ChatGPT

Plan d’évolution de Recon Gotham pour un Audit Red Team Évolué

Contexte et Objectifs de l’Évolution

Recon Gotham doit évoluer d’un outil de **reconnaissance** à un cadre complet d’**audit Red Team** avec validation déterministe des vulnérabilités. Actuellement, Recon Gotham identifie des vulnérabilités potentielles (statut *théorique*) via des scans automatisés, sans preuve concrète. L’objectif est de passer à une logique *evidence-based* où chaque vulnérabilité détectée est accompagnée d’une **preuve** et d’un **niveau de confiance**, classée comme **CONFIRMED**, **LIKELY** ou **THEORETICAL** (Confirmée, Probable ou Théorique). Cette évolution doit respecter les **règles d’engagement (ROE)** en vigueur, sans changer l’architecture existante. Concrètement, il s’agit d’intégrer de nouveaux modules de vérification par technologie et de nouveaux agents *CrewAI* spécialisés, tout en maintenant l’orchestrateur par phases et le graphe de connaissance existants.

L’architecture actuelle de Recon Gotham est conservée : un orchestrateur gère des microservices par phases (OSINT → ACTIVE_RECON → ENDPOINT_INTEL → VERIFICATION → PLANNER → REPORTING) et alimente un graphe de connaissances (nœuds *ENDPOINT*, *HYPOTHESIS*, *VULNERABILITY*, etc., reliés par *AFFECTS_ENDPOINT*, etc.). Nous allons enrichir la phase **VERIFICATION** pour y orchestrer des vérifications ciblées. Ces vérifications seront réalisées via un **runner sandboxé** (conteneur isolé) chargé d’exécuter des scripts Python ou requêtes HTTP de manière sécurisée, conformément aux ROE. Aucun service externe ni LLM non maîtrisé ne sera utilisé : tous les traitements s’exécutent localement, orchestrés par l’orchestrateur CrewAI existant [1].

Règles d’Engagement et Niveaux d’Intrusivité

Les nouvelles fonctionnalités respecteront strictement les ROE définies :

- **Méthodes HTTP autorisées** : Seules les requêtes **GET**, **HEAD** et **OPTIONS** sont permises sur toutes cibles en mode passif. Les requêtes **POST** sont autorisées uniquement dans les modes **BALANCED** et **AGGRESSIVE**, et encore, seulement pour des sondes sûres et non destructives. Les méthodes potentiellement dangereuses (**PUT**, **PATCH**, **DELETE**) restent désactivées par défaut, sauf autorisation explicite sur certaines cibles (allowlist spécifique).
- **Authentification** : Si des identifiants ou tokens d’accès sont fournis, les modules pourront les utiliser (injection de tokens JWT, cookies de session, etc.) pour tester des fonctionnalités authentifiées. Cependant, **aucune attaque de force brute ou de credential stuffing** ne sera effectuée – l’authentification sera utilisée uniquement pour reproduire des scénarios fournis, pas pour en découvrir de nouveaux.
- **Intrusivité par mode** : En mode **AGGRESSIVE**, l’outil peut réaliser du fuzzing encadré (listes de mots bornées, parallélisme et débit limités) et envoyer des requêtes POST avec JSON/XML sur des endpoints connus, ainsi que jouer des workflows d’authentification complets si on dispose des accès.

En mode **BALANCED**, certaines actions actives modérées sont permises (ex: POST non destructifs). En mode **STEALTH/PASSIVE**, on se limite aux requêtes en lecture (GET/HEAD) sans altérer l'état de la cible. Tous les modules de vérification seront marqués du niveau d'intrusivité requis, et le plan de validation n'autorisera leur exécution que si le mode en cours le permet (par ex., un module nécessitant un POST ne sera exécuté qu'en Balanced ou Aggressive).

Ces règles garantissent que la phase de validation approfondie reste dans le cadre éthique et contractuel de l'audit. L'orchestrateur appliquera ces contraintes globalement, et le *validation_planner_agent* filtrera ou adaptera les modules en conséquence.

