AI-04702-GCP to On-Premise Cloudflare Setup

I have a working DNS gravelcloud.com from Cloudflare which is pointed to my GCP account to expose my Dockerlab projects to the internet. In my condo (PLDT CGNAT), have 2 Latitude laptops running Ubuntu servers running a few dockerlabs as well and I have a Tailscale on it. I want to learn how to backup/restore from GCP to On-Prem or On-Prem to GCP. Will I incur additional expenses with the setup? Create a step by step instructions on how to add my on-prem dockerlab to Cloudflare.

Yes. Your existing setup is a very good foundation for building a **hybrid DockerLab**:

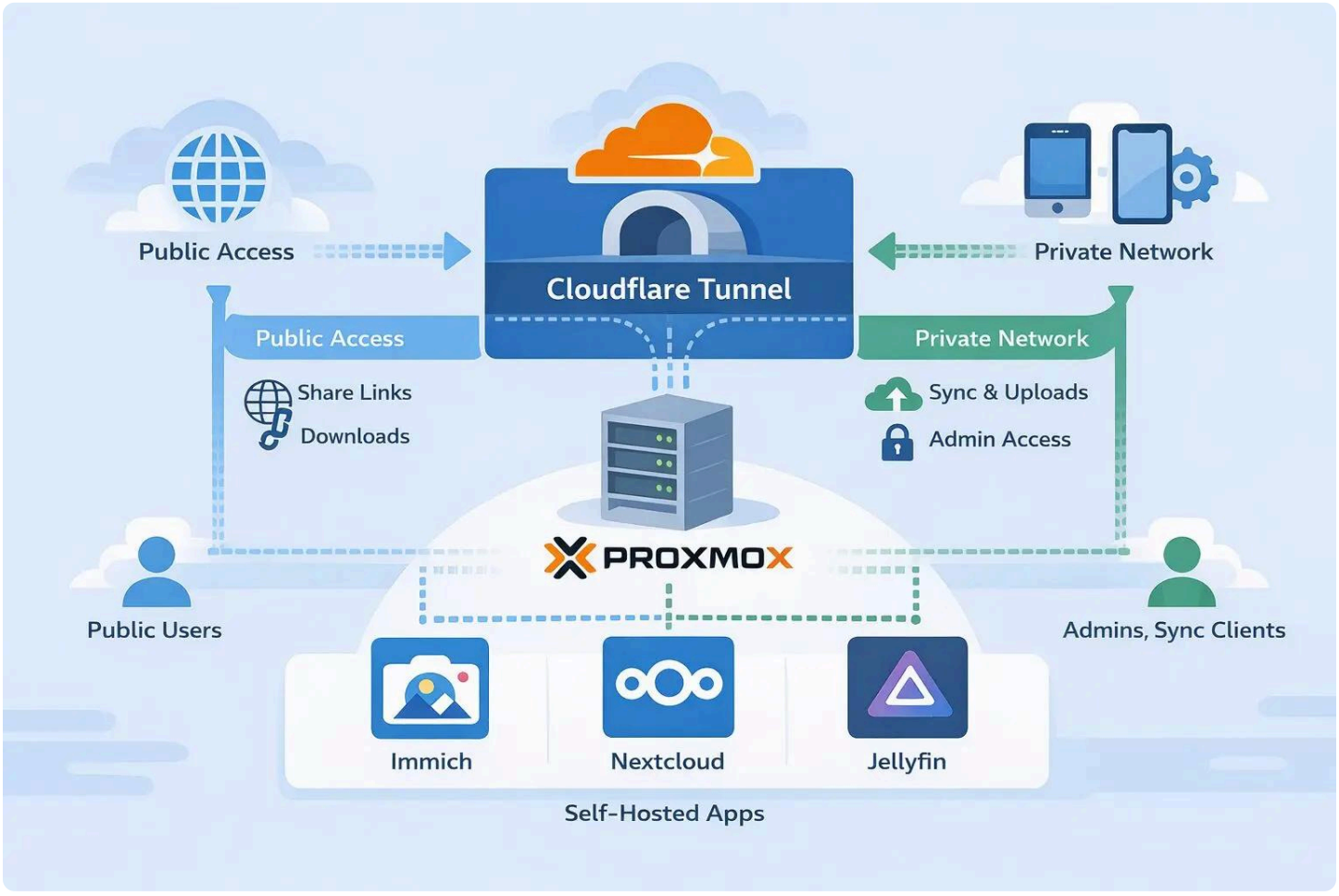
- **GCP:** Your current internet-facing DockerLab and public services.
- **Condo / On-Prem:** Two Ubuntu Latitude servers running additional DockerLab projects.
- **Cloudflare:** Your domain, DNS, and public ingress.
- **Tailscale:** Secure connectivity between your GCP and condo environments, even behind PLDT CGNAT.
- **Backup storage:** A separate location for recovering your DockerLab if one environment fails.

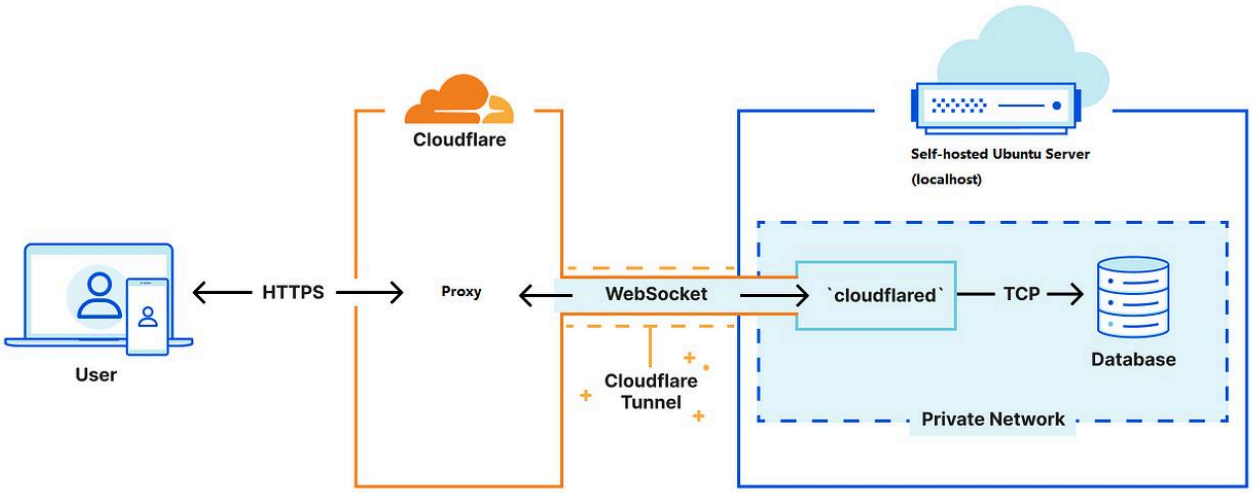
The key is to distinguish **publishing an application** from **connecting two private networks**. They are related, but they are not the same thing.

