STEP-BY-STEP SETUP GUIDE

GitLab CI/CD on Proxmox VE

Turn a single-node Proxmox VE homelab into a complete GitLab CI/CD platform: one Debian 12 cloud-init template, a Docker runner VM, and an app VM that deploys itself on every push.

Debian 12 · Docker executor · docker compose + nginx

October 2026 · Version 1.0

Table of Contents

1. Overview and Architecture . . . 1

2. Prerequisites and Planning . . . 2

3. Prepare the GitLab TLS Certificates . . . 6

3.1 Optional: issue your own certificate on the GitLab VM . . . 6

3.2 Keep the CA ready for the template . . . 7

4. Prepare the SSH Keys . . . 8

5. Create the Runner in GitLab . . . 10

6. Build the All-in-One Template . . . 10

6.1 Creating the template VM on the host . . . 10

6.2 Filling it once: agent, Docker, TLS trust and the kit scripts . . . 11

6.3 Converting to a template and verifying . . . 12

7. Cloning the Virtual Machines . . . 13

8. Finish the Runner VM: Install and Register . . . 14

8.1 Installing the runner package . . . 14

8.2 Registering the runner against your instance . . . 14

9. Preparing the Deployment VM . . . 15

9.1 The CI deploy key from chapter 4 . . . 15

9.2 Run the preparation script . . . 16

9.3 Place the compose project . . . 16

10. CI/CD Variables and Secrets . . . 17

11. The Pipeline Explained . . . 18

11.1 Path-based triggering . . . 18

11.2 The shared SSH bootstrap . . . 18

11.3 The container track . . . 18

11.4 The static track . . . 19

12. First Run and Verification . . . 19

13. Maintenance and Troubleshooting . . . 20

13.1 Troubleshooting reference . . . 20

13.2 Keeping the platform healthy . . . 22

1. Overview and Architecture

This guide turns a single-node Proxmox VE homelab into a small but complete GitLab CI/CD platform. By the end you will operate two Debian 12 virtual machines built from one cloud-init template: a **runner VM** that executes GitLab CI jobs inside isolated Docker containers, and an **app VM** that automatically receives deployments - a containerized Flask API served on port 8000 through docker compose, plus a static site served by nginx on port 80. GitLab itself is not touched: your existing standalone instance runs as a virtual machine on the same Proxmox host and acts as the repository, the container registry and the pipeline brain - the guide only registers a runner against it and consumes its registry. Every file referenced here ships in the accompanying `gitlab-cicd-proxmox` kit, so most chapters come down to running a prepared script and understanding what it does.

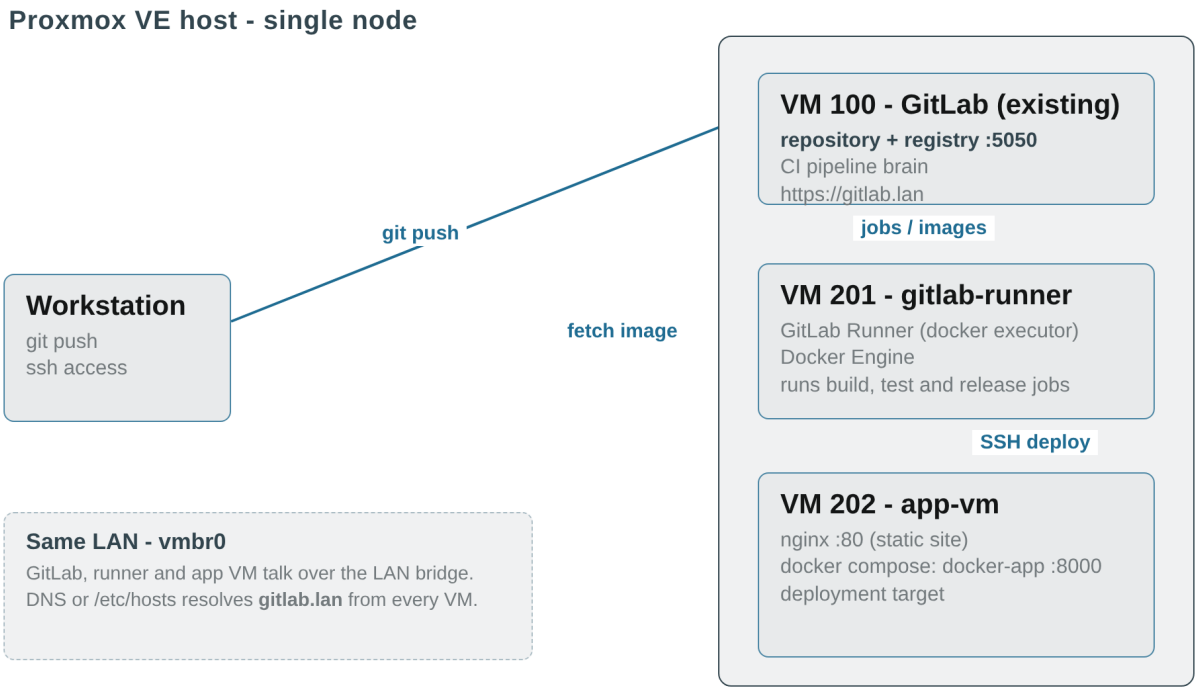


Figure 1: Target architecture - one Proxmox host hosting the existing GitLab VM, the runner VM and the app VM.

The flow works as follows. You push code to GitLab; the pipeline definition (`.gitlab-ci.yml`) travels with the repository. The runner VM picks up jobs, and each job runs in a fresh Docker container chosen per job (python, node, docker, alpine). Release jobs build a Docker image and push it to the GitLab container registry. Deploy jobs then reach the app VM over SSH: the containerized app is refreshed with `docker compose pull && docker compose up -d`, and the static site is synchronized with `rsync --delete` into the nginx web root. Nothing on the app VM is built by hand - it only ever reflects what the pipeline shipped.

Component	Role	Where it lives
GitLab VM (existing)	Repository, CI pipeline engine, container registry (port 5050), secrets store	Proxmox VM on this node, e.g. gitlab.lan
Runner VM (201)	GitLab Runner service with docker executor; executes every job in a disposable container	Proxmox VM, Debian 12
App VM (202)	Deployment target: nginx for the static site, docker compose for the containerized app	Proxmox VM, Debian 12
Proxmox VE host	Runs both VMs, holds the reusable template, provides snapshots and backups	Your homelab node
Workstation	git client, SSH key pair management, first-time configuration	Your laptop or admin box

Table 1: Components of the setup and where each one lives.

Two independent delivery tracks run side by side in one pipeline. Jobs trigger only when files in their own application folder change, so touching the static site never rebuilds the container image and vice versa. The container track runs pytest, builds and pushes the image, then deploys via docker compose over SSH. The static track builds the Vite bundle, keeps it as a pipeline artifact and rsyncs it to the nginx web root. Release and deploy jobs are gated to the default branch, while test and build jobs also run on merge request branches.