Matrice des Modules de Validation par Technologie

Nous introduisons une **matrice de modules de validation ciblés par technologie**. Chaque module est un script ou gabarit de requête conçu pour vérifier une vulnérabilité spécifique, adaptée à une stack technologique particulière (WordPress, Laravel, Node.js, etc.). Cette approche s'inspire des scanners à base de templates comme Nuclei, qui proposent une librairie de checks organisée par technologies et failles connues [2]. Chaque module de validation est défini de manière standardisée avec un identifiant unique, une description, la méthode HTTP à utiliser, d'éventuelles conditions/prérequis, l'indicateur de preuve attendu, le statut maximal atteignable (Confirmé ou Probable) et un **template** (script ou requête HTTP). Ces modules permettront de reproduire à volonté la vérification – de la même façon que les templates Nuclei encapsulent des requêtes et des critères de réussite pour partager des tests réutilisables [3].

Nous présentons ci-dessous quelques exemples de modules par technologie (cette liste pourra être enrichie au fil du temps ; l'objectif ici est de définir la structure de la matrice) :

WordPress

- **Module WP1 – wordpress_xmlrpc_ping**
  Description : Vérifie si le service XML-RPC de WordPress est exposé et actif. L'interface XML-RPC est souvent une surface d'attaque (par ex. pour des attaques par amplification ou brute force XML-RPC) et sa présence peut être considérée comme une faiblesse à signaler.
  Méthode : `POST` (XML payload de ping minimal) sur `/xmlrpc.php` (**autorisé uniquement en modes Balanced/Aggressive**).
  Conditions : Le point d'accès `/xmlrpc.php` doit répondre avec un code 200 OK et renvoyer une réponse XML valide (par exemple une structure `<methodResponse>` ou éventuellement un message d'erreur spécifique de WordPress, tant que ce n'est pas une erreur HTTP 404/501).
  Preuve attendue : Un extrait de la réponse XML du serveur prouvant que l'appel XML-RPC a été accepté (par ex. élément `<methodResponse>` ou message « XML-RPC server accepts POST requests »). On peut stocker un hash de la réponse complète si nécessaire.
  Statut max : *LIKELY* – La présence d'XML-RPC est confirmée (preuve à l'appui), ce qui indique une surface d'attaque potentielle, mais ce n'est pas en soi une compromission prouvée. Le statut Probable convient, avec un niveau de confiance moyen.
  Template/Requête : Une requête POST XML-RPC de ping (méthode system.listMethods ou similaire) est envoyée via un script Python. Le template JSON du module contient l'URL cible, les en-têtes (`Content-Type: text/xml`), et le corps XML minimal requis.

- **Module WP2 – wordpress_user_enum**
  Description : Tente d'énumérer les utilisateurs WordPress publiquement disponibles, soit via l'API REST par défaut, soit via le formulaire de login (pour détecter si des noms d'utilisateur peuvent être révélés). De nombreuses versions WordPress exposent la liste des utilisateurs administrateurs via `/wp-json/wp/v2/users` ou laissent deviner l'existence d'un utilisateur via des messages d'erreur de `/wp-login.php`.
  Méthode : `GET` sur `/wp-json/wp/v2/users` (appel REST public) **ou** petite série de requêtes `POST` sur `/wp-login.php` avec des noms d'utilisateur plausibles (ex: *admin*) pour comparer les messages d'erreur. En mode Stealth, on privilégie l'appel REST en lecture seule; le test de login (très limité à 1–2 essais) serait réservé au mode Balanced.
  Conditions : – API REST: une réponse 200 contenant une liste JSON d'utilisateurs (champs `name`, `slug`, etc.).
  – Formulaire login: différence dans la réponse entre « nom d'utilisateur invalide » et « mot de passe incorrect » indiquant que l'utilisateur testé existe.
  Preuve attendue : – API REST: un extrait JSON montrant un nom d'utilisateur ou un ID d'utilisateur récupéré.
  – Login: le message précis renvoyé par WordPress pour chaque cas, démontrant qu'un nom d'utilisateur était valide (par ex. preuve textuelle « Unknown username » vs « incorrect password »).
  Statut max : *CONFIRMED* – Si l'on parvient à obtenir la liste des utilisateurs ou à confirmer l'existence d'un compte, c'est une preuve directe d'une fuite d'information (vulnérabilité confirmée). On associera un niveau de confiance élevé, car la preuve est explicite (par ex. liste d'utilisateurs).
  Template/Requête : Une requête GET simple pour l'API REST (pas de corps, en-têtes JSON). Pour la variante login, une paire de requêtes POST est effectuée avec un utilisateur fictif et un utilisateur commun (ex: *admin*) et leurs réponses HTML sont comparées par un script pour détecter une différence indicative.