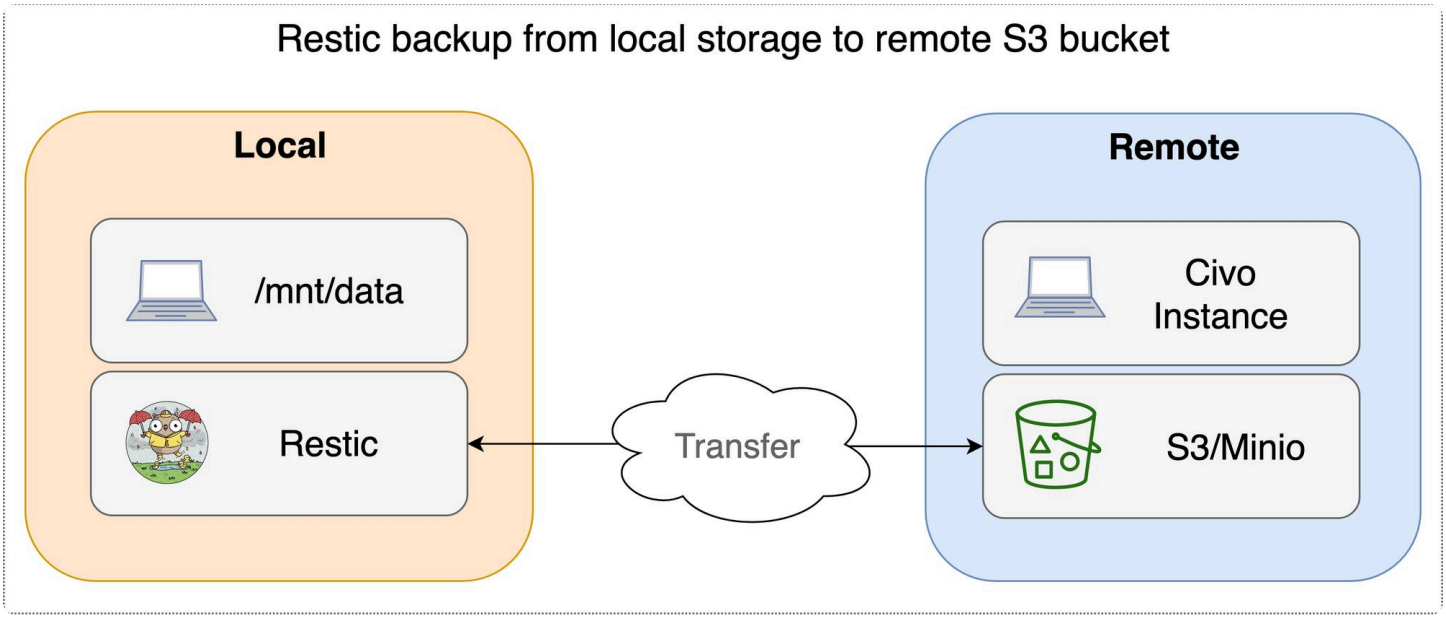
---

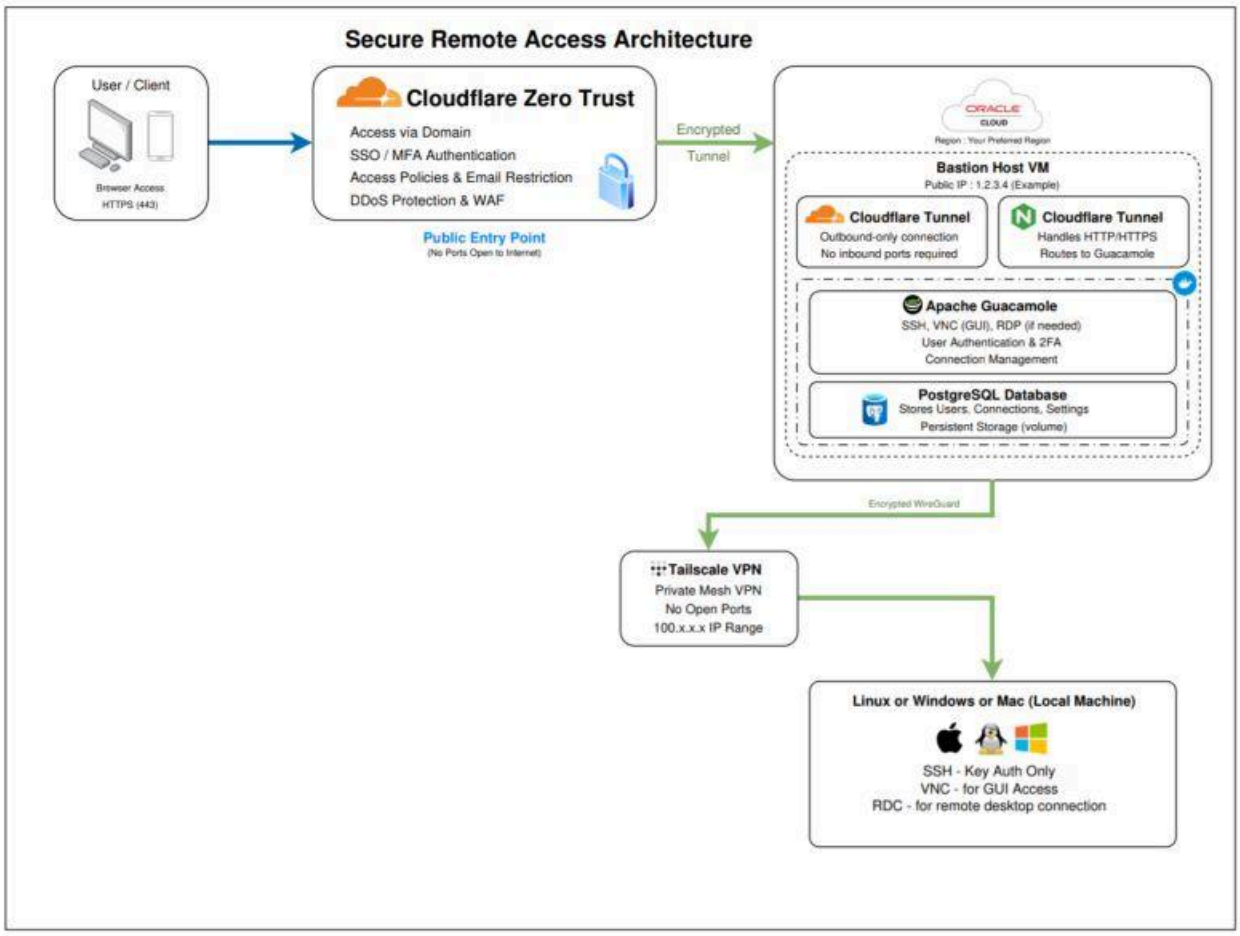
1. Recommended Architecture

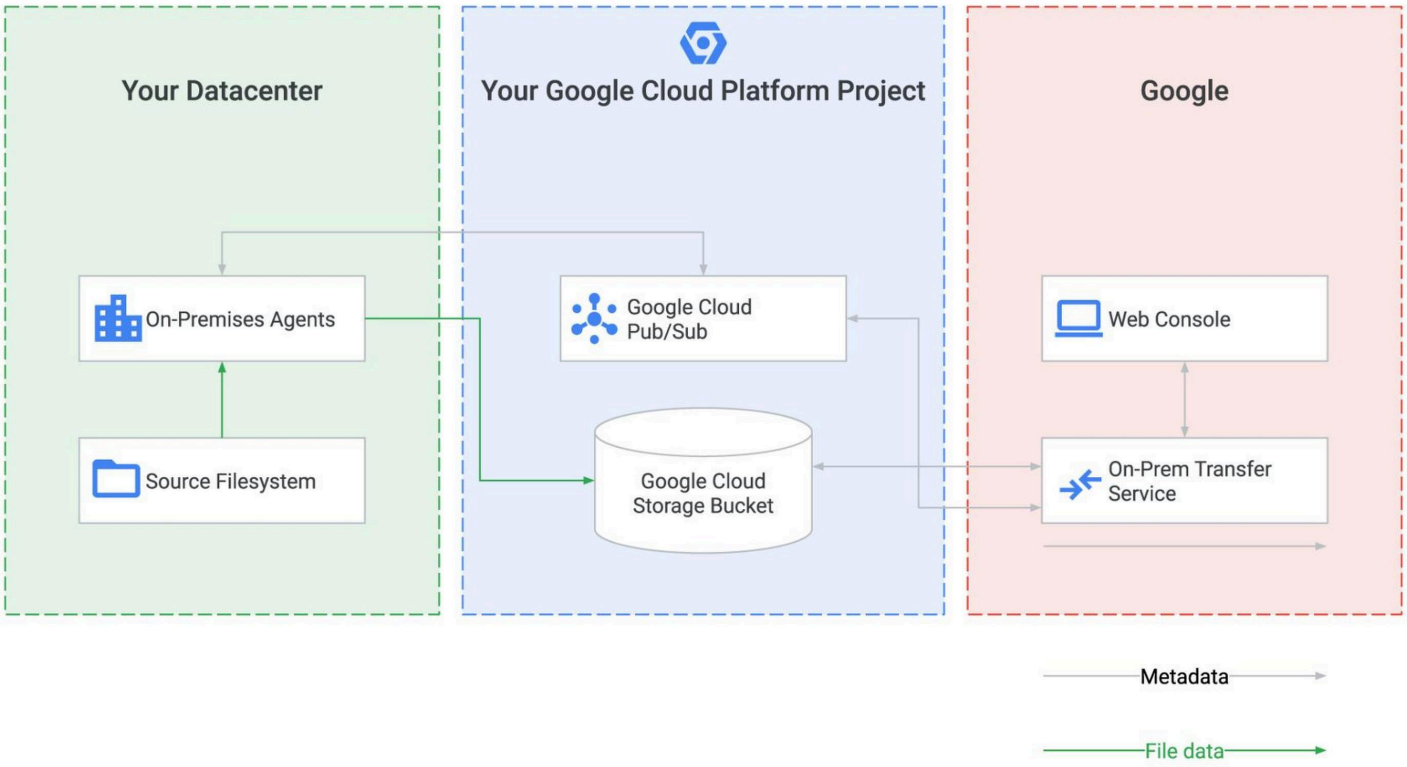
I would build your environment like this:



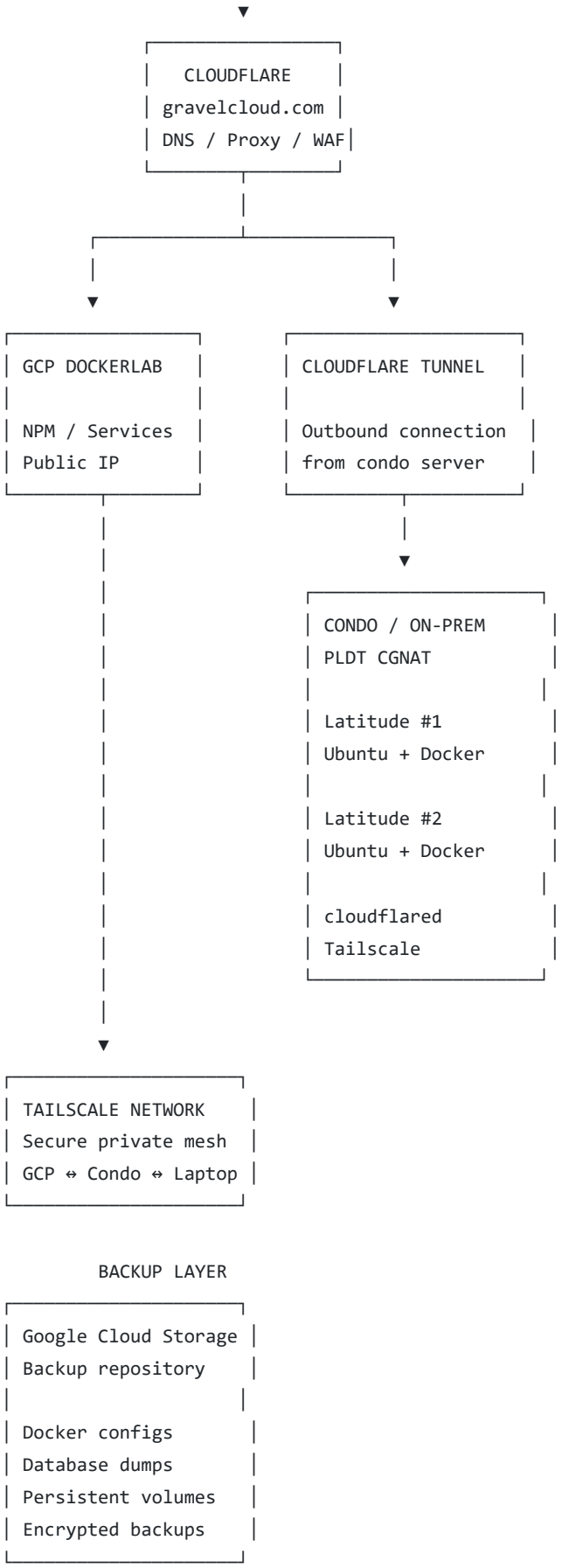

Cloudflare
Self-hosted Ubuntu Server
(localhost)
User
HTTPS
Proxy
WebSocket
`cloudflared`
TCP
Database
Cloudflare
Tunnel
Private Network


Restic backup from local storage to remote S3 bucket
Local
/mnt/data
Restic
Transfer
Remote
Civo
Instance
S3/Minio


Secure Remote Access Architecture
User / Client
Browser Access
HTTPS (443)
Cloudflare Zero Trust
Access via Domain
SSO / MFA Authentication
Access Policies & Email Restriction
DDoS Protection & WAF
Public Entry Point
(No Ports Open to Internet)
Encrypted
Tunnel
ORACLE CLOUD
Region : Your Preferred Region
Bastion Host VM
Public IP : 1.2.3.4 (Example)
Cloudflare Tunnel
Outbound-only connection
No inbound ports required
Cloudflare Tunnel
Handles HTTP/HTTPS
Routes to Guacamole
Apache Guacamole
SSH, VNC (GUI), RDP (if needed)
User Authentication & 2FA
Connection Management
PostgreSQL Database
Stores Users, Connections, Settings
Persistent Storage (volume)
Encrypted WireGuard
Tailscale VPN
Private Mesh VPN
No Open Ports
100.x.x.x IP Range
Linux or Windows or Mac (Local Machine)
SSH - Key Auth Only
VNC - for GUI Access
RDC - for remote desktop connection


Your Datacenter
Your Google Cloud Platform Project
Google
On-Premises Agents
Source Filesystem
Google Cloud Pub/Sub
Google Cloud Storage Bucket
Web Console
On-Prem Transfer Service
Metadata
File data

```
INTERNET
    |
```



My recommendation: Keep your existing GCP public services working exactly as they do now. Add the condo environment through a **separate Cloudflare Tunnel**, rather than immediately redesigning your GCP reverse proxy.