2. Prerequisites and Planning

The setup adds two small VMs to a host that already runs your standalone GitLab instance. Before starting, make sure the host itself is healthy: Proxmox VE installed (version 8.x recommended), root SSH access to the host web UI and shell, and enough free capacity for the GitLab VM that is already there plus the two new VMs and the template. All of them can use the same storage pool; the examples in this guide use `local-lvm` and the default `vmbr0` bridge.

Machine	vCPU	RAM	Root disk
GitLab VM (existing)	4	8 GiB (4 min.)	40-60 GiB
Template VM 9000	2	2 GiB	20 GiB
Runner VM 201 (gitlab-runner)	2	4 GiB	20 GiB
App VM 202 (app-vm)	1	1-2 GiB	10-20 GiB
Host headroom (recommended)	-	>= 8 GiB free	>= 60 GiB free

Table 2: Resource plan. The GitLab VM row is your existing installation - listed so you can check the host total.

Network requirements are minimal but worth checking up front, because most first-run failures in this kind of setup are DNS or firewall related rather than configuration mistakes:

- Every VM must resolve the GitLab hostname (`$GITLAB_HOST` in the variable block below; example gitlab.lan) - via LAN DNS or `/etc/hosts` entries - and reach it on port 443 (API, clones) and 5050 (registry).
- The runner VM additionally needs internet access: `packages.gitlab.com` for the runner package and Docker Hub for job images (python, node, docker, alpine).
- The runner VM needs outbound SSH (port 22) to the app VM for deployments.
- Your workstation needs SSH to all VMs and HTTP to the app VM (ports 80 and 8000).
- The clone script configures static addresses (recommended - pass an IP/prefix and gateway to it); DHCP on `vmbr0` also works - clones carry the guest agent baked into the template, so Proxmox shows their lease either way.

> **TLS in the homelab.** Self-hosted instances usually run either plain HTTP or a certificate from a private CA (or a self-signed certificate). Both work, but a private CA has to be trusted by every client that talks to the instance. Chapter 3 settles the certificate question before anything is built, and chapter 6 bakes the resulting trust into the template once - no VM ever needs per-VM certificate plumbing. Plain HTTP needs no certificates but requires marking the registry as insecure for Docker.

Software	Version used here	Notes
Proxmox VE	8.x	Single node; commands run as root on the host
Debian 12 generic cloud image	bookworm	Includes cloud-init and the sudo-enabled "debian" user, but NO qemu-guest-agent (script 03 adds it while filling the template)
Docker Engine	27.x	From download.docker.com; includes buildx and the compose plugin
GitLab Runner	16.x / 17.x	Uses the glrt- runner authentication token flow
git + OpenSSH client	any recent	On your workstation only

Table 3: Software components and versions the guide was written against.

Nothing in this guide is hard-coded, because no two homelab networks look alike. Every value that varies - addresses, hostnames, URLs, VMIDs - lives in one of two kinds of placeholders. **Shell variables** (written `$LIKE_THIS`) appear in commands meant to be pasted into a terminal: set them once per shell with the block below, on your workstation and on the Proxmox host, and every command in this guide runs exactly as printed. **Angle placeholders** (written `<like-this>`) mark values that only exist at runtime - the template's fill-session address, a token shown once - or that are typed inside a fresh VM shell where your variables are not set; replace them by hand when you meet them. The example column shows the values this guide uses in prose, tables and the architecture diagram; static addresses for the clones are the recommended case (chapter

7), and the VMIDs work as printed unless your host already uses them.

SET ONCE PER SHELL - the variables every pasted command relies on

```
# paste into your shell - or add to ~/.bashrc - on the WORKSTATION
# and on the PROXMOX HOST; edit the right-hand values to your LAN

PVE_HOST=192.168.1.5 # Proxmox VE host (web UI + root SSH)
GITLAB_HOST=gitlab.lan # hostname of your GitLab instance
GITLAB_IP=192.168.1.10 # address of the GitLab VM
GITLAB_URL=https://$GITLAB_HOST
REGISTRY=$GITLAB_HOST:5050 # container registry endpoint (Omnibus default)

VMID_TPL=9000 # VMID: cloud-init template (script 02)
VMID_RUNNER=201 # VMID: runner VM
RUNNER_IP=192.168.1.40 # address: runner VM
VMID_APP=202 # VMID: app VM
APP_VM_HOST=192.168.1.50 # address: app VM (same name as the CI variable)
GW=192.168.1.1 # LAN gateway, also handed to VMs as DNS
```

Placeholder	Example in this guide	What it is / where it appears
`$PVE_HOST`	192.168.1.5	Proxmox host: web UI, root SSH, scripts 02 and 04, scp targets `root@$PVE_HOST`
`$GITLAB_HOST`	gitlab.lan	Instance hostname: web UI, API, TLS certificate, git remotes
`$GITLAB_IP`	192.168.1.10	GitLab VM address: scp of the TLS helper (ch. 2), SAN entry (ch. 3)
`$GITLAB_URL`	https://gitlab.lan	Instance URL: runner registration in script 05 (chapter 8)
`$REGISTRY`	gitlab.lan:5050	Container registry endpoint: Docker trust (ch. 6), image paths (ch. 9, 12)
`$VMID_TPL` / `$VMID_RUNNER` / `$VMID_APP`	9000 / 201 / 202	VMIDs of the template and the two clones (ch. 6-7, snapshots, backups)
`$RUNNER_IP`	192.168.1.40	Runner VM address: SSH logins, cloning command (ch. 7)
`$APP_VM_HOST`	192.168.1.50	App VM address: deploys and checks; doubles as the CI variable name (ch. 10)
`$GW`	192.168.1.1	LAN gateway, passed to clones as gateway and DNS (script 04)
`<template-ip>`	assigned at boot	Fill-session address of the template VM - DHCP lease or `TPL_IP` (ch. 6)

Placeholder	Example in this guide	What it is / where it appears
`<group>/<project>`	homelab/gitlab-cicd-proxmox	Your GitLab project path: registry image path, git remote (ch. 9, 12)
`glrt-...`	shown once	Runner authentication token from the web UI (ch. 5), consumed in ch. 8

Table 4: Every varying value in one place - set the shell variables once per shell, replace angle placeholders by hand.

Finally, run three quick checks on the GitLab VM itself, because the rest of the guide assumes they hold. The instance is version 16 or newer, so the `glrt-` runner token flow works. The container registry is enabled - on an Omnibus installation verify with `grep ^registry_external_url /etc/gitlab/gitlab.rb` and expect an https entry on port 5050. And the instance URL resolves from the VMs, not only from your workstation. Then create an empty project (for example `homelab/gitlab-cicd-proxmox`), copy the kit contents into it, and you are ready for chapter 3.

