5G Core (5GC)

The 5G Standalone (SA) core is deployed by the `services/5gc.yml` playbook. Every network function (NF) is installed by a single shared role — `5gc_nf` — parameterised per NF. This page explains that role, the NFs it builds, and the configuration each one reads.

Overview

Rather than one role per NF, OmniCore uses **one role (`5gc_nf`) for all eleven 5GC NFs**. The playbook invokes the role once per NF and passes an `nf_package` variable that selects the Debian package, the systemd service, and the Jinja2 config template. This keeps install, config-versioning, licensing, and restart logic identical across every NF.

```
- name: Setup AMF
  hosts: amf
  roles:
    - {role: 5gc_nf, nf_package: omniamf, become: yes}
```

Network Functions

Each NF has its own inventory group, package, systemd service, and config template (`roles/5gc_nf/templates/<nf_package>.j2`).

NF	`nf_package`	Inventory group	Config template	Role
Network Repository Function	`omninrf`	`nrf`	`omninrf.j2`	Service discovery / NF registration
Access & Mobility Management	`omniamf`	`amf`	`omniamf.j2`	N1/N2 signalling, registration, connection mgmt
Authentication Server	`omniausf`	`ausf`	`omniausf.j2`	5G-AKA / EAP-AKA' authentication
Binding Support Function	`omnibsf`	`bsf`	`omnibsf.j2`	PCF binding for a PDU session
Charging Function	`omnichf`	`chf`	`omnichf.j2`	Converged online/offline charging
Network Slice Selection	`omnissf`	`nssf`	`omnissf.j2`	NSSAI / slice selection
Policy Control Function	`omnipcf`	`pcf`	`omnipcf.j2`	Policy and QoS rules
Service Communication Proxy	`omniscp`	`scp`	`omniscp.j2`	SBI message routing / indirect comms

NF	`nf_package`	Inventory group	Config template	Role
Session Management Function	`omnismf`	`smf`	`omnismf.j2`	PDU session and UPF (N4) control
Unified Data Management	`omniudm`	`udm`	`omniudm.j2`	Subscription data, SUCI de-concealment
Unified Data Repository	`omniudr`	`udr`	`omniudr.j2`	Backing data store for UDM/PCF

How the `5gc_nf` role works

For each NF the role performs the same sequence:

1. **Resolve `group_vars_omnicore`** — set to `<inventory>/group_vars/`, so config lookups resolve relative to the active inventory.
2. **Discover license servers** — if `license_server_api_urls` is not set, it is derived from the `license_server` inventory group (`https://<host>:8443/api`).
3. **Install prerequisites** — `lsb-release`, `curl`, `gpg`, `unzip`, `libglu1-mesa`, `libsctp1`.
4. **Install the NF package** from the Omnitouch APT repository. Latest by default; pin with `nf_version` (see Version pinning).
5. **Template the runtime config (versioned)** — renders `templates/<nf_package>.j2` to `/etc/<nf_package>/runtime-<deb_version>.exs`.
6. **Symlink `runtime.exs`** → the active version's config, and mirror that symlink into each `/opt/<nf_package>/releases/*` directory.
7. **Restart the NF** (handler) whenever the active config changes.

Version pinning and rollback

Config is written per Debian version (`runtime-<deb_version>.exs`) and `runtime.exs` is a symlink to the active one. Previous configs are preserved on disk, so a rollback is just reinstalling the older package version:

```
- {role: 5gc_nf, nf_package: omniamf, nf_version: "1.2.2-1",
become: yes}
```

Omit `nf_version` to track the latest package.

UDM HNET keys (SUCI de-concealment)

The UDM (`nf_package: omniudm`) has extra steps to support SUCI de-concealment (TS 33.501 §6.12) — recovering the SUPI from the concealed SUCI a UE presents at registration. This needs the Home Network **private** keys that pair with the Home Network **public** keys on the USIMs.

- Keys live in `group_vars/hnet_udm/` on the Ansible controller as `<curve>-<id>.key` (PEM), where `<curve>` is `curve25519` (Profile A, X25519) or `secp256r1` (Profile B, prime256v1), and `<id>` is the Home Network Public Key Identifier.
- If the directory is **empty**, a default `curve25519-1.key` is auto-generated so de-concealment works out of the box. This is idempotent — an existing key is never overwritten.
- For each `curve25519-*.key`, the matching raw 32-byte public value is exported (hex) to a `.pub` file for USIM provisioning. `.pub` files stay in inventory; only `*.key` files deploy to `/etc/omniudm/hnet/` on the UDM (mode `0600`, root).

> An auto-generated key is random and only de-conceals SUCIs from USIMs programmed with its **matching** public key. Take the exported `.pub` and provision it as the Home Network Public Key on the USIM. SIMs using the null-scheme need no HNET key and can leave the directory empty.

Full detail and file-layout examples: Group Variables Configuration — Example 8.

Related Documentation

- Service Playbooks - All service playbooks and the deployment hierarchy
- Circuit-Switched Core - The legacy CS domain (BSC, MSC, SS7)
- Group Variables Configuration - Per-NF config templates and HNET keys
- Deployment Architecture - How the domains fit together
- **OmniCore Configuration**: https://docs.omnitouch.com.au/docs/repos/OmniCore - What goes in the NF config files

OmniCore IP Planning Standard

Overview

This document outlines the standard IP planning approach for OmniCore deployments. The architecture requires **four internal subnets** to properly segment network functions for security, performance, and operational clarity.

IP Allocation Requirements

Standard Allocation: Four /24 Subnets

Each OmniCore deployment requires four distinct subnets for internal networking:

1. **Packet Core Network** - First /24
2. **Signaling Network** - Second /24
3. **IMS Internal Network** - Third /24
4. **IMS Public Network** - Fourth /24

Important: These are Recommendations, Not Requirements

The subnet allocation described in this document is a **recommended best practice** for organizing OmniCore deployments. However, the architecture is **completely flexible**:

- **All hosts in one subnet**: You can place all components in a single subnet if that suits your deployment needs
- **Each host type in its own subnet**: You can create separate subnets for each component type (one for MMEs, one for HSS, etc.)

- **Custom groupings**: You can organize hosts into any subnet structure that makes sense for your specific requirements
- **Mix internal and public IPs**: Some hosts can use internal (RFC 1918) addresses while others use public IPs, all within the same deployment

The recommended four-subnet approach provides optimal **security isolation, traffic management, and operational clarity**, which is why we suggest it for production deployments. However, you should adapt the IP plan to fit your specific network topology, available address space, and operational requirements.

Network Segment Breakdown

1. Packet Core Network (First /24)

Purpose: User plane and core control plane elements

Components:

- OmniMME (Mobility Management Entity)
- OmniSGW (Serving Gateway)
- OmniPGW-C (PDN Gateway Control Plane)
- OmniUPF/PGW-U (User Plane Function / PDN Gateway User Plane)

Example: `10.179.1.0/24`

```
mme:
  hosts:
    customer-mme01:
      ansible_host: 10.179.1.15
      gateway: 10.179.1.1
      host_vm_network: "v400-omni-packet-core"
```

---

2. Signaling Network (Second /24)

Purpose: Diameter signaling, policy, charging, and management functions

Components:

- OmniHSS (Home Subscriber Server)
- OmniCharge OCS (Online Charging System)
- OmniHSS PCRF (Policy and Charging Rules Function)
- OmniDRA DRA (Diameter Routing Agent)
- DNS Servers (internal/infrastructure DNS - distinct from the customer-facing DNS in the IMS Public segment; both are intentional)
- TAP3/CDR Servers
- Monitoring/OAM
- SIP capture
- License Server
- RAN Monitor
- Omnitouch Warning Link CBC (Cell Broadcast Center) - if deployed

Example: `10.179.2.0/24`

```
hss:
  hosts:
    omni-site-hss01:
      ansible_host: 10.179.2.140
      gateway: 10.179.2.1
      host_vm_network: "v401-omni-signalling"
```

---

3. IMS Internal Network (Third /24)

Purpose: IMS core signaling and services (internal SIP signaling)

Components:

- OmniCSCF S-CSCF (Serving Call Session Control Function)

- OmniCSCF I-CSCF (Interrogating Call Session Control Function)
- OmniTAS (Telephony Application Server / Application Server)
- OmniMessage (SMS Controller, SMPP, IMS)

Example: `10.179.3.0/24`

```
scscf:
  hosts:
    omni-site-scscf01:
      ansible_host: 10.179.3.45
      gateway: 10.179.3.1
      host_vm_network: "v402-omni-ims-internal"
```

---

4. IMS Public Network (Fourth /24)

Purpose: User-facing IMS services, customer-facing DNS, and SS7/legacy interconnect

Components:

- OmniCSCF P-CSCF (Proxy Call Session Control Function)
- XCAP Servers
- Visual Voicemail Servers
- Customer DNS
- OmniSS7 STP (SS7 Signaling Transfer Point)
- OmniSS7 HLR (Home Location Register) - for 2G/3G
- OmniSS7 IP-SM-GW (MAP SMSc)
- OmniSS7 CAMEL Gateway
- MSC (Mobile Switching Centre)

> **Note:** SS7/HLR and related legacy interconnect functions commonly sit in the public segment, since they peer with external 2G/3G networks over SIGTRAN/M3UA rather than internal IMS SIP.

Example: `10.179.4.0/24`

```
pcscf:
  hosts:
    omni-site-pcscf01:
      ansible_host: 10.179.4.165
      gateway: 10.179.4.1
      host_vm_network: "v403-omni-ims-public"
```

Implementation Methods

OmniCore supports two primary methods for implementing this network segmentation:

Method 1: Physical/Virtual Network Interfaces (Recommended for Production)

Use separate NICs or virtual bridges for each network segment. This provides the strongest isolation and is the recommended approach for production deployments. The convention is to name each bridge after its segment (`v400-omni-packet-core`, `v401-omni-signalling`, `v402-omni-ims-internal`, `v403-omni-ims-public`), though a single shared bridge (e.g. `vmbr0`) is also valid where segmentation is handled elsewhere.

Example:

```yaml
# Packet Core - v400-omni-packet-core
mme:
  hosts:
    omni-site-mme01:
      ansible_host: 10.179.1.15
      gateway: 10.179.1.1
      host_vm_network: "v400-omni-packet-core"

# Signaling - v401-omni-signalling
hss:
  hosts:
    omni-site-hss01:
      ansible_host: 10.179.2.140
      gateway: 10.179.2.1
      host_vm_network: "v401-omni-signalling"

# IMS Internal - v402-omni-ims-internal
icscf:
  hosts:
    omni-site-icscf01:
      ansible_host: 10.179.3.55
      gateway: 10.179.3.1
      host_vm_network: "v402-omni-ims-internal"

# IMS Public - v403-omni-ims-public
pcscf:
  hosts:
    omni-site-pcscf01:
      ansible_host: 10.179.4.165
      gateway: 10.179.4.1
      host_vm_network: "v403-omni-ims-public"
```

Method 2: VLAN-Based Segmentation

Use a single physical interface with VLAN tagging to separate networks. This is suitable for smaller deployments or when physical NICs are limited.

Example:

```yaml
# All components share one bridge (ovsbr1); each segment uses a
different VLAN tag.
# applicationserver (TAS) is an IMS Internal component -> VLAN 402
/ 10.178.3.x
applicationserver:
  hosts:
    omni-site-sbc01:
      ansible_host: 10.178.3.213
      gateway: 10.178.3.1
      host_vm_network: "ovsbr1"
      vlanid: "402"

# dra and dns are Signaling components -> VLAN 401 / 10.178.2.x
dra:
  hosts:
    omni-site-dra01:
      ansible_host: 10.178.2.211
      gateway: 10.178.2.1
      host_vm_network: "ovsbr1"
      vlanid: "401"

dns:
  hosts:
    omni-site-dns01:
      ansible_host: 10.178.2.178
      gateway: 10.178.2.1
      host_vm_network: "ovsbr1"
      vlanid: "401"
```

Network Configuration:

- Configure VLANs on the physical switch
- Tag traffic appropriately at the hypervisor level
- Route between VLANs at the gateway/firewall

Example VLAN Mapping:

```
VLAN 400: 10.x.1.0/24 (Packet Core)
VLAN 401: 10.x.2.0/24 (Signaling)
VLAN 402: 10.x.3.0/24 (IMS Internal)
VLAN 403: 10.x.4.0/24 (IMS Public)
```

Working with Public IP Addresses

Overview

Many OmniCore deployments require some components to have public IP addresses for external connectivity, such as:

- **DRA** - For roaming diameter signaling with external carriers
- **Roaming SGW/PGW** - For GTP traffic from roaming partners
- **ePDG** - For WiFi calling (IPsec tunnels from UEs)
- **SMSC Gateway** - For SMPP connections to external SMS aggregators
- **P-CSCF** (in some deployments) - For direct UE SIP registration

How to Assign Public IPs

Public IPs are handled **exactly the same way as internal IPs** in your host inventory files. Simply specify the public IP address in the `ansible_host` field along with the appropriate gateway and netmask.

Example: Roaming SGW/PGW with Public IPs

```yaml
sgw:
  hosts:
    # Internal SGWs on private network
    customer-site-sgw01:
      ansible_host: 10.4.1.25
      gateway: 10.4.1.1
      host_vm_network: "v400-omni-packet-core"

    # Roaming SGWs with public IPs
    customer-site-roaming-sgw01:
      ansible_host: 203.0.113.10
      gateway: 203.0.113.9
      netmask: 255.255.255.248       # /29 subnet
      host_vm_network: "498-public-servers"
      in_pool: False
      cdrs_enabled: True

smf:  # PGWs
  hosts:
    # Roaming PGW with public IP
    customer-site-roaming-pgw01:
      ansible_host: 203.0.113.20
      gateway: 203.0.113.17
      netmask: 255.255.255.240       # /28 subnet
      host_vm_network: "497-public-services-LTE"
      in_pool: False
      ip_pools:
        - '100.64.24.0/22'
```

Example: DRA with Public IP

```yaml
dra:
  hosts:
    customer-site-dra01:
      ansible_host: 198.51.100.50
      gateway: 198.51.100.49
      netmask: 255.255.255.240       # /28 subnet
      host_vm_network: "497-public-services-LTE"
```

Example: ePDG with Public IP

```yaml
epdg:
  hosts:
    customer-site-epdg01:
      ansible_host: 198.51.100.51
      gateway: 198.51.100.49
      netmask: 255.255.255.240       # /28 subnet
      host_vm_network: "497-public-services-LTE"
```

Mixing Internal and Public IPs

It's common to have a mix of internal and public IPs within the same component group. For example:

- Internal SGWs for local sites using GTP
- Public SGWs specifically for roaming traffic from external carriers
- The same PGW-C can manage both internal and external SGWs

OmniCore's architecture handles this seamlessly - just configure each host with its appropriate IP addressing.

Introduction to Ansible Deployment at Omnitouch

Overview

Omnitouch Network Services uses Ansible as its infrastructure automation platform to deploy complete cellular network solutions (4G/5G) in a consistent, repeatable, and automated manner. This document provides an overview of how we leverage Ansible to orchestrate complex telecom deployments.

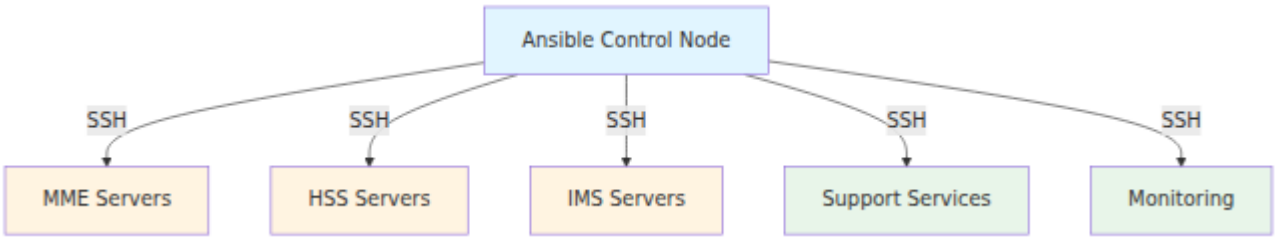
What is Ansible?

Ansible is an open-source automation tool that allows you to:

- Configure systems
- Deploy software
- Orchestrate complex workflows
- Manage infrastructure as code

Ansible uses a declarative approach - you describe the **desired state** of your systems, and Ansible ensures they reach that state.

How Omnitouch Uses Ansible



Key Concepts

1. Inventory (Hosts Files)

Defines **what** systems to manage. Each customer deployment has a hosts file that describes:

- All virtual machines in the network
- Their IP addresses
- Network configuration
- Service-specific parameters

Host files are what you will be working with to define your network.

See: Hosts File Configuration

2. Roles

Defines **how** to configure each component. Roles are reusable units that contain:

- Tasks (steps to execute)
- Templates (configuration file templates)
- Handlers (actions triggered by changes)
- Variables (default configuration values)

Example roles for OmniCore components: `omnihss`, `omnisgwc`, `omnipgwc`, `omnidra`, etc

These are defined by the ONS team, while you can edit them, there's generally cleaner ways to make any tweaks you might need from within your hosts file.

3. Playbooks

Orchestrates **when** and **where** roles are applied:

```
- name: Setup Omnicore MME
  hosts: mme
  roles:
    - omnimme
```

We use these essentially as groups for the roles.

In practice, each service playbook applies a single role, and shared setup is pulled in separately. A service bundle like `services/epc.yml` uses `import_playbook` to first run `common.yml` and then each per-service playbook:

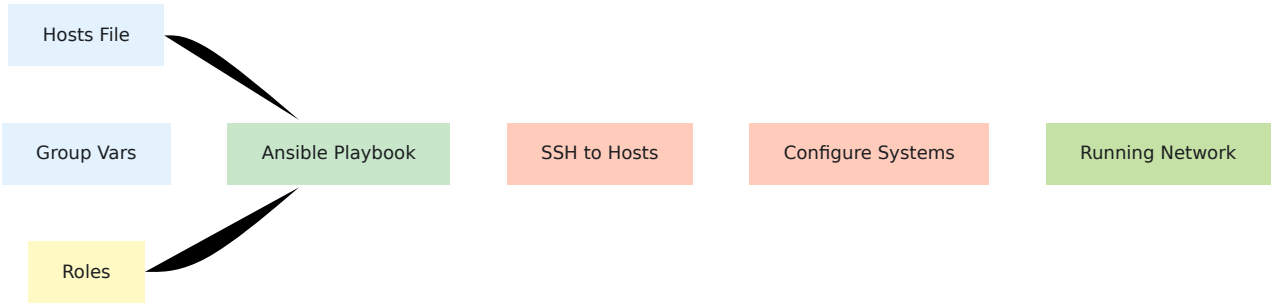
```
- name: Install Common
  import_playbook: common.yml

- name: Run MME Playbook
  import_playbook: omnimme.yml
```

4. Group Variables

Provides **customer-specific configuration** that overrides role defaults. This is where customer customization happens without modifying the base roles.

See: Group Variables and Configuration

Deployment Architecture



The Deployment Process

1. Define Infrastructure

Create a hosts file describing your network topology:

Planning Note: Before defining infrastructure, review the IP Planning Standard for guidance on network segmentation, IP address allocation, and subnet organization.

Proxmox Users: If deploying on Proxmox, see Proxmox VM/LXC Deployment for automated VM/container provisioning.

See: Hosts File Configuration and Configuration Reference

```
mme:
  hosts:
    customer-mme01:
      ansible_host: 10.10.1.15
      mme_code: 1
```

2. Customize Configuration

Set customer-specific variables in `group_vars`:

```
plmn_id:
  mcc: '001'
  mnc: '01'
customer_name_short: customer
```

#ToDo - Add link here to conifg reference for complete list

3. Run Playbooks

Deploy the network:

```
ansible-playbook -i hosts/customer/host_files/production.yml
services/epc.yml
```

4. Automated Deployment

Ansible will:

- Create/provision VMs (if using Proxmox/VMware integration)
- Configure networking
- Install software packages from APT cache
- Deploy application code
- Configure services with customer settings
- Start services
- Validate deployment

Key Components We Deploy

OmniCore (4G/5G Packet Core Platform)

- **OmniHSS** - Home Subscriber Server
- **OmniSGW** - Serving Gateway (Control plane)
- **OmniPGW** - Packet Gateway (Control plane)
- **OmniUPF** - User Plane Function
- **OmniDRA** - Diameter Routing Agent
- **OmniTWAG** - Trusted WLAN Access Gateway

See: https://docs.omnitouch.com.au/docs/repos/OmniCore

OmniCall (Voice & Messaging Platform)

- **OmniCall CSCF** - Call Session Control Function (P-CSCF, I-CSCF, S-CSCF)
- **OmniTAS** - IMS Application Server (VoLTE/VoNR services)
- **OmniMessage** - SMS Center (SMS-C)

- **OmniMessage SMPP** - SMPP protocol support
- **OmniSS7** - SS7 signaling components (STP, HLR, CAMEL)
- **VisualVoicemail** - Voicemail functionality

See: https://docs.omnitouch.com.au/docs/repos/OmniCall

OmniCharge/OmniCRM

- **CRM Platform** - Customer relationship management, self-signup, billing

See: https://docs.omnitouch.com.au/docs/repos/OmniCharge

Support Services

- **DNS** - Network DNS resolution
- **License Server** - License management
- **Monitoring** - Prometheus, Grafana

See: Deployment Architecture Overview

Package Management

We use a hybrid package distribution model:

Pre-compiled APT Packages

All Omnitouch software is distributed as Debian packages (`.deb` files):

- Built from source in our CI/CD pipeline
- Versioned and tested
- Hosted on package repositories

APT Cache System

Customers can choose between:

1. **Local APT Cache** - Mirror of required packages on-site for offline deployment
2. **Public Repository** - Direct access to Omnitouch's hosted package repository

There is also a unified `omnicore` `.deb` (built from `packaging/`) that bundles the playbooks and tooling for self-provisioning a deployment.

See: APT Cache System

License Management

All Omnitouch software components require valid licenses managed through a central license server:

- Components check license validity on startup
- Features are enabled/disabled based on license
- License server can be local or cloud-hosted

See: License Server

Benefits of This Approach

Repeatability

The same Ansible playbooks can deploy:

- Development labs
- Testing environments
- Production networks
- Customer sites

Consistency

Every deployment uses the same tested configurations, reducing human error.

Version Control

Infrastructure is defined as code in Git:

- Track all changes
- Review before deployment
- Roll back if needed

Customization Without Complexity

Customers can customize their deployment through `group_vars` without modifying core roles.

Rapid Deployment

Deploy a complete cellular network in hours instead of days or weeks.

Getting Started

Prerequisites

Before running Ansible playbooks, you need to install the required dependencies. This project uses uv to manage the Python environment, so **uv is required**.

1. Install uv

If uv isn't already installed, install it using whichever method suits your control node — see the uv installation docs. For example:

```
pipx install uv                                  # via pipx
curl -LsSf https://astral.sh/uv/install.sh | sh  # standalone
installer
```

2. Install Dependencies

From the repository root, run:

```
uv sync
```

This creates an isolated `.venv/` and installs Ansible plus all necessary Python packages (pinned in `uv.lock`) for Omnitouch deployment automation.

3. Activate the Environment

Activate the virtual environment, and keep it activated for all Ansible commands:

```
source .venv/bin/activate      # Windows: .venv\Scripts\activate
```

4. Install Ansible Collections

Install the Ansible collections the playbooks depend on (from `requirements.yml`):

```
ansible-galaxy collection install -r requirements.yml
```

5. Run Ansible Commands

With the environment activated, run Ansible commands directly:

```
ansible --version
```

Note: Deactivate the environment when finished by running `deactivate`.

Deployment Steps

1. Review the Hosts File Configuration to understand how to define your network
2. Learn about Group Variables for customization
3. Understand the APT Cache System for package management

4. Review the Deployment Architecture to see how everything fits together
5. Deploy!

Next Steps

- IP Planning Standard - **Plan your network architecture and IP allocation**
- Hosts File Configuration - Learn how to define your network topology
- APT Cache System - Understand package distribution
- License Server - Learn about license management
- Deployment Architecture Overview - See the complete picture
- Group Variables Configuration - Customize your deployment
- Utility Playbooks - Operational tools for health checks, backups, and maintenance

APT Repository & Package Distribution

Overview

The Omnitouch APT system provides package distribution for all deployments. Two types of content are served:

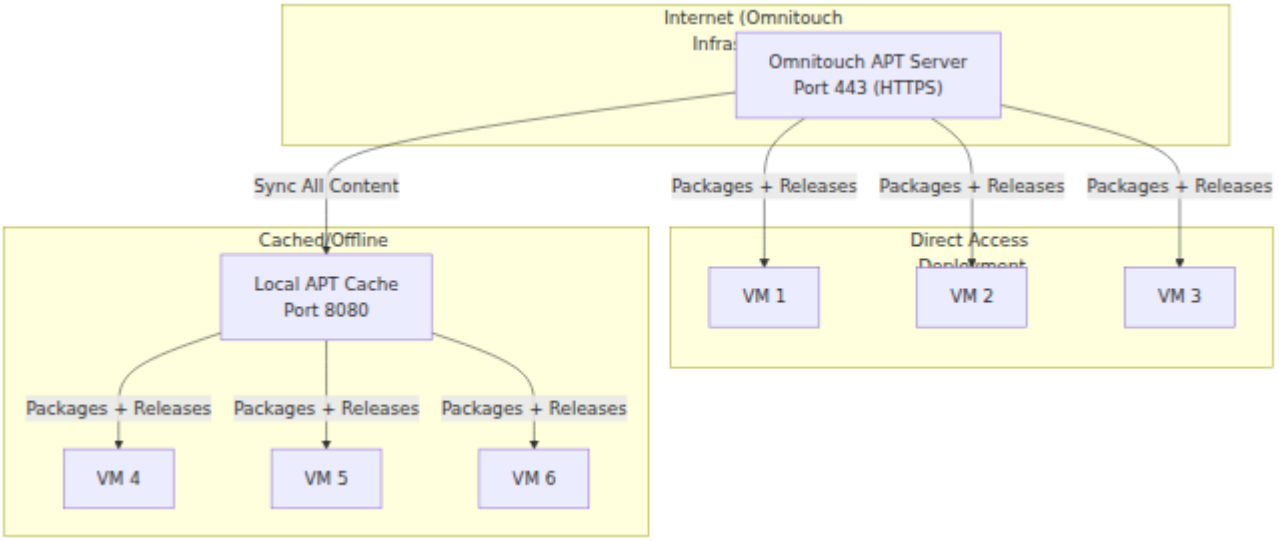
1. **APT Packages** — Debian packages installed via `apt install`
2. **Binary Releases** — Pre-built binaries downloaded directly (Prometheus exporters, agents, etc.)

The packages page on the APT server also includes a **Self-Setup (OmniCore)** tab, which provides the `omnicore` package (served at `/pool/main/o/omnicore/`) containing the full Ansible automation toolkit for customers who want to deploy and manage their network themselves.

Two deployment models are supported:

1. **Direct Access** — VMs pull packages directly from the Omnitouch APT server (e.g. `apt.<region>.omnitou.ch`)
2. **Local Cache Mirror** — A local server syncs from Omnitouch and serves packages to VMs (for offline/airgapped deployments)

Architecture



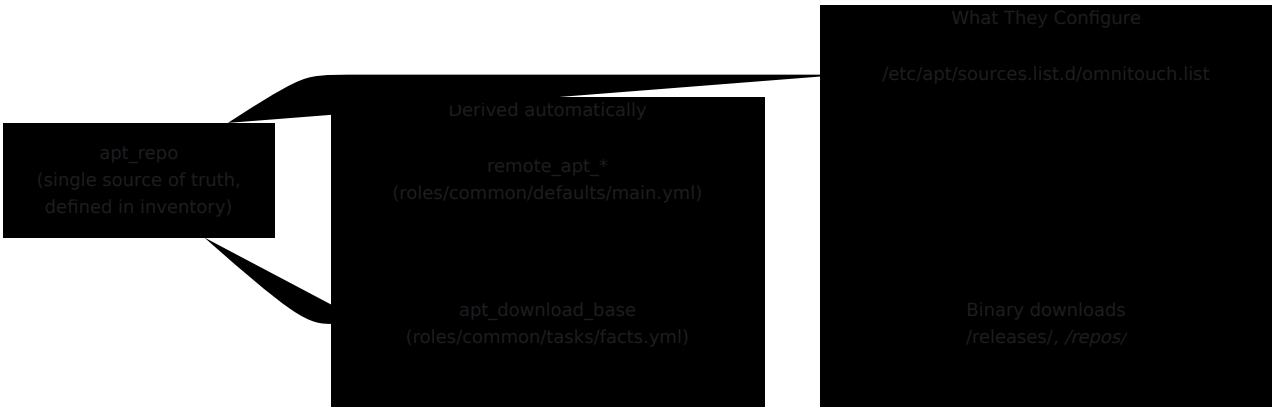
Content Served

The APT server hosts all content required for deployments:

Content Type	Description	Path
Omnitouch Packages	Custom-built `.deb` packages (omnihss, omnimme, etc.)	`/dists/<distro>/`
OmniCore Toolkit	The `omnicore` package — full Ansible automation (roles, playbooks, deps) for self-service deployment	`/pool/main/o/omnicore/`
Ubuntu Packages	Cached Ubuntu packages with all dependencies	`/<distro>/pool/main/`
GitHub Releases	Pre-built binaries (Prometheus, Grafana, Homer, etc.)	`/releases/<org>/<repo>/`
Source Tarballs	Source archives for web apps (CGrateS_UI, speedtest)	`/repos/`
Third-Party Packages	Galera, FRR, InfluxDB, KeyDB, etc.	`/releases/<vendor>/`

Configuration Variables

`apt_repo` is the single source of truth for package distribution. Define it once in your inventory and everything else is derived from it — you do not configure the binary-download URL, scheme, port, or credentials separately.


apt_repo
(single source of truth,
defined in inventory)
Derived automatically
remote_apt_*
(roles/common/defaults/main.yml)
apt_download_base
(roles/common/tasks/facts.yml)
What They Configure
/etc/apt/sources.list.d/omnitouch.list
Binary downloads
/releases/, /repos/

How the derivation works

Variable	Where set	Role
`apt_repo`	Your inventory	The one block you configure. Templated directly into the APT source line.
`remote_apt_*`	`roles/common/defaults/main.yml`	Derived from `apt_repo`. Ultimate fallback when `apt_repo` is absent: `apt.us-fl.omnitou.ch` / `443` / `https`.
`apt_download_base`	`roles/common/tasks/facts.yml`	Single computed base URL every role uses for binary downloads, so the APT source line and binary downloads always agree on host, scheme, port and credentials.

Because both derived values are built from the same `apt_repo` block, a normal inventory **only needs** **`apt_repo`** — you do not set `remote_apt_*` yourself.

> **Exception — the cache-mirror server.** A host in the `apt_cache_servers` group is the one place `remote_apt_*` is still set explicitly. Its own `apt_repo` points at the *local cache*, so `remote_apt_*` is used to describe the *upstream* Omnitouch server it syncs from (see Option 2).

Scheme and port

The scheme follows the port automatically: **port `443` → `https`, any other port → `http`**. When `apt_repo.apt_repo_port` is omitted:

- **Direct access** defaults to **`443` (HTTPS)**
- **Cache mode** defaults to **`8080` (HTTP)**

When each mode is used

Scenario	APT source line	Binary downloads (`apt_download_base`)
`use_apt_cache: true` (cache)	`http://<cache>:8080/...` — no auth	`http://<cache>:8080/...` — no auth
`use_apt_cache: false` (direct)	`https://<user>:<pass>@<server>/...`	`https://<user>:<pass>@<server>:443/...`

Option 1: Direct Access

For deployments with internet connectivity, VMs pull packages directly from the Omnitouch APT server.

Network Requirements

Source IP Whitelisting: Your public IP address must be whitelisted on the Omnitouch APT server. During setup, provide your source subnets to Omnitouch. In return, you will receive:

- **Username** and **password** for HTTP Basic Auth
- **FQDN** for the APT server (e.g. `apt.<region>.omnitou.ch`)

Firewall Requirements: Outbound access to the following Omnitouch IP ranges must be allowed:

Network	Range
IPv4	`144.79.167.0/24`
IPv4	`160.22.43.0/24`
IPv6	`2001:df3:dec0::/48`
ASN	`AS152894`

Services requiring access to Omnitouch infrastructure:

Service	Port	Protocol	Purpose
APT Server	443	TCP	Package downloads (HTTPS)
APT Server	53	TCP/UDP	DNS resolution for the APT FQDN
License Server	123	UDP	NTP time synchronization for license validation
License Server	53	TCP/UDP	DNS resolution for license validation

Ensure HTTPS (TCP/443), NTP (UDP/123), and DNS (TCP+UDP/53) traffic is allowed to the Omnitouch IP ranges.

Configuration

```yaml
all:
  vars:
    use_apt_cache: false

    # Single source of truth. Binary-download configuration
(remote_apt_*) and
    # the apt_download_base URL are derived from this
automatically — you do NOT
    # set them separately.
    apt_repo:
      apt_server: "apt.<region>.omnitou.ch"   # FQDN provided by
Omnitouch
      apt_repo_username: "your-username"
      apt_repo_password: "your-password"
      # apt_repo_port: 443                     # optional;
defaults to 443 (HTTPS)
```

Parameters

APT Package Sources (`apt_repo`)

Parameter	Type	Required	Default	Descripti
`apt_repo.apt_server`	String	Yes	-	APT server hostname IP address
`apt_repo.apt_repo_username`	String	Yes	-	HTTP Basi Auth username
`apt_repo.apt_repo_password`	String	Yes	-	HTTP Basi Auth password
`apt_repo.apt_repo_port`	Integer	No	`443`	Server por Scheme follows the port (`443` HTTPS, els HTTP).

Binary Downloads (`remote_apt_*`)

Derived automatically from `apt_repo` — do not set these in a direct-access inventory. They exist only so the cache-mirror server can point at a different upstream (see Option 2). For reference, they default as follows:

Parameter	Derived from	Fallback
`remote_apt_server`	`apt_repo.apt_server`	`apt.us-fl.omnitou.ch`
`remote_apt_port`	`apt_repo.apt_repo_port`	`443`
`remote_apt_protocol`	port (`443` → `https`)	`https`
`remote_apt_user`	`apt_repo.apt_repo_username`	`""`
`remote_apt_password`	`apt_repo.apt_repo_password`	`""`

General

Parameter	Type	Required	Default	Description
`use_apt_cache`	Boolean	Yes	-	Must be `false` for direct access

URL Patterns (Direct Access)

APT Package Sources (configured in `/etc/apt/sources.list.d/omnitouch.list`):

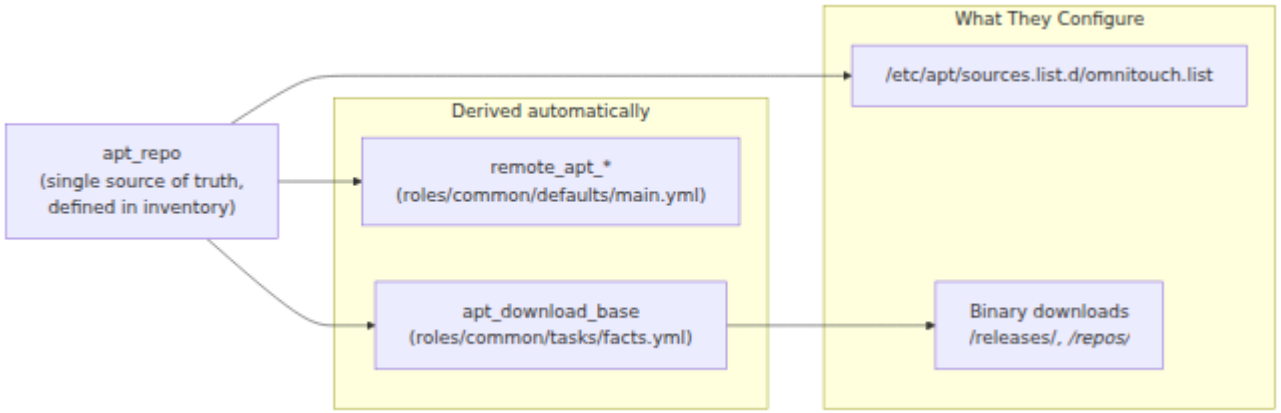
```
deb [trusted=yes] https://{username}:{password}@{apt_server}/
noble main
```

Binary Downloads (built from `apt_download_base` by Ansible `get_url` tasks):

```
https://{username}:
{password}@{apt_server}/releases/prometheus/node_exporter/node_export
1.8.1.linux-amd64.tar.gz
```

> Port `443` is omitted from the URL (it is the HTTPS default). A non-standard port is appended as `:<port>` and switches the scheme to HTTP unless it is `443`.

How It Works



VMs authenticate with HTTP Basic Auth for both APT packages and binary downloads. Ubuntu system packages are also served from the Omnitouch server (pre-cached), so VMs do not need access to Ubuntu mirrors.

Package Signing (GPG)

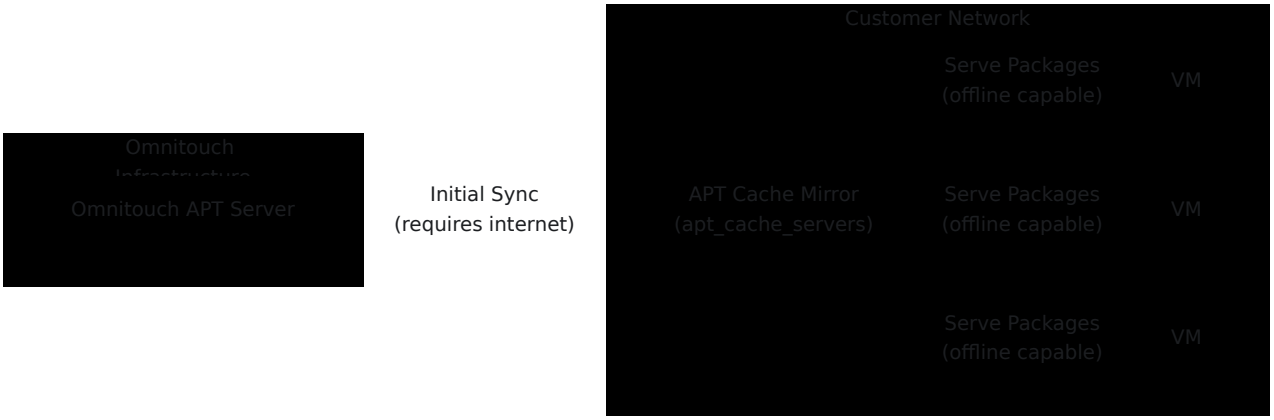
The Omnitouch APT server generates a repository signing key and serves the armored public key at `/omnitouch.gpg`. For direct-access hosts, the `common` role automatically fetches that key and installs it into `/etc/apt/trusted.gpg.d/omnitouch.gpg` so `apt-get update` verifies repository metadata without `NO_PUBKEY` warnings. This happens automatically; no manual key management is required.

> Cache-mirror mode does not use GPG verification — hosts pulling from a local cache use `[trusted=yes]` instead (see Option 2).

Option 2: Local Cache Mirror

For offline, airgapped, or bandwidth-constrained deployments, deploy a local APT cache that syncs all content from Omnitouch.

Architecture



Configuration

Define the cache server in your hosts file. It is the one place `remote_apt_*` is set explicitly — those variables describe the **upstream** Omnitouch server the cache syncs from:

```
apt_cache_servers:
  hosts:
    customer-apt-cache:
      ansible_host: 192.168.1.100
      gateway: 192.168.1.1
  vars:
    # Cache server syncs content FROM the upstream Omnitouch
repository
    remote_apt_server: "apt.<region>.omnitou.ch"
    remote_apt_port: 443
    remote_apt_protocol: "https"
    remote_apt_user: "your-username"
    remote_apt_password: "your-password"

all:
  vars:
    # use_apt_cache: true  # Auto-set when apt_cache_servers group
exists
    # apt_repo.apt_server: auto-derived to 192.168.1.100 (first
cache server)
```

How it works:

- **Cache server** (`192.168.1.100`): Uses `remote_apt_*` credentials to sync content from the upstream Omnitouch server over HTTPS.
- **All other hosts**: Automatically derive `apt_repo.apt_server: "192.168.1.100"` and pull from the cache at port `8080` without credentials.

Parameters

APT Package Sources (`apt_repo`)

Parameter	Type	Required	Default	Descri
`apt_repo.apt_server`	String	Yes	Auto-derived	Local cach IP. Automa derived fr `apt_cache` host if not specified.
`apt_repo.apt_repo_username`	String	No	-	Not requir using cach auth need
`apt_repo.apt_repo_password`	String	No	-	Not requir using cach auth need
`apt_repo.apt_repo_port`	Integer	No	`8080`	Port hosts reach the cache serv (Replaces deprecate `apt_cache`

Cache Server Sync (`remote_apt_*`)

These variables configure how the cache server syncs content from Omnitouch. This is the only scenario where `remote_apt_*` is set by hand:

Parameter	Type	Required	Default	Description
`remote_apt_server`	String	Yes	-	Upstream Omnitouch APT server to sync from
`remote_apt_port`	Integer	No	`443`	Upstream Omnitouch APT server port
`remote_apt_protocol`	String	No	`https`	Protocol for sync connection
`remote_apt_user`	String	Yes	-	Credentials for syncing from Omnitouch
`remote_apt_password`	String	Yes	-	Credentials for syncing from Omnitouch

General

Parameter	Type	Required	Default	Description
`use_apt_cache`	Boolean	No	`true`	Automatically set to `true` when `apt_cache_servers` group exists

> **Deprecated:** `apt_cache_port` is no longer used. The port hosts connect to is now set via `apt_repo.apt_repo_port` (defaults to `8080` when omitted).

URL Patterns (Cache Mode)

APT Package Sources (configured in `/etc/apt/sources.list.d/omnitouch.list`):

```
deb [trusted=yes] http://192.168.1.100:8080/noble noble main
```

Binary Downloads (built from `apt_download_base`):

```
http://192.168.1.100:8080/releases/prometheus/node_exporter/node_expo
1.8.1.linux-amd64.tar.gz
```

No credentials required for cache access—it uses `[trusted=yes]` APT configuration.

Deploying the Cache

1. **Provision the cache server** (VM or LXC container with 50+ GB disk)
2. **Run the cache setup playbook:**

   ```
   ansible-playbook -i hosts/customer/production.yml
   services/apt_cache.yml
   ```

3. **Verify the cache** by browsing to `http://192.168.1.100:8080/`

What Gets Synced

The cache mirror syncs **all content** from the Omnitouch APT server. The mechanism differs by content type:

- **APT packages** (`/dists/` + `/pool/` for both Omnitouch and Ubuntu) are synced with `curl` driven by the Debian repository index. The mirror fetches the upstream `Release` file (and short-circuits if it is unchanged), parses `Packages.gz` for the authoritative filename/SHA256/size of every package, and downloads only files that are missing or whose SHA256 does not match

the local copy. Each downloaded file is SHA256-verified before being committed.

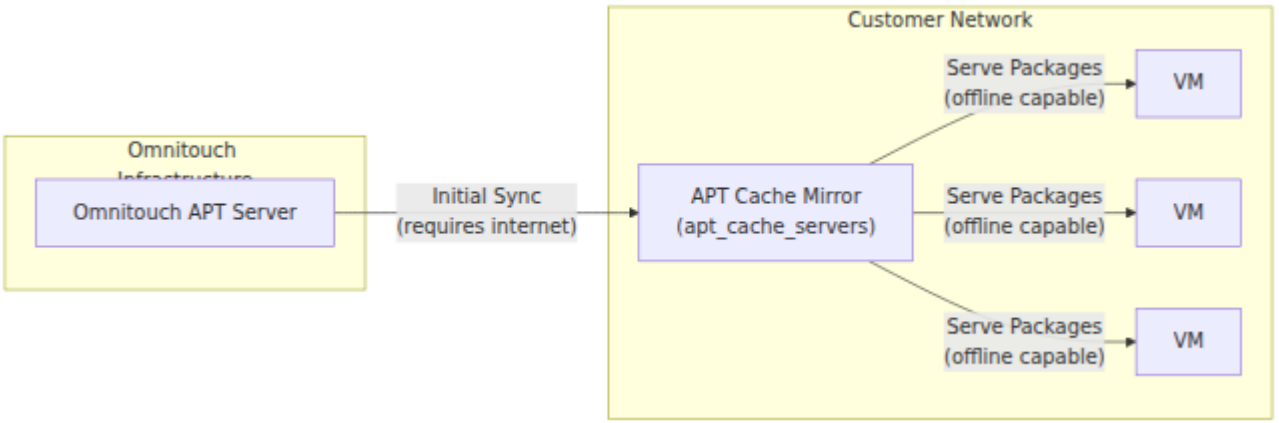
- **Binary releases and source tarballs** (`/releases/`, `/repos/`) have no Debian-style index, so they fall back to `wget --recursive --timestamping`, which only re-downloads files with a newer `Last-Modified` upstream.

Content directories synced:

Path	Content
`/dists/<distro>/`	APT repository metadata (Packages, Release files)
`/pool/main/`	Omnitouch custom .deb packages
`/<distro>/pool/main/`	Ubuntu packages and all dependencies
`/releases/`	GitHub releases (Prometheus, Grafana, Zabbix, etc.)
`/repos/`	Source tarballs (Erlang, Elixir, CGrateS_UI, etc.)

After initial sync, the cache can serve all packages without internet connectivity.

How It Works



The cache mirror authenticates with HTTP Basic Auth for all content. APT packages are synced with `curl` against the Debian repository index (`Release` + `Packages.gz`), downloading only files whose SHA256 differs from the local copy and verifying each one after download; `/releases/` and `/repos/` fall back to `wget --recursive --timestamping`. Either way, subsequent syncs only fetch new or changed files.

Automatic Configuration

When an `apt_cache_servers` group exists in your inventory, the system automatically:

1. Sets `use_apt_cache: true` for all hosts (unless explicitly overridden)
2. Derives `apt_repo.apt_server` from the first cache server's `ansible_host` IP

Minimal Configuration Example

```yaml
apt_cache_servers:
  hosts:
    apt-cache-01:
      ansible_host: 192.168.1.100
      gateway: 192.168.1.1
  vars:
    # Cache server syncs content from the upstream Omnitouch
repository
    remote_apt_server: "apt.<region>.omnitou.ch"
    remote_apt_user: "your-username"
    remote_apt_password: "your-password"
```

What happens automatically:

- All hosts (except cache server) get `use_apt_cache: true`
- All hosts (except cache server) get `apt_repo.apt_server: "192.168.1.100"`
- All hosts pull from `http://192.168.1.100:8080/` without credentials
- Cache server syncs packages from `https://your-username:your-password@apt.<region>.omnitou.ch/`

Override Automatic Behavior

To force direct access even with cache servers defined:

```yaml
all:
  vars:
    use_apt_cache: false  # Force direct access even with cache
servers defined

    apt_repo:
      apt_server: "apt.<region>.omnitou.ch"
      apt_repo_username: "user"
      apt_repo_password: "pass"
```

Configuration Summary

Scenario 1: Direct Access to APT Server (No Cache)

All hosts pull packages directly from the APT repository server. Only `apt_repo` is needed — `remote_apt_*` are derived from it.

```yaml
all:
  vars:
    use_apt_cache: false

    # Single source of truth for both APT sources and binary downloads
    apt_repo:
      apt_server: "apt.<region>.omnitou.ch"
      apt_repo_username: "user"
      apt_repo_password: "pass"
```

Result: All hosts generate `deb [trusted=yes] https://user:pass@apt.<region>.omnitou.ch/ noble main`

Scenario 2: APT Cache Server Defined in Hosts File (Automatic)

Cache server is in your inventory and will be deployed/synced by Ansible.

```yaml
apt_cache_servers:
  hosts:
    cache-server:
      ansible_host: 192.168.1.100
      gateway: 192.168.1.1
  vars:
    # Cache server syncs packages from the upstream authenticated
repository
    remote_apt_server: "apt.<region>.omnitou.ch"
    remote_apt_port: 443
    remote_apt_protocol: "https"
    remote_apt_user: "user"
    remote_apt_password: "pass"

# No configuration needed in all: vars:
# Everything auto-derived from apt_cache_servers group
```

Result:

- **Cache server**: Syncs from `https://user:pass@apt.<region>.omnitou.ch/`
- **All other hosts**: Generate `deb [trusted=yes] http://192.168.1.100:8080/noble noble main` (no credentials)

Scenario 3: Remote APT Cache NOT in Hosts File (Manual)

Cache server exists elsewhere and is already set up (not managed by your Ansible).

```yaml
all:
  vars:
    use_apt_cache: true

    # Point all hosts to the external cache server
    apt_repo:
      apt_server: "192.168.1.100"  # IP of external cache server
      apt_repo_port: 8080           # Cache typically runs on port
8080

# No apt_cache_servers group needed
# No remote_apt_* needed (cache is already set up externally)
```

Result: All hosts generate `deb [trusted=yes] http://192.168.1.100:8080/noble noble main` (no credentials)

Complete Example

Here's a complete working example showing cache server configuration with multiple application hosts:

```yaml
# APT Cache Server Group
apt_cache_servers:
  hosts:
    customer-apt-cache:
      ansible_host: 10.179.1.114
      gateway: 10.179.1.1
      host_vm_network: "vmbr0"
      num_cpus: 4
      memory_mb: 16384
      proxmoxLxcDiskSizeGb: 120
  vars:
    # Cache server syncs packages from the upstream authenticated
repository
    remote_apt_server: "apt.<region>.omnitou.ch"
    remote_apt_port: 443
    remote_apt_protocol: "https"
    remote_apt_user: "customer-username"
    remote_apt_password: "customer-secure-token"

# Application Servers
hss:
  hosts:
    customer-hss01:
      ansible_host: 10.179.2.140
      gateway: 10.179.2.1

mme:
  hosts:
    customer-mme01:
      ansible_host: 10.179.1.15
      gateway: 10.179.1.1

dns:
  hosts:
    customer-dns01:
      ansible_host: 10.179.2.177
      gateway: 10.179.2.1

# Global Configuration
all:
  vars:
    # Auto-configuration (no manual config needed):
    # - use_apt_cache: true (auto-enabled when apt_cache_servers
```

```
exists)
    # - apt_repo.apt_server: "10.179.1.114" (auto-derived from
cache server)
```

What happens during deployment:

1. **Cache server (`10.179.1.114`):**
   - Uses `remote_apt_*` from its `vars:` section
   - Downloads all packages from `https://customer-username:customer-secure-token@apt.<region>.omnitou.ch/`
   - Serves packages on port 8080 via nginx
2. **Application hosts (`customer-hss01`, `customer-mme01`, `customer-dns01`):**
   - Auto-detect `apt_cache_servers` group exists
   - Auto-set `use_apt_cache: true`
   - Auto-derive `apt_repo.apt_server: "10.179.1.114"`
   - Generate: `deb [trusted=yes] http://10.179.1.114:8080/noble noble main`
   - Pull all packages from cache server (no credentials required)

Updating the Cache

To sync new packages or updates:

```
ansible-playbook -i hosts/customer/production.yml
services/apt_cache.yml
```

This incrementally syncs all content from the Omnitouch APT server:

- New Omnitouch package versions
- New Ubuntu packages and dependencies
- New GitHub releases
- Updated source tarballs

Re-syncs are incremental: APT packages are compared against the upstream Debian index by SHA256 (and skipped entirely if the upstream `Release` file is unchanged), while `/releases/` and `/repos/` use `wget --timestamping`. Existing unchanged files are skipped, making re-sync fast.

Note: The Omnitouch APT server is the single source of truth for all packages. Omnitouch builds and publishes packages to it; running `services/apt_cache.yml` on cache mirrors then syncs whatever is currently published.

Troubleshooting

APT Update Fails with 401 Unauthorized

Symptoms:

```
Failed to fetch
https://10.179.1.115/noble/dists/noble/main/binary-amd64/Packages
401  Unauthorized
```

Possible causes:

- `apt_repo` configuration defined in `all: vars:` instead of `apt_cache_servers: vars:`
- Hosts trying to access authenticated repository directly instead of cache
- Incorrect `apt_repo_username` or `apt_repo_password`
- Source IP not whitelisted on Omnitouch APT server
- Using cache credentials for direct access or vice versa

Resolution:

1. **Check configuration scope**: Ensure `apt_repo` with credentials is defined in `apt_cache_servers: vars:`, NOT in `all: vars:`
2. **Verify cache mode**: When using cache, hosts should connect to cache server (port 8080), not the upstream repository

3. **Check generated sources**: On failing host, check `/etc/apt/sources.list.d/omnitouch.list`
   - **Correct (cache mode)**: `deb [trusted=yes] http://10.179.1.114:8080/noble noble main`
   - **Incorrect (has credentials in wrong place)**: `deb [trusted=yes] https://user:pass@10.179.1.115/noble noble main`
4. Verify credentials are correct for your deployment mode
5. Confirm your public IP is whitelisted with Omnitouch (if using direct access)

Binary Downloads Fail (Node Exporter, Zabbix, etc.)

Symptoms: Ansible playbook fails downloading files from `/releases/` path

Possible causes:

- `apt_repo` credentials incorrect (binary downloads derive their auth from `apt_repo` in direct mode)
- On a cache-mirror server, incorrect `remote_apt_user` / `remote_apt_password`
- Wrong scheme/port — remember port `443` implies HTTPS and any other port implies HTTP

Resolution:

1. In direct mode, verify `apt_repo.apt_repo_username` / `apt_repo.apt_repo_password` match those provided by Omnitouch
2. On a cache-mirror server, verify the `remote_apt_*` upstream credentials
3. Confirm the derived `apt_download_base` uses the expected host/scheme/port (check with `ansible-playbook ... -vvv` or by inspecting the failing `get_url` task)

Cache Server Cannot Sync

Symptoms: Cache server playbook fails to download packages

Possible causes:

- Cache server has no internet access
- `remote_apt_*` credentials incorrect
- Firewall blocking outbound connections to Omnitouch

Resolution:

1. Verify cache server can reach the upstream Omnitouch APT server on port 443
2. Check `remote_apt_*` credentials
3. Review firewall rules for outbound access

Related Documentation

- Hosts File Configuration — Inventory and variable configuration
- Configuration Reference — Complete parameter reference
- Deployment Architecture — Overall system architecture
- Proxmox Deployment — Deploying cache server as LXC container

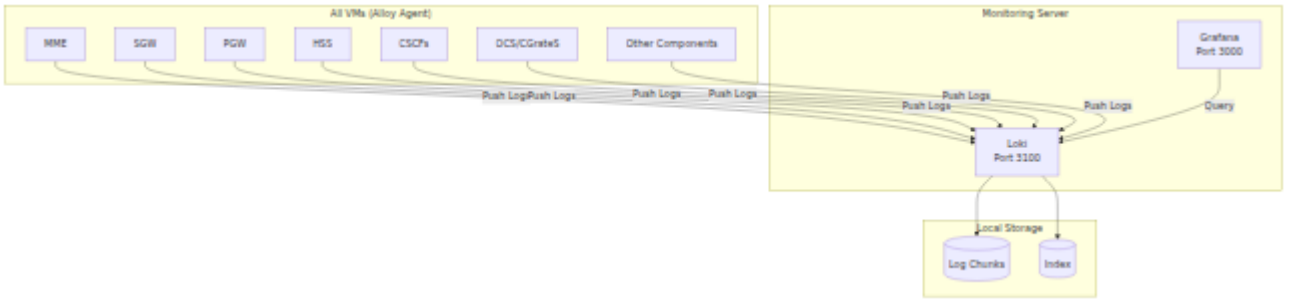
Centralized Logging

Overview

OmniCore includes a centralized logging system that collects logs from all components and makes them searchable through Grafana. The system uses:

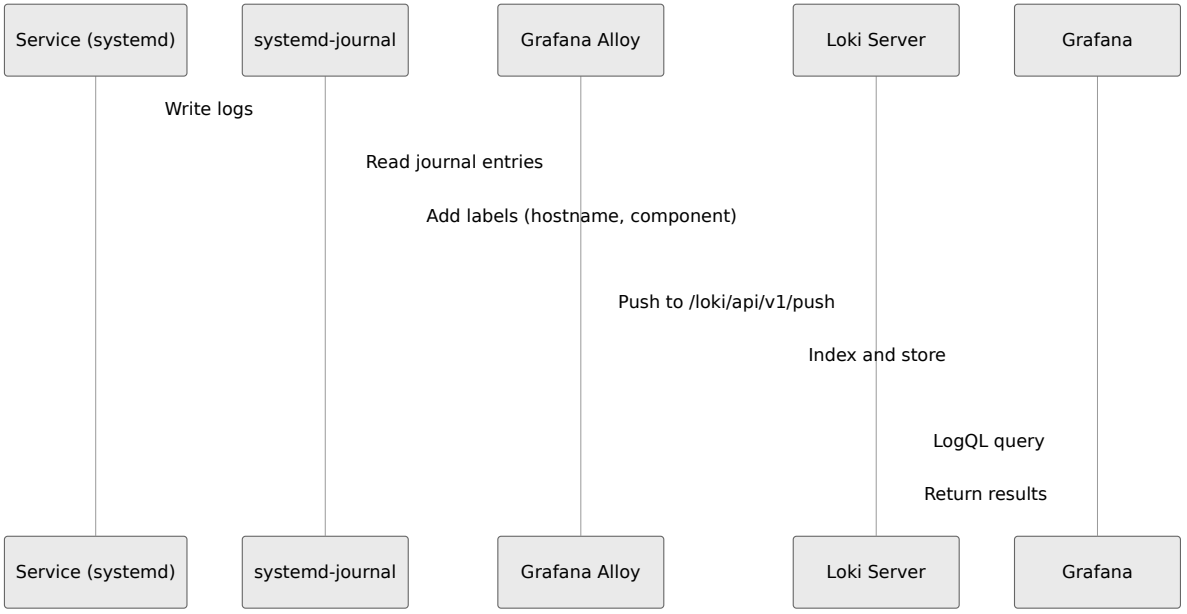
- **Grafana Alloy** — Log collection agent deployed on all VMs
- **Grafana Loki** — Log aggregation and storage server on monitoring hosts
- **Grafana Dashboards** — Pre-built dashboards for each component type

Architecture



How It Works

Log Collection Flow



Log Labels

Every log entry includes these labels for filtering and querying:

Label	Description	Example
`job`	Hostname of the source VM	`customer-mme01`
`hostname`	Same as job (for compatibility)	`customer-mme01`
`component`	Component type from inventory group	`mme`, `pcscf`, `ocs`
`unit`	systemd unit name	`omnimme.service`
`level`	Log severity	`info`, `warning`, `error`
`service`	Syslog identifier	`omnimme`

Configuration

Automatic Deployment

Logging is automatically configured when:

1. A `monitoring` group exists in your inventory
2. The common role runs on target hosts

No additional configuration is required for basic functionality.

Inventory Requirements

```
monitoring:
  hosts:
    customer-monitoring01:
      ansible_host: 10.10.2.200
      gateway: 10.10.2.1
```

When the `monitoring` group is defined:

- **Monitoring hosts**: Run Loki server (receives and stores logs)
- **All other hosts**: Run Alloy agent (collects and sends logs)

Retention Configuration

Loki is configured with time-based retention. The compactor deletes logs older than the retention period:

Setting	Default	Description
Time-based retention	72 hours (3 days)	Logs older than the retention period are deleted

To customize retention, override this variable in your inventory:

```yaml
all:
  vars:
    loki_retention_period: "168h"  # e.g. 7 days in hours
```

Alloy Buffer Configuration

Alloy buffers logs locally in a write-ahead log (WAL) when Loki is unreachable. Buffered segments are aged out to prevent indefinite disk growth:

Setting	Default	Description
WAL max segment age	1 hour	Buffered log segments older than 1 hour are dropped

When Loki is unreachable for longer than the segment age, the oldest buffered logs are dropped.

Grafana Dashboards

Pre-built dashboards are automatically provisioned for each component type.

Available Dashboards

Dashboard	Location	Description
CSCF Logs	Logs → CSCF Logs	P-CSCF, S-CSCF, I-CSCF (OmniCSCF) logs
MME Logs	Logs → MME Logs	MME attach/detach events and errors
SGW Logs	Logs → SGW Logs	SGW session and bearer events
PGW Logs	Logs → PGW Logs	PGW PDN and session events
HSS Logs	Logs → HSS Logs	HSS Diameter messages and auth events
OmniMessage Logs	Logs → OmniMessage Logs	SMS delivery and SMPP events
OCS/CGrateS Logs	Logs → OCS/CGrateS Logs	Charging events and JSONRPC traffic
CGrateS RPC Calls	Logs → CGrateS RPC Calls	Search and correlate RPC requests/responses with latency

Dashboard Features

Each dashboard includes:

- **Search box** — Free-text search across all logs

- **Log rate graphs** — Volume over time by host and severity
- **Component sections** — Grouped views (e.g., P-CSCF, S-CSCF, I-CSCF)
- **Error filtering** — Pre-filtered views for errors and failures
- **Live tail** — Real-time log streaming (10-second refresh)

Using the Dashboards

1. Navigate to Grafana (typically `http://<monitoring-host>:3000`)
2. Go to **Dashboards** → **Logs** → select component
3. Use the **Search** variable to filter logs
4. Adjust time range as needed

Querying Logs

LogQL Basics

Loki uses LogQL for querying. Basic syntax:

```
{label="value"} |= "search string"
```

Common Queries

All logs from a specific host:

```
{hostname="customer-mme01"}
```

All MME component logs:

```
{component="mme"}
```

Search for errors across all CSCFs:

```
{component=~"pcscf|scscf|icscf"} |~ "(?i)error"
```

Filter by systemd unit:

```
{unit="omnicscf.service"}
```

Combine filters:

```
{component="hss", hostname="customer-hss01"} |= "ULR" |=
"DIAMETER_SUCCESS"
```

Log Rate Queries

Log volume per component:

```
sum by (component) (rate({job=~".+"} [5m]))
```

Error rate for CSCF:

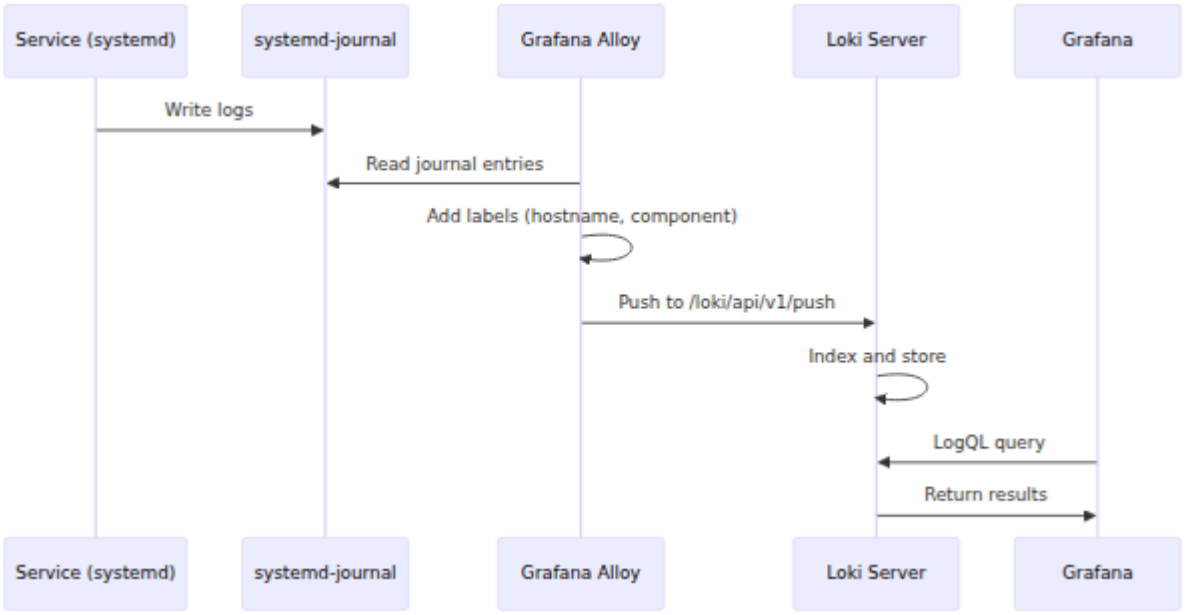
```
sum(rate({component=~"pcscf|scscf|icscf"} |~ "(?i)error" [5m]))
```

CGrateS JSONRPC Logging

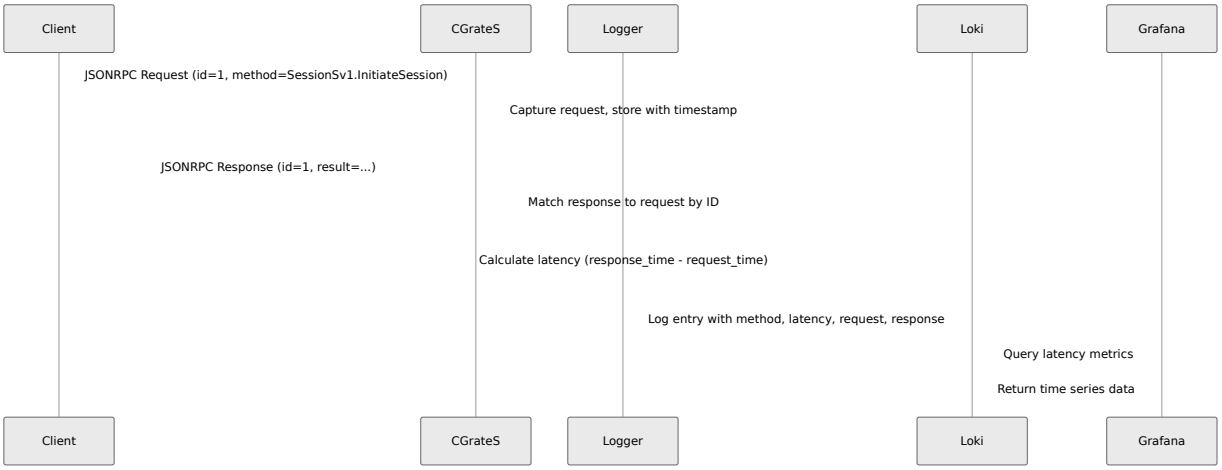
For OCS/CGrateS deployments, JSONRPC traffic is captured and logged with request/response correlation and latency tracking for performance analysis and debugging.

The JSONRPC logger is defined in the `cgrates` role and its log file (`/var/log/cgrates/jsonrpc.log`) is collected by Alloy on both `ocs` and `cgrates` host groups.

Architecture



Request/Response Flow



What Gets Captured

Traffic on these CGrateS ports is captured:

Port	Service Name	Protocol	Description
2012	`rpc_json`	TCP	Raw JSONRPC over TCP
2080	`http`	HTTP	HTTP API (filters out `/metrics` and `/health`)

The logger automatically filters out:

- Prometheus metrics scrapes (`GET /metrics`)
- Health check requests (`GET /health`)
- Gzip-compressed responses (binary data)

Log Format

JSONRPC logs are written to `/var/log/cgrates/jsonrpc.log`. Each log entry represents a **completed request/response pair** with latency measurement:

```
2026-03-01T10:30:45.123456 port=2012 service=rpc_json request_id=1
method=SessionSv1.InitiateSession latency_ms=2.45 status=ok error=
src=10.10.1.50:45678 dst=10.10.2.100:2012 request=
{"id":1,"method":"SessionSv1.InitiateSession",...} response=
{"id":1,"result":{...}}
```

Log Fields

Field	Description	Example
`timestamp`	ISO 8601 timestamp when response was received	`2026-03-01T10:30:45.123456`
`port`	CGrateS port the request was made to	`2012`, `2080`
`service`	Service name for the port	`rpc_json`, `http`
`request_id`	JSONRPC request ID for correlation	`1`, `42`, (empty for null)
`method`	JSONRPC method name	`SessionSv1.InitiateSession`
`latency_ms`	Round-trip latency in milliseconds	`2.45`
`status`	Result status	`ok`, `error`
`error`	Error message if status is error	`INSUFFICIENT_CREDIT`
`src`	Source IP:port (client)	`10.10.1.50:45678`
`dst`	Destination IP:port (CGrateS)	`10.10.2.100:2012`
`request`	Full JSONRPC request payload (compacted JSON)	`{"id":1,"method":"..."}`
`response`	Full JSONRPC response payload (compacted JSON)	`{"id":1,"result":{...}}`

Grafana Dashboard

The **OCS / CGrateS Logs** dashboard includes dedicated panels for JSONRPC analysis.

JSONRPC Latency by Method

A time series graph showing latency for each JSONRPC method over time. Useful for identifying:

- Slow methods that may need optimization
- Latency spikes indicating system issues
- Performance trends over time

The legend displays mean and max latency per method.

JSONRPC Traffic

A logs panel showing all JSONRPC request/response pairs with:

- Timestamp
- Method name
- Latency
- Full request and response payloads (prettified)

Method Drill-down

Filter JSONRPC traffic by a specific method using the **Method** variable at the top of the dashboard. Enter the method name (e.g., `SessionSv1.InitiateSession`) to see only traffic for that method.

CGrateS RPC Calls Dashboard

The dedicated **CGrateS RPC Calls** dashboard provides focused tools for troubleshooting API calls and correlating requests with responses.

Use Case: Finding a Failed Call

When a user reports "I tried to call 1234 and it failed":

1. Open **Dashboards → Logs → CGrateS RPC Calls**
2. In the **Search** field, enter the phone number or account ID: `1234`
3. Set **Status** to **Error** to see only failed calls
4. The **RPC Call Search** panel shows all matching calls with full request/response

Each log entry shows the complete request payload (what was sent) and response payload (what CGrateS returned), making it easy to identify the exact error.

Dashboard Variables

Variable	Purpose	Example
Search	Free-text search in request/response payloads	`1234`, `Account123`, `INSUFFICIENT_CREDIT`
Method	Filter by RPC method name	`SessionSv1.InitiateSession`, `CDRsV1.ProcessEvent`
Status	Filter by result status	All, Success, Error

Dashboard Panels

Summary Stats (top row):

- **Total RPC Calls** — Count of all calls in the time range
- **Errors** — Count of failed calls (color-coded: green=0, yellow=1-9, red=10+)
- **Avg Latency** — Average round-trip time (color-coded thresholds)
- **Max Latency** — Highest latency call

RPC Latency by Method: Time series showing latency for each method. Hover to see specific values.

RPC Call Search: Main log panel showing matched calls. Click any entry to expand and see full request/response JSON.

Failed RPC Calls: Pre-filtered view showing only `status=error` calls.

Method Breakdown: Pie charts showing call distribution and error distribution by method.

Querying JSONRPC Logs

Basic Queries

All JSONRPC traffic:

```
{job="cgrates-jsonrpc"}
```

Filter by method:

```
{job="cgrates-jsonrpc"} |= "method=SessionSv1.InitiateSession"
```

Search request/response payloads:

```
{job="cgrates-jsonrpc"} |= "Account\":\"12345"
```

Errors only:

```
{job="cgrates-jsonrpc"} |= "status=error"
```

Latency Analysis

Extract latency as metric (for graphs):

```
max_over_time(
  {job="cgrates-jsonrpc"}
  |= "method="
  | regexp "method=(?P<method>[^ ]+) latency_ms=(?P<latency>[0-
9.]+)"
  | __error__=""
  | unwrap latency [$__auto]
) by (method)
```

Find slow requests (latency > 100ms):

```
{job="cgrates-jsonrpc"}
| regexp "latency_ms=(?P<latency>[0-9.]+)"
| latency > 100
```

Method-Specific Queries

CDR processing:

```
{job="cgrates-jsonrpc"} |= "method=CDRsV1"
```

Session management:

```
{job="cgrates-jsonrpc"} |~ "method=SessionSv1"
```

Account operations:

```
{job="cgrates-jsonrpc"} |~ "method=ApierV"
```

Service Management

The JSONRPC logger runs as a systemd service on OCS/CGrateS hosts.

Check service status:

```
systemctl status cgrates-jsonrpc-logger
```

View service logs:

```
journalctl -u cgrates-jsonrpc-logger -f
```

Restart the service:

```
systemctl restart cgrates-jsonrpc-logger
```

Troubleshooting

No JSONRPC Logs Appearing

Symptoms: JSONRPC Traffic panel shows no data

Possible causes:

- Logger service not running
- ngrep not installed
- Network interface mismatch
- Alloy not collecting the log file

Resolution:

1. Check logger service status:
   ```
   systemctl status cgrates-jsonrpc-logger
   journalctl -u cgrates-jsonrpc-logger -n 50
   ```
2. Verify ngrep is installed:
   ```
   which ngrep
   ```
3. Check log file is being written:

```
tail -f /var/log/cgrates/jsonrpc.log
```

4. Verify Alloy is collecting the file:

```
journalctl -u alloy | grep jsonrpc
```

Latency Graph Shows No Data

Symptoms: JSONRPC Latency by Method panel shows "No data"

Possible causes:

- No JSONRPC traffic in the selected time range
- Log format mismatch (regex not matching)

Resolution:

1. Verify logs exist for the time range:

```
{job="cgrates-jsonrpc"} |= "method="
```

2. Check that `latency_ms` field is present in logs
3. Expand the time range

High Pending Request Count

Symptoms: Debug logs show many stored requests but few logged completions

Possible causes:

- Requests without responses (client disconnected)
- Response captured before request (packet ordering)
- Request ID mismatch between request and response

Resolution:

1. The logger automatically cleans up pending requests older than 60 seconds
2. Check for network issues between client and CGrateS

3. Verify CGrateS is responding to requests

File Logs

In addition to systemd journal logs, Alloy collects specific log files:

FreeSWITCH Logs (Application Servers)

Path	Job Label
`/var/log/freeswitch/freeswitch.log`	`freeswitch`
`/var/log/omnitas-freeswitch.log`	`freeswitch-structured`

Nginx Logs (OmniCRM)

Path	Job Label	Type Label
`/var/log/nginx/access.log`	`nginx`	`access`
`/var/log/nginx/error.log`	`nginx`	`error`

CGrateS JSONRPC (OCS)

Path	Job Label
`/var/log/cgrates/jsonrpc.log`	`cgrates-jsonrpc`

Troubleshooting

Logs Not Appearing in Grafana

Symptoms: No logs visible in Loki dashboards

Possible causes:

- Alloy service not running on source host
- Loki service not running on monitoring host
- Network connectivity between hosts blocked
- Alloy cannot reach Loki on port 3100

Resolution:

1. Check Alloy status on source host:

   ```
   systemctl status alloy
   journalctl -u alloy -f
   ```

2. Check Loki status on monitoring host:

   ```
   systemctl status loki
   journalctl -u loki -f
   ```

3. Test connectivity from source to monitoring:

   ```
   curl http://<monitoring-ip>:3100/ready
   ```

4. Verify firewall allows TCP 3100 from all hosts to monitoring

Alloy High Disk Usage

Symptoms: `/var/lib/alloy` consuming significant disk space

Possible causes:

- Loki unreachable for extended period

- WAL buffer filling up

Resolution:

1. Check Loki connectivity (see above)
2. If Loki is down, logs buffer locally in the WAL
3. Once Loki is reachable, buffered logs are sent automatically
4. Buffered segments older than 1 hour are dropped (by design)

Loki Storage Full

Symptoms: Loki stops accepting logs, disk full on monitoring server

Possible causes:

- Log volume exceeds available disk before retention deletes old logs
- Retention compaction not running

Resolution:

1. Check Loki storage usage:

   ```
   du -sh /var/lib/loki/
   ```

2. Verify compactor is running (check Loki logs)
3. Manually trigger compaction if needed by restarting Loki
4. Consider increasing disk allocation or reducing retention period

Missing Component Labels

Symptoms: Logs appear but `component` label is `infrastructure` instead of expected value

Possible causes:

- Host not in expected inventory group
- Alloy config not regenerated after inventory change

Resolution:

1. Verify host is in correct inventory group
2. Re-run Ansible to regenerate Alloy config:

```
ansible-playbook -i hosts/customer/hosts.yml services/all.yml -
-limit <hostname>
```

3. Restart Alloy:

```
systemctl restart alloy
```

Service Ports

Service	Port	Protocol	Description
Loki	3100	HTTP	Log ingestion and query API
Loki	9096	gRPC	Internal gRPC (not used externally)
Alloy	12345	HTTP	Alloy metrics and UI
Grafana	3000	HTTP	Dashboard access

Related Documentation

- Monitoring & Observability — Grafana, Prometheus, dashboards, and alerting
- Deployment Architecture — Overall system architecture
- Hosts File Configuration — Inventory configuration
- Configuration Reference — Complete parameter reference

Circuit-Switched Core (CS)

The legacy circuit-switched domain — 2G/3G voice and SMS — is deployed by the `services/cs.yml` playbook. It provisions the radio-facing BSC, the OmniMSC softswitch, and the SS7/SIGTRAN signalling stack.

Overview

`cs.yml` runs three roles, each against its own inventory group:

Component	Role	Inventory group	Packages	Des
BSC	`circuit_switched/bsc`	`bsc`	`osmo-bsc`, `osmo-mgw`	Bas Con Mec Gat (Osr
MSC	`circuit_switched/omnimsc`	`msc`	`omnimsc`	Mob Swit Cen soft
SS7	`omniss7`	`ss7`	`omniss7`	SS7 stac HLR CAM test

```
- name: Setup MSC
  hosts: msc
  roles:
    - {role: circuit_switched/omnimsc, become: yes}
```

BSC (`circuit_switched/bsc`)

Installs the Osmocom **OsmoBSC** and **OsmoMGW** from the Omnitouch APT server and configures both under `/etc/osmocom/`.

- Templates `osmo-bsc.cfg.j2` → `/etc/osmocom/osmo-bsc.cfg` and `osmo-mgw.cfg.j2` → `/etc/osmocom/osmo-mgw.cfg`.
- Installs a systemd override for `osmo-mgw` that **disables real-time CPU scheduling** (`CPUSchedulingPolicy`/`Priority`/`AmbientCapabilities` cleared) — RT scheduling is not supported inside VMs.
- Handlers restart each service and then **verify it reaches `active`** (6 retries, 5s apart) before continuing.

MSC (`circuit_switched/omnimsc`)

Installs the `omnimsc` package and configures it much like a 5GC NF:

- Resolves `group_vars_omnicore` and derives `license_server_api_urls` from the `license_server` group.
- Renders the built-in `runtime.exs.j2` to `/etc/omnimsc/`, or — if `omnimsc_template_config` is set — a custom template from `group_vars/`.
- Ensures a CDR output directory (`omnimsc_cdr_output_dir`, default `/var/lib/omnimsc/cdr`).
- Generates a self-signed TLS certificate for the web UI / API.
- Uses the versioned-config + `runtime.exs` symlink pattern, restarting on change.

SS7 (`omniss7`)

A single role that installs the `omniss7` package and renders **one config template per SS7 role**, selected by the `ss7_role` host variable:

`ss7_role`	Template	Function
`stp`	`stp.j2`	Signal Transfer Point (SS7 routing)
`hlr`	`hlr.j2`	Home Location Register
`smsc`	`ipsmgw.j2`	SMSC / IP-SM-GW
`camel`	`camel.j2`	CAMEL service control
`tester`	`tester.j2`	Signalling test/load agent

Additional selectors:

- Setting `ss7_protocol: m2pa` renders `m2pa.j2` (an M2PA endpoint) instead of the `tester` template.
- Setting `ss7_template_config` (a path under `group_vars/`) overrides **all** built-in templates with a host-specific config.

Like the other Elixir components, `omniss7` uses the versioned-config + `runtime.exs` symlink pattern and mirrors the symlink into `/opt/omniss7/releases/*`.

Related Documentation

- Service Playbooks - All service playbooks and the deployment hierarchy
- 5G Core - The 5G Standalone core and the shared `5gc_nf` role
- Group Variables Configuration - Custom config templates (e.g. `ss7_template_config`, `omnimsc_template_config`)
- **OmniCore Configuration**: https://docs.omnitouch.com.au/docs/repos/OmniCore - What goes in the

component config files

Configuration Reference

Overview

This document provides a comprehensive reference for configuring OmniCore deployments through hosts files. Configuration is primarily defined in host inventory files with minimal group_vars overrides needed for modern deployments.

For product-specific documentation, see:

- **OmniCore**: https://docs.omnitouch.com.au/docs/repos/OmniCore
- **OmniCall**: https://docs.omnitouch.com.au/docs/repos/OmniCall
- **OmniCharge**: https://docs.omnitouch.com.au/docs/repos/OmniCharge

Configuration Approach

Modern OmniCore deployments use a simplified configuration model:

Key Principle: Most configuration is defined directly in the hosts file, and role defaults handle many settings. Some components, however, are configured exclusively from `group_vars` and have no role default — notably OmniCRM (see below). In those cases a missing value fails the run loudly rather than falling back to a default.

> **Security note**: Several role defaults are insecure placeholders intended for lab use only and **must** be overridden for any real deployment. Examples include database passwords (e.g. `sql_pass: password`, `omnitouch2024`) and APT repository credentials (`omni`/`omni`). Always set real, unique secrets in your inventory or group_vars before deploying.

Network Planning

Before configuring hosts, review the IP Planning Standard for guidance on:

- Network segmentation strategies
- IP address allocation
- Subnet organization
- Public IP handling

Common Host Parameters

#ToDo - Just say to check hosts-file-configuration.md for this

Service-Specific Flags

```
cdrs_enabled: True                  # Enable CDR generation
in_pool: False                      # Exclude from load balancing
pool
populate_crm: False                 # Populate CRM with initial data
```

Global Variables (all:vars)

The `all:vars` section contains deployment-wide settings. Modern deployments use minimal global variables, with many settings covered by role defaults. Note the exceptions below (OmniCRM, Grafana SMTP) where configuration must be supplied explicitly because no role default exists.

Essential Global Variables

Authentication & Access

```
ansible_connection: ssh
ansible_user: root
ansible_password: password
ansible_become_password: password
```

Alternative: Use SSH keys instead of passwords:

```
ansible_ssh_private_key_file: '/path/to/key.pem'
```

Customer Identity

```
customer_name_short: omnitouch
customer_legal_name: "Customer Lab"
site_name: Customer-Site
region: AU
TZ: Australia/Melbourne
```

PLMN Configuration

```
plmn_id:
  mcc: '001'              # Mobile Country Code (3 digits)
  mnc: '01'              # Mobile Network Code (2-3 digits)
  mnc_longform: '001'     # Zero-padded MNC (always 3 digits)

diameter_realm: epc.mnc{{ plmn_id.mnc_longform }}.mcc{{
plmn_id.mcc }}.3gppnetwork.org
```

Purpose: Uniquely identifies your mobile network. Used for Diameter realm construction.

Network Names

```
network_name_short: Omni
network_name_long: Omnitouch
tac_list: [10100,100]           # Default TAC list (can override
per-MME)
```

Displayed: Network names shown on UE devices in Settings > Mobile Network.

DNS Configuration

```
manage_resolv_conf: True          # Set to False to prevent Ansible
from managing /etc/resolv.conf
```

Note: When `manage_resolv_conf` is set to `False`, Ansible will not overwrite `/etc/resolv.conf` on that host. This is useful for hosts that require custom DNS configuration or are managed by external systems. Can be set per-host in the inventory or globally in `all:vars`.

APT Repository Configuration

`apt_repo` **is the single source of truth.** Define it once and the binary-download variables (`remote_apt_*`) and the `apt_download_base` URL are derived from it automatically — you do not set them separately in a normal inventory.

Automatic Defaults: When an `apt_cache_servers` group is defined with hosts:

- `use_apt_cache` automatically defaults to `True` (unless explicitly set to `False`)
- `apt_repo.apt_server` automatically defaults to the first cache server's IP

```yaml
use_apt_cache: False              # Direct repo access (True =
local APT cache)

apt_repo:
  apt_server: "apt.<region>.omnitou.ch"  # FQDN provided by
Omnitouch (or cache IP)
  apt_repo_username: "your-username"     # Required for direct
access
  apt_repo_password: "your-password"     # Required for direct
access
  # apt_repo_port: 443                   # Optional; defaults to
443 (HTTPS).
                                         # Scheme follows the
port: 443 -> https, else http.
```

The scheme and port default to **HTTPS / 443** for direct access and **HTTP / 8080** in cache mode. Only a cache-mirror server (`apt_cache_servers` group) sets `remote_apt_*` by hand, to describe the upstream Omnitouch server it syncs from.

See: APT Cache System

License Server

```yaml
# Optional: auto-derived from the `license_server` inventory group
when unset.
# Each host in that group becomes https://<ansible_host>:8443/api.
license_server_api_urls: ["https://10.10.2.150:8443/api"]
license_enforced: false          # Effective default is false
(roles/omnihss/defaults/main.yml)
```

Note: If a `license_server` group exists in the inventory, `license_server_api_urls` is derived automatically from its members; you only need to set it manually to override that behaviour.

See: License Server

MME Settings

The MME selects S-GW and P-GW peers using a configurable method, set per gateway under `omnimme`. The method can be a single value or split by `home`/`roaming` traffic. Valid values are `dns_lookup` (resolve peers via DNS) and `random_peer` (pick from configured peers).

```
omnimme:
  sgw:
    selection_method: random_peer        # single value, or...
    # selection_method:
    #   home: random_peer
    #   roaming: dns_lookup
  pgw:
    selection_method:
      home: random_peer
      roaming: dns_lookup
```

AMF Settings

```
# tac_list also drives the AMF's supported_ta_list.  Every TAC
here is
# advertised to connecting gNBs during NG Setup.  If unset the AMF
defaults
# to TAC 1 only, which rejects any gNB not broadcasting TAC 1.
tac_list: [1, 3]

# Disable NAS ciphering (force NEA0).  Defaults to true because
omniamf
# >=20260727 negotiates NEA2 but cannot decode ciphered uplink NAS
(Security
# Mode Complete and subsequent messages), causing every 5G
registration to
# fail.  Set to false once the vendor confirms uplink NEA2
deciphering works
# against real UEs.  NAS integrity (NIA2) is unaffected.
amf_force_null_ciphering: true
```

SAEGW Settings

```
mtu: 1400                          # Maximum Transmission Unit
```

OmniHSS Settings

OmniHSS builds its Diameter peer list automatically from the inventory — the `dra`, `mme`, `pgwc`, `scscf`, `icscf`, `pcscf` and `applicationserver` groups (which of those are used depends on `packet_core_dra_support` / `ims_dra_support`). `exclude_diameter_peers` drops individual hosts from that generated list, for peers that are in the inventory but should not be Diameter peers of the HSS.

```
omnihss:
  exclude_diameter_peers:          # optional; defaults to []
(nothing excluded)
    - customer-as01
    - customer-as02
```

- Entries are matched against **`inventory_hostname`**, not the Diameter FQDN or IP.
- The exclusion applies to every auto-discovered group, so one list covers a host regardless of which group it sits in.
- Names must exist in the inventory. The role asserts this before deploying, so a typo fails the run loudly rather than silently leaving the peer in `/etc/omnihss/runtime.exs`.
- Set it wherever the rest of the `omnihss` dict is set (per host, or under a group's `vars:`). Because Ansible replaces rather than merges dicts, it must live in the same `omnihss` block as the other keys.
- This affects the Diameter peer list only. The S-CSCF SIP peer list under `config :hss, ims:` is unrelated and is not filtered by this key.

> The DRA has its own, separate mechanism: setting `dra_peer_exclude: True` on a host removes it from the OmniDRA peer list. The two are independent — excluding a host from the HSS does not exclude it from the DRA, or vice versa.

OmniCRM Settings

OmniCRM configuration is sourced **exclusively** from the customer's `group_vars` (under `hosts/`); no defaults are baked into the role. `crm_env_override` is **required** — it names the per-site `.env` template, and a missing value fails the run loudly rather than shipping another site's secrets.

The remaining files are optional (skipped if unset). All paths are resolved relative to the customer's group_vars directory.

```
crm_env_override: my-site.env    # REQUIRED: per-site UI .env
template (no default)
crm_config.yaml                  # API config (crm_config.yaml in
group_vars; copied to the API)
cell_sites: cellSites.txt        # optional: cell site data for
the UI
message_templates: constants.js  # optional: cell broadcast
message templates
site_data: site_data.json        # optional: cell broadcast geo
bounds
```

Grafana SMTP Settings

Used by the monitoring role for Grafana alert email. `grafana_smtp_user` and `grafana_smtp_password` are **required** (no default — they must be set); `grafana_smtp_host` defaults to `in-v3.mailjet.com:587`. See `roles/monitoring/templates/grafana/grafana.ini`.

```
grafana_smtp_user: "smtp-user"           # REQUIRED, no default
grafana_smtp_password: "smtp-password"   # REQUIRED, no default
grafana_smtp_host: "in-v3.mailjet.com:587"  # optional, this is
the default
```

IMS Settings

```
ims_dra_support: False           # Route IMS through DRA
enable_homer: False              # Enable Homer SIP capture
```

RAN Monitor Configuration

```yaml
use_nokia_monitor: True
use_casa_monitor: True
install_influxdb: True

influxdb_user: monitor
influxdb_password: "secure-password"
influxdb_organisation_name: omnitouch
influxdb_nokia_bucket_name: nokia-monitor
influxdb_casa_bucket_name: casa-monitor
influxdb_operator_token: "generated-token"
influxdb_url: http://127.0.0.1:8086

enable_pm_collection: False
enable_alarm_collection: False
enable_location_collection: False
enable_ran_status_collection: True
enable_nokia_rectifier_collection: False
collection_interval_in_seconds: 120

ran_monitor:
  sql:
    user: ran_monitor
    password: "secure-password"
    database_host: 127.0.0.1
    database_name: ran_monitor
  influxdb:
    address: 10.10.2.135
    port: 8086
  nokia:
    airscales:
    - address: 10.7.15.66
      name: site-Lab-Airscale
      port: 8080
      web_password: nemuuser
      web_username: Nemuadmin
```

Firewall Configuration

```yaml
firewall:
  allowed_ssh_subnets:
    - '10.0.1.0/24'
    - '10.0.0.0/24'
  allowed_ue_voice_subnets:
    - '10.0.1.0/24'
  allowed_carrier_voice_subnets:
    - '10.0.1.0/24'
  allowed_signaling_subnets:
    - '10.0.1.0/24'
```

Roaming DNS Servers

```yaml
roaming_dns_servers:
  wildcard: ['10.0.99.1']
  # Carrier-specific DNS (PLMN-based)
  123456:                          # Example Carrier 1
    - '10.10.2.197'
  654321:                          # Example Carrier 2
    - '10.10.0.4'
```

Local Users (SSH Keys)

```yaml
local_users:
  usera:
    name: Example User A
    public_key: "ssh-rsa AAAAB3Nza..."
  userb:
    name: Example User B
    public_key: "ssh-ed25519 AAAAC3..."
```

Hypervisor Configuration

Proxmox

A site deploys as either VMs or LXC containers, selected per `proxmoxServers` entry via `deployment_type` (default `vm`). All entries in a site must agree; mixing fails validation. See Proxmox Deployment for the full walkthrough.

VM site

```
proxmoxServers:
  customer-prxmx01:
    # deployment_type omitted → defaults to "vm"
    proxmoxServerAddress: 10.10.0.100
    proxmoxServerPort: 8006
    proxmoxApiTokenName: AnsibleToken
    proxmoxApiTokenSecret: "token-secret"
    proxmoxNodeName: pve01
    proxmoxTemplateName: ubuntu-24.04-cloud-init-template
    proxmoxTemplateId: 9000
    proxmoxTemplateUser: omnitouch        # optional cloud-init
username; defaults to the first local_users key
    proxmoxTemplatePassword: omnitouch    # optional cloud-init
password; defaults to the cloud-init username
    storage: SSD_RAID0            # optional, default VM storage
```

LXC site

```yaml
proxmox_lxc_nameserver: "1.1.1.1"  # optional, injected into LXCs
at create

proxmoxServers:
  customer-prxmx01:
    deployment_type: lxc
    proxmoxServerAddress: 10.10.0.100
    proxmoxServerPort: 8006
    proxmoxApiTokenName: AnsibleToken
    proxmoxApiTokenSecret: "token-secret"
    proxmoxNodeName: pve01
    proxmoxLxcOsTemplate: "local:vztmpl/ubuntu-24.04-
standard_24.04-2_amd64.tar.zst"
    proxmoxLxcDefaultStorage: SSD_RAID0   # optional, rootfs
fallback
```

Group-level overrides (both types)

```yaml
dns:
  vars:
    proxmox_interface: vmbr0              # required: bridge
    gateway: 10.10.0.1                    # required
    netmask: 255.255.255.0                # required
    vlanid: 100                           # optional
    proxmoxLxcCores: 2                    # optional (LXC only)
    proxmoxLxcMemoryMb: 4096              # optional (LXC only)
    proxmoxLxcDiskSizeGb: 30              # optional (LXC only)
    proxmoxLxcRootFsStorageName: SSD_RAID0 # optional (LXC only)
    host_vm_network: vmbr1                # optional bridge
override
```

VMware vCenter

```yaml
vcenter_ip: "vcenter.example.com"
vcenter_username: "administrator@vsphere.local"
vcenter_password: "password"
vcenter_datacenter: "DC1"
vcenter_vm_template: ubuntu-24.04-model
vcenter_vm_disk_size: 50
vcenter_folder: "Omnicore"
host_vm_network: "Management"

vhosts:
  "10.0.0.23":
    vcenter_cluster_ip: 10.0.0.23
    vcenter_datastore: "datastore1 (3)"

netmask: 255.255.255.0
```

Related Documentation

- IP Planning Standard - Network architecture and IP allocation guidelines
- Hosts File Configuration - How to structure hosts files
- Group Variables Configuration - When and how to use group_vars
- Netplan Configuration - Secondary IPs and multi-NIC setup
- Deployment Architecture - How components interact
- APT Cache System - Package management
- License Server - License configuration

Product Documentation

For detailed operational guides and advanced configuration:

- **OmniCore Components**: https://docs.omnitouch.com.au/docs/repos/OmniCore

- **OmniCall Components**: https://docs.omnitouch.com.au/docs/repos/OmniCall
- **OmniCharge/OmniCRM**: https://docs.omnitouch.com.au/docs/repos/OmniCharge

Deployment Architecture Overview

Overview

This document provides a complete view of how Omnitouch Network Services' cellular network software is deployed using Ansible, showing how all components fit together to create a working 4G/5G network.

See the IP Planning Standard for detailed component placement, IP address assignment guidelines, and public IP handling.

Complete Deployment Example

0. Infrastructure Provisioning (Optional)

For Proxmox deployments, provision VMs/LXCs before configuration:

```
# Deploy VMs and/or LXC containers on Proxmox (per-host
deployment_type decides which)
ansible-playbook -i hosts/Customer/hosts.yml
util_playbooks/proxmox.yml
```

See: Proxmox VM/LXC Deployment

1. Infrastructure Definition (Hosts File)

```yaml
# Define what to deploy and where
mme:
  hosts:
    customer-mme01:
      ansible_host: 10.10.1.15

hss:
  hosts:
    customer-hss01:
      ansible_host: 10.10.2.140

# ... all other components
```

See: Hosts File Configuration

2. Customization (group_vars)

The `group_vars` folder is where we can store any config overrides needed at a host, site or network level.

For example you'd have a folder with your OmniMessage SMSc config, the SIP trunks your TAS connects to would live here, all your Diameter Routing logic, etc, etc.

OmniCRM is configured here too — its settings are `group_vars`-only, and its GitHub source is cloned on the Ansible controller (`delegate_to: localhost`) using the controller's own credentials before being synced to the target, so no credentials ever land on the target host.

See: Group Variables Configuration

3. Package Distribution (APT Cache)

```yaml
# Configure where to get packages
apt_repo:
  apt_server: "10.254.10.223"  # Cache server IP or direct repo
server
use_apt_cache: false  # true = use local cache, false = direct
repo access
```

See: APT Cache System

4. License Configuration

```yaml
# Point components to license server
license_server_api_urls: ["https://10.10.2.150:8443/api"]
license_enforced: true
```

See: License Server

5. Execute Deployment

Individual components can be deployed by running `services/twag.yml` for example, but the `services/all.yml` will handle everything, and you can use `--limit=myhost` or `--limit=mmee,sgw`, etc, to limit the hosts we're working on.

```
# Deploy complete network
ansible-playbook -i hosts/customer/host_files/production.yml
services/all.yml

# Or deploy specific components
ansible-playbook -i hosts/customer/host_files/production.yml
services/epc.yml
ansible-playbook -i hosts/customer/host_files/production.yml
services/ims.yml
```

Deployment Models

There are two ways to get the OmniCore toolkit onto a controller and run it.

Model A: Clone the repo on the controller

The traditional model: clone this repository onto your Ansible controller, keep your inventory under `hosts/`, and run the playbooks directly from the checkout. This is the model assumed by the deployment example above.

Model B: Self-provisioning `omnicore` .deb

OmniCore is also packaged as a self-contained `omnicore` Debian package, so a controller can install the toolkit from apt rather than cloning the repo.

- **What it installs.** The package (`name: omnicore`, see `packaging/nfpm.yaml`) ships the Ansible roles, playbooks, *vendored* collections, a bundled Python interpreter (`/opt/omnicore/python`) and an offline wheelhouse to `/opt/omnicore`. The post-install step (`packaging/postinstall.sh`) builds a self-contained venv (`/opt/omnicore/venv`) from the bundled interpreter + wheelhouse and symlinks the bundled `ansible*` onto `PATH` via `/usr/local/bin`, so the controller runs the exact `uv.lock` versions of Ansible and its Python libraries — no distro Ansible, no system Python, no network. Only genuine system tools (`git`, `rsync`, `openssh-client`) are pulled in by apt. The package is **amd64** and targets **Ubuntu 22.04+** (glibc ≥ 2.35).
- **How it's built.** `packaging/build.sh` stages the repo into `/opt/omnicore` (excluding `hosts/`, `.git/`, virtualenvs and build files), vendors the Ansible collections from `requirements.yml`, bundles the standalone interpreter + a wheelhouse of the locked Python deps, runs a fail-closed secret scan, and then builds the `.deb` with nfpm.
- **How it's published.** `.github/workflows/build-deb.yml` runs `build.sh` on every push to `main` and publishes the `.deb` (plus a `_metadata.json` sidecar) as a GitHub Release. The apt server's `discover_github_debs.py` auto-discovers that release asset across the Omnitouch org and adds it to aptly — no apt-role changes are needed.

Customer flow:

```
# 1. Install the toolkit from apt
apt-get install omnicore

# 2. Supply your own inventory
#    (the package does not ship one — it lives outside
/opt/omnicore)
vi /etc/omnicore/inventory

# 3. Run playbooks from the installed toolkit
cd /opt/omnicore
ansible-playbook -i /etc/omnicore/inventory services/all.yml
```

Because the inventory lives at `/etc/omnicore/inventory` (separate from the installed `/opt/omnicore` toolkit), upgrading the package never touches your infrastructure definition or customisation.

Related Documentation

- Introduction to Ansible Deployment - Getting started
- Service Playbooks - **Playbook reference and hierarchy**
- Hosts File Configuration - Defining infrastructure
- IP Planning Standard - **Network architecture and IP allocation**
- Group Variables Configuration - Customization
- APT Cache System - Package management
- License Server - License management
- Monitoring & Observability - Grafana, Prometheus, alerting, and dashboards
- Centralized Logging - Loki and Alloy log collection

Product Documentation

For detailed information on configuring each component:

- **OmniCore** (4G/5G Packet Core): https://docs.omnitouch.com.au/docs/repos/OmniCore
    - OmniHSS, OmniSGW, OmniPGW, OmniUPF, OmniDRA, OmniTWAG
- **OmniCall** (Voice & Messaging): https://docs.omnitouch.com.au/docs/repos/OmniCall
    - OmniTAS, OmniCall CSCF, OmniMessage, OmniSS7, VisualVoicemail
- **OmniCharge/OmniCRM** (Billing): https://docs.omnitouch.com.au/docs/repos/OmniCharge
- **Main Documentation**: https://docs.omnitouch.com.au/

Per-Role Firewall

The per-role firewall gives every host an `iptables` and `ip6tables` ruleset that is derived from a single declarative model of the network's traffic, rather than hand-written per-host rules. One file describes every permitted flow in the core (who may reach which role, on which protocol and port, and why). From that model the platform produces two things that always agree with each other:

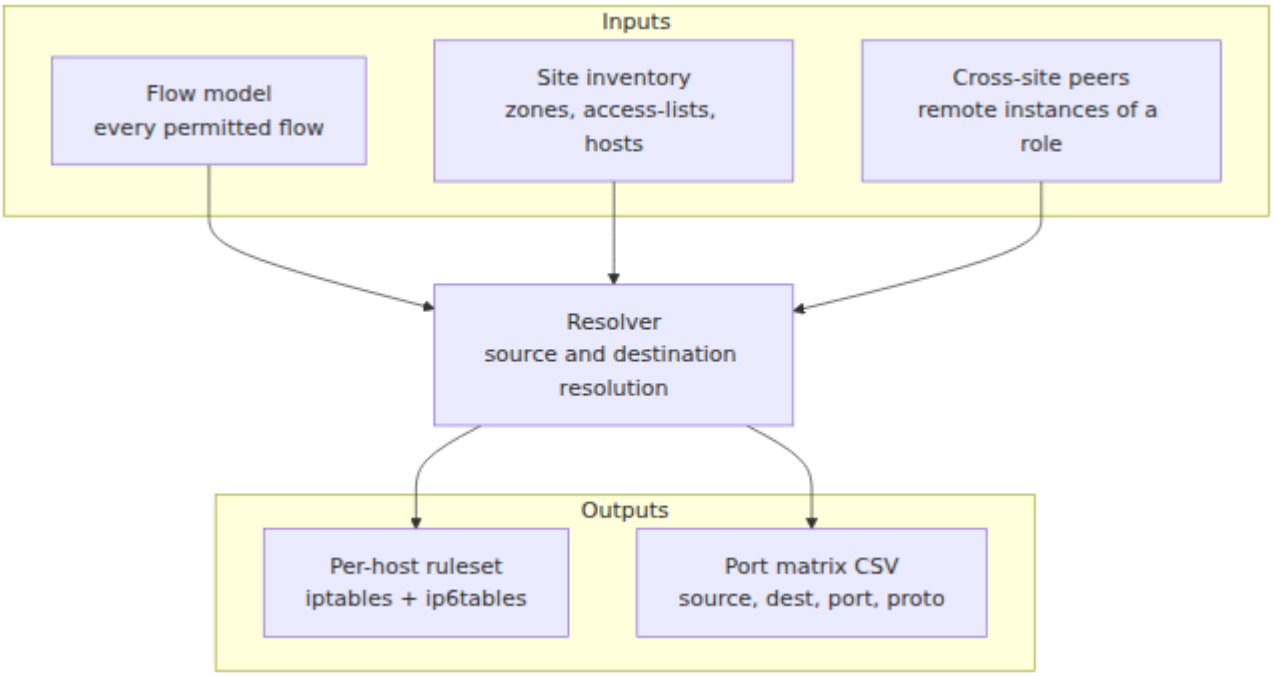
- the **enforced ruleset** applied on each host, and
- a **communications / port matrix** (CSV) for review and for handing to an upstream firewall.

Rules run in one of two modes: **monitor** (log every flow, drop nothing) or **enforce** (default-deny). Blocking is deliberate and fail-safe, and every apply is protected by an on-host automatic rollback so a mistaken ruleset cannot lock operators out.

Table of Contents

Architecture Overview

A single model plus each site's own inventory feed one resolver. The resolver produces both the on-host rules and the port matrix, so the documentation can never describe something different from what is actually enforced.



Dropped and audited traffic is logged through the standard observability pipeline:

The Flow Model

The flow model is the single source of truth for **who may reach what**. It holds two parts: a set of structural defaults (which carry no site addressing), and the list of flows. A flow says nothing about port numbers directly - it names a **service** exposed by the destination role (see Service Catalog), and the port is looked up from that role's declaration:

```yaml
- id: '13'                          # stable label, appears in each
rule's comment
  dst: hss                          # inventory group the rule is
applied to
  service: diameter                 # named service exposed by the
hss role (-> 3868/tcp+sctp)
  purpose: 'Diameter'               # description, shown in the
matrix
  sources: [{ zone: internal }]     # who may reach it (list of
source tokens)
```

Parameter	Type	Required	Default	Description
`id`	String	Yes	-	Stable identifier for the flow. Appears in each generated rule's comment as `ans:<dst>:<id>` so a live rule can be traced back to the model.
`dst`	String	Yes	-	The inventory group the rule lands on, or the pseudo-destinations `all` (every host) and `five_g_sbi` (the 5G network-function groups). A flow whose destination group is not deployed at a site produces no rules and no matrix rows there.
`service`	String	Yes*	-	A named service declared by the destination role (see Service Catalog). The port(s) are looked up from the role's declaration, so the flow never restates a port number.
`proto_ports`	List	Yes*	-	The literal alternative to `service`, for flows not tied to a single role (for example `all`-host flows). Each entry is `PORT/proto` (`tcp`, `udp`, `sctp`; port may be a `LO-HI` range) or a bare protocol (`icmp`, `esp`, `proto 89`).