Cloudflare Tunnel is specifically designed to publish services without requiring a public IP or inbound firewall ports. It uses an outbound connection from `cloudflared` to Cloudflare. (**Cloudflare Docs**)

---

2. What You Are Actually Building

There are four separate capabilities:

Capability	Purpose	Recommended technology
Public DNS	`app.gravelcloud.com` points to the correct service	Cloudflare DNS
Public application access	Internet users access your condo-hosted app	Cloudflare Tunnel
Private server-to-server access	GCP communicates with condo servers	Tailscale
Backup and recovery	Restore DockerLab after failure	Restic + Google Cloud Storage

These should not be confused:

> **Cloudflare Tunnel is for application ingress. Tailscale is for private administration and infrastructure connectivity. Restic is for backup and recovery.**

---

3. Will You Incur Additional Expenses?

Short answer

Possibly, but the basic architecture can be built with little or no significant additional recurring cost, assuming you already have the hardware, Cloudflare domain, Tailscale account, and GCP environment.

The largest potential new expense is **backup storage and data transfer**, not Cloudflare Tunnel itself.

Cost breakdown

Component	Additional cost expectation	Notes
Cloudflare DNS	Usually $0	Your existing domain can host more subdomains
Cloudflare Tunnel	Usually $0 for basic tunnel functionality	Available on all Cloudflare plans according to current documentation
`cloudflared`	$0	Open-source client
Tailscale	Potentially $0	Personal/free plan may be sufficient, depending on your device and feature requirements
Ubuntu	$0	Existing servers
Docker / Compose	$0	Existing infrastructure
GCP VM	$0 additional if using existing VM	Depends on CPU, memory, disk, and traffic

Google Cloud Storage	**Yes, usage-based**	Storage, operations, retrieval, and network charges may apply
Backup data transfer	**Possibly**	Depends on where backups are stored and restored
External backup drive	Optional	Useful for a third independent backup copy
Cloudflare Access	Potentially $0 for basic use	Useful for protecting admin applications, subject to current plan/feature limits

Cloudflare currently documents Tunnel as available on all plans, with outbound-only connectivity and no inbound ports required. (**Cloudflare Docs**)

Google Cloud Storage is not simply a flat monthly service. Pricing depends on storage class, stored data, operations, retrieval, replication, and network usage. (**Google Cloud**)

Example: 100 GB of backup data

If you stored **100 GB** in a standard regional Cloud Storage bucket, the raw storage cost could be relatively small—roughly a few dollars per month depending on the region and billing calculation.

But your actual bill could also include:

- Backup upload/download operations
- Restores
- Egress
- Additional backup versions
- Database dump frequency
- Retention policy
- Storage class

Important: Do not assume that "100 GB of backups" means only 100 GB billed. Retention and multiple snapshots can multiply storage usage.

My practical recommendation

Start with:

> **One encrypted backup repository in Google Cloud Storage + one local backup copy.**

Do not immediately add expensive replication, multi-region storage, or enterprise backup software.

---

4. The Most Important Design Decision: What Gets Backed Up?

A DockerLab is not just containers.

A container can usually be recreated from its image and configuration. The irreplaceable data is generally:

1. **Docker Compose files**
2. **Environment variables and secrets**
3. **Reverse proxy configuration**
4. **Database data**
5. **Application persistent volumes**

6. **Uploaded documents/files**
7. **SSL certificates, if you manage them locally**
8. **Custom scripts**
9. **Network configuration**
10. **Cloudflare and Tailscale configuration references**

Example DockerLab structure

I recommend organizing each project like this:

```
/opt/dockerlab/
│
├── projects/
│   │
│   ├── wordpress/
│   │   ├── compose.yml
│   │   ├── .env
│   │   ├── config/
│   │   └── README.md
│   │
│   ├── openproject/
│   │   ├── compose.yml
│   │   ├── .env
│   │   └── README.md
│   │
│   ├── zammad/
│   │   ├── compose.yml
│   │   └── README.md
│   │
│   └── inventree/
│       ├── compose.yml
│       ├── .env
│       └── README.md
│
├── scripts/
│   ├── backup.sh
│   ├── restore.sh
│   └── healthcheck.sh
│
├── secrets/
│   └── [protected files]
│
└── backups/
```

Do not assume that backing up `/var/lib/docker` is the best backup strategy.

Instead, back up the application configuration and persistent data in a deliberate, application-aware manner.

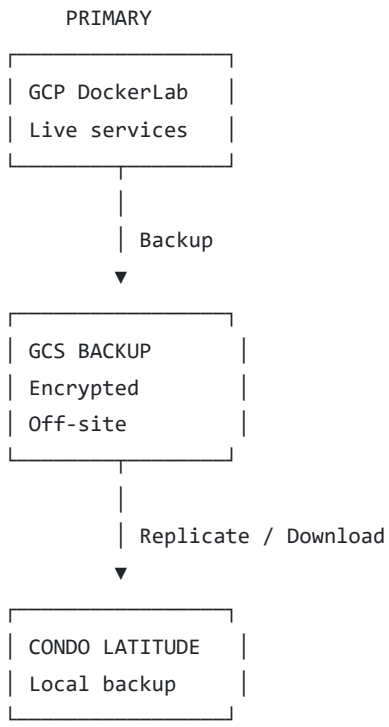
---

5. Backup Strategy I Recommend

For your environment, I would use a **3-2-1 backup strategy**:

```
3 copies of important data
2 different storage locations/media
1 copy off-site
```

Example



Or in the opposite direction:

```
CONDO DOCKERLAB
      │
      ├── Backup to GCS
      │
      └── Backup to local external disk

GCP DOCKERLAB
      │
      ├── Backup to GCS
      │
      └── Backup to local external disk
```

Backup tool recommendation: Restic

I recommend **Restic** for your first implementation because it supports:

- Encrypted repositories
- Incremental backups
- Deduplication
- Snapshots
- Retention policies

- Local repositories
- SFTP repositories
- S3-compatible repositories
- Google Cloud Storage repositories

This lets you use the same backup philosophy on both GCP and on-prem.

Alternative: BorgBackup is also excellent, especially for Linux-to-Linux backups, but Restic is a straightforward fit for a hybrid cloud setup.

---

6. Architecture for Backup and Restore

I would separate backup into two layers.

Layer A — Configuration backup

This is small and easy to restore.

```
Docker Compose files
.env files
NPM configuration
Cloudflare configuration notes
Tailscale configuration notes
Scripts
Firewall rules
System configuration
```

Layer B — Application data backup

This is the large and critical layer.

```
PostgreSQL databases
MariaDB databases
WordPress uploads
OpenProject attachments
Zammad data
InvenTree data
OpenEMR data
Other Docker persistent volumes
```

Why database dumps matter

For PostgreSQL and MariaDB, a raw file-level copy of a live database directory is not always the safest backup method.

Prefer:

```
Live database
      │
      ▼
Logical database dump
      │
```

```
        ▼
Encrypted backup
```

For example:

- PostgreSQL → `pg_dump` or `pg_dumpall`
- MariaDB → `mariadb-dump`
- WordPress → database dump + `wp-content`
- OpenEMR → database dump + application data/configuration

A database dump gives you a portable recovery artifact, rather than relying on the exact same database container and filesystem state.

---

7. Step-by-Step: Add Your On-Prem DockerLab to Cloudflare

There are two possible methods:

Method A — Cloudflare Tunnel ⭐ Recommended

```
Internet
   ↓
Cloudflare
   ↓
Cloudflare Tunnel
   ↓
On-prem cloudflared
   ↓
Docker service
```

Works behind:

- PLDT CGNAT
- Dynamic IP
- No port forwarding
- No public IPv4
- No inbound firewall opening

Method B — Tailscale-only access

```
Your laptop
   ↓
Tailscale
   ↓
On-prem server
```

This is excellent for private administration, but it does **not** make your application publicly accessible through ordinary DNS for the general internet.