- **Module WP3 – wordpress_version_disclosure**
  Description : Vérifie si la version de WordPress est divulguée quelque part publiquement (ce qui permettrait de savoir si le site est à jour ou non). Par exemple, beaucoup de sites WordPress laissent accessible le fichier `/readme.html` par défaut ou affichent la version dans la balise meta générator du HTML.
  Méthode : `GET` sur des ressources connues : `/readme.html`, `/wp-includes/js/wp-emoji-release.min.js` (qui contient un numéro de version en commentaire), ou la page d'accueil pour extraire la meta `\<meta name="generator" content="WordPress ..."\>`.
  Conditions : Code 200 sur l'une des ressources ciblées et présence d'une chaîne indiquant un numéro de version (regex du type `WordPress [0-9\.]+`).
  Preuve attendue : Un extrait de texte contenant la version exacte (ex: **WordPress 5.8.2** dans le readme). La preuve peut être stockée en clair si non sensible (il s'agit d'une version publique) ou hachée si besoin.
  Statut max : *LIKELY* – La fuite de version est confirmée, ce qui augmente la probabilité de vulnérabilités connues si la version est obsolète. Toutefois, tant qu'aucune vulnérabilité spécifique n'est exploitée, on considère cela comme un indice fort (Probable) plutôt qu'une compromission confirmée.
  Template/Requête : Requêtes GET multiples (définies dans le module comme une liste d'URLs à tenter). Un script peut automatiser la détection de version (parser le HTML ou le contenu). Le module peut intégrer une liste de patterns regex à chercher pour identifier la version.

(D'autres modules WordPress pourraient être ajoutés dans la matrice, par ex. détection de plugin vulnérable via un fichier readme.txt *de plugin, présence de* `/wp-config.php.bak` *sauvegardé accessible, etc. Le principe reste le même : chaque module correspond à un test unitaire, encapsulant la logique de détection et la preuve.)*

Laravel

- **Module L1 – laravel_env_leak**
  Description : Tente d'accéder au fichier de configuration sensible `.env` à la racine du site Laravel. Par défaut, ce fichier ne devrait jamais être exposé, mais s'il l'est, il contient des informations critiques (clés API, identifiants DB). C'est une vérification déterministe classique pour les applis Laravel.
  Méthode : `GET` sur `/.env` à la racine du site.
  Conditions : Réponse HTTP 200 avec un contenu textuel contenant typiquement des clés Laravel (par ex. la présence de chaînes comme `APP_KEY=` ou `DB_PASSWORD=` en début de ligne).
  Preuve attendue : Un extrait du contenu du fichier `.env` prouvant la fuite, par exemple la clé d'application (hash base64) ou le début du mot de passe base de données (en masquant une partie si nécessaire). On peut par sécurité stocker un hash cryptographique du contenu complet au lieu du contenu brut, afin de ne pas manipuler les secrets en clair plus que de besoin.
  Statut max : *CONFIRMED* – Preuve irréfutable d'une faille de configuration (données sensibles exfiltrées). La vulnérabilité est confirmée avec une confiance maximale, car nous avons obtenu une donnée qui ne devrait pas être publique.
  Template/Requête : Un simple GET via un script Python ou HTTP client. Le module JSON précise le chemin `/.env`. Aucune en-tête spéciale n'est requise. Le script analyserait la réponse pour localiser des tokens connus (`APP_KEY=`) et renverrait un extrait.

