Web server management

- Apache setup & Virtual servers
- SQL
- SSL & Certificate management
- PostgreSQL
- SQL Reference
- Mailcow Dockerized Setup Guide (Home Server + VPS Proxy + Tailscale)

Apache setup & Virtual servers

Initial setup

Install packages `sudo apt install apache2 mariadb-server libapache2-mod-php php-mysql -y`

Open ports at `cd /etc/apache2/ports.conf` and add the line `Listen NEW_PORT_NUMBER`

Creating virtual servers

Create a file named NAME.conf and add something like this

```
<VirtualHost *:80>
        ServerName pdb.conorbriggs.com.au
        DocumentRoot /mnt/drives/10tb/10tb/web_servers/pdb/

<Directory /mnt/drives/10tb/10tb/web_servers/pdb/>
        Options -Indexes -FollowSymLinks
        AllowOverride All
        Require all granted
</Directory>
ErrorLog /var/log/apache2/pdb-error.log
CustomLog /var/log/apache2/pdb-access.log combined

RewriteEngine on
</VirtualHost>
```

Enable the site & restart apache with `sudo a2ensite site.conf && systemctl restart apache2`

SSL Certificate management

Install certbot and run setup

```
sudo apt install certbot python3-certbot-apache
a2enmod ssl
sudo certbot --apache
```

List all certificates with `certbot certificates`
Delete a certificaite `certbot delete --cert-name CERT_NAME_HERE`

Managing the apache2 service

Start, stop & restart `sudo service apache2 start/restart/stop`

Enable a config `sudo a2ensite SITE_CONF_FILE.conf`
Disable a config `sudo a2dissite SITE_CONF_FILE.conf`

SQL

After install

ensure secure access `sudo mysql_secure_installation`

Logging in

`mysql -u USERNAME -p`

add `-h HOSTNAME IP` when logging in remotely

Creating a user

```
CREATE USER 'NEW_USERNAME'@'localhost' IDENTIFIED BY 'NEW_PASSWORD';
```

Creating a database

```
CREATE DATABASE DATABASE_NAME
```

Allow user to modify Database

```
GRANT ALL PRIVILEGES ON DATABASE_NAME.* TO 'USERNAME'@'localhost';
```

Update permissions

`FLUSH PRIVILEGES;`

Open Database to the world

1. Open file at `/etc/mysql/maridadb.conf.d/50-server.cnf`
2. Edit line `bind-address = 127.0.0.1` to -> `bind-address = 0.0.0.0`
3. Restart db service `systemctl restart mariadb`
4. Allow port through firewall with `ufw allow 3306`

Automated backup script

```
#!/bin/bash

# === Config ===
BACKUP_DIR="/var/backups/mysql"
DATE=$(date +"%Y-%m-%d_%H-%M")
DB_USER="backupuser"
DB_PASS="yourpassword"

# Create backup dir if not exist
mkdir -p "$BACKUP_DIR"

# Dump all databases
mysqldump -u "$DB_USER" -p"$DB_PASS" --all-databases --single-transaction > "$BACKUP_DIR/mariadb-$DATE.sql"

# Optional: Compress the backup
gzip "$BACKUP_DIR/mariadb-$DATE.sql"

# Delete backups older than 7 days
find "$BACKUP_DIR" -name "*.sql.gz" -type f -mtime +7 -delete
```

SSL & Certificate management

Install certbot and run setup

```
sudo apt install certbot python3-certbot-apache
a2enmod ssl
sudo certbot --apache
```

List all certificates

```
certbot certificates
```

Delete a certificaite

```
certbot delete --cert-name CERT_NAME_HERE
```

PostgreSQL

Install

Install postgresql and postgis for coordinates data

```
sudo apt install postgresql postgis -y
```

Login

```
sudo -u postgres psql
```

Change password to default account

```
ALTER USER postgres WITH PASSWORD 'new password';
```

Restrict remote access to localhost only

Open the config file

```
nano /etc/postgresql/15/main/postgresql.conf
```

uncomment this line

```
listen_addresses = 'localhost'
```

if you need remote access

```
listen_addresses = '0.0.0.0'
```

and Edit `/etc/postgresql/15/main/pg_hba.conf` and add a line like:

```
host    all             all             192.168.1.0/24          scram-sha-256
```

SQL Reference

Create a table that auto deletes on other table deletion

```
CREATE TABLE tradie_postcode_covered (
    tradie_id INT,
    postcode VARCHAR(10),
    PRIMARY KEY (tradie_id, postcode), -- primary key ensures no tradie can be in the same postcode twice
    FOREIGN KEY (tradie_id) REFERENCES tradies(id) ON DELETE CASCADE
);
```

Mailcow Dockerized Setup Guide (Home Server + VPS Proxy + Tailscale)

This guide installs **Mailcow** on your **home server** (behind CGNAT), with your **VPS** handling public-facing TLS/proxying via **Caddy** (web UI/webmail) and **HAProxy** (raw mail ports), connected over **Tailscale**. Sourced from the current official Mailcow docs (docs.mailcow.email) as of mid-2026.

---

0. Architecture overview

```
Internet
  │
  ▼
VPS (public IP)
  ├── Caddy   → HTTPS termination for mail.seabee.me (webmail/admin UI)
  │             reverse_proxy → home server Tailscale IP:8005 (HTTP; X-Forwarded-Proto https)
  ├── HAProxy → raw TCP forward for 25 / 465 / 587 / 993 / 4190
  │             → home server Tailscale IP (same ports), with PROXY protocol
  │               so the real client IP survives (Section 8)
  └── Postfix relay (send-only, Tailscale-facing)
                ← mailcow relays ALL outbound here; egresses from the VPS
                  public IP so it matches your PTR + SPF (Section 9)
                    │
                    ▼ (Tailscale tunnel)
              Home server (behind CGNAT)
                └── Mailcow (Docker): Postfix, Dovecot, SOGo, Rspamd, ClamAV, etc.
```

Two things this topology forces that a normal single-box mailcow doesn't:

- **Outbound must be relayed through the VPS** (Section 9). The home server is behind CGNAT, so mail sent directly would leave from the ISP's shared IP — wrong PTR, no SPF match, port 25 likely blocked. Relaying through the VPS makes the sending IP match

`mail.seabee.me`.
- **Inbound must carry the client's real IP via PROXY protocol** (Section 8), otherwise every connection looks like it came from the VPS and mailcow's fail2ban will ban your own relay.

Mailcow itself runs entirely on your **home server**. The VPS never stores mail — it just forwards TLS/TCP traffic through the Tailscale tunnel.

On certificates: Caddy on the VPS obtains a cert for the *web UI* (it has public 80/443, so HTTP-01 works there). But the raw mail ports (465/587/993) are TCP-passed-through, so their TLS terminates *inside* Dovecot/Postfix on the home server — those services need their own valid cert or clients hit "untrusted certificate". The home server is behind CGNAT and can't answer HTTP-01, so it can't run the normal ACME flow. This guide solves it by **syncing the cert Caddy already holds down to the home server** (Section 4a, "Option B") — cbcore pulls it over Tailscale, drops it into mailcow, and restarts the mail services. mailcow's own ACME client stays **off** (`SKIP_LETS_ENCRYPT=y`) so it doesn't fight the synced cert. A DNS-01 alternative is documented too, for anyone whose DNS provider has a usable API.

