AI-04031 GCP vs Git Repository

Owner	Ⓡ Rigel Arcayan
Tags	Project
Created time	July 14, 2026 10:27 AM

Master Prompt: Securely Implement Git-Based Backup and Recovery for My GCP DockerLab

I am a beginner with Google Cloud Platform (GCP) and currently building my cloud engineering skills through a hands-on DockerLab environment.

My current GCP DockerLab includes:

- OpenProject
- OpenEMR
- Docmost
- NGINX Proxy Manager (NPM)
- Portainer
- PostgreSQL
- MariaDB/MySQL
- pgAdmin
- phpMyAdmin
- Docker Compose deployments using YAML (docker-compose.yml) files
- Ubuntu-based server environment

I have recently created a public Git repository account and want to learn the correct DevOps approach for using Git as a secure backup, version control, and disaster recovery system for my DockerLab infrastructure.

Please guide me step-by-step through setting up Git properly for this environment.

My objectives:

1. **Understand Git fundamentals for DevOps**
    - Explain why Git is important for cloud infrastructure management.
    - Explain what files should and should not be stored in Git.
    - Explain Git best practices for Docker, Docker Compose, and GCP environments.
2. **Securely configure my Git repository**
    - Help me decide whether my repository should be public or private.
    - Configure secure authentication using SSH keys or GitHub CLI.
    - Explain how to protect sensitive information.
    - Explain secrets management best practices.
3. **Design my DockerLab repository structure**
   Create a professional folder structure such as:

```
dockerlab/
├── apps/
│   ├── openemr/
│   │   ├── docker-compose.yml
│   │   └── README.md
│   ├── openproject/
│   ├── docmost/
│   └── wordpress/
├── databases/
│   ├── postgres/
│   └── mariadb/
├── proxy/
│   └── nginx-proxy-manager/
├── scripts/
```

├── backups/

├── documentation/

├── .gitignore

└── README.md

Explain how to customize this structure for a production-style DevOps workflow.

1. **Create a secure .gitignore strategy**
   Explain what should never be committed, including:

- .env files
- Database credentials
- SSL certificates
- API keys
- Private SSH keys
- Docker volumes
- Database dumps containing sensitive information

Provide a sample .gitignore file for my DockerLab.

1. **Backup and recovery strategy**
   Design a disaster recovery workflow:

Example scenario:

“My GCP VM is accidentally deleted. How do I rebuild my entire DockerLab?”

Explain:

- What Git restores.
- What backups restore.
- What data needs separate backup storage.
- How to rebuild containers from YAML files.
- How to restore PostgreSQL and MariaDB databases.
- How to restore OpenEMR, OpenProject, and other applications.

1. **Integrate Git with my current GCP workflow**
   Show me how to:

- Connect VS Code on my MacBook to my GCP Ubuntu server.
- Clone my repository.
- Commit changes.
- Push updates.
- Pull changes during recovery.
- Use Git as an Infrastructure-as-Code learning foundation before moving into Terraform.

1. **Create a beginner-to-advanced learning roadmap**
   Build a progression:

Beginner:

- Git basics
- Repository management
- Branches
- Commits
- Pull/push workflow

Intermediate:

- Git workflows for Docker
- CI/CD concepts
- GitHub Actions
- Automated backups

Advanced:

- Infrastructure as Code
- Terraform
- Ansible

- Kubernetes concepts
- Cloud-native DevOps practices

My ultimate goal is to transform my personal GCP DockerLab into a professional DevOps learning environment where I can safely experiment, break things, recover systems, and gradually progress toward Terraform and cloud engineering mastery.

This is a fantastic way to level up. Treating your personal lab like production is the absolute best way to learn cloud engineering. By treating your configurations as code, you are laying the exact foundational mindset needed for Infrastructure as Code (IaC) tools like Terraform.

Let’s break this down step-by-step into a clean, scannable, production-ready blueprint for your GCP Ubuntu server.

---

1. DevOps Git Fundamentals & Strategy

In DevOps, we treat infrastructure the same way developers treat software. This concept is called **GitOps** or **Infrastructure as Code**.

- **Why Git matters for infrastructure:** If your virtual machine (VM) catches fire or gets deleted, you shouldn't have to remember which buttons you clicked in the GCP console or what commands you ran. Git acts as your **Single Source of Truth**. If it's in Git, it can be recreated.
- **The Golden Rule of Git:** Git stores **blueprints (declarative code)**, never **state (data)** or **secrets (credentials)**.

What Goes in Git vs. What Stays Out

Store in Git (Blueprints)	Keep OUT of Git (State & Secrets)
`docker-compose.yml` files	`.env` files (containing actual passwords)
Custom `nginx.conf` or proxy rules	Persistent volume data (`/var/lib/docker/volumes`)
Automation scripts (`.sh`, `.py`)	Database dumps (`.sql`, `.tar`)
Sample environment files (`.env.example`)	SSL Private Keys (`.key`)

Store in Git (Blueprints)	Keep OUT of Git (State & Secrets)
Documentation (`README.md`)	API Tokens / Service Account Keys

2. Secure Repository Configuration

> **Critical Warning:** Change your repository visibility to **Private** immediately. While you can use a public repo with a pristine `.gitignore`, a single accidental commit of an unencrypted `.env` file can expose your database credentials to automated scrapers within seconds.

Setting Up Secure SSH Authentication

Instead of typing passwords, you will use an SSH keypair to securely authenticate your GCP Ubuntu server with GitHub. Run these commands on your **GCP Ubuntu VM**:

```
# 1. Generate a secure Ed25519 SSH key
ssh-keygen -t ed25519 -C "your_email@example.com"

# 2. Start the ssh-agent in the background
eval "$(ssh-agent -s)"

# 3. Add your SSH private key to the agent
ssh-add ~/.ssh/id_ed25519

# 4. Display the public key - COPY THIS OUTPUT
cat ~/.ssh/id_ed25519.pub
```

Next Steps on GitHub:

1. Go to your GitHub account settings → **SSH and GPG keys** → **New SSH Key**.
2. Paste the public key output and name it something clear (e.g., `GCP-DockerLab-Ubuntu`).

3. Designing the Repository Structure

Here is how to structure your repository to mirror a professional microservices layout. Each application gets its own isolated stack directory, keeping dependencies clean.

```
dockerlab/
├── .gitignore
├── README.md
├── apps/
│   ├── openemr/
│   │   ├── docker-compose.yml
│   │   └── README.md
│   ├── openproject/
│   │   └── docker-compose.yml
│   └── docmost/
│       └── docker-compose.yml
├── databases/
│   ├── postgres/
│   │   └── docker-compose.yml
│   └── mariadb/
│       └── docker-compose.yml
├── proxy/
│   └── nginx-proxy-manager/
│       └── docker-compose.yml
├── scripts/
│   ├── backup-db.sh
│   └── restore-db.sh
└── documentation/
    └── architecture.md
```

Production Best Practice: Decoupled vs. Unified Stacks

As a beginner, keeping everything inside separate directories under `apps/` or `databases/` is ideal. It allows you to update, restart, or troubleshoot **OpenEMR** without touching your global **PostgreSQL** or **NGINX Proxy Manager** instances.

To link them together across different directories, create a shared external Docker network on your VM:

```
docker network create proxy-tier
```

In your individual `docker-compose.yml` files, you can join this network using:

```yaml
networks:
  proxy-tier:
    external: true
```

---

4. A Secure `.gitignore` Strategy

Create a `.gitignore` file at the root of your `dockerlab/` folder. This acts as an automated shield preventing sensitive data from leaving your server.

Production-Grade `.gitignore` Template

```
# Ignore all environment file containing actual secrets
*.env
.env.*
!.env.example

# Ignore local database storage and persistent volume directories
**/data/
**/db/
**/postgres_data/
**/mysql_data/
**/volumes/

# Ignore security assets
*.pem
*.key
*.crt
*.p12

# Ignore database backups and dumps
*.sql
*.sql.gz
*.tar.gz
backups/

# Ignore OS-specific metadata
.DS_Store
Thmubs.db
```

Handling Secrets: The `.env.example` Pattern

Since your real `.env` file is ignored, commit an `.env.example` file right next to your `docker-compose.yml` with placeholder values:

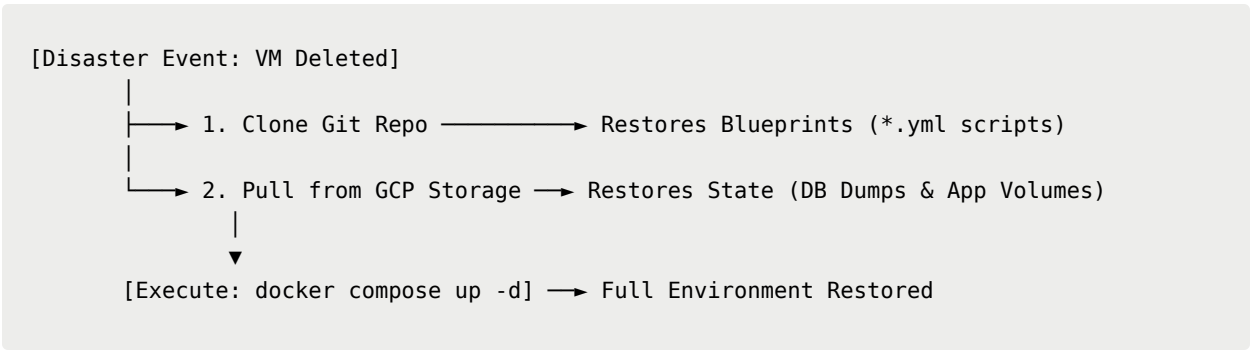
```
# .env.example - Commit this to Git
DB_USER=openemr_user
DB_PASSWORD=REPLACE_ME_WITH_A_SECURE_PASSPHRASE
DB_NAME=openemr_prod
```

When restoring your system, you simply copy the template (`cp .env.example .env`) and populate the real secrets on the server manually.

---

5. Backup and Recovery Strategy

If your GCP Compute Engine instance is accidentally deleted, you face two distinct recovery recovery phases: **Configuration Recovery** (via Git) and **State Recovery** (via Object Storage).



Division of Labor

- **What Git Restores:** Your structure, multi-container configurations (`docker-compose.yml`), reverse proxy rules, and automation scripts.
- **What Backups Restore:** The raw rows in your databases, user-uploaded medical records in OpenEMR, project attachments in OpenProject, and SSL certificates generated by NPM.
- **Where Backup Data Belongs: Google Cloud Storage (GCS) buckets**. Never store backups on the local boot disk of the VM they are backing up.

Step-by-Step Disaster Recovery Walkthrough

Step 1: Spin up a fresh GCP Ubuntu VM & Install Docker

Ensure your user belongs to the Docker group so you can execute commands without `sudo`:

```
sudo usermod -aG docker $USER && newgrp docker
```

Step 2: Clone your Blueprint Infrastructure

```
git clone git@github.com:yourusername/dockerlab.git
cd dockerlab
```

Step 3: Rehydrate Environment Variables

Navigate to your target application directory, copy your committed templates, and fill in your secrets:

```
cd apps/openemr
cp .env.example .env
nano .env
```

Step 4: Restore the Databases (State)

Before launching the application frontend, you must bring up your database containers blank, then stream your backup data back into them.

For PostgreSQL (OpenProject / Docmost):

```
# 1. Start just the database container
docker compose up -d postgres-db

# 2. Stream your compressed local backup file back into the d
atabase engine
```

```
gunzip -c /path/to/backup/postgres-backup.sql.gz | docker exe
c -i postgres-db psql -U postgres_user -d target_db
```

For MariaDB/MySQL (OpenEMR):

```
# 1. Start just the database container
docker compose up -d mariadb-db

# 2. Stream your compressed backup file back into MySQL
gunzip -c /path/to/backup/mysql-backup.sql.gz | docker exec -
i mariadb-db mysql -u mysql_user -p'your_password' target_db
```

Step 5: Launch the Remaining Services

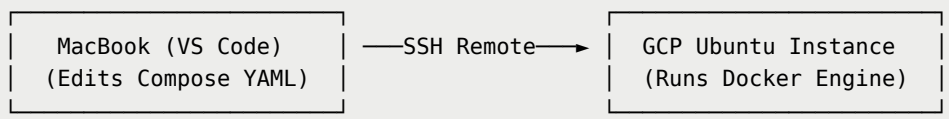
Once the databases are populated, launch the rest of the application ecosystem:

```
docker compose up -d
```

---

6. Integrating Git with Your GCP Workflow

To comfortably manage your environment, you can use your MacBook as your control center while executing the heavy lifting directly on your cloud hardware.



1. **Configure VS Code Remote Development:** On your Mac.
   Install the official **Remote - SSH** extension marketplace pack by Microsoft.
   Open your local Mac terminal and configure your SSH access file at `~/.ssh/config`:

```
Host gcp-dockerlab
    HostName YOUR_GCP_VM_EXTERNAL_IP
```

```
    User your_gcp_username
    IdentityFile ~/.ssh/id_ed25519
```

1. **Establish Connection and Clone:** Inside VS Code.
   Click the green indicator icon in the bottom-left corner of VS Code, choose **Connect to Host...**, and select `gcp-dockerlab`. Open a new folder targeting your home directory (`/home/your_username/`) and clone your private repository directly within that cloud window.
2. **The Iterative Lifecycle Loop:** Daily Operations.
   When adding a new tool (like WordPress), open your local VS Code terminal panel (which runs directly inside the VM environment) and execute your standard Git cadence:

```
# Create changes locally on the server
git status
git add apps/wordpress/
git commit -m "feat: integrate wordpress service stack to pro
xy tier"
git push origin main
```

---

7. The DevOps Learning Roadmap

This logical progression transforms your self-hosted lab environment into a enterprise-ready cloud engineering resume piece.

🟩 Beginner: Git & Container Basics

- Master working with single feature branches (`git checkout -b feature/add-phpmyadmin`).
- Learn the fundamental Docker commands (`docker compose logs -f`, `docker compose down`, `docker compose ps`).
- Successfully back up your data volumes off-instance using basic standard Linux toolsets (`tar`, `cron`, `scp`).

🟨 Intermediate: Automation & CI/CD

- **GitHub Actions:** Automate syntax checking of your `docker-compose.yml` files every time you push code to GitHub to catch formatting issues early.
- **Automated Cloud Backups:** Write a bash script that triggers a nightly database dump via `cron`, wraps it up, and pushes it directly to a secure **GCP Cloud Storage (GCS) bucket** using the `gcloud storage` CLI utility tool.

🟦 Advanced: Production IaC & Cloud Native

- **Terraform Integration:** Stop building your GCP VMs via the browser interface click-dashboard. Write declarations that define your VPC networks, firewalls, and Compute instances programmatically.
- **Ansible Configuration Management:** Use Ansible playbooks to automatically connect to a freshly built base VM, install Docker, adjust kernel security settings, and pull your Git lab components without executing instructions manually.
- **Kubernetes (K8s) Migration:** Migrate standalone Compose stacks over into a local single-node cluster environment (like K3s or minikube) using native Pod, Deployment, and Service definitions.