- **Module L2 – laravel_debug_mode**
  Description : Vérifie si l'application Laravel est en mode debug sur l'environnement de production. En mode debug, une exception ou erreur génère une page détaillée (symphony error page) exposant le stack trace, des variables d'environnement, etc. C'est un comportement non sécurisé.
  Méthode : `GET` sur une URL volontairement erronée ou inexistante (p. ex. `/<URL>/__testing__`) pour provoquer une exception non gérée.
  Conditions : Réponse HTTP 500 avec une page HTML contenant des indices de debug Laravel (par ex. la présence de la classe `Illuminate\Exception\Handler` ou du mot "Exception" dans le body, ou encore une structure HTML spécifique à la page d'erreur Laravel debug).
  Preuve attendue : Un extrait du message d'erreur ou du stack trace démontrant que le debug est activé (par ex. la mention du chemin du fichier PHP dans le stack trace, ou la variable APP_DEBUG=true apparaissant). Même une capture de l'en-tête `X-Debug-Token` (si un debug bar est présent) peut servir de preuve.
  Statut max : *CONFIRMED* – La configuration dangereuse est vérifiée par la présence effective d'une page de debug. On a une preuve directe (page d'erreur détaillée), ce qui confirme la vulnérabilité (niveau de confiance élevé).
  Template/Requête : Requête GET standard. Le module peut réessayer sur 1 ou 2 chemins aléatoires pour déclencher une erreur 500. Le script capture la réponse HTML et y recherche des marqueurs (regex sur `Exception in` ou `Stack trace` par exemple).

- **Module L3 – laravel_log_exposure**
  Description : Tente d'accéder aux fichiers de log Laravel exposés (par ex. `storage/logs/`

`laravel.log`). Si le serveur web n'est pas correctement configuré, ces fichiers peuvent être accessibles publiquement et contenir des traces d'erreur ou des données sensibles.
Méthode : `GET` sur des emplacements de logs connus (`/storage/logs/laravel.log` ou d'autres noms de logs rotatifs avec date).
Conditions : Réponse 200 avec un contenu texte contenant vraisemblablement des entrées de log Laravel (souvent reconnaissables à des timestamps, niveaux genre "ERROR" et mention d'erreurs PHP ou requêtes SQL).
Preuve attendue : Un extrait du log récupéré, par exemple une ligne d'erreur avec timestamp et message, pour prouver l'accès. Si possible, on choisira une ligne anodine (ne contenant pas de secret utilisateur) ou on hachera partiellement l'information.
Statut max : *CONFIRMED* – Avoir accès aux logs constitue une preuve tangible d'exposition de données internes. C'est une vulnérabilité confirmée, avec preuve à l'appui.
Template/Requête : GET sur une liste de chemins possible (le module pourrait définir plusieurs variantes de nom de fichier selon date, à essayer). Un script tente chaque chemin et dès qu'une réponse valide est reçue, lit quelques lignes.

Applications Node.js (Express, etc.)

- **Module N1 – node_package_exposed**
  Description : Vérifie si le fichier de description de l'application Node (souvent `package.json`) est accessible via le web. Si oui, cela divulgue des informations sur les dépendances et la structure de l'application (noms de modules, versions), ce qui peut aider à identifier des vulnérabilités connues.
  Méthode : `GET` sur `/package.json` à la racine du site.
  Conditions : Réponse 200 contenant du JSON valide, avec des champs typiques comme `"name"`, `"version"`, `"dependencies"`...
  Preuve attendue : Un extrait du fichier JSON confirmant qu'il s'agit bien du `package.json` de l'application (par ex. le champ "name" de l'application et une dépendance listée). Ces informations ne sont pas hautement sensibles en elles-mêmes, on peut donc les stocker en clair comme preuve.
  Statut max : *LIKELY* – Cela confirme une fuite d'information (le fichier est accessible), vulnérabilité configurée comme Probable. Ce n'est pas aussi critique qu'une compromission directe, mais cela fournit assez de détails pour potentiellement faciliter une attaque (confiance moyenne dans l'impact).
  Template/Requête : Une simple requête GET. Le module définit l'URL cible et un parseur JSON dans le script de vérification pour extraire quelques champs (name, version) en guise de preuve.