---

1. Prerequisites

Hardware (home server)

- 2+ CPU cores (4+ for production/multiple domains)
- 6 GB RAM minimum + 1 GB swap (8 GB recommended if running ClamAV/full-text search)
- 20 GB disk minimum, more for mailbox storage

If your home server is resource-constrained, you can disable ClamAV and full-text search in `mailcow.conf` (`SKIP_CLAMD=y`, `SKIP_FTS=y`) to run comfortably on ~2 GB RAM.

OS support

Mailcow supports KVM/ESXi/Hyper-V VMs and bare metal. **It explicitly does not support** Synology/QNAP NAS devices, OpenVZ, or LXC containers — only real Docker hosts.

Required packages

```
sudo apt update
sudo apt install -y git openssl curl gawk coreutils grep jq
```

(`jq` is a fairly recent addition to Mailcow's requirements — make sure it's installed.)

Docker (latest engine, not distro package)

```
curl -sSL https://get.docker.com/ | CHANNEL=stable sh
sudo systemctl enable --now docker
```

Docker Compose plugin (v2.0+)

```
sudo apt install docker-compose-plugin
```

Confirm:

```
docker compose version   # must be >= 2.0
```

Debian 13 (trixie) note

If you upgraded from Debian 12→13, a package called **exim** may get pulled in and bind port 25 on the host, blocking Mailcow's Postfix container from using it. Remove it first:

```
sudo apt remove --purge exim4 exim4-base exim4-config -y
```

---

2. DNS records to configure

Set these at your domain registrar (Namecheap → Advanced DNS), pointing at your **VPS's public IP** (since that's what actually receives inbound connections):

```
# Name            Type       Value
mail              IN A       203.57.114.42
autodiscover      IN CNAME   mail.seabee.me.
autoconfig        IN CNAME   mail.seabee.me.
@                 IN MX 10   mail.seabee.me.
vps               IN A       203.57.114.42
```

The `vps` A record matters here: the VPS relay identifies itself as `vps.seabee.me` when sending outbound (Section 9), and its PTR must match that name (below) — which only forward-confirms if

`vps.seabee.me` also resolves to the VPS IP.

In Namecheap specifically

- The **A record** and **CNAME records** go under **Advanced DNS → Host Records** (Add New Record).
- The **MX record** goes under **Advanced DNS → Mail Settings → Custom MX**, filled in as:

Field	Value
Type	MX Record
Host	`@`
Value	`mail.seabee.me.`
Priority	`10`
TTL	Automatic

- **Important:** the MX record only resolves correctly if the `mail` A record above already exists — MX points to a hostname, not an IP directly.
- Check Host Records for a stray CNAME on `@` (bare domain) — Namecheap gives CNAME priority over MX on the same host, which silently breaks mail delivery if one exists.

SPF, DKIM, DMARC

```
@               IN TXT     "v=spf1 mx a -all"
_dmarc          IN TXT     "v=DMARC1; p=none; rua=mailto:postmaster@seabee.me"
```

Both go under **Host Records** as TXT records. DKIM is generated **inside the Mailcow admin UI** after install (Configuration → ARC/DKIM keys) — you'll copy that TXT record in afterward, also as a Host Record.

On the SPF `mx a -all`: this only holds true because *outbound mail is relayed through the VPS* (Section 9), so mail leaves from the VPS IP, which is exactly what `mail.seabee.me`'s A record (and therefore `mx`) resolves to. If you ever let the home server send directly, it egresses from the CGNAT IP and this SPF fails hard — see Section 9.

Stage DMARC — do not launch at `p=reject`. Start at `p=none` as above. It changes nothing about delivery but makes receivers send you aggregate reports at the `rua` address. Watch those for a week or two and confirm your legitimate mail is passing DKIM *and* SPF with alignment. Only then tighten to `p=quarantine`, and later `p=reject`. Launching straight at `reject` means any alignment mistake silently bins your own mail at the recipient with no warning.

Reverse DNS (PTR)

Critical for deliverability: your **VPS provider**, not your DNS zone, controls this — PTR belongs to whoever owns the IP block. The PTR for `203.57.114.42` should resolve to **`vps.seabee.me`**, *not* `mail.seabee.me`.

Why `vps`, not `mail`: the box that actually connects to Gmail/etc. and sends your outbound mail is the **VPS relay** (Section 9), and it greets remote servers with HELO `vps.seabee.me` (it can't use `mail.seabee.me` — that name belongs to cbcore, and a shared name makes Postfix think it's talking to itself; see Section 9). Receivers run FCrDNS: they check that the connecting IP's PTR matches the HELO name and forward-confirms. So HELO (`vps.seabee.me`), PTR (`vps.seabee.me`), and the `vps` A record must all agree. A PTR of `mail.seabee.me` while the relay HELOs `vps.seabee.me` is a mismatch and a (small but real) spam signal. Note SPF is unaffected either way — it checks the sending *IP* against `v=spf1 mx a -all`, and the IP is authorised regardless of name.

On Binary Lane: mPanel → Network/IP settings → the "Reverse DNS" field next to your public IPv4 address. Set it to `vps.seabee.me`. Binary Lane uses a 12-hour TTL for PTR records, so allow up to half a day to propagate. Make sure the `vps` A record exists first.

Verify once propagated:

```
dig +short mail.seabee.me       # → your VPS IP, and ONLY that
dig +short -x 203.57.114.42     # → vps.seabee.me.   (matches the relay HELO)
dig +short vps.seabee.me        # → 203.57.114.42    (forward-confirms the PTR)
```

> **`mail` must resolve to exactly one IP — the VPS.** If `dig +short mail.seabee.me` returns two addresses (e.g. the VPS *and* your home connection's public IP), delete the stray record. A common cause is a leftover dynamic-DNS A record or a wildcard `*` A record pointing home. With two A records, clients pick one at random, so ~half of inbound mail and client connections hit your home IP (behind CGNAT, nothing listening) and time out. Since your MX points at `mail.seabee.me`, this silently corrupts delivery. Fix it before any mail-flow testing.

Changing the PTR only affects reverse lookups on that IP — any other hostnames already pointing at the same IP (e.g. a `vps.seabee.me` you use for SSH) keep working exactly as before, since PTR and A records are independent.

Optional but recommended: autoconfig SRV records

Full zone-file form:

```
_autodiscover._tcp  IN SRV  0 1 443  mail.seabee.me.
_imaps._tcp         IN SRV  0 1 993  mail.seabee.me.
_submission._tcp    IN SRV  0 1 587  mail.seabee.me.
_submissions._tcp   IN SRV  0 1 465  mail.seabee.me.
_sieve._tcp         IN SRV  0 1 4190 mail.seabee.me.
```

In Namecheap, SRV records use separate fields rather than one string. Add each one under Host Records → Add New Record → SRV Record:

Service	Protocol	Priority	Weight	Port	Target
`_autodiscover`	`_tcp`	0	1	443	`mail.seabee.me`
`_imaps`	`_tcp`	0	1	993	`mail.seabee.me`
`_submission`	`_tcp`	0	1	587	`mail.seabee.me`
`_submissions`	`_tcp`	0	1	465	`mail.seabee.me`
`_sieve`	`_tcp`	0	1	4190	`mail.seabee.me`

These aren't required for mail to function — the autodiscover/autoconfig CNAMEs above already cover most clients — but they add broader compatibility (notably Outlook via `_autodiscover`). Fine to add later once core mail delivery is confirmed working.

