WAIT

KI-gestütztes IT-Helpdesk

Installationsanleitung

Vom leeren Debian-Server zur laufenden WAIT-Instanz mit GPU-beschleunigter KI und HTTPS

Stand: November 2026 · Referenzserver: Debian 13 (trixie), NVIDIA GeForce RTX 2060 (6 GB)
Repository: `github.com/I3MC-eu/wait` (privat) · Adresse: `https://wait.i3mc.fun`

Inhalt

1. Überblick 3
1.1 Voraussetzungen 3
2. Debian installieren 5
2.1 sudo-Rechte für deinen Benutzer 5
2.2 DNS reparieren, wenn apt keine Namen auflöst 5
2.3 Optional: Anmeldung mit SSH-Schlüssel 5
3. NVIDIA-Treiber 6
3.1 Ohne Desktop starten 6
4. Docker und das NVIDIA Container Toolkit 7
4.1 Docker 7
4.2 Containern die GPU freigeben 7
5. Den Code holen 8
6. Konfigurieren 9
6.1 Geheimnisse und Modelle (.env) 9
6.2 Modelle für deine GPU wählen 9
6.3 Serverspezifische Einstellungen 10
6.4 HTTPS mit eigener Domain 11
6.5 Anmeldung über Active Directory 12
7. WAIT starten 14
7.1 Prüfen, ob alles läuft 14
7.2 Erste Anmeldung 14
8. Laufender Betrieb 15
8.1 Auf die neueste Version aktualisieren 15
8.2 Sicherungen 15
8.3 Nützliche Befehle 15
9. Fehlerbehebung 16

1. Überblick

Diese Anleitung installiert WAIT auf einem einzelnen Linux-Server. Alles läuft in Docker-Containern, die nach einem Neustart automatisch wieder starten. Danach stellt der Server Folgendes bereit:

Adresse	Was es ist
`https://wait.i3mc.fun`	Die WAIT-App mit gültigem Let's-Encrypt-Zertifikat: Portal, Tickets und Admin-Bereich unter `/admin`
`http://wait.i3mc.fun`	Leitet automatisch auf die HTTPS-Adresse weiter
`http://<server-ip>/`	Produkt-Website (Überblick und Preise), ohne Anmeldung
`http://<server-ip>:8080`	Die App direkt ohne HTTPS (Fallback, zum Beispiel zur Fehlersuche)
`http://<server-ip>:8000`	Backend-API (wird von der App genutzt, normalerweise nicht direkt aufgerufen)
`http://<server-ip>:11434`	Ollama, der lokale KI-Modellserver

Die Umgebung besteht aus sechs Containern. Fünf sind in `docker-compose.yml` definiert, der HTTPS-Proxy kommt über die serverspezifische Datei aus Abschnitt 6.3 dazu:

Dienst	Aufgabe
`db`	PostgreSQL-17-Datenbank (Tickets, Benutzer, Wissensdatenbank)
`ollama`	Führt die KI-Modelle auf der GPU aus: ein Textmodell für Chat-Antworten und ein Bildmodell für Screenshots
`backend`	FastAPI-Anwendung: Anmeldung (lokal und Active Directory), Tickets, Wissensdatenbank, der WAIT-Assistent
`frontend`	nginx liefert die React-App aus und leitet `/api` an das Backend weiter
`site`	nginx liefert die statische Produkt-Website aus
`caddy`	HTTPS-Proxy für `wait.i3mc.fun`: holt und erneuert das Zertifikat automatisch

> **Hinweis:** Ersetze in dieser Anleitung `192.168.92.110` durch die IP-Adresse deines Servers, `wait` durch deinen Linux-Benutzernamen und `wait.i3mc.fun` durch deine Adresse, falls sie abweichen.

1.1 Voraussetzungen

Komponente	Minimum	Referenzserver
CPU	4 Kerne, x86-64	16 Kerne
RAM	16 GB	16 GB
Festplatte	60 GB SSD	450 GB NVMe
GPU	NVIDIA mit 6 GB VRAM, vom Treiber unterstützt (Turing oder neuer)	GeForce RTX 2060, 6 GB
Betriebssystem	Debian 13 (trixie), Minimalinstallation	Debian 13 netinst
Netzwerk	Feste LAN-IP, Internetzugang für die Installation (ausgehend Port 443)	192.168.92.110
Domain	Eine eigene Domain bei Cloudflare (kostenloser Plan), nur für HTTPS	`i3mc.fun`

Ohne GPU funktioniert die App ebenfalls, aber jede KI-Antwort läuft dann auf der CPU und dauert 30 bis 60 Sekunden statt 5 bis 10. Überspringe in diesem Fall die NVIDIA-Abschnitte.

2. Debian installieren

1. Installiere Debian 13 vom netinst-Image. Wähle bei der **Softwareauswahl** nur **SSH-Server** und **Standard-Systemwerkzeuge**. Eine Desktop-Umgebung wird nicht gebraucht und belegt GPU-Speicher.
2. Lege einen normalen Benutzer an (in dieser Anleitung `wait`) und setze ein Root-Passwort.
3. Gib dem Server eine feste IP-Adresse (im Installer oder als DHCP-Reservierung im Router).
4. Melde dich nach dem ersten Start von deinem PC aus an: `ssh wait@192.168.92.110`.

2.1 sudo-Rechte für deinen Benutzer

Eine Debian-Minimalinstallation hat oft kein `sudo`. Werde einmal mit dem Root-Passwort root und installiere es:

```
su -
apt-get update && apt-get install -y sudo
echo "wait ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/wait
chmod 440 /etc/sudoers.d/wait
exit
```

> **Hinweis:** Mit `NOPASSWD` können Skripte `sudo` ohne Passworteingabe ausführen. Wenn du lieber nach dem Passwort gefragt werden willst, nutze stattdessen `usermod -aG sudo wait` und melde dich neu an.

2.2 DNS reparieren, wenn apt keine Namen auflöst

Wenn `apt-get update` mit „Temporary failure resolving“ abbricht, hat der DHCP-Server keinen Nameserver geliefert. Setze einen:

```
printf "nameserver 1.1.1.1\nnameserver 8.8.8.8\n" | sudo tee /etc/resolv.conf
```

2.3 Optional: Anmeldung mit SSH-Schlüssel

Erstelle auf deinem Windows-PC (PowerShell) einmalig einen Schlüssel und kopiere ihn auf den Server. Das Server-Passwort gibst du dabei ein letztes Mal ein:

```
ssh-keygen -t ed25519 -f $env:USERPROFILE\.ssh\wait_deploy_ed25519 -N '""'
type $env:USERPROFILE\.ssh\wait_deploy_ed25519.pub | ssh wait@192.168.92.110 "mkdir -p
~/.ssh && cat >> ~/.ssh/authorized_keys && chmod 700 ~/.ssh && chmod 600
~/.ssh/authorized_keys"
```

Ab dann: `ssh -i ~/.ssh/wait_deploy_ed25519 wait@192.168.92.110`.

> **Wichtig:** Führe diese beiden Befehle auf deinem PC aus, nicht in einer SSH-Sitzung auf dem Server. Zeigt deine Eingabezeile `wait@...:~$`, bist du auf dem Server: Tippe zuerst `exit`.

3. NVIDIA-Treiber

Der Treiber liegt im `non-free`-Bereich von Debian. Aktiviere ihn, installiere den Treiber und starte neu:

```
sudo sed -i 's/ main non-free-firmware$/ main non-free-firmware contrib non-free/'
/etc/apt/sources.list
sudo apt-get update
sudo apt-get install -y linux-headers-amd64 nvidia-driver firmware-misc-nonfree nvidia-
smi
sudo reboot
```

Prüfe nach dem Neustart, ob die Karte erkannt wird:

```
nvidia-smi
```

Du solltest den GPU-Namen, die Treiberversion (550 oder neuer) und den Gesamtspeicher sehen. Meldet der Befehl keine Geräte und startet der Server mit **Secure Boot**, deaktiviere Secure Boot im BIOS oder registriere den Signaturschlüssel des Treibers.

3.1 Ohne Desktop starten

Falls ein Desktop installiert wurde, schalte ihn ab, damit er keinen GPU-Speicher belegt:

```
sudo systemctl set-default multi-user.target
sudo systemctl isolate multi-user.target
```

4. Docker und das NVIDIA Container Toolkit

4.1 Docker

Installiere Docker aus dem offiziellen Docker-Repository (das Debian-Paket ist für Compose v2 zu alt):

```
sudo apt-get install -y ca-certificates curl gnupg git
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/debian $(. /etc/os-release && echo $VERSION_CODENAME)
stable" | sudo tee /etc/apt/sources.list.d/docker.list
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
docker-buildx-plugin
sudo usermod -aG docker wait
```

Melde dich ab und wieder an, damit die Gruppe `docker` wirksam wird.