- **Module N2 – node_error_stacktrace**
  Description : Teste si l'application Node/Express divulgue des *stack traces* ou messages d'erreur détaillés lors de requêtes erronées. En environnement de production, Express masque les erreurs par défaut, sauf si un middleware de debug est laissé actif. Ce module tente de provoquer une erreur pour voir si des détails internes sont retournés.
  Méthode : `GET` ou `POST` sur une URL factice ou en envoyant des paramètres incorrects à une route connue, visant à provoquer une exception non gérée. Par exemple, envoyer un JSON mal formé à une route API POST (si documentée) ou accéder à `/error-test` si un tel routeur de test existe.
  Conditions : Réponse contenant un indice de stack trace Node.js (par ex. la présence de lignes mentionnant `.js:` avec numéro de ligne, ou le texte "Error:" suivi d'un message de debug). Le code HTTP peut être 500.

Preuve attendue : Un extrait de la réponse montrant une partie du stack trace ou un message d'erreur interne. Par exemple: `TypeError: Cannot read property 'x' of undefined at app.js:45:13`. Ce genre de sortie prouve qu'on a un aperçu du code interne.
Statut max : *CONFIRMED* – Si un stack trace est visible, c'est une preuve directe de fuite d'information (bonne pratique de sécurité violée). Vulnérabilité confirmée donc, avec confiance élevée (puisqu'on voit réellement le comportement inapproprié).
Template/Requête : Dépend de l'application – le module pourrait avoir un paramètre pour la route à tester. Dans la configuration générale, on peut d'abord essayer une requête volontairement mal formée sur la page d'accueil (ex: envoyer un corps JSON sur une route GET) et voir si cela déclenche une erreur. Le script analysera la réponse textuelle.

(Ces exemples de modules démontrent la structure attendue. D'autres pourront être ajoutés pour d'autres stacks : par ex. un module Drupal pour vérifier si `/CHANGELOG.txt` est accessible et révèle la version de Drupal, un module Apache Struts pour tester une commande de test non destructive sur une URL connue vulnérable, etc. La matrice sera un référentiel évolutif de checks adaptés à chaque technologie identifiée.)

Chaque module comporte un champ indiquant la **méthode HTTP** à utiliser, ce qui permet d'automatiquement filtrer son exécution selon le mode d'intrusivité. Par exemple, les modules marqués GET/HEAD peuvent tourner même en mode passif, tandis qu'un module comme *wordpress_xmlrpc_ping* marqué POST sera ignoré en mode Stealth. De même, des modules impliquant du fuzzing intensif seraient marqués comme **AGGRESSIVE only** et ne seraient activés que si la configuration du scan le permet.

Enfin, pour chaque module est associé un **gabarit de script** (ou requête) pouvant être exécuté via l'exécuteur isolé. Il peut s'agir soit d'une unique requête HTTP simple définie entièrement par la spec JSON (voir section schéma JSON ci-dessous), soit d'un script Python plus complexe (présent dans un référentiel de scripts) que le runner saura appeler. Dans tous les cas, la définition du module inclut tout ce qui est nécessaire pour le lancer et vérifier son résultat.

Nouveaux Agents CrewAI et Intégration dans le Pipeline

Pour orchestrer ces modules de validation ciblée, nous introduisons plusieurs **agents CrewAI spécialisés**. L'approche multi-agents s'aligne parfaitement avec l'architecture CrewAI existante, qui est conçue pour assigner des tâches spécifiques à des agents nommés et enchaîner leurs résultats dans un pipeline red team [4]. Ces nouveaux agents s'intègrent dans la phase **VERIFICATION** du pipeline sans perturber les autres phases. Leur rôle est de sélectionner les cibles, choisir les modules appropriés, planifier les vérifications et intégrer les résultats au graphe de connaissances. Voici la liste des agents à créer/modifier, avec leur rôle, position dans le pipeline et interactions :

- `vuln_triage_agent` – *Agent de triage des vulnérabilités à valider*
  Rôle : Sélectionner, parmi l'ensemble des endpoints découverts et vulnérabilités théoriques identifiées, lesquels doivent faire l'objet d'une validation approfondie. Il priorise les cibles en fonction de critères comme la criticité potentielle, la fréquence d'occurrence ou la présence d'indices exploitables. Par exemple, s'il existe 100 endpoints découverts mais seulement 5 avec des *HYPOTHESIS* de vulnérabilités sérieuses (ex. version d'un logiciel vulnérable connue), l'agent va focaliser sur ces 5 en premier.
  Place dans le pipeline : Au début de la phase **VERIFICATION** (juste après ENDPOINT_INTEL et les