---

3. Install Mailcow on the home server

```
umask 0022
mkdir -p /home/conor/Docker
cd /home/conor/Docker
git clone https://github.com/mailcow/mailcow-dockerized mailcow
cd mailcow
./generate_config.sh
```

Note: Mailcow's own docs default to `/opt/mailcow-dockerized`, but any path works — Mailcow doesn't hardcode `/opt` anywhere; it just uses whatever directory you run `generate_config.sh` and `docker compose` from. The only requirement is that the user running these commands has read/write access to the directory and to the Docker socket (i.e. is in the `docker` group, or you run the commands with `sudo`).

You'll be prompted for:

- **Mailcow hostname** → enter `mail.seabee.me` (must match your DNS `A`/`MX` records)
- **Timezone**

This creates `mailcow.conf`.

---

4. Edit `mailcow.conf` — key settings for your proxy setup

```
nano mailcow.conf
```

Change/confirm these values:

```
MAILCOW_HOSTNAME=mail.seabee.me

# Turn mailcow's own ACME client OFF. We supply the cert externally by syncing
# Caddy's cert into mailcow (Section 4a, Option B), so mailcow must NOT try to
# obtain/renew its own — otherwise it overwrites the synced files.
# (If you instead choose the DNS-01 alternative in 4a, set these the other way:
#  SKIP_LETS_ENCRYPT=n / ACME_DNS_CHALLENGE=y.)
SKIP_LETS_ENCRYPT=y
ACME_DNS_CHALLENGE=n

# Skip the public-IP-matches-DNS check, since this host is behind CGNAT
SKIP_IP_CHECK=y

# Disable IPv6 — nothing in the VPS→Tailscale→home path uses it, and half-enabling
# it (detected but not configured in Docker's daemon.json) just invites confusion
ENABLE_IPV6=false

# Web UI / webmail (nginx). Caddy reverse-proxies to the HTTP port (8005) —
# see Section 8. Two settings make that work behind Caddy without a redirect loop:
#   HTTP_REDIRECT=n          -> mailcow must NOT do its own HTTP->HTTPS redirect.
#                               Caddy already guarantees HTTPS to the outside world;
#                               with =y, mailcow 301-redirects every proxied request
#                               and you get an infinite loop (ERR_TOO_MANY_REDIRECTS).
#   ADDITIONAL_SERVER_NAMES   -> the hostname your reverse proxy uses, so mailcow's
#                               nginx serves the right vhost instead of falling back
```

```
#                           to its internal address (which breaks SOGo webmail).
HTTP_BIND=100.64.0.3
HTTP_PORT=8005
HTTPS_BIND=100.64.0.3
HTTPS_PORT=8443
HTTP_REDIRECT=n
ADDITIONAL_SERVER_NAMES=mail.seabee.me

# Mail services (Postfix/Dovecot). Each is bound with a single IP:PORT variable.
# This is the native mailcow mechanism for scoping mail ports to one interface —
# do NOT use a docker-compose.override.yml for this (see Section 5 for why).
SMTP_PORT=100.64.0.3:25
SMTPS_PORT=100.64.0.3:465
SUBMISSION_PORT=100.64.0.3:587
IMAP_PORT=100.64.0.3:143
IMAPS_PORT=100.64.0.3:993
SIEVE_PORT=100.64.0.3:4190

# Leave the local-only admin/DB ports on 127.0.0.1 — do not scope these to Tailscale
# DOVEADM_PORT=127.0.0.1:19991
# SQL_PORT=127.0.0.1:13306
# REDIS_PORT=127.0.0.1:7654

# Adjust if these clash with anything else on your Docker host
IPV4_NETWORK=172.22.1
```

Replace `100.64.0.3` with your home server's actual Tailscale IP (`tailscale ip -4`).

Why `HTTP_PORT=8005` rather than the Mailcow default of `8080`: purely to avoid clashing with anything else already using 8080 on this host — pick whatever's free. Caddy reverse-proxies to **this** port (8005) over plain HTTP inside the Tailscale tunnel; `HTTPS_PORT=8443` ends up unused externally. This is the reverse-proxy layout mailcow documents (proxy to the HTTP port, pass `X-Forwarded-Proto https`), and it's what makes SOGo webmail generate correct `mail.seabee.me` URLs — see Section 8.

4a. Certificates for the mail ports

Caddy on the VPS handles TLS for the web UI. But HAProxy passes the mail ports (465/587/993) through as raw TCP, so their TLS is terminated by Dovecot/Postfix on the home server — and with a

self-signed cert, every mail client shows a warning and remote MTAs distrust your STARTTLS. mailcow's cert is used for mail transport, not just HTTPS, so the home server genuinely needs a trusted one.

The home server can't answer an HTTP-01 challenge (no public 80/443 — the CGNAT problem), so it can't run the normal ACME flow itself. Two ways around that:

- **Option B — sync Caddy's cert into mailcow (this guide's default).** Caddy on the VPS already holds a valid Let's Encrypt cert for `mail.seabee.me`. cbcore pulls that cert over Tailscale, drops it into mailcow, and restarts the mail services. No DNS-provider API needed, nothing changes at your registrar. Requires `SKIP_LETS_ENCRYPT=y` (Section 4).
- **Option A — DNS-01** (documented at the end of this section): mailcow gets its own cert by writing a DNS TXT record. Cleaner and fully self-contained, but only if your DNS provider has a usable API — Namecheap's is gated and IPv4-whitelist-only, which doesn't work from a CGNAT box, so Option B is the default here.

Option B — pull Caddy's cert onto cbcore

The design is **pull, not push**: cbcore reaches *out* to the VPS to fetch one cert, rather than the internet-facing VPS holding a key that can write files and restart services on your home server. The VPS key is locked to a *forced command* that can only emit that one cert — a stolen key can't get a shell.

Step 1 — on the VPS, create the emitter script (streams just the cert + key as a tar on stdout). The `'EOF'` is single-quoted so the shell writes the variables literally instead of expanding them now:

```
sudo tee /usr/local/bin/emit-mail-cert >/dev/null <<'EOF'
#!/usr/bin/env bash
set -euo pipefail
DOMAIN="mail.seabee.me"
CADDY_DATA="/var/lib/caddy/.local/share/caddy/certificates"
crt="$(find "$CADDY_DATA" -type f -name "${DOMAIN}.crt" | head -n1)"
key="$(find "$CADDY_DATA" -type f -name "${DOMAIN}.key" | head -n1)"
[[ -r "$crt" && -r "$key" ]] || { echo "cert not found" >&2; exit 1; }
tar -C "$(dirname "$crt")" -cf - "$(basename "$crt")" "$(basename "$key")"
EOF
sudo chmod 0755 /usr/local/bin/emit-mail-cert
```

Step 2 — on cbcore, create the pull key and print the public half:

```
ssh-keygen -t ed25519 -f /home/conor/.ssh/vps_cert_pull -N ''
cat /home/conor/.ssh/vps_cert_pull.pub
```

Step 3 — on the VPS, authorise that key but lock it to the emitter. Caddy's key is `0600` and root-owned, so the emitter must run as root; put the line in **root's** `authorized_keys`. The `command=` / `no-*` restrictions mean this key can *only* emit the cert:

```
# on the VPS — paste the cbcore public key where shown
sudo mkdir -p /root/.ssh && sudo chmod 700 /root/.ssh
echo 'command="/usr/local/bin/emit-mail-cert",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-
pty ssh-ed25519 AAAA...PASTE_CBCORE_cert_pull.pub...' \
  | sudo tee -a /root/.ssh/authorized_keys >/dev/null
sudo chmod 600 /root/.ssh/authorized_keys
```

Verify the lock from cbcore — connect as **root** (that's whose `authorized_keys` holds the line), which should list exactly two files and give no shell:

```
ssh -i /home/conor/.ssh/vps_cert_pull root@100.64.0.2 | tar -tf -
# → mail.seabee.me.crt  and  mail.seabee.me.key  only
# A "PTY allocation request failed on channel 0" line is EXPECTED and good —
# it's the no-pty restriction refusing an interactive shell.
```

This requires the VPS to allow key-based root login. Check with `sudo sshd -T | grep -i permitrootlogin`; if it's `no`, set `PermitRootLogin prohibit-password` (allows key, blocks password root login) and `sudo sshd -t && sudo systemctl reload ssh`. Don't use plain `yes` — that permits password root login too, which is a brute-force magnet on a public IP.

