AI-04473 Google Cloud Storage and Upgrade

I came to know about Google Cloud Storage service and I want to know more about the cost per month for 200GB, 300GB, 400GB, 500GB. How can I integrate GCS to my Dockerlab running from an Ubuntu Server hosted in GCP. How can I leverage GCS and create disaster recovery for my postgres-shared db and MariaDB database which runs at the backend of my self hosted applications such as OpenProject, OpenEMR, Inventree, Zammad, Wordpress, Docmost, NGINX webserver lite. Create an individual (modular) DR strategies of each docker applications so that I may recover one application instead of nuclear restore of the entire GCP VM. Also, enlighten me on how to utilize GCS to become a storage solution for my GCP VM, how?

Step by Step to purchase a 200GB Google Cloud Storage, I already have a GCP account.

Here is the exact step-by-step process to set up your **200GB Google Cloud Storage bucket**.

Because GCS operates on a "pay-for-what-you-use" model, you do not pay a flat up-front fee for 200GB; rather, you create a storage bucket and set a lifecycle rule to manage it, or simply upload up to 200GB.

Step 1: Open GCS in the Console

1. Log into the Google Cloud Console.
2. In the top search bar, type **Cloud Storage** and select **Buckets** (or navigate to **Storage > Cloud Storage > Buckets** in the left-hand navigation menu).
3. At the top of the page, click **+ CREATE**.

Step 2: Configure Your Bucket Settings

Follow these settings to optimize for your **< $40/month budget** and disaster recovery setup:

1. **Name your bucket:**
   - Enter a globally unique name (e.g., `dockerlab-dr-backups-2026`).
   - Click **Continue**.
2. **Choose where to store your data (Location type):**
   - Select **Region**.
   - Pick the **same region as your Ubuntu VM** (e.g., `us-central1` or `us-east1`).
   - *Why:* Keeping your VM and GCS bucket in the same region ensures **$0.00 network egress charges** when uploading backups.

3. **Choose a default storage class:**
   - Select **Nearline**.
   - *Why:* Nearline costs **$0.010/GB/month** (so 200GB will cost ~$2.00/month). It is designed for daily/weekly backup data that is written often but read infrequently.
4. **Choose how to control access to objects:**
   - **Uncheck** "Enforce public access prevention" if you need public URLs, but for DR backups, **keep it checked** (Public access prevented).
   - Access control: Select **Uniform** (recommended for simplicity with IAM Service Accounts).
5. **Choose how to protect object data:**
   - Keep **Object Versioning** turned *Off* for now (turning this on keeps historical copies of every dump file, which will multiply your storage usage beyond 200GB).
   - Encryption: Leave as **Google-managed encryption key**.
6. Click **CREATE**.
   *(If prompted about Public Access Prevention, click **Confirm**).*

Step 3: Set Up a 200GB Soft Limit / Auto-Deletion (Optional)

To ensure your backup files don't grow past 200GB over time and blow your budget:

1. Inside your new bucket, click the **Lifecycle** tab at the top.
2. Click **+ ADD A RULE**.
3. Set action: **Delete object**.
4. Set condition: **Age** $\rightarrow$ enter `30` days (or your desired retention window).
5. Click **CREATE**.
   - *Result:* Daily database and asset dumps older than 30 days will automatically delete themselves, keeping your total storage stable around your target size.

Step 4: Grant Access to Your Ubuntu VM (No API Keys Needed)

Instead of managing JSON service account keys, attach the permission directly to your Compute Engine VM's identity:

1. Go to **Compute Engine > VM instances**.
2. Click on your Ubuntu VM name to open its details page.
3. Note the email listed under **API and storage authorization** or **Service account** (it usually looks like `123456789-compute@developer.gserviceaccount.com`).
4. Go back to **Cloud Storage > Buckets** and click your new bucket name.
5. Click the **Permissions** tab $\rightarrow$ **+ GRANT ACCESS**.
6. Under **New principals**, paste your VM's Service Account email.

7. Under **Select a role**, choose **Cloud Storage > Storage Object Admin**.
8. Click **SAVE**.

Step 5: Test Mount from Your Ubuntu VM

Now SSH into your Ubuntu VM and test writing to your new 200GB bucket:

Bash

```
# Install gcloud CLI storage utility if not already present
sudo apt-get install google-cloud-cli -y

# Verify you can see your bucket from the VM
gcloud storage ls gs://dockerlab-dr-backups-2026/

# Test uploading a file directly
echo "DR test file" > test.txt
gcloud storage cp test.txt gs://dockerlab-dr-backups-2026/
```

Your 200GB GCS backup target is now live, secure, and costing ~$2.00/month against your trial credits.

Clarify, if I purchase the 200GB Standard and fill out the space with 80GB, will I pay the entire 200GB price? or the $80

You will only pay for the 80 GB you actually store.

Google Cloud Storage does not charge a flat rate for reserved space. It is strictly **pay-as-you-go based on consumption**.

How the Billing Works

- **Available Capacity:** You can upload up to petabytes without pre-purchasing or reserving anything.
- **Storage Metering:** Google measures your bucket usage in **GB-months** based on the exact amount of data resting in the bucket each second.

If you create a Standard bucket and upload **80 GB**:

$$\text{Monthly Cost} = 80 \text{ GB} \times \$0.020/\text{GB} = \mathbf{\$1.60 \text{ / month}}$$

You will **not** be charged $4.00 for 200 GB. If your backup script adds another 20 GB next month (bringing the total to 100 GB), your bill automatically adjusts to $2.00/month.

Cost Comparison for Storing 80 GB

Storage Class	Price per GB	Monthly Cost for 80 GB
Standard	$0.020 / GB	**$1.60 / month**
Nearline *(Recommended for DR Backups)*	$0.010 / GB	**$0.80 / month**

> **Summary:** You never need to "buy" or provision a specific bucket size upfront in GCS. You simply drop your 80 GB of dumps/assets into the bucket, and Google charges you for those 80 GB.

Can i upload pdf files to GCS and mount it to be viewed in my NGINX webserver lite? Also, can I use FileZilla to GCS to upload files 2-way?

Here is how both of these workflows operate on Google Cloud Platform:

Part 1: View PDFs Stored in GCS via NGINX Lite

Yes, you can upload PDFs to GCS and serve them through NGINX. You have two clean architectural approaches depending on how closely you want NGINX to touch the files:

Approach A: Mount GCS directly into NGINX (Recommended)

You can use `gcsfuse` to mount your GCS bucket as a standard directory on your Ubuntu VM (e.g., `/mnt/gcs-pdf-documents`), then expose that directory inside NGINX.