éventuels scans passifs). C'est le point d'entrée de la phase de validation active.
Communication : L'agent consomme les données du graphe de connaissances – en particulier les nœuds *ENDPOINT* (avec leurs attributs comme techno détectée, ports ouverts, etc.) et les nœuds *HYPOTHESIS/VULNERABILITY* marqués théoriques issus de la phase précédente (par exemple générés par l'agent VulnCheck/Nuclei). Il peut exécuter des requêtes au graphe (ou via l'orchestrateur) pour lister les vulnérabilités théoriques présentes et leurs endpoints liés (*AFFECTS_ENDPOINT*). Sur cette base, il décide d'une liste d'endpoints (et éventuellement de vulnérabilités hypothétiques associées) à vérifier. Il transmet cette liste structurée à l'orchestrateur ou directement à l'agent suivant (`stack_policy_agent`). Concrètement, il peut ajouter un attribut aux nœuds *HYPOTHESIS* sélectionnés du graphe (ex: champ `triage=selected`), ou pousser un message à l'orchestrateur contenant les ID des endpoints retenus. L'orchestrateur assure la mémoire entre agents, ce qui permet de chaîner ces résultats proprement [5] [6].

- `stack_policy_agent` – *Agent de sélection de modules par stack technologique*
  Rôle : Mapper chaque endpoint sélectionné vers les modules de validation pertinents en fonction de sa technologie et de son contexte. En d'autres termes, il applique une *policy* de stack qui décide « Pour une application WordPress, exécuter les modules WP1, WP2, WP3… ; pour un API Node.js, exécuter les modules N1, N2… ». Il se base sur la matrice de modules de validation et sur les informations techniques de chaque endpoint.
  Place dans le pipeline : Immédiatement après le triage des endpoints (toujours dans la phase VERIFICATION).
  Communication : L'agent lit, pour chaque endpoint fourni par le `vuln_triage_agent`, son profil technique dans le graphe. Ces informations peuvent provenir de la phase ENDPOINT_INTEL (fingerprinting par outils comme Wappalyzer, bannières serveur HTTP, etc., déjà stockées dans le graphe). Par exemple, un nœud *ENDPOINT* pourrait avoir des propriétés `tech: WordPress`, `version: 5.8`, ou des relations vers des nœuds *TECHNOLOGY* plus détaillés. Si l'info n'est pas explicite, l'agent peut effectuer quelques vérifications légères (par ex. vérifier l'existence de `/wp-login.php` pour inférer WordPress, ou l'en-tête `X-Powered-By: Express` pour Node.js). Une fois la stack identifiée, il consulte la matrice des modules pour récupérer la liste des modules applicables. Il filtre ceux-ci selon les contraintes de mode (par exemple, si orchestrateur indique que le run est en mode Passive, il écartera les modules marqués Aggressive). L'agent produit en sortie un ensemble structuré du type : *Endpoint X -> [Module1, Module2,…]* pour chaque endpoint. Il peut enrichir le graphe en liant les endpoints aux modules (par ex. créer des nœuds *CHECK* ou ajouter un champ listant les IDs de modules à lancer) ou transmettre la liste à l'orchestrateur. Essentiellement, `stack_policy_agent` formalise le **plan de tests candidat** sans encore décider de l'ordre d'exécution.

- `validation_planner_agent``** – *Agent planificateur de la validation*
  Rôle : Élaborer un plan d'exécution optimisé des modules de validation sur les endpoints choisis. Cet agent prend la liste {endpoint -> modules} et détermine *comment* et *quand* exécuter chaque check de manière efficiente et sûre. Ses responsabilités incluent : ordonnancer les tests (peut-être en parallèle quand c'est possible, ou séquencement si certains tests doivent précéder d'autres), gérer la charge pour ne pas sur-solliciter un même endpoint (ex: éviter de lancer 10 fuzzings en même temps sur un serveur fragile), et respecter les limites d'intrusion (ralentir ou espacer les requêtes en mode Stealth, etc.). Il doit aussi intégrer les *timeouts* et

quotas prévus pour chaque test afin d'éviter qu'un module ne s'éternise ou consomme trop de ressources.