Step 4 — on cbcore, install the sync script. It pulls, checks the cert fingerprint against what's already deployed, and only installs + restarts when it actually changed (so it's safe to run daily and is a no-op the ~59 days out of 60 that nothing changed):

```
sudo tee /usr/local/bin/pull-mailcow-cert.sh >/dev/null <<'EOF'
#!/usr/bin/env bash
# Pulls the Caddy-issued cert for mail.seabee.me FROM the VPS, installs it into
# mailcow, and restarts mail services ONLY when the cert changed. Idempotent.
set -euo pipefail

VPS_USER="root"                          # whose authorized_keys holds the locked key
VPS_HOST="100.64.0.2"                        # VPS, Tailscale IP
SSH_KEY="/home/conor/.ssh/vps_cert_pull"
MAILCOW="/home/conor/Docker/mailcow"
SSL="${MAILCOW}/data/assets/ssl"
DOMAIN="mail.seabee.me"
SSH_OPTS=(-i "$SSH_KEY" -o BatchMode=yes -o ConnectTimeout=15 \
          -o StrictHostKeyChecking=accept-new)
```

```
log()  { printf '%s  %s\n' "$(date -Is)" "$*"; }
fail() { log "ERROR: $*"; exit 1; }

tmp="$(mktemp -d)"; trap 'rm -rf "$tmp"' EXIT

ssh "${SSH_OPTS[@]}" "${VPS_USER}@${VPS_HOST}" | tar -C "$tmp" -xf - \
    || fail "pull from VPS failed (ssh/forced-command/tar)"

crt="$(find "$tmp" -type f -name "${DOMAIN}.crt" | head -n1 || true)"
key="$(find "$tmp" -type f -name "${DOMAIN}.key" | head -n1 || true)"
[[ -r "${crt:-}" && -r "${key:-}" ]] || fail "expected cert/key not in pulled tar"

openssl x509 -in "$crt" -noout -checkend 0 >/dev/null \
    || fail "pulled cert has already expired — refusing to deploy"

new_fp="$(openssl x509 -in "$crt" -noout -fingerprint -sha256 | cut -d= -f2)"
cur_fp="$(openssl x509 -in "${SSL}/cert.pem" -noout -fingerprint -sha256 2>/dev/null \
          | cut -d= -f2 || true)"
if [[ -n "$cur_fp" && "$new_fp" == "$cur_fp" ]]; then
    log "mailcow already current (${new_fp}); nothing to do."
    exit 0
fi
log "new cert (${new_fp}, was ${cur_fp:-none}) — installing and restarting."

install -m 0644 "$crt" "${SSL}/cert.pem"
install -m 0600 "$key" "${SSL}/key.pem"
cd "$MAILCOW"
docker compose restart postfix-mailcow dovecot-mailcow nginx-mailcow \
    || fail "cert installed but container restart failed — check docker"
log "cert deployed and mail services restarted."
EOF
sudo chmod 0755 /usr/local/bin/pull-mailcow-cert.sh
```

Step 5 — bootstrap once (replaces the snake-oil cert immediately) and confirm cbcore's `conor` can drive Docker:

```
docker compose version                  # must work without sudo (conor in docker group)
/usr/local/bin/pull-mailcow-cert.sh
```

Step 6 — schedule it daily on cbcore via a systemd timer. Daily rather than monthly on purpose: Caddy renews ~30 days before expiry and that won't align to a monthly cron, so a fixed monthly copy can leave cbcore's cert lapsing before the next run. Daily + the fingerprint guard eliminates that with no extra load:

```
sudo tee /etc/systemd/system/mailcow-cert-pull.service >/dev/null <<'EOF'
[Unit]
Description=Pull Caddy cert from VPS into mailcow
Wants=network-online.target
After=network-online.target

[Service]
Type=oneshot
User=conor
ExecStart=/usr/local/bin/pull-mailcow-cert.sh
EOF

sudo tee /etc/systemd/system/mailcow-cert-pull.timer >/dev/null <<'EOF'
[Unit]
Description=Daily mailcow cert pull

[Timer]
OnCalendar=daily
Persistent=true
RandomizedDelaySec=1h

[Install]
WantedBy=timers.target
EOF

sudo systemctl daemon-reload
sudo systemctl enable --now mailcow-cert-pull.timer
```

Step 7 — expiry watchdog (the one thing people skip and regret). The sync breaks *silently* if Caddy's storage path changes on an update or the key rotates — you'd only find out when the mail cert lapses ~60 days later. This checks daily and, on a warning, appends a timestamped line to `/home/conor/warning.txt`:

```
sudo tee /usr/local/bin/mailcow-cert-expiry-check.sh >/dev/null <<'EOF'
#!/usr/bin/env bash
# Warns if the mailcow mail cert is within 10 days of expiry (or missing).
```

```
# On warning, appends a timestamped line to /home/conor/warning.txt.
set -euo pipefail

CERT="/home/conor/Docker/mailcow/data/assets/ssl/cert.pem"
WARN_FILE="/home/conor/warning.txt"
DAYS=10

if ! openssl x509 -in "$CERT" -noout -checkend $((DAYS*86400)) >/dev/null 2>&1; then
    enddate="$(openssl x509 -in "$CERT" -noout -enddate 2>/dev/null | cut -d= -f2 || true)"
    printf '%s  WARNING: mailcow mail cert expires within %d days or is missing (%s) notAfter=%s\n' \
        "$(date -Is)" "$DAYS" "$CERT" "${enddate:-unknown}" >> "$WARN_FILE"
    exit 1
fi
exit 0
EOF
sudo chmod 0755 /usr/local/bin/mailcow-cert-expiry-check.sh

# schedule daily from CONOR's crontab (not root) so /home/conor/warning.txt is conor-owned
( crontab -l 2>/dev/null; \
  echo '30 6 * * * /usr/local/bin/mailcow-cert-expiry-check.sh' ) | crontab -
```

Note the cron is added with plain `crontab` (as conor), not `sudo crontab` — that way the warning file it creates is owned by you, not root. On a healthy day the script writes nothing and exits 0; it only touches `warning.txt` when the cert is genuinely close to expiry or missing. Force a test line without waiting for a real expiry by asking whether the cert is valid 89 days out (it won't be):

```
CERT=/home/conor/Docker/mailcow/data/assets/ssl/cert.pem
openssl x509 -in "$CERT" -noout -checkend $((89*86400)) \
  || echo "$(date -Is)  TEST warning line" >> /home/conor/warning.txt
cat /home/conor/warning.txt
```

> **Alerting:** a line in a file only helps if something reads it. Since you run Telegram (the "James" agent), point a notifier at `warning.txt` — or swap the `printf` for a `curl` to the Telegram Bot API — so the alert reaches you rather than sitting in a file you never open.