One practical detail before the template build: the scripts have to run on the machines they configure, but right now they live only on your workstation. The transfer happens once and then never again: copy the whole scripts directory to the Proxmox host (scripts 02 and 04 run there), hand the GitLab VM the optional TLS helper of section 3.1, and stage a second copy inside the template VM. The fill step in section 6.2 installs that staged copy into `/opt/cicd-scripts`, the template bakes it in, and from then on EVERY clone ships the whole kit on disk - the runner and app VM chapters simply run the scripts from there. Proxmox permits root SSH logins out of the box, and the VMs accept your key from birth, because chapter 4 prepares the SSH keys and script 02 injects them into the template in the same breath. Cloning the GitLab project on the target machine works too once its TLS and credentials are set up; scp is just the path with the fewest moving parts.

ON YOUR WORKSTATION - moving the kit where it runs (numbers = run order)

```
# once, to the host - the host scripts (02, 04) plus the step-1a keys
scp -r scripts/ root@$PVE_HOST:/root/
scp ~/.ssh/id_ed25519.pub root@$PVE_HOST:/root/workstation.pub
scp ~/gitlab-ci-deploy.pub root@$PVE_HOST:/root/ci-deploy.pub

# optional, only for section 3.1 - the TLS helper runs on the GitLab VM
scp scripts/01-prepare-gitlab-tls.sh root@$GITLAB_IP:~/

# step 3 - the fill session stages the kit inside the template VM
scp -r scripts/ debian@<template-ip>:~/
```

Edit the right-hand sides of the variable block once and every command in this guide runs exactly as printed - `$PVE_HOST` is simply the address you already type into the Proxmox web UI. Everything that follows assumes the relevant script is already on the machine named in the label above each code block. The two `.pub` files are the public halves of the SSH key pairs chapter 4 creates on your workstation - the same scp round trip delivers them, and once script 02 has baked them into the template, no VM ever needs key

plumbing again and the clone script works without a key argument. Values that exist only later - the template's fill-session address - keep their `<angle>` placeholders and are replaced by hand when you get there.

3. Prepare the GitLab TLS Certificates

Everything in this setup talks to your GitLab instance over the network - the runner fetching jobs, Docker pulling and pushing images, CI checking out sources. Before any VM is built, decide how TLS to the instance is handled, because the answer is baked into the template once (chapter 6) instead of being repeated on every VM. Three situations cover every instance: it already uses a publicly trusted certificate (nothing to do here), it runs plain HTTP (also fine - see the callout below), or it uses - or should use - a certificate from a private CA, which is the usual self-hosted case and what this chapter prepares.

> **Plain HTTP instances skip this chapter.** If your GitLab URL starts with `http://`, no certificates are involved; Docker instead refuses to talk to the registry until your registry endpoint (`$REGISTRY`, example gitlab.lan:5050) is listed under `insecure-registries` - script 03 writes that into the template when you pass `INSECURE_REGISTRY=<gitlab-host>:5050` (section 6.2). HTTPS with a private CA is the tidier option - the rest of this chapter creates exactly that.

3.1 Optional: issue your own certificate on the GitLab VM

Use this section when your instance has no TLS at all - when nothing was ever configured and `/etc/gitlab/ssl/` is simply empty. Ten minutes as root on the GitLab VM give you a proper HTTPS setup backed by a small private CA: tidier than plain HTTP, no `insecure-registries` workarounds, and it produces the exact `/etc/gitlab/ssl/` layout that traditional Omnibus instructions assume. The commands create a 10-year CA and a server certificate whose name and SAN come from two variables at the top of the block - edit them to your instance before pasting. Note the `subjectAltName` line: without it, every client rejects the certificate no matter how it is installed. The kit ships this whole procedure as `scripts/01-prepare-gitlab-tls.sh`, which takes the same two values as environment variables, runs on the GitLab VM and prints the `gitlab.rb` lines when done; chapter 6 then bakes trust in the resulting CA into the template.

ON THE GITLAB VM - private CA plus server certificate (core of scripts/01-prepare-gitlab-tls.sh)

```
# shell variables for this block - adjust to your instance
GITLAB_HOST=gitlab.lan # <- your instance's hostname
GITLAB_IP=192.168.1.10 # <- its VM address (goes into the SAN)

mkdir -p /etc/gitlab/ssl && chmod 755 /etc/gitlab/ssl
cd /etc/gitlab/ssl

# a small private CA, valid 10 years
openssl genrsa -out gitlab-ca.key 4096
```

```
openssl req -x509 -new -nodes -key gitlab-ca.key -sha256 -days 3650 \
-subj '/CN=GitLab Homelab CA' -out gitlab-ca.crt

# the server certificate, signed by that CA
openssl genrsa -out "$GITLAB_HOST.key" 4096
openssl req -new -key "$GITLAB_HOST.key" -subj "/CN=$GITLAB_HOST" \
-out "$GITLAB_HOST.csr"
echo "subjectAltName=DNS:$GITLAB_HOST,DNS:localhost,IP:$GITLAB_IP" \
> "$GITLAB_HOST.ext"
echo 'extendedKeyUsage=serverAuth' >> "$GITLAB_HOST.ext"
openssl x509 -req -in "$GITLAB_HOST.csr" -CA gitlab-ca.crt -CAkey gitlab-ca.key \
-CAcreateserial -days 825 -sha256 -extfile "$GITLAB_HOST.ext" \
-out "$GITLAB_HOST.crt"
chmod 644 gitlab-ca.crt "$GITLAB_HOST.crt" && chmod 600 "$GITLAB_HOST.key"
openssl x509 -in "$GITLAB_HOST.crt" -noout -text \
| grep -A1 'Alternative\|Extended'
```

The final grep prints the two extensions the finished certificate must carry: the SAN listing `$GITLAB_HOST` and its IP, and the server-auth key usage. If either line is missing, fix the `subjectAltName` echo line before pointing Omnibus at the files - a certificate without the right SAN is rejected by every client no matter how it is installed. Then point Omnibus at the new files. The heredoc appends the HTTPS lines to `/etc/gitlab/gitlab.rb` in the same shell (skip any line you already have - duplicated keys only confuse reconfigure); if `registry_external_url` still says `http://`, point it at `https://` with port 5050 as well, and reconfigure - GitLab then terminates TLS itself, for the web UI and the container registry alike:

ON THE GITLAB VM - enable HTTPS for web UI and registry, then reconfigure

```
cat >> /etc/gitlab/gitlab.rb <<EOF
external_url "https://$GITLAB_HOST"
letsencrypt['enable'] = false
nginx['ssl_certificate'] = "/etc/gitlab/ssl/$GITLAB_HOST.crt"
nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/$GITLAB_HOST.key"
registry_nginx['ssl_certificate'] = "/etc/gitlab/ssl/$GITLAB_HOST.crt"
registry_nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/$GITLAB_HOST.key"
# registry_external_url "https://$GITLAB_HOST:5050" # uncomment if it says http
EOF

gitlab-ctl reconfigure # takes a few minutes
```

3.2 Keep the CA ready for the template

Whatever the TLS origin, one PEM file with the CA - or the chain the instance serves - ends up on your workstation, where the template build in chapter 6 picks it up as `GITLAB_CA_CERT`. If you issued the certificate in section 3.1, copy the CA off the GitLab VM and verify it:

ON YOUR WORKSTATION - fetch the new CA and verify it

```
# on your workstation
scp root@$GITLAB_IP:/etc/gitlab/ssl/gitlab-ca.crt ~/gitlab-ca.crt
```

```
curl --cacert ~/gitlab-ca.crt -sI https://$GITLAB_HOST | head -1
```

If the certificate existed before this guide - a private CA someone already runs - there is nothing to issue, and the file to keep is the chain your instance actually serves. Build it without any SSH access to the GitLab VM: the chain it hands out contains the anchor you need. For a self-signed setup that is the server certificate itself; with a private CA the CA cert is part of the chain. Saving the whole chain as one PEM file satisfies every consumer in chapter 6:

ON YOUR WORKSTATION - fetch the chain the instance serves

```
openssl s_client -connect $GITLAB_HOST:443 -servername $GITLAB_HOST -showcerts \
</dev/null 2>/dev/null \
| sed -n '/BEGIN CERTIFICATE/,/END CERTIFICATE/p' > gitlab-root.crt
openssl x509 -in gitlab-root.crt -noout -subject -issuer # sanity check
```

Prefer pulling the certificate off the GitLab VM? It only sits at a predictable place when TLS was configured manually: Omnibus looks in `/etc/gitlab/ssl/<hostname>.crt`, but only if someone actually put files there, and a custom location shows up in `grep ssl_certificate /etc/gitlab/gitlab.rb`. With Let's Encrypt enabled the files are managed automatically elsewhere (locate them with `find /var/opt/gitlab -name "*.crt"`). The `openssl s_client` method above works in every case and is the one to use.

The `curl` check should answer with an HTTP status line instead of a certificate error. Optionally, import the CA into the trust store of your workstation OS or browser so the GitLab web UI stops showing certificate warnings. With the certificate question settled and the PEM file in hand, you are ready for chapter 4, which settles SSH access the same one-time way - and for the template build in chapter 6, where every VM inherits both the certificate trust and the keys.

4. Prepare the SSH Keys

TLS is not the only trust decision to settle before any VM exists. SSH authenticates every interactive session in this setup - you logging into the VMs from your workstation - and one machine-to-machine relationship: the CI pipeline reaching the app VM to deploy. Cloud-init applies SSH keys only on the first boot of a VM, so - exactly like the certificates of chapter 3 - the keys must exist before the template is built and are then baked in once (chapter 6) instead of being sorted out per VM. Two key pairs cover everything: your workstation key for all admin logins, and a dedicated CI deploy pair named `gitlab-ci-deploy` whose private half later travels into a GitLab variable and whose public half ends up on the app VM. This chapter is step 1a of the big picture - it produces the files that script 02 bakes into the template.

ON YOUR WORKSTATION - step 1a: the two key pairs

```
# the admin key - skip this line if ~/.ssh/id_ed25519.pub already exists
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519 -N "" -C "workstation"

# the CI deploy pair - no passphrase, the pipeline cannot type one
ssh-keygen -t ed25519 -f ~/gitlab-ci-deploy -N "" -C "gitlab-ci-deploy"
```

The admin key is the one cloud images expect: from the first boot of every clone you log in as `debian` with it - no passwords anywhere. The CI pair is deliberately separate so it can be revoked without touching your personal keys. The private half `gitlab-ci-deploy` goes into a protected GitLab variable in chapter 10, and the public half is installed for the `deploy` user on the app VM (section 9.2). Because the template already carries that public half on disk, the installation step becomes a local file move instead of a paste - one more payoff of baking early. The empty passphrase on the CI key is intentional: the pipeline cannot answer a prompt, and the key is only as powerful as the GitLab variable store that guards its private half.

> **Generate keys on the machine that will use them.** The private admin half must live on your workstation; the private CI half ends up in the GitLab variable store. A key generated ON the Proxmox host would make the VMs trust only the host and lock your workstation out. If you already have a workstation key, reuse it - there is nothing to gain from a second one.

With both public halves in `~/.ssh/id_ed25519.pub` and `~/gitlab-ci-deploy.pub`, hand them to the Proxmox host - the same scp round that already moved the scripts there (chapter 2). Script 02 looks for the two files below, merges whichever exist into one cloud-init key list and injects it with `qm set --sshkeys` before the template VM boots its fill session. Override the paths with `ADMIN_PUBKEY=` and `CI_PUBKEY=` if you keep the files elsewhere; skip the CI key now and script 06 will simply ask for it later.

WORKSTATION + PROXMOX HOST - both public halves onto the host

```
# on your workstation - the host then holds both public halves
scp ~/.ssh/id_ed25519.pub root@$PVE_HOST:/root/workstation.pub
scp ~/gitlab-ci-deploy.pub root@$PVE_HOST:/root/ci-deploy.pub

# script 02 (next: chapter 6) then reports:
# ==> Injecting the SSH public keys (step 1a - guide ch. 4)...
# + /root/workstation.pub
# + /root/ci-deploy.pub
# Injected 2 key(s) for user 'debian'.
```

That is the whole step: two files on the host, a few seconds inside script 02, and every VM this kit ever clones is born trusting both keys. The next chapter spends its minutes in the GitLab web UI instead, and the template build in chapter 6 then bakes certificate trust, SSH keys and the kit scripts in one pass.

5. Create the Runner in GitLab

The runner is the worker that pulls jobs from GitLab and executes them. Since GitLab 16 the old registration-token flow is deprecated; the modern flow creates a runner in the web UI first and hands you a one-time runner authentication token starting with `glrt-` that the `gitlab-runner register` command consumes once, in chapter 8. Creating that entry now, before any VM exists, means the token is waiting when the runner VM is ready - it stays valid until a registration consumes it, no matter how long that takes, and no VM or certificate is involved in getting it.

- Open your project, then Settings - CI/CD - Runners and click `New project runner`.
- Enter a description (e.g. proxmox-runner-01). Tags are optional - leave them empty and enable `Run untagged jobs` so the sample pipeline works without tag plumbing.
- Click `Create runner` and copy the `glrt-...` token from the step 2 instruction box. GitLab shows it only once.