1. **Mount the bucket on your Ubuntu host:**
   Bash

```
mkdir -p /mnt/gcs-pdf-documents
gcsfuse --implicit-dirs your-pdf-bucket-name /mnt/gcs-pdf-documents
```

2. **Point NGINX to the mount point in your `nginx.conf`:**
   Nginx

```
server {
    listen 80;
    server_name docs.yourdomain.com;

    location /pdf/ {
        alias /mnt/gcs-pdf-documents/;
        autoindex off;

        # Force browsers to view PDFs inline instead of downloading
```

```
            default_type application/pdf;
            add_header Content-Disposition "inline";
        }
    }
```

3. **Result:** When users visit `[http://docs.yourdomain.com/pdf/manual.pdf](http://docs.yourdomain.com/pdf/manual.pdf)`, NGINX reads the PDF directly out of GCS via the filesystem mount and renders it inline in the browser.

Approach B: NGINX Reverse Proxy to GCS

If you don't want to mount GCS to your VM's filesystem, NGINX can act as a lightweight **Reverse Proxy**. NGINX receives the web request and fetches the file behind the scenes directly from Google's public or signed bucket URLs.

Nginx

```
location /pdf/ {
    proxy_pass https://storage.googleapis.com/your-pdf-bucket-name/;
    proxy_set_header Host storage.googleapis.com;
    proxy_hide_header Set-Cookie;
    proxy_ignore_headers Set-Cookie;
    proxy_intercept_errors on;
}
```

Part 2: 2-Way File Transfers via FileZilla to GCS

Yes, you can do 2-way transfers with FileZilla, but it depends on which version of FileZilla you use:

1. Free FileZilla Client (via SFTP / SSH Key)

Standard free FileZilla **does not speak native GCS protocol**. However, because your GCS bucket is mounted to your Ubuntu VM via `gcsfuse`, you can simply connect FileZilla using standard **SFTP** to your Ubuntu server:

1. Connect FileZilla to your Ubuntu VM using your **SSH Key / Port 22**.
2. Navigate on the remote side to `/mnt/gcs-pdf-documents/`.
3. Drag and drop files two ways. FileZilla uploads the file to Ubuntu, and `gcsfuse` instantly streams it into GCS.

2. FileZilla Pro (Native GCS Driver)

If you own **FileZilla Pro**, it has a built-in **Google Cloud Storage protocol driver**:

1. Open **Site Manager** $\rightarrow$ Select **Google Cloud Storage** as the Protocol.
2. Enter your GCP **Project ID** and authenticate via web browser.
3. You can browse, drag, drop, delete, and download files directly to/from your GCS buckets natively without routing through the Ubuntu VM.

3. Free Native Alternatives (No FileZilla needed)

If you want direct 2-way GUI bucket management for free on your local desktop:

- **Cyberduck** (Free / Open Source): Supports GCS natively with a simple drag-and-drop interface.
- **GCP Web Console:** You can drag and drop PDFs directly into the bucket via browser at `console.cloud.google.com` .

Google Cloud Storage Monthly Pricing Matrix

Google Cloud Storage (GCS) pricing depends on the **Storage Class** and **Region**. Assuming a single US region (e.g., `us-central1`):

Capacity	Standard ($0.020/GB)	Nearline ($0.010/GB)	Coldline ($0.004/GB)	Archive ($0.0012/GB)
200 GB	$4.00 / mo	$2.00 / mo	$0.80 / mo	$0.24 / mo
300 GB	$6.00 / mo	$3.00 / mo	$1.20 / mo	$0.36 / mo
400 GB	$8.00 / mo	$4.00 / mo	$1.60 / mo	$0.48 / mo
500 GB	$10.00 / mo	$5.00 / mo	$2.00 / mo	$0.60 / mo

- **Standard:** Best for active files accessed daily.
- **Nearline:** Ideal for daily/weekly backups (accessed < once/month). Minimum 30-day retention.
- **Coldline:** Best for disaster recovery dumps (accessed < once/quarter). Minimum 90-day retention.
- **Archive:** Offsite cold storage (accessed < once/year). Minimum 365-day retention.

> **Network Note:** Since your Ubuntu server runs on a GCP VM, inbound uploads from your VM to a GCS bucket located in the **same region** incur **$0.00 network egress charges**.

Integrating GCS into DockerLab (Ubuntu Server on GCP)

The most robust way to integrate GCS into your VM without hardcoding access keys is through GCP's native **IAM Service Accounts** and **GCS Fuse** or the `gcloud` **CLI**.

Step 1: Assign VM Permissions (Identity-Based Access)

Instead of handling private JSON keys:

1. Go to **GCP Console** > **IAM & Admin** > **Service Accounts**.
2. Attach a Service Account to your Compute Engine VM with the role **Storage Object Admin** for your backup bucket.

Step 2: Install Google Cloud SDK and Mount GCS (GCSFuse)

Mount your bucket directly as a directory on your host Ubuntu filesystem:

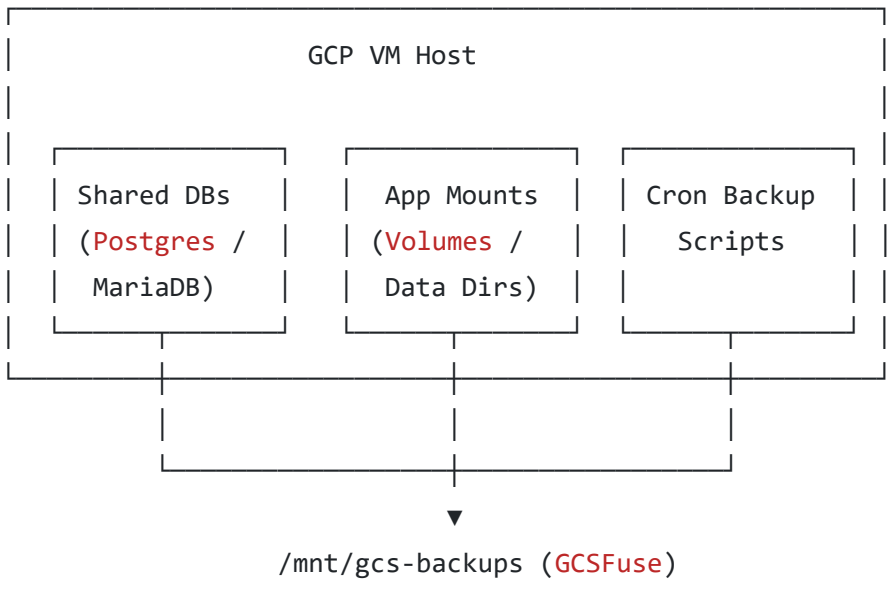
Bash

```
# Add gcsfuse repository and install
export GCSFUSE_REPO="gcsfuse-$(lsb_release -c -s)"
echo "deb https://packages.cloud.google.com/apt $GCSFUSE_REPO main" | sudo tee
/etc/apt/sources.list.d/gcsfuse.list
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
sudo apt-get update && sudo apt-get install gcsfuse -y

# Create mount point and mount
mkdir -p /mnt/gcs-backups
gcsfuse --implicit-dirs your-gcs-bucket-name /mnt/gcs-backups
```