Verify the whole thing in Section 12 (the `openssl s_client ... -issuer` check should now show Let's Encrypt, not the mailcow snake-oil CA).

Option A — DNS-01 (alternative, only if your DNS API is usable)

If you can use DNS-01, it's cleaner: mailcow gets and renews its own cert, no cross-host sync. Set `SKIP_LETS_ENCRYPT=n` and `ACME_DNS_CHALLENGE=y` in Section 4, then pass your DNS provider's API token to the `acme-mailcow` container. mailcow runs `acme.sh`, so `ACME_DNS_PROVIDER` takes an acme.sh dnsapi code. Cloudflare example:

```
cd /home/conor/Docker/mailcow
sudo tee docker-compose.override.yml >/dev/null <<'EOF'
services:
  acme-mailcow:
    environment:
      - CF_Token=<cloudflare-scoped-api-token>
      - CF_Zone_ID=<zone-id-for-seabee.me>
EOF
docker compose up -d acme-mailcow
docker compose logs -f acme-mailcow          # watch for a successful issue
```

> **Namecheap can't do this from CGNAT.** Namecheap's API is granted only above an account threshold *and* requires whitelisting the calling IPv4 — your acme requests would leave via the ISP's shared, rotating CGNAT address, which you can't reliably whitelist. Moving DNS *hosting* (not the registrar) to Cloudflare removes both problems (free `dns_cf`, token auth, no whitelist); recreate the Section 2 records there and repoint nameservers. If you'd rather not touch DNS, stay on Option B above.

---

5. Why the mail ports live in `mailcow.conf`, not an override file

By default, Mailcow binds mail ports to all interfaces (`0.0.0.0`). Since your home server sits behind CGNAT, nothing on the raw internet can reach these ports regardless — but leaving them on `0.0.0.0` still means every device on your **home LAN** and every peer on your **tailnet** can connect to them directly, bypassing HAProxy entirely. Scoping to the Tailscale IP limits that down to just the

VPS.

Mailcow binds every non-http service through a single `IP:PORT` variable in `mailcow.conf`. You already set these in Section 4 — the format is documented at docs.mailcow.email → *IP bindings* (`SMTP_PORT=1.2.3.4:25` binds SMTP to `1.2.3.4:25`):

```
SMTP_PORT=100.64.0.3:25
SMTPS_PORT=100.64.0.3:465
SUBMISSION_PORT=100.64.0.3:587
IMAP_PORT=100.64.0.3:143
IMAPS_PORT=100.64.0.3:993
SIEVE_PORT=100.64.0.3:4190
```

Do not use a `docker-compose.override.yml` for this (an earlier draft of this guide did — it's wrong). The base `docker-compose.yml` already renders each of these ports from the variable above (e.g. `${SMTP_PORT:-25}:25`). An override file that *also* defines a `ports:` list does **not** replace the base list — Compose merges the two, and for the multi-value `ports` option it concatenates/merges by uniqueness key rather than overriding. Depending on your Compose version you either end up with **both** `0.0.0.0:25` and `100.64.0.3:25` bound (which fails at launch with `port is already allocated`, since `0.0.0.0:25` already covers the Tailscale IP) or a silent, version-dependent merge. Setting the `mailcow.conf` variable is the officially documented method and has no merge ambiguity — the base compose file is left untouched, so `./update.sh` never fights your config.

A few points worth knowing:

- **HTTP is the exception.** The web UI is bound with the separate `HTTP_BIND`/`HTTPS_BIND` + `HTTP_PORT`/`HTTPS_PORT` pairs (Section 4), not the single `IP:PORT` form — a mailcow quirk, for technical reasons http is handled differently from the mail services.
- **This disables IPv6 for those services.** Per the mailcow docs, specifying an explicit IPv4 address skips all IPv6 bindings for that service since Docker 20.x. Fine here — `ENABLE_IPV6=false` already.
- **Local-only ports stay put.** `DOVEADM_PORT`, `SQL_PORT` and `REDIS_PORT` remain on their `127.0.0.1` defaults. Don't scope these to Tailscale.
- **LAN clients:** the `IP:PORT` variable takes a single address. If you also want phones/laptops on your home WiFi to reach the mail ports directly (not via the VPS), that's the one case where you'd add a `docker-compose.override.yml` — a *second*, LAN-IP binding per port. Do it deliberately, and remember the merge behaviour above: those override bindings are *added* to the `mailcow.conf` ones, so leave the `IP:PORT` variable on the Tailscale IP and let the override supply only the extra LAN binding.

Apply the change with a full down/up, since you're changing the rendered port mappings (a bare `docker compose up -d` won't always re-map ports cleanly):

```
docker compose down
docker compose up -d
```

```
docker compose ps
```

Verify the bindings landed on the right interface:

```
sudo ss -tlpn | grep -E ':(25|143|465|587|993|4190|8005|8443)\b'
```

You should see `100.64.0.3` (not `0.0.0.0`) next to each mail/web port. `19991`, `13306` and `7654` should still show `127.0.0.1`.

---

6. Home server firewall — and why `ufw` isn't the right tool here

The obvious instinct is a `ufw` rule allowing only the VPS:

```
# This does NOT actually filter the mailcow ports — see below
sudo ufw allow from <VPS_TAILSCALE_IP> to any port 25,465,587,993,4190,8005,8443 proto tcp
```

This gives false confidence. Docker publishes container ports by inserting its own rules into the `DOCKER` iptables chain, which is evaluated *before* ufw's `INPUT` chain — so ufw `allow`/`deny` rules do not govern published container ports at all. A `ufw deny` here wouldn't block anything.

What's *actually* limiting exposure is the work you already did in Section 4/5: binding every mail port to the Tailscale IP (`100.64.0.3`) means they only exist on the tailnet interface, unreachable from the LAN or the internet. The correct place to then restrict *which tailnet peers* may reach them is a **Tailscale ACL**, not a host firewall. In your tailnet policy file, scope the mail ports to the VPS only — for example:

```
// Tailscale ACL — only the VPS may reach the mail/UI ports on the home server
{
  "acls": [
    {
      "action": "accept",
      "src":    ["<vps-node>"],
      "dst":    ["<home-node>:25,465,587,993,4190,8005,8443"]
    }
  ]
}
```

(Replace `<vps-node>` / `<home-node>` with the device names or tags from your tailnet.) This is enforced by Tailscale regardless of Docker's iptables behaviour.

If you *do* want host-level enforcement as defence in depth, add rules to the `DOCKER-USER` chain (which Docker leaves for exactly this) rather than ufw:

```
# Debian 13, nftables-backed iptables — verify against `iptables -L DOCKER-USER`
sudo iptables -I DOCKER-USER -i tailscale0 -s <VPS_TAILSCALE_IP> -j RETURN
sudo iptables -I DOCKER-USER -i tailscale0 -j DROP
```

Persist these (e.g. `netfilter-persistent save`) — `DOCKER-USER` rules are not saved across reboots by default.

---

7. Start Mailcow

```
docker compose pull
docker compose up -d
docker compose ps
```

You should see 15+ containers reporting `Up`.