Parameter	Type	Required	Default	Description
`purpose`	String	Yes	-	Human description of the flow. Shown in the `service` column of the port matrix.
`sources`	List	Yes	-	The permitted sources, as a list of source tokens. See Source Types.

* A flow provides **either** `service` **or** `proto_ports`, not both.

Source Types

Each entry in a flow's `sources` list is a single-key token that is resolved against the site's configuration and inventory. A token that resolves to nothing (for example an empty access-list) contributes no rule: it is never emitted as a placeholder.

Token	Resolves to
`{ zone: <name> }`	The CIDRs of that plane zone (`firewall.zones.<name>`).
`{ list: <name> }`	An operator access-list, for example `allowed_oam_subnets`.
`{ group: <name> }`	The inventory group's host addresses, plus that role's cross-site peers. See Cross-Site Peers.
`{ ue_pools: true }`	The site's UE / subscriber pools (`firewall.ue_pools`).
`{ dra_peers: true }`	Every address configured on the DRA host's own Diameter peer list.
`{ roaming: true }`	Roaming (IR.21 / GRX-IPX) ranges. Currently any source: see Operational Cautions.
`{ literal: "..." }`	A free-text source that is not derived from inventory.

Service Catalog

Port numbers are **owned by the role that opens them**, not by the central model. Each role that exposes ports ships a `firewall_services.yml` in its role directory, keyed by the inventory group(s) it serves (a role such as `cscf` serves several groups, so it declares each). This keeps the port definition next to the code that opens it:

```yaml
# roles/omnihss/firewall_services.yml
firewall_services:
  hss:
    diameter: [3868/tcp, 3868/sctp]   # S6a / Cx-Dx / Sh
    api_oam:  8080/tcp                # HSS API (OAM)
    api_nf:   8443/tcp                # HSS API (queried by other
NFs)
    postgres: 5432/tcp                # logical replication
between HSS peers
```

The resolver globs every role's `firewall_services.yml` into one catalog. A flow's `service:` is looked up in the catalog for that flow's `dst` group, so the model expresses only the **relationship** (who may reach `hss`'s `diameter` service) while the **port** lives with the role. Add or move a listener in a role and update its `firewall_services.yml`; nothing in the central model changes.

This split has a safety net that a central-only model lacks: a **drift check** compares what roles declare against what hosts are actually listening on.

```
ansible-playbook -i hosts/<Deployment>/host_files/<Site>.yml \
    util_playbooks/firewall_services_audit.yml
```

For each host it reports:

- **undeclared** - a live listener on a port no role declared (either add it to the role's `firewall_services.yml`, or it is an unexpected listener), and
- **not listening** - a declared service that is not actually up.

Hosts whose groups declare no services are skipped, so the check is useful while roles are migrated onto the catalog incrementally rather than all at once.

Conditional listeners. Some ports are opened only when another role is deployed: the MME opens its LCS ports (SBc-AP `29168/sctp`, SLs `9082/sctp`) only when an `omnilcs` host is in the inventory. A static declaration cannot express that condition, so such ports are noted in the role's `firewall_services.yml` but not declared. On a site that does run the dependent role, the drift check reports them as undeclared, which is the correct prompt to review and permit them.

Site Configuration

Each site declares a `firewall` block under `all.vars` in its inventory. There are no address fallbacks: a site must describe its own network. Anything a site does not declare is simply empty, and no other environment's addressing is ever inherited.

The example below is fully populated so that every field shows a representative value. The addresses use documentation ranges and a coherent per-plane layout (each plane is its own subnet within the `internal` supernet), which is what makes the zones useful: they let one flow permit, say, only the IMS plane rather than the whole core.

```
firewall:
  allowed_oam_subnets:           ['10.8.0.0/24']       #
management source (SSH, UI, scrape, mgmt APIs)
  allowed_ran_subnets:           ['10.20.0.0/16']      # RAN
signalling source (S1AP, NGAP, GTP-U, Abis)
  allowed_ue_voice_subnets:      ['100.64.0.0/16']     # external
UE SIP into the P-CSCF
  allowed_carrier_voice_subnets: ['203.0.113.0/24']    # carrier
interconnect SIP
  zones:                                               # plane
zones: each plane's real subnet(s)
    internal:    ['10.8.0.0/16']                       #   trusted
core supernet
    mobile_core: ['10.8.1.0/24']                       #   mobile
core NFs (EPC + 5GC + CS core)
    core_svc:    ['10.8.2.0/24']                       #   core
services plane (HSS, DRA, OCS, OAM)
    ims:       ['10.8.3.0/24']                         #   IMS
plane (CSCF, TAS)
    ue_svc:    ['10.8.4.0/24']                         #   UE-
reachable plane (P-CSCF, DNS, ePDG)
    core_v6:   ['2001:db8:0:1::/64']                   #   IPv6
addresses of core NFs (not UEs)
  ue_pools: ['100.64.0.0/16', '2001:db8:64::/48']      # UE /
subscriber IP ranges (v4 and v6)
  extra_rules:                                         # manual
accept rules (see below)
    - { proto: tcp, dport: '8443', source: '203.0.113.10/32',
comment: 'partner API' }
```

Only `allowed_oam_subnets` is mandatory. Every other list may be left empty (`[]`) on a site that lacks that function: a site with no RAN sets `allowed_ran_subnets: []`, a site with no IPv6 core sets `core_v6: []`, and a site with no UE pools sets `ue_pools: []`. An empty list simply produces no rules for the flows that use it (see the `Required` column below).

Parameter	Type	Required	Default	Descrip
`allowed_oam_subnets`	List	Yes	-	Manager source subnets. Governs SSH, the OmniWel Prometh scrape, a the manager APIs. If empty th generato stops wit error, because empty va would lea manager with no permitte source a cause a lockout under enforce.
`allowed_ran_subnets`	List	No	`[]`	Source subnets RAN signallin (S1AP, NGAP, G U, Abis). be empt

Parameter	Type	Required	Default	Descrip
				a site wit no RAN.
`allowed_ue_voice_subnets`	List	No	`[]`	Source subnets external SIP into t P-CSCF.
`allowed_carrier_voice_subnets`	List	No	`[]`	Source subnets carrier interconn SIP.
`zones`	Map	Yes	-	Plane zo mapping each pla name to real subnet(s See Zone Paramet
`ue_pools`	List	No	`[]`	UE / subscrib ranges. Empty if site has none.
`extra_rules`	List	No	`[]`	Manual accept ru applied every ho See Man

Parameter	Type	Required	Default	Descrip
				Rule Paramet

Zone Parameters

Each zone maps a plane name to a list of CIDRs. Set only the planes the site uses; leave the rest empty.

Zone	Description
`internal`	Trusted core supernet. Broadest internal signalling source.
`mobile_core`	Mobile core network functions across generations: EPC (MME, S/P-GW, UPF), 5GC (AMF, SMF, and the SBI mesh), and CS core (MSC, SS7/SIGTRAN).
`ims`	IMS plane (I-CSCF, S-CSCF, TAS).
`ue_svc`	UE-reachable services plane (P-CSCF, DNS, ePDG).
`core_svc`	Core services plane (HSS, DRA, OCS, OAM).
`core_v6`	IPv6 core service addresses. Empty if the site has no IPv6 core.

Manual Rule Parameters

`extra_rules` (global, all hosts) and the per-host `firewall_extra_rules` append raw accept rules for cases the model does not cover. A `source` containing `:` is applied on IPv6.

Parameter	Type	Required	Default	Description
`proto`	String	No	-	Protocol, for example `tcp`, `udp`, `sctp`.
`dport`	String	No	-	Destination port or range.
`source`	String	No	-	Source CIDR. A `:` in the value applies the rule on IPv6.
`comment`	String	No	`override`	Label recorded in the rule comment.

Operating Modes

The apply mode is controlled by `firewall_mode`.

Mode	Behaviour
`monitor`	Every accept rule is installed, the `INPUT` policy stays `ACCEPT`, and fall-through traffic is logged (as `FW-AUDIT`) then allowed. Nothing is dropped. Use this to validate the model against real traffic before enforcing.
`enforce`	Fall-through traffic is logged (as `FW-DROP`) then dropped, and the `INPUT` and `FORWARD` policies are set to `DROP`.

Blocking is opt-in and fail-safe: only the exact value `enforce` ever drops traffic. Any other value, including an unset variable or a typo, is treated as monitor, so a host cannot begin dropping traffic by accident. This holds at three levels: the platform default is `monitor`, sites declare `firewall_mode: monitor` in their inventory, and the generator treats anything that is not `enforce` as monitor. Enforcing requires a deliberate change.

```yaml
all:
  vars:
    firewall_mode: monitor   # change to 'enforce' only after the
audit log confirms coverage
```

`firewall_mode` may be set per site under `all.vars`, or overridden per group or per host through normal variable precedence.

Cross-Site Peers

Roles that federate across sites (CSCF, TAS, OmniMessage, HSS replication, OCS) list their remote peers in the site's peers file (for example `prod_master_peers.yml`). Those remote addresses are folded into `{ group: <role> }` resolution, so a flow that permits a role automatically covers both same-site and cross-site instances of it under one identical policy. The same resolution is used for the rules and for the matrix, so the peers appear in both.

Address Resolution

A host is resolved to **every address it is declared with**, not just its primary. That means `ansible_host` plus every `secondary_ips.<name>.ip_address` in the inventory. Datapath and signalling nodes bind services to secondary NICs: the UPF answers N3 GTP-U on its `n3` address, and the STP receives and originates SIGTRAN on its public address. Both the rules and the matrix cover all of those addresses, as sources and as destinations, so a peer that allowlists a node by the address handed to it is allowlisting the address the traffic actually uses.

A host that belongs to **several inventory groups** receives the **union** of every group's flows and declared services. A converged node exposes everything for all the roles it runs, and its expected listener set (used by the drift check) is the union across its groups.

Because resolution uses every declared address rather than tracking which service binds which NIC, the matrix can list a service on an address it does not actually bind (for example N3 GTP-U shown on the UPF's N6 address). This is

deliberate over-approximation: a complete but slightly broad matrix is safe to hand to an upstream firewall, whereas omitting a real datapath or SIGTRAN address silently breaks traffic. If per-address precision is needed later, a service can name the NIC it binds; until then, resolution stays fact-based and complete.

Communications / Port Matrix

The port matrix is the flow model resolved against a site's inventory, flattened to one row per source, destination host, and single port/protocol. It is the form to review or to hand to an upstream firewall. Generate it with:

```
ansible-playbook -i hosts/<Deployment>/host_files/<Site>.yml \
    util_playbooks/render_port_matrix.yml
```

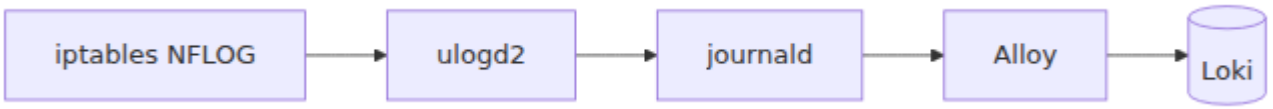
This writes `generated/firewall-port-matrix-exploded.csv` (per-site, not committed). The columns are:

Column	Description
`id`	The flow identifier.
`service`	The flow's purpose.
`source`	A resolved source CIDR or address.
`destination_role`	The destination inventory group.
`destination_host`	The destination host name.
`destination_ip`	The destination host address.
`port`	The destination port or range (empty for protocol-only entries such as ICMP or ESP).
`protocol`	The protocol.

A row appears only when its destination is deployed at the site and its source resolves to a real address. Undeployed roles and unresolved sources produce no rows.

Lockout Protection

Applying rules is guarded by an automatic rollback, controlled by `firewall_rollback_guard` (default on) and `firewall_rollback_timeout`:



1. The live ruleset is snapshotted.
2. A timer is armed on the host that will restore the snapshot. It survives the management session being cut.
3. The new rules are applied.
4. The controller re-establishes a fresh connection and runs a trivial check. If the new rules locked it out, this fails and the run stops for that host.
5. If connectivity is confirmed the timer is disarmed. If it is not, the timer fires and restores the working ruleset with no operator action.

Parameter	Type	Required	Default	Descripti
`firewall_rollback_guard`	Boolean	No	`true`	Enables th snapshot, timer, and reconnect check arou every appl
`firewall_rollback_timeout`	Integer	No	`120`	Seconds th timer wait before automatica restoring t snapshot.

IPv6 and Dual-Stack

Address family is inferred per CIDR: a `:` in the value means IPv6, and the rule is applied on `ip6tables`. A zone or list containing only IPv4 CIDRs therefore produces no IPv6 rules for that source. Sites with no IPv6 core leave the IPv6 zones and pools empty (for example `core_v6: []`). Under enforce the IPv6 `INPUT` policy is set to `DROP`, so validate IPv6 traffic in monitor mode before enforcing a host whose only legitimate IPv6 ingress arrives through an IPv4-only source.

Logging

Audited and dropped traffic is logged with NFLOG and carried by the standard pipeline to Loki. Three prefixes are used:

Prefix	Meaning
`FW-FLOW` / `FW-FLOW6`	A new connection seen in monitor mode. Builds the live flow map of traffic actually in use.
`FW-AUDIT` / `FW-AUDIT6`	Fall-through traffic in monitor mode: would be dropped under enforce, but is allowed here. Review these before enforcing.
`FW-DROP` / `FW-DROP6`	Fall-through traffic actually dropped under enforce.

Example Loki queries:

```
# Traffic that would be blocked once this host is enforced
{job="firewall"} |= "FW-AUDIT"

# Traffic being dropped under enforce, by host
sum by (host) (count_over_time({job="firewall"} |= "FW-DROP"
[5m]))
```

Operational Cautions

- **Roaming and literal sources are currently any-source.** Flows whose source is `roaming` or `literal` are emitted with no source restriction (`0.0.0.0/0`). Treat them as open until the roaming ranges are resolved to real CIDRs.
- **ICMP is accepted from any source.** The generated ruleset allows ICMP unconditionally, which is broader than the management scope shown for the ping flow.
- **Do not enforce a user-plane host without forwarding rules.** Under enforce the `FORWARD` policy is set to `DROP`, and no forwarding rules are generated. Enforcing on a UPF, PGW, or SGW without adding forwarding rules would break the user plane. Validate first.
- **The management list is coarse.** `allowed_oam_subnets` governs SSH, the UI, scrape, and the management APIs together. A site that scrapes from a different network than it administers from needs an `extra_rules` entry to separate them.

Group Variables Configuration

Overview

The `group_vars` directory is where you store custom configuration files that override default templates.

This is where your customer-specific configurations live - SIP trunks, Diameter routing rules, SMS routing logic, dialplans, and any other customizations where you don't want the default config - It lives in `group_vars`.

Location: `hosts/{Customer}/group_vars/`

Roles resolve this directory at runtime as `group_vars_omnicore` - the `group_vars/` directory sitting adjacent to the inventory directory (`{{ inventory_dir.rsplit('/', 1)[0] }}/group_vars/`). Wherever a file path below references `{{ group_vars_omnicore }}`, it points at this directory.

How It Works

Ansible roles have default configuration templates. To customize for a specific deployment, place your custom files in `group_vars` and reference them in your hosts file.

```
Role Default Template → group_vars Override (if specified) →
Deployed Config
```

Example 1: Custom Configuration Template (OmniMessage)

Some components accept custom Jinja2 configuration templates.

File Structure

```
hosts/Customer/
└── group_vars/
    └── smsc_controller.exs        # Your custom config template
```

Reference in Hosts File

```yaml
omnimessage:
  hosts:
    customer-smsc-controller01:
      ansible_host: 10.10.3.219
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      smsc_template_config: smsc_controller.exs   # Reference your
template filename in group_vars
```

What happens:

1. Ansible finds `smsc_template_config: smsc_controller.exs`
2. Looks in `hosts/Customer/group_vars/smsc_controller.exs`
3. Templates it with Jinja2 (can use `{{ inventory_hostname }}`, `{{ plmn_id.mcc }}`, etc.)
4. Deploys to `/etc/omnimessage/runtime.exs`
5. Restarts the service

Without `smsc_template_config`, the default template from the role is used.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCall

---

Example 2: Configuration File Collections (OmniTAS Gateways & Dialplans)

Some components use directories of configuration files.

File Structure

```
hosts/Customer/
└── group_vars/
    ├── gateways_prod/              # SIP gateway configs
    │   ├── gateway_carrier1.xml
    │   ├── gateway_carrier2.xml
    │   └── gateway_emergency.xml
    ├── gateways_lab/               # Lab gateways
    │   └── gateway_test.xml
    ├── dialplan/                   # Call routing rules (default)
    │   ├── mo_dialplan.xml         # Mobile Originated (outgoing)
    │   ├── mt_dialplan.xml         # Mobile Terminated (incoming)
    │   └── emergency.xml
    └── dialplan_lab/               # Lab dialplans
        └── mo_dialplan.xml
```

Reference in Hosts File

```yaml
applicationserver:
  hosts:
    customer-tas01:
      ansible_host: 10.10.3.60
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      gateways_folder: "gateways_prod"   # Reference your gateway
folder to use on this host
      dialplan_folder: "dialplan"         # Optional - defaults to
"dialplan" if not set
```

What happens:

1. Ansible finds `gateways_folder: "gateways_prod"`
2. Copies all files from `hosts/Customer/group_vars/gateways_prod/` to `/etc/freeswitch/sip_profiles/`
3. Copies all files from `hosts/Customer/group_vars/dialplan/` (or the folder specified by `dialplan_folder`) to OmniTAS templates directory
4. Services load the configurations

Different environments: Use different folders per environment:

- `gateways_folder: "gateways_lab"`
- `gateways_folder: "gateways_prod"`
- `gateways_folder: "gateways_customer_specific"`
- `dialplan_folder: "dialplan_lab"`
- `dialplan_folder: "dialplan_prod"`

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCall

---

Example 3: Custom Configuration Template (OmniHSS)

The Home Subscriber Server accepts custom runtime configuration templates.

File Structure

```
hosts/Customer/
└── group_vars/
    └── hss_runtime.exs.j2        # Your custom HSS config
template
```

Reference in Hosts File

```
omnihss:
  hosts:
    customer-hss01:
      ansible_host: 10.10.3.50
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      hss_template_config: hss_runtime.exs.j2   # Reference your
template filename in group_vars
```

What happens:

1. Ansible finds `hss_template_config: hss_runtime.exs.j2`
2. Looks in `hosts/Customer/group_vars/hss_runtime.exs.j2`
3. Templates it with Jinja2 (can use `{{ inventory_hostname }}`, `{{ plmn_id.mcc }}`, etc.)
4. Deploys to `/etc/omnihss/runtime.exs`
5. Restarts the service

Without `hss_template_config`, the default template from the role is used.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCore

---

Example 4: Custom Configuration Template (OmniMME)

The Mobility Management Entity accepts custom runtime configuration templates.

File Structure

```
hosts/Customer/
└── group_vars/
    └── mme_runtime.exs.j2        # Your custom MME config
template
```

Reference in Hosts File

```yaml
omnimme:
  hosts:
    customer-mme01:
      ansible_host: 10.10.3.51
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      mme_template_config: mme_runtime.exs.j2   # Reference your
template filename in group_vars
```

What happens:

1. Ansible finds `mme_template_config: mme_runtime.exs.j2`
2. Looks in `hosts/Customer/group_vars/mme_runtime.exs.j2`
3. Templates it with Jinja2 (can use `{{ inventory_hostname }}`, `{{ plmn_id.mcc }}`, etc.)
4. Deploys to `/etc/omnimme/runtime.exs`
5. Restarts the service

Without `mme_template_config`, the default template from the role is used.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCore

---

Example 5: Required Configuration Files (OmniCRM)

OmniCRM is sourced **exclusively** from group_vars - there is no role default to fall back on. The API and UI config are deliberately not baked into the role so a missing file fails the run loudly rather than silently shipping another site's secrets.

File Structure

```
hosts/Customer/
└── group_vars/
    ├── crm_config.yaml          # API config (required)
    ├── customer_crm.env         # UI .env (required, referenced
by crm_env_override)
    ├── cellSites.txt            # Optional cell site data
    ├── constants.js             # Optional cell broadcast message
templates
    └── site_data.json           # Optional cell broadcast geo
bounds
```

Reference in Hosts File

```yaml
omnicrm:
  hosts:
    customer-crm01:
      ansible_host: 10.10.3.70
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      crm_env_override: customer_crm.env   # REQUIRED - UI .env
filename in group_vars
      cell_sites: cellSites.txt            # Optional
      message_templates: constants.js      # Optional
      site_data: site_data.json            # Optional
```

What happens:

1. The API role templates `{{ group_vars_omnicore }}/crm_config.yaml` to `/etc/omnicrm/OmniCRM-API/crm_config.yaml`. This file is **required** - there is no role default.
2. The UI role templates `{{ group_vars_omnicore }}/{{ crm_env_override }}` to `/etc/omnicrm/OmniCRM-UI/.env`. `crm_env_override` is **required** - if it is unset the run fails.
3. If `cell_sites`, `message_templates`, or `site_data` are set, their files are copied into the UI (cell site list, cell broadcast templates, and geo bounds respectively).

Important: Unlike the template-config examples above, there is no role fallback. Both `crm_config.yaml` and `crm_env_override` must be present or the deployment fails.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCore

---

Example 6: Provisioning Data Directory (OmniHSS)

The OmniHSS provisioning service (`services/provision_omnihss.yml`) reads subscriber and profile data from a dedicated `omnihss/` directory inside group_vars.

File Structure

```
hosts/Customer/
└── group_vars/
    └── omnihss/
        ├── subscribers.csv              # Subscribers to provision
        ├── apn_identifiers.json
        ├── apn_qos_profiles.json
        ├── apn_profiles.json
        ├── eir_rules.json
        ├── ims_profiles.json
        ├── epc_profiles.json
        ├── roaming_rules.json
        ├── roaming_profiles.json
        ├── flow_information_rules.json
        ├── charging_rules.json
        └── pcrf_profiles.json
```

What happens:

1. The service reads `{{ group_vars_omnicore }}/omnihss/subscribers.csv` for the list of subscribers to provision.
2. It reads the accompanying JSON files from the same `omnihss/` directory (`apn_identifiers.json`, `apn_qos_profiles.json`, `apn_profiles.json`, `eir_rules.json`, `ims_profiles.json`, `epc_profiles.json`, `roaming_rules.json`, `roaming_profiles.json`, `flow_information_rules.json`, `charging_rules.json`, `pcrf_profiles.json`) to define the APN, QoS, IMS, EPC, roaming, flow, charging and PCRF data applied to those subscribers.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCore

---

Example 7: Grafana SMTP Credentials (Monitoring)

The monitoring role's `grafana.ini` template wires the Grafana alert email (SMTP) settings from group_vars. The user and password have **no role default** - they are deliberately not baked in, so a missing value fails the run loudly rather than shipping credentials.

Reference in Hosts File

```yaml
monitoring:
  hosts:
    customer-oam01:
      ansible_host: 10.10.3.135
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      grafana_smtp_user: "smtp-user"          # REQUIRED - no
default
      grafana_smtp_password: "smtp-password"  # REQUIRED - no
default
      grafana_smtp_host: "in-v3.mailjet.com:587"  # Optional -
defaults to in-v3.mailjet.com:587
```

What happens:

1. `grafana_smtp_user` and `grafana_smtp_password` are templated into the `[smtp]` section of `/etc/grafana/grafana.ini`. Both are required - there is no fallback.
2. `grafana_smtp_host` is optional and defaults to `in-v3.mailjet.com:587` if not set.

Configuration details: See https://docs.omnitouch.com.au/docs/repos/OmniCore

---

Example 8: HNET Keys for UDM SUCI De-concealment

The OmniUDM performs SUCI de-concealment (TS 33.501 §6.12) — recovering the SUPI from the concealed SUCI a UE presents at registration. This requires the Home Network **private** keys that pair with the Home Network **public** keys provisioned on the USIMs. These private keys live in a dedicated `hnet_udm/` directory inside group_vars and are deployed to `/etc/omniudm/hnet/` on the UDM host by the `5gc_nf` role.

File Structure

```
hosts/Customer/
└── group_vars/
    └── hnet_udm/
        ├── curve25519-1.key       # X25519 private key, PEM (Home
Network Public Key ID 1)
        ├── curve25519-1.pub       # Raw 32-byte public value
(hex) — for USIM provisioning
        ├── secp256r1-2.key        # EC (prime256v1) private key,
PEM (HNET Public Key ID 2)
        └── ...
```

File naming: `<curve>-<id>.key`, where `<id>` is the Home Network Public Key Identifier the UDM matches against the SUCI, and `<curve>` is `curve25519` (Profile A, X25519) or `secp256r1` (Profile B, prime256v1). The filename is how the key is bound to its HNET key ID — it must match the ID programmed onto the USIM.

What happens:

1. The role ensures `{{ group_vars_omnicore }}/hnet_udm/` exists on the Ansible controller (created with `become: false` — keys stay on the controller, never generated on the remote box).
2. It scans the directory for `*.key` files.
3. **If none are found, a default `curve25519-1.key` (X25519) is generated automatically** with `openssl genpkey`, so SUCI de-

concealment works out of the box. This is idempotent (`creates:` guard) — an existing key is never overwritten.

4. For each `curve25519-*.key`, the matching raw 32-byte public value is exported to a `.pub` file (hex) for USIM provisioning. The `.pub` files stay in inventory and are **not** deployed to the UDM.
5. All `*.key` files are copied to `/etc/omniudm/hnet/` on the UDM (mode `0600`, root-owned), and `omniudm` is restarted.

> **Important:** An auto-generated key is random and will only de-conceal SUCIs from USIMs programmed with its **matching** public key. For a lab where you control SIM programming, take the exported `.pub` and provision it as the Home Network Public Key (ID 1) on the USIM. If your SIMs use the null-scheme (no concealment), no HNET key is needed and this directory can stay empty.

To export a public key from an existing private key manually:

```
openssl pkey -in curve25519-1.key -pubout -outform DER | tail -c
32 | xxd -p -c 32
```

Configuration details: See
https://docs.omnitouch.com.au/docs/repos/OmniCore

Real-World Directory Structure

Example

```
hosts/Customer/
├── host_files/
│   └── production.yml          # Hosts file references group_vars
files
└── group_vars/
    ├── smsc_controller.exs     # OmniMessage custom template
    ├── smsc_smpp.exs           # OmniMessage SMPP custom template
    ├── tas_runtime.exs.j2      # TAS custom template
    ├── hss_runtime.exs.j2      # HSS custom template
    ├── mme_runtime.exs.j2      # MME custom template
    ├── dra_runtime.exs.j2      # DRA custom template
    ├── pgwc_runtime.exs.j2     # PGW custom template
    ├── dea_runtime.exs.j2      # DEA custom template
    ├── upf_config.yaml         # UPF configuration
    ├── crm_config.yaml         # CRM configuration
    ├── stp.j2                  # SS7 STP template
    ├── hlr.j2                  # SS7 HLR template
    ├── camel.j2                # SS7 CAMEL template
    ├── ipsmgw.j2               # IP-SM-GW template
    ├── omnicore_smsc_ims.yaml.j2  # SMSC IMS config
    ├── pytap.yaml              # TAP3 configuration
    ├── sip_profiles/           # SIP gateways (folder)
    │   └── gateway_otw.xml
    ├── dialplan/               # Call routing rules (folder)
    │   ├── mo_dialplan.xml     # Mobile Originated
    │   ├── mt_dialplan.xml     # Mobile Terminated
    │   └── mo_emergency.xml    # Emergency routing
    ├── omnihss/                # OmniHSS provisioning data
(folder)
    │   ├── subscribers.csv         # Subscribers to provision
    │   ├── apn_identifiers.json
    │   ├── apn_qos_profiles.json
    │   ├── apn_profiles.json
    │   ├── eir_rules.json
    │   ├── ims_profiles.json
    │   ├── epc_profiles.json
    │   ├── roaming_rules.json
    │   ├── roaming_profiles.json
    │   ├── flow_information_rules.json
    │   ├── charging_rules.json
```

```
    │   └── pcrf_profiles.json
    └── hnet_udm/               # UDM SUCI de-concealment keys
(folder)
        ├── curve25519-1.key        # X25519 private key (HNET
Public Key ID 1)
        └── curve25519-1.pub        # Raw public value (hex) for
USIM provisioning
```

Common Parameters That

Reference group_vars

Parameter	Component	References
`smsc_template_config`	omnimessage	Jinja2 template file (e.g., `smsc_controller.exs`)
`smsc_smpp_template_config`	omnimessage_smpp	Jinja2 template file (e.g., `smsc_smpp.exs`)
`gateways_folder`	applicationserver	Folder name (e.g., `gateways_prod`)
`dialplan_folder`	applicationserver	Folder name (e.g., `dialplan`) - defaults to `dialplan` if not set
`tas_runtime_template`	applicationserver	Jinja2 template file (e.g., `tas_runtime.exs.j2`) - defaults to `tas_runtime.exs.j2` if not set
`hss_template_config`	omnihss	Jinja2 template file (e.g., `hss_runtime.exs.j2`)
`mme_template_config`	omnimme	Jinja2 template file (e.g., `mme_runtime.exs.j2`)
`dra_template_config`	dra	Jinja2 template file (e.g., `dra_runtime.exs.j2`)

Parameter	Component	References
`pgwc_template_config`	pgwc	Jinja2 template file (e.g., `pgwc_runtime.exs.j2`)
`frr_template_config`	omniupf	Jinja2 template file (e.g., `frr.conf.j2`)
SS7 templates	ss7 (various roles)	Jinja2 template files (e.g., `stp.j2`, `hlr.j2`, `camel.j2`)
`crm_config.yaml`	omnicrm	API config file - **required**, sourced only from `{{ group_vars_omnicore }}/crm_config.yaml` (no role default)
`crm_env_override`	omnicrm	UI .env filename (e.g., `customer_crm.env`) - **required**, no role default
`cell_sites`	omnicrm	Optional cell site data file (e.g., `cellSites.txt`)
`message_templates`	omnicrm	Optional cell broadcast message templates (e.g., `constants.js`)
`site_data`	omnicrm	Optional cell broadcast geo bounds (e.g., `site_data.json`)

Parameter	Component	References
`grafana_smtp_user`	monitoring	Grafana SMTP user - **required**, no role default
`grafana_smtp_password`	monitoring	Grafana SMTP password - **required**, no role default
`grafana_smtp_host`	monitoring	Optional Grafana SMTP host - defaults to `in-v3.mailjet.com:587`
OmniHSS provisioning	omnihss	`omnihss/` directory with `subscribers.csv` + profile JSON files
HNET keys	5gc_nf (omniudm)	`hnet_udm/` directory of `<curve>-<id>.key` private keys for SUCI de-concealment - auto-generates a default `curve25519-1.key` if empty
Config YAMLs	Various components	Direct config files (e.g., `upf_config.yaml`, `crm_config.yaml`)

Key Points

1. **group_vars holds customizations** - Overrides for default configurations

2. **Reference by name** - Use parameters like `smsc_template_config` or `gateways_folder`
3. **Templates support Jinja2** - Access any Ansible variable with `{{ variable_name }}`
4. **Folders deploy everything** - All files in referenced folders are copied
5. **Version control everything** - Commit all group_vars to Git

When to Use group_vars

✅ Use group_vars for:

- Custom component configuration templates
- SIP gateway definitions
- Call routing dialplans
- Diameter routing rules
- Customer-specific settings that override defaults

❌ Don't use group_vars for:

- Basic host configuration (IPs, hostnames) - Use hosts file
- One-off testing - Use host-specific vars in hosts file
- Temporary changes - Edit on target, commit to group_vars if permanent

---

Related Documentation

- Configuration Reference - All host parameters and what they do
- Hosts File Configuration - How to structure hosts files
- **OmniCall Configuration**: https://docs.omnitouch.com.au/docs/repos/OmniCall - What goes in the config files
- **OmniCore Configuration**: https://docs.omnitouch.com.au/docs/repos/OmniCore - Component configuration details

Hosts File Configuration

Overview

The hosts file (also called inventory file) is the central configuration document that defines your entire cellular network deployment. It specifies:

- What network functions to deploy
- Where they run (IP addresses, network segments)
- How they're configured (service-specific parameters)
- Customer-specific settings (PLMN, credentials, features)

File Location

Hosts files live under the top-level `hosts/` directory (set as the Ansible `inventory` in `ansible.cfg`), organized by customer:

```
hosts/
└── Customer_Name/
    └── host_files/
        ├── Production.yml
        ├── Staging.yml
        └── Customer_Lab.yml
```

Filenames are arbitrary — there is one file per deployment, typically named after the deployment or site (e.g. `Production.yml`, `Staging.yml`) rather than a fixed `production.yml`/`staging.yml`/`lab.yml` scheme.

Example Hosts File Structure

Here's a simplified example showing the key sections:

```yaml
# EPC Components
mme:
  hosts:
    customer-mme01:
      ansible_host: 10.10.1.15
      gateway: 10.10.1.1
      host_vm_network: "vmbr1"
      mme_code: 1
      network_name_short: Customer
      tac_list: [600, 601, 602]

sgw:
  hosts:
    customer-sgw01:
      ansible_host: 10.10.1.25
      gateway: 10.10.1.1
      cdrs_enabled: true

pgwc:
  hosts:
    customer-pgw01:
      ansible_host: 10.10.1.21
      gateway: 10.10.1.1
      ip_pools:
        - '100.64.16.0/24'

# IMS Components
pcscf:
  hosts:
    customer-pcscf01:
      ansible_host: 10.10.4.165

# Support Services
license_server:
  hosts:
    customer-licenseserver:
      ansible_host: 10.10.2.150

# Global Variables
all:
  vars:
    ansible_connection: ssh
    ansible_password: password
```

```
    customer_name_short: customer
    plmn_id:
      mcc: '001'
      mnc: '01'
```

Common Host Parameters

Network Configuration

Every host typically includes:

```
pcscf:
  hosts:
    customer-pcscf01:
      ansible_host: 10.10.1.15      # IP address for SSH access
      gateway: 10.10.1.1             # Default gateway
      host_vm_network: "vmbr1"       # name of NIC to use on
Hypervisor
```

Note: For guidance on IP address planning and network segmentation strategies, see the IP Planning Standard which outlines the recommended four-subnet architecture for OmniCore deployments.

Proxmox Users: The `host_vm_network` parameter specifies which bridge to use. See Proxmox VM/LXC Deployment for automated provisioning.