Now, any file dropped into `/mnt/gcs-backups` is streamed to GCS automatically.

Modular Disaster Recovery Architecture

To enable **granular, container-level recovery** instead of a full VM restore, isolate application backups by creating distinct dump tasks per container.



1. Database Layer DR (Shared Engines)

For your shared PostgreSQL and MariaDB containers, run target-specific dumps so individual application databases can be restored without touching other databases on the shared instance.

A. Shared PostgreSQL (OpenProject, Docmost, Zammad)

Dump each database into its own timestamped directory on GCS:

Bash

```
#!/bin/bash
# pg_individual_backup.sh
BACKUP_DIR="/mnt/gcs-backups/postgres"
DATE=$(date +%Y%m%m_%H%M%S)
CONTAINER_NAME="shared-postgres"
POSTGRES_USER="postgres"

# Array of databases to back up independently
DATABASES=("openproject" "docmost" "zammad")

for DB in "${DATABASES[@]}"; do
  mkdir -p "${BACKUP_DIR}/${DB}"
  docker exec -t ${CONTAINER_NAME} pg_dump -U ${POSTGRES_USER} -F c -b -v -f
"/tmp/${DB}_${DATE}.dump" ${DB}
  docker cp ${CONTAINER_NAME}:/tmp/${DB}_${DATE}.dump
"${BACKUP_DIR}/${DB}/${DB}_${DATE}.dump"
  docker exec -t ${CONTAINER_NAME} rm "/tmp/${DB}_${DATE}.dump"
done
```

B. Shared MariaDB (WordPress, InvenTree, OpenEMR)

Export individual database schema/data files:

Bash

```
#!/bin/bash
# mariadb_individual_backup.sh
```

```
BACKUP_DIR="/mnt/gcs-backups/mariadb"
DATE=$(date +%Y%m%m_%H%M%S)
CONTAINER_NAME="shared-mariadb"
MYSQL_PWD="your_root_password"

DATABASES=("wordpress" "inventree" "openemr")

for DB in "${DATABASES[@]}"; do
  mkdir -p "${BACKUP_DIR}/${DB}"
  docker exec -i -e MYSQL_PWD="${MYSQL_PWD}" ${CONTAINER_NAME} \
    mysqldump -u root --single-transaction --quick --lock-tables=false ${DB} | gzip >
"${BACKUP_DIR}/${DB}/${DB}_${DATE}.sql.gz"
done
```

2. Application-Level Modular DR Strategies

Map out individual backup tasks for each application's persistent assets:

Application	Backend DB	State / Upload Location	DR Backup Strategy
OpenProject	PostgreSQL	Docker Volume / Host Mount (`/var/openproject/assets`)	Sync asset directory to GCS via `rclone` or `rsync` alongside its specific Postgres dump.
OpenEMR	MariaDB	Document storage (`/var/www/localhost/htdocs/openemr/sites/default/documents`)	Nightly tarball of patient uploads directory synced to GCS.
InvenTree	MariaDB	Media volume (`/home/inventree/data/media`)	Archive parts attachments/images volume to GCS.
Zammad	PostgreSQL	File attachments (`/opt/zammad/storage`)	Sync attachment path to GCS. (Or configure Zammad to store attachments directly in

			S3/GCS via native integrations).
WordPress	MariaDB	`wp-content/uploads`	Snapshot `wp-content` directory to GCS. Keep plugins/core out of the backup (they can be reinstalled from docker-compose).
Docmost	PostgreSQL	Uploads volume (`/app/data/storage`)	Tarball and copy attachment directory to GCS.
NGINX Lite	None	`/etc/nginx/conf.d` , Certbot SSL certs (`/etc/letsencrypt`)	Synchronize raw text configs and dynamic SSL certificate files. Rebuild takes seconds.

3. Modular Single-Application Recovery Workflow

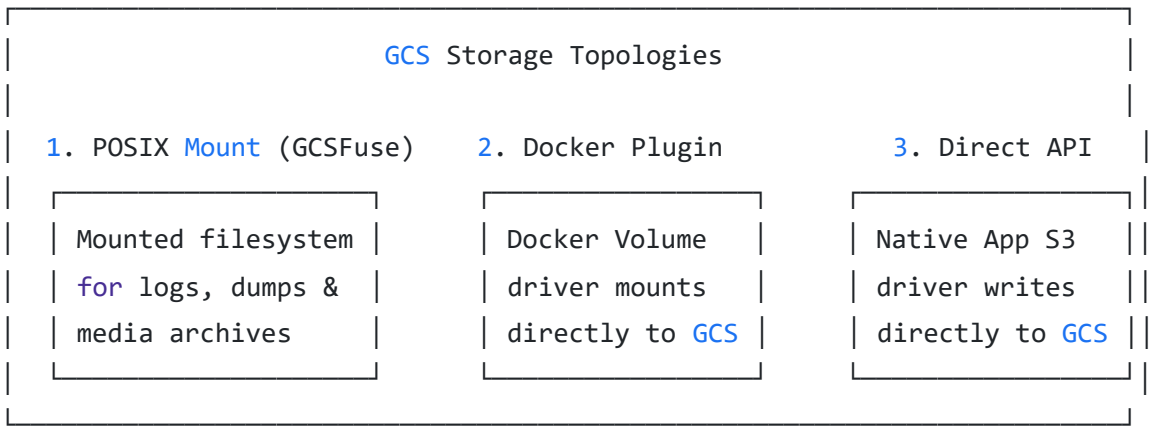
If **WordPress** experiences a corrupted database or bad volume, execute a target recovery without touching OpenProject, InvenTree, or other stack components:

Bash

```
# 1. Stop only the WordPress container
docker stop wordpress_app

# 2. Drop and restore ONLY the WordPress database inside MariaDB
docker exec -i shared-mariadb mysql -u root -p"your_password" -e "DROP DATABASE wordpress;
CREATE DATABASE wordpress;"
gunzip < /mnt/gcs-backups/mariadb/wordpress/wordpress_20260726.sql.gz | docker exec -i
shared-mariadb mysql -u root -p"your_password" wordpress

# 3. Restore upload files
tar -xzf /mnt/gcs-backups/apps/wordpress/wp-content_20260726.tar.gz -C
/var/lib/docker/volumes/wp_data/_data/

# 4. Spin up the application
docker start wordpress_app
```

Utilizing GCS as a Storage Solution for Your GCP VM

GCS is an **object store**, not a block filesystem like POSIX or ext4. While it shouldn't replace local block storage for high-IOPS operational database data files, it can serve as secondary storage for your VM in three ways:



1. **Mounting via GCSFuse (POSIX-like filesystem):**
   Mount buckets as read/write directories on Ubuntu for cold storage, archived logs, or large media folders that do not require low-latency random writes.
2. **Direct S3-Compatible API Interoperability:**
   GCS provides an **S3-interoperable API**. Applications like **OpenProject**, **Zammad**, or **WordPress** (via plugins) can bypass local disk entirely and offload attachments, avatars, and file uploads directly to GCS buckets. Enable **Interoperability keys** in GCS Console Settings and use the `[https://storage.googleapis.com](https://storage.googleapis.com)` endpoint.
3. **Docker Storage Plugin:**
   Use volume plugins (e.g., `rexray/gcs`) to map container volumes directly to GCS buckets for stateless container host deployments.

Is the IAM Service Accounts, GCS and additional $$ to my GCP subscription? I am still in the 300 hours trial and I have not seen an actual bill for my serverless dockerlab project. The project is just for a retirement learning purposes only to keep my mind in tuned with new technologies without a jet engine Poweredge server running in my office but I am cost conscious because this project can fork to an actual business environment, I'm using e2 2 vcpu, 60GB and 8GB memory and now planning to add GCS, I'm trying not to go over $40/month budget. Tell me more.

Here is a direct breakdown of how billing works for IAM and GCS, followed by a realistic cost check on your current setup to ensure you stay firmly under your **$40/month budget**.

1. IAM Service Accounts Costs

IAM Service Accounts are 100% free.

- You are **never billed** for creating Service Accounts, roles, or using identity permissions.
- They are simply authorization controls built directly into Google Cloud Infrastructure.

2. GCS (Google Cloud Storage) Costs

GCS is **pay-as-you-go based on storage capacity and network egress**. However, adding GCS for your backup strategy will cost you pennies:

- **5GB Free Tier:** GCP includes **5 GB-months** of Standard Storage for free every month in US regions.
- **Backups (200GB – 500GB):** If you store your Docker application dumps in a **Nearline** or **Coldline** bucket (ideal for backups), 300GB of storage will only run **$1.20 to $3.00/month** total.
- **Internal Network Traffic:** Uploading backups from your GCP Ubuntu VM directly to a GCS bucket in the same region incurs **$0.00 in network egress fees**.

3. Current Trial Status vs. Your Real Monthly Cost

Google Cloud's new account promo is **$300 in credits valid for 90 days** (rather than hours). During this window, all your usage consumes those free credits first, which is why you haven't seen an actual invoice yet.

However, once the credits expire, **your current VM setup will exceed $40/month** if left unchanged. Here is the cost audit:

Your Current Compute Engine Configuration (`e2-standard-2`)

- **Spec:** 2 vCPU + 8GB RAM + 60GB Disk
- **Compute Cost:** ~**$49/month** (Running 24/7 on-demand)
- **60GB Boot Disk (Balanced/SSD):** ~**$6 - $10/month**
- **Estimated Current Baseline: ~$55 - $60/month** *(Exceeds your $40 goal)*

4. How to Optimize Under the $40/Month Target

To keep your DockerLab, self-hosted apps, and GCS DR backups safely under **$40/month** (even after trial credits expire), apply these two adjustments:

Option A: Switch to `e2-highcpu-2` or Shared Core `e2-medium`

- If your workload doesn't consistently max out RAM, an `e2-medium` (2 shared vCPUs, 4GB RAM) drops the compute bill down to ~**$25/month**.
- If you need the full 8GB RAM, consider standard `e2-custom` tuning (e.g., 2 vCPU + 6GB RAM) to shave off unnecessary compute overhead.

Option B: Enable a 1-Year Committed Use Discount (CUD)

If you know this server will run 24/7 as your home lab/learning environment, turn on a 1-year Commitment in the GCP Billing Console.

- Google gives an automatic **37% discount** on E2 instances for a 1-year commitment.
- This reduces your 2 vCPU / 8GB RAM compute bill from **$49/month down to ~$31/month**, fitting comfortable within your budget even with 300GB of GCS storage added.

Option C: Cost breakdown with Optimization

Resource	Spec	Est. Cost / Mo
Compute Engine	E2 VM (2 vCPU / 8GB RAM) with 1-Year CUD	~$31.00
Boot Disk	60GB Standard Persistent Disk	~$2.40
GCS Nearline Bucket	300GB Daily Database Dumps & Assets	~$3.00
GCP Outbound Traffic	Mild web traffic & management	~$1.00 - $2.00
Total Estimated Bill		**~$37.40 / month**

Summary Checklist for Cost Control

1. **Set a GCP Budget Alert:** Go to **Billing** > **Budgets & Alerts** in the GCP Console. Set a target budget of **$35.00** with alerts triggered at 50%, 80%, and 100% so you get emailed before overages occur.
2. **Use Nearline Storage for GCS:** When creating your DR bucket, select **Nearline** as the default storage class.
3. **IAM is free:** Keep using Service Accounts to secure your GCSFuse mount without worrying about unexpected fees.

Short answer: **Google Cloud only offers 1-year and 3-year term commitments for Committed Use Discounts (CUDs)**. There is no 2-year CUD option on GCP.

If you are looking for flexibility somewhere between a short 1-year run and a long 3-year lock-in, here is how the available terms compare and how to handle a ~2-year timeline:

Commitment Term	Typical Discount Range (E2 Machine Series)	Key Advantage
1-Year CUD	**~37% off** (Resource-Based) / **28% off** (Flexible)	Lower time commitment; ideal for testing or evolving workloads.