---

8. VPS side — Caddy and HAProxy (inbound)

Caddy (web UI / webmail / autodiscover — HTTPS termination)

Add a site block to your existing Caddyfile. Proxy to mailcow's **HTTP** port (8005) — this is mailcow's documented reverse-proxy layout, and it's what makes SOGo webmail work. Getting this right depends on the three things set together, here and in Section 4:

- **`HTTP_REDIRECT=n`** in `mailcow.conf` (Section 4) — otherwise mailcow's nginx 301-redirects every proxied request to HTTPS, producing an infinite loop (`ERR_TOO_MANY_REDIRECTS`). It ignores `X-Forwarded-Proto` for this, so the header alone won't save you; the redirect has to

be disabled.

- **`ADDITIONAL_SERVER_NAMES=mail.seabee.me`** (Section 4) — so mailcow serves the right vhost instead of falling back to its internal address (which makes SOGo emit `https://100.64.0.3:8443/...` URLs that hang for anyone off your tailnet).
- **`header_up X-Forwarded-Proto https`** below — tells mailcow the public side is HTTPS even though the Caddy→cbcore hop is plain HTTP.

```
mail.seabee.me {
    reverse_proxy 100.64.0.3:8005 {
        header_up Host {host}
        header_up X-Forwarded-Proto https
    }
}

autodiscover.seabee.me, autoconfig.seabee.me {
    reverse_proxy 100.64.0.3:8005 {
        header_up Host {host}
        header_up X-Forwarded-Proto https
    }
}
```

Plain HTTP to 8005 (no `https://`, no `tls_insecure_skip_verify`) — the hop rides inside the encrypted Tailscale tunnel, so there's nothing to gain from TLS on that leg, and it sidesteps the snake-oil-cert-on-8443 problem entirely. The browser only ever talks to Caddy, which serves its own valid Let's Encrypt cert for `mail.seabee.me`, and Caddy handles ACME issuance/renewal automatically.

> **Do not proxy to the HTTPS port (8443).** It seems tempting (it dodges the redirect loop without `HTTP_REDIRECT=n`), but mailcow's nginx on 8443 emits its own internal-address redirects that make SOGo build `100.64.0.3:8443` resource URLs — fine on your tailnet (the IP resolves), but they hang forever for real external users. The 8005 + `HTTP_REDIRECT=n` combination above is the correct one.

> **Verify no loop after reloading**, from a machine (or `curl`) — `curl -skI https://mail.seabee.me/` should return `HTTP/2 200`, not `301 → https://mail.seabee.me/`. And test webmail login from **off** your tailnet (phone on mobile data), since a tailnet machine can reach `100.64.0.3:8443` directly and will mask a lingering internal-URL problem.

HAProxy (raw mail ports — full `/etc/haproxy/haproxy.cfg`)

This writes the **whole** file, so it includes `global` and `defaults` (a HAProxy config can't run without them) plus all six mail frontends/backends. Design points baked in:

1. **Binds scoped to the VPS *public* IP, not `*`.** `bind *:25` occupies port 25 on *every* interface including Tailscale — which collides with the outbound relay Postfix in Section 9 (it needs `100.64.0.2:25`). Scoping to the public IP keeps inbound and outbound cleanly separated.
2. **Plain passthrough to start — no `send-proxy` yet.** `check send-proxy` makes HAProxy speak PROXY protocol on the *health checks* too, so until the home side is configured to accept it (Section 8 "Home side"), the backend rejects the header, the check fails, the server shows **DOWN**, and no mail flows. So bring it up plain, confirm mail works, then switch **both ends together** (command at the end of this section).
3. **`mode tcp` + long timeouts in `defaults`.** This HAProxy only forwards mail, so `defaults` is TCP with 1-hour client/server timeouts — SMTP is fine either way, but IMAP IDLE holds a connection open ~29 minutes and the stock 50-second timeout would cut it. (The HTTP `errorfile`s from the stock Debian config are dropped; they only apply in `mode http`.)

Back up the current file first, then write it. Replace `203.57.114.42` with your VPS public IP if different (`ip -4 addr` to confirm it's actually on the interface, or the bind fails):

```
sudo cp -a /etc/haproxy/haproxy.cfg /etc/haproxy/haproxy.cfg.bak.$(date +%F)

sudo tee /etc/haproxy/haproxy.cfg >/dev/null <<'EOF'
global
    log /dev/log    local0
    log /dev/log    local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
    daemon
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private
    ssl-default-bind-ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384
    ssl-default-bind-ciphersuites
```

```
TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256
    ssl-default-bind-options ssl-min-ver TLSv1.2 no-tls-tickets

defaults
    log     global
    mode    tcp
    option  tcplog
    option  dontlognull
    timeout connect 10s
    timeout client  1h
    timeout server  1h

# ============ Inbound mail -> cbcore (100.64.0.3) over Tailscale ============
# Binds scoped to the VPS public IP so mail ports stay free on the Tailscale
# interface for the outbound relay (Section 9). Plain passthrough for now;
# add " send-proxy-v2" to the server lines once the home side is ready.

frontend smtp_in
    bind 203.57.114.42:25
    default_backend smtp_home
backend smtp_home
    server home 100.64.0.3:25 check

frontend submission_in
    bind 203.57.114.42:587
    default_backend submission_home
backend submission_home
    server home 100.64.0.3:587 check

frontend smtps_in
    bind 203.57.114.42:465
    default_backend smtps_home
backend smtps_home
    server home 100.64.0.3:465 check

frontend imaps_in
    bind 203.57.114.42:993
    default_backend imaps_home
backend imaps_home
    server home 100.64.0.3:993 check
```

```
frontend sieve_in
    bind 203.57.114.42:4190
    default_backend sieve_home
backend sieve_home
    server home 100.64.0.3:4190 check
EOF

sudo haproxy -c -f /etc/haproxy/haproxy.cfg     # validate BEFORE reloading
sudo systemctl reload haproxy
```

Confirm every backend is **UP** (this is where a stray `send-proxy` would show DOWN):

```
echo 'show servers state' | sudo socat stdio /run/haproxy/admin.sock | awk '{print $4, $5, $6}'
# each home server should be state 2 (UP); or just: sudo systemctl status haproxy
```

Then, and only after the home side accepts PROXY protocol (next subsection), enable it — this appends `send-proxy-v2` to every backend in one shot (run once):

```
sudo sed -i 's/\(server home [0-9.]*:[0-9]*\) check$/\1 check send-proxy-v2/' /etc/haproxy/haproxy.cfg
sudo haproxy -c -f /etc/haproxy/haproxy.cfg && sudo systemctl reload haproxy
```

Why bother with `send-proxy-v2` at all: without it the home-side Postfix/Dovecot see every connection as coming from the VPS, so mailcow's fail2ban bans the VPS on the first brute-force burst (killing *all* inbound mail) and Rspamd loses its RBL/reputation signal.

Home side — teach Postfix/Dovecot to accept PROXY protocol

mailcow persists these overrides in `data/conf/`, so they survive `./update.sh`. Set the trusted source to the **VPS Tailscale IP** (`100.64.0.2` in this guide) — the only thing that will ever send a PROXY header. Note the `data/` tree is **root-owned** (mailcow's containers create it as root), so these use `sudo tee` — a plain `tee`/`>>` gives `Permission denied`, and prefixing the whole pipe with `sudo` wouldn't help because that only elevates the left side of the pipe, not the `tee` doing the write. Run from your mailcow dir on cbcore:

```
cd /home/conor/Docker/mailcow

# Postfix: postscreen handles port 25 (append only if not already present)
grep -q postscreen_upstream_proxy_protocol data/conf/postfix/extra.cf 2>/dev/null || \
```

```
  echo 'postscreen_upstream_proxy_protocol = haproxy' | sudo tee -a data/conf/postfix/extra.cf

# Dovecot: trust the VPS for PROXY headers on imaps/sieve
sudo tee -a data/conf/dovecot/extra.conf >/dev/null <<'EOF'
haproxy_trusted_networks = 100.64.0.2
haproxy_timeout = 30s
EOF
```

Confirm both writes actually landed (each should echo the line back):

```
grep postscreen_upstream_proxy_protocol data/conf/postfix/extra.cf
tail -n2 data/conf/dovecot/extra.conf
```

The submission/smtps ports (587/465) and the Dovecot listeners also each need a per-service flag — `-o smtpd_upstream_proxy_protocol=haproxy` on those Postfix `master.cf` services, and `haproxy = yes` on the Dovecot imaps/sieve `inet_listener` blocks.

> **Verify before editing master.cf / Dovecot listeners:** the exact form of those per-service edits differs between mailcow releases, and `master.cf`/Dovecot listener blocks aren't as cleanly overridable as the `extra.*` files — check the current method against docs.mailcow.email and your running `data/conf/postfix/master.cf` rather than pasting blind. Then restart: `docker compose restart postfix-mailcow dovecot-mailcow`.

> **Order of operations — do the whole home side, THEN flip HAProxy.** This `extra.cf` line only covers port 25 (postscreen). If you enable `send-proxy-v2` on HAProxy while the submission/smtps ports and Dovecot listeners still *don't* expect PROXY protocol, those ports break. So finish all the home-side edits above (including the per-service `master.cf`/listener flags), restart the containers, and only then run the `send-proxy-v2` `sed` from the previous subsection — both ends in one change. Until then, leave HAProxy on plain passthrough (the state you verified as all-backends-UP).

VPS firewall

```
sudo ufw allow 25,465,587,993,4190/tcp
```

(80/443 should already be open for Caddy. Note the ufw caveat from Section 6 applies only to the *home* server where Docker owns the chains — on the VPS, HAProxy/Postfix are host processes, so

ufw governs them normally.)

---

9. Outbound mail — relay through the VPS

Why this is mandatory here. The home server is behind CGNAT, so if mailcow sends directly, mail leaves via your ISP's shared IP: port 25 outbound is usually blocked, the PTR isn't yours, and it fails SPF (your SPF authorises the VPS IP via `mx`, not the CGNAT IP). Routing outbound through the VPS makes the sending IP the VPS's public IP — which has the matching PTR (Section 2) and is covered by SPF. DKIM is unaffected: mailcow signs on the home server *before* handing the message off, so the relay just forwards an already-signed message.

9a. Send-only Postfix relay on the VPS

Run a minimal Postfix on the VPS that accepts mail **only** from the home server (over Tailscale) and sends it to the internet. It listens on the Tailscale IP so it doesn't collide with HAProxy (which now owns the public IP, Section 8).

> **Environment:** the config below is for **Postfix 3.x on Debian 12/13** (matching your home server; adjust if your VPS runs something else). Verify directives against `postconf` and the Debian `postfix` man pages before applying — Postfix defaults do shift between majors.

Install:

```
sudo apt update && sudo apt install -y postfix
# choose "Internet Site" / system mail name mail.seabee.me when prompted
```

Set the relay-relevant parameters with `postconf -e` (idempotent and scriptable — it edits `main.cf` in place, so you can re-run it safely). Replace `<VPS_TAILSCALE_IP>` with the VPS's Tailscale IP (`100.64.0.2` here); `100.64.0.3` is cbcore, the only permitted sender.

> **The relay's hostname MUST differ from cbcore's.** cbcore is `mail.seabee.me`. If the VPS relay also calls itself `mail.seabee.me`, then when cbcore hands it a message and the relay greets cbcore with `HELO mail.seabee.me`, cbcore's Postfix

> sees its *own* hostname coming back and aborts with `mail for [100.64.0.2]:25 loops back to myself` — mail never leaves. So the relay uses **`vps.seabee.me`** for both `myhostname` and `smtp_helo_name`. (This is also why the PTR is `vps.seabee.me` in Section 2 — HELO and PTR must match.)

```
sudo postconf -e \
  'myhostname = vps.seabee.me' \
  'smtp_helo_name = vps.seabee.me' \
  'mydestination =' \
  'relay_domains =' \
  'relayhost =' \
  'inet_interfaces = 100.64.0.2, 127.0.0.1' \
  'inet_protocols = ipv4' \
  'mynetworks = 127.0.0.0/8, 100.64.0.3/32' \
  'smtpd_relay_restrictions = permit_mynetworks, reject_unauth_destination' \
  'smtpd_recipient_restrictions = permit_mynetworks, reject_unauth_destination' \
  'smtpd_helo_required = yes' \
  'disable_vrfy_command = yes' \
  'smtp_tls_security_level = may' \
  'smtp_tls_loglevel = 1'
```

What these do: `myhostname`/`smtp_helo_name = vps.seabee.me` gives the relay a distinct identity (see the loop note above). `mydestination=`/`relay_domains=` empty means it's a pure relay — accepts local mail for nothing and is a backup MX for nothing. `relayhost=` empty means the relay delivers **direct to the internet via MX lookup** (it's the end of the chain — cbcore relays to it, it relays to the world; if *it* had a relayhost pointing back, you'd loop). `inet_interfaces` binds it to the **Tailscale IP only** (not the public IP — that's HAProxy's, Section 8). `mynetworks` trusts *only* cbcore, and `reject_unauth_destination` in **both** restriction lists is what stops it being an open relay — only cbcore may relay to arbitrary destinations. Apply:

```
sudo postfix check && sudo systemctl restart postfix
# postconf lives in /usr/sbin — to READ values as non-root, use: sudo postconf myhostname relayhost
```

Confirm it's listening on the Tailscale IP only, and is **not** an open relay:

```
sudo ss -tlpn | grep ':25'          # expect <VPS_TAILSCALE_IP>:25, not 0.0.0.0:25
# open-relay test from a THIRD host on your tailnet (not the home server):
#   swaks --to test@gmail.com --server <VPS_TAILSCALE_IP> --from x@evil.example
# → must be REJECTED with "Relay access denied"
```

Also confirm your VPS provider actually permits **outbound** port 25 (some block it by default and unblock on request) — otherwise delivery silently fails regardless of config.

9b. Point mailcow at the relay

This is **two steps in two different screens** — creating the transport is not the same as using it.

Create the transport: mailcow admin UI → **Configuration → Routing → Sender-Dependent Transports** → add:

- **Host:** `[100.64.0.2]:25` — the square brackets disable MX lookup so it goes straight to that host
- **Username / Password:** leave blank (the VPS trusts cbcore by Tailscale IP; no SASL)

There's a **Test** button on the new entry — useful, but be aware it only proves the transport is *reachable*; it does **not** prove your domain routes through it. That's the next step.

Attach it to the domain (the step that actually routes mail): Mail Setup → **Domains → edit** `seabee.me` → set the **Sender-Dependent Transport** dropdown to your `[100.64.0.2]:25` entry → save. Until you do this, mailcow sends outbound *directly* from cbcore's CGNAT IP, which fails SPF at the recipient (mail-tester will report `SPF fail ... ip=<your home IP>`).

Then send from a real `@seabee.me` mailbox to mail-tester.com and confirm SPF, DKIM, DMARC all pass **and the sending IP shown is the VPS (`203.57.114.42`)**, not your home IP.

> **Troubleshooting `loops back to myself`:** if the bounce says `mail for [100.64.0.2]:25 loops back to myself`, the relay is sharing cbcore's hostname — check `sudo postconf myhostname smtp_helo_name` on the VPS reads `vps.seabee.me`, not `mail.seabee.me` (see the loop note in 9a). Watch the actual handoff on cbcore with `docker compose logs -f postfix-mailcow` while sending: a healthy send shows `relay=100.64.0.2[100.64.0.2]:25 ... status=sent`; the loop shows `greeted me with my own hostname`.

> **Hardening option:** if you'd rather not trust by IP (e.g. you expect the tailnet to grow), run the VPS relay on submission (587) with SASL auth instead of `mynetworks`, and put credentials in the mailcow relayhost fields. IP-trust over a Tailscale-ACL-restricted link is defensible for a single known peer; SASL is the belt-and-suspenders version.

---

10. First login and initial setup (order matters)

```
https://mail.seabee.me/admin
```

Default credentials: `admin` / `moohoo` — **change this immediately** under Configuration → Access → Admin details.

Then set things up **in this order** — several later steps silently no-op if the domain doesn't exist yet:

1. **Add the domain first.** Configuration → Mail Setup → **Domains** → Add `seabee.me`. Leave all the **Relay** checkboxes *unchecked* — those are for relaying inbound mail to *another* server; your mailboxes live here. Choose "add domain and restart SOGo" so webmail picks it up.
2. **Add a mailbox.** Mailboxes → add `you@seabee.me` with a password. This is the account you send/receive as.
3. **Now generate DKIM** (Configuration → ARC/DKIM keys): pick `seabee.me`, selector `dkim`, **2048** bits. Do *not* do this before step 1 — DKIM generation returns a success (HTTP 200) but stores nothing if the domain doesn't exist, and the UI list stays mysteriously empty.
4. **Attach the outbound relay** (Section 9b): Mail Setup → Domains → edit `seabee.me` → set the Sender-Dependent Transport to your `[100.64.0.2]:25` relay.

> **Where DKIM keys actually live:** current mailcow stores them in **Redis**, not on disk — there is no `data/dkim/keys/` directory (its absence is normal). To verify a key exists from the CLI: `source mailcow.conf && docker compose exec redis-mailcow redis-cli -a "$REDISPASS" HKEYS DKIM_PUB_KEYS` (should list `seabee.me`), and `... HGET DKIM_PUB_KEYS seabee.me` prints the `v=DKIM1;...` TXT record to publish.

---

11. Post-install checklist

- **DKIM**: generate the key (step 3 above), then publish the TXT record it gives you — Host `dkim._domainkey` (Namecheap appends the domain; do **not** type the full `dkim._domainkey.seabee.me`), Value the full `v=DKIM1;...` string
- **Change the default admin password** (step 10)
- **Confirm outbound relay works** (Section 9) — mail-tester should show the VPS as the sending IP with SPF/DKIM/DMARC all passing

- **Confirm the mail-port cert is real** (Section 4a) — the `openssl s_client` check in Section 12 should show Let's Encrypt, not the mailcow snake-oil CA; confirm the `mailcow-cert-pull.timer` is enabled (`systemctl list-timers | grep cert-pull`)
- **Confirm webmail loads off-tailnet** — log into `https://mail.seabee.me/` from a phone on mobile data (Tailscale off) and check it stays on `mail.seabee.me` (depends on `HTTP_REDIRECT=n` + `ADDITIONAL_SERVER_NAMES` + the 8005 Caddy backend from Sections 4/8)
- **DMARC**: start at `p=none` (Section 2), confirm a mail-tester send passes SPF+DKIM+DMARC, *then* tighten `p=none` → `p=quarantine` → `p=reject`
- **Fail2Ban**: built into Mailcow, verify it's enabled in the admin panel
- **Watchdog notifications**: configure so you get alerted if a component goes down
- **Updates**: run `./update.sh` from `/home/conor/Docker/mailcow` monthly at minimum; more often for security patches. Because all your interface bindings now live in `mailcow.conf` (not a compose override), `./update.sh` can overwrite `docker-compose.yml` freely without touching your port config.

---

12. Testing

```
# From an external machine, confirm each port reaches Mailcow
openssl s_client -starttls smtp -connect mail.seabee.me:587
openssl s_client -connect mail.seabee.me:993
```

Verify the presented certificate is real, not self-signed (catches the Section 4a failure mode — a snakeoil cert on the mail ports). The issuer should be Let's Encrypt and the dates current:

```
openssl s_client -connect mail.seabee.me:993 2>/dev/null </dev/null \
  | openssl x509 -noout -issuer -subject -dates
# issuer= ... Let's Encrypt ...   subject=CN=mail.seabee.me   notAfter in the future
openssl s_client -starttls smtp -connect mail.seabee.me:587 2>/dev/null </dev/null \
  | openssl x509 -noout -issuer
```

If the issuer shows the mailcow self-signed CA, the cert sync hasn't landed — on cbcore run `/usr/local/bin/pull-mailcow-cert.sh` by hand and read its output (Section 4a). On the DNS-01 alternative, check `docker compose logs acme-mailcow` instead.

DNS/deliverability checks:

- MX Toolbox — MX, SPF, blacklist check
- Mail-tester.com — send a test email, get a deliverability score covering SPF/DKIM/DMARC
- Send to `check-auth@verifier.port25.com` for a raw authentication report

Quick reference: port map

Service	Port	VPS component	Forwards to
Webmail/Admin UI	443	Caddy (HTTPS)	home:8005 (HTTP; HTTP_REDIRECT=n)
SMTP	25	HAProxy (TCP)	home:25
Submission	587	HAProxy (TCP)	home:587
SMTPS	465	HAProxy (TCP)	home:465
IMAPS	993	HAProxy (TCP)	home:993
ManageSieve	4190	HAProxy (TCP)	home:4190

Appendix: interface bindings at a glance

All host-interface scoping is done in `mailcow.conf` — no override file at all with the default cert setup (Option B, Section 4a). A `docker-compose.override.yml` only appears if you choose the DNS-01 alternative, and even then only to pass the ACME provider credentials to `acme-mailcow` — never for ports.

Variable	Value	Notes
`HTTP_BIND` / `HTTP_PORT`	`100.64.0.3` / `8005`	Separate bind+port pair (http quirk)
`HTTPS_BIND` / `HTTPS_PORT`	`100.64.0.3` / `8443`	Separate bind+port pair
`SMTP_PORT`	`100.64.0.3:25`	Single `IP:PORT` form
`SMTPS_PORT`	`100.64.0.3:465`	
`SUBMISSION_PORT`	`100.64.0.3:587`	
`IMAP_PORT`	`100.64.0.3:143`	
`IMAPS_PORT`	`100.64.0.3:993`	
`SIEVE_PORT`	`100.64.0.3:4190`	
`DOVEADM_PORT`	`127.0.0.1:19991`	Leave local-only
`SQL_PORT`	`127.0.0.1:13306`	Leave local-only
`REDIS_PORT`	`127.0.0.1:7654`	Leave local-only