V1.0

EK NETSEC LABS

PARTIE 1

IA INFRASTRUCTURE — UN ORCHESTRATEUR MULTI-AGENTS LOCAL ET SECURISE

ELIAS KARISS © EK NETSEC LABS - TOUS DROITS RESERVES

Table des matières

Elias KARISS © EK netsec labs - Tous droits réservés

EK NETSEC LABS

1. SYNTHESE DU PROJET

Cette première partie établit un socle local pour un futur générateur d'infrastructures et de contrôles de cybersécurité piloté par des agents IA. Le projet privilégie la maîtrise technique : modèles locaux, code Python lisible, permissions minimales, états persistants et validation humaine obligatoire.

L'environnement est installé sur une VM Debian sans interface graphique. Ollama assure l'inférence uniquement sur l'interface loopback. L'application expose une CLI capable de vérifier le service, lister les modèles, poser une question, créer un projet et exécuter le workflow séquentiel ATLAS → FORGE → ARGOS.

Élément	Résultat Partie 1	Garantie
Socle système	Debian 13, Python 3.13, outils DevSecOps	Installation idempotente
Inférence	Ollama local	Écoute 127.0.0.1 uniquement
Orchestration	ATLAS → FORGE → ARGOS	Séquence persistée
Décision	État awaiting_approval	Validation humaine obligatoire
Exécution	Absente	Aucun terminal donné au LLM

RESULTAT CLE

Validation — Le test d'intégration réel a produit les trois sorties dans l'ordre, persisté l'état final en attente d'approbation et confirmé qu'aucune action d'infrastructure n'avait été exécutée.

Elias KARISS © EK netsec labs - Tous droits réservés

2. VISION, OBJECTIFS ET PERIMETRE

2.1 VISION

L'ambition est de transformer une demande exprimée en langage naturel en une proposition d'architecture structurée, des artefacts techniques révisables et un audit de sécurité. La chaîne doit rester compréhensible, traçable et réversible.

2.2 OBJECTIFS DE LA PARTIE 1

- Préparer une VM Debian stable et reproductible.
- Installer et encadrer Ollama pour une machine disposant d'environ 11 Gio de RAM.
- Attribuer trois rôles spécialisés à des modèles déjà présents.
- Développer une application Python moderne sans framework multi-agents lourd.
- Séparer planification, génération, audit, approbation et future exécution.
- Valider le code et l'exploitation avec des tests automatisés.

2.3 HORS PERIMETRE

- Aucune connexion Proxmox, aucun token API et aucune création de VM.
- Aucun exécuteur shell générique et aucune commande générée puis exécutée automatiquement.
- Aucune exposition réseau d'Ollama et aucun accès root confié aux modèles.
- Aucun secret, mot de passe ou rapport sensible publié dans Git.

Elias KARISS © EK netsec labs - Tous droits réservés

3. PREPARATION DE LA VM DEBIAN

3.1 AUDIT INITIAL

Le travail a commencé par un audit strictement en lecture seule : version Debian, noyau, CPU, mémoire, swap, stockage, réseau, DNS, Internet, SSH, pare-feu, services, heure, Ollama, modèles et chaîne Python. Les changements n'ont été engagés qu'après validation.

Ressource	État constaté
Système	Debian GNU/Linux 13.6, noyau 6.12 amd64
Calcul	8 vCPU, environ 11 Gio de RAM
Mémoire virtuelle	4,1 Gio de swap
Stockage	75 Gio, plus de 50 Gio disponibles lors de l'audit
Temps	NTP actif, fuseau Europe/Paris
Services	Aucune unité systemd en échec

3.2 SOCLE INSTALLE

Le socle comprend Git, curl, jq, rsync, tmux, les outils de compilation Python, pip, venv, pipx, Ansible, ansible-lint, ShellCheck, Yamllint, OpenSSL, ACL et logrotate. APT a été utilisé de façon idempotente sans changement de version majeure de Debian.

3.3 COMPTES ET PERMISSIONS

Le code appartient à l'utilisateur principal du projet. Un compte système distinct, non interactif et sans mot de passe, est réservé au futur service. Les fichiers de code sont en 0640, les scripts en 0750 et les répertoires de données en 2770.

```
cd /opt/infra-agent && source .venv/bin/activate
```

Elias KARISS © EK netsec labs - Tous droits réservés

4. OLLAMA ET MODELES IA LOCAUX

4.1 SERVICE LOCAL

Ollama 0.32.0 était déjà installé. Le service a été vérifié actif, activé au démarrage et accessible uniquement sur 127.0.0.1:11434. Les endpoints de version et de catalogue ont répondu correctement.

4.2 MAITRISE DE LA MEMOIRE

```
OLLAMA_MAX_LOADED_MODELS=1
OLLAMA_NUM_PARALLEL=1
OLLAMA_CONTEXT_LENGTH=8192
OLLAMA_KEEP_ALIVE=2m
```

Cette surcharge systemd évite le chargement simultané de plusieurs modèles, limite la concurrence, borne le contexte et libère rapidement la mémoire après inactivité.

4.3 MODELES DETECTES

Modèle source	Taille	Rôle prévu
qwen3:4b	2,5 Go	ATLAS — architecture et planification
qwen2.5-coder:3b	1,9 Go	FORGE — génération technique
deepseek-r1:8b-0528-qwen3-q4_K_M	5,2 Go	ARGOS — audit et risques

Gestion des noms — Les rôles ATLAS, FORGE et ARGOS sont des identités logiques portées par les prompts et les Modelfiles. Les modèles sources restent référencés sans duplication de leurs blobs.

Elias KARISS © EK netsec labs - Tous droits réservés

5. AGENTS ATLAS, FORGE ET ARGOS

5.1 ATLAS — ARCHITECTE

ATLAS analyse la demande, explicite les hypothèses, identifie les composants et produit un plan structuré. Il ne réalise aucune action et termine par un point d'approbation humaine.

5.2 FORGE — GENERATEUR TECHNIQUE

FORGE transforme le plan en artefacts Python, Bash, YAML, Ansible, cloud-init ou OpenTofu. Ces artefacts restent inertes, documentés et soumis à audit ; FORGE ne les exécute jamais.

5.3 ARGOS — AUDITEUR CYBERSECURITE

ARGOS recherche secrets, permissions excessives, exposition réseau, injections, actions destructrices et défauts de retour arrière. Il peut déclarer des constats bloquants, sans administrer le système.

5.4 CHAINE DE CONFIANCE

Étape	Entrée	Sortie	Pouvoir
ATLAS	Demande	Plan	Proposer
FORGE	Plan	Artefacts inertes	Générer
ARGOS	Plan + artefacts	Audit	Contrôler / bloquer
Humain	Dossier complet	Décision	Approuver ou refuser

Elias KARISS © EK netsec labs - Tous droits réservés

6. ORCHESTRATEUR PYTHON ET CLI

6.1 Structure du projet

Le projet est organisé sous /opt/infra-agent : application CLI, clients HTTP, prompts, orchestrateur, schémas Pydantic, outils contraints, configuration, données, journaux, scripts, tests et documentation. Un environnement virtuel isole les dépendances.

6.2 Workflow persistant

SequentialOrchestrator appelle chaque rôle dans l'ordre et sauvegarde un JSON après chaque étape. Une erreur ne peut donc pas faire croire qu'une étape ultérieure a réussi. La requête et les sorties précédentes sont délimitées comme données non fiables et leurs marqueurs sont échappés.

```
requested → planned → audited → awaiting_approval → approved
```

6.3 Commandes disponibles

Commande	Fonction
infra-agent health	Vérifier l'API locale Ollama
infra-agent models	Lister les modèles installés
infra-agent ask	Interroger un modèle et écrire un rapport
infra-agent new-project	Créer un identifiant et un état local
infra-agent orchestrate	Exécuter ATLAS, FORGE puis ARGOS
infra-agent approve-project	Enregistrer l'approbation, sans exécuter

Elias KARISS © EK netsec labs - Tous droits réservés

6.4 CODE COMMENTE — ORCHESTRATION

L'extrait ci-dessous est une adaptation pédagogique de la logique centrale. Chaque résultat est sauvegardé avant de passer à l'agent suivant ; aucune fonction d'exécution d'infrastructure n'est appelée.

```
class SequentialOrchestrator:
    async def run(self, project_name: str, request: str) -> WorkflowState:
        # 1. Créer un état traçable puis le sauvegarder immédiatement.
        state = WorkflowState.new(project_name, request, self.models)
        self.save(state)

        # 2. ATLAS ne fait que proposer une architecture.
        atlas_plan = await self.generator.generate(
            self.models.atlas,
            stage_prompt("Produire un plan d'architecture", request),
            load_system_prompt("atlas"),
        )
        state = next_review_stage(state).model_copy(
            update={"atlas_plan": atlas_plan}
        )
        self.save(state)  # point de reprise après ATLAS

        # 3. FORGE produit des artefacts inertes, sans les exécuter.
        forge_artifacts = await self.generator.generate(
            self.models.forge,
            stage_prompt("Générer des artefacts révisables", request, atlas_plan),
            load_system_prompt("forge"),
        )
        state = next_review_stage(state).model_copy(
            update={"forge_artifacts": forge_artifacts}
        )
        self.save(state)  # point de reprise après FORGE
```

À retenir — Le générateur renvoie uniquement du texte. Il ne reçoit ni terminal, ni privilège root, ni fonction permettant d'exécuter ce qu'il propose.

6.5 CODE COMMENTE — AUDIT ET GARDE-FOUS

```
# 4. ARGOS reçoit le plan et les artefacts comme données non fiables.
argos_input = f"ATLAS PLAN:\n{atlas_plan}\n\nFORGE:\n{forge_artifacts}"
argos_audit = await self.generator.generate(
    self.models.argos,
    stage_prompt("Auditer et signaler les blocages", request, argos_input),
    load_system_prompt("argos"),
)
state = next_review_stage(state).model_copy(
    update={"argos_audit": argos_audit}
)
self.save(state)

# 5. La séquence doit obligatoirement s'arrêter ici.
if state.stage is not WorkflowStage.AWAITING_APPROVAL:
    raise RuntimeError("La frontière d'approbation n'a pas été respectée")
return state

# Le client refuse une URL Ollama distante pendant cette phase.
base_url = str(settings.ollama_url).rstrip("/")
if base_url not in {
    "http://127.0.0.1:11434",
    "http://localhost:11434",
}:
    raise ValueError("Seul Ollama local est autorisé")

# Une approbation change l'état, mais ne déclenche aucune action.
state = state.model_copy(update={
    "stage": WorkflowStage.APPROVED,
    "approved_by": approved_by.strip(),
    "infrastructure_actions_executed": False,
})
```

Dans le code complet, les noms de projet sont contrôlés par expression régulière, les chemins sont résolus sous une racine autorisée, les entrées sont bornées, les réponses HTTP sont validées et les erreurs exposées à l'utilisateur sont assainies.

7. ARCHITECTURE DE SECURITE

7.1 GARDE-FOUS APPLICATIFS

- Aucune utilisation de shell=True, eval ou exec.
- Endpoints Ollama distants refusés dans cette étape.
- Entrées bornées et noms de fichiers validés.
- Résolution des chemins sous des racines contrôlées.
- Timeout de connexion court et timeout global borné à 300 secondes.
- Sortie de génération limitée à 2 048 tokens.
- Rapports et états écrits avec des permissions non publiques.

7.2 GESTION DES SECRETS

Le fichier .env est ignoré par Git et .env.example ne contient aucune valeur sensible. Les clés, certificats, credentials, logs et rapports sont exclus du dépôt. Les modèles ne reçoivent ni secret ni environnement complet.

7.3 FRONTIERE D'APPROBATION

Blocage structurel — La commande approve-project ne fait qu'enregistrer une identité humaine. Même approuvé, un projet ne peut rien déployer car aucun exécuteur n'existe dans la Partie 1.

Elias KARISS © EK netsec labs - Tous droits réservés

8. TESTS, RESULTATS ET CORRECTIONS

8.1 CHAINE QUALITE

Contrôle	Résultat
Ruff	Réussi
MyPy strict	Réussi sur 18 fichiers source
Pytest	10 tests réussis
Yamllint	Réussi
ShellCheck	Réussi
API Ollama	Version et catalogue validés
Test réel orchestrateur	3 étapes, arrêt awaiting_approval

8.2 DIFFICULTES RENCONTREES

- Une première clé SSH créée dans un sandbox Windows n'était pas lisible par le processus SSH normal.
- Un canal local APT s'est fermé alors que le processus distant continuait correctement.
- Le modèle qwen3:4b nécessitait environ deux minutes uniquement pour son chargement CPU.
- Les premiers contrôles Ruff, MyPy et Yamllint ont révélé des problèmes de style et de typage.
- Une substitution Bash a été interprétée localement par PowerShell pendant un test combiné.

8.3 CORRECTIONS

Les processus ont été vérifiés avant toute reprise, le timeout applicatif a été porté à 300 secondes, le typage strict a été conservé, les délimiteurs anti-injection ont été échappés et les tests ont été relancés après chaque correction.

Elias KARISS © EK netsec labs - Tous droits réservés

9. GUIDE D'UTILISATION

9.1 ACTIVER L'ENVIRONNEMENT

```
cd /opt/infra-agent
source .venv/bin/activate
```

9.2 VERIFIER OLLAMA

```
infra-agent health
infra-agent models
```

9.3 LANCER UNE ORCHESTRATION

```
infra-agent orchestrate --name demo --request "Prépare un plan auditable"
```

9.4 EXAMINER L'ETAT

Le fichier JSON du projet contient la demande, les modèles choisis, la sortie ATLAS, les artefacts FORGE, l'audit ARGOS, le statut et l'identité éventuelle de l'approbateur.

9.5 ENREGISTRER UNE APPROBATION

```
infra-agent approve-project --project-id "ID" --approved-by "operateur"
```

Important — Cette commande n'exécute aucune infrastructure.

Elias KARISS © EK netsec labs - Tous droits réservés

10. BILAN ET PARTIE 2 : PROXMOX

10.1 Acquis

La Partie 1 fournit un socle local fonctionnel, audité et documenté. Le choix d'un orchestrateur simple permet de voir précisément les frontières de confiance et de tester chaque étape sans dépendre d'un framework opaque.

10.2 Limites assumees

- Aucun connecteur Proxmox ou Ansible opérationnel.
- Aucune création, modification ou suppression d'infrastructure.
- Pas encore d'interface web ni de gestion avancée de reprise.
- Les décisions bloquantes d'ARGOS restent à formaliser dans un schéma structuré dédié.

10.3 Partie 2

La prochaine phase commencera par l'installation et l'audit de Proxmox sur un système séparé. Un compte dédié et un token à privilèges minimaux seront créés après validation. Le premier connecteur sera exclusivement en lecture seule : version, nœuds, stockage, réseaux et machines existantes.

Progression contrôlée — Les capacités de création ne seront ajoutées qu'après audit ARGOS, mode simulation, journalisation et validation humaine explicite.

Elias KARISS © EK netsec labs - Tous droits réservés