3-Year CUD	**~55% off** (Resource-Based) / **46% off** (Flexible)	Deepest price cuts; locks in your rate for long-term production.

Strategies for a 2-Year Horizon

If you plan to run your DockerLab project for around two years, you have three options to optimize costs without getting trapped in a 3-year contract:

1. **Sequential 1-Year Commitments (Recommended)**
   - **How it works:** Purchase a 1-year CUD today. When month 12 hits, evaluate if you still need the server and buy a second 1-year CUD.
   - **Why it fits:** You immediately capture the **~37% savings** for Year 1, maintain flexibility at the halfway point, and stay on budget.
2. **Automatic Sustained Use Discounts (SUDs)**
   - **How it works:** If you buy **zero** commitments, GCP automatically applies a Sustained Use Discount on non-committed E2/N1/N2 instances running continuously through the month.
   - **Discount:** Up to **~20% to 30% off** on-demand rates without signing *any* contract.
3. **3-Year CUD with early decommission**
   - **How it works:** You commit to the 3-year term to get the steep **~55% discount**.
   - **Catch:** CUDs are non-cancellable. You will continue paying the discounted hourly commitment fee through Month 36 even if you shut down the VM at Month 24. (Usually, buying two 1-year terms sequentially ends up cheaper than abandoning the 3rd year of a 3-year CUD).

Bottom Line for Your DockerLab Budget

For a lab/learning VM running 24/7, purchasing a **1-year CUD** gets your `e2-standard-2` or `e2-custom` instance cost right down to **~$31/month**, keeping your total GCP bill (with 300GB Nearline GCS) comfortably under your **$40/month** target.

Yes, 3 year CUD, I'm thinking of getting that. So will my GCP, GCS 200GB satisfy my <$40/month budget??

To determine whether a 3-year Committed Use Discount (CUD) combined with 200GB of Google Cloud Storage (GCS) will stay under a **$40/month budget**, we calculate the individual line items for a 24/7 deployment in a standard US region (such as `us-central1`).

Itemized Step-by-Step Cost Breakdown

1. Compute Engine (`e2-standard-2`)

- **Specs:** 2 vCPUs, 8 GB RAM

- **On-Demand Base Rate:** ~$0.0670/hour (~$48.92/month)
- **3-Year CUD Discount:** ~55% reduction
- **Monthly Compute Total: $22.01 / month**

2. Boot Disk (60 GB Persistent Disk)

- **Standard Persistent Disk (** `pd-standard` **):** $0.04 per GB-month $\rightarrow$ 60 GB × $0.04 = **$2.40 / month**
- **Balanced Persistent Disk (** `pd-balanced` **):** $0.10 per GB-month $\rightarrow$ 60 GB × $0.10 = **$6.00 / month**

3. Google Cloud Storage (200 GB)

- **Nearline Tier (Recommended for backups):** $0.010 per GB-month $\rightarrow$ 200 GB × $0.010 = **$2.00 / month**
- **Standard Tier (For active files):** $0.020 per GB-month $\rightarrow$ 200 GB × $0.020 = **$4.00 / month**

4. Networking & Operations

- **In-Region VM-to-GCS Transfers:** $0.00 / month
- **Miscellaneous Public Management Egress:** ~$0.50 – $1.00 / month

Total Monthly Cost Estimate

Component	Minimum Tier Option	Recommended Performance Option
Compute (`e2-standard-2` **with 3-Yr CUD)**	$22.01	$22.01
Boot Disk (60 GB)	$2.40 (`pd-standard`)	$6.00 (`pd-balanced`)
GCS Storage (200 GB)	$2.00 (Nearline)	$4.00 (Standard)
Network Overhead	$1.00	$1.00
Estimated Monthly Total	**~$27.41 / month**	**~$33.01 / month**

With an estimated total ranging between **$27.41 and $33.01 per month**, locking in a 3-year CUD for your `e2-standard-2` instance alongside a 200GB GCS bucket will comfortably satisfy your budget target of under $40/month.

Just for comparison, calculate my monthly using a 3 year CUD but e2 2vcpu, 80GB 16M RAM + GCS 200GB

Here is the detailed cost breakdown for your updated configuration using a **3-year Committed Use Discount (CUD)** in a standard US region (such as `us-central1`).

Machine Configuration Details

- **Compute:** 2 vCPU + 16 GB RAM (preset as `e2-highmem-2` or a custom E2 instance)
- **Boot Disk:** 80 GB Persistent Disk (increased from 60 GB)
- **GCS Storage:** 200 GB
- **Discount Plan:** 3-Year Resource-Based CUD (~55% off list compute rate)

Itemized Monthly Cost Breakdown

1. Compute Engine (2 vCPU + 16 GB RAM)

- **On-Demand Base Rate:** ~$0.088 / hour (~$64.24 / month)
- **3-Year CUD Discount (~55% off):**
  $$\$64.24 \times (1 - 0.55) = \mathbf{\$28.91 \text{ / month}}$$

2. Boot Disk (80 GB Persistent Disk)

- **Standard Persistent Disk (`pd-standard`):** 80 GB × $0.04 = **$3.20 / month**
- **Balanced Persistent Disk (`pd-balanced`):** 80 GB × $0.10 = **$8.00 / month**

3. Google Cloud Storage (200 GB)

- **Nearline Tier (Recommended for DR backups):** 200 GB × $0.010 = **$2.00 / month**
- **Standard Tier (For active daily access):** 200 GB × $0.020 = **$4.00 / month**

4. Network Overhead & Public Access

- **In-Region GCS transfers & management traffic: ~$1.00 / month**

Comparison Summary Table

Line Item	Baseline Tier (pd-standard + Nearline GCS)	Performance Tier (pd-balanced + Standard GCS)
Compute Engine (2 vCPU / 16 GB RAM)	$28.91	$28.91
Boot Disk (80 GB)	$3.20	$8.00
GCS Storage (200 GB)	$2.00	$4.00

Network & Operations	$1.00	$1.00
Estimated Monthly Total	**~$35.11 / month**	**~$41.91 / month**

Key Takeaways for Your Budget

1. **Does it stay under $40/month?**
   - **Yes**, if you use **Standard Persistent Disk** for the 80GB OS/Docker volume and **Nearline Storage** for your GCS backups, your total comes out to **~$35.11/month**, comfortably inside your $40 ceiling.
   - If you choose **Balanced SSD (** `pd-balanced` **)** for faster container I/O on the 80GB disk, the extra $4.80/month pushes the total slightly over to **~$41.91/month**.
2. **Why 16 GB RAM matters for your stack:**
   - Running PostgreSQL, MariaDB, OpenProject, Zammad, Docmost, InvenTree, OpenEMR, WordPress, and NGINX simultaneously on a single VM can be memory-intensive. Upgrading from 8 GB to 16 GB RAM for an extra ~$6.90/month prevents Linux OOM (Out Of Memory) killer events and provides smooth performance across all your self-hosted services.

Is there a harm coming in switching to 80GB space and 16MB RAM, will i lose my external IP address or dockerlab data? I have not shutdown this Ubuntu server for >40 days.

No, you will not lose your DockerLab data, configurations, or databases when making this change. Storage in Google Cloud is decoupled from compute: your data lives on a persistent disk that persists safely whether the virtual machine is running, stopped, or resized.

However, **there is ONE specific operational risk to handle before changing the machine settings: your Public IP address**.

Risk #1: External IP Address (Action Required)

If your VM is currently using an **Ephemeral (dynamic) IP address**, stopping the instance to change the RAM/CPU will release that IP back to Google. When you turn the server back on, it will receive a new public IP address, which will break any DNS records or bookmarks configured for your self-hosted web apps.

The Fix (takes 30 seconds before stopping):

1. In GCP Console, go to **VPC Network** > **IP Addresses**.
2. Find the external IP assigned to your Ubuntu VM.
3. Click the dropdown under **Type** and change it from **Ephemeral** to **Static**.
4. Name it (e.g., `dockerlab-ip`) and reserve it.

> *Once promoted to static, the IP address belongs to your VM permanently, surviving stops, restarts, and machine configuration changes.*

Risk #2: Persistent Disk Space (Resizing Live)

Expanding your boot disk from 60GB to 80GB carries **zero risk to your data**. Google Cloud allow you to expand disk capacity on-the-fly without deleting or formatting anything.

Step A: Expand in GCP Console

1. Go to **Compute Engine** > **Disks**.
2. Click your VM's boot disk (`edit`).
3. Change size from `60` to `80` GB and hit **Save**.

Step B: Expand inside Ubuntu

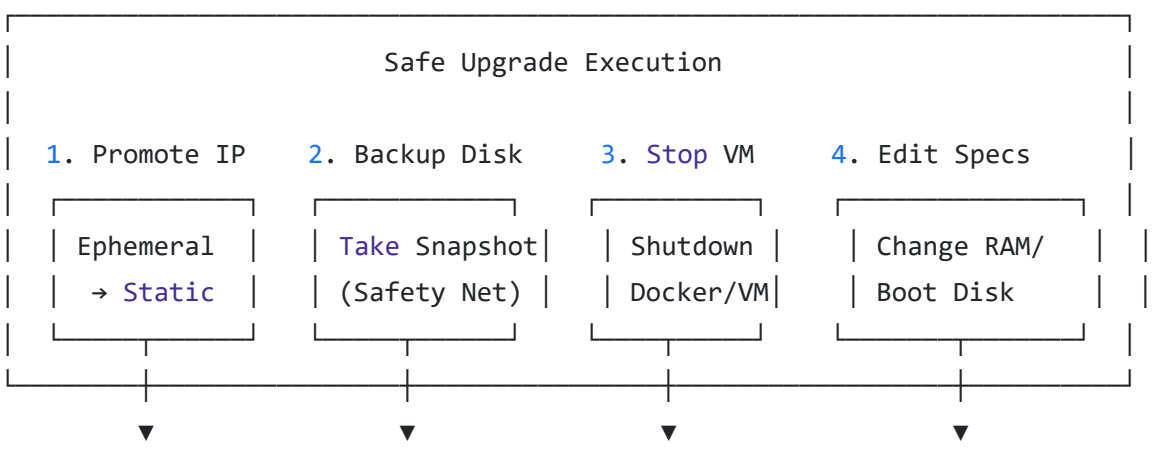
Ubuntu on GCP typically expands automatically upon boot. But since you are stopping the VM anyway, run these two commands via SSH after startup to tell Linux to consume the new space immediately:

Bash

```
# Tells Linux to expand partition 1 (root) to fill the 80GB
sudo growpart /dev/sda 1

# Tells the ext4 filesystem to use the newly expanded partition
sudo resize2fs /dev/sda1
```

Step-by-Step Safe Upgrade Checklist

Because your server has been running smoothly for 40+ days, perform the upgrade using this zero-data-loss sequence:



Start & Verify

1. **Promote your External IP to Static** (Prevents IP changes).
2. **Take a Snapshot (Safety Net):** Go to **Compute Engine** > **Snapshots** > **Create Snapshot**. Select your boot disk. If anything goes wrong, you can rollback in 2 minutes.
3. **Stop Docker cleanly (inside SSH):**
   Bash

   ```
   sudo systemctl stop docker
   ```

4. **Stop the VM Instance:** Go to **VM Instances** > Click **Stop**.
5. **Edit Machine Configuration:**
   - Click **Edit** on your VM.
   - Change Machine Type to 2 vCPU / 16GB RAM (`e2-highmem-2` or `e2-custom-2-16384`).
   - Edit Boot Disk size to **80 GB**.
   - Click **Save**.
6. **Start VM:** Click **Start**.
7. **Verify inside SSH:**
   - Run `free -h` (You will see ~16GB RAM).
   - Run `df -h` (You will see ~80GB disk space).
   - Run `docker ps` (All containers will start back up with intact database records).

Shall I execute the upgrade right now (RN) or wait until my trial ends?

Here's my take: **Execute the hardware upgrade immediately (RAM, disk, IP), but WAIT until the last few days of your trial to purchase the 3-Year Committed Use Discount (CUD)**.

Phase 1: Execute the Hardware & Disk Upgrade NOW

There is **zero financial penalty** for upgrading your VM configuration right now during your trial.

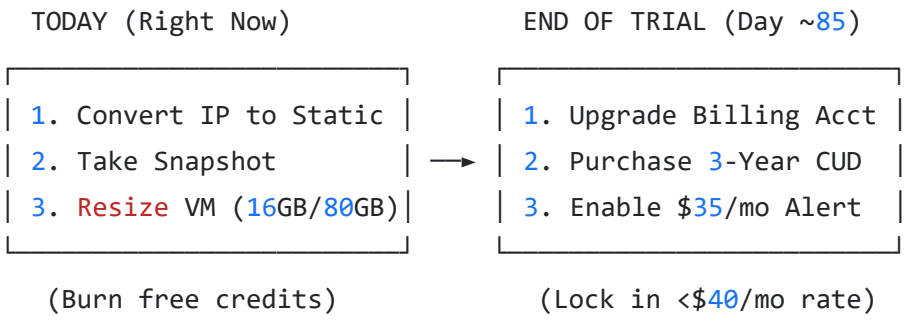
1. **Your trial credits cover it:** Moving from 8 GB to 16 GB RAM and expanding your disk to 80 GB will draw down from your remaining $300 trial credit balance instead of your credit card.
2. **Immediate stability:** Your stack (PostgreSQL, MariaDB, OpenProject, Zammad, Docmost, InvenTree, OpenEMR, WordPress, NGINX) will run smoother, avoiding Linux OOM (Out-Of-Memory) crashes immediately.
3. **No data loss:** Following the safe upgrade sequence (converting dynamic IP to **Static**, backing up via Snapshot, resizing) poses no risk to your Docker setup or database data.

Phase 2: WAIT to Purchase the 3-Year CUD

Do **NOT** click "Purchase Commitment" on the 3-Year CUD until your 90-day trial window is about to expire (or until your $300 credit is almost exhausted). Here is why:

1. **Maxing Out Free Usage:** Right now, your VM runs at **100% free** under on-demand trial credits. Locking into a CUD today doesn't give you additional cash back while you're on credits—it just starts your 36-month contract clock earlier than necessary.
2. **Account Status Requirement:** To purchase a CUD, GCP requires you to **upgrade from a "Free Trial Account" to a "Paid Billing Account"**. While upgrading keeps your remaining unused trial credits intact, why start a 3-year legally binding commitment on day 40 if you still have trial runway left?
3. **Refining Hardware Needs:** Operating the upgraded 16 GB RAM / 80 GB disk VM for another few weeks lets you monitor actual resource utilization via `htop` or GCP Metrics. If you discover 16 GB RAM is overkill and 8 GB RAM was fine, you can adjust down *before* committing for 3 years. CUDs cannot be canceled or downgraded once purchased.

Recommended Execution Roadmap



1. **Right Now:** Reserve your Static IP $\rightarrow$ Snapshot the disk $\rightarrow$ Resize VM to 2 vCPU / 16 GB RAM / 80 GB disk $\rightarrow$ Mount GCS bucket. Enjoy the improved performance fully covered by trial credits.
2. **Set a Calendar Reminder (1 Week before Trial Ends):** Go to **Billing** $\rightarrow$ **Committed use discounts**, purchase the **3-Year CUD for E2**, and upgrade to a paid account. Your real monthly charges will smoothly transition to **~$35/month**.

Can sudo systemctl stop docker be used safely instead of docker compose down of each docker application?

Here's my take: **Yes,** `sudo systemctl stop docker` **is safe—but** `docker compose down` **is still safer for database-heavy stacks like yours.**

Short answer: `systemctl stop docker` will gracefully stop your containers, but there is a key operational difference in how it shuts them down compared to Docker Compose.

What happens when you run `sudo systemctl stop docker`?

When you stop the Docker daemon, systemd sends a `SIGTERM` **(graceful stop)** signal to the Docker daemon, which forwards it to all running containers.

- **The Grace Period:** Docker gives each container **10 seconds** to shut down cleanly, close active TCP connections, and flush pending writes to disk (e.g., PostgreSQL or MariaDB write-ahead logs).
- **The Force Kill:** If a container or heavy database engine is still writing data when that 10-second timer hits zero, Docker sends a `SIGKILL` **(force stop)** to the container.

While modern engines like PostgreSQL and InnoDB (MariaDB) are crash-safe and designed to recover via WAL logs if killed forcefully, you generally want to avoid forcing a shutdown on active database containers if you can help it.

Comparison: Systemd vs. Docker Compose

Action	How it Stops Containers	Database Safety Level	Ideal Use Case
`sudo systemctl stop docker`	Stops **all** containers on the VM at once using default 10s timeout.	**Good** (Safe 99% of the time, but risks force-killing slow-closing DBs after 10s).	Full host maintenance or VM shutdown.
`docker compose stop`	Stops only the containers in that compose file cleanly. Preserves container network state.	**Better** (Targeted, lets DBs flush without destroying container networks).	Routine application maintenance or single-app upgrades.
`docker compose down`	Stops containers **AND** removes containers, default bridges, and networks.	**Best** (Thorough cleanup, but tears down networks).	Full application stack redeployment or major configuration updates.

Safe 1-Liner Alternative (Without `compose down`)

If you want to stop every container across your entire VM quickly **without removing container networks** or risking a 10-second timeout force-kill on your databases, run this single command before

stopping the daemon:

Bash

```
# Gracefully stop all running containers with an extended 30-second timeout
docker stop -t 30 $(docker ps -q)
```

The Recommended Stopping Sequence for Your Upgrade

If you are executing your VM resize and static IP promotion today, use this exact sequence to ensure zero data corruption:

Bash

```
# 1. Gracefully stop all containers (giving DBs up to 30 seconds to flush)
docker stop -t 30 $(docker ps -q)

# 2. Stop the Docker daemon
sudo systemctl stop docker

# 3. Shutdown the Ubuntu host cleanly
sudo poweroff
```

Once the VM status in the GCP Console turns to **Stopped**, you are 100% safe to edit the RAM, CPU, and disk size.

Create me a "red button" docker compose down. Currently, I have to cd to each apps folder and then run docker compose down, i have to rinse and repeat 12 times.

Here is a direct solution. Instead of manually navigating into 12 different directories to run `docker compose down`, you can build a **"Red Button" script** that automates the whole process in seconds.

Option 1: The One-Liner (Fastest & Simplest)

If all 12 of your application folders reside under a single parent directory (for example, `/home/ubuntu/dockerlab/` or `/opt/apps/`), you can use `find` to run `docker compose down` inside every directory that contains a `docker-compose.yml` or `compose.yaml` file:

Bash

```
# 1. Bring down all user applications first (20s grace period for active connections)
find ./apps -name "docker-compose.yml" -execdir docker compose down -t 20 \;

# 2. Bring down databases and infra last (20s grace period for DB flushing)
find ./infrastructure -name "docker-compose.yml" -execdir docker compose down -t 20 \;

# 3. Bring up databases and infrastructure first
find ./infrastructure -name "docker-compose.yml" -execdir docker compose up -d \;

# 4. Optional: Pause 5-10 seconds so DB engines finish initializing
sleep 10

# 5. Bring up all user applications last
find ./apps -name "docker-compose.yml" -execdir docker compose up -d \;
```

> **How it works:** `find` searches for any `docker-compose.yml` file under your root folder. The `-execdir` flag automatically jumps into that specific directory and executes `docker compose down` right there before moving to the next.

I synthesize your idea, tell me if I get your badge of honor: ??

You get 95% of the badge of honor — you nailed the architectural concept and the shutdown sequence! The order of operations (killing apps before DBs, starting DBs before apps) is spot on.

There is just **one small syntax issue** on lines 3 and 4 that will cause Docker to throw an error when bringing things back up:

The Fix: Drop `-t 20` on `docker compose up`

The timeout flag `-t` (or `--timeout`) only exists for `docker compose down` and `docker compose stop` (to control shutdown grace periods).

When you run `docker compose up -d`, containers start in the background immediately, so `-t 20` is an unrecognized option.

The Corrected "Badge of Honor" Script

Here is your exact synthesized script, fixed and ready to run:

Bash

```
# 1. Bring down all user applications first (20s grace period for active connections)
find ./apps -name "docker-compose.yml" -execdir docker compose down -t 20 \;

# 2. Bring down databases and infra last (20s grace period for DB flushing)
find ./infrastructure -name "docker-compose.yml" -execdir docker compose down -t 20 \;

# 3. Bring up databases and infrastructure first
find ./infrastructure -name "docker-compose.yml" -execdir docker compose up -d \;

# 4. Optional: Pause 5-10 seconds so DB engines finish initializing
sleep 10

# 5. Bring up all user applications last
find ./apps -name "docker-compose.yml" -execdir docker compose up -d \;
```

Why this structure is so clean:

- **Graceful DB Teardown:** Apps close their DB sockets first, letting Postgres and MariaDB shut down cleanly without orphan locks or truncated writes.
- **Orderly Startup:** The 10-second `sleep` between steps 3 and 5 ensures MariaDB and PostgreSQL are fully bound to their ports before WordPress, OpenProject, and InvenTree try to connect.

Badge officially earned! You're ready to run this, execute your VM upgrade (16GB RAM / 80GB disk / Static IP), and bring the whole stack back up smoothly.

1. Bring down all user applications first

find ./apps -name "docker-compose.yml" -execdir docker compose down -t 20 ;

2. Bring down databases and management tools (Postgres, MariaDB, NPM, Portainer) last

find ./infrastructure -name "docker-compose.yml" -execdir docker compose down -t 20 ;

3. Bring up databases and management tools (Postgres, MariaDB, NPM, Portainer) first

find ./infrastructure -name "docker-compose.yml" -execdir docker compose up -d -t 20 ;

4. Bring up all user applications last

find ./apps -name "docker-compose.yml" -execdir docker compose up -d -t 20 ;

Option 2: The "Red Button" Bash Script (Recommended)

To handle database grace periods, log progress, and give you both a **Down (Red Button)** and an **Up (Green Button)** workflow, create a single master script on your host.

Step 1: Create the script file

Bash

```
nano ~/red-button.sh
```

Step 2: Paste the following script

(Be sure to update the `APP_DIRS` *array to match your actual directory paths on the VM).*

Bash

```
#!/bin/bash
# ==============================================================================
# "RED BUTTON" DOCKER COMPOSE CONTROLLER
# ==============================================================================

# List all 12 application directories here (absolute paths)
APP_DIRS=(
  "/home/ubuntu/dockerlab/shared-postgres"
  "/home/ubuntu/dockerlab/shared-mariadb"
  "/home/ubuntu/dockerlab/nginx-lite"
  "/home/ubuntu/dockerlab/wordpress"
  "/home/ubuntu/dockerlab/openproject"
  "/home/ubuntu/dockerlab/openemr"
  "/home/ubuntu/dockerlab/inventree"
  "/home/ubuntu/dockerlab/zammad"
  "/home/ubuntu/dockerlab/docmost"
  # Add your remaining application paths here...
)
```

```
ACTION=$1

if [ "$ACTION" == "down" ]; then
  echo -e "\031[1;31m[!] EXECUTING RED BUTTON: BRINGING DOWN ALL CONTAINERS...\033[0m"

  for DIR in "${APP_DIRS[@]}"; do
    if [ -d "$DIR" ]; then
      echo "--> Stopping: $DIR"
      # Using -t 20 gives database engines 20 seconds to flush cleanly
      (cd "$DIR" && docker compose down -t 20)
    else
      echo "[X] Directory not found: $DIR"
    fi
  done

  echo -e "\033[1;32m[✓] All applications stopped successfully.\033[0m"

elif [ "$ACTION" == "up" ]; then
  echo -e "\033[1;32m[+] EXECUTING GREEN BUTTON: BRINGING UP ALL CONTAINERS...\033[0m"

  for DIR in "${APP_DIRS[@]}"; do
    if [ -d "$DIR" ]; then
      echo "--> Starting: $DIR"
      (cd "$DIR" && docker compose up -d)
    else
      echo "[X] Directory not found: $DIR"
    fi
  done

  echo -e "\033[1;32m[✓] All applications started.\033[0m"

else
  echo "Usage: $0 {up|down}"
  exit 1
fi
```

Step 3: Make it executable & add a System Alias

Make the script executable:

Bash

```
chmod +x ~/red-button.sh
```

To make it accessible from **any directory** as a short command, add aliases to your `~/.bashrc`:

Bash

```
echo "alias redbutton='~/red-button.sh down'" >> ~/.bashrc
echo "alias greenbutton='~/red-button.sh up'" >> ~/.bashrc
source ~/.bashrc
```

How to Use It

Now, whenever you need to perform host maintenance, resize your VM, or take snapshots:

- **To stop all 12 apps gracefully:**
  Bash

  ```
  redbutton
  ```

- **To spin all 12 back up after maintenance:**
  Bash

  ```
  greenbutton
  ```

> **Pro-Tip on DB Order:** In your `APP_DIRS` list inside the script, put your frontend apps (like WordPress, OpenProject, Docmost) **first** for the `down` action so they close their connections before the shared database containers (`shared-postgres`, `shared-mariadb`) run their shutdown steps..