiled into a distributable package for GitLab (and in many cases, other frameworks). The harmonization of the system can be literally visualized by the UI presentation layer, Puma. Without delving into the architecture, the user view has architectural transparency to the massive list of features that GitLab provides.

Contrasting GitLab Architecture with a Microservices Architecture

In the previous section we identified GitLab as a microkernel architecture, with a modular core system and pointed out the extensibility benefits of such a choice. In our use cases section, we discussed how configuration changes allow components within GitLab to scale horizontally. Both extensibility and horizontal scaling are the types of quality attributes that cause some organizations to consider a microservices architecture. Additionally, GitLab themselves have made a point to emphasize their choice of a monolithic Ruby on Rails core in contrast to microservices. (https://thenewstack.io/why-were-sticking-with-ruby-on-rails-at-gitlab/) Considering all this, we believe a discussion of the tradeoffs between GitLab's microkernel architecture and microservices would be informative.

We begin by clarifying the distinction between microkernel and microservice architectures. Though a microkernel pattern does make efforts to decouple the components of an application they generally have the following features that contrast them to microservices. Microkernels share a single core code repository and are deployed together in the same package with included plug-ins, microservices are split among different independent repository contexts and are deployed separately. Scaling a microkernel generally requires scaling the entire app or the resources the app has access to. In other words, scaling a microkernel typically involves vertical scaling. On the other hand, scaling microservices can generally be done horizontally. Idealized microkernels are generally compiled into a single executable which means that calls between components are function calls rather than network calls like microservices.

Many of these features of microkernels obviously describe GitLab GCE as we have described it, but some are less clear. For instance, the GCE Rails app runs four processes, is each process a microservice? While GCE runs four processes, they all spawn from a single process and use the same executable; So, no, the multiple processes do not make GCE a microservice. But wasn't it discussed in the use cases section that GCE allows resources like Redis to be scaled horizontally and connected over the network? Yes, configuration changes can be made to scale different storage components of GCE but processing components do not scale horizontally. For instance, a second GitLab Puma cannot be added without a new instance of GCE, and GitLab was designed to use a single PostgreSQL instance for internal data management. The distinction between horizontally scaling storage and processing is a key one. The simplicity of storage interfaces means that splitting storage between different datastores is a far simpler process than splitting processing. Furthermore, microservices are distinguished not just by the ability to add components but the ability to swap them. GCE can add Redis instances but swapping Redis for an alternative like memcached would require code changes to all the components that rely on Redis[xxiv].

Now that the difference between microkernel and microservices is clear, let's discuss the tradeoffs between the two approaches. We have already mentioned one tradeoff, components in a microkernel tend to be more tightly coupled because their interfaces expose more of their internals to other components. As a result, changes to one component more frequently require changes to another. The extent of coupling however is implementation specific and even some microservice implementations can be very tightly coupled. The second tradeoff is deployments, according to GitLab, they average 900 monthly commits to GCE and unless committed work is backing up, deploy about an equal number (https://about.gitlab.com/blog/release-manager-the-invisible-hero/). That is a huge number of changes and releasing them together accounts for the complexity and monthly stress for release managers described in this article. Deploying microservices would limit this massive coupling since each component would have its own dependencies however because GCE is a self-managed application, upgrade tools would need to grow somewhat more complex to manage the deployment of multiple packages to customers' environments. A third major trade off is the speed of inter-component communication. Though we are not certain to what extent this is the case for GitLab, microkernels are more likely to rely on shared memory which avoids the latency hit of communicating over the network.

Limitations of GitLab Architecture

Scalability

A great starting place to discuss the limitations of GitLab's Community Edition's chosen architecture style is GitLab.com. GitLab.com lets users host their code on GitLab's servers but it isn't just a GCE instance that GitLab hosts. It is designed quite differently because it was built to support millions of users. How different you might ask? GCE uses a single Redis instance, while GitLab.com splits state into eleven Redis instances. GCE uses a single PostgreSQL instance while GitLab.com uses Cloud SQL to create an auto-scaling PostgreSQL HA cluster. GCE has a single Gitaly instance for Git storage, while GitLab.com has a Gitaly Cluster with multiple shards. This means that my GitLab.com project code might be stored on a different Gitaly instance than your GitLab.com project. Finally, GCE network security relies on user configured firewall rules while GitLab.com uses Cloudflare. All this scaling is supported by Kubernetes which orchestrates the container creation and service discovery needed for each component to run in containerized workloads that auto-scale and self-heal. The ultimate effect is that even though many of the components are shared, gitlab.com has both higher availability and the ability to horizontally scale up or down according to usage.

Addressing Possible Approaches to Solving Scalability

Achieving the same level of scalability on GCE would be difficult because GCE is built for VMs not Kubernetes pods. For instance, GCE uses systemd to configure its components as services and define service dependencies. New configurations would need to be created if someone were to attempt to set up GCE on Kubernetes. However, some measure of scalability can still be achieved within the original GCE architecture by manually configuring highly available components. All the scaled components could follow a similar pattern, but it should be noted that not all the features of Kubernetes could be achieved. For instance, VMs can restart given a failure but individual apps cannot be restarted without additional software like Kubernetes does. Ultimately, though scalability is possible for GCE, the high degree of manual configuration means it is not a strong suit of the software.

Federated Model

Another limitation of GitLab GCE is that it sends very little data back to GitLab. The only phone home it does is to send version data. This is certainly a feature because many companies value the data protection that self-hosted GitLab provides them. But it is also a limitation because data can help answer many questions about the product. For instance, does the Gitaly service need to get restarted more often in the most recent version? Could a UX change mean that fewer issues are being created per user in the newest version?

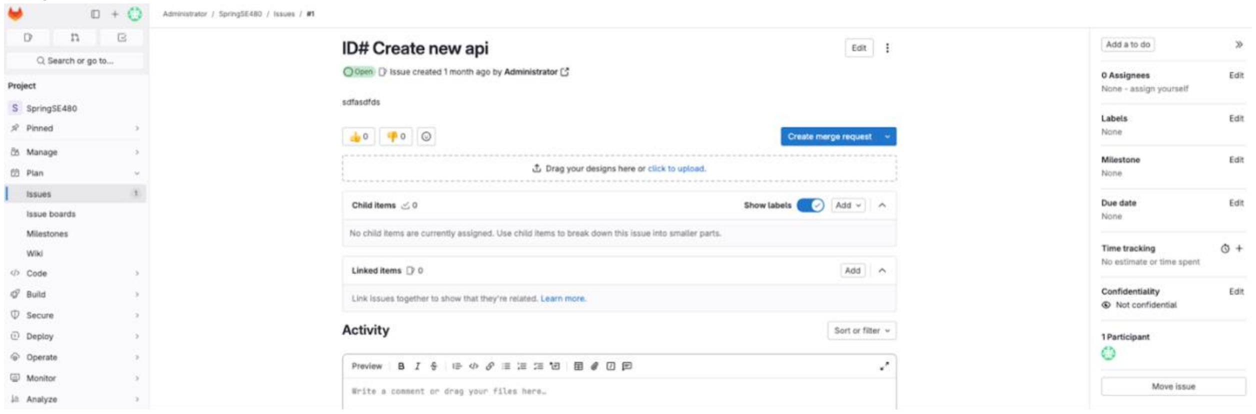
Addressing Possible Approaches to Solving Data Federation

Addressing this is straightforward but not necessarily desirable, GitLab could ask users if they wish to send data back to GitLab to help improve the product. From there GitLab could decide whether to ask for and users could decide whether to share more data like in the "number of issues" example we gave above.

Appendix I: Case Study

For hands-on experience with this research, a GCE was created in a home lab. To learn the system and use the functionalities described in this document, we were able to interact with the actual product and configure many of the described components. Having come from developer backgrounds, and users of CI/CD platforms, we were generally impressed with how much ground the application could cover as a single system.

Our first task was to investigate the project management that GitLab provided. We created git users for ourselves, as admins, and proceeded to create story cards that developers would utilize to communicate business requirements into functional requirements.



ID# Create new api

Open Issue created 1 month ago by Administrator

sdfasdfds

Drag your designs here or click to upload.

Child items 0

No child items are currently assigned. Use child items to break down this issue into smaller parts.

Linked items 0

Link issues together to show that they're related. Learn more.

Activity

Write a comment or drag your files here…

0 Assignees: None - assign yourself

Labels: None

Milestone: None

Due date: None

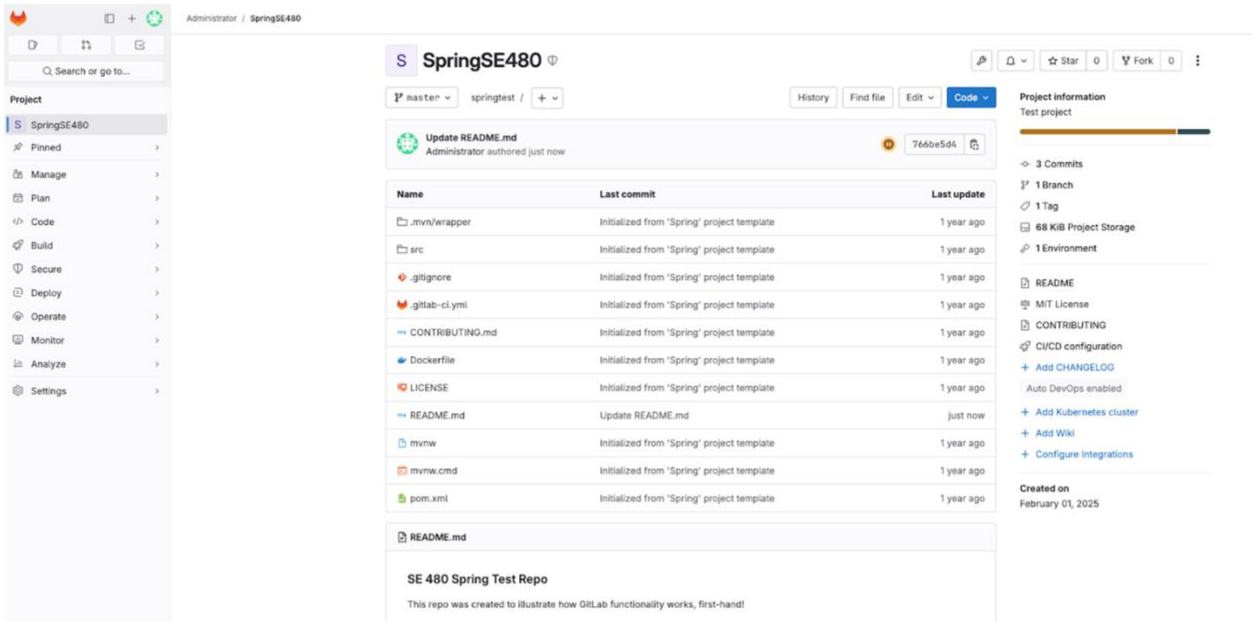
Time tracking: No estimate or time spent

Confidentiality: Not confidential

1 Participant

Figure 10 - Sample Kanban board Issue from GitLab

Next, our task was to create a few services, starting with a Java Spring Boot starter application. We used the Puma UI to generate the repository. The repository was a familiar interface that gave us full git control of the code base.



SpringSE480

Update README.md
Administrator authored just now
766be5d4

Name	Last commit	Last update
.mvn/wrapper	Initialized from 'Spring' project template	1 year ago
src	Initialized from 'Spring' project template	1 year ago
.gitignore	Initialized from 'Spring' project template	1 year ago
.gitlab-ci.yml	Initialized from 'Spring' project template	1 year ago
CONTRIBUTING.md	Initialized from 'Spring' project template	1 year ago
Dockerfile	Initialized from 'Spring' project template	1 year ago
LICENSE	Initialized from 'Spring' project template	1 year ago
README.md	Update README.md	just now
mvnw	Initialized from 'Spring' project template	1 year ago
mvnw.cmd	Initialized from 'Spring' project template	1 year ago
pom.xml	Initialized from 'Spring' project template	1 year ago

Project information
Test project

3 Commits
1 Branch
1 Tag
68 KiB Project Storage
1 Environment

README
MIT License
CONTRIBUTING
CI/CD configuration
Add CHANGELOG
Auto DevOps enabled
Add Kubernetes cluster
Add Wiki
Configure Integrations

Created on
February 01, 2025

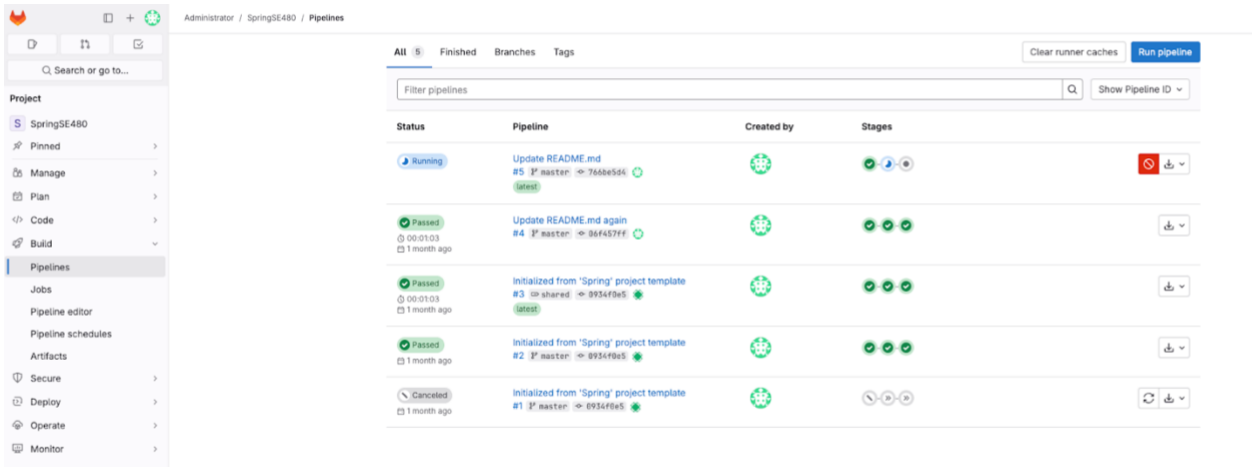
README.md

SE 480 Spring Test Repo

This repo was created to illustrate how GitLab functionality works, first-hand!

Figure 11 - Sample Java SpringBoot Repository View

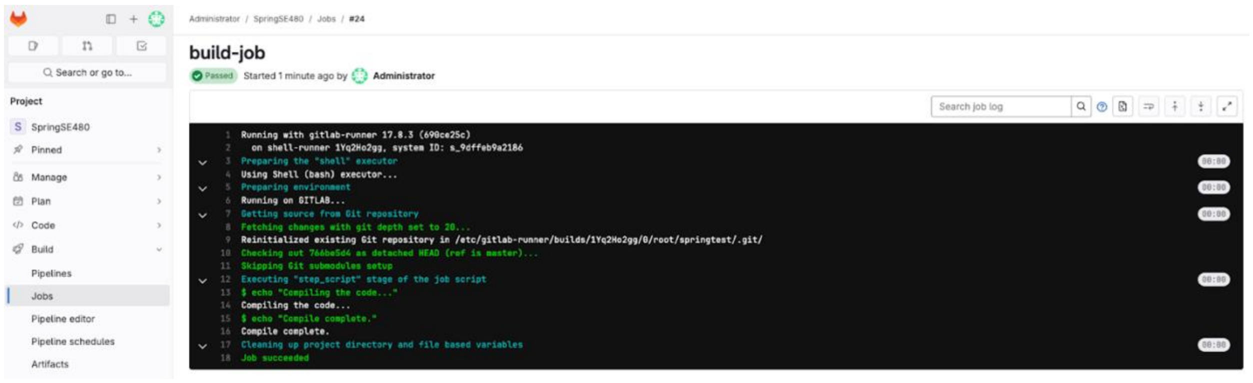
Merged commits were made to the code base, invoking background job queues with Redis, which created tasks to run on the pipeline. We registered a few logically tagged GitLab Runner processes via GitLab Shell to operate on the pipeline tasks.



Status	Pipeline	Created by	Stages
Running	Update README.md #5 master 766be5d4 latest		
Passed 00:01:03 1 month ago	Update README.md again #4 master 06f457ff		
Passed 00:01:03 1 month ago	Initialized from 'Spring' project template #3 shared 0934f0e5 latest		
Passed 1 month ago	Initialized from 'Spring' project template #2 master 0934f0e5		
Canceled 1 month ago	Initialized from 'Spring' project template #1 master 0934f0e5		

Figure 12 - Pipeline view of all runs

We were able to automate the pipeline tasks, including linting, unit testing, and even deploying our compiled jar to a Kubernetes cluster.

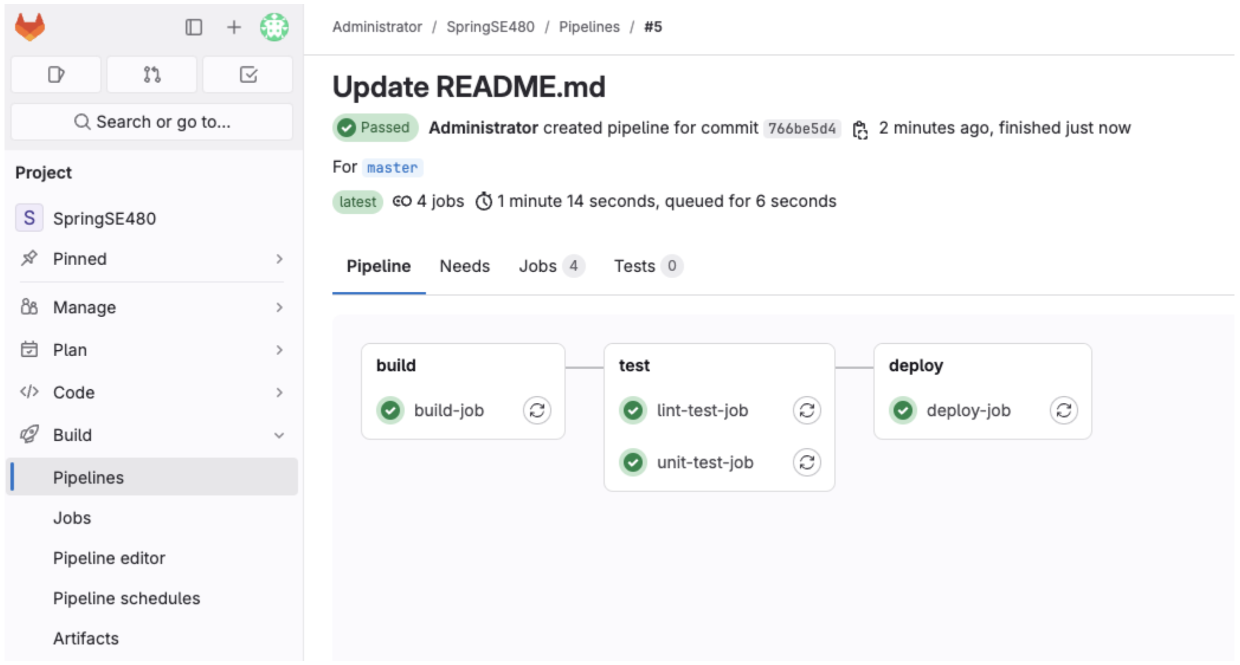

Administrator / SpringSE480 / Jobs / #24

build-job

Passed Started 1 minute ago by Administrator

```
Running with gitlab-runner 17.8.3 (690ce25c)
  on shell-runner 1Yq2Ho2gg, system ID: s_9dffeb9a2186
Preparing the "shell" executor
Using Shell (bash) executor...
Preparing environment
Running on GITLAB...
Getting source from Git repository
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in /etc/gitlab-runner/builds/1Yq2Ho2gg/0/root/springtest/.git/
Checking out 766be5d4 as detached HEAD (ref is master)...
Skipping Git submodules setup
Executing "step_script" stage of the job script
$ echo "Compiling the code..."
Compiling the code...
$ echo "Compile complete."
Compile complete.
Cleaning up project directory and file based variables
Job succeeded
```

Figure 13 - Build task output


Administrator / SpringSE480 / Pipelines / #5

Update README.md

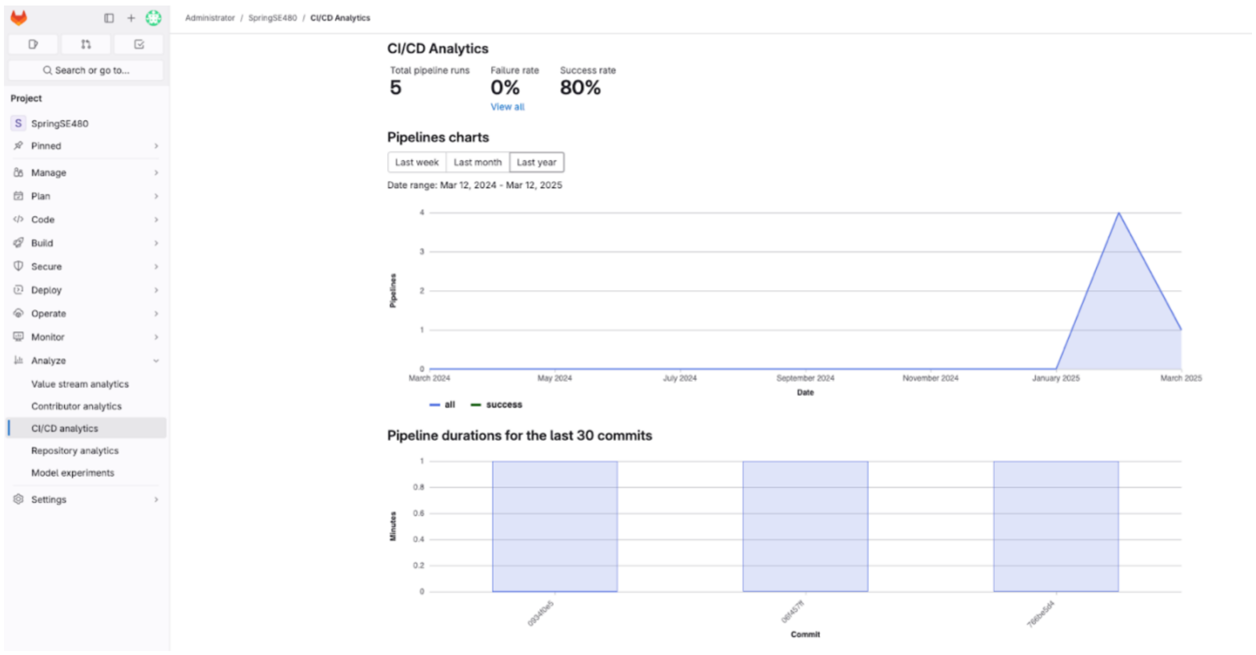
Passed **Administrator** created pipeline for commit 766be5d4 2 minutes ago, finished just now

For master

latest 4 jobs 1 minute 14 seconds, queued for 6 seconds

Figure 14 - Individual pipeline view with task status

Some analytics were available immediately regarding the pipeline, but this was a highly configurable area within GitLab that we did not fully explore.


Administrator / SpringSE480 / CI/CD Analytics

CI/CD Analytics

Total pipeline runs 5

Failure rate 0%

Success rate 80%

Pipelines charts

Date range: Mar 12, 2024 - Mar 12, 2025

Pipeline durations for the last 30 commits

Figure 15 - Sample analytics showing pipeline run statistics

Appendix II: Proposals for Extension of Gitlab

GitLab has grown massively since it was first released in 2011. There are more features and more developers than ever before and maintaining GitLab is far more complex as a result. Managing this complexity within the context like a microkernel architecture is restrictive and why we endorse introducing greater modularity into the GitLab core.

We examined a proposition from a pair of GitLab engineers that within, proposed that the core GitLab Rails application begin to re-organize to a Modular Monolith for a variety of reasons[xxv]. The first is to provide a more isolated structure to the architecture and contain all the features and components it offers into a logical grouping. It borrows concepts from Domain Driven Design and specifically the idea of a bounded context to aid with the distinction between modules. It is mentioned that modules that support external interfaces (SideKiq, REST protocol, Web protocol, GraphQL and ActionCable) would be clearly bounded as a port layer. Similarly, internal application domains are bounded and packaged to decouple unnecessarily intertwined components. Strictly bounding layers and components of the core GitLab Rails application will allow developers to think about fewer components at a time, reducing complexity.

The hope is that this new architectural design allows integrators to work with a smaller set of packages, due to the encapsulation of each domain that is created. They mention other requirements such as having a small and well-documented public interface provided by each domain, as well as the creation of a single-source-of-truth. This single-source-of-truth brings obvious benefit to disparately integrated domains and further promotes the bounded context which they desire.

The team mentioned challenges to this new adaptation, first beginning with the change in mindset that development would need to overcome in order not to fall back on past patterns. Greatly considered is the amount of development work needed to modularize a domain. They mentioned the need for clear guidelines, which is validated by the discussion on domain boundaries. Choices of what is in and what is not in a domain will need to be made and likely at the very end of this, clearly documented interfaces need to be defined for external domain access. This introduces of course the possibility of bad choices, leading to further iterations. Their listed opportunities described a major benefit of this change, that directly addressed our concerns with a system like GitLab. Instead of needing an individual with knowledge of the entire system, it would be likely that experts could exist within each domain. We are unanimously eager and waiting for results of the first iterations. If this is successful, we expect GitLab to become more of an industry standard and architectural example for other systems of its kind.

Appendix III: Evaluation of Architecture Risks and Trade-offs

As we've explored GitLab in this report, it is evident that it is a feature-rich open-core end-to-end software development platform. This open-core model creates trust because developers can verify behavior and buy in because developers know the status of promised new features.

While developers enjoy open source, the fact that anyone can pull request the GitLab GCE means that responsibility for guarding against mistakes and malicious actors falls on GitLab alone. If not managed to perfection, this often means an increased risk of vulnerabilities. Not only are you trusting GitLab to thoroughly test all new code integrations, but also the developers must follow a long list of guidelines when using GitLab's features. In addition, these updates and patches occur frequently, and attention must be given to ensure the security of data.

One such example of this was in 2023 when GitLab exposed the vulnerability CVE-2023-7028, in which an HTTP request associated with GitLab Offers sends a password reset link to an attacker-controlled email, allowing them to reset the password and access that account[xxvi]. It was not disclosed who made this change and whether the vulnerability was malicious or a mistake (though this information is conceivably public record because GitLab is open source we were unable to find a definitive answer). Regardless, GitLab is an obvious candidate for adversarial states to attempt supply chain attacks that can bring down important services. The ability to integrate the attack into the codebase of GitLab and possibly the code bases of users is a massive opportunity for adversarial actors.

Another open source GitLab security concern emerges from the open-source code repositories hosted on the GitLab platform. The attack works by adversarial actors hiding phishing within Gitlab.com pull request comments and issues[xxvii]. While this concern primarily applies to GitLab.com not self-hosted GCE instances where all users are authenticated by sources trusted by a company, GCE users are nonetheless likely to find themselves browsing GitLab.com repositories and exposed to this risk.

Appendix IV: References

[i] GitLab 2025, GitLab CI YAML Instructions
https://docs.gitlab.com/ci/
[ii] GitLab 2025, GitLab Feature Flags
https://docs.gitlab.com/user/feature_flags/
[iii] GitLab 2025, Monitoring: Prometheus
https://docs.gitlab.com/administration/monitoring/prometheus/#sample-prometheus-queries
[iv] GitLab 2025, Transient Prevention Patterns
https://docs.gitlab.com/development/transient/prevention-patterns/#backend
[v] GitLab 2025, GitLab release and maintenance policy
https://docs.gitlab.com/policy/maintenance/
[vi] GitLab 2025, GitLab Shell Command.go
https://gitlab.com/gitlab-org/gitlab-shell/-/blob/main/cmd/gitlab-shell/command/command.go?ref_type=heads
[vii] GitLab 2025 GitLab Shell Shell.go
https://gitlab.com/gitlab-org/gitlab-shell/-/blob/main/internal/command/commandargs/shell.go?ref_type=heads
[viii] Prometheus 2025, Alert Manager
https://prometheus.io/docs/alerting/latest/alertmanager/
[ix] GitLab 2025, Prometheus Alert Manager
https://github.com/prometheus/alertmanager
[x] GitLab 2025, Shell
https://gitlab.com/gitlab-org/gitlab-shell
[xi] GitLab 2025, Workhorse
https://docs.gitlab.com/development/workhorse/
[xii] Grafana 2025, Grafana
https://grafana.com/
[xiii] GitLab 2025, Jaeger
https://github.com/jaegertracing/jaeger/blob/main/README.md
[xiv] Nginx 2025, Nginx
https://nginx.org/
[xv] GitLab 2025, Prometheus
https://github.com/prometheus/prometheus/blob/main/README.md
[xvi] GitLab 2025, Puma
https://gitlab.com/gitlab-org/gitlab/-/blob/master/README.md
[xvii] GitLab 2025, Redis
https://github.com/redis/redis/blob/unstable/README.md
[xviii] GitLab 2025, Runner
https://gitlab.com/gitlab-org/gitlab-runner/blob/main/README.md
[xix] GitLab 2025, Sentry
https://github.com/getsentry/sentry/
[xx] GitLab 2025, Sidekiq

https://github.com/sidekiq/sidekiq/blob/main/README.md
[xxi] GitLab 2025, Minio
https://github.com/minio/minio/blob/master/README.md
[xxii] GitLab 2025, Elastic Search
https://github.com/elastic/elasticsearch/?tab=readme-ov-file#readme
[xxiii] GitLab 2025, GitLab architecture overview.
https://docs.gitlab.com/ee/development/architecture.html
[xxiv] GitLab 2025 Architecture Component Details
https://docs.gitlab.com/development/architecture/#component-details
[xxv] GitLab 2024, Modular Monolith
https://handbook.gitlab.com/handbook/engineering/architecture/design-documents/modular_monolith/
[xxvi] The Register 2024, Vulnerability CVE-2023-7028
https://www.theregister.com/2024/05/02/critical_gitlab_vulnerability/
[xxvii] InfoSecWriteUps 2024, GitHub and GitLab Phishing
https://infosecwriteups.com/dont-trust-links-unveiling-hidden-phishing-threats-in-github-and-gitlab-10bf47548248