4.2 Containern die GPU freigeben

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o
/etc/apt/keyrings/nvidia-container-toolkit.gpg
curl -fsSL https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-
toolkit.list | sed 's#deb https://#deb [signed-by=/etc/apt/keyrings/nvidia-container-
toolkit.gpg] https://#g' | sudo tee /etc/apt/sources.list.d/nvidia-container-
toolkit.list
sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker
```

Test: Dieser Befehl muss den GPU-Namen ausgeben:

```
docker run --rm --gpus all ubuntu nvidia-smi -L
```

5. Den Code holen

Das Repository ist privat. Der Server bekommt deshalb einen eigenen **Deploy-Key** mit reinem Lesezugriff:

```
ssh-keygen -t ed25519 -f ~/.ssh/github_deploy_key -N "" -C "wait-server"
cat ~/.ssh/github_deploy_key.pub
```

1. Kopiere die ausgegebene Zeile (sie beginnt mit `ssh-ed25519`).
2. Öffne auf GitHub **I3MC-eu/wait → Settings → Deploy keys → Add deploy key**.
3. Title: der Name des Servers. Key: die Zeile einfügen. **Allow write access** nicht anhaken. Auf **Add key** klicken.

Sag SSH, dass es diesen Schlüssel für GitHub nutzen soll, und klone dann:

```
cat >> ~/.ssh/config <<'EOF'
Host github.com-wait
    HostName github.com
    User git
    IdentityFile ~/.ssh/github_deploy_key
    IdentitiesOnly yes
EOF
chmod 600 ~/.ssh/config
git clone git@github.com-wait:I3MC-eu/wait.git ~/wait
cd ~/wait
```

6. Konfigurieren

6.1 Geheimnisse und Modelle (.env)

Lege `~/wait/.env` an. Die beiden Geheimnisse werden zufällig erzeugt; verwende sie nie auf mehreren Servern:

```
cd ~/wait
cat > .env <<EOF
POSTGRES_PASSWORD=$(openssl rand -hex 24)
JWT_SECRET=$(openssl rand -hex 32)
ALLOWED_EMAIL_DOMAINS=
SEED_ON_START=1
APP_URL=https://wait.i3mc.fun
SITE_PORT=127.0.0.1:8081
OLLAMA_MODEL=qwen2.5:7b
OLLAMA_VISION_MODEL=qwen2.5vl:3b
EOF
chmod 600 .env
```

Einstellung	Bedeutung
`POSTGRES_PASSWORD`	Datenbank-Passwort. Wird nur zwischen den Containern verwendet.
`JWT_SECRET`	Signiert die Anmelde-Tokens. Eine Änderung meldet alle ab.
`ALLOWED_EMAIL_DOMAINS`	Kommagetrennte Domains, die sich selbst registrieren dürfen (zum Beispiel `schule.at`). Leer = alle.
`SEED_ON_START`	1 legt beim ersten Start die Demo-Benutzer und die 62 Anleitungen (DE + EN) an. Kann aktiv bleiben.
`APP_URL`	Ziel der „WAIT öffnen“-Buttons auf der Website.
`SITE_PORT`	Lokaler Port der Website. Mit HTTPS übernimmt Caddy Port 80, deshalb nur `127.0.0.1:8081`. Ohne HTTPS: weglassen (Standard 80).
`OLLAMA_MODEL`	Textmodell für Chat-Antworten und Übergabezusammenfassungen.
`OLLAMA_VISION_MODEL`	Bildmodell, das hochgeladene Bilder beschreibt.

6.2 Modelle für deine GPU wählen

Die Werte wurden auf der Referenz-RTX-2060 (6 GB) gemessen. Wähle das größte Textmodell, das **vollständig** in den GPU-Speicher passt. Läuft es in den Arbeitsspeicher über, wird es um ein Vielfaches langsamer.

Modell	Zweck	Speicher	Tempo	Ergebnis
`qwen2.5:7b`	Text	4,8 GB	~58 Tokens/s	**Gewählt.** Klügstes Modell, das ganz passt.
`llama3.1:8b`	Text	5,6 GB	~33 Tokens/s	Passt nicht, läuft teilweise auf der CPU.
`gemma3:4b`	Text + Bild	2,9 GB	~76 Tokens/s	Schnell, aber spürbar schwächer.
`llama3.2:3b`	Text	2,6 GB	~115 Tokens/s	Alter Standard: schnell, aber schwach.
`qwen2.5vl:3b`	Bild	2,8 GB	~1–4 s/Bild	**Gewählt.** Liest Fehlercodes exakt.

Modell	Zweck	Speicher	Tempo	Ergebnis
`moondream`	Bild	1,2 GB	schnell	Lieferte leere Beschreibungen.

Mit 8 GB VRAM oder mehr lohnt sich ein Test mit einem größeren Textmodell wie `qwen2.5:14b` (braucht ~10 GB). Ändere den Namen in `.env` und führe `docker compose up -d` aus; das Modell wird automatisch heruntergeladen.

> **Wichtig:** Ohne diese beiden Zeilen in `.env` nutzt WAIT die alten Standardmodelle `llama3.2:3b` und `moondream`.

6.3 Serverspezifische Einstellungen

Lege `~/wait/docker-compose.override.yml` an. Docker Compose lädt sie automatisch, und weil sie nicht im Repository liegt, überschreibt `git pull` sie nie:

```
services:
  ollama:
    environment:
      OLLAMA_KEEP_ALIVE: 24h
      OLLAMA_MAX_LOADED_MODELS: "1"
      OLLAMA_NUM_PARALLEL: "1"
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
  backend:
    environment:
      CORS_ORIGINS: https://wait.i3mc.fun,http://192.168.92.110:8080
    extra_hosts:
      - "WIN-8G6NFMFI9UE.wait.internal:192.168.92.111"
  caddy:
    build: ./https
    restart: unless-stopped
    depends_on: [frontend, site]
    environment:
      CF_API_TOKEN: ${CF_API_TOKEN}
    ports: ["80:80", "443:443"]
    volumes:
      - ./https/Caddyfile:/etc/caddy/Caddyfile:ro
      - caddy-data:/data
      - caddy-config:/config

volumes:
  caddy-data:
  caddy-config:
```

Zeile	Warum
`deploy … gpu`	Gibt dem Ollama-Container die Grafikkarte.
`OLLAMA_KEEP_ALIVE: 24h`	Hält das Modell geladen, damit die erste Antwort des Tages nicht langsam ist.

Zeile	Warum
`OLLAMA_MAX_LOADED_MODELS: "1"`	Nur ein Modell gleichzeitig auf der GPU. Beide zusammen (~7,6 GB) passen nicht in 6 GB; der Versuch führte zu einem GPU-Fehler, der stillschweigend alles auf die CPU verlagerte. Ollama wechselt jetzt zwischen den Modellen (dauert ein paar Sekunden).
`CORS_ORIGINS`	Die Adressen, unter denen die App geöffnet wird. Sie müssen exakt passen, sonst blockiert der Browser jede Anfrage.
`extra_hosts`	Damit der Backend-Container den Domänencontroller per Namen findet (nötig für LDAPS, siehe Abschnitt 6.5). Nur bei Active-Directory-Anmeldung.
`caddy`	Der HTTPS-Proxy (siehe Abschnitt 6.4). Ohne HTTPS weglassen.

Blende die Dateien in `git status` aus:

```
printf 'docker-compose.override.yml\nhttps/\n' >> .git/info/exclude
```

6.4 HTTPS mit eigener Domain

WAIT bekommt ein echtes Let's-Encrypt-Zertifikat, ohne dass der Server aus dem Internet erreichbar sein muss. Caddy beweist den Besitz der Domain über einen DNS-Eintrag bei Cloudflare (DNS-01-Verfahren). Dafür muss nur ausgehend Port 443 offen sein; die Firewall muss nicht geändert werden.

> **Hinweis:** Cloudflare Tunnel funktioniert im Schulnetz nicht, weil ausgehend Port 7844 gesperrt ist. Deshalb ist WAIT nur im Schulnetz erreichbar.

Schritt 1: Domain und API-Token bei Cloudflare

1. Die Domain (hier `i3mc.fun`) muss bei Cloudflare liegen (kostenloser Plan, Nameserver beim Domain-Anbieter auf Cloudflare umgestellt).
2. **My Profile → API Tokens → Create Token → Create Custom Token**.
3. Berechtigungen: **Zone → DNS → Edit** und **Zone → Zone → Read**. Zone Resources: **Include → Specific zone → i3mc.fun**. Ablauf: **No expiration**, sonst bricht die Verlängerung des Zertifikats später unbemerkt ab.
4. Token in `~/wait/.env` eintragen: `CF_API_TOKEN=...`

Schritt 2: DNS-Eintrag

Lege bei Cloudflare unter **DNS → Records** einen A-Eintrag an: Name `wait`, Inhalt `192.168.92.110`, **Proxy aus** (graue Wolke, „DNS only“). Die Adresse zeigt so auf die interne IP und funktioniert nur im Schulnetz.

Schritt 3: Caddy einrichten

```
mkdir -p ~/wait/https && cd ~/wait/https
cat > Dockerfile <<'EOF'
FROM caddy:2.10-builder AS builder
RUN xcaddy build --with github.com/caddy-dns/cloudflare
FROM caddy:2.10
COPY --from=builder /usr/bin/caddy /usr/bin/caddy
EOF
cat > Caddyfile <<'EOF'
wait.i3mc.fun {
      tls {
             dns cloudflare {env.CF_API_TOKEN}
```

```
            resolvers 1.1.1.1
      }
      reverse_proxy frontend:80
}

:80 {
      reverse_proxy site:80
}
EOF
```

Der erste Block liefert die App über HTTPS aus; `http://wait.i3mc.fun` leitet Caddy automatisch dorthin weiter. Der zweite Block zeigt unter `http://192.168.92.110` weiterhin die Produkt-Website. Starten:

```
cd ~/wait
docker compose up -d --build caddy site backend
docker compose logs caddy | grep -i "certificate obtained"
```

Nach etwa einer Minute erscheint „certificate obtained successfully“. Das Zertifikat ist 90 Tage gültig und wird von Caddy automatisch rechtzeitig erneuert.

6.5 Anmeldung über Active Directory

> **Hinweis:** Die ausführliche Schritt-für-Schritt-Anleitung (Dienstkonto, Admin-Gruppe, LDAPS-Zertifikat am Domänencontroller, Tests und Fehlersuche) steht in der separaten **AD-Login-Anleitung** (`WAIT-AD-Login-Anleitung`). Hier nur die Kurzfassung.

Schüler:innen und Lehrkräfte melden sich mit ihrem Schulkonto an. WAIT sucht die Person mit einem Dienstkonto mit Lesezugriff und prüft dann das eingegebene Passwort direkt am Domänencontroller über LDAPS. AD-Passwörter werden nie in WAIT gespeichert. Ergänze `~/wait/.env`:

```
AUTH_MODE=both
LDAP_URL=ldaps://WIN-8G6NFMFI9UE.wait.internal:636
LDAP_BASE_DN=DC=wait,DC=internal
LDAP_BIND_DN=CN=svc-wait,CN=Users,DC=wait,DC=internal
LDAP_BIND_PASSWORD=<Passwort des Dienstkontos>
LDAP_ADMIN_GROUP=CN=WAIT-
Admins,CN=Users,DC=wait,DC=internal;CN=Administratoren,CN=Builtin,DC=wait,DC=internal
LDAP_USER_GROUP=
LDAP_EMAIL_DOMAIN=wait.internal
LDAP_CA_CERT_FILE=/certs/ca.pem
```

Einstellung	Bedeutung
`AUTH_MODE`	`local` = nur WAIT-Konten, `ldap` = nur AD, `both` = zuerst AD, lokale Konten (zum Beispiel der Seed-Admin) als Rückfallebene.
`LDAP_URL`	LDAPS-Adresse des Domänencontrollers. Der Name muss zum Zertifikat passen; deshalb der Eintrag `extra_hosts` in Abschnitt 6.3.
`LDAP_BIND_DN` / `LDAP_BIND_PASSWORD`	Dienstkonto mit reinem Lesezugriff, das Benutzer sucht.
`LDAP_ADMIN_GROUP`	Mitglieder werden WAIT-Admins (verschachtelte Gruppen zählen). Mehrere Gruppen mit `;` trennen.
`LDAP_USER_GROUP`	Optional: nur Mitglieder (oder Admins) dürfen sich anmelden. Leer = jedes AD-Konto.

Einstellung	Bedeutung
`LDAP_EMAIL_DOMAIN`	Baut `benutzer@domain` für AD-Konten ohne E-Mail-Attribut.
`LDAP_CA_CERT_FILE`	Zertifikat der internen Zertifizierungsstelle, falls das DC-Zertifikat nicht öffentlich ist. Datei nach `~/wait/certs/ca.pem` legen.

AD-Konten werden bei der ersten Anmeldung angelegt; Name und Admin-Status aus der AD-Gruppe werden bei jeder Anmeldung aktualisiert. Wer im Admin-Bereich mit **Zum Admin machen** befördert wurde, bleibt Admin, auch wenn er nicht in einer AD-Gruppe ist. Übernehmen mit `docker compose up -d backend`. Schnelltest vom Server: `nc -zv 192.168.92.111 636`.

7. WAIT starten