Place dans le pipeline : Phase VERIFICATION, après la sélection des modules par **stack_policy_agent**. Le planner est le dernier agent conceptuel avant l'exécution effective des checks.

Communication : L'agent récupère la liste des modules à exécuter pour chaque endpoint (soit via un input direct de l'orchestrateur, soit via le graphe mis à jour). Il peut aussi consulter le contexte global (par ex. le mode intrusif courant, le nombre total de modules à lancer, des paramètres de concurrence maxi définis par l'orchestrateur). En sortie, il produit un **plan d'action** structuré, essentiellement une file de tâches ou un petit workflow. Par exemple, il pourrait produire une liste ordonnée: [(Endpoint X, Module A), (Endpoint X, Module B), (Endpoint Y, Module C), …] avec éventuellement des groupes à exécuter en parallèle. Ce plan tiendra compte des dépendances (la plupart des checks sont indépendants, mais si certains résultats conditionnent d'autres checks, il peut intégrer cette logique). Le **validation_planner_agent` communique ce plan à l'orchestrateur pour exécution. Selon l'implémentation, soit l'agent invoque directement l'exécution de chaque module (via le service runner ou scanner-proxy) en respectant son plan, soit il fournit la séquence à l'orchestrateur qui se chargera de déclencher chaque tâche selon le scheduling défini. Dans les deux cas, l'agent collabore étroitement avec l'orchestrateur : il peut par exemple lancer des threads ou envoyer des requêtes au** scanner-proxy** pour démarrer les checks, tout en surveillant les retours. Il appliquera également les timeouts et quotas : l'orchestrateur/CrewAI prévoit déjà la possibilité de tuer une tâche qui dépasse un certain temps [7], ce que le planner utilisera pour chaque module (par ex. arrêter un fuzzing après X minutes).

- `evidence_curator_agent` – *Agent curateur des preuves et résultats*
  Rôle : Agréger et interpréter les résultats de tous les modules exécutés, afin de mettre à jour le graphe de vulnérabilités avec les conclusions *evidence-based*. Plus précisément, pour chaque test effectué, cet agent va décider s'il confirme une vulnérabilité, l'infirme, ou la laisse potentielle, puis créer ou mettre à jour les nœuds *VULNERABILITY* correspondants dans le graphe. Il attache les preuves collectées (hash ou extrait de réponse, métadonnées) et assigne le statut final (Confirmé, Probable, Théorique) avec un champ de confiance associé. Il veille à ne pas dupliquer les vulnérabilités : si un nœud existant *HYPOTHESIS* ou *VULNERABILITY* correspond, il sera mis à jour, sinon un nouveau nœud est créé.
  Place dans le pipeline : C'est l'étape finale de la phase VERIFICATION, se déroulant une fois que tous (ou la majorité) des modules planifiés ont été exécutés.
  Communication : L'agent reçoit en entrée les résultats bruts des checks. Ces résultats peuvent être fournis par le scanner-proxy/runner (par ex. sous forme d'un rapport JSON indiquant pour chaque module : success/fail, output/response captured). Le `evidence_curator_agent` va parcourir ces résultats : pour chaque module marquant un succès (c'est-à-dire répondant aux critères de vulnérabilité), il détermine de quelle vulnérabilité il s'agit. Par exemple, si `wordpress_env_leak` a réussi, on sait qu'il s'agit de la vulnérabilité "Fuite de fichier .env". L'agent va chercher dans le graphe si un nœud *VULNERABILITY* ou *HYPOTHESIS* correspondant existe déjà (éventuellement repéré par un identifiant de vulnérabilité ou par la relation au même endpoint). S'il existe et était au statut Théorique, il le promeut au statut Confirmé/Likely avec mise à jour du champ *status*. S'il n'existe pas

(par ex. on n'avait pas détecté cette hypothèse auparavant via du scanning), l'agent crée un nouveau nœud *VULNERABILITY* avec les attributs appropriés.
L'agent attache ensuite la **preuve** au nœud : cela peut être un attribut `evidence` contenant soit un extrait textuel (ex: les premières 100 caractères du .env) ou un hash de l'ensemble de la réponse (si on veut prouver l'existence sans exposer le contenu). Des attributs comme `evidence_type` (ex: "file_snippet", "http_header", etc.) et `confidence_score` peuvent également être ajoutés. La relation *AFFECTS_ENDPOINT* est créée si ce n'est pas déjà fait, liant la vulnérabilité confirmée à son endpoint.
Interactions : le curator communique avec le graphe via son API (ou un SDK) pour effectuer ces créations/maj de nœuds et relations. Il peut également signaler à l'orchestrateur le résumé des résultats (par ex. *N* vulnérabilités confirmées, *M* restées théoriques sans preuve). L'agent n'a pas besoin d'appeler les modules externes, il travaille localement avec les données en mémoire ou sur disque que les modules ont produites.

En résumé, la phase VERIFICATION de Recon Gotham est remaniée en une sous-pipeline orchestrée de façon déterministe : **triage des cibles → mapping techno/modules → planification → exécution → curation des preuves**. Chacun de ces rôles est confié à un agent distinct, ce qui s'aligne sur la philosophie CrewAI d'assigner des tâches spécialisées à des agents collaboratifs [4]. L'orchestrateur existant fait le lien entre eux, transmettant le contexte et les données d'un agent à l'autre, en profitant de la capacité de CrewAI à maintenir l'état entre exécutions [5]. Cette approche modulaire facilite l'évolution future (on peut ajouter un nouvel agent ou en modifier un sans tout repenser) et la traçabilité (chaque agent a un scope clair et laisse des logs de son action).

Notons que l'**agent existant de vérification** (service verification actuel) sera essentiellement enrichi/découpé en ces nouveaux agents. On ne remplace pas l'architecture : on la complète. Par exemple, si le service *verification* actuel exposait une API pour exécuter des requêtes HTTP simples, il peut être conservé et intégré comme composant appelé par le `validation_planner_agent` pour exécuter les modules (ou remplacé par le scanner-proxy plus sophistiqué). Les autres phases (OSINT, ACTIVE_RECON, etc.) restent inchangées, si ce n'est qu'elles alimenteront mieux la phase de vérification (ex: l'agent VulnCheck continuerait d'ajouter des *HYPOTHESIS* théoriques, qui seront ensuite traitées par les nouveaux agents de validation).

Modèle Standardisé de Validation des Vulnérabilités

Une fois les vérifications effectuées, Recon Gotham adoptera un **modèle uniformisé pour qualifier chaque vulnérabilité**. Ce modèle comporte un champ *status* à trois niveaux (**THEORETICAL**, **LIKELY**, **CONFIRMED**), un *score de confiance*, et une *preuve associée*, conformément à l'approche *evidence-based* souhaitée. Voici la signification de ces éléments :

- **Statut de vulnérabilité** :
- **THEORETICAL** (Théorique) indique une vulnérabilité *soupçonnée* ou *potentielle*, sans vérification concluante. C'est l'état par défaut aujourd'hui pour les issues détectées automatiquement. Par exemple, un scan qui repère une version logicielle vulnérable sans exploit direct donnera une vulnérabilité théorique. Dans la terminologie Rapid7 Nexpose, cela correspond aux *vulnérabilités potentielles* non vérifiées [8] [9].
- **LIKELY** (Probable) indique une vulnérabilité jugée très plausible, appuyée par des indices forts mais sans preuve absolue. Par exemple, si un module détecte une configuration anormale ou un

comportement suggérant fortement la présence de la faille (message d'erreur, version obsolète connue vulnérable), on la marque comme Likely. Cela correspond à un niveau de confiance intermédiaire (anciennement on pourrait dire *unconfirmed* avec forte probabilité selon certains scanners [8]). Ce statut est utile pour hiérarchiser les trouvailles : il y a de bonnes chances que ce soit exploitable, mais nous n'avons pas pu ou voulu effectuer la preuve ultime (souvent pour des raisons de sécurité ou de temps).

- **CONFIRMED** (Confirmée) est attribué lorsque la vulnérabilité est *directement vérifiée par une preuve concrète*. Cela signifie que l'outil a réussi une action qui démontre la faille sans ambiguïté. Par exemple, récupérer le contenu d'un fichier sensible, exécuter une commande bénigne à distance ou extraire une information protégée. C'est le niveau de confiance maximal. