Dear GitLab:
A Dev’s Devotion

A technical love letter by Doby

Table of Contents

Introduction 1

GitLab Functional Analysis 2

GitLab vs GitHub Comparison 5

C4 Architecture Diagrams 6

Data Flow Diagrams (DFDs) 9

ERD – Entity Relationship Diagram 12

Use Case Diagram 13

Introduction

This report is not just technical documentation; it’s a personal journey. GitLab has become the heart of my development process: the space where I plan, code, build, and deploy. From my first CI/CD pipeline to publishing real-world projects like Micro TCU-9 and Sentinel Pi, every push, merge, and release has meant more than just code.

This document details GitLab’s platform capabilities, DevSecOps pipeline, data structures, and user interactions through structured diagrams and analysis. It’s my way of saying thank you to the platform that didn’t just support my work but helped shape the developer I’m becoming.

GitLab Functional Analysis

Repository Management

GitLab offers a full-featured Git repository management system that empowers both individuals and teams to collaborate efficiently. It supports all standard Git operations such as clone, commit, push, pull, and branching. GitLab's repository tools provide a reliable foundation for source code control with a powerful web UI that includes:

- **Branch Management:** Developers can easily create, manage, and protect branches. GitLab supports branch protection rules that prevent unwanted changes to critical branches like main or production.
- **Merge Requests (MRs):** GitLab’s merge request system enables code reviews, discussions, and approvals before code is merged. This ensures code quality and collaboration.
- **Commit History and Tags:** Detailed logs and version tracking are maintained with commit messages, change diffs, and tagged releases.
- **.gitignore and File Templates:** GitLab supports reusable file templates, improving consistency across projects.

CI/CD Pipelines

GitLab's built-in CI/CD system is one of its most powerful features. It uses a declarative YAML file (.gitlab-ci.yml) to define automated

workflows that can include multiple stages and jobs such as build, test, and deploy. Key highlights:

- **Ease of Setup:** Built directly into each GitLab project, CI/CD pipelines require no external services.
- **Stages and Jobs:** Workflows can be broken into logical stages with conditional job execution, parallelism, and matrix builds.
- **Runners:** Jobs are executed using GitLab Runners, which can be shared or self-hosted.
- **Deployment Options:** From GitLab Pages to Docker containers and cloud services, GitLab simplifies deployment through automated scripts and environments.
- **Artifact Storage:** CI/CD outputs (like executables, logs, or coverage reports) can be saved and referenced between jobs.

🔄 DevSecOps Lifecycle

GitLab excels as an all-in-one DevSecOps platform, integrating the full software development lifecycle into a single UI. The platform supports every phase of DevOps:

1. **Plan:** Create epics, issues, milestones, and boards to manage agile workflows.
2. **Create:** Use built-in IDE or connect local editors to commit, edit, and version control.
3. **Verify:** Run automated unit tests and code linting via CI/CD.
4. **Secure:** GitLab includes out-of-the-box static and dynamic application security testing (SAST, DAST), dependency scanning, and license compliance.
5. **Package:** Host private container images, npm packages, and Maven artifacts.
6. **Release:** Use pipelines to manage version tagging and release cycles.
7. **Configure:** Environments and Kubernetes integration make configuration seamless.
8. **Monitor:** GitLab integrates with Prometheus and other monitoring tools for performance visibility and error tracking.

This lifecycle integration removes the need for external plugins or manual connections, reducing friction for solo developers and teams alike.

🧩 Integration Tools

GitLab's platform includes auxiliary tools that elevate productivity and collaboration:

- **Issue Tracking:** Advanced issue tracking includes labels, assignees, time tracking, and automation rules.
- **Boards and Milestones:** Visual boards allow teams to plan and track work in real time.
- **Snippets and Wikis:** Code snippets can be shared across projects. Wikis provide rich documentation space per project.
- **Container and Package Registries:** GitLab hosts your Docker images and packages directly within the project.
- **API Access:** GitLab's REST and GraphQL APIs allow for deep automation and customization.
- **Visibility Settings:** Projects can be made public, internal, or private—supporting flexible sharing strategies.

GitLab vs GitHub Comparison

Feature	GitLab	GitHub
CI/CD	Native and built-in CI/CD with YAML config	Requires GitHub Actions setup
Wiki + Snippets	Integrated into each project	Available, but as separate features
Pages Hosting	Built-in GitLab Pages with CI/CD integration	GitHub Pages, limited to static sites
DevSecOps Tools	Integrated SAST, DAST, dependency & license scans	External tools needed or paid plans
Project Visibility	Public, private, internal (granular control)	Public or private
Project Management	Agile boards, milestones, and epics	Boards via Projects (less integrated)
Container Registry	Fully hosted per-project Docker registry	GitHub Container Registry (separate)

Why I chose GitLab: “As a solo dev, I wanted fewer steps, less friction, and full visibility. GitLab gives me all of that in one space. It feels like a lab, not just a repo.”

Diagrams

C4 Architecture Diagrams

- **C4 Level 0 – System Context Diagram**

This diagram provides a high-level overview of how Doby (the developer) interacts with the GitLab platform within their development ecosystem. It identifies external tools and deployed products, clarifying the system's role and boundaries.

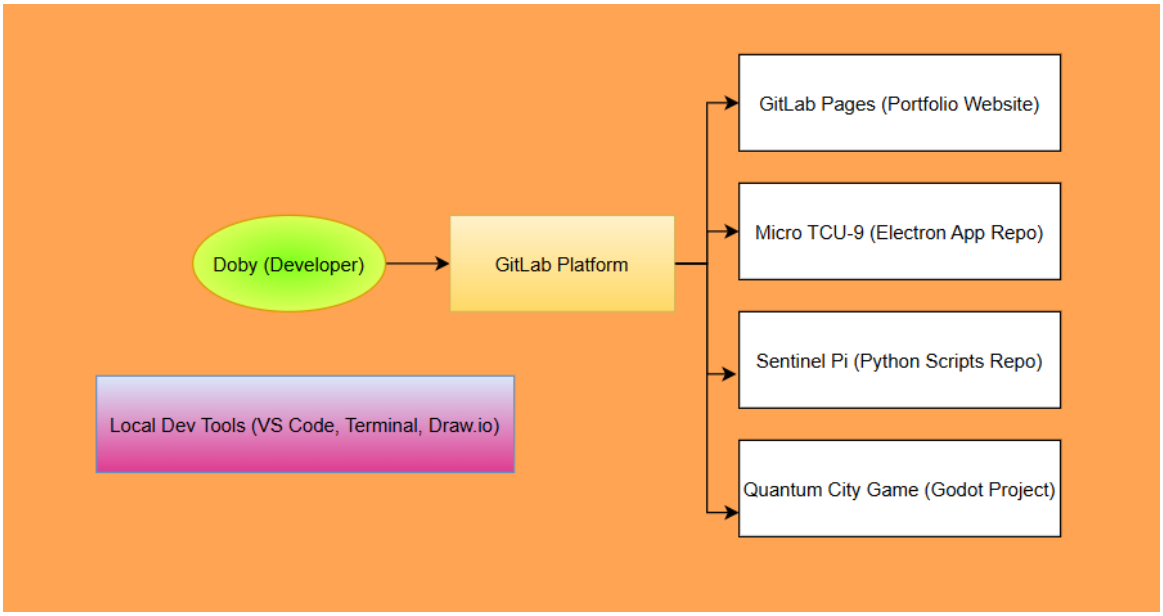
- **Purpose:**

To show how GitLab fits into Doby’s development workflow and what major systems or tools it interfaces with.

- **Key Elements:**

- **Actor:** Doby (Developer)
- **System:** GitLab Platform
- **External Tools:** VS Code, Terminal, Draw.io

- **Outputs:** Deployed projects including:
  - Micro TCU-9 (Electron app)
  - Sentinel Pi (Python scripts)
  - Quantum City Game (Godot)
  - Portfolio site via GitLab Pages



- ## C4 Level 1 – Container Diagram

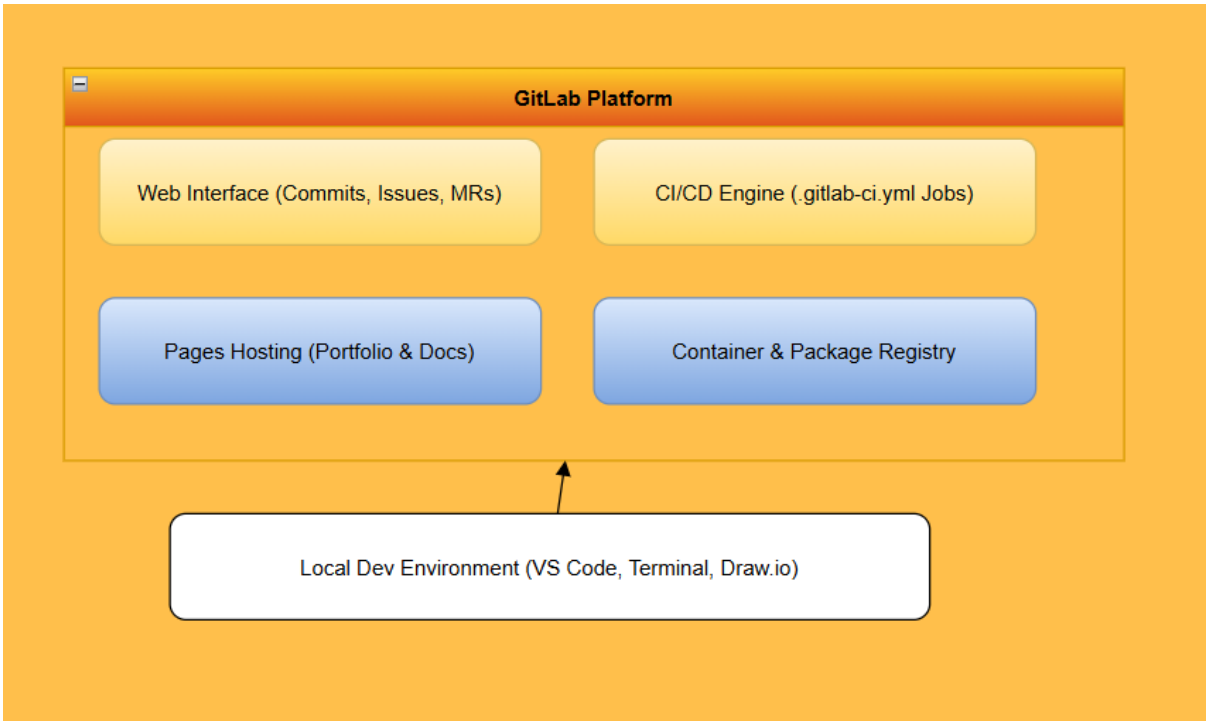
This diagram zooms into the **GitLab platform** to illustrate its major internal containers—essentially, the main services and environments that support development and deployment.

- ## Purpose:

To break down GitLab into functional containers and show how Doby's local development environment connects with each part of the system.

- ## Key Elements:

- **Containers:**
  - **Web Interface**: for interacting with repositories, issues, and merge requests
  - **CI/CD Engine**: where .gitlab-ci.yml pipelines are executed
  - **GitLab Pages Hosting**: deploys portfolio sites and documentation
  - **Container & Package Registry**: hosts Docker images, artifacts, and packages
- **External Input:** Code is authored and pushed via local tools (VS Code, Terminal)



- **C4 Level 2 – Component Diagram (Example: Micro TCU-9)**

This diagram focuses on the internal architecture of the Micro TCU-9 project as structured within its GitLab repository. It illustrates the major components involved in the development, packaging, and delivery of the application — highlighting both GitLab's role and external hosting on Itch.io.

🎯 *Purpose*

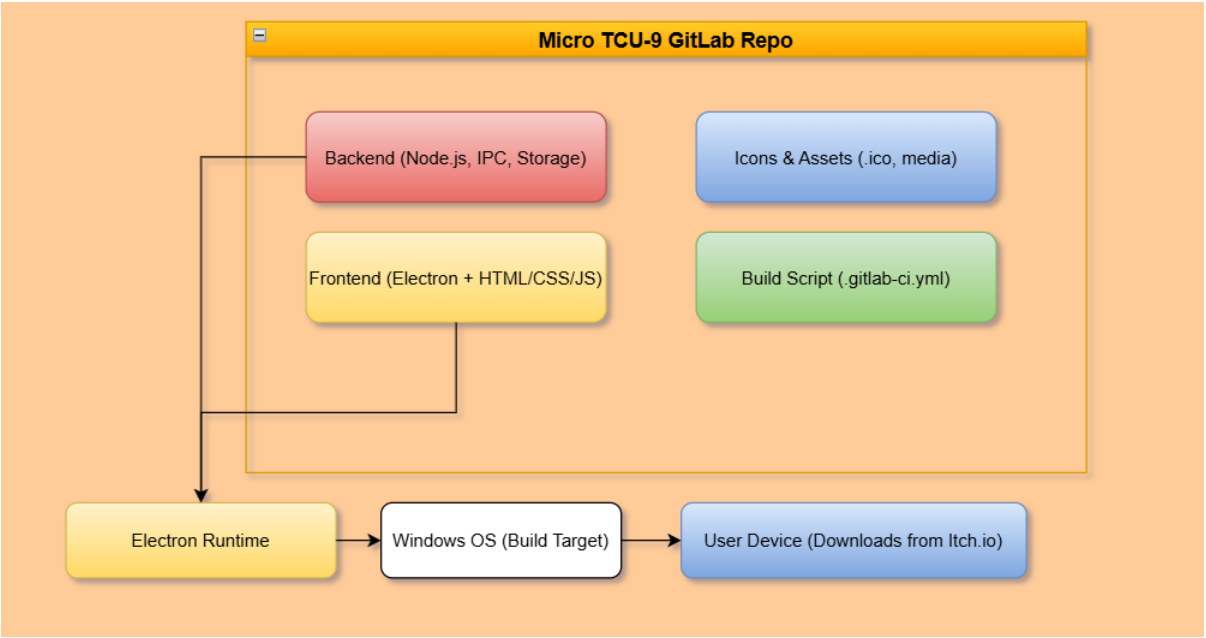
To visualize how Micro TCU-9's source code, build logic, and deployment workflow are organized across GitLab and external platforms.

🔧 *Key Components*

- **Frontend (Electron + HTML/CSS/JS):**
  Manages the user interface, interactions, and visual elements like timers, task boards, and stickers.
- **Backend (Node.js + IPC + Storage):**
  Handles logic processing, communication between modules, and persistent storage for user data and settings.

- **CI/CD Pipeline (.gitlab-ci.yml):**
  Automates validation, basic build preparation, and version tracking. Due to .exe upload limitations, final packaging is done locally and hosted elsewhere.
- **Assets & Icons:**
  Custom visual elements including app icons and media files embedded into the final build.
- **External Runtime:**
  The Electron runtime environment installed on the end user's machine, required to execute the packaged desktop app.
- **Target OS:**
  Designed specifically for Windows as a .exe desktop application.
- **Itch.io Hosting:**
  Serves as the external distribution platform for the packaged .exe, allowing users to safely download the app.
- **End User:**
  Interacts with the application post-download, benefiting from its features and accessibility enhancements.

Note: While GitLab manages the codebase and pipeline automation, the final executable is uploaded manually to Itch.io for distribution due to file upload restrictions.



- ## DFD Level 0 – Context-Level Diagram

This high-level diagram illustrates the overall data flow for the **Micro TCU-9** application. It treats the entire system as a single black-box process and focuses on how information moves between the developer, the development platform (GitLab), and the end users.

🎯 *Purpose*

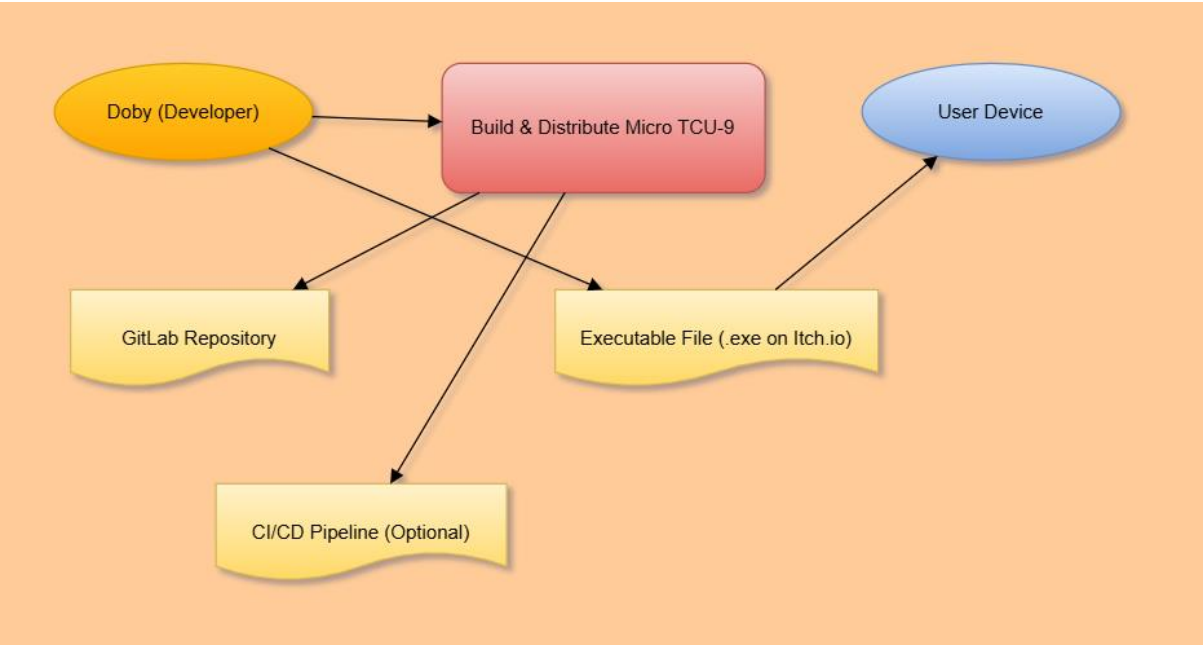
To provide a bird's-eye view of Micro TCU-9's lifecycle — from development to distribution — using a single abstracted process.

🚀 *Key Elements*

- **External Entity:**
  Doby (Developer) – creates and updates Micro TCU-9 code.
- **Process:**
  "Build and Distribute Micro TCU-9" – the central system process that handles development and delivery.
- **Data Stores/Outputs:**
  - **GitLab Repository** – stores the source code and project files.
  - **(Optional) CI/CD Pipeline** – used for building artifacts (not used to host .exe in this case).
  - **Executable File** – manually uploaded and hosted on **Itch.io**.
  - **User Device** – downloads and runs the .exe application.

🔁 *Data Flow Overview*

1. The **developer pushes code** to the **GitLab repository**.
2. GitLab may optionally **trigger a CI/CD pipeline** for builds (not for publishing).
3. The **developer manually packages and uploads** the final .exe to **Itch.io**.
4. **Users download the application** from Itch.io and run it on their local devices.



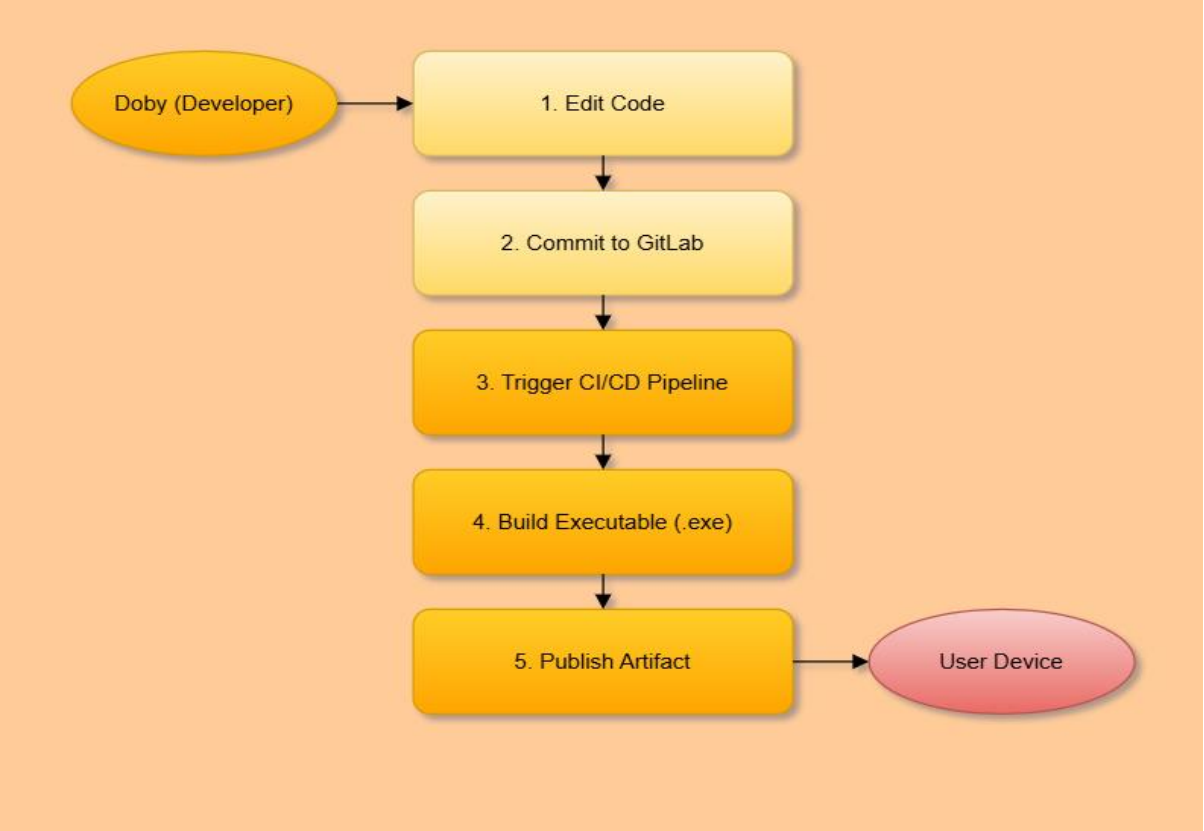
- **DFD Level 1 – High-Level Internal Workflow**

🎯 Purpose: This diagram breaks down the Micro TCU-9 lifecycle into internal sub-processes, focusing on the path from development to deployment.

🧩 Key Processes:

1. **Edit Code** – Doby writes and modifies source code using local tools (e.g., VS Code).
2. **Commit to GitLab** – Code is pushed to the GitLab repository.
3. **Trigger CI/CD Pipeline** – If enabled, .gitlab-ci.yml initiates the pipeline.
4. **Build Executable** – Code is compiled and packaged into a .exe application.
5. **Publish Artifact** – Final .exe is manually uploaded to Itch.io.
6. **User Device** – End users download and run the application.

🔁 Flow: The development process starts with local editing, passes through version control and optional automation, and ends with distribution and user execution.



- **DFD Level 2 – Detailed CI/CD Pipeline Breakdown**

This diagram zooms into the CI/CD Pipeline process (step 3 from Level 1), detailing each task performed during automation that prepares your code for distribution.

🎯 Purpose:

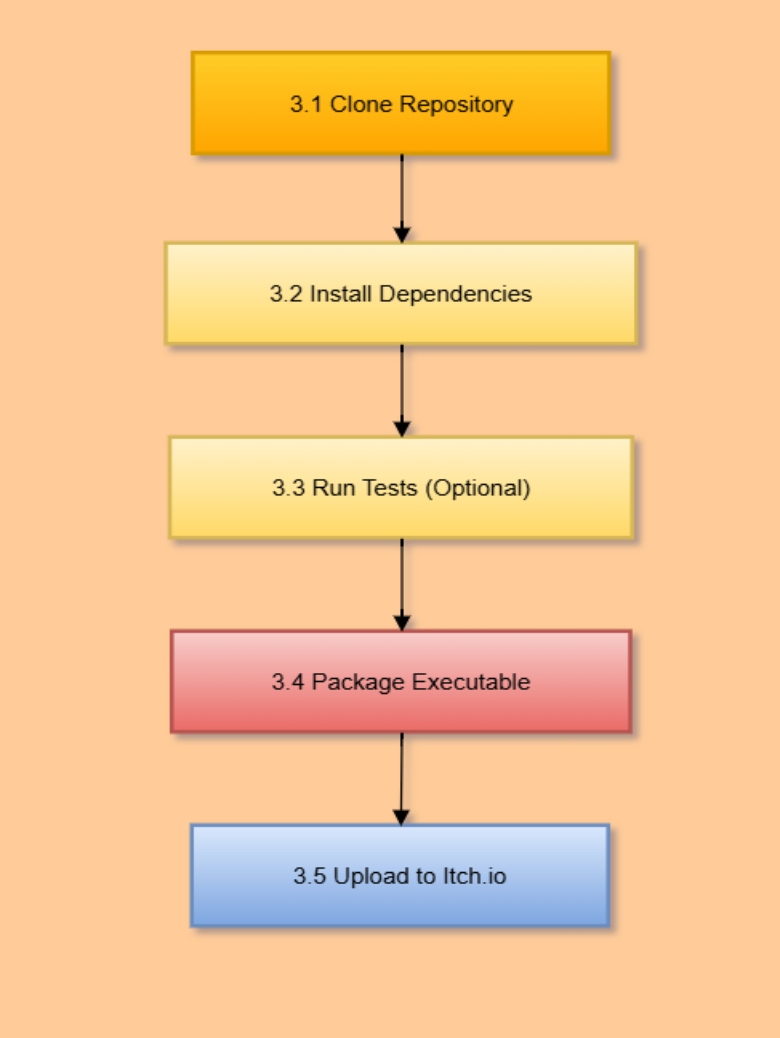
To explain the step-by-step breakdown of the GitLab CI/CD build process as defined in .gitlab-ci.yml, leading up to the artifact upload.

🧩 Key Sub-Processes:

1. **Clone Repository** – Fetches the latest Micro TCU-9 code from GitLab.
2. **Install Dependencies** – Installs required packages (e.g., Node modules, Electron).
3. **Run Tests (Optional)** – Executes any automated tests or linters, if configured.
4. **Package Executable** – Compiles and packages the application into a Windows .exe.
5. **Upload to Itch.io** – Manually upload or via API to make the executable available for users.

🔁 Flow:

Once changes are pushed, GitLab triggers the pipeline; each subprocess outputs to the next, culminating in the executable being uploaded to Itch.io for user download.



- **ERD – Entity Relationship Diagram: GitLab Project Structure**

The **Entity Relationship Diagram (ERD)** represents the core data entities and relationships that structure your GitLab projects and workflows. This model reflects how different parts of a project interact and how they're connected to contributors and processes.

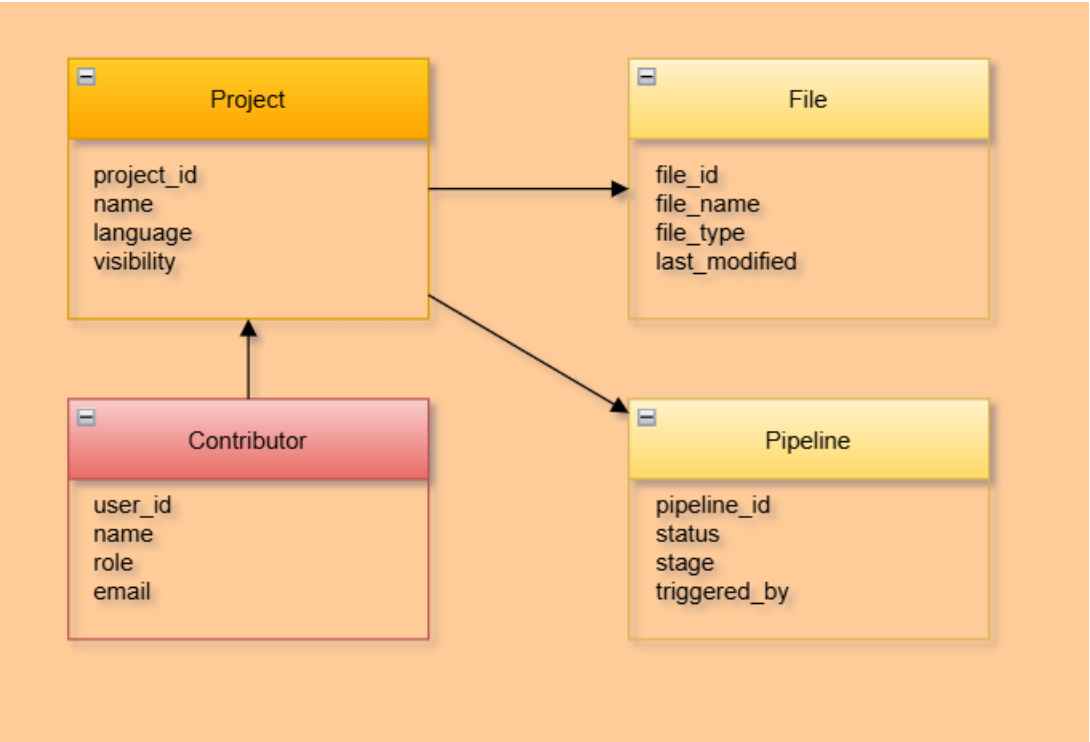
Purpose:

To define and visualize the underlying data model of a typical GitLab-based project – including contributors, files, repositories, and automation workflows.

Key Entities:

- **Project**
  Contains metadata like name, visibility, programming language, and project ID.
  - *Attributes:* project_id, name, language, visibility
- **File**
  Represents individual files within a project (code, assets, config).
  - *Attributes:* file_id, file_name, file_type, last_modified
  - *Relationship:* Each project can contain many files.
- **Contributor**
  Users with access to the project (e.g., Doby, assistants, collaborators).
  - *Attributes:* user_id, name, role, email
  - *Relationship:* Contributors can work on multiple projects.
- **Pipeline**
  Represents automated processes triggered by changes (CI/CD).
  - *Attributes:* pipeline_id, status, stage, triggered_by
  - *Relationship:* Each project can have multiple pipelines.

This ERD lays the groundwork for understanding how development, collaboration, and deployment are managed in a GitLab environment.



- **Use Case Diagram – GitLab Developer Actions**

The **Use Case Diagram** illustrates how the **Developer** interacts with GitLab's functionality throughout the lifecycle of a software project. It focuses on the **user goals and actions** rather than technical structure.

Purpose:

To highlight the specific tasks and interactions that the developer performs within the GitLab ecosystem, aligned with the processes visualized in the ERD and C4/DFD diagrams.

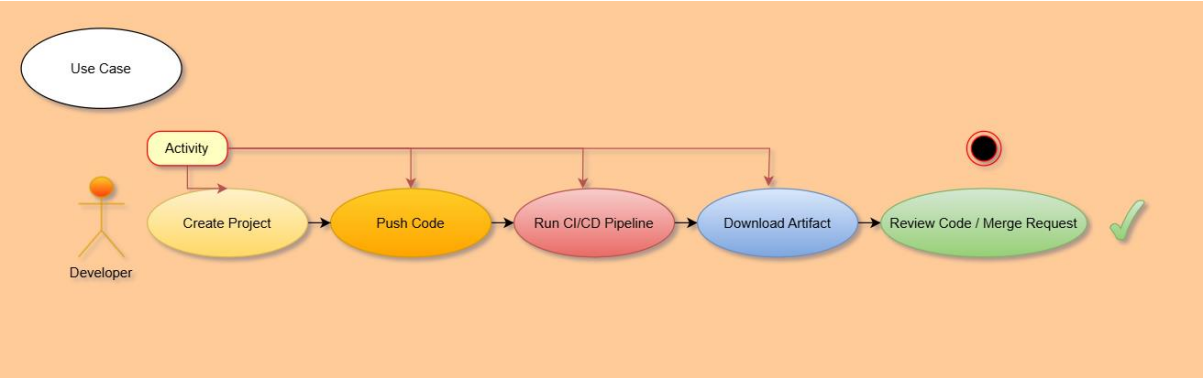
Primary Actor:

- **Developer** (Doby)

Use Cases:

1. **Create Project** – Initiate a new GitLab repository for organizing code and tasks.
2. **Push Code** – Upload files and commit changes using Git.
3. **Run CI/CD Pipeline** – Trigger automated jobs defined in .gitlab-ci.yml.

4. **Download Artifact** – Access and download build outputs (e.g., .exe files).
5. **Review Code / Merge Request** – View, review, and merge contributions via GitLab's collaboration tools.



🗐 Conclusion: What GitLab Taught Me

Through GitLab, I gained more than technical knowledge—I learned how to structure ideas, version my dreams, and deploy my own creations confidently into the world. With each project I launched, GitLab empowered me to:

- Build structure and professionalism into every creative idea
- Understand and trust continuous delivery systems
- Navigate DevOps tools with ease and excitement
- Showcase my skills through real-world portfolio deployments
- Own the full development lifecycle from planning to monitoring

With deepest gratitude,

Doby