```
cd ~/wait
docker compose up -d --build
```

Der erste Start baut die Images und lädt die KI-Modelle herunter (etwa 8 GB). Plane 5 bis 15 Minuten ein. Den Modell-Download verfolgst du mit `docker compose logs -f ollama` (Strg+C beendet die Anzeige).

7.1 Prüfen, ob alles läuft

```
docker compose ps                                  # alle Dienste "Up", db und ollama
"healthy"
curl -s https://wait.i3mc.fun/api/health; echo     # {"status":"ok"}
docker compose exec ollama ollama list             # beide Modelle aufgelistet
docker compose exec ollama ollama ps               # PROCESSOR muss "100% GPU" zeigen
```

> **Hinweis:** `ollama ps` zeigt ein Modell erst, nachdem es einmal benutzt wurde. Öffne die App, lege ein Testticket an und führe den Befehl erneut aus. Steht dort CPU statt GPU, siehe Fehlerbehebung.

7.2 Erste Anmeldung

Öffne `https://wait.i3mc.fun` im Browser. Der Seed legt zwei lokale Konten an:

Rolle	E-Mail	Passwort
Admin (IT-Team)	`admin@wait.dev`	`admin-wait-2026`
Nutzer:in (Demo)	`j.alvarez@company.com`	`wait-demo-2026`

> **Wichtig:** Ändere beide Passwörter sofort (Profilmenü → Passwort ändern) oder lösche die Demo-Konten. Wer diese Anleitung liest, kennt die Standardwerte. Mit Active Directory meldet sich das IT-Team ohnehin mit dem eigenen Schulkonto an.

8. Laufender Betrieb

8.1 Auf die neueste Version aktualisieren

```
cd ~/wait
git pull origin main
docker compose up -d --build backend frontend site
docker compose restart frontend caddy
```

Datenbank-Migrationen laufen beim Start des Backends automatisch. Der abschließende `restart` ist nötig, wenn das Backend neu erstellt wurde (siehe Fehlerbehebung, „502-Fehler“).

8.2 Sicherungen

Sichere die Datenbank vor jedem Update und regelmäßig (zum Beispiel täglich per cron):

```
mkdir -p ~/backups
cd ~/wait && docker compose exec -T db sh -c 'pg_dump -U $POSTGRES_USER $POSTGRES_DB' |
gzip > ~/backups/wait-$(date +%F-%H%M).sql.gz
```

Hochgeladene Bilder liegen im Docker-Volume `wait_uploads`. Sichere außerdem `.env`, `docker-compose.override.yml` und den Ordner `https/`, denn sie sind nicht im Repository. Kopiere alles vom Server weg: Eine Sicherung auf derselben Platte überlebt keinen Plattendefekt.

Sicherung wiederherstellen

```
cd ~/wait
docker compose stop backend
gunzip -c ~/backups/wait-JJJJ-MM-TT-HHMM.sql.gz | docker compose exec -T db sh -c 'psql
-U $POSTGRES_USER -d $POSTGRES_DB'
docker compose start backend && docker compose restart frontend
```

> **Wichtig:** In eine leere Datenbank wiederherstellen. Zum vorherigen Leeren: `docker compose exec db sh -c 'dropdb -U $POSTGRES_USER $POSTGRES_DB && createdb -U $POSTGRES_USER $POSTGRES_DB'`. Das löscht alle aktuellen Daten.

8.3 Nützliche Befehle

Befehl (in ~/wait)	Wirkung
`docker compose ps`	Status aller Dienste
`docker compose logs -f backend`	Live-Log des Backends (Fehler, Anfragen)
`docker compose logs caddy`	HTTPS-Proxy und Zertifikat
`docker compose restart <dienst>`	Einen Dienst neu starten
`docker compose down` / `up -d`	Alles stoppen / starten (Daten bleiben erhalten)
`nvidia-smi`	GPU-Speicher und Temperatur
`docker compose exec ollama ollama ps`	Welches Modell geladen ist, GPU oder CPU

Alle Dienste starten nach einem Neustart oder Absturz automatisch wieder (`restart: unless-stopped`).

9. Fehlerbehebung

KI-Antworten dauern 30 bis 60 Sekunden, CPU auf 100 %

Ollama führt das Modell auf der CPU aus. Prüfen mit `docker compose exec ollama ollama ps`: PROCESSOR zeigt `CPU`. Meist hatte die GPU einen Fehler (`sudo dmesg | grep Xid`). **Lösung: Server neu starten.** Stelle sicher, dass `OLLAMA_MAX_LOADED_MODELS: "1"` gesetzt ist (Abschnitt 6.3). Wurde die Override-Datei überschrieben, fehlen auch die GPU-Zeilen.

Die Seite zeigt nach einem Update eine alte Version

Einmal **Strg+F5** drücken. Aktuelle Versionen sagen dem Browser, dass er die Seite nicht zwischenspeichern soll.

Jede Aktion scheitert mit „502 Bad Gateway“

Das Backend wurde neu erstellt und nginx zeigt noch auf den alten Container: `docker compose restart frontend`.

„Internal Server Error“ bei der AD-Anmeldung

`docker compose logs backend | grep -i ldap`. Häufige Ursachen: Domänencontroller nicht erreichbar (`nc -zv 192.168.92.111 636`), der Name in `LDAP_URL` passt nicht zum Zertifikat, oder `LDAP_TIMEOUT_SECONDS` ist keine ganze Zahl.

Ein Admin verliert bei jeder Anmeldung seine Rechte

Ist er über AD Admin, muss er in einer Gruppe aus `LDAP_ADMIN_GROUP` sein. Sonst im Admin-Bereich unter **Team** einmal **Zum Admin machen**; das wird dauerhaft gespeichert.

`wait.i3mc.fun` öffnet sich auf einem PC nicht

Auf dem PC `nslookup wait.i3mc.fun` ausführen. Kommt `192.168.92.110`, ist der Name in Ordnung und der PC erreicht den Server nicht (anderes Netz, Gast-WLAN). Kommt keine Adresse, nutzt der PC einen DNS-Server, der den Eintrag nicht kennt: dort einen A-Eintrag `wait.i3mc.fun → 192.168.92.110` anlegen.

Kein Zertifikat, Caddy meldet Fehler

`docker compose logs caddy`. Meist fehlt `CF_API_TOKEN` in `.env`, der Token ist abgelaufen oder hat nicht beide Berechtigungen (DNS Edit, Zone Read).

Anmeldung klappt nur am Server selbst, Anfragen blockiert

`CORS_ORIGINS` passt nicht zur Adresse im Browser (Protokoll, Name und Port müssen exakt stimmen). Korrigieren, dann `docker compose up -d backend && docker compose restart frontend`.

„Ich konnte deinen Screenshot nicht auswerten“

`docker compose logs backend | grep vision` prüfen. Bilder werden automatisch auf 1024 Pixel verkleinert; scheitert das Bildmodell trotzdem, lädt das Backend es neu und versucht es noch einmal. Bei dauerhaften Fehlern: `docker compose restart ollama`.

Container `ollama` startet nicht: Adresse bereits belegt

Ollama wurde zusätzlich direkt auf dem Host installiert: `sudo systemctl disable --now ollama`.

`nvidia-smi`: „No devices were found“

Treiber nicht geladen: nach der Installation neu starten, Secure Boot prüfen (Abschnitt 3) und ob die Karte richtig sitzt und Strom hat.