GitLab displays the token only once and cannot show it again - store it somewhere safe immediately. If it is ever lost, or another registration consumes it, simply create a new runner: nothing else is wasted. The same dialog exists instance-wide under `Admin Area > CI/CD > Runners` if you prefer a shared runner over a project runner.

6. Build the All-in-One Template

The whole setup rests on one reusable template: a Debian 12 VM with cloud-init enabled that already carries the QEMU guest agent, Docker Engine, the kit's own scripts, your SSH keys from chapter 4 and - when chapter 3 produced a private CA - trust in your GitLab certificate. Built once, it guarantees that every VM you clone later has the same disk layout, the same first-boot behavior and a working toolchain on disk: clones boot ready to work, with no per-VM package installs, no per-VM certificate plumbing and no key plumbing. Debian publishes official cloud images that contain cloud-init and a default `debian` user with passwordless sudo; use the `generic` variant (not `genericcloud`) because it ships the full driver set and works on any hypervisor.

6.1 Creating the template VM on the host

ON THE PROXMOX HOST - core of scripts/02-create-debian-template.sh (script 02 holds these values in its config block)

```
cd /var/tmp
BASE=https://cloud.debian.org/images/cloud/bookworm/latest
wget $BASE/debian-12-generic-amd64.qcow2

qm create $VMID_TPL --name debian-12-cloudinit \
--memory 2048 --cores 2 \
--net0 virtio,bridge=vmbr0 \
--agent enabled=1 --ostype l26
```

```
qm importdisk $VMID_TPL debian-12-generic-amd64.qcow2 local-lvm

qm set $VMID_TPL --scsihw virtio-scsi-pci \
--scsi0 local-lvm:vm-$VMID_TPL-disk-0,discard=on,ssd=1
qm set $VMID_TPL --ide2 local-lvm:cloudinit
qm set $VMID_TPL --boot order=scsi0
qm set $VMID_TPL --serial0 socket --vga serial0

qm disk resize $VMID_TPL scsi0 20G

# script 02 then boots the VM once for the fill-in (section 6.2);
# conversion into a template happens after that, in section 6.3
```

Each line has a purpose worth understanding, because you will maintain this template for a long time:

Command / flag	What it does
qm importdisk	Imports the downloaded qcow2 as an unused disk attached to the VM, ready to be wired up
--scsihw virtio-scsi-pci + discard=on	Modern SCSI controller with TRIM support, so deleted blocks return to the storage pool
--ide2 ...:cloudinit	Attaches the special cloud-init drive that Proxmox populates with user, network and SSH-key data
--boot order=scsi0	Boots from the imported disk (cloud images have no BIOS bootloader on other devices)
--serial0 socket --vga serial0	Routes the console to a serial device - how cloud images expect to talk to the Proxmox console
qm disk resize	Grows the root filesystem; 20 GiB is comfortable for Docker images and build caches
qm template (section 6.3)	Converts the finished VM into an immutable template that can be cloned but never started

Table 5: What each template-building step achieves.

6.2 Filling it once: agent, Docker, TLS trust and the kit scripts

A template that only boots would still leave every clone one package-install session away from usefulness, so the build fills it once while it is still a normal VM: script 02 starts the new VM (cloud-init gives the `debian` user the keys from chapter 4 and a DHCP or `TPL_IP` address), and inside it you run `scripts/03-prepare-template.sh`. Everything the script puts on disk is inherited by every future clone: the `qemu-guest-agent` that the stock image lacks (so Proxmox always shows the IP), Docker Engine with the compose plugin from the official Docker repository, and - the payoff of chapter 3 - trust in your GitLab certificate. Pass the CA file from section 3.2 and the script installs it system-wide and into the Docker

registry trust directory in one go; for plain-HTTP instances pass `INSECURE_REGISTRY=<gitlab-host>:5050` instead and it writes the `daemon.json` entry. As a final touch it pre-pulls `docker:27`, the default job image, so the first pipeline does not wait on Docker Hub.

The staged scripts directory makes this fill session the last logistics chore of the whole setup: script 03 installs whatever you scp'd into `~/scripts` to `/opt/cicd-scripts` and the template bakes it in. Every clone therefore ships its own finishing scripts - section 8.1 runs the runner installer from that directory and section 9.2 the app VM preparation - and no script is ever scp'd to an individual VM again.

FILL THE TEMPLATE VM ONCE - scripts/03-prepare-template.sh (CA from ch. 3; the staged scripts/ dir gets baked in)

```
# on the Proxmox host - script 02 leaves the VM running
scp -r scripts/ debian@<template-ip>:~/
scp ~/gitlab-ca.crt debian@<template-ip>:~/ # if ch. 3 produced a CA

# inside the template VM
GITLAB_CA_CERT=~/gitlab-ca.crt ./scripts/03-prepare-template.sh

# verification, after re-login
docker --version && docker compose version
ls /opt/cicd-scripts
```

> **Group membership needs a fresh login.** The script adds the `debian` user to the `docker` group so you can run Docker without sudo, but group changes only apply to new login sessions. Log out and back in (or reconnect SSH) before the verification commands - clones inherit the group automatically.

6.3 Converting to a template and verifying

With the software on disk, shut the VM down and flip it into an immutable template:

ON THE PROXMOX HOST - the last step of the template build

```
qm shutdown $VMID_TPL && qm template $VMID_TPL
```

The template appears under your `$VMID_TPL` (example 9000, name `debian-12-cloudinit`) in the Proxmox web UI. A quick sanity check from the host shell: `qm config $VMID_TPL` should list a `scsi0` disk, an `ide2` cloudinit drive and `template: 1`. The downloaded image stays cached in `/var/tmp`, so rebuilding the template later - for example when a new Debian point release ships, or to pick up a renewed CA - is a rerun of script 02 plus the fill and convert steps above, a ten-minute chore at most.

7. Cloning the Virtual Machines

With the template in place, both VMs are one command each - and each clone already carries the guest agent, Docker, the GitLab trust, your SSH keys and the kit scripts from the template. The clone script injects a cloud-init user, optional extra SSH keys and the network settings, then starts the VM. Passing a static IP and gateway is recommended: both VMs then boot at addresses you chose, which the CI variables and your DNS entries will expect anyway. Full clones copy all data, so they are independent of the template afterwards; on a single-node homelab that isolation is worth the extra few gigabytes.

The SSH question has already been answered: the keys chapter 4 prepared are part of the template, so a clone is reachable without any key argument at all. The optional third argument exists for exceptions - a second admin, a recovery key - and adds public keys on top of the inherited ones for this one VM. Whatever you pass has to be readable on the Proxmox host, and it must not be a key generated on the host itself (such as `/root/.ssh/id_ed25519.pub`): the VMs would then trust only the host, and your workstation would be locked out. Script 04 checks the file before cloning, refuses anything that is not a public key, and asks for explicit confirmation when the path looks like a host key.