VM Resource Allocation

For services needing specific resources:

```
num_cpus: 4                     # CPU cores
memory_mb: 8192                 # RAM in megabytes
proxmoxLxcDiskSizeGb: 50      # Disk size in GB
```

Service-Specific Parameters

Each network function has its own parameters. Examples:

MME:

```yaml
mme_code: 1                      # MME identifier (1-255)
mme_gid: 1                       # MME Group ID
network_name_short: Customer     # Network name (shown on phones)
network_name_long: Customer Network
tac_list: [600, 601, 602]      # Tracking Area Codes
```

PGW:

```yaml
ip_pools:                        # IP pools for subscribers
  - '100.64.16.0/24'
  - '100.64.17.0/24'
combined_CP_UP: false            # Separate control/user plane
```

For detailed explanation of what each variable controls, see: Configuration Reference

Application Server:

```yaml
online_charging_enabled: true  # Enable OCS integration
tas_branch: "main"             # Software branch to deploy
gateways_folder: "gateways_prod"  # SIP gateway configuration
```

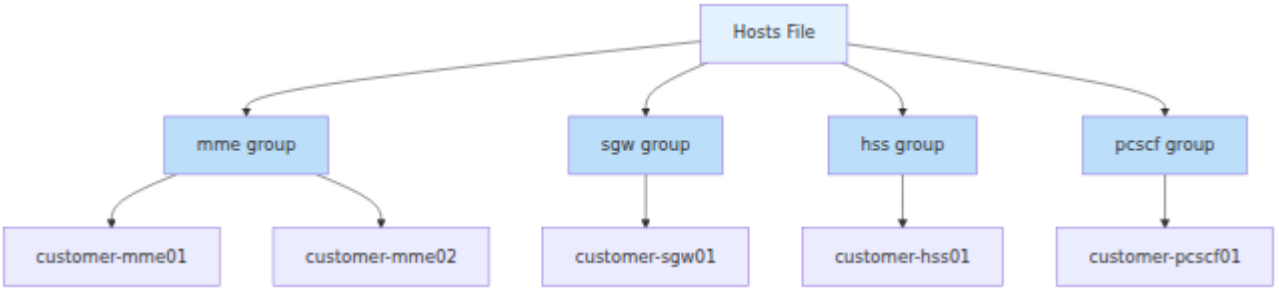
Global Variables Section

The `all:vars` section contains settings that apply to the entire deployment:

```yaml
all:
  vars:
    # Authentication
    ansible_connection: ssh
    ansible_password: password
    ansible_become_password: password

    # Customer Identity
    customer_name_short: customer
    customer_legal_name: "Customer Inc."
    site_name: "Chicago DC1"
    region: US

    # PLMN (Mobile Network) Identifier
    plmn_id:
      mcc: '001'              # Mobile Country Code
      mnc: '01'               # Mobile Network Code
      mnc_longform: '001'     # Zero-padded MNC

    # Network Names
    network_name_short: Customer
    network_name_long: Customer Network

    # APT Repository
    # Note: If apt_cache_servers group is defined with hosts,
    # use_apt_cache defaults to true and apt_repo.apt_server
    # defaults to the first cache server's IP automatically
    apt_repo:
      apt_server: "10.254.10.223"
      apt_repo_username: "customer"
      apt_repo_password: "secure-password"
    use_apt_cache: false

    # Timezone
    TZ: America/Chicago
```

Understanding Host Groups

Ansible organizes hosts into groups that correspond to roles:



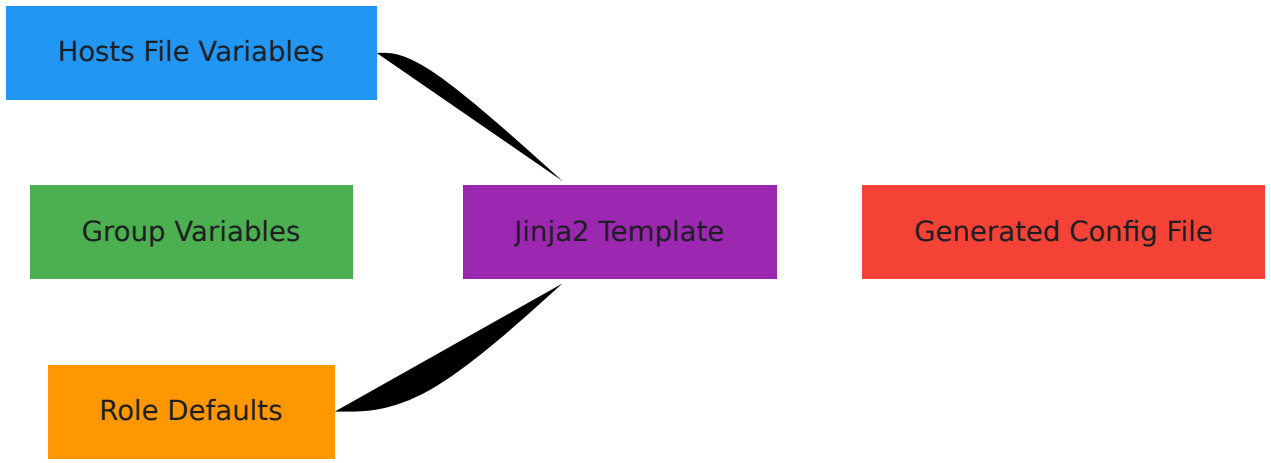
When you run a playbook targeting `mme`, it applies to all hosts in the `mme:hosts:` section.

The diagram above shows only a handful of groups. Many more exist, including `ocs`, `crm`, `smsc`, `omnimessage`, `dra`, `xcap`, `homer`, `cbc`, and the 5GC groups `amf`, `smf`, and `upf`, among others.

Configuration with Jinja2 Templates

Ansible uses **Jinja2 templating** to generate configuration files from the variables defined in your hosts file and group_vars.

How Jinja2 Works



Example Template Usage

Hosts file defines:

```yaml
plmn_id:
  mcc: '001'
  mnc: '01'
customer_name_short: acme
```

Jinja2 template (in role):

```yaml
# mme_config.yml.j2
network:
  plmn:
    mcc: {{ plmn_id.mcc }}
    mnc: {{ plmn_id.mnc }}
  operator: {{ customer_name_short }}
  realm: epc.mnc{{ plmn_id.mnc_longform }}.mcc{{ plmn_id.mcc
}}.3gppnetwork.org
```

Generated configuration file:

```yaml
network:
  plmn:
    mcc: 001
    mnc: 01
  operator: acme
  realm: epc.mnc001.mcc001.3gppnetwork.org
```

Common Jinja2 Patterns

Accessing nested variables:

```
{{ plmn_id.mcc }}
{{ apt_repo.apt_server }}
```

Conditional logic:

```
{% if online_charging_enabled %}
  charging:
    enabled: true
    ocs_ip: {{ ocs_ip }}
{% endif %}
```

Loops:

```
tracking_areas:
{% for tac in tac_list %}
  - {{ tac }}
{% endfor %}
```

Formatting:

```
# Zero-pad to 3 digits
mnc{{ '%03d' | format(plmn_id.mnc|int) }}
```

Overriding Variables with group_vars

While the hosts file defines infrastructure and host-specific settings, `group_vars` can override defaults for groups of hosts.

See: Group Variables Configuration

Complete Example Hosts File

Here's a more complete example (with sensitive data obscured):

```yaml
# EPC Core
mme:
  hosts:
    customer-mme01:
      ansible_host: 10.10.1.15
      gateway: 10.10.1.1
      host_vm_network: "vmbr1"
      mme_code: 1
      mme_gid: 1
      network_name_short: Customer
      network_name_long: Customer Network
      tac_list: [600, 601, 602, 603]
      omnimme:
        sgw_selection_method: "random_peer"
        pgw_selection_method: "random_peer"

sgw:
  hosts:
    customer-sgw01:
      ansible_host: 10.10.1.25
      gateway: 10.10.1.1
      host_vm_network: "vmbr1"
      cdrs_enabled: true

pgwc:
  hosts:
    customer-pgw01:
      ansible_host: 10.10.1.21
      gateway: 10.10.1.1
      host_vm_network: "vmbr1"
      ip_pools:
        - '100.64.16.0/24'
      combined_CP_UP: false

hss:
  hosts:
    customer-hss01:
      ansible_host: 10.10.2.140
      gateway: 10.10.2.1
      host_vm_network: "vmbr2"

# IMS Core
pcscf:
```

```
  hosts:
    customer-pcscf01:
      ansible_host: 10.10.4.165
      gateway: 10.10.4.1
      host_vm_network: "vmbr4"

icscf:
  hosts:
    customer-icscf01:
      ansible_host: 10.10.3.55
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"

scscf:
  hosts:
    customer-scscf01:
      ansible_host: 10.10.3.45
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"

applicationserver:
  hosts:
    customer-as01:
      ansible_host: 10.10.3.60
      gateway: 10.10.3.1
      host_vm_network: "vmbr3"
      online_charging_enabled: false
      gateways_folder: "gateways_prod"

# Support Services
license_server:
  hosts:
    customer-licenseserver:
      ansible_host: 10.10.2.150
      gateway: 10.10.2.1
      host_vm_network: "vmbr2"

monitoring:
  hosts:
    customer-oam01:
      ansible_host: 10.10.2.135
      gateway: 10.10.2.1
      host_vm_network: "vmbr2"
      num_cpus: 4
```

```
      memory_mb: 8192

dns:
  hosts:
    customer-dns01:
      ansible_host: 10.10.2.177
      gateway: 10.10.2.1
      host_vm_network: "vmbr2"

# Global Variables
all:
  vars:
    ansible_connection: ssh
    ansible_password: password
    ansible_become_password: password

    customer_name_short: customer
    customer_legal_name: "Customer Network Inc."
    site_name: "Primary DC"
    region: US
    TZ: America/Chicago

    # PLMN Configuration
    plmn_id:
      mcc: '001'
      mnc: '01'
      mnc_longform: '001'
    diameter_realm: epc.mnc{{ plmn_id.mnc_longform }}.mcc{{
plmn_id.mcc }}.3gppnetwork.org

    # Network Names
    network_name_short: Customer
    network_name_long: Customer Network
    tac_list: [600, 601]

    # APT Configuration
    apt_repo:
      apt_server: "10.254.10.223"
      apt_repo_username: "customer"
      apt_repo_password: "secure-password"
    use_apt_cache: false

    # Charging Configuration
    charging:
```

```
      data:
        online_charging:
          enabled: false
      voice:
        online_charging:
          enabled: true
          domain: "mnc{{ plmn_id.mnc_longform }}.mcc{{ plmn_id.mcc
}}.3gppnetwork.org"

    # Firewall Rules
    firewall:
      allowed_ssh_subnets:
        - '10.0.0.0/8'
        - '192.168.0.0/16'
      allowed_ue_voice_subnets:
        - '10.0.0.0/8'
      allowed_signaling_subnets:
        - '10.0.0.0/8'

    # Hypervisor Configuration (Proxmox VM example)
    proxmoxServers:
      customer-prxmx01:
        # deployment_type omitted → defaults to "vm"
        proxmoxServerAddress: 10.10.0.100
        proxmoxServerPort: 8006
        proxmoxApiTokenName: Customer
        proxmoxApiTokenSecret: "token-secret"
        proxmoxNodeName: pve01
        proxmoxTemplateName: ubuntu-24.04-cloud-init-template
        proxmoxTemplateId: 9000
        proxmoxTemplateUser: omnitouch        # optional cloud-
init username; defaults to the first local_users key
        proxmoxTemplatePassword: omnitouch    # optional cloud-
init password; defaults to the cloud-init username

    # For an LXC site, set deployment_type: lxc and
    # proxmoxLxcOsTemplate in each proxmoxServers entry instead.
```

See Proxmox VM/LXC Deployment for complete Proxmox setup and configuration details.

Product Documentation References

For detailed configuration of each component, refer to the official product documentation:

OmniCore Components:

- **OmniCore Documentation**: https://docs.omnitouch.com.au/docs/repos/OmniCore
- **OmniHSS** - Home Subscriber Server
- **OmniSGW** - Serving Gateway (Control plane)
- **OmniPGW** - Packet Gateway (Control plane)
- **OmniUPF** - User Plane Function
- **OmniDRA** - Diameter Routing Agent
- **OmniTWAG** - Trusted WLAN Access Gateway

OmniCall Components:

- **OmniCall Documentation**: https://docs.omnitouch.com.au/docs/repos/OmniCall
- **OmniTAS** - IMS Application Server (VoLTE/VoNR)
- **OmniCall CSCF** - Call Session Control Functions
- **OmniMessage** - SMS Center
- **OmniMessage SMPP** - SMPP Protocol Support
- **OmniSS7** - SS7 Signaling Stack
- **VisualVoicemail** - Voicemail

OmniCharge/OmniCRM:

- **OmniCharge Documentation**: https://docs.omnitouch.com.au/docs/repos/OmniCharge

Related Documentation

- Introduction to Ansible Deployment - Overall deployment process

- Configuration Reference - **Complete guide to all configuration variables**
- Group Variables Configuration - Overriding default configurations
- IP Planning Standard - **Network architecture and IP allocation guidelines**
- Netplan Configuration - **Secondary IPs and advanced network configuration**
- APT Cache System - Package distribution
- License Server - License management
- Deployment Architecture Overview - Complete system view

Next Steps

1. Create your hosts file based on this template
2. Define your PLMN and network identity
3. Configure APT repository access
4. Set up license server
5. Customize with group_vars as needed
6. Deploy with Ansible playbooks

License Server

Overview

The License Server manages feature activation for all Omnitouch components. Each component validates its license on startup and periodically during operation.

Setup

1. Define in Hosts File

```yaml
license_server:
  hosts:
    customer-licenseserver:
      ansible_host: 10.10.2.150
      gateway: 10.10.2.1
      host_vm_network: "vmbr2"

all:
  vars:
    customer_legal_name: "Customer Name"
    license_server_api_urls: ["https://10.10.2.150:8443/api"]
```

> **Note:** `license_enforced` is **HSS-scoped**, not a global enforcement toggle. It is only defined for the `omnihss` role (`roles/omnihss/defaults/main.yml`, where it defaults to `false`) and is currently only read by the HLD generator; no other role consumes it. Set it under the omnihss group vars if you need it.

2. Provide License File

Place `license.json` (provided by Omnitouch) in `hosts/Customer/group_vars/`

3. Deploy

```
ansible-playbook -i hosts/Customer/host_files/production.yml
services/license_server.yml
```

Real inventory directories are capitalized and prefixed (e.g. `hosts/Omnicore_Customer/`), and the host file is named after the site (e.g. `hosts/Omnicore_Customer/host_files/Production.yml`).

You can check the status of all license by browsing to https://license_server .

Removing a License

To remove a license, use `util_playbooks/delete_license.yml`.

Network Requirements

Firewall Configuration

Client site firewalls must be configured to allow HTTPS (port 443) traffic to the Omnitouch license validation servers:

Hostname	Purpose
1.time.omnitouch.com.au	License validation server 1
2.time.omnitouch.com.au	License validation server 2
3.time.omnitouch.com.au	License validation server 3

Required outbound rules:

- Protocol: HTTPS (TCP/443)
- Destination: the hostnames above, resolved via DNS
- Direction: Outbound

> **Resolve by DNS — do not hardcode IPs.** Always allow these hosts by FQDN (or by the addresses your firewall resolves them to dynamically). The underlying IPs can change without notice, so static IP allow-lists are not supported and will eventually break license validation.

DNS Requirements

The license server requires functional DNS resolution to communicate with the Omnitouch license validation infrastructure.

Required DNS configuration:

- The license server must have access to public DNS servers
- Configure DNS to use one of the following:
  - 1.1.1.1 (Cloudflare - supports secure DNS)
  - 8.8.8.8 (Google Public DNS)
- Do not use internal/corporate DNS servers for the license server

Note: The Omnitouch license servers use secure DNS (DoH/DoT). Using public DNS servers ensures proper DNSSEC validation and prevents issues with DNS interception by security appliances.

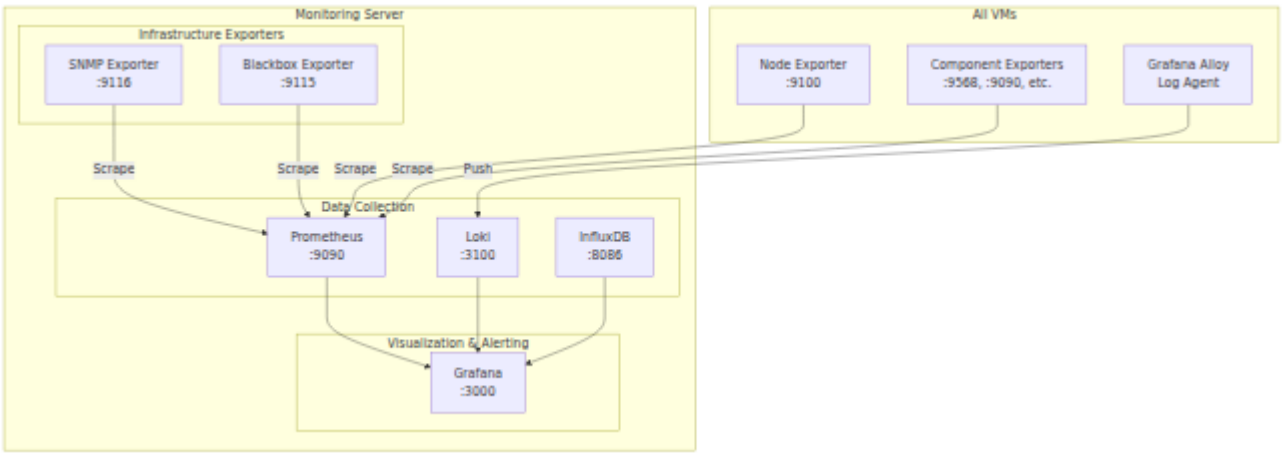
Related Documentation

- Configuration Reference
- Hosts File Configuration

Monitoring & Observability

Overview

OmniCore includes a comprehensive monitoring and observability stack that provides metrics collection, log aggregation, visualization, and alerting. The system is deployed automatically by the monitoring role.



Components

Component	Purpose	Port	Data Retention
Prometheus	Time-series metrics storage	9090	30 days or 5GB
Loki	Log aggregation	3100	7 days or 50GB
InfluxDB	RAN metrics (Nokia)	8086	30 days
Grafana	Visualization and alerting	3000	N/A
Node Exporter	Host metrics	9100	N/A
SNMP Exporter	Network device metrics	9116	N/A
Blackbox Exporter	ICMP/HTTP probes	9115	N/A

Data Sources

Prometheus

Prometheus scrapes metrics from all OmniCore components every 1 minute.

Datasource UID: `omnicore_prometheus`

Scraped Targets

Job Name	Inventory Group	Port	Metrics
`Node Exporter`	`all`	9100	Host CPU, memory, disk, network
`MMEs`	`mme`	9568	Session counts, attach/detach rates
`HSS`	`hss`	9568	Auth requests, subscriber lookups
`SGW-C`	`sgw`	9568	Bearer counts, GTP-C messages
`PGW-C`	`pgwc`	9090	PDN connections, IP allocations
`UPF Standalone`	`upf`	9090	Packet counts, throughput
`CSCF`	`pcscf`, `scscf`, `icscf`	9090	Registrations, transactions
`ApplicationServer FreeSWITCH`	`applicationserver`	9090	Call counts, channel usage
`DRA`	`dra`	9568	Routing decisions, peer status
`OmniMessage`	`omnimessage`	9568	SMS delivery, SMPP sessions
`OmniSS7`	`omniss7`	8080	SS7/SIGTRAN gateway metrics

Job Name	Inventory Group	Port	Metrics
`KeyDB`	`ocs`	9121	Memory usage, operations/sec
`CGrateS`	`ocs`	2080	Rating, charging, CDR stats

Infrastructure Exporters

Exporter	Targets	Configuration Variable
SNMP (MikroTik)	Routers/switches	`mikrotik.hosts`
SNMP (iDRAC)	Dell servers	`idrac.hosts`
SNMP (Synology)	NAS devices	`synology.hosts`
SNMP (SAF)	Microwave links	`saf.hosts`
SNMP (Cisco)	Switches	`cisco.hosts`
Blackbox ICMP	All hosts + 8.8.8.8	Automatic
VMware	vCenter clusters	`vcenter_ip`, `vcenter_password`
Proxmox	PVE clusters	`proxmoxServers`

Loki

Loki receives logs from Grafana Alloy agents running on all VMs.

Datasource UID: `omnicore_loki`

See Centralized Logging for detailed Loki/Alloy configuration.

Log Labels

Label	Description	Example
`hostname`	Source VM hostname	`customer-mme01`
`component`	Component type	`mme`, `cscf`, `ocs`
`unit`	systemd unit name	`omnimme.service`
`level`	Log severity	`info`, `error`

InfluxDB

InfluxDB stores time-series data for specific use cases requiring longer retention or different query patterns.

Datasource UID: `omnicore_influxdb`

Databases

Database	Purpose	Retention
`nokia-monitor`	Nokia RAN performance metrics	30 days
`dra`	DRA routing statistics	365 days
`Omnicharge_TAP3`	Roaming TAP file metrics	90 days

Grafana Dashboards

Dashboard Organization

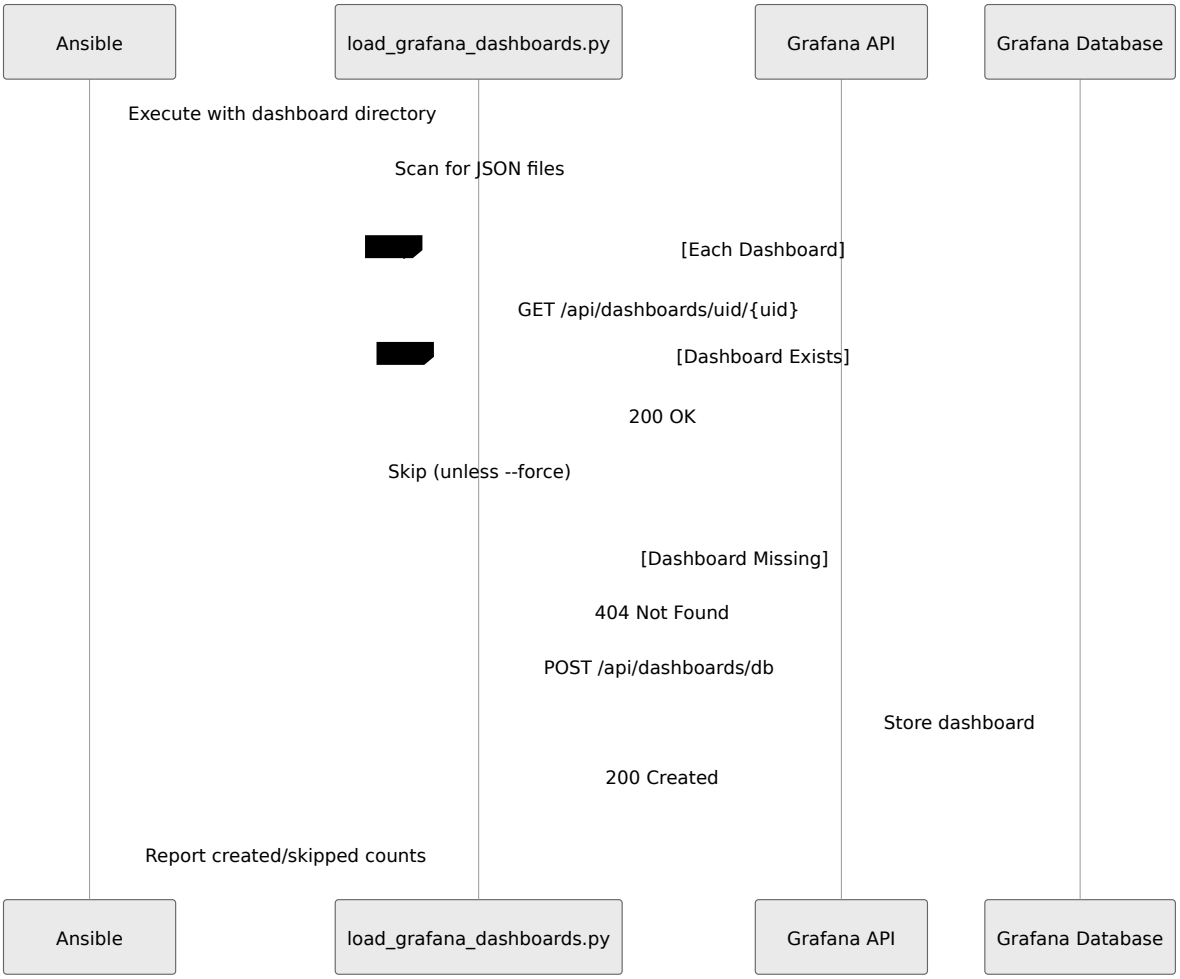
Dashboards are organized into folders by functional area:

Folder	Contents
BSS	CGrateS, KeyDB, charging dashboards
EPC	MME, SGW, PGW, UPF, HSS, DRA dashboards
IMS	CSCF, Application Server, OmniMessage dashboards
Infrastructure	Node Exporter, SNMP, network monitoring
RAN	Nokia/OmniRAN performance dashboards
Logs	Component-specific log viewers

Dashboard Loading (API-Based)

Dashboards are loaded via the Grafana API rather than file-based provisioning. This allows dashboards to be:

- Edited through the Grafana UI
- Modified via API
- Version controlled in the Ansible repository



Loading Dashboards

Dashboards are loaded automatically during Ansible deployment. To manually reload:

```
# From the Ansible controller
python3 roles/monitoring/files/load_grafana_dashboards.py \
    --url http://<monitoring-ip>:3000 \
    --user admin \
    --password <grafana_admin_password> \
    --dashboards-dir roles/monitoring/templates/grafana/dashboards

# Force update existing dashboards
python3 roles/monitoring/files/load_grafana_dashboards.py \
    --url http://<monitoring-ip>:3000 \
    --user admin \
    --password <grafana_admin_password> \
    --dashboards-dir roles/monitoring/templates/grafana/dashboards
\
    --force
```

Dashboard Source Files

Dashboard JSON files are stored in the Ansible repository:

```
roles/monitoring/templates/grafana/dashboards/
├── BSS/
│   ├── KeyDB_Cluster.json
│   ├── cgrates_mysql.json
│   └── cgrates_stats.json
├── EPC/
│   ├── MME_Dashboard.json
│   ├── OmniHSS.json
│   ├── OmniDRA.json
│   ├── SGW.json
│   ├── PGWs.json
│   └── ...
├── IMS/
│   ├── SMSc.json
│   ├── MMSc.json
│   └── ...
├── Infrastructure/
│   ├── Node_Exporter_Full.json
│   ├── MikroTik_Dashboard.json
│   └── ...
├── RAN/
│   ├── Nokia_Overview.json
│   ├── Nokia_Detailed.json
│   └── ...
└── Logs/
    ├── CSCF_Logs.json
    ├── MME_Logs.json
    └── ...
```

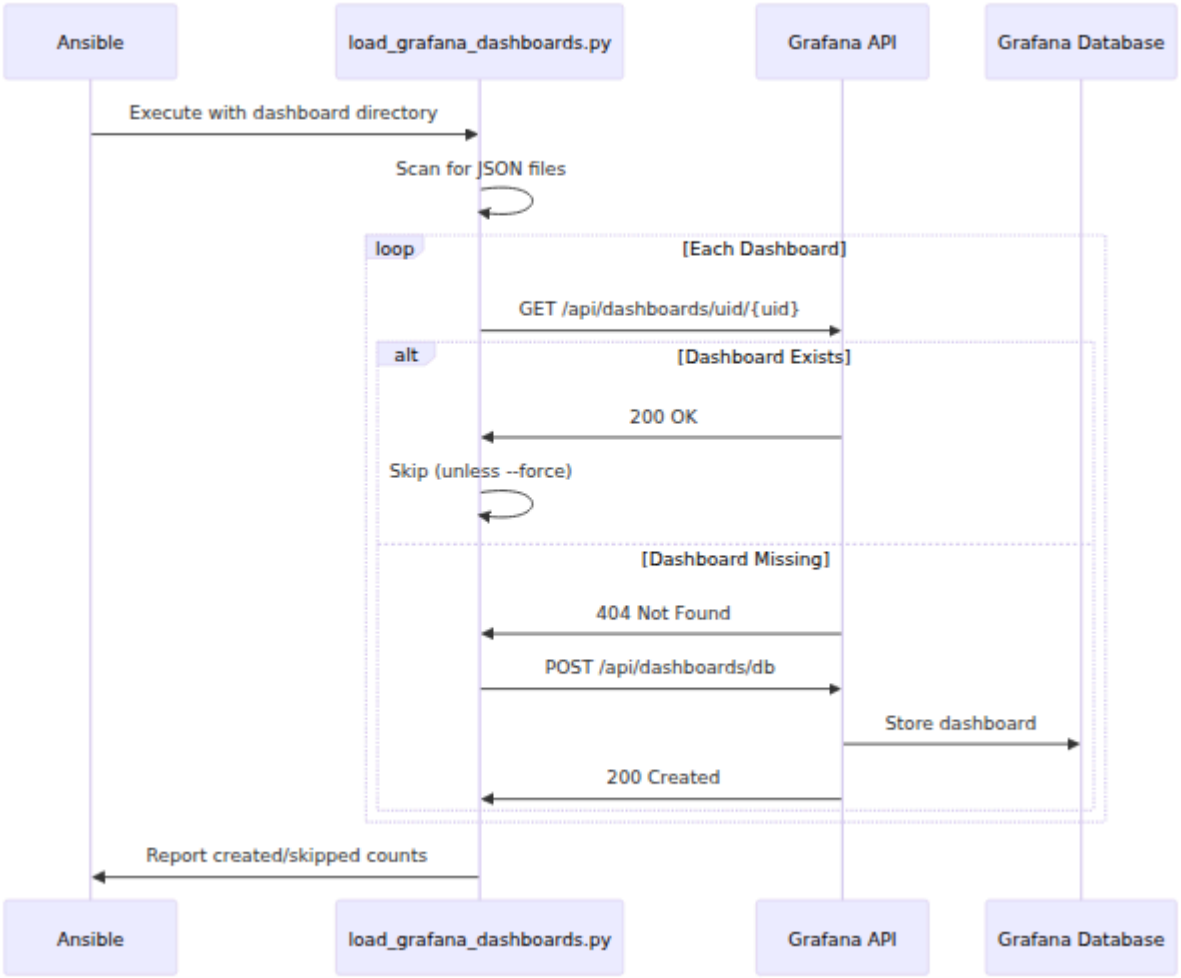
Dashboard Backup

Export dashboards and alerts from a running Grafana instance to the repository for version control:

```
# From roles/monitoring directory
python3 export_grafana.py --customer <CUSTOMER>
```

This exports:

- All dashboards to `roles/monitoring/templates/grafana/dashboards/{folder}/*.json` (shared across customers)
- Alert rules to `hosts/Omnicore_{CUSTOMER}/group_vars/grafana/alerts/all_alert_rules.json`
- Contact points to `hosts/Omnicore_{CUSTOMER}/group_vars/grafana/alerts/contact_points.json`
- Notification policies to `hosts/Omnicore_{CUSTOMER}/group_vars/grafana/alerts/notification_policies.json`



Export Behavior

- Only writes files that have changed (ignoring volatile fields like `version` and `id`)

- Preserves folder structure matching Grafana organization
- Maps dashboard titles to consistent filenames

Custom Dashboards

Customer-specific dashboards can be placed in `group_vars/grafana/dashboards/` and will be loaded alongside the standard dashboards:

```
hosts/customer/group_vars/
└── grafana/
    └── dashboards/
        ├── Custom_Dashboard_1.json
        └── Custom_Dashboard_2.json
```

Alerting

Architecture


Grafana Alerting
Alert Rules
alerts.yaml
Evaluation Engine
1m interval
Alert triggers
Notification Router
Data Sources
Prometheus
Loki
Contact Points
Slack Webhook
Google Chat
Email
Pushover

Alert Loading

Alerts are loaded via the Grafana API from per-customer exports rather than a single static YAML file. At deploy time the static alert provisioning file (`/etc/grafana/provisioning/alerting/alerts.yaml`) is removed so that alerts can be edited via the UI/API.

The primary path loads per-customer alert rules, contact points, and notification policies from `hosts/Omnicore_<CUSTOMER>/group_vars/grafana/alerts/` using `load_grafana_from_export.py`:

```
python3 roles/monitoring/files/load_grafana_from_export.py \
    --url http://<monitoring-ip>:3000 \
    --user admin \
    --password <grafana_admin_password> \
    --alerts-dir
hosts/Omnicore_<CUSTOMER>/group_vars/grafana/alerts \
    --dashboards-dir roles/monitoring/templates/grafana/dashboards
\
    --force
```

If no customer-specific alerts exist, the role falls back to loading the role-template `alerts.yaml` via `load_grafana_alerts.py`:

```
python3 roles/monitoring/files/load_grafana_alerts.py \
    --url http://<monitoring-ip>:3000 \
    --user admin \
    --password <grafana_admin_password> \
    --alerts-file
roles/monitoring/templates/grafana/provisioning/alerting/alerts.yaml
```

Default Alert Rules

Alert	Folder	Condition	For Duration
Disk Space 90% Used	Infrastructure	Root filesystem > 90%	5 minutes
CPU Above 90%	Infrastructure	CPU usage > 90%	5 minutes
System Load over 90%	Infrastructure	Load average > 90%	5 minutes
Host Down	Infrastructure	Prometheus target unreachable	5 minutes
MME Subs Dropped 40%	EPC	Session count drops 40%	30 seconds
0 Subs on MME	EPC	No attached subscribers	5 minutes
IMS Subs down 40%	IMS	P-CSCF registrations drop 40%	30 seconds
Average MOS below 4	IMS	Voice quality degraded	5 minutes
eNodeB Count Dropped	RAN	Connected eNBs decreased	1 minute
Diameter Response Delay High	EPC	P95 latency spike	5 minutes

Alert Rule Structure

Alert rules in `alerts.yaml`:

```yaml
apiVersion: 1
groups:
  - orgId: 1
    name: Alerts Group
    folder: Infrastructure
    interval: 1m
    rules:
      - uid: alert-lowDiskSpace
        title: Disk Space 90% Used on host
        condition: C
        data:
          - refId: A
            datasourceUid: omnicore_prometheus
            model:
              expr: |
                100 - ((node_filesystem_avail_bytes{job="Node
Exporter",mountpoint="/"} * 100)
                / node_filesystem_size_bytes{job="Node
Exporter",mountpoint="/"})
          # ... threshold expressions
        noDataState: OK
        execErrState: Error
        for: 5m
        annotations:
          summary: Disk space hit above 90% on host
        labels:
          type: resource
```

Contact Points Configuration

Contact points are configured via inventory variables:

```yaml
# In group_vars/all.yml or customer-specific vars
monitoring_data:
  contactPoints:
    - orgId: 1
      name: default
      receivers:
        # Slack integration
        - uid: slack-alerts
          type: slack
          disableResolveMessage: false
          settings:
            url: "https://hooks.slack.com/services/xxx/yyy/zzz"

        # Google Chat integration
        - uid: gchat-alerts
          type: googlechat
          disableResolveMessage: false
          settings:
            url:
"https://chat.googleapis.com/v1/spaces/xxx/messages?key=yyy"
```

Supported contact point types include Slack, Google Chat, Email, and Pushover (see `roles/monitoring/templates/grafana/contact-points.yaml.j2`). Pushover is configured via a `notifications.pushover` block (with `enabled`, `api_token`, and `user_key`).

Notification Policies

Notification policies route alerts to appropriate contact points:

```yaml
# templates/grafana/notification-policies.yaml.j2
apiVersion: 1

policies:
  - orgId: 1
    receiver: default
    group_by: ['grafana_folder', 'alertname']
    group_wait: 30s
    group_interval: 5m
    repeat_interval: 4h
```

Configuration Reference

Prometheus Configuration

Located at `/etc/prometheus/prometheus.yml` on monitoring server.

Parameter	Default	Description
`scrape_interval`	1m	How often to scrape targets
`evaluation_interval`	1m	How often to evaluate recording rules
`scrape_timeout`	50s	Timeout for scrape requests

Grafana Configuration

Located at `/etc/grafana/grafana.ini` on monitoring server.

Key settings configured by the monitoring role:

Setting	Value	Description
`allow_embedding`	true	Enable embedding in iframes
`provisioning`	`/etc/grafana/provisioning`	Provisioning directory

The admin password is not set in `grafana.ini` (the `admin_password` line is left commented). Instead it is applied via the `grafana_admin_password` variable, which is passed to the API loader tasks (dashboard/alert loaders) during deployment.

Email (SMTP) Alerting

Email alerting requires SMTP credentials to be set in `group_vars` (see `roles/monitoring/templates/grafana/grafana.ini`):

Variable	Required	Default	Description
`grafana_smtp_user`	Yes	*(none)*	SMTP username. No default — the deploy fails loudly if unset, rather than shipping a secret.
`grafana_smtp_password`	Yes	*(none)*	SMTP password. No default — the deploy fails if unset.
`grafana_smtp_host`	No	`in-v3.mailjet.com:587`	SMTP server host:port.
`grafana_smtp_enabled`	No	`true`	Whether SMTP is enabled.

The `from_address` and `from_name` are derived from `customer_legal_name` (e.g. `alerts+<customer_legal_name>@omnitouch.com.au` and `Omnitouch Alerts - <customer_legal_name>`).

Loki Configuration

See Centralized Logging for Loki retention and storage settings.

InfluxDB Configuration

Located at `/etc/influxdb/influxdb.conf` on monitoring server.

Database	Retention Policy	Duration
`nokia-monitor`	`autogen`	30 days
`dra`	`autogen`	365 days
`Omnicharge_TAP3`	`autogen`	90 days

Service Ports

Service	Port	Protocol	Description
Grafana	3000	HTTP	Dashboard UI and API
Prometheus	9090	HTTP	Metrics storage and query
Loki	3100	HTTP	Log ingestion and query
InfluxDB	8086	HTTP	InfluxDB API
Node Exporter	9100	HTTP	Host metrics
SNMP Exporter	9116	HTTP	SNMP metrics proxy
Blackbox Exporter	9115	HTTP	Probe metrics
Alloy	12345	HTTP	Alloy UI and metrics

Common Operations

Viewing Metrics

1. Open Grafana at `http://<monitoring-ip>:3000`
2. Navigate to **Dashboards** and select the appropriate folder
3. Use time range selector to adjust the view window

Searching Logs

1. Open Grafana at `http://<monitoring-ip>:3000`
2. Go to **Explore** and select **Loki** datasource
3. Use LogQL queries:

```
# All logs from a specific host
{hostname="customer-mme01"}

# Errors from all MME components
{component="mme"} |~ "(?i)error"

# Search for specific IMSI
{component="hss"} |= "123456789012345"
```

Creating Custom Alerts

The recommended workflow is to create/edit alerts in the Grafana UI, then export them to the customer's `group_vars/grafana/alerts/` directory (see Dashboard Backup) so they are version controlled and re-applied on the next deploy.

To load customer-specific alerts manually:

```
python3 roles/monitoring/files/load_grafana_from_export.py \
    --url http://<monitoring-ip>:3000 \
    --user admin --password <password> \
    --alerts-dir
hosts/Omnicore_<CUSTOMER>/group_vars/grafana/alerts \
    --dashboards-dir roles/monitoring/templates/grafana/dashboards
\
    --force
```

The role-template `roles/monitoring/templates/grafana/provisioning/alerting/alerts.yaml` is only used as a fallback when a customer has no exported alerts.

Backing Up Dashboards

After making changes in Grafana UI, export to repository:

```
cd roles/monitoring
python3 export_grafana.py --customer <CUSTOMER> --url
http://<monitoring-ip>:3000
git add ../../hosts/Omnicore_<CUSTOMER>/group_vars/grafana/
git commit -m "Update alerts from production"
```

The `--customer` argument is required. This script exports alert rules, contact points, and notification policies to the customer's `group_vars/grafana/alerts/` directory; it no longer exports dashboards, which are shared across customers in the role directory (`roles/monitoring/templates/grafana/dashboards/`).

Troubleshooting

Prometheus Target Down

Symptoms: Dashboard shows "No Data" or target status shows DOWN

Possible causes:

- Service not running on target host
- Firewall blocking exporter port
- Incorrect hostname resolution

Resolution:

1. Check if service is running on target:

   ```
   systemctl status <service>
   curl http://localhost:<port>/metrics
   ```

2. Verify connectivity from monitoring server:

   ```
   curl http://<target>:<port>/metrics
   ```

3. Check Prometheus targets page: `http://<monitoring-ip>:9090/targets`

Dashboard Not Loading

Symptoms: Dashboard shows loading spinner or error

Possible causes:

- Datasource connection failed
- Query timeout
- Dashboard JSON corruption

Resolution:

1. Test datasource in Grafana: **Configuration → Data Sources → Test**
2. Check Grafana logs: `journalctl -u grafana-server -f`
3. Try re-loading dashboard from repository

Alerts Not Firing

Symptoms: Condition is met but no notification received

Possible causes:

- Alert is paused
- Contact point misconfigured
- Notification policy not matching

Resolution:

1. Check alert state in Grafana: **Alerting** → **Alert rules**
2. Verify contact point test works: **Alerting** → **Contact points** → **Test**
3. Review notification policy routing

Grafana Cannot Connect to Datasource

Symptoms: "Bad Gateway" or connection timeout in Grafana

Possible causes:

- Prometheus/Loki/InfluxDB service down
- Firewall blocking localhost connections
- Service bound to wrong interface

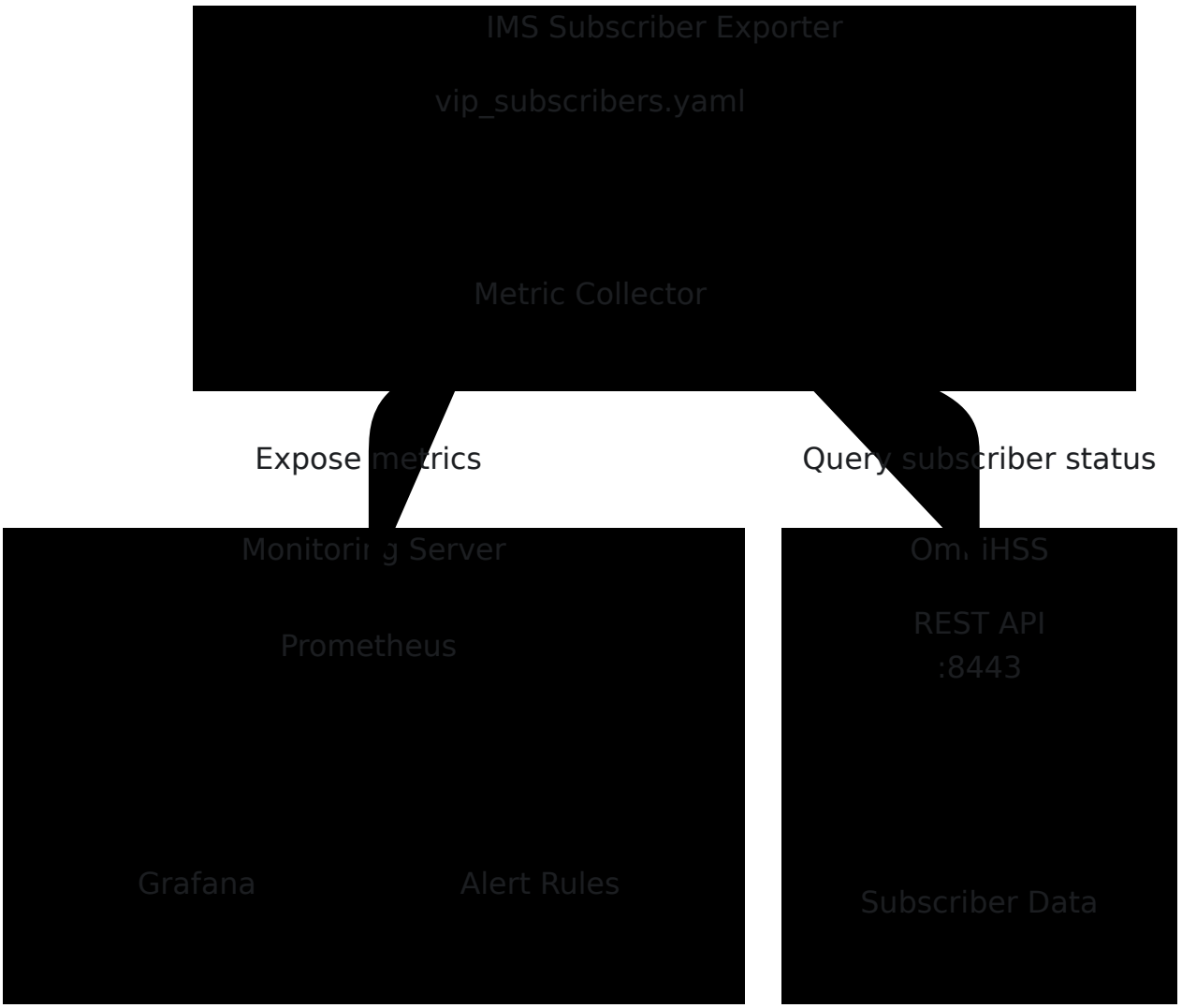
Resolution:

1. Check service status:
   ```
   systemctl status prometheus loki influxdb
   ```
2. Verify service is listening:
   ```
   ss -tlnp | grep -E '9090|3100|8086'
   ```
3. Test local connectivity:
   ```
   curl http://localhost:9090/api/v1/status/config
   curl http://localhost:3100/ready
   ```

VIP Subscriber Monitoring

VIP Subscriber Monitoring provides real-time status tracking for critical subscribers. The system monitors IMS registration, EPC attachment, and UE reachability for designated high-priority subscribers such as emergency services, hospitals, and key infrastructure.

Architecture



Service Types

Subscribers are categorized by expected service type, which determines health evaluation:

Type	IMS Required	EPC Required	Use Case
`full`	Yes	Yes	Standard mobile subscribers with voice and data
`voip_only`	Yes	No	VoIP-only devices (IP phones, softphones)
`data_only`	No	Yes	Data-only devices (routers, IoT, M2M)

Health evaluation:

- **full**: Healthy when both IMS registered (`assigned_scscf` not null) AND EPC attached (`last_seen_mme` not null)
- **voip_only**: Healthy when IMS registered (`assigned_scscf` not null)
- **data_only**: Healthy when EPC attached (`last_seen_mme` not null)

Configuration

VIP subscribers are configured in the host group variables:

File: `hosts/<customer>/group_vars/vip_subscribers.yaml`

```yaml
vip_subscriber_monitoring:
  hss_url: "https://10.80.12.140:8443"

  # Monitoring settings
  scrape_interval_seconds: 30
  ping_timeout_seconds: 2
  ping_count: 2

  # Subscribers to monitor
  subscribers:
    # Full service subscribers (IMS + EPC)
    - imsi: "001010000000001"
      label: "Emergency Services"
      type: "full"

    - imsi: "001010000000002"
      label: "Critical Infrastructure"
      type: "full"

    # Data-only services (routers, IoT - no IMS expected)
    - imsi: "001010000000003"
      label: "Site Router"
      type: "data_only"

    # VoIP-only services (IMS expected, no data bearer)
    - imsi: "001010000000004"
      label: "IP Phones"
      type: "voip_only"
```

Configuration Parameters

Parameter	Type	Required	Default	Descrip
`hss_url`	String	Yes	-	OmniHSS RES (HTTPS)
`scrape_interval_seconds`	Integer	No	30	How often to subscriber sta
`ping_timeout_seconds`	Integer	No	2	Timeout for U tests
`ping_count`	Integer	No	2	Number of pi packets to se
`blackbox_exporter_url`	String	No	-	Remote black exporter URL tests (e.g., `http://pcscf` Uses remote l probe instead ping.
`subscribers`	List	Yes	-	List of subscri monitor

Subscriber Parameters

Parameter	Type	Required	Default	Description
`imsi`	String	Yes	-	Subscriber IMSI (15 digits)
`label`	String	No	IMSI	Human-readable name for display
`type`	String	No	`full`	Service type: `full`, `voip_only`, or `data_only`

Note: MSISDN is automatically retrieved from the HSS API response and does not need to be configured.

Metrics

The exporter exposes metrics on port 9550 at `/metrics`.

Status Metrics

Metric	Type	Description
`vip_subscriber_service_healthy`	Gauge	1 if subscriber meets health criteria for their service type, 0 otherwise
`vip_subscriber_ims_registered`	Gauge	1 if subscriber has assigned S-CSCF, 0 otherwise
`vip_subscriber_epc_registered`	Gauge	1 if subscriber has active MME attachment, 0 otherwise
`vip_subscriber_ue_ip_reachable`	Gauge	1 if UE IP responds to ping, 0 otherwise
`vip_subscriber_hss_reachable`	Gauge	1 if HSS API returned valid data, 0 otherwise
`vip_subscriber_enabled`	Gauge	1 if subscriber account is enabled in HSS, 0 otherwise

Age Metrics

Metric	Type	Description
`vip_subscriber_ims_registration_age_seconds`	Gauge	Seconds since last IMS registration (-1 if not registered)
`vip_subscriber_epc_registration_age_seconds`	Gauge	Seconds since last EPC update location (-1 if not attached)

Info Metric

Metric	Type	Description
`vip_subscriber_info`	Gauge	Always 1, carries all status and info as labels

Labels on all metrics:

Label	Description	Example
`imsi`	Subscriber IMSI	`001010000000002`
`label`	Human-readable name	`Critical Infrastructure`
`msisdn`	Phone number (from HSS)	`61400000002`
`type`	Service type	`full`, `voip_only`, `data_only`

Additional labels on `vip_subscriber_info`:

Label	Description	Example
`healthy`	Service health status	`1` or `0`
`ims`	IMS registration status	`1` or `0`
`epc`	EPC attachment status	`1` or `0`
`ping`	UE IP reachability	`1` or `0`
`assigned_scscf`	S-CSCF hostname	`scscf01.ims.example.com`
`last_seen_mme`	MME hostname	`mme02.epc.example.com`
`ue_ip`	UE assigned IP address	`100.72.83.20`

Example Queries

```
# Count of healthy VIP subscribers
count(vip_subscriber_service_healthy == 1)

# Count of unhealthy VIP subscribers
count(vip_subscriber_service_healthy == 0)

# IMS registration status for VoIP services
vip_subscriber_ims_registered{type=~"voip_only|full"}

# EPC attachment status for data services
vip_subscriber_epc_registered{type=~"data_only|full"}

# Subscribers with stale IMS registration (>1 hour)
vip_subscriber_ims_registration_age_seconds > 3600
```

Dashboard

The **IMS VIP Subscribers** dashboard provides real-time visibility into VIP subscriber status.

Location: Grafana → IMS → IMS VIP Subscribers

Dashboard URL: `/d/ims-vip-subscribers/`

Panels

Panel	Description
Services Healthy	Count of subscribers meeting health criteria
Services Unhealthy	Count of subscribers failing health criteria
IMS Registered	Count of subscribers with active IMS registration
EPC Registered	Count of subscribers with active EPC attachment
UE Reachable	Count of subscribers with pingable UE IP
Total Monitored	Total number of configured VIP subscribers
VIP Subscriber Status	Table showing all subscribers with status columns
Service Health Over Time	State timeline showing health history

Alerting

Alert rules fire when VIP subscriber services become unhealthy.

VIP Subscriber Service Unhealthy

Alert: `VIP Subscriber Service Unhealthy` **Severity**: Critical **For**: 1 minute
Condition: `vip_subscriber_service_healthy == 0`

Annotations:

- **Summary**: `VIP Subscriber {{ $labels.label }} is unhealthy`
- **Description**: Includes service type and appropriate identifier:
  - VoIP services: Name and MSISDN
  - Data services: Name and IMSI
  - Full services: Name, MSISDN, and IMSI

Example alert descriptions:

- `IP Phones (voip_only) service is DOWN. MSISDN: 61400000001`
- `Site Router (data_only) service is DOWN. IMSI: 001010000000003`
- `Critical Infrastructure (full) service is DOWN. MSISDN: 61400000002 IMSI: 001010000000002`

Adding New VIP Subscribers

1. Edit the configuration file:

```yaml
# hosts/<customer>/group_vars/vip_subscribers.yaml
subscribers:
  - imsi: "123456789012345"
    label: "New VIP Subscriber"
    type: "full"  # or voip_only, data_only
```

2. Deploy the updated configuration:

```
scp vip_subscribers.yaml <monitoring-server>:/tmp/
ssh <monitoring-server> "sudo cp /tmp/vip_subscribers.yaml
/etc/prometheus/vip_subscribers.yaml && sudo systemctl restart
ims-subscriber-exporter"
```

3. Verify the new subscriber appears in metrics:

```
curl -s http://<monitoring-server>:9550/metrics | grep "New VIP
Subscriber"
```

Related Documentation

- Centralized Logging — Detailed Loki and Alloy configuration
- Deployment Architecture — Overall system architecture
- Hosts File Configuration — Inventory configuration

Netplan Configuration

Overview

OmniCore can automatically configure network interfaces on deployed VMs using netplan. This is useful for:

- Setting up the primary management interface (eth0)
- Adding secondary interfaces for public IPs, peering connections, or dedicated traffic
- Configuring static routes for specific destinations

Enabling Netplan Configuration

To enable automatic netplan configuration for a host, add the `netplan_config` variable pointing to a Jinja2 template in your `group_vars` folder:

```
dra:
  hosts:
    <hostname>:
      ansible_host: 10.0.1.100
      gateway: 10.0.1.1
      netplan_config: netplan.yaml.j2
```

The template will be sourced from `hosts/<customer>/group_vars/netplan.yaml.j2`.

Template Reference

There is no shared, canonical `netplan.yaml.j2`. Each customer creates its own `netplan.yaml.j2` in `hosts/<customer>/group_vars/`, and these templates diverge - the primary interface name, DNS search domain, and secondary-interface naming all vary by customer. When onboarding a new customer,

create a per-customer template that matches that customer's interface layout rather than copying another customer's verbatim.

The following is one customer's template as an **example**. It is illustrative, not a spec - other customers differ. Comments are added here for explanation:

```
network:
  version: 2
  ethernets:
    # Primary interface - uses ansible_host and gateway from
inventory
    eth0:
      addresses:
      - "{{ ansible_host }}/{{ mask_cidr | default(24) }}"
      nameservers:
        addresses:
{% if 'dns' in group_names %}
        # If this host IS a DNS server, use external DNS to avoid
circular dependency
        - 8.8.8.8
{% else %}
        # Otherwise, use DNS servers from the 'dns' group in
inventory
{% for dns_host in groups['dns'] | default([]) %}
        - {{ hostvars[dns_host]['ansible_host'] }}
{% endfor %}
{% endif %}
        # DNS search domain - customer-specific (varies per
customer; some set none)
        search:
        - customer.example
      routes:
      - to: "default"
        via: "{{ gateway }}"

{% if secondary_ips is defined %}
    # Secondary interfaces - loop through secondary_ips dict from
inventory
    # This example names them ens19, ens20, ens21... (18 +
loop.index). This is
    # template-specific; see "Interface Naming" below.
{% for nic_name, nic_config in secondary_ips.items() %}
    ens{{ 18 + loop.index }}:
      addresses:
      # Mask comes from the per-NIC mask_cidr (defaults to 24)
      - "{{ nic_config.ip_address }}/{{ nic_config.mask_cidr |
default(24) }}"
{% if nic_config.routes is defined %}
      # Static routes for this interface - each route uses this
```

```
interface's gateway
      routes:
{% for route in nic_config.routes %}
      - to: "{{ route }}"
        via: "{{ nic_config.gateway }}"
{% endfor %}
{% endif %}
{% endfor %}
{% endif %}
```

Key points:

- `ansible_host` and `gateway` come from the host's inventory entry
- DNS servers are dynamically pulled from hosts in the `dns` group
- The DNS `search` domain is customer-specific (varies; some customers omit it)
- Secondary-interface naming varies by customer (some use `ens19`, `ens20`, ...; see below)
- Each secondary IP can have its own mask (`mask_cidr`), gateway, and static routes

Primary Interface Configuration

The primary interface (eth0) is configured automatically using:

- `ansible_host` - The IP address
- `gateway` - The default gateway
- `mask_cidr` - Network mask (defaults to 24)

DNS servers are automatically set to:

- Hosts in the `dns` group (uses their `ansible_host` IPs)
- Falls back to `8.8.8.8` if the host is itself a DNS server

Secondary Interfaces

For hosts requiring additional network interfaces (public IPs, peering, etc.), use the `secondary_ips` configuration.

Schema

```
secondary_ips:
  <logical_name>:
    ip_address: <ip_address>
    mask_cidr: <prefix_length>   # Optional - per-NIC mask,
defaults to 24
    gateway: <gateway_ip>
    host_vm_network: <proxmox_bridge>
    vlanid: <vlan_id>
    nic_name: <interface_name>   # Optional - explicit interface
name override
                                 # (supported by templates that
honour it)
    routes:                      # Optional - static routes via
this interface
      - '<destination_cidr>'
      - '<destination_cidr>'
    table_routes:                # Optional - advanced, template-
specific
      - { to: '<cidr>', via: '<gateway>', table: <table_id> }
    routing_policy:              # Optional - advanced, template-
specific
      from: '<source_cidr>'
      via: '<gateway>'
      table: <table_id>
      priority: <priority>       # Optional, defaults to 100
```

> The `mask_cidr` key sets the prefix length for that one secondary interface. It is read from inside the secondary IP entry (`nic_config.mask_cidr`), not from a top-level `mask_cidr`. If omitted it defaults to `/24`.

> `table_routes` and `routing_policy` enable policy/source routing but are only rendered by templates that implement them. Advanced routing is

> template-specific — confirm the customer's `netplan.yaml.j2` honours these keys before using them.

Interface Naming

Secondary-interface naming is template-specific - it is not standardised across customers:

- Some customers name interfaces by index: `ens19`, `ens20`, `ens21`, ... (`18 + loop.index`). This matches the interface names assigned by Proxmox when adding additional NICs to a VM.
- Other customers use an explicit `nic_config.nic_name` when set, falling back to an index-based name (e.g. `ens(34 + loop.index)`) otherwise.

If a template honours `nic_name`, you can pin a secondary interface to a specific name by setting `nic_name` on that secondary IP entry; otherwise the template's index-based scheme applies.

Example Configuration

```yaml
dra:
  hosts:
    <hostname>:
      ansible_host: 10.0.1.100
      gateway: 10.0.1.1
      host_vm_network: "ovsbr1"
      vlanid: "100"
      netplan_config: netplan.yaml.j2
      secondary_ips:
        public_ip:
          ip_address: 192.0.2.50
          mask_cidr: 28
          gateway: 192.0.2.1
          host_vm_network: "vmbr0"
          vlanid: "200"
          routes:
            - '198.51.100.0/24'
            - '203.0.113.0/24'
        peering_ip:
          ip_address: 172.16.50.10
          gateway: 172.16.50.1
          host_vm_network: "ovsbr2"
          vlanid: "300"
          routes:
            - '172.17.0.0/16'
```

Generated Netplan Output

Rendered with the example template, the above configuration generates (the DNS `search` domain and `ens19`/`ens20` naming are customer-specific):

```
network:
  version: 2
  ethernets:
    eth0:
      addresses:
      - "10.0.1.100/24"
      nameservers:
        addresses:
        - 10.0.1.53
        search:
        - customer.example
      routes:
      - to: "default"
        via: "10.0.1.1"
    ens19:
      addresses:
      - "192.0.2.50/28"
      routes:
      - to: "198.51.100.0/24"
        via: "192.0.2.1"
      - to: "203.0.113.0/24"
        via: "192.0.2.1"
    ens20:
      addresses:
      - "172.16.50.10/24"
      routes:
      - to: "172.17.0.0/16"
        via: "172.16.50.1"
```

The `public_ip` interface renders as `/28` because its entry set `mask_cidr: 28`, while `peering_ip` falls back to the `/24` default.

Proxmox Integration

When using the `proxmox.yml` playbook, secondary NICs are automatically created on the VM:

1. **New VMs**: Secondary NICs are added during initial provisioning
2. **Existing VMs**: Secondary NICs are added and the VM is rebooted to apply changes

The Proxmox configuration uses:

- `host_vm_network` - The bridge to attach the NIC to
- `vlanid` - VLAN tag for the interface

How It Works

1. Existing netplan configs in `/etc/netplan` are backed up (`*.bak`) then removed to prevent conflicts
2. The customer's Jinja2 template renders to `/etc/netplan/01-netcfg.yaml` (mode `0600`)
3. `netplan try --timeout 30` applies the config with automatic rollback if connectivity is lost
4. `wait_for_connection` confirms the host stayed reachable
5. `netplan apply` commits the configuration

Common Use Cases

Diameter Edge Agent (DEA) with Public IP

```
<hostname>:
  ansible_host: 10.0.1.100           # Internal management IP
  gateway: 10.0.1.1
  netplan_config: netplan.yaml.j2
  secondary_ips:
    diameter_roaming:
      ip_address: 192.0.2.50         # Public IP for roaming
partners
      gateway: 192.0.2.1
      host_vm_network: "vmbr0"
      vlanid: "200"
      routes:
        - '198.51.100.0/24'          # Roaming partner network
```

PGW with S5/S8 Interface

```
<hostname>:
  ansible_host: 10.0.2.20            # Internal IP
  gateway: 10.0.2.1
  netplan_config: netplan.yaml.j2
  secondary_ips:
    s5s8_interface:
      ip_address: 203.0.113.17       # Public S5/S8 IP
      gateway: 203.0.113.1
      host_vm_network: "vmbr0"
      vlanid: "50"
```

Multi-homed Server with Separate Management and Data Networks

```
<hostname>:
  ansible_host: 10.0.1.100           # Management network
  gateway: 10.0.1.1
  netplan_config: netplan.yaml.j2
  secondary_ips:
    data_network:
      ip_address: 10.0.2.100         # Data network
      gateway: 10.0.2.1
      host_vm_network: "ovsbr2"
      vlanid: "200"
    backup_network:
      ip_address: 10.0.3.100         # Backup network
      gateway: 10.0.3.1
      host_vm_network: "ovsbr3"
      vlanid: "300"
```

Referencing Secondary IPs in Templates

You can reference secondary IP addresses in other Jinja2 templates and configuration files.

On the Same Host

When configuring a service on the same host that has secondary IPs, you can reference directly or use `inventory_hostname`:

```
# Reference directly (simplest)
{{ secondary_ips.diameter_public_ip.ip_address }}

# Or explicitly via inventory_hostname (same result)
{{ hostvars[inventory_hostname]['secondary_ips']
['diameter_public_ip']['ip_address'] }}

# Access other properties
{{ secondary_ips.diameter_public_ip.gateway }}
{{ secondary_ips.diameter_public_ip.vlanid }}
```

From Another Host

When you need to reference a *different* host's secondary IP (e.g., configuring a peer connection), use `hostvars` with the target hostname:

```
# Reference first host in dra group
{{ hostvars[groups['dra'][0]]['secondary_ips']
['diameter_public_ip']['ip_address'] }}

# Loop through all DRA hosts and get their public IPs
{% for host in groups['dra'] %}
{% if hostvars[host]['secondary_ips'] is defined %}
  - {{ hostvars[host]['secondary_ips']['diameter_public_ip']
['ip_address'] }}
{% endif %}
{% endfor %}
```

Example: DRA Peer Configuration

Configure a diameter peer to bind to its own public IP:

```
# In dra_config.yaml.j2 - use inventory_hostname for the current
host
peers:
  - name: external_peer
    # Bind to this host's public diameter IP
    local_ip: {{ hostvars[inventory_hostname]['secondary_ips']
['diameter_public_ip']['ip_address'] }}
    remote_ip: 198.51.100.50
    port: 3868
```

Checking if Secondary IPs Exist

Always check if the variable exists before using it:

```
{% if secondary_ips is defined and
secondary_ips.diameter_public_ip is defined %}
public_ip: {{ secondary_ips.diameter_public_ip.ip_address }}
{% else %}
public_ip: {{ ansible_host }}
{% endif %}
```

Troubleshooting

Verify Interface Names

SSH to the VM and check interface names:

```
ip link show
```

Expected output for a VM with two secondary interfaces:

```
1: lo: <LOOPBACK,UP,LOWER_UP> ...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
3: ens19: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
4: ens20: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
```

Check Netplan Configuration

```
cat /etc/netplan/01-netcfg.yaml
```

Apply Netplan Manually

```
netplan apply
```

Debug Netplan

```
netplan --debug apply
```

Verify Routes

```
ip route show
```

Related Documentation

- Hosts File Configuration - Host inventory setup
- Proxmox VM/LXC Deployment - VM provisioning
- Configuration Reference - All configuration variables

OmniHSS Replication & Node Operations

OmniHSS stores its subscriber, authentication (AuC), and IMS data in a local PostgreSQL database on each node. Multiple OmniHSS nodes are kept in sync with **bidirectional PostgreSQL logical replication**, forming an **active-active mesh**: every node is a full read/write primary, and a write on any node propagates to all the others. There is no primary/standby — any node can serve S6a, Cx/Dx, and Sh traffic and accept provisioning at any time.

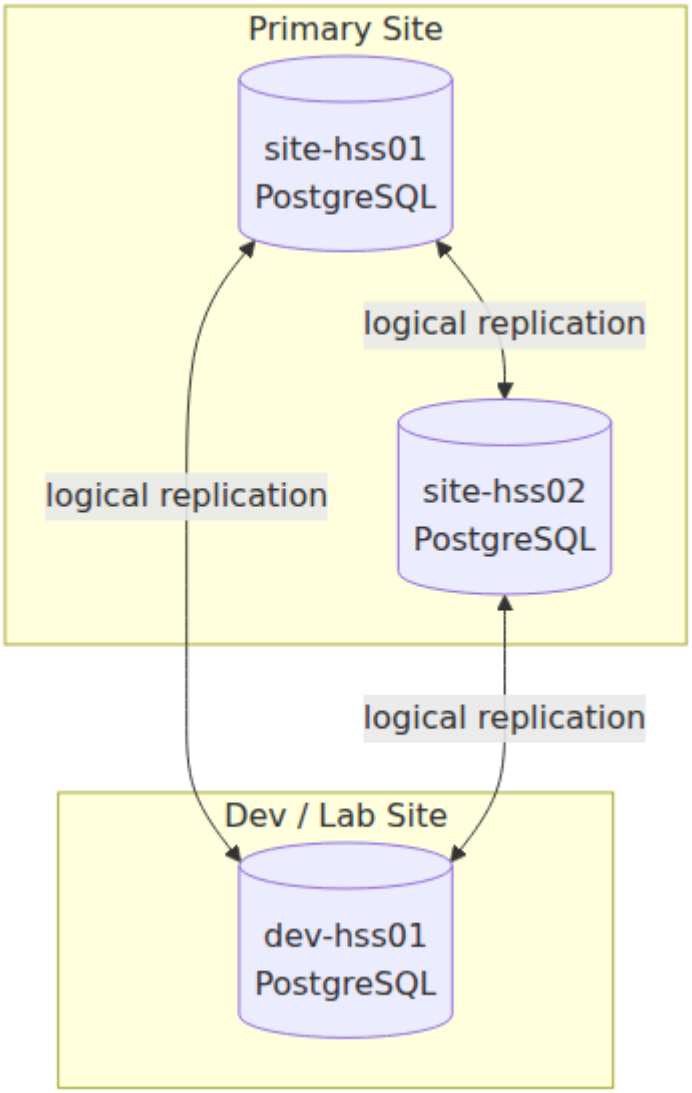
This guide covers the replication model, how to add a new OmniHSS node and load it with data, and how to validate, repair, and remove peers.

Table of Contents

Architecture Overview

Each OmniHSS node runs its own PostgreSQL instance. Every node **publishes** all of its application tables and **subscribes** to every peer's publication.

Replication loops are prevented with `origin = none`, and write conflicts on the fast-changing state tables are resolved **last-write-wins (LWW)** by comparing `updated_at`.

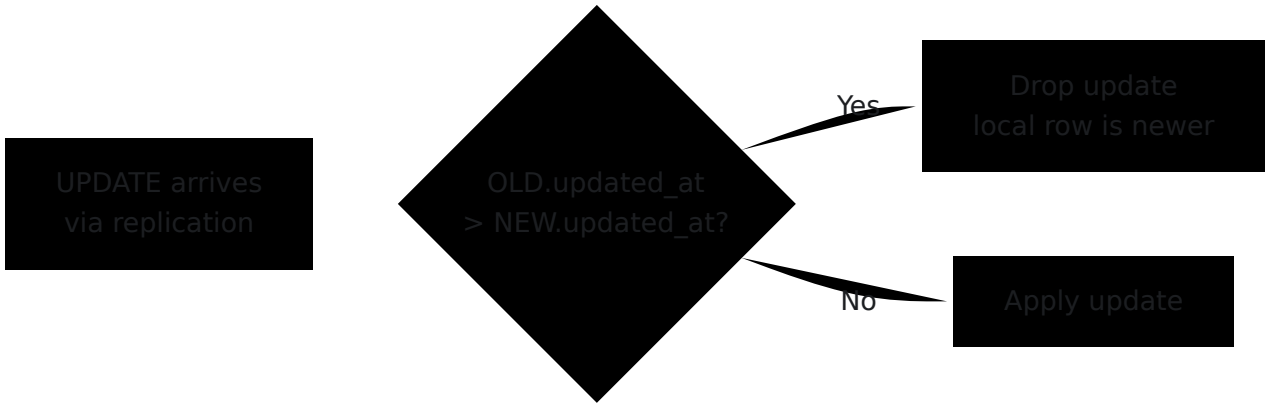


Each link above is two one-way subscriptions (one per direction). For a mesh of `N` nodes, every node holds `N-1` subscriptions and `N-1` of its publication's replication slots are consumed by peers.

What Gets Replicated

Table category	Examples	Conflict handling
Master / provisioned data	`subscriber`, `key_set` (AuC), `sim`, `msisdn`	None needed — provisioned from one place at a time
Dynamic state	`subscriber_state`, `pdn_session`, `lte_call`	Last-write-wins trigger on `updated_at`
Migration bookkeeping	`schema_migrations`	**Excluded** from replication

The publication covers every table in the `public` schema **except** `schema_migrations` (each node runs its own migrations independently). The LWW triggers ensure that when the same state row is updated on two nodes concurrently, the newest `updated_at` wins rather than whichever update arrived last.



How Peers Are Discovered

When replication is enabled, the OmniHSS role builds its peer list automatically from two sources:

1. **Other hosts in the local `hss` inventory group** — the OmniHSS nodes defined in the *same* inventory/host file (for example a local `hss01` / `hss02`

pair). These do **not** need to be listed in `connected_omnihss`; group membership is enough.

2. **The `connected_omnihss` list** in the site's remote peers file — OmniHSS nodes defined in a *different* host file (other sites). Cross-file peers are only discoverable here, because one inventory has no knowledge of another's `hss` group.

> **Note:** The legacy `connected_sites.hss` list is for the older HSS software and is **not** used by OmniHSS. OmniHSS replication peers come only from the local `hss` group plus `connected_omnihss`.

Mixed groups (legacy + OmniHSS): local-group peers are filtered to hosts that are OmniHSS replication participants — those whose `omnihss.replication.enabled` is `true`. If an `hss` group also contains legacy HSS boxes (different software), set the `omnihss` block **per-host** on the OmniHSS nodes only (a YAML anchor keeps it DRY) rather than as a group var, so the legacy hosts are not picked up as replication peers. A homogeneous OmniHSS group can set it as a group var.

The role excludes the current host from its own peer list automatically, so the same `connected_omnihss` block can be shared by every site without a node trying to replicate from itself.

Object Naming

Each node names its replication objects so that a mesh of any size never collides:

Object	Name	Created on	Unique per
Publication	`<self>_pub`	this node	node
Subscription	`sub_from_<source>`	this node (subscriber)	(subscriber)
Replication slot	`<subscriber>_from_<source>`	the source (publisher)	(subscriber, source) pair

The slot name **must** encode the subscriber. If it only encoded the source (PostgreSQL's default, where the slot inherits the subscription name), two nodes subscribing to the same source would both try to create a slot of the same name on that source, and the second subscription would fail with *"replication slot already exists"*. Encoding the subscriber gives every link a globally unique slot.

Reconciliation and `--limit` Safety

Every role run **reconciles** the node's replication objects against the current valid peer set (local `hss` group + `connected_omnihss`, minus self):

- Publications other than `<self>_pub` are dropped (cleans up stale/renamed publications automatically).
- Subscriptions whose peer is no longer in the valid set are disabled, detached from their remote slot, and dropped.
- Inactive logical slots that do not feed a current valid peer are dropped. Active slots are never touched.

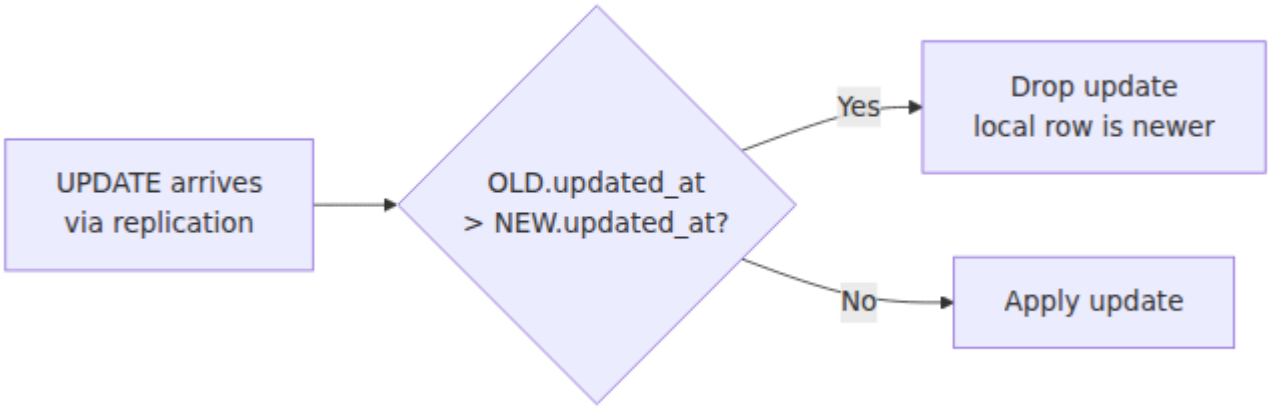
This makes `connected_omnihss` the **authoritative** mesh list: a peer removed from it (and from the relevant `hss` groups) is pruned from every node on the next run. Two safety properties keep this from being destructive:

- **`--limit` is safe.** The valid set is derived from `groups['hss']` and `connected_omnihss`, which always reflect *full* inventory membership regardless of `--limit`. Running `--limit hss01` reconciles only `hss01`, but

`hss01` still sees `hss02`/`hss03` as valid and keeps their subscriptions — `--limit` never prunes the excluded peers.

- **Empty sets never prune.** If the valid peer set computes to empty (e.g. a partial inventory with a single HSS and `connected_omnihss` not loaded), the prune steps are skipped entirely rather than wiping all replication.

> **Authoritative list caveat:** because pruning is automatic, a cross-file peer that is genuinely part of the mesh but missing from `connected_omnihss` will be pruned. Always keep `connected_omnihss` complete for every cross-file peer.



Configuration

Enabling Replication

Replication is controlled per deployment by the `omnihss.replication.enabled` flag. It is `false` by default. Set it in the site's inventory or group_vars:

```yaml
omnihss:
  database_type: "postgres"
  database_host: "localhost"
  database_port: 5432
  database_name: "omnihss"
  database_username: "hss"
  database_password: "password"
  replication:
    enabled: true
```

Parameter	Type	Required	Default	Descrip
`database_type`	String	No	`postgres`	Database back Replication app to `postgres`.
`database_host`	String	No	`localhost`	Database host Replication is c only when the is local (`locall`
`database_port`	Integer	No	`5432`	PostgreSQL po used as the de for peer conne
`database_name`	String	No	`omnihss`	Database nam match across a
`database_username`	String	Yes	`hss`	Login role. Cre the `REPLICATI` attribute so it c logical replicat
`database_password`	String	Yes	`password`	Password for th role. Must mat all peers (used connection stri
`replication.enabled`	Boolean	No	`false`	Master switch. `true`, PostgreS configured for replication and publications/su are created.

When `replication.enabled` is `true`, the role applies the following PostgreSQL settings on the local instance:

Setting	Value
`wal_level`	`logical`
`max_wal_senders`	`10`
`max_replication_slots`	`10`
`listen_addresses`	`*`
`max_slot_wal_keep_size`	`10GB` (override via `omnihss.replication.max_slot_wal_keep_size`)

Setting	Value

`pg_hba.conf` is updated to allow each peer's IP (`/32`) for both `all` and `replication` connections.

Declaring Cross-Site Peers (`connected_omnihss`)

Remote OmniHSS nodes are declared in the remote peers file referenced by each site's `remote_peers_file` variable (commonly `prod_master_peers.yml`):

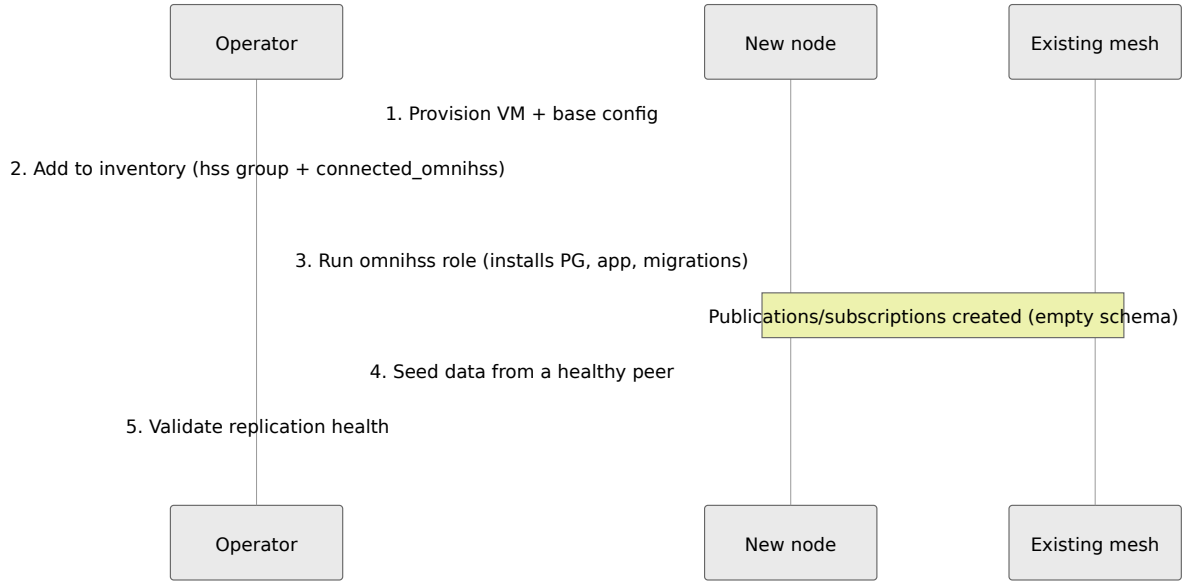
```
connected_omnihss:
  - name: site-hss01
    host: 10.4.2.142
    port: 5432
  - name: site-hss02
    host: 10.4.2.143
    port: 5432
  - name: dev-hss01
    host: 10.4.10.142
    port: 5432
```

Parameter	Type	Required	Default	Description
`name`	String	Yes	-	The peer's `inventory_hostname`. Used to name the subscription (`sub_from_<name>`) and to resolve the peer's publication (`<name>_pub`), so it **must** match the peer's actual inventory hostname.
`host`	String	Yes	-	The peer's IP address, reachable from this node over the signalling network.
`port`	Integer	No	`5432`	The peer's PostgreSQL port. Falls back to `database_port`.

Because the remote peers file is loaded by every site, the full `connected_omnihss` list should contain **every** OmniHSS node in the mesh. Each site automatically skips its own local nodes (they are already covered by the local `hss` group) and skips itself.

Adding a New OmniHSS Node

Adding a node has two distinct phases: **joining the mesh** (so changes flow in both directions) and **loading the existing data** onto the new node. The second phase matters because a brand-new node starts with an empty database — see Loading a New Node With Data.



Step 1 — Provision the host

Provision the VM and base networking as for any OmniCore node (see Proxmox Deployment). The node must reach every peer's PostgreSQL port (`5432` by default) over the signalling network, and peers must be able to reach it.

Step 2 — Add the node to inventory

Add the new host to the site host_file under the `hss:` group:

```
hss:
  hosts:
    site-hss01:
      ansible_host: 10.4.2.142
      gateway: 10.4.2.1
      host_vm_network: "v401-omni-signalling"
    site-hss03:        # new node
      ansible_host: 10.4.2.144
      gateway: 10.4.2.1
      host_vm_network: "v401-omni-signalling"
```

Then add it to `connected_omnihss` in the remote peers file so the **other sites** also learn about it:

```yaml
connected_omnihss:
  - name: site-hss03
    host: 10.4.2.144
    port: 5432
  # ... existing entries ...
```

Step 3 — Run the OmniHSS role

Run the OmniHSS service playbook against the new host. This installs PostgreSQL and OmniHSS, runs the database migrations (creating an **empty** schema), configures logical replication, and creates the publication plus a subscription to each peer:

```
ansible-playbook -i hosts/Customer_Name/host_files/Production.yml
\
  services/omnihss.yml --limit site-hss03
```

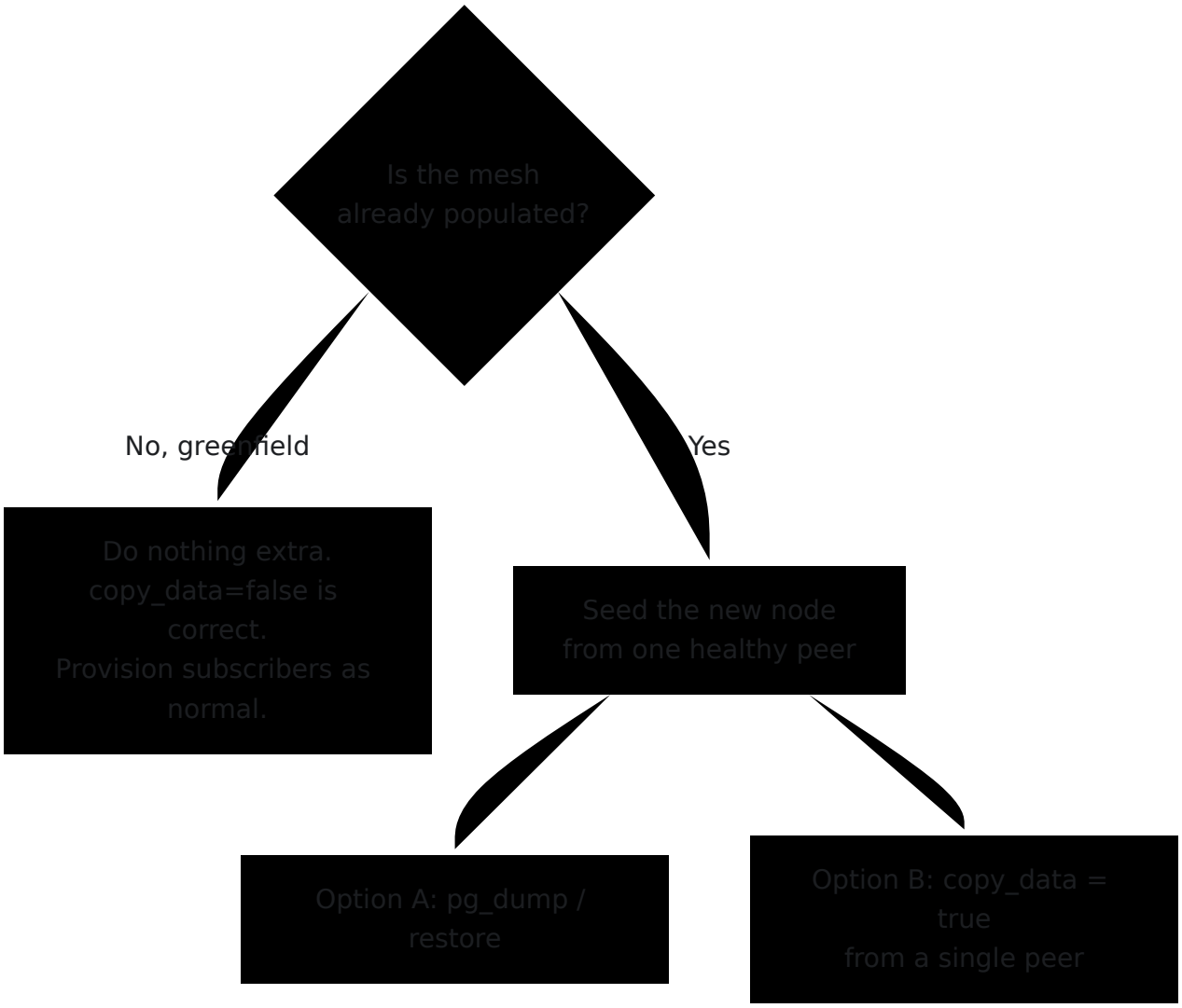
Re-run the role on the existing peers (or let the next scheduled run handle it) so they open `pg_hba.conf` for the new node and create their subscription back to it. Subscriptions are idempotent — existing ones are detected and skipped, and `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` is issued so any new tables are picked up.

At this point the new node is a member of the mesh and will receive **all future writes**, but it does **not** yet contain the existing subscriber base. Proceed to load its data.

Loading a New Node With Data

The automated subscriptions are created with `copy_data = false`. This is correct for a **greenfield mesh** (every node starts empty at the same time — there is nothing to copy, and copying would cause duplicate-row conflicts), but it means a node joining an **already-populated** mesh stays empty except for changes that occur after it joins.

Choose the path that matches your situation:



Greenfield Mesh

If you are standing up all OmniHSS nodes together with no pre-existing data, no seeding is required. Enable replication, run the role across all nodes, and provision subscribers normally — every write replicates to all members.

Option A — Seed via dump and restore (recommended)

Take a snapshot of a healthy peer and restore it onto the new node, then let replication carry ongoing changes. This reuses the standard backup/restore tooling and is the most predictable approach, including for large datasets.

1. **Back up a healthy peer.** This produces a `pg_dump` of the OmniHSS database (taken with `--no-publications --no-subscriptions`, so

replication objects are not carried over) on the control machine:

```
ansible-playbook -i
hosts/Customer_Name/host_files/Production.yml \
  services/backup.yml --limit site-hss01
```

The dump is saved as `backups/<Site_Name>/hss_dump_<hostname>_<timestamp>.sql`.

2. **Restore onto the new node.** The restore playbook stops OmniHSS, drops and recreates the database, restores the dump, and restarts the service. It takes a pre-restore backup of the target first:

```
ansible-playbook -i
hosts/Customer_Name/host_files/Production.yml \
  util_playbooks/restore_hss.yml --limit site-hss03
```

When prompted, supply the path to the SQL dump from step 1 and leave the config path empty (the new node already has its own config from the role run).

3. **Confirm replication is flowing.** The new node's existing subscriptions (`copy_data = false`) now apply changes on top of the restored snapshot. Run the validation below.

> **Maintenance window:** There is a small window between the dump and the moment replication catches up where provisioning changes on peers are not yet on the new node. Master subscriber data changes infrequently, and dynamic state (`subscriber_state`, `pdn_session`, `lte_call`) self-heals from live traffic, so a low-activity window is sufficient. Avoid provisioning new subscribers during the seed.

Option B — Seed with the seed playbook (recommended)

The `seed_hss_replication.yml` utility playbook seeds from **one** chosen source peer without the pitfalls of a raw bulk copy. It recreates the subscription as

streaming-only (`copy_data = false`, no tablesync) and then backfills the existing **master-table** rows with a conflict-safe load. `subscriber_state` **is** part of the backfill: it is structural 1:1 data (every subscriber references one by FK, created once at provisioning), and skipping it leaves every backfilled subscriber orphaned — breaking the API and S6a on the target — which streaming cannot heal, because logical replication silently drops UPDATEs to rows that don't exist. The truly transient session tables (`pdn_session`, `lte_call`) are deliberately **not** bulk-copied — a `COPY` of those hot tables collides (duplicate-key) with concurrent writes on a live system and crash-loops the tablesync; they repopulate from streaming replication and live traffic instead.

Interactive (recommended): omit the source and let the playbook find which targets are empty, probe the peers for ones holding data, and prompt you to pick:

```
ansible-playbook -i hosts/Customer_Name/host_files/Production.yml
\
  util_playbooks/seed_hss_replication.yml \
  --limit site-hss01,site-hss02
```

```
Peers with data available to seed from:
  1) dev-hss01 (10.4.10.142) - 838 subscribers
Enter the NUMBER of the source to seed from
```

Explicit: name the source directly (forces a re-seed even if the target already has data — use for recovery):

```
ansible-playbook -i hosts/Customer_Name/host_files/Production.yml
\
  util_playbooks/seed_hss_replication.yml \
  --limit site-hss01 \
  -e "seed_source_name=dev-hss01 seed_source_host=10.4.10.142"
```

See HSS Replication Seed for the full parameter and behaviour reference.

Always `--limit` to the target node(s), and seed from exactly one source. Prefer Option A for very large datasets.

The equivalent manual steps (run on the target node) — create the streaming subscription (note the explicit per-link `slot_name`, see Object Naming), then backfill the master tables conflict-safely while skipping the hot state tables:

```
-- 1. stream all tables, no bulk copy
DROP SUBSCRIPTION IF EXISTS sub_from_dev_hss01;
CREATE SUBSCRIPTION sub_from_dev_hss01
  CONNECTION 'host=10.4.10.142 port=5432 dbname=omnihss user=hss
password=<omnihss.database_password>'
  PUBLICATION dev_hss01_pub
  WITH (copy_data = false, origin = none,
        slot_name = 'site_hss01_from_dev_hss01');
```

```
# 2. backfill master tables only (run on the target; -t every
table except
#    schema_migrations, pdn_session, lte_call — subscriber_state
MUST be
#    included, or every backfilled subscriber is orphaned: its
#    subscriber_state_id points at a row that never arrives,
breaking the
#    API/S6a, and streaming cannot heal it because logical
replication
#    silently drops UPDATEs to missing rows).
#    The load MUST be stamped with a replication origin:
origin=none
#    subscriptions filter stamped changes at the publisher, so the
backfill is
#    invisible to the mesh. An unstamped load is decoded from WAL
like any
#    local write and echoes the inserts to peers that already hold
the rows,
#    duplicate-key-stalling their apply workers.
sudo -u postgres psql -d omnihss -c \
  "SELECT pg_replication_origin_create('omnihss_seed_load');"
{ echo "SELECT
pg_replication_origin_session_setup('omnihss_seed_load');"
  PGPASSWORD=<pw> pg_dump -h 10.4.10.142 -U hss -d omnihss \
    --data-only --inserts --on-conflict-do-nothing --disable-
triggers \
    -t subscriber -t subscriber_state -t key_set -t sim -t msisdn
... ; } \
  | sudo -u postgres psql -d omnihss
sudo -u postgres psql -d omnihss -c \
  "SELECT pg_replication_origin_drop('omnihss_seed_load');"
```

Validating Replication

The role runs a health check automatically after setup, and it can also be run on its own. The validation asserts that:

- Every subscription has a live worker process.
- Every logical replication slot is active.
- WAL lag is under 100 MB per slot.

- Each slot's WAL status is `reserved` or `extended` (not `lost`).

To check health on demand, re-run the OmniHSS role (the validation runs at the end), or inspect directly on a node:

```
# Subscription workers — pid should be non-null for each
sudo -u postgres psql -d omnihss -c \
  "SELECT subname, pid IS NOT NULL AS worker_alive FROM
pg_stat_subscription;"

# Replication slots — active should be true, wal_status
reserved/extended
sudo -u postgres psql -d omnihss -c \
  "SELECT slot_name, active, wal_status, \
     pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),
confirmed_flush_lsn)) AS lag \
   FROM pg_replication_slots WHERE slot_type = 'logical';"
```

Symptom	Meaning
`worker_alive = false`	Subscription worker is down — the peer is unreachable or the credentials/connection string are wrong.
`active = false` (slot)	No subscriber is consuming the slot — the peer's subscription is down or was dropped.
Growing `lag`	The peer is slow or the link is degraded; WAL is accumulating and consuming disk.
`wal_status = lost`	The slot fell too far behind and WAL was removed. The peer must be **re-seeded** (see Loading a New Node With Data).

The health check report (`health_check.yml`) also includes an **HSS Postgres Replication** panel for nodes with replication enabled, showing each subscription's worker state and each slot's active state and WAL status at a glance.

Data Integrity

Logical-replication apply does not enforce foreign keys. The apply worker runs with `session_replication_role = replica`, which suppresses the internal RI triggers that implement `FOREIGN KEY` constraints. A row that arrives via replication is therefore **never re-checked** against its parent. FK constraints give you *per-node* integrity, not *mesh-wide* integrity.

In an active-active mesh that opens a hole no single node can catch:

- Node A deletes `epc_profile` X — its `ON DELETE RESTRICT` passes because A has no local subscriber referencing X at that instant.
- Node B (or a provisioning step) points subscriber 203 → X — passes because X still exists on B.
- Both changes replicate and apply without FK checks. Every node converges on "203 → X" **and** "X deleted": a dangling reference that no node's FK ever locally violated.

A dangling `subscriber.epc_profile_id` makes the HSS unable to build subscription data for that subscriber and it answers `DIAMETER_ERROR_USER_UNKNOWN (5001)` on ULR/AIR — even though the subscriber row exists and is enabled. The schema-before-data window (below) makes it more likely, because skipped transactions can split a parent's create/delete from the child's re-point.

Detection (not enforcement)

Enforcing FKs on the apply path would halt the mesh on any anomaly, so instead we **detect and alarm**. Each node installs a `SECURITY DEFINER` function, `omnihss_fk_orphan_count()`, that counts every single-column FK whose value points at a missing parent across all tables. `postgres_exporter` selects it and Grafana alarms when it is greater than zero (see Metrics).

Run the sweep by hand at any time:

```
sudo -u postgres psql -d omnihss -tAc "SELECT
omnihss_fk_orphan_count();"   # 0 = clean
```

To locate offending rows for a specific reference (example: subscribers pointing at a non-existent EPC profile):

```
SELECT s.imsi, s.epc_profile_id
FROM subscriber s
LEFT JOIN epc_profile e ON e.id = s.epc_profile_id
WHERE s.epc_profile_id IS NOT NULL AND e.id IS NULL;
```

Remediate by repointing the row to a valid parent (an `UPDATE` on an existing row replicates cleanly across the mesh) or removing it.

Preventing schema drift

The root trigger of the incident that motivated this section was a **schema migration applied on one node while its peers were still on the old schema**, then new data written against the new fields. The source published rows the old-schema peers could not apply → apply errors and **skipped transactions** → divergence.

- **Migrate the whole mesh together.** Apply migrations to every OmniHSS node before provisioning against new fields. Do not write new-schema data while any peer is behind.
- **Watch for drift.** `omnihss_schema_max_version` / `omnihss_schema_migration_count` are exported per node; compare them across the mesh — a node behind its peers is a half-applied migration.
- **Watch the error counters.** `omnihss_subscription_error_apply_error_count` / `_sync_error_count` climbing means the apply worker is failing (and possibly skipping) — the "HSS Replication Apply/Sync Errors" alert fires on this.

Removing a Peer

Removing a peer drops the subscription **from** that peer and removes its `pg_hba.conf` entry. Data already replicated remains; only future changes stop flowing. Removal must be confirmed explicitly.

On the node you are removing the peer *from*:

```
DROP SUBSCRIPTION IF EXISTS sub_from_<peer_name>;
```

Then remove the peer's line(s) from `pg_hba.conf` and reload PostgreSQL:

```
sudo -u postgres psql -c 'SELECT pg_reload_conf();'
```

The peer being removed still holds its subscription back to this node — remove it there too if the link should be fully torn down, and remove the node from `connected_omnihss` and the `hss` group so it is not re-created on the next role run.

Backup & Restore

OmniHSS backup and restore use standard PostgreSQL dumps:

Operation	Playbook	Notes
Backup	`services/backup.yml`	Produces `hss_dump_<host>_<ts>.sq` (database) and `hss_<host>_<ts>.tar.gz` (the `/etc/omnihss` config) under `backups/<Site_Name>/`. The dump excludes publications/subscriptions.
Restore	`util_playbooks/restore_hss.yml`	Prompts for the SQL dump and/or config archive. Stop OmniHSS, takes a pre-restore backup, drops/recreates the database, restores, and restarts.

See Utility Playbooks for the broader backup/restore workflow.

Migrating From Legacy HSS

When replacing the legacy HSS software (the entries under `connected_sites.hss`) or a MySQL-backed OmniHSS, data is moved over the REST API rather than by SQL replication, because the schemas differ:

- **MySQL-backed OmniHSS → PostgreSQL OmniHSS** and **PyHSS mesh alignment** are handled by the migration utilities shipped with the role (`roles/omnihss/files/`). These read entities from the source HSS API and write them to the target, mapping integer IDs to deterministic UUIDs so foreign-key relationships are preserved, with options to offset IDs (to merge into a site that already holds data) and to align AuC/subscriber IDs with the PyHSS production mesh.

Once data is in the target OmniHSS, ongoing sync between OmniHSS nodes uses the logical replication described above.

Metrics

Each OmniHSS node runs `postgres_exporter`, exposing PostgreSQL metrics (including replication) to Prometheus. Useful series for replication health:

Metric: `pg_replication_slots_active` **Type**: Gauge **Description**: Whether each logical replication slot has an active consumer. **Labels**:

- `slot_name` — the replication slot (one per subscribing peer)

Metric: `pg_stat_replication_*` / slot lag **Type**: Gauge **Description**: WAL position lag per slot — how far behind a consuming peer is.

Example queries:

```
# Any inactive logical replication slot (alert if > 0)
count(pg_replication_slots_active == 0)

# Subscriptions that should exist vs workers running can be cross-
checked
# against the validation playbook output during deployment.
```

OmniHSS custom metrics

`postgres_exporter` on each node also exports OmniHSS-specific replication-health and integrity series via a custom query file (`roles/omnihss/files/postgres_exporter_queries.yaml`), scraped under the `HSS Postgres` job. These back the alert rules in the **HSS Alerts** group (`roles/monitoring/templates/grafana/rules/`):

Metric	Type	M
`omnihss_subscription_worker_alive{subname}`	Gauge	1 if inb wor run
`omnihss_replication_slot_active{slot_name}`	Gauge	1 if con the
`omnihss_replication_slot_wal_status_code{slot_name}`	Gauge	0=r 1=e 2=u **3=l**
`omnihss_replication_slot_lag_bytes{slot_name}`	Gauge	WA reta the
`omnihss_subscription_error_apply_error_count{subname}`	Counter	Cur app erro
`omnihss_subscription_error_sync_error_count{subname}`	Counter	Cur tab erro

Metric	Type	M
`omnihss_integrity_fk_orphans`	Gauge	Rov dar fore (sh
`omnihss_schema_max_version` / `omnihss_schema_migration_count`	Gauge	App mig hig ma

See Monitoring & Observability for the metrics pipeline and dashboards.

Troubleshooting

New node stays empty after joining

Symptoms: Replication workers are alive and slots are active, but the new node has no (or few) subscribers.

Possible causes:

- The node joined an already-populated mesh and was never seeded — subscriptions use `copy_data = false`, so only post-join changes arrive.

Resolution:

1. Seed the node from a healthy peer — see Loading a New Node With Data.

Subscription worker not running

Symptoms: `pid IS NULL` for a subscription in `pg_stat_subscription`.

Possible causes:

- Peer unreachable on PostgreSQL port (firewall / routing).
- Peer's IP missing from this node's `pg_hba.conf`.
- Wrong credentials in the connection string (`database_username` / `database_password` differ between nodes).

Resolution:

1. Confirm TCP reachability to the peer on `5432`.
2. Verify the peer appears in `connected_omnihss` (or the local `hss` group) and re-run the role so `pg_hba.conf` is updated.
3. Ensure `database_username`/`database_password` are identical across all nodes.

Replication slot shows `wal_status = lost`

Symptoms: A slot's `wal_status` is `lost`; the corresponding peer is far behind.

Possible causes:

- The peer was down or unreachable long enough that required WAL was removed.

Resolution:

1. Re-seed the affected peer from a healthy node — see Loading a New Node With Data.

Duplicate-key errors during initial copy

Symptoms: Errors applying changes right after creating a subscription with `copy_data = true`.

Possible causes:

- The initial copy was run from more than one peer, copying the same rows twice.

Resolution:

1. Seed from **exactly one** peer with `copy_data = true`; keep all other subscriptions at `copy_data = false`.

Conflicting state updates between sites

Symptoms: A state row (e.g. `subscriber_state`) appears to "flap" between values set at different sites.

Possible causes:

- Concurrent updates to the same row at two sites.

Resolution: This is expected and handled — the last-write-wins trigger keeps the update with the newest `updated_at`. Confirm clocks are synchronised (NTP) across all sites so `updated_at` ordering is meaningful.

Proxmox VM/LXC Deployment

The majority of our customers run the OmniCore stack on Proxmox. A single playbook — `util_playbooks/proxmox.yml` — provisions both QEMU VMs and LXC containers based on a `deployment_type` toggle. The toggle can be set per site (on the `proxmoxServers` entry) and overridden per host, so a site can run mostly as LXC while individual hosts (e.g. the UPF) stay as full VMs.

We still continue to support VMware, HyperV and cloud (Vultr / AWS / GCP) for deployments.

See Also:

- Hosts File Configuration — define VMs/LXCs to deploy
- IP Planning Standard
- Netplan Configuration
- Deployment Architecture

LXC vs VM

LXC Containers:

- Lightweight, shares host kernel
- Fast startup, low overhead
- Limited isolation
- Cannot run custom kernels or kernel modules
- **Not suitable for production EPC/IMS deployments**
- **Cannot run UPF** (requires kernel modules/TUN devices)

VMs (KVM):

- Full virtualization with dedicated kernel
- Complete isolation, runs kernel modules / custom networking

- Higher resource overhead
- **Recommended for production**
- **Required for UPF deployments**

Use Cases:

- **VMs**: production sites, UPF, all network functions
- **LXC**: lab/test environments, lightweight services (apt-cache, monitoring, dns)

Choosing the Deployment Type

`deployment_type` is resolved for each host in this order:

1. **Per-host** `deployment_type` (set under the hostname) — wins if present.
2. **Per-site** `deployment_type` on the `proxmoxServers` entry the round-robin selects for the host.
3. **Default** `vm` if neither is set.

The common pattern is to set the site default on the `proxmoxServers` entry (e.g. `deployment_type: lxc`) and override only the hosts that must differ. The canonical example is a lab that runs everything as LXC except the UPF, which needs kernel modules / TUN and must be a VM:

```
all:
  vars:
    proxmoxServers:
      pve-node-01:
        deployment_type: lxc     # site default → everything is
LXC
        # ...

upf:
  hosts:
    site-upf01:
      ansible_host: 192.168.1.24
      deployment_type: vm         # per-host override → this host
is a VM
```

As a convention, keep the site default consistent and only override the exceptions — scattering VM and LXC across many hosts makes node placement depend on host index and is hard to reason about.

The playbook only hard-fails on type collisions with existing resources: if a resource with the target hostname already exists as the wrong type (e.g. an LXC exists but the host resolves to `deployment_type: vm`, or vice versa), it aborts with a clear error. See Type Guards.

Proxmox Setup

1. Create API Token

```
# Proxmox UI: Datacenter → Permissions → API Tokens
# Create token: root@pam!<TokenName>
# Copy the token secret (shown once)
# Uncheck the `Privilege Separation` box
```

2a. Create Cloud-Init VM Template (VM sites only)

Run this script on the Proxmox host. It downloads Ubuntu's cloud image and creates a clean template — cloud-init credentials and SSH keys are injected at clone time by `proxmox.yml` (see Notes below).

```
#!/bin/bash
set -e

TEMPLATE_ID=9000
IMAGE_URL="https://cloud-images.ubuntu.com/noble/current/noble-
server-cloudimg-amd64.img"
IMAGE="noble-server-cloudimg-amd64.img"

cd /var/lib/vz/template/iso
wget -N "$IMAGE_URL"

qm destroy $TEMPLATE_ID --purge 2>/dev/null || true

qm create $TEMPLATE_ID --name ubuntu-2404-template --memory 2048 -
-cores 2 --net0 virtio,bridge=vmbr0
qm importdisk $TEMPLATE_ID $IMAGE local-lvm
qm set $TEMPLATE_ID --scsihw virtio-scsi-pci --scsi0 local-
lvm:vm-${TEMPLATE_ID}-disk-0
qm set $TEMPLATE_ID --ide2 local-lvm:cloudinit
qm set $TEMPLATE_ID --boot c --bootdisk scsi0
qm set $TEMPLATE_ID --vga std
qm set $TEMPLATE_ID --agent enabled=1
qm template $TEMPLATE_ID
```

Notes:

- The template has no hardcoded users or passwords — `proxmox.yml` sets `ciuser`, `cipassword`, and `sshkeys` at clone time.
- Cloud-init user precedence: `proxmoxTemplateUser` (on the matching `proxmoxServers` entry) → first `local_users` key.
- Cloud-init password precedence: `proxmoxTemplatePassword` (on the matching `proxmoxServers` entry) → the cloud-init user (i.e., the value chosen above).
- SSH keys are injected from every `local_users.*.public_key` entry (key-based auth is the intended login path).
- Set `proxmoxTemplateUser` (and optionally `proxmoxTemplatePassword`) on each `proxmoxServers` entry whenever the template was prepared with a cloud-init user that is not the first `local_users` entry, so the right account is selected.

- `--vga std` ensures the Proxmox web console works.
- `-N` flag on wget only downloads if newer than the local copy.

Alternative: Manual Template from ISO

If cloud images aren't available or you need a custom install:

Step 1: Create VM via Web UI

- Create New VM → VM ID 9000, Name: ubuntu-2404-template
- OS: Upload Ubuntu Server ISO or use existing ISO
- System: Default (SCSI Controller: VirtIO SCSI)
- Disks: 32GB, Bus: SCSI
- CPU: 2 cores
- Memory: 2048 MB
- Network: VirtIO, bridge vmbr0
- Start VM and install Ubuntu Server

Step 2: Inside VM - Clean and prepare

```
# Install cloud-init
sudo apt update
sudo apt install cloud-init qemu-guest-agent -y

# Clean machine-specific data
sudo cloud-init clean
sudo rm -f /etc/machine-id /var/lib/dbus/machine-id
sudo rm -f /etc/ssh/ssh_host_*
sudo truncate -s 0 /etc/hostname
sudo truncate -s 0 /etc/hosts

# Clear bash history and shutdown
history -c
sudo poweroff
```

Step 3: Add Cloud-Init and Convert to Template

- Select VM → Hardware → Add → CloudInit Drive (select storage e.g., local-lvm)

- Hardware → Options → QEMU Guest Agent → Enable
- Right-click VM → Convert to Template
- Do **not** set cloud-init user/password on the template — `proxmox.yml` injects these at clone time from `proxmoxTemplateUser` / `proxmoxTemplatePassword` (with `local_users` fallback for the username).

2b. Download LXC OS Template (LXC sites only)

```
# On the Proxmox node shell:
pveam update
pveam download local ubuntu-24.04-standard_24.04-2_amd64.tar.zst
```

The resulting volume path (e.g. `local:vztmpl/ubuntu-24.04-standard_24.04-2_amd64.tar.zst`) is what you configure as `proxmoxLxcOsTemplate` below.

Hosts File Configuration

VM Deployment

```yaml
all:
  vars:
    # deployment_type omitted → defaults to "vm"
    proxmoxServers:
      pve-node-01:
        proxmoxServerAddress: 192.168.1.100
        proxmoxServerPort: 8006
        proxmoxApiTokenName: ansible
        proxmoxApiTokenSecret: "your-token-secret-uuid"
        proxmoxNodeName: pve-node-01
        proxmoxTemplateName: ubuntu-2404-template
        proxmoxTemplateId: 9000
        proxmoxTemplateUser: omnitouch        # optional cloud-
init username; defaults to the first local_users key
        proxmoxTemplatePassword: omnitouch    # optional cloud-
init password; defaults to the cloud-init username
        storage: local-lvm  # optional
      pve-node-02:
        # ... second node config

    local_users:
      admin_user:
        name: Admin User
        public_key: "ssh-rsa AAAA..."

mme:
  hosts:
    site-mme01:
      ansible_host: 192.168.1.10
  vars:
    proxmox_interface: vmbr0
    gateway: 192.168.1.1
    mask_cidr: 24  # optional, default 24
    vlanid: 100  # optional
```

LXC Deployment

```yaml
all:
  vars:
    proxmox_lxc_nameserver: "1.1.1.1"   # optional, default
1.1.1.1
    proxmoxServers:
      pve-node-01:
        deployment_type: lxc
        proxmoxServerAddress: 192.168.1.100
        proxmoxServerPort: 8006
        proxmoxApiTokenName: ansible
        proxmoxApiTokenSecret: "your-token-secret-uuid"
        proxmoxNodeName: pve-node-01
        proxmoxLxcOsTemplate: "local:vztmpl/ubuntu-24.04-
standard_24.04-2_amd64.tar.zst"
        proxmoxLxcDefaultStorage: local-lvm  # optional, rootfs
fallback
      pve-node-02:
        deployment_type: lxc
        # ... same shape

    local_users:
      admin_user:
        name: Admin User
        public_key: "ssh-rsa AAAA..."

dns:
  hosts:
    site-dns01:
      ansible_host: 192.168.1.20
  vars:
    proxmox_interface: vmbr0
    gateway: 192.168.1.1
    mask_cidr: 24                        # optional, default 24
    vlanid: 100                         # optional
    proxmoxLxcCores: 2                  # optional override
    proxmoxLxcMemoryMb: 4096            # optional override
    proxmoxLxcDiskSizeGb: 30            # optional override
    proxmoxLxcRootFsStorageName: local-lvm  # optional, overrides
server proxmoxLxcDefaultStorage
```

Usage

Provision (VM or LXC)

```
ansible-playbook -i hosts/Customer/site.yml
util_playbooks/proxmox.yml
```

Delete

```
ansible-playbook -i hosts/Customer/site.yml
util_playbooks/proxmox_delete.yml
```

The delete playbook handles both VMs and LXCs automatically, regardless of `deployment_type`.

Playbook Behavior

Type Guards

- If a VM with the target hostname exists but `deployment_type: lxc` → playbook fails with clear error.
- Inverse case (LXC exists, site configured as VM) → same.
- UPF group on an LXC site → warn-only (not a hard fail, but UPF will not function).

New Resource

- Round-robins across `proxmoxServers` entries by host index within group.
- Queries Proxmox API for next free VMID.
- **VM**: clones from `proxmoxTemplateId`, sets cloud-init user/password/SSH keys (cloud-init user = `proxmoxTemplateUser` from the server entry if set, otherwise the first `local_users` key; cloud-init password =

`proxmoxTemplatePassword` if set, otherwise the cloud-init username), resizes `scsi0`, starts.

- **LXC**: creates from `proxmoxLxcOsTemplate`, rootfs on `proxmoxLxcRootFsStorageName` (group) → `proxmoxLxcDefaultStorage` (server) → `storage` → `local-lvm` (final fallback), injects all `local_users.*.public_key` via `ssh-public-keys`, starts. Root password hardcoded to `0mn1Touch@`. All LXCs are unprivileged (`unprivileged: 1`) with nesting enabled (`features: nesting=1`).

Existing Resource

Both VM and LXC update paths diff current vs desired and apply changes:

- Network config (bridge, VLAN tag, IP for LXC)
- Cores / memory
- Disk size (VM `scsi0` / LXC `rootfs`)
- Secondary NICs from `secondary_ips` Reboots the resource if network or resources changed.

> **OmniUPF:** a `secondary_ips` entry named `n3` (or `data`/`gtp`) is treated as a dedicated user-plane NIC — the `omniupf` role binds N3/N9 + XDP to it and reserves a CPU for the primary interface, so heavy datapath load can't starve PFCP/API/SSH (which stay on the primary IP).

Tagging

Every resource gets tagged with its Ansible group names + `site_name`.

Distribution Across Nodes

Same round-robin logic for VMs and LXCs:

```
mme01 → pve-node-01 (0 % 3 = 0)
mme02 → pve-node-02 (1 % 3 = 1)
mme03 → pve-node-03 (2 % 3 = 2)
mme04 → pve-node-01 (3 % 3 = 0)
```

Per-Host Overrides

`deployment_type` and any of the group-level LXC knobs (`proxmoxLxcCores`, `proxmoxLxcMemoryMb`, `proxmoxLxcDiskSizeGb`, `proxmoxLxcRootFsStorageName`, `proxmoxLxcOsTemplate`, `host_vm_network`) can all be set per-host under the hostname.

```
dns:
  hosts:
    site-dns01:
      ansible_host: 192.168.1.20
      deployment_type: vm         # per-host override of the site
default
      proxmoxLxcDiskSizeGb: 120   # per-host override
      host_vm_network: vmbr1      # per-host bridge override
  vars:
    proxmox_interface: vmbr0
    gateway: 192.168.1.1
    mask_cidr: 24  # optional, default 24
```

Service Playbooks

Service playbooks deploy and configure OmniCore infrastructure. Located in the `services/` directory.

Playbook Hierarchy

Playbooks are structured hierarchically to enable fast, targeted deployments:

```
all.yml
├── setup_users.yml
├── apt_cache.yml          # only when apt_cache_servers group is
defined
├── dns.yml
├── common.yml
├── license_server.yml
├── monitoring.yml
│   └── grafana.yml
├── device_tester.yml
├── epc.yml
│   ├── common.yml
│   ├── omnimme.yml
│   ├── omnisgwc.yml
│   ├── omnipgwc.yml
│   ├── upf.yml
│   ├── omnihss.yml
│   └── omnidra.yml
├── ims.yml
│   ├── pcscf.yml
│   ├── icscf.yml
│   ├── scscf.yml
│   ├── as.yml
│   ├── omnimessage.yml
│   └── omnisep.yml
├── omniepdg.yml
├── omnitwag.yml
├── omniss7.yml
├── ocs.yml
├── crm.yml
├── ran_monitor.yml
├── 5gc.yml
└── ../util_playbooks/health_check.yml
```

Run only what you need. Deploying a single component (e.g., `omnimme.yml`) takes seconds. Running `all.yml` across 50 hosts takes longer but handles everything. Use `--limit` to narrow scope further.

Quick Reference

Top-Level Playbooks

Playbook	Scope	Description
`all.yml`	Full network	Complete deployment: users, DNS, monitoring, device tester, EPC, IMS, ePDG, TWAG, SS7, OCS, CRM, RAN monitor, 5G Core, and a final health check (from `util_playbooks/`)
`epc.yml`	Packet Core	MME, SGW-C, PGW-C, UPF, HSS, DRA
`ims.yml`	Voice/IMS	P/I/S-CSCF, Application Server, Messaging, OmniSEP (XCAP/Entitlements/BSF)
`ocs.yml`	Charging	CGrateS, KeyDB cluster, OCS sync
`monitoring.yml`	Observability	Prometheus, Grafana, exporters, HOMER

Infrastructure Playbooks

Playbook	Description
`common.yml`	Base OS config, packages, NTP, logging agents
`setup_users.yml`	Local user accounts and SSH keys
`dns.yml`	DNS server deployment
`license_server.yml`	OmniCore license server
`does_netplan_exist.yml`	Check whether netplan is installed on each host
`update_mtu.yml`	Set and apply interface MTU via netplan
`firewall.yml`	iptables/nftables rules
`apt_cache.yml`	Local APT mirror (if enabled)

EPC Components

Playbook	Component	Description
`omnimme.yml`	OmniMME	Mobility Management Entity (4G)
`omnisgwc.yml`	OmniSGW-C	Serving Gateway Control Plane
`omnipgwc.yml`	OmniPGW-C	PDN Gateway Control Plane
`upf.yml` / `omniupf.yml`	OmniUPF	User Plane Function (combined SGW-U/PGW-U)
`omnihss.yml` / `hss.yml`	OmniHSS	Home Subscriber Server
`omnidra.yml`	OmniDRA	Diameter Routing Agent
`omnitwag.yml`	OmniTWAG	Trusted Wireless Access Gateway
`omniepdg.yml`	OmniEPDG	Evolved Packet Data Gateway (WiFi Calling)
`gtp_proxy.yml`	GTP Proxy	GTP traffic proxy

IMS Components

Playbook	Component	Description
`pcscf.yml`	P-CSCF	Proxy Call Session Control Function
`icscf.yml`	I-CSCF	Interrogating CSCF
`scscf.yml`	S-CSCF	Serving CSCF
`as.yml`	OmniTAS	Telephony Application Server
`omnisep.yml`	OmniSEP	XCAP, Entitlement Server, BSF, Visual Voicemail
`xcap.yml`	OmniSEP	Legacy alias of `omnisep.yml` (deploys the same `omnisep` role)
`omnimessage.yml`	OmniMessage / IMS SMSC	SMS-over-IP controller, SMPP, and the IMS SMSC (deployed via the `cscf` role with `cscf_type: smsc_ims`)

5G Core

Deployed via `5gc.yml`. Each network function is installed by the shared `5gc_nf` role with a per-NF `nf_package`. See 5G Core for the role internals, per-NF config, version pinning, and UDM HNET/SUCI keys.

Playbook	Component	Description
`5gc.yml`	5G Core	Deploys all 5G Standalone network functions
	OmniNRF	Network Repository Function
	OmniAMF	Access and Mobility Management Function
	OmniAUSF	Authentication Server Function
	OmniBSF	Binding Support Function
	OmniCHF	Charging Function
	OmniNSSF	Network Slice Selection Function
	OmniPCF	Policy Control Function
	OmniSCP	Service Communication Proxy
	OmniSMF	Session Management Function
	OmniUDM	Unified Data Management
	OmniUDR	Unified Data Repository

Circuit-Switched

Deployed via `cs.yml` (roles under `circuit_switched/`). See Circuit-Switched Core for the BSC, MSC, and SS7 role details.

Playbook	Component	Description
`cs.yml`	Circuit-Switched	Deploys the legacy CS domain
	BSC	Base Station Controller (`circuit_switched/bsc`)
	OmniMSC	Mobile Switching Centre (`circuit_switched/omnimsc`)
	SS7	SS7/SIGTRAN stack (`omniss7` role)

Supporting Services

Playbook	Description
`ocs.yml`	Online Charging System (CGrateS + KeyDB)
`crm.yml`	Customer management portal (OmniCRM; configuration now lives in `group_vars/`)
`omniss7.yml`	SS7/SIGTRAN gateway
`homer.yml`	SIP/Diameter packet capture
`grafana.yml`	Grafana dashboards and alerting
`ran_monitor.yml`	RAN monitoring integration
`omniroam.yml`	OmniRoam roaming suite (TAP3/RAP UI, Elixir API, generator)
`omnilcs.yml`	OmniLCS Location Services and Cell Broadcast
`device_tester.yml`	OmniWeb phone agent test bench (Test Devices)

> **Proxmox provisioning** (`proxmox.yml`, `proxmox_delete.yml`) lives under `util_playbooks/` — see Utility Playbooks and Proxmox VM/LXC Deployment.

Operations

Playbook	Description
`backup.yml`	Backup databases and configs
`reboot.yml`	Controlled reboot of hosts
`shutdown.yml`	Graceful shutdown
`apt_update.yml`	Update packages
`apt_refresh_metadata.yml`	Refresh APT cache metadata
`speedtest.yml`	Network throughput testing

Restore Playbooks

Playbook	Description
`restore_applicationserver.yml`	Restore OmniTAS from backup
`restore_smsc.yml`	Restore SMSC from backup — OmniMessage Controller, SMPP gateway, and IMS SMSc nodes

> Note: `health_check.yml`, `restore_hss.yml`, and `restore_ocs.yml` are **not** in `services/` — they live in `util_playbooks/`. See Utility Playbooks. `all.yml` runs the health check as `../util_playbooks/health_check.yml`.

Usage

Deploy Everything

```
ansible-playbook -i hosts/customer/host_files/production.yml
services/all.yml
```

Deploy Specific Subsystem

```
# Just the packet core
ansible-playbook -i hosts/customer/host_files/production.yml
services/epc.yml

# Just IMS/voice
ansible-playbook -i hosts/customer/host_files/production.yml
services/ims.yml
```

Deploy Single Component

```
# Update only the MME
ansible-playbook -i hosts/customer/host_files/production.yml
services/omnimme.yml

# Update only monitoring
ansible-playbook -i hosts/customer/host_files/production.yml
services/monitoring.yml
```

Limit to Specific Hosts

```
# Run all.yml but only on one host
ansible-playbook -i hosts/customer/host_files/production.yml
services/all.yml --limit mme01

# Run on multiple specific hosts
ansible-playbook -i hosts/customer/host_files/production.yml
services/all.yml --limit "mme01,hss01"

# Run on a group
ansible-playbook -i hosts/customer/host_files/production.yml
services/all.yml --limit mme
```

Related Documentation

- Deployment Architecture - Overall deployment workflow
- Hosts File Configuration - Defining infrastructure
- Group Variables Configuration - Customization
- Utility Playbooks - Operational tools (health check, restore, etc.)
- Monitoring & Observability - Grafana, Prometheus, alerting

Utility Playbooks

Utility playbooks provide operational tools for managing deployed OmniCore infrastructure. These playbooks are located in the `util_playbooks/` directory and can be run independently to perform common maintenance and troubleshooting tasks.

Quick Reference

Playbook	Purpose
`proxmox.yml`	Provision VMs and/or LXC containers on Proxmox (per `deployment_type`)
`proxmox_delete.yml`	Delete VMs/LXCs on Proxmox
`proxmox_restart_hung.yml`	Force power-cycle VMs/LXCs via the Proxmox API when SSH is unresponsive
`vmware.yml`	Provision VMs on VMware vSphere
`vmware_delete.yml`	Delete VMs on VMware vSphere
`health_check.yml`	Generate comprehensive health report for all services
`restore_hss.yml`	Restore HSS database and/or configuration from backup
`seed_hss_replication.yml`	Seed an empty OmniHSS node from a populated peer (initial replication data load)
`audit_hss_drift.yml`	Audit the OmniHSS mesh for data drift (row count + content digest per reference table across all nodes); read-only
`reconcile_hss_drift.yml`	Reconcile OmniHSS mesh data drift by union-backfilling missing rows across nodes (dry-run by default)
`restore_ocs.yml`	Restore OCS KeyDB, MySQL, and configuration from backup

Playbook	Purpose
`ip_plan_generator.yml`	Generate network documentation with Mermaid diagrams
`get_ports.yml`	Audit open ports and listening services across all hosts
`getLocalCapture.yml`	Retrieve packet capture files from hosts
`delete_local_user.yml`	Remove a local user account from all hosts
`clear_mnesia.yml`	Clear the Mnesia database for OmniSS7 / OmniMessage and restart (destructive)
`delete_license.yml`	Delete the license file on the license server and restart it

VM Provisioning

VM provisioning playbooks create and manage virtual machines on hypervisor platforms. All playbooks support `--limit` to target specific hosts or groups.

See Proxmox Deployment for detailed configuration.

Proxmox

Files: `util_playbooks/proxmox.yml`, `util_playbooks/proxmox_delete.yml`

`proxmox.yml` provisions both VMs and LXC containers; whether a given host becomes a VM or an LXC is decided by its resolved `deployment_type` (per-host override → `proxmoxServers` entry → default `vm`). See Proxmox Deployment.

```
# Provision (VMs and/or LXC containers, per deployment_type)
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/proxmox.yml

# Provision specific group only
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/proxmox.yml --limit mme

# Delete VMs/LXCs
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/proxmox_delete.yml --limit old-host
```

Proxmox Restart Hung Hosts

File: `util_playbooks/proxmox_restart_hung.yml`

Checks each host's SSH responsiveness and force power-cycles any unresponsive VMs or LXC containers through the Proxmox API. QEMU VMs are hard-stopped and started; LXC containers are started (or rebooted if already running). Hosts that respond to SSH are skipped.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/proxmox_restart_hung.yml

# Restrict to a single host or group
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/proxmox_restart_hung.yml --limit hss01
```

VMware vSphere

Files: `util_playbooks/vmware.yml`, `util_playbooks/vmware_delete.yml`

Both playbooks include the per-VM task files under `util_playbooks/vm/vmware/` (`create_individual_vm.yml` and `delete_individual_vm.yml`). `vmware_delete.yml` is a thin wrapper around the delete task; `vmware.yml` additionally waits for SSH after provisioning, then removes stale netplan files and applies netplan on the new VM.

Prerequisites: Requires the Python dependencies to be installed (`uv sync`).

Usage:

```
# Provision VMs
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/vmware.yml

# Provision specific group only
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/vmware.yml --limit mme

# Delete VMs
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/vmware_delete.yml --limit old-host
```

Required Variables:

```yaml
all:
  vars:
    vcenter_ip: "vcenter.example.com"
    vcenter_username: "administrator@vsphere.local"
    vcenter_password: "password"
    vcenter_datacenter: "Datacenter"
    vcenter_folder: "OmniCore"
    vcenter_vm_template: "ubuntu-2404-template"
    vhosts:
      esxi-01:
        vcenter_cluster_ip: "192.168.1.10"
        vcenter_datastore: "datastore1"
```

Vultr

Files: `util_playbooks/vm/vultr/vmProvisionVultr.yml`, `util_playbooks/vm/vultr/vmTeardownVultr.yml`, `util_playbooks/vm/vultr/vmCommonSetup.yml`

Provisions and tears down instances on Vultr via the Vultr API. `vmProvisionVultr.yml` creates instances, `vmTeardownVultr.yml` removes

them, and `vmCommonSetup.yml` performs post-provision base setup (DNS, reboot, etc.). All three are top-level entry-point playbooks rather than task includes. They read the API token from the `VULTR_API_TOKEN` environment variable, and the provisioning playbook references a `cloud-config.yaml` from the same directory, so run them from `util_playbooks/vm/vultr/`.

```
export VULTR_API_TOKEN=<token>
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/vm/vultr/vmProvisionVultr.yml
```

Health Check

File: `util_playbooks/health_check.yml`

Generates a comprehensive HTML health report covering all deployed OmniCore and OmniCall services.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/health_check.yml
```

Output: `/tmp/health_check_YYYY-MM-DD HH:MM:SS.html`

Information Collected

Component	Data Collected
All services	Service status, version, uptime
OmniHSS	Database status, Diameter peer connections
OmniDRA	Diameter peer connections and status
OmniTAS	Active calls, sessions, registrations, CPU usage
OCS	KeyDB replication status

HSS Restore

File: `util_playbooks/restore_hss.yml`

Restores OmniHSS from backup files. Supports restoring database only, configuration only, or both.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/restore_hss.yml
```

Backup File Formats

Type	Filename Pattern	Contents
Database	`hss_dump_<hostname>_<timestamp>.sql`	PostgreSQL (pg_dump) dump of the OmniHSS database (default `omnihss`)
Config	`hss_<hostname>_<timestamp>.tar.gz`	Archive of `/etc/omnihss` directory

The current HSS state (database and/or config) is backed up before the restore runs.

Prompts

Prompt	Required	Description
HSS SQL dump path	No	Full path to the HSS SQL dump file (leave empty to skip database restore)
HSS config tar.gz path	No	Full path to the HSS config tar.gz file (leave empty to skip config restore)
Confirmation	Yes	Type `yes` to confirm the restore operation on the listed host(s)

HSS Replication Seed

File: `util_playbooks/seed_hss_replication.yml`

Performs the one-time initial data load for an OmniHSS node joining an already-populated active-active replication mesh. A new node comes up empty (subscriptions are created with `copy_data = false`); this playbook recreates the subscription to a single chosen source peer with `copy_data = true`, so PostgreSQL copies the existing data and then continues streaming with no gap.

It runs in one of two modes:

Interactive (default) — omit the `seed_source_*` variables. The playbook detects which limited target(s) are empty, probes the valid peers for ones that hold data, lists them with subscriber counts, and prompts you to choose a source. Only empty targets are seeded.

```
ansible-playbook -i hosts/customer/host_files/production.yml \
  util_playbooks/seed_hss_replication.yml \
  --limit site-hss01,site-hss02
```

```
Peers with data available to seed from:
  1) dev-hss01 (10.4.10.142) - 838 subscribers
Enter the NUMBER of the source to seed from
```

Explicit — pass the source directly to skip the menu. This **forces** a re-seed even if the target already holds data (use for recovery).

```
ansible-playbook -i hosts/customer/host_files/production.yml \
  util_playbooks/seed_hss_replication.yml \
  --limit site-hss01 \
  -e "seed_source_name=dev-hss01 seed_source_host=10.4.10.142"
```

Parameters

Parameter	Required	Default	Description
`seed_source_name`	No	(prompted)	Inventory hostname of the populated source peer. Used to name the subscription (`sub_from_<nam` and resolve its publication (`<name>_pub`). Omit for interactive mod
`seed_source_host`	No	(prompted)	IP address of th source peer, reachable on its PostgreSQL port Omit for interactive mod
`seed_source_port`	No	target's `omnihss.database_port` (5432)	PostgreSQL port the source peer

> Both `seed_source_name` and `seed_source_host` must be supplied together to use explicit mode; supplying neither triggers interactive mode.

Behaviour

1. Counts subscribers on each limited target to determine which are empty.
2. **Interactive mode:** probes the valid peers (local `hss` group + `connected_omnihss`) for ones holding data, lists them, and prompts for a choice. **Explicit mode:** prompts for `yes` to confirm.

3. Ensures the source IP is allowed in the target's `pg_hba.conf`.
4. Drops and recreates the subscription to the source `WITH (copy_data = false)` (per-link `slot_name` of `<target>_from_<source>`) — it streams **all** tables but performs no bulk tablesync.
5. Backfills the existing **master-table** rows from the source with a conflict-safe load (`pg_dump --data-only --inserts --on-conflict-do-nothing --disable-triggers`). `subscriber_state` **is** included — every subscriber references one by FK, so skipping it orphans every backfilled subscriber. The transient session tables (`pdn_session`, `lte_call`) are **not** backfilled.
6. Reports the subscriber count on the seeded node.

> **Why the session tables are skipped:** they are hot tables; a bulk `COPY` of them on a live system collides (duplicate-key) with concurrent writes and crash-loops the tablesync worker. They repopulate from streaming replication and live traffic instead. `subscriber_state` is different: rows are only INSERTed at provisioning time (1:1 with the subscriber) and only UPDATEd afterwards, so the `--on-conflict-do-nothing` backfill is safe — and without it the target's subscribers point at state rows that never arrive, since logical replication silently drops UPDATEs to missing rows.

> **Important:** Always `--limit` to the target node(s). In interactive mode, non-empty targets are skipped automatically. See OmniHSS Replication & Node Operations for the full workflow.

HSS Drift Audit

Files: `util_playbooks/audit_hss_drift.yml` (standalone) and `roles/omnihss/tasks/audit_drift.yml` (the shared implementation).

Reports **data drift** across the OmniHSS active-active mesh. Logical-replication subscriptions are created with `copy_data = false`, so data that diverged before the mesh link (seed-time differences) or via a direct/unreplicated write is never reconciled and never alarmed — replication shows 0 lag while the data differs, and per-site Prometheus can't see it because the mesh spans two sites.

The audit connects to every mesh member (local `hss` group + `connected_omnihss` peers) and, for each reference/config table, compares an

exact row count **and** an order-independent md5 of the full contents. It also compares `schema_migrations` (count + max version) to catch schema drift. Hot state tables (`subscriber_state`, `pdn_session`, `lte_call`) are excluded — they legitimately differ under live traffic. One structural **health check** is included on top of the cross-node comparison: `__orphaned_subscribers__`, the per-node count of subscribers whose `subscriber_state` row is missing. A non-zero count on **any** node fails the audit even when every node agrees — a seed that copies `subscriber` without `subscriber_state` orphans every node identically, which pure cross-node comparison can never see. The comparison is centralised on one runner that reaches every peer's Postgres, so a single run covers the whole cross-site mesh from either inventory, and it is `--limit` safe.

Runs automatically after every HSS deploy. At the end of the `omnihss` role (when replication is enabled) the same audit runs as a post-deploy consistency check, and by default **FAILS the play** if drift is found, so a broken mesh is impossible to miss during a deploy. It is non-interactive (no pause): the audit must be safe under the `free` strategy of the aggregate `services/all.yml` deploy, where a pause would error. In that aggregate deploy the in-role audit is suppressed and the audit instead runs once in a dedicated **linear** play at the end of `services/deploy_nfs.yml`. Behaviour is controlled by two role variables:

Variable	Default	Effect
`omnihss_audit_drift`	`true`	run the audit at the end of the role
`omnihss_audit_fail_on_drift`	`true`	fail the play on drift; set `false` for report-only (non-interactive cron/CI)

The audit is **read-only** either way. The standalone playbook defaults to report-then-**fail** on drift (so it can gate cron/CI):

```
# on demand
ansible-playbook -i hosts/<inv>/host_files/<site>.yml
util_playbooks/audit_hss_drift.yml
# report only, always pass
ansible-playbook -i ... util_playbooks/audit_hss_drift.yml -e
audit_fail_on_drift=false
# disable the auto-audit for a single deploy
ansible-playbook -i ... services/omnihss.yml -e
omnihss_audit_drift=false
```

HSS Drift Reconcile

File: `util_playbooks/reconcile_hss_drift.yml`

Brings divergent reference tables back into sync by making every node hold the **union** of the rows: for each drifted table it backfills the rows a node is missing from its peers, using the same conflict-safe primitive as the seed playbook (`pg_dump --data-only --inserts --on-conflict-do-nothing --disable-triggers`).

- **Non-destructive** — rows are only ever INSERTed (skipping conflicts). Nothing is deleted or updated. It resolves "a node is missing rows"; it does **not** resolve "same primary key, different values" or remove rows that should not exist — those need a human decision.
- **Interactive by default** — it scans, prints the plan, and **asks** (`yes` to proceed) before writing; anything else aborts with no changes. Use `-e dry_run=true` to look only, or `-e assume_yes=true` for non-interactive runs.
- Writes run locally on each target as the postgres superuser (for `--disable-triggers`); peers are only ever read, as the app user.
- **Symmetric union, not prod-authoritative** — running it against a site pulls *that* site's rows onto the others. It does **not** treat prod as the master, so running it against the Lab inventory would propagate Lab rows toward prod. If prod must be the source of truth, drive it from the prod inventory (or request the authoritative mode).

- **Cross-site:** a node can only be a *target* when it is in the play, so run this against **each** site's inventory to cover every node. The re-scan at the end reports anything that still needs a run.

```
# interactive (default) — asks before writing
ansible-playbook -i hosts/<inv>/host_files/<site>.yml
util_playbooks/reconcile_hss_drift.yml
# look only, never writes
ansible-playbook -i ... util_playbooks/reconcile_hss_drift.yml -e
dry_run=true
# non-interactive apply (CI)
ansible-playbook -i ... util_playbooks/reconcile_hss_drift.yml -e
assume_yes=true
```

> Run `audit_hss_drift.yml` first to see what drifted, and again after reconciling (against both inventories) to confirm convergence. See OmniHSS Replication & Node Operations.

OCS Restore

File: `util_playbooks/restore_ocs.yml`

Restores OCS (CGrateS) from backup files. Handles multi-master KeyDB replication by restoring to one node and allowing replication to sync to others.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/restore_ocs.yml
```

Restore Process

1. Stops CGrateS and KeyDB on all OCS nodes
2. Clears AOF files to prevent conflicts with restored data
3. Restores KeyDB RDB to first node, starts KeyDB, lets replication sync
4. Restores MySQL StoreDB (optional)
5. Restores `/etc/cgrates` configuration (optional)
6. Starts CGrateS on all nodes

Backup File Formats

Type	Filename Pattern	Contents
KeyDB DataDB	`keydb_dump_<hostname>_<timestamp>.rdb`	KeyDB RDB snapshot
MySQL StoreDB	`cgrates_dump_<hostname>_<timestamp>.sql`	MySQL dump of `cgrates` database
Config	`cgrates_<hostname>_<timestamp>.tar.gz`	Archive of `/etc/cgrates` directory

Prompts

Prompt	Required	Description
KeyDB RDB dump path	Yes	Full path to the KeyDB RDB backup file
MySQL SQL dump path	No	Full path to the CGrateS SQL dump (skip to preserve current StoreDB)
Config tar.gz path	No	Full path to the config backup (skip to preserve current config)

IP Plan Generator

File: `util_playbooks/ip_plan_generator.yml`

Generates network documentation from inventory, including:

- Host IP assignments (primary and secondary NICs)

- Network segment overview
- Interface connectivity diagrams (Diameter, GTP, PFCP, SIP, SS7)

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/ip_plan_generator.yml
```

Output Files

File	Format	Description
`/tmp/ip_plan_<customer>_<site>.md`	Markdown	Text-based documentation
`/tmp/ip_plan_<customer>_<site>.html`	HTML	Interactive diagram with filterable layers

Port Audit

File: `util_playbooks/get_ports.yml`

Audits all listening ports across the deployment and generates documentation.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/get_ports.yml
```

Output Files

File	Description
`/tmp/all_ports.csv`	CSV with hostname, IP, protocol, port, service
`./open_ports.rst`	reStructuredText table for Sphinx documentation

Data Collected

Field	Description
Hostname	Inventory hostname
IP	Host's `ansible_host` IP address
IP Version	IPv4 or IPv6
Transport	TCP or UDP
Port	Listening port number
Service	Process name

Local Capture Retrieval

File: `util_playbooks/getLocalCapture.yml`

Retrieves the two most recent packet capture files from each host's `/etc/localcapture` directory.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/getLocalCapture.yml
```

Output: `./localCapturePcaps/<hostname>/*.pcap`

User Management

File: `util_playbooks/delete_local_user.yml`

Removes a local user account from all hosts in the inventory.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/delete_local_user.yml
```

Prompt: Enter the username to delete when prompted.

Clear Mnesia Database

File: `util_playbooks/clear_mnesia.yml`

Stops the service, deletes its Mnesia database directory, and restarts it. Runs against the `ss7` group (OmniSS7) and the `omnimessage` group (OmniMessage).

> **Warning:** This is destructive. The Mnesia database is permanently deleted and rebuilt empty on restart.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/clear_mnesia.yml

# Limit to a single service or host
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/clear_mnesia.yml --limit ss7
```

Delete License

File: `util_playbooks/delete_license.yml`

Deletes `/etc/license_server/license.json` on the `license_server` group and restarts the license server.

```
ansible-playbook -i hosts/customer/host_files/production.yml
util_playbooks/delete_license.yml
```

Running Utility Playbooks

Basic Syntax

```
ansible-playbook -i <inventory_file> util_playbooks/<playbook>.yml
```

Common Options

Option	Description
`-i <inventory>`	Specify inventory file
`--limit <hosts>`	Limit to specific hosts or groups
`-v` / `-vv` / `-vvv`	Increase verbosity
`--check`	Dry run (show what would change)
`--diff`	Show file differences

Examples

```
# Run health check on production
ansible-playbook -i hosts/acme/host_files/production.yml
util_playbooks/health_check.yml

# Restore HSS on a specific host
ansible-playbook -i hosts/acme/host_files/production.yml
util_playbooks/restore_hss.yml --limit hss01

# Generate IP plan with verbose output
ansible-playbook -i hosts/acme/host_files/production.yml
util_playbooks/ip_plan_generator.yml -v
```