Hamac

SPÉCIFICATION

Spécification du provisioning

Profils de provisioning, bundles, écriture de l'image OS sur le disque et contrat du serveur de découverte : le provisionnement d'OS par PXE avec Hamac.

Version 0.2 · 2026-10-30 · BROUILLON

OCamlPro SAS — forge.ocamlpro.com/OSS/hamac — AGPL-3.0

Sommaire

1 Objectif **2**
2 Vue d’ensemble **3**
3 Profil de provisioning **4**
3.1 Bloc `os` 4
4 Bundles **6**
4.1 Fragment cloud-init 6
4.2 Règle de fusion cloud-init 6
5 Référencement depuis une infrastructure **7**
6 CLI Hamac **8**
7 Contrat du serveur de découverte **9**
7.1 Stockage 9
7.2 Points d’accès 9
7.3 Point d’accès des nœuds 9
8 Flux de bout en bout **10**
9 Hors périmètre (v0.2) **11**
10 Évolutions futures **12**
10.1 Court terme 12
10.2 Moyen terme : description de plus haut niveau 12
10.3 Long terme 12
11 Références **13**

> **NOTE**
> **Statut** : brouillon — fonctionnalité en cours d'implémentation (milestone 6 de `HAMAC_ROADMAP.md`). Ce document définit le contrat pour le provisionnement d'OS via PXE par Hamac.

SECTION 1

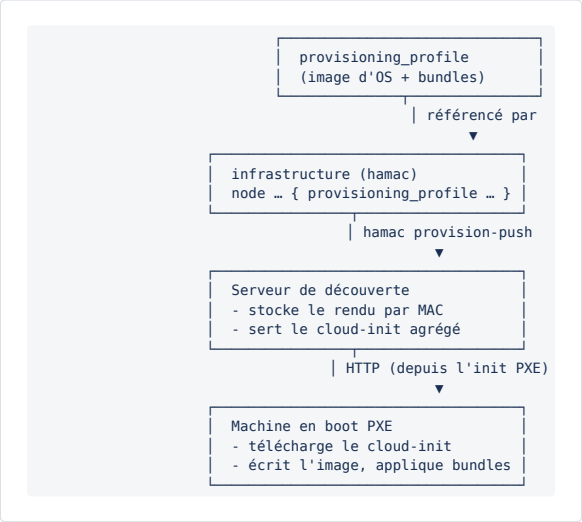
Objectif

Permettre à **Hamac** de provisionner un OS complet (base Linux + services préconfigurés) sur n'importe quelle machine qui démarre en PXE, qu'elle fasse partie de l'infrastructure du SI ou non (typiquement : poste de développeur).

Le provisionnement vit **entièrement côté Hamac** : manifestes LOAF, rendu en OCaml, serveur de découverte. Une couche de description de plus haut niveau reste possible (voir §10), mais n'entre pas dans le périmètre de cette spécification.

SECTION 2

Vue d’ensemble



SECTION 3

Profil de provisioning

Les manifestes `provisioning_profile` et `bundle` sont des blocs du vocabulaire LOAF de Hamac, définis par la spécification des manifestes (`doc/typst/manifest.typ`, sections « Bundle » et « Profil de provisioning ») : champs, valeurs par défaut, diagnostics. Ce document décrit la façon dont Hamac les applique.

Un profil s'écrit ainsi (`provisioning-profiles/dev-workstation.loaf`) :

```
loaf 0.1.
use hamac 0.2.

provisioning_profile dev_workstation {
  cloud_init_extra "{runcmd: [systemctl enable --now ssh]}".

  os "debian-12-amd64-genericcloud" {
    url "https://cdimage.debian.org/cdimage/cloud/bookworm/latest/debian-12-genericcloud-amd64.qcow
2".
    sha256 "0000000000000000000000000000000000000000000000000000000000000000".
    format qcow2.
    family debian.
  }

  bundles [dev_workstation].
}
```

Les paramètres des bundles ne figurent pas dans ce profil : ils viennent de l'enregistrement de la machine, à sa création. Un profil **PEUT** aussi les donner lui-même, dans un sous-bloc `bundle` par bundle de sa liste (`provisioning-profiles/dev-workstation-demo.loaf`) :

```
bundle dev_workstation {
  param username { value "devuser". }
  param timezone { value "Europe/Paris". }
  param ssh_public_keys {
    items ["ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIEXAMPLE example@hamac"].
  }
}
```

`value` donne un paramètre scalaire (`true` ou `false` pour un booléen), `items` un paramètre liste ; un paramètre `object` ne vient que de l'enregistrement de la machine. Un paramètre que le bundle ne déclare pas, ou donné sous une autre forme que celle de son type, donne `E007`. Un paramètre `required` du bundle qu'aucune source ne fournit fait échouer le rendu.

3.1 Bloc `os`

Les champs du sous-bloc `os` sont définis par la spécification des manifestes. Le `qcow2` désigné par `url` est l'image de référence ; `raw_zst_url`, `raw_zst_sha256` et `bmap_url` désignent une copie dérivée, utilisée pour écrire le disque d'une machine physique (ci-dessous). Avec le format `netinstall`, Hamac télécharge un noyau et un initrd Debian d'installation et lance une installation automatisée (preseed).

3.1.1 Écriture des images « bloc » (`qcow2` / `raw`) sur le disque

Un disque cible contient des **octets bruts** ; le `qcow2` est un format de *transport* (compressé, sparse) qui doit être « déplié » en raw avant d'être écrit. Hamac ne fait donc **pas** un `dd` du qcow2 tel quel :

- **`qcow2` = format canonique** d'un profil (source de vérité, aussi consommé tel quel par le chemin VM / `QemuCluster`) — `url`, `sha256` et `format` gardent le même sens quand `raw_zst_url` et `bmap_url` sont renseignés.
- Pour l'installation métal, Hamac sert **deux artefacts dérivés déterministes** du qcow2 : un **`raw.zst`** (`qemu-img convert -O raw` + compression zstd) et son **`.bmap`** (`bmaptool create` — carte des blocs non vides + sha256 par plage). Ils sont générés par `tools/discovery-prototype/pxe-build/prepare-os-assets.sh`, invoqué par la CI (`build:hamac-discovery-image`, conditionné à `provisioning-profiles/**`) et intégrés à l'image

`hamac-discovery` (`/usr/share/hamac/images`, **hors** du volume Swarm persistant — un chemin dans le volume ne serait jamais réécrasé par un nouveau build après le premier déploiement).

- L'init écrit via **`bmaptool`** (embarqué dans l'initramfs — python3 minimal + stdlib, pas de chaîne TLS complète) :

```
blkdiscard <device>
bmaptool copy <raw_zst_url> <device> --bmap <bmap_url>
```

`bmaptool copy` accepte des URL HTTP directement pour l'image *et* le `.bmap` — pas de téléchargement local préalable côté `pxe-build/init`. Il télécharge, décompresse à la volée, **n'écrit que les blocs mappés** et **vérifie le sha256 de chaque bloc** contre le `.bmap` — écriture minimale *et* intégrité forte, sans copie locale intermédiaire. C'est strictement plus robuste qu'un sha256 global sur un fichier d'abord téléchargé. `raw_zst_sha256` sert à l'audit et à l'outillage externe ; ce chemin ne l'utilise pas pour la vérification.

- **Repli** (si `raw_zst_url` / `bmap_url` sont absents, ou si l'initrd déployée ne connaît pas `bmaptool` — déploiement progressif) : `pxe-build/init` retombe silencieusement sur `url` / `format` / `sha256` — téléchargement local (`wget`) puis `qemu-img convert -O raw` (construit statiquement depuis les sources, pas le paquet apt : `qemu-utils` forcerait une vingtaine de paquets de dépendances transitives, TLS/PKCS#11 pour l'essentiel, hors de propos pour une conversion locale). Ce repli **vérifie activement le sha256** (`os_image_sha256`) du fichier téléchargé avant écriture.
  - Le pilote **curl** de qemu-img (streaming direct `qemu-img convert -f qcow2 -O raw -t none <URL> <device>`, sans copie locale intermédiaire) est construit et fonctionnellement vérifié dans le binaire statique — le CDN Debian répond correctement aux requêtes HTTP Range nécessaires — mais **n'est pas utilisé** : le qcow2 de référence (~330 Mo) tient sans souci dans le tmpfs de cette initrd, et le téléchargement local préserve la vérification sha256 active plutôt que de ne compter que sur TLS. À activer côté `pxe-build/init` si un profil sert une image sensiblement plus grosse ou vise du matériel avec peu de RAM.
  - Le format `raw` fourni directement par l'exploitant (sans `raw_zst_url` / `bmap_url`) suit le même chemin, sans conversion (juste `dd`).
- **Limite connue** : le serveur de découverte ne génère pas lui-même `raw.zst` + `.bmap` à la première demande d'un profil qui n'en a pas. Un profil sans `raw_zst_url` / `bmap_url` renseignés reste sur le chemin qemu-img, sans promotion automatique vers le chemin rapide et vérifié. Ce n'est pas bloquant : le repli fonctionne, simplement sans le raccourci bmaptool.
- `blkdiscard` préalable (sans `-f`, option que la version busybox embarquée ne connaît pas et dont l'écriture n'a pas besoin) pour que les zones non mappées relisent zéro en cas de reprovisionnement.
- **Redimensionnement au premier démarrage délégué à cloud-init** (`growpart` + `resize_rootfs`, activés par défaut sur les images cloud Debian, non désactivés par les gabarits `templates/bundles/*/cloud-init*.yaml.j2`) : relocalise la GPT, étend la dernière partition et agrandit le système de fichiers pour remplir un disque de taille quelconque — **pas d'outil de redimensionnement dans l'initramfs**.

SECTION 4

Bundles

Un bundle est un répertoire `<nom>/` qui contient son manifeste `bundle.loaf` et un gabarit de fragment cloud-init. Hamac cherche les bundles, dans cet ordre, dans :

1. le répertoire donné par l'option `--bundles-dir` ;
2. le répertoire que nomme la variable `HAMAC_BUNDLES_DIR` ;
3. `./templates/bundles/` ;
4. `/usr/share/hamac/bundles/`.

Le premier répertoire où un bundle apparaît l'emporte. Un profil nomme ses bundles : Hamac le charge avec les `bundle.loaf` de ces répertoires, dans un même ensemble de fichiers.

4.1 Fragment cloud-init

Le gabarit Jinja2 `cloud-init.yaml.j2` (ou sa variante `cloud-init.<family>.yaml.j2` pour la famille d'OS du profil, prise en priorité) produit un fragment de cloud-init. Il reçoit les valeurs des paramètres sous `params`.

```
{# cloud-init.yaml.j2 #}
users:
  - name: {{ params.username }}
    sudo: ALL=(ALL) NOPASSWD:ALL
    shell: /bin/bash
    ssh_authorized_keys:
{% for key in params.ssh_public_keys %}
      - {{ key }}
{% endfor %}
{% if params.dotfiles_repo %}
write_files:
  - path: /home/{{ params.username }}/.config/dotfiles-repo
    content: "{{ params.dotfiles_repo }}"
    owner: "{{ params.username }}:{{ params.username }}"
{% endif %}
```

4.2 Règle de fusion cloud-init

Quand plusieurs bundles produisent chacun un fragment cloud-init, Hamac les fusionne dans l'ordre :

1. fragments des bundles (ordre de la liste `bundles` du profil) ;
2. `cloud_init_extra` du profil.

Stratégie : pour les listes (`runcmd`, `packages`, `users`, etc.) → concaténation, les doublons de `packages` étant retirés. Pour les objets → fusion clé par clé. Pour les scalaires (`timezone`, `hostname`) → le dernier gagne. Pour les blocs `write_files` → concaténation ; deux fragments qui écrivent le même chemin avec des entrées différentes font échouer le rendu, en nommant le chemin et les deux fragments.

SECTION 5

Référencement depuis une infrastructure

Un nœud d'infrastructure (voir la spécification des manifestes) désigne le profil qui l'installe par son champ `provisioning_profile` :

```
loaf 0.1.
use hamac 0.2.

infrastructure dev_fleet {
  backend physical.

  node workstation {
    count 5.
    ram 16, gi.
    cpu 4.
    provisioning_profile dev_workstation.
  }
}
```

Le profil est un manifeste de l'ensemble de fichiers : l'infrastructure se valide avec lui et avec ses bundles (code `L002` du langage pour un profil absent). Si `provisioning_profile` est présent, Hamac **ignore** les champs `os_image` et `os_sha256` du nœud : le profil prend le dessus.

SECTION 6

CLI Hamac

```
# Valider des profils avec leurs bundles (champs, empreintes, paramètres)
hamac validate provisioning-profiles templates/bundles

# Afficher le cloud-init et le script iPXE d'un profil, sans les pousser
hamac provision-dryrun --profile=provisioning-profiles/dev-workstation-demo.loaf

# Rendre un profil et le pousser vers le serveur de découverte pour une MAC
hamac provision-push --profile=provisioning-profiles/dev-workstation.loaf \
  --mac=aa:bb:cc:dd:ee:ff [--discovery=http://…] [--bundles-dir=DIR]

# Retirer l'enregistrement d'une MAC
hamac provision-clear --mac=aa:bb:cc:dd:ee:ff [--discovery=http://…]
```

`--discovery` vaut par défaut `http://localhost:8877`.

SECTION 7

Contrat du serveur de découverte

7.1 Stockage

Le serveur de découverte persiste les enregistrements sur disque dans `$HAMAC_DISCOVERY_STATE/provisioning/<mac>.json` (par défaut `/var/lib/hamac-discovery/provisioning/`). Un redémarrage du serveur ne casse pas un démarrage PXE en cours.

Le contenu de chaque fichier est le **cloud-init final agrégé** (bundles + extra), pas le profil source. Tout le rendu Jinja2 a déjà eu lieu côté Hamac. Le serveur de découverte est volontairement « bête » : il ne sait pas ce qu'est un bundle, il sert juste un document YAML.

7.2 Points d'accès

Méthode	Chemin	Description
`POST`	`/provisioning/<mac>`	Hamac pousse un cloud-init prérendu pour cette MAC
`GET`	`/provisioning/<mac>`	L'init PXE récupère son enregistrement
`GET`	`/provisioning/<mac>/cloud-init`	le cloud-init seul, en YAML
`DELETE`	`/provisioning/<mac>`	Retire l'association MAC → profil
`GET`	`/provisioning`	Liste les associations (débogage, tableau de bord)

Format de `POST /provisioning/<mac>` :

```
{
  "profile_name": "dev_workstation",
  "cloud_init": "#cloud-config\n...",
  "ipxe_script": "#!ipxe\n...",
  "os_image_url": "https://...",
  "os_image_sha256": "...",
  "os_format": "qcow2",
  "os_raw_zst_url": "",
  "os_raw_zst_sha256": "",
  "os_bmap_url": ""
}
```

`profile_name` est le nom DNS du profil : son nom LOAF, `_` remplacé par `-`. `os_raw_zst_url` / `os_raw_zst_sha256` / `os_bmap_url` sont vides quand le profil source n'a pas de `raw_zst_url` / `bmap_url` (§3.1) — chaîne vide, pas de champ absent, pour simplifier l'analyse côté init.

Format de `GET /provisioning/<mac>` : renvoie le même JSON, plus un champ `created_at` (horodatage Unix).

7.3 Point d'accès des nœuds

`GET /nodes/:id/cloud-init` renvoie le cloud-init d'un nœud enregistré, pour l'installation K3s. Il consulte d'abord `/provisioning/<mac>` pour la MAC du nœud : un enregistrement présent prend le dessus.

SECTION 8

Flux de bout en bout

1. L'exploitant écrit le profil (`dev-workstation.loaf`) et, pour une flotte, l'infrastructure qui le désigne (`dev-fleet.loaf`).
2. `hamac provision-push --profile=… --mac=…` :
   - charge le profil avec les bundles des répertoires de recherche ;
   - génère le cloud-init final (gabarits Jinja2 + fusion) et le script iPXE ;
   - envoie un POST sur `/provisioning/<mac>` du serveur de découverte.
3. La machine démarre en PXE → script d'init :
   - détecte la MAC locale ;
   - `GET /provisioning/<mac>` → reçoit le cloud-init + l'URL de l'OS (+ éventuellement `os_raw_zst_url` / `os_bmap_url`, cf. §7.2) ;
   - installe selon `format` (cf. §3.1) :
     - `qcow2` / `raw` → écrit l'image bloc sur le disque : `blkdiscard` puis, si `os_raw_zst_url` / `os_bmap_url` sont présents et `bmaptool` disponible dans l'initrd, `bmaptool copy` **directement depuis ces URL** (streaming, décompression à la volée, vérification sha256 par bloc — pas de téléchargement local préalable) ; sinon repli : téléchargement local, puis vérification sha256 (`os_image_sha256`), puis `qemu-img convert` (ou `dd` direct pour le format `raw`). Le redimensionnement de la partition est délégué à cloud-init au premier démarrage (`growpart` + `resize_rootfs`) ;
     - `netinstall` → preseed Debian ;
   - entre dans le système installé (chroot) et y applique cloud-init ;
   - redémarre.
4. La machine démarre sur l'OS installé → cloud-init s'applique au premier démarrage → utilisateur créé avec sa clé SSH → dotfiles installés.

SECTION 9

Hors périmètre (v0.2)

- **Multi-tenant** : un seul serveur de découverte par flotte.
- **Signature des profils** : pourra être ajoutée avec la même mécanique que les services (Ed25519).
- **Bundles avec dépendances cycliques** : non détectés ; on fait confiance à l'exploitant.
- **Mise à jour à chaud d'un OS déjà provisionné** : ce n'est pas un agent de configuration (cf. Ansible / Salt). Le profil ne s'applique qu'au moment du démarrage PXE.
- **Bundles avec état d'exécution** (base de données, etc.) : les services de type Zulip avec PostgreSQL passent par les stacks de Hamac.
- **Description depuis un langage de plus haut niveau** : voir §10.

SECTION 10

Évolutions futures

10.1 Court terme

- `hamac provision-push` sur une infrastructure : pousser, pour chaque nœud dont la MAC est connue (issue du planner ou fournie), le rendu du profil qu'il désigne.
- Diff entre deux versions d'un profil.
- Héritage de profils : un profil qui en complète un autre.

10.2 Moyen terme : description de plus haut niveau

Une fois la mécanique stabilisée, un langage ou un outil de plus haut niveau pourra produire les profils, comme il peut produire des manifestes de service : les manifestes LOAF le permettent sans changement côté Hamac. On pourra alors envisager :

- la vérification des valeurs des paramètres d'un profil contre leur `type` dans le bundle ;
- une vue web des profils ;
- la génération de profils paramétriques, par exemple un profil par utilisateur.

10.3 Long terme

- Signature Ed25519 des profils (parité avec les services).
- Provisioning de conteneurs d'OS (LXC, systemd-nspawn) en plus du bare-metal et des VM.
- Gestion de versions avec retour arrière sur l'OS installé (partitions A/B).

SECTION 11

Références

- `doc/typst/manifest.typ` : spécification des manifestes (vocabulaire LOAF de Hamac, dont les blocs `provisioning_profile` et `bundle`).
- `pkgs/hamac-manifest/vocabulary/hamac.loaf` : le vocabulaire.
- `pkgs/hamac-provisioning/provisioning_gen.ml` : rendu du cloud-init et du script iPXE.
- `pkgs/hamac-discovery/` : serveur de découverte.
- `tools/discovery-prototype/pxe-build/init` : script d'init PXE.
- `doc/HAMAC_ROADMAP.md` : milestone 6.

Financement

« Ce projet a été financé par le Gouvernement dans le cadre du plan France 2030 opéré par Cap Digital et financé par l'Union européenne – NextGeneration EU »
« Ce projet a été financé par le gouvernement dans le cadre de France 2030 »
« Financé par l'Union européenne - Next Generation EU dans le cadre du plan France Relance »