ON THE PROXMOX HOST - scripts/04-clone-vm.sh (static IP + gateway; keys come from the template)

```
./scripts/04-clone-vm.sh $VMID_RUNNER gitlab-runner-vm $RUNNER_IP/24 $GW
./scripts/04-clone-vm.sh $VMID_APP app-vm $APP_VM_HOST/24 $GW
```

The actions behind the scenes: `qm clone ... --full 1` copies the template, `--ciuser debian` selects the default cloud user, and the template's authorized keys apply unless you pass an extra key file, in which case `--sshkeys` injects one public key per line from it - so logins work from your workstation without further setup. `--ipconfig0` writes the network config - static when you passed an address and gateway, DHCP otherwise. Note that cloud-init applies keys and passwords only on the first boot: adding a key to an already-booted VM with `qm set --sshkeys` has no effect. The guest agent is already on disk (section 6.2 filled the template), so the Proxmox Summary tab and `qm guest cmd` report each VM as soon as it has booted - even in DHCP mode, where you can alternatively read the lease from your router or scan the subnet with `nmap -sn`, matching against the MAC address from `qm config <vmid>`.

Log in to confirm cloud-init did its job - and that the template content came along:

ON YOUR WORKSTATION

```
ssh debian@$RUNNER_IP # runner VM
ssh debian@$APP_VM_HOST # app VM
hostnamectl # should read Debian GNU/Linux 12
docker --version # already on disk from the template
ls /opt/cicd-scripts # the kit's scripts, baked in (section 6.2)
```

Optional but recommended before the finishing chapters: take a clean snapshot of each VM (`qm snapshot $VMID_RUNNER pristine` and likewise for `$VMID_APP`). If a later step goes wrong you can roll back in seconds without redoing the clone. Note that snapshots are not backups - chapter 13 covers proper `vzdump`

backups for long-term safety.

8. Finish the Runner VM: Install and Register

The runner VM left the cloning chapter with Docker, the guest agent and the GitLab TLS trust already on disk. Two steps remain, and script 05 does both: install the `gitlab-runner` package from its official repository, then register the VM against your instance using the token from chapter 5 - from that moment the runner appears online in GitLab and starts pulling jobs.

8.1 Installing the runner package

No fetch needed this time: the fill step in section 6.2 baked the kit scripts into the template, so the runner VM already has this one on disk in `/opt/cicd-scripts` - only if you built the template without that step, scp it over from your workstation as in chapter 2. Run the script inside the VM; the two code blocks below show what it does.

INSIDE THE RUNNER VM - the baked-in scripts/05-install-gitlab-runner.sh

```
bash /opt/cicd-scripts/05-install-gitlab-runner.sh
```

8.2 Registering the runner against your instance

The register command below takes one credential: the runner authentication token from chapter 5. It comes only from the GitLab web UI - it is never generated on any of your VMs and has nothing to do with the certificates of chapter 3. If you still have the `glrt-...` string you copied there, use it. If you skipped chapter 5, or the token is gone (GitLab displays it only once and cannot show it again, and another registration consumes it), create the runner now: open your project, go to `Settings > CI/CD > Runners`, click `New project runner`, leave tags empty with `Run untagged jobs` enabled, click `Create runner`, and copy the `glrt-...` token from the step 2 instruction box. Paste exactly that string into the `--token` line below. The same dialog exists instance-wide under `Admin Area > CI/CD > Runners` if you want a shared runner instead of a project runner.

WHAT SCRIPT 05 DOES - the register step it performs for you

```
sudo gitlab-runner register \
--non-interactive \
--url https://<gitlab-host> \
--token glrt-xxxxxxxxxxxxxxxxxxx \
--executor docker \
--description proxmox-runner-01 \
--docker-image docker:27 \
--docker-volumes /var/run/docker.sock:/var/run/docker.sock \
--docker-volumes /cache

sudo systemctl enable --now gitlab-runner
```

```
sudo gitlab-runner verify
```

There is no certificate argument to worry about: the template already trusts your GitLab CA system-wide (section 6.2), and the runner uses that trust for its API connection. Only if you built the template without the CA - or the instance changed its certificate since - pass `GITLAB_CA_CERT=/path/to/ca.pem` to script 05: it installs the trust and adds `--tls-ca-file` to the registration by itself.

Two details in that registration matter for everything that follows. First, the **docker executor** means each CI job runs in its own container (the `docker:27` image is only the default, jobs override it). Second, the host Docker socket is mounted into job containers, so release jobs can talk to the VM Docker daemon directly to build and push images - without a privileged Docker-in-Docker service. Alternatives exist: privileged DinD isolates builds better, and kaniko builds images rootless, but for a single-tenant homelab runner the socket mount is the pragmatic standard.

> **Security note.** Mounting `/var/run/docker.sock` gives jobs root-equivalent access to the runner VM Docker daemon. Keep this VM dedicated to CI and only build code you trust. If you later share the runner across projects, switch to DinD or kaniko instead.

Verify from the VM with `sudo gitlab-runner list` and from your GitLab web UI: the runner entry under Settings - CI/CD - Runners should show a green dot within seconds. The resulting configuration lives in `/etc/gitlab-runner/config.toml`; the file `runner/config.toml.example` in the kit documents the expected shape - the `tls-ca-file` entry appears only when script 05 was given `GITLAB_CA_CERT`. If you ever edit it by hand, reload with `sudo systemctl restart gitlab-runner`.

9. Preparing the Deployment VM

The app VM is deliberately boring: it never builds anything, it only runs what the pipeline ships. Everything on it is owned by a dedicated `deploy` user - not your personal account - and the CI pipeline authenticates as that user with the key pair created in chapter 4. Preparing it is a matter of running one script and placing one file.

9.1 The CI deploy key from chapter 4

The key pair was created back in step 1a: `gitlab-ci-deploy` (private) sits on your workstation and goes into a GitLab variable in chapter 10, while `gitlab-ci-deploy.pub` was handed to script 02 and baked into the template. The app VM therefore already carries the public half among the `debian` user's authorized keys; the preparation script below moves it into the `deploy` user's own `authorized_keys` - a local file move, no pasting. If you skipped that step, the script falls back to asking for the public key, and you can scp the `.pub` file over the same way as any other. A dedicated pair matters: it can be revoked independently of your personal keys, and its private half never needs to sit in your SSH agent.

9.2 Run the preparation script

The script is already on the app VM: the template ships the kit in `/opt/cicd-scripts` (section 6.2). Run it from there; everything it needs is prompted for or picked up from the template, so nothing else has to be transferred in advance.

INSIDE THE APP VM - the baked-in scripts/06-prepare-app-vm.sh

```
bash /opt/cicd-scripts/06-prepare-app-vm.sh
```

The script walks through five prompts and actions. It creates the `deploy` user and adds it to the `docker` group; installs your CI public key into `/home/deploy/.ssh/authorized_keys`; creates `/opt/docker-app` for the compose project and `/var/www/static-app` as the nginx web root; installs nginx with a vhost that serves the static site with SPA-style fallback routing; and finally performs `docker login` for the deploy user against the container registry that your GitLab VM serves - the `$REGISTRY` endpoint from the variable block, typed at the prompt.

NGINX VHOST - installed as /etc/nginx/sites-available/static-app

```
server {
listen 80 default_server;
server_name _;
root /var/www/static-app;
index index.html;
location / { try_files $uri $uri/ /index.html; }
}
```

For the registry login you need a **deploy token**: in your GitLab open Settings - Repository - Deploy tokens, create one with only the `read_registry` scope, and give its username and secret when the script asks. The registry certificate is nothing to arrange either: the template already trusts your GitLab CA (section 6.2). Only if you skipped that, scp the certificate file to the app VM and pass its path there with `GITLAB_CA_CERT=/path/to/ca.crt` when invoking the script, so it lands in `/etc/docker/certs.d/` before the login. With a private registry this token is what allows `docker compose pull` on the app VM to succeed without embedding credentials anywhere in the repository.

9.3 Place the compose project

ON YOUR WORKSTATION

```
scp deploy/docker-compose.prod.yml \
debian@$APP_VM_HOST:/tmp/compose.yml
ssh debian@$APP_VM_HOST \
"sudo mv /tmp/compose.yml /opt/docker-app/docker-compose.yml \
&& sudo chown deploy:deploy /opt/docker-app/docker-compose.yml"
```

Then edit one line on the app VM: the `image:` path must point at your real registry location - with the registry on your own instance that is `$REGISTRY/<group>/<project>/docker-app`. You do not have to guess: the release job prints the exact value of `$CI_REGISTRY_IMAGE` in its log, and the compose file must

match it. The first deploy will populate the running container; until then nginx alone answers on port 80.

Final check before wiring up CI - deploy access must work from anywhere the runner could be:

ON YOUR WORKSTATION

```
ssh -i gitlab-ci-deploy deploy@$APP_VM_HOST "docker ps"
```

10. CI/CD Variables and Secrets

The pipeline reads every environment-specific value from CI/CD variables, so the repository itself stays free of hostnames, usernames and keys. Set them under Project - Settings - CI/CD - Variables. Four variables are required by the kit as shipped.

Variable	Type	Flags	Purpose
SSH_PRIVATE_KEY	File	Protect	Private half of the CI deploy key created in chapter 4; File type avoids multi-line escaping issues
SSH_KNOWN_HOSTS	Variable	Protect	Pinned host keys of the app VM, produced by ssh-keyscan
DEPLOY_USER	Variable	Protect	Always deploy in this kit
APP_VM_HOST	Variable	Protect	App VM address - the `$APP_VM_HOST` value from the variable block

Table 6: Required CI/CD variables.

RUN ON YOUR WORKSTATION - paste the output into SSH_KNOWN_HOSTS

```
ssh-keyscan -H $APP_VM_HOST
```

The `SSH_KNOWN_HOSTS` entry is what makes deploys immune to man-in-the-middle attacks on your LAN: the deploy job writes it to `~/.ssh/known_hosts` and OpenSSH then refuses to connect if the app VM ever presents a different key. If you rebuild the app VM from scratch, re-run `ssh-keyscan` and update the variable.

> **Protected variables only reach protected refs.** With the Protect flag set, variables are available on the default branch (where release and deploy jobs run) but not on feature branches - a good default. The multi-line private key must use the **File** type; masking does not work for multi-line values.

One more credential lives outside the variables: the registry deploy token from chapter 9.3, already stored on the app VM by `docker login`. Keep its scope at `read_registry` only - the app VM should never be able to push images. CI itself needs no extra registry credential because your GitLab injects `$CI_REGISTRY_USER` and `$CI_REGISTRY_PASSWORD` (a per-job token) into every job automatically. The

built-in registry variables are instance-aware: `$CI_REGISTRY` resolves to `$REGISTRY` and `$CI_REGISTRY_IMAGE` to `$REGISTRY/<group>/<project>` without any configuration.

11. The Pipeline Explained

The complete pipeline ships as `.gitlab-ci.yml` in the kit root. It declares four stages - test, build, release, deploy - and a `workflow` rule that runs pipelines on pushes and merge requests. Rather than walking through all 100 lines, this chapter explains the four mechanisms you will actually maintain.

11.1 Path-based triggering

Every job carries a `rules:changes` clause naming its application folder. A commit that only touches `apps/static-app/**` triggers the static track and skips the container track entirely, and vice versa. Release and deploy jobs additionally require `$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH`, so deploys happen only from the default branch while tests still guard every merge request.

11.2 The shared SSH bootstrap

EXCERPT FROM .gitlab-ci.yml - deploy-job preamble as a YAML anchor

```
.ssh-setup: &ssh-setup
- apk add --no-cache openssh-client rsync
- eval $(ssh-agent -s)
- echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
- mkdir -p ~/.ssh && chmod 700 ~/.ssh
- echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts
- chmod 644 ~/.ssh/known_hosts
```

Both deploy jobs reuse this preamble through the `&ssh-setup` YAML anchor. It loads the private key into a per-job agent (File-type variables arrive with a trailing newline that `tr -d` strips), pins the app VM host key, and installs rsync - the only package besides the SSH client that the deploy jobs need.

11.3 The container track

EXCERPT FROM .gitlab-ci.yml - release job

```
docker-app:release:
stage: release
image: docker:27
rules:
- if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
changes:
- apps/docker-app/**/*
script:
- docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
- docker build
-t "$CI_REGISTRY_IMAGE/docker-app:$CI_COMMIT_SHORT_SHA"
```

```
-t "$CI_REGISTRY_IMAGE/docker-app:latest"
apps/docker-app
- docker push "$CI_REGISTRY_IMAGE/docker-app:$CI_COMMIT_SHORT_SHA"
- docker push "$CI_REGISTRY_IMAGE/docker-app:latest"
```

The job image `docker:27` provides the Docker CLI; thanks to the socket mount it talks to the runner VM daemon. Every image is pushed twice: once tagged with the short commit SHA (immutable, lets you roll back by retagging) and once as `latest` (what the app VM pulls). Deployment then needs no image knowledge at all:

EXCERPT FROM .gitlab-ci.yml - deploy job

```
docker-app:deploy:
script:
- ssh "$DEPLOY_USER@$APP_VM_HOST" \
"cd /opt/docker-app && docker compose pull && docker compose up -d"
```

11.4 The static track

The static build runs in `node:22-alpine` with `npm ci` (lockfile-exact, reproducible) and publishes `apps/static-app/dist/` as a pipeline artifact with a one-week expiry. The deploy job declares `needs: [static-app:build]` so it downloads that artifact explicitly and pushes it to the web root with `rsync -rc --delete` - checksum mode avoids rewrites of unchanged files, and `--delete` prunes assets from older builds so the site never accumulates stale hashed bundles.

12. First Run and Verification

Everything is wired. Commit the kit to your GitLab project and push - the first pipeline starts within seconds. Expect the container track to take 2-4 minutes on the first run (image pulls dominate) and the static track well under a minute. The pipeline overview shows each job moving from pending through running to green.

ON YOUR WORKSTATION - first push (SSH remote on your instance)

```
git init && git add -A && git commit -m "homelab CI/CD kit"
git remote add origin git@$GITLAB_HOST:<group>/<project>.git
git push -u origin main
```

When the pipeline is green, verify each deliverable at the destination instead of trusting the logs alone:

Check	Command	Expected result
Runner online	Settings - CI/CD - Runners	Green dot next to proxmox-runner-01
Image in registry	Left menu - Deploy - Container Registry	docker-app with latest + SHA tags

Check	Command	Expected result
Container app	`curl http://$APP_VM_HOST:8000/health`	{"status":"ok"}
Static site	`curl -I http://$APP_VM_HOST/`	HTTP/1.1 200 OK
Running container	ssh deploy@$APP_VM_HOST, then `docker ps`	docker-app Up, port 8000 published
Web root fresh	ssh deploy@$APP_VM_HOST, then `ls /var/www/static-app`	index.html + assets/ from dist/

Table 7: Post-deploy verification checklist.

> **Pipeline green but app not updated?** Check the compose file first: its `image:` path must match `$CI_REGISTRY_IMAGE` exactly (same registry host, same group/project path). A mismatch means compose pulls a stale image that still exists elsewhere.

From now on the workflow is deliberately dull: change code, push, watch the relevant track rebuild and redeploy. If you break the static site, only the static track runs; if a pytest case fails, the pipeline stops before any image is built or any server is touched.

13. Maintenance and Troubleshooting

A homelab CI platform earns its keep only if it is boring to operate. The routine is small: keep Debian and the runner updated, back up both VMs from Proxmox, and glance at disk usage now and then - Docker images and build caches are the only things that grow without limit.

13.1 Troubleshooting reference

Symptom	Likely cause	Fix
Runner shows offline in GitLab	gitlab-runner service stopped or token invalid	`systemctl status gitlab-runner`, then `sudo gitlab-runner verify`
Jobs stuck in pending	Run untagged jobs not enabled, or concurrent limit reached	Enable untagged on the runner or raise `concurrent` in config.toml
`permission denied ... docker.sock`	User not in docker group	Re-login so the docker group applies (the template set it); check `groups`
`guest agent is not running` / no IP in the Summary tab	The VM was cloned from a template that script 03 never filled	Re-clone from a filled template, or run script 03 inside the VM once; with a static IP the agent is never needed to find the address

Symptom	Likely cause	Fix
`x509: certificate signed by unknown authority` on register/verify	The template was built without the GitLab CA	Bake the CA into the template (`GITLAB_CA_CERT`, chapter 6) and clone again, or pass `GITLAB_CA_CERT` to script 05, which installs trust and `--tls-ca-file` itself
`x509 ...` or `http: server gave HTTP response to HTTPS client` on push/pull	Docker daemon does not trust the registry certificate (or registry is plain HTTP)	Install the CA under `/etc/docker/certs.d/<registry-host>/` or add `insecure-registries`
Registration fails with 401 / unauthorized	Token from a different instance, or URL mistyped	Register against the exact instance URL that shows the token - correct scheme, hostname and port
`Host key verification failed` in deploy job	SSH_KNOWN_HOSTS missing or stale	Re-run `ssh-keyscan -H` and update the variable
`denied: access forbidden` on compose pull	Deploy token expired or wrong scope	Create a fresh token with read_registry and `docker login` as deploy
Pipeline does not trigger	Changed files outside both app folders	Path rules are working as designed; touch the app folder or run manually via Run pipeline
`rsync: command not found` in deploy job	ssh-setup preamble not applied	Ensure `before_script: *ssh-setup` on the job
SSH into the VMs works only from the Proxmox host	The clone injected the public key of the HOST, not of the workstation	Copy the workstation key to the host (`scp ~/.ssh/id_ed25519.pub root@$PVE_HOST:/root/workstation.pub`), then from the host append it per VM: `ssh debian@<vm-ip> "cat >> ~/.ssh/authorized_keys" < /root/workstation.pub`
Runner VM disk filling up	Dangling images and build cache	`docker image prune -f` and `docker builder prune -f`

Table 8: The failures you are most likely to meet - the first three are self-hosting classics.

The host-key lockout is worth spelling out, because it is the one mistake cloud-init will not let you undo from the workstation: keys and passwords are applied only on the first boot, so `qm set --sshkeys` on an already-booted VM changes nothing. The way back in runs through the key that does work. On your workstation, copy the key to the host with `scp ~/.ssh/id_ed25519.pub root@$PVE_HOST:/root/workstation.pub`; then, on the host, append it to each VM with `ssh debian@$RUNNER_IP "cat >> ~/.ssh/authorized_keys" < /root/workstation.pub` and confirm from the workstation that `ssh debian@$RUNNER_IP` now logs in. Leaving the host key in place is harmless - it is your own hypervisor. The clean-slate alternative is destroying the VM and cloning it again with the correct key file.

13.2 Keeping the platform healthy

- **Updates.** Monthly `sudo apt-get update && sudo apt-get upgrade` on both VMs; bump GitLab Runner from its official repository with the same command. The runner tolerates a quick restart.
- **Backups.** Add both VMs to a Proxmox Storage - Backup job (`vzdump` under the hood) with a modest retention, e.g. 3 weekly copies. The VMs are stateless-ish; the app VM only holds the registry credential and nginx config.
- **Registry hygiene.** Set a cleanup policy (Settings - Packages and registries) to prune old SHA-tagged images and keep `latest`.
- **Scaling out.** Need more parallel jobs? Clone the template again into VM 203 and register a second runner - nothing else changes. For dynamic per-job VMs, the Proxmox autoscaler for GitLab Runner is a natural next step beyond this guide.
- **Hardening.** Consider an isolated bridge or VLAN for the VMs, fail2ban on the app VM, and keeping the Protect flag on all four CI variables.

With those habits in place the platform fades into the background, which is the point: the runner VM churns through pipelines, the app VM always reflects the default branch, and Proxmox quietly provides snapshots, backups and a template for whatever you decide to clone next.