For your stated goal—adding on-prem DockerLab services to `gravelcloud.com` —use **Cloudflare Tunnel**.

---

8. Step 1 — Decide Which Laptop Will Host the Tunnel

You have two Latitude Ubuntu servers.

I recommend:

```
Latitude #1 = Primary On-Prem DockerLab
Latitude #2 = Secondary / Backup / Recovery DockerLab
```

Install `cloudflared` on **Latitude #1** initially.

Why?

- One tunnel is enough to publish multiple services.
- You do not need a tunnel on every Docker host.
- You can route to services on the same machine.
- Later, you can add a second connector for redundancy.

Cloudflare supports multiple published applications through a single tunnel, with each public hostname mapped to a local origin service. (**Cloudflare Docs**)

Example

```
Latitude #1
├── Nginx Proxy Manager
├── Portainer
├── WordPress
├── OpenProject
├── InvenTree
└── cloudflared
```

---

9. Step 2 — Confirm Your On-Prem Network

On Latitude #1:

```
hostname
```

```
hostname -I
```

```
ip addr
```

```
ip route
```

Example output might look like:

```
192.168.1.50
```

Your actual condo subnet may be different.

Check Docker:

```
docker ps
```

Check listening ports:

```
sudo ss -tulpn
```

Example:

```
Portainer       9000
NPM             80 / 443 / 81
WordPress       8080
OpenProject     8081
InvenTree       8082
```

Do not expose these ports directly to the internet. The purpose of Cloudflare Tunnel is to avoid that.

---

10. Step 3 — Choose Your Tunnel Design

You have two choices.

Option A — `cloudflared` directly on Ubuntu

```
Cloudflare
    |
    ▼
cloudflared on Ubuntu
    |
    ▼
Docker services
```

Advantages

- Simple
- Easy to troubleshoot
- Independent of Docker
- Easy to restart
- Good for learning

Disadvantages

- Requires a systemd service
- Configuration is on the host

Option B — `cloudflared` as a Docker container

```
Cloudflare
    |
    ▼
cloudflared container
    |
    ▼
Docker services
```

Advantages

- Everything is managed through Docker Compose
- Easy to move to another server
- Fits your DockerLab philosophy

Disadvantages

- Docker networking must be configured correctly
- A container cannot automatically reach every other container unless the networks are connected or the host service is reachable

My recommendation for you

Since you want to learn infrastructure and recovery:

> **Start with cloudflared directly on Ubuntu. Later, containerize it after you understand the networking.**

This gives you a clean learning progression.

---

11. Step 4 — Create a Cloudflare Tunnel

Log in to your Cloudflare dashboard.

1. Open your Cloudflare account.
2. Select `gravelcloud.com`.
3. Go to **Networking** → **Tunnels**.
4. Select **Create Tunnel**.
5. Choose **Cloudflared**.
6. Name it something like:

```
condo-dockerlab
```

7. Create the tunnel.

Cloudflare's current setup flow uses the dashboard to create a tunnel and then provides installation instructions for the `cloudflared` connector. (**Cloudflare Docs**)

---

12. Step 5 — Install Cloudflared on Ubuntu

Cloudflare provides installation methods for Linux distributions.

For Ubuntu, use Cloudflare's official package instructions rather than downloading an arbitrary binary from a third-party website.

Official documentation:

Cloudflare Tunnel Setup Documentation

After installation, verify:

```
cloudflared --version
```

You should see a version number.

---

13. Step 6 — Connect the Ubuntu Server to Cloudflare

In the Cloudflare Tunnel dashboard, Cloudflare will provide a connector command containing your tunnel token.

It will look conceptually like:

```
sudo cloudflared service install <TUNNEL_TOKEN>
```

Do not copy a token from this example. Use the actual token generated by your Cloudflare dashboard.

Then check the service:

```
sudo systemctl status cloudflared
```

Enable it at boot:

```
sudo systemctl enable cloudflared
```

If necessary:

```
sudo systemctl restart cloudflared
```

View logs:

```
sudo journalctl -u cloudflared -f
```

Expected result

You want to see that the tunnel is connected.

The tunnel should establish an **outbound** connection from your condo to Cloudflare. PLDT CGNAT should not prevent this normal outbound connection.

Cloudflare documents that Tunnel does not require inbound ports or a publicly routable IP address. (Cloudflare Docs)

---

14. Step 7 — Create Your First Public Hostname

In your tunnel configuration, choose **Add Public Hostname**.

Let's use Portainer as the example.

Public hostname

```
portainer.condo.gravelcloud.com
```

Service type

```
HTTP
```

URL

If Portainer is running on the same Ubuntu host:

```
http://localhost:9000
```

Or:

```
http://127.0.0.1:9000
```

If Portainer is running on another Docker host:

```
http://192.168.1.51:9000
```

Use the actual IP and port.

Conceptual mapping

```
portainer.condo.gravelcloud.com
                │
                ▼
Cloudflare Tunnel
                │
                ▼
http://localhost:9000
```

Cloudflare's routing model is exactly this: a public hostname maps to a service reachable from the `cloudflared` connector. (**Cloudflare Docs**)

---

15. Step 8 — Add More DockerLab Services

You can add multiple public hostnames to the same tunnel.

Example:

Public hostname	Origin service	Exposure recommendation
`portainer.condo.gravelcloud.com`	`http://localhost:9000`	**Private / Access-protected**
`npm.condo.gravelcloud.com`	`http://localhost:81`	**Private / Access-protected**

wordpress.condo.gravelcloud.com	http://localhost:8080	Public if intended
openproject.condo.gravelcloud.com	http://localhost:8081	Access-protected
inventree.condo.gravelcloud.com	http://localhost:8082	Access-protected
zammad.condo.gravelcloud.com	http://localhost:8083	Access-protected

Important naming suggestion

Keep your GCP and on-prem namespaces clear.

```
GCP:
portainer.gravelcloud.com
openproject.gravelcloud.com

ON-PREM:
portainer.condo.gravelcloud.com
openproject.condo.gravelcloud.com
```

This prevents confusion about which environment you are accessing.

---

16. Step 9 — Add Cloudflare Access for Administrative Apps

This is strongly recommended.

Do not publish Portainer, Nginx Proxy Manager, pgAdmin, phpMyAdmin, SSH, or database administration interfaces to the entire internet without additional protection.

Use:

- Cloudflare Access
- Identity authentication
- MFA
- Email allowlists
- Service tokens where appropriate
- Tailscale for private administrative access

Example security model

```
portainer.condo.gravelcloud.com
              |
              ▼
      Cloudflare Access
              |
       MFA / Identity
              |
              ▼
         Portainer
```

Cloudflare documents Access as a way to enforce Zero Trust policies for services connected through Tunnel. (**Cloudflare Docs**)

My recommended exposure policy

Service	Public internet?	Recommendation
Personal WordPress	Maybe	Public if intended
Public demo application	Maybe	Public if intended
Portainer	**No**	Access + MFA or Tailscale
NPM admin	**No**	Access + MFA or Tailscale
pgAdmin	**No**	Tailscale preferred
phpMyAdmin	**No**	Tailscale preferred
PostgreSQL	**No**	Never directly publish
MariaDB	**No**	Never directly publish
SSH	**No, preferably**	Tailscale SSH or private access
OpenEMR	Carefully	Strong authentication and security review

---

17. Step 10 — Verify DNS

After adding a public hostname, Cloudflare will create or manage the DNS routing for the tunnel.

You should see a record conceptually similar to:

```
portainer.condo.gravelcloud.com
        CNAME
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.cfargotunnel.com
```

Do not manually point this record to your PLDT public IP.

That would defeat the purpose of Tunnel.

Test DNS:

```
dig portainer.condo.gravelcloud.com
```

Or:

```
nslookup portainer.condo.gravelcloud.com
```

Then test in a browser:

```
https://portainer.condo.gravelcloud.com
```

Expected traffic flow

```
Browser
   |
```

```
   ▼
Cloudflare DNS
   │
   ▼
Cloudflare Edge
   │
   ▼
Encrypted Tunnel
   │
   ▼
PLDT CGNAT
   │
   ▼
Latitude #1
   │
   ▼
Docker Portainer
```

No inbound port forwarding is required.

---

18. Step 11 — Use Tailscale for Private Connectivity

You already have Tailscale on your Ubuntu servers.

That is excellent.

Tailscale assigns devices stable private `100.x.y.z` addresses within your tailnet, which remain usable even when the device changes physical networks. (**Tailscale**)

Example:

```
GCP DockerLab
100.100.10.10

Latitude #1
100.100.10.20

Latitude #2
100.100.10.21
```

Your actual addresses will differ.

Check Tailscale

On each server:

```
tailscale status


tailscale ip
```

Test connectivity:

```
ping <TAILSCALE_IP_OF_OTHER_SERVER>
```

Or:

```
ssh <user>@<TAILSCALE_IP>
```

Example

```
ssh rigel@100.x.y.z
```

Use the actual username and Tailscale IP.

---

19. Step 12 — Decide Whether You Need a Tailscale Subnet Router

For your current setup, if Tailscale is installed directly on both Ubuntu Docker servers, you may **not need a subnet router yet**.

Direct installation is simpler:

```
GCP Ubuntu
   │
   │ Tailscale
   │
   ▼
Condo Ubuntu
```

A subnet router is useful when you want to access devices that **do not run Tailscale**, such as:

- PLDT router
- NAS
- Printer
- Other LAN servers
- IoT devices
- A Docker host without Tailscale

Tailscale subnet routers allow devices on your tailnet to reach services on a private subnet without installing Tailscale on every device. (**Tailscale**)

Example

```
Tailscale client
      │
      ▼
Latitude #1
Subnet Router
      │
      ▼
192.168.1.0/24
      │
```

```
        ├── PLDT Router
        ├── Latitude #2
        ├── NAS
        └── Other devices
```

For now:

> **Install Tailscale directly on both Latitude servers. Add subnet routing later if needed.**

---

20. Step 13 — Connect GCP and On-Prem for Backup

There are two good approaches.

Approach A — Backup through Google Cloud Storage ⭐ Recommended

```
GCP DockerLab
      │
      ▼
Encrypted Restic Backup
      │
      ▼
Google Cloud Storage
      ▲
      │
Encrypted Restic Backup
      │
      ▲
Condo DockerLab
```

This is the simplest hybrid architecture.

Why this is good

- Both environments can back up independently.
- No need for GCP to directly mount condo storage.
- No need for condo to expose SSH publicly.
- CGNAT is irrelevant.
- Backups can be encrypted before upload.
- Either environment can restore from GCS.

Example

```
GCP → GCS
Condo → GCS
GCS → GCP restore
GCS → Condo restore
```

This is the architecture I would implement first.

---

21. Step 14 — Create a Google Cloud Storage Backup Bucket

In GCP:

1. Open **Google Cloud Console**.
2. Go to **Cloud Storage** → **Buckets**.
3. Select **Create**.
4. Choose a unique bucket name.

Example:

```
gravelcloud-dockerlab-backups
```

The actual bucket name must be globally unique.

Suggested settings

Setting	Recommendation
Location	Same region as your GCP workload or a suitable backup region
Storage class	Standard initially
Access control	Uniform bucket-level access
Public access	Prevent public access
Versioning	Consider enabling
Retention	Start with a reasonable retention policy
Encryption	Restic client-side encryption + GCS security

Important

A GCS bucket should **never be publicly readable**.

Your backup repository should be private.

---

22. Step 15 — Create a Dedicated GCP Service Account

Do not use your personal GCP owner account for automated backups.

Create a dedicated service account, for example:

```
dockerlab-backup@PROJECT_ID.iam.gserviceaccount.com
```

Give it only the permissions necessary for the backup bucket.

A typical least-privilege design is:

```
Service Account
      |
      ▼
Backup Bucket Only
```

Avoid giving the backup process broad project-owner permissions.

Credential options

You can use:

- Workload Identity / attached service account on GCP
- A service account key for the on-prem server
- Another supported GCP authentication method

For your initial learning setup, a dedicated service account with a restricted key may be easier to understand, but **protect the key carefully**.

Do not place credentials in:

- Public GitHub repositories
- Docker images
- Public `.env` files
- WordPress directories
- Cloudflare public routes

---

23. Step 16 — Install Restic on Both Servers

Install Restic on:

- GCP Ubuntu server
- Condo Latitude #1
- Condo Latitude #2, if you want each server to back up independently

Example:

```
sudo apt update
```

```
sudo apt install restic
```

Verify:

```
restic version
```

Use the official Restic documentation for the current GCS backend and authentication setup.

---

24. Step 17 — Initialize a Restic Repository in GCS

On the first machine, configure the GCS repository.

Conceptually:

```
export RESTIC_REPOSITORY="gs:gravelcloud-dockerlab-backups:/"
```

Then set a strong repository password:

```
export RESTIC_PASSWORD="USE_A_LONG_RANDOM_PASSWORD"
```

Then initialize:

```
restic init
```

The exact GCS repository syntax and authentication requirements should be verified against the Restic version and backend documentation you install.

Do not use a weak password.

Critical rule

Restic encryption protects your backup contents, but **losing the repository password can make the backup unrecoverable**.

Store the password securely, separately from the backup server.

---

25. Step 18 — Back Up DockerLab Configuration

Create a backup directory:

```
sudo mkdir -p /opt/dockerlab-backup/config
```

Example configuration backup:

```
sudo rsync -a /opt/dockerlab/ /opt/dockerlab-backup/config/
```

But before doing this, review your directory structure and make sure you are not unintentionally copying:

- Temporary files
- Large caches
- Docker image layers
- Secrets you do not intend to back up
- Unnecessary logs

Better approach

Create a curated backup manifest:

```
/opt/dockerlab/projects/*/compose.yml
/opt/dockerlab/projects/*/.env
/opt/dockerlab/scripts/
/etc/cloudflared/
```

```
/etc/systemd/system/
/etc/netplan/
```

Be careful with `.env` files: they often contain database passwords and API tokens. They should be encrypted in the backup and never committed to public GitHub.

---

26. Step 19 — Back Up PostgreSQL Databases

For PostgreSQL, create a logical dump.

If PostgreSQL is running in Docker, first identify the container:

```
docker ps
```

Example:

```
postgres
```

Then inspect the database configuration.

A conceptual command:

```
docker exec -t postgres pg_dumpall -U postgres > postgres-backup.sql
```

Or use `pg_dump` for individual databases.

Important: The actual container name, database user, and database name must match your deployment.

Compress the dump:

```
gzip postgres-backup.sql
```

Then include it in your Restic backup.

Why logical dumps?

A logical dump can be restored into:

- A newer PostgreSQL container
- A different Ubuntu server
- A different Docker host
- A different cloud environment

That is much more portable than relying on raw database files.

---

27. Step 20 — Back Up MariaDB

For MariaDB:

```
docker exec mariadb mariadb-dump -u root -p --all-databases > mariadb-backup.sql
```

You will be prompted for the password.

Compress:

```
gzip mariadb-backup.sql
```

Again, use the actual container name and credentials.

Better production practice

For databases with significant activity:

- Schedule backups during a controlled window
- Use consistent dumps
- Verify dump completion
- Record the database version
- Test restoration

A backup that has never been restored is only a **backup hypothesis**.

---

28. Step 21 — Back Up Persistent Docker Volumes

List volumes:

```
docker volume ls
```

Inspect a volume:

```
docker volume inspect <volume_name>
```

Example:

```
docker volume inspect wordpress_data
```

You may see a mount path such as:

```
/var/lib/docker/volumes/wordpress_data/_data
```

Important distinction

Not every volume should be backed up the same way.

Data type	Backup method
PostgreSQL	Logical dump + required files
MariaDB	Logical dump + required files

WordPress uploads	File backup
OpenProject attachments	File backup + database
Zammad assets	File backup + database
InvenTree media	File backup + database
Cache directories	Often exclude
Temporary files	Usually exclude

For application-specific volumes, consult the application's backup documentation before assuming that copying its directory is sufficient.

---

29. Step 22 — Run the Restic Backup

Once you have prepared:

```
/opt/dockerlab-backup/
```

Run a backup:

```
restic backup /opt/dockerlab-backup
```

Then list snapshots:

```
restic snapshots
```

Check repository integrity:

```
restic check
```

Example backup flow

```
Docker containers
      |
      ▼
Database dumps
      |
      ▼
Persistent files
      |
      ▼
Compose/configuration
      |
      ▼
/opt/dockerlab-backup
      |
      ▼
```

```
Restic encryption
      |
      ▼
Google Cloud Storage
```

---

30. Step 23 — Restore GCP → On-Prem

This is one of your main goals.

Imagine your GCP DockerLab fails, and you want to recover a service on Latitude #1.

Recovery flow

```
GCP DockerLab
      ×
      |
      ▼
Google Cloud Storage Backup
      |
      ▼
Latitude #1
      |
      ▼
Restore configuration
      |
      ▼
Restore database
      |
      ▼
Restore persistent files
      |
      ▼
docker compose up -d
      |
      ▼
Test application
```

Step-by-step

1. Install Ubuntu updates.
2. Install Docker.
3. Install Docker Compose plugin.
4. Install Restic.
5. Configure access to the GCS repository.
6. Restore the backup files.
7. Restore database dumps.
8. Restore application volumes/data.
9. Restore `.env` files.
10. Start Docker services.

1. Check logs.
2. Test the application locally.
3. Update Cloudflare Tunnel origin if necessary.
4. Test the public hostname.

Restore example

```
restic snapshots
```

Select the snapshot you want.

```
restic restore latest --target /restore
```

Then copy the restored files into the appropriate DockerLab directories after verifying their contents.

Do not blindly overwrite a live DockerLab during your first restore test.

---

31. Step 24 — Restore On-Prem → GCP

The reverse direction is almost identical.

```
Condo DockerLab
      ×
      |
      ▼
Google Cloud Storage Backup
      |
      ▼
GCP Ubuntu VM
      |
      ▼
Restore configuration
      |
      ▼
Restore databases
      |
      ▼
Restore persistent files
      |
      ▼
docker compose up -d
      |
      ▼
Update GCP routing
      |
      ▼
Application restored
```

Example scenario

Your condo Latitude #1 has:

```
WordPress
OpenProject
InvenTree
Zammad
```

You can restore these onto GCP, assuming:

- The Docker images are available
- The Compose files are compatible
- The database versions are compatible
- The required persistent data was backed up
- The required secrets are available
- DNS and reverse proxy configuration are updated

Important

A successful restore requires more than the files. It requires a **reproducible environment**.

---

32. Step 25 — Make Your DockerLab Portable

This is where your learning becomes valuable.

You want to reach this state:

```
DockerLab Project
      │
      ├── compose.yml
      ├── .env.example
      ├── README.md
      ├── backup instructions
      ├── restore instructions
      └── data backup
```

Example README

```
Project: InvenTree

Primary host:
Latitude #1

Public hostname:
inventree.condo.gravelcloud.com

Docker Compose:
./compose.yml

Persistent data:
- inventree_media
```

```
- inventree_static

Database:
PostgreSQL

Backup:
- PostgreSQL logical dump
- Media directory
- Compose configuration

Restore:
1. Install Docker
2. Restore .env
3. Restore database
4. Restore media
5. Start containers
6. Verify application
```

This is how you turn DockerLab from a collection of containers into **infrastructure you can rebuild**.

---

33. Step 26 — Add Automated Backups

Once manual backup and restore work, automate.

A reasonable starting schedule:

Task	Frequency
Database dumps	Daily
Configuration backup	Daily
Persistent data backup	Daily
Restic snapshot	Daily
Backup integrity check	Weekly
Restore test	Monthly
Full disaster recovery exercise	Quarterly

Example retention policy

```
Keep:
7 daily snapshots
4 weekly snapshots
12 monthly snapshots
```

This is only an example. Adjust it based on data size and cost.

Example Restic retention command

```
restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 12 --prune
```

Run retention carefully. Understand that pruning removes data no longer referenced by retained snapshots.

---

34. Step 27 — Create a Backup Script

A simplified conceptual script:

```
#!/usr/bin/env bash

set -euo pipefail

BACKUP_DIR="/opt/dockerlab-backup"
DATE=$(date +%F)

mkdir -p "$BACKUP_DIR/$DATE"

echo "Backing up DockerLab configuration..."

rsync -a \
  /opt/dockerlab/projects/ \
  "$BACKUP_DIR/$DATE/projects/"

echo "Creating PostgreSQL dump..."

docker exec -t postgres pg_dumpall -U postgres \
  > "$BACKUP_DIR/$DATE/postgres.sql"

echo "Creating MariaDB dump..."

docker exec mariadb mariadb-dump \
  -u root -p --all-databases \
  > "$BACKUP_DIR/$DATE/mariadb.sql"

echo "Uploading to Restic..."

restic backup "$BACKUP_DIR/$DATE"

echo "Backup complete."
```

Important warnings

This is a **learning template**, not a production-ready universal script.

You should improve it with:

- Secrets handling
- Database credential management
- Error checking
- Container existence checks
- Dump compression
- Dump cleanup
- Backup logging

- Retention
- Notification on failure
- Application-specific volume selection

---

35. Step 28 — Use Tailscale for Backup Administration

You do **not** need to expose SSH publicly through Cloudflare.

Instead:

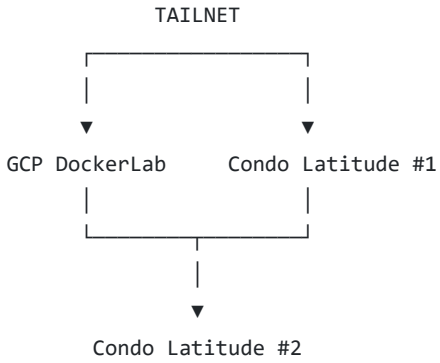
```
Your computer
      |
      ▼
Tailscale
      |
      ▼
GCP / Condo Ubuntu
```

Example:

```
ssh user@<tailscale-ip>
```

This is much cleaner than exposing port 22 to the internet.

Tailscale architecture



If you later need access to the entire condo LAN, use a subnet router. Tailscale's official documentation describes subnet routers as the mechanism for reaching non-Tailscale devices and connecting separate networks. (Tailscale)

---

36. Cloudflare vs Tailscale: Which Should Handle What?

This is the most important operational distinction.

Task	Cloudflare Tunnel	Tailscale
Public WordPress	⭐ Yes	No
Public demo app	⭐ Yes	No

Private Portainer	Possible with Access	⭐ Excellent
Private SSH	Possible	⭐ Excellent
GCP ↔ Condo server access	Not ideal as primary method	⭐ Excellent
Database replication	Possible through private networking, but not first choice	⭐ Better
Backup transfer	GCS preferred	Useful for private administration
Access from anywhere	Public app / Access	⭐ Private device access
Works behind CGNAT	⭐ Yes	⭐ Yes

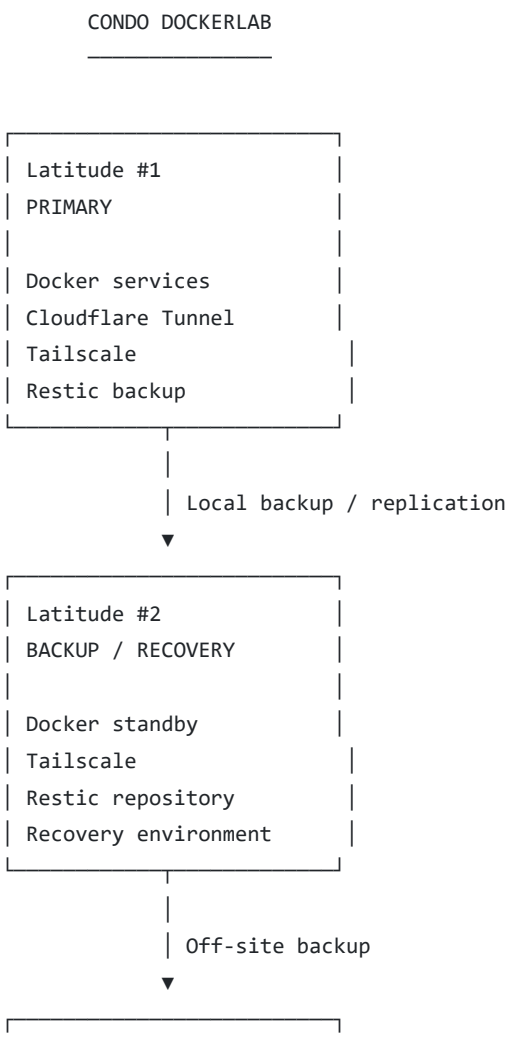
My recommendation

```
Cloudflare Tunnel → Public application access
Tailscale         → Private infrastructure access
GCS + Restic      → Backup and disaster recovery
```

---

37. A Better Long-Term Architecture for Your Two Latitudes

Because you have two laptops, I would eventually structure them like this:



```
              │ Google Cloud Storage     │
              │                          │
              │ Encrypted backup         │
              │ Long-term retention      │
              └──────────────────────────┘
```

Important clarification

Do not initially try to make both laptops an automatic active-active Docker cluster.

That introduces complexity:

- Shared storage
- Database replication
- Service discovery
- Failover
- Split-brain risk
- Persistent volume synchronization
- Networking complexity

For learning and reliability, start with:

> **Primary → Backup → Restore**

Then later explore:

> **Primary → Replicated standby → Controlled failover**

---

38. What I Would Do First, in Your Exact Situation

Given your existing GCP DockerLab, Cloudflare domain, Tailscale, and two Ubuntu laptops, I would follow this order.

Phase 1 — Connectivity

- Confirm Tailscale works between GCP and both Latitudes.
- Identify the primary on-prem Docker host.
- Confirm Docker services and ports.
- Install `cloudflared`.
- Create `condo-dockerlab` Cloudflare Tunnel.
- Publish one test service.
- Add Cloudflare Access.
- Verify from an external network.

Phase 2 — Backup foundation

- Create GCS backup bucket.
- Create restricted backup service account.
- Install Restic.
- Initialize encrypted repository.
- Back up Compose files.
- Back up one database.
- Back up one persistent volume.

- Verify snapshot exists.

Phase 3 — Restore testing

- Restore to a temporary directory.
- Restore one database.
- Restore one Docker application.
- Confirm application works.
- Document the restore procedure.

Phase 4 — Automation

- Add daily backup script.
- Add systemd timer or cron.
- Add retention.
- Add backup integrity check.
- Add failure notification.
- Test GCP → On-Prem restore.
- Test On-Prem → GCP restore.

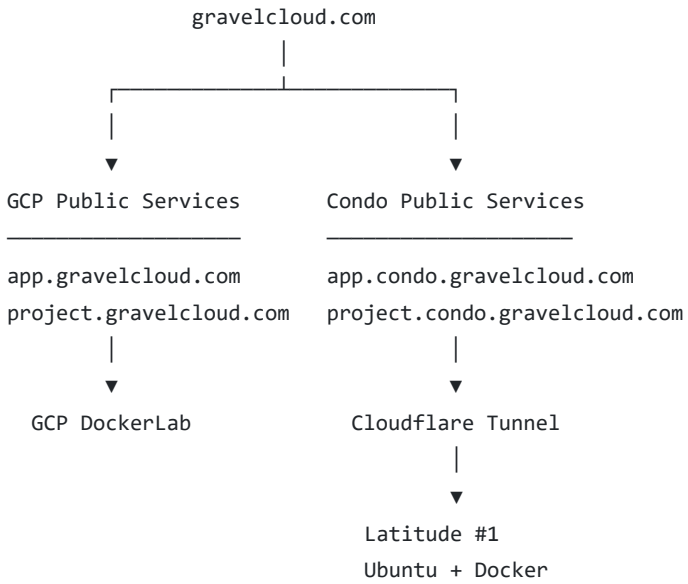
Phase 5 — Advanced hybrid lab

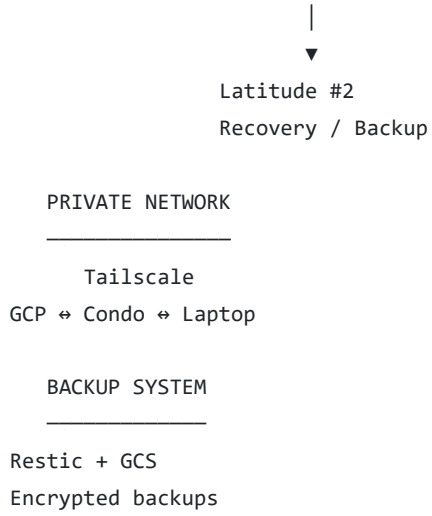
- Tailscale subnet routing.
- Private GCP ↔ condo network.
- Secondary Latitude recovery node.
- Infrastructure-as-code.
- Terraform for GCP.
- Automated DockerLab provisioning.
- Documented disaster recovery runbook.

---

39. My Recommended Final Design

If I were designing your `gravelcloud.com` environment, I would use:





Bottom line

Yes, you can add your on-prem DockerLab to Cloudflare without needing a public IP or bypassing PLDT CGNAT. The cleanest method is to install `cloudflared` on one Ubuntu Latitude, create a separate Cloudflare Tunnel, and map subdomains to your local Docker services.

For backup and restore, I would **not** make Cloudflare or Tailscale the primary backup mechanism. Use:

> **Restic + encrypted Google Cloud Storage repository + local recovery copy.**

Then use Tailscale to administer both environments privately.

That gives you a practical hybrid-cloud lab that is inexpensive, portable, and teaches you real infrastructure concepts rather than just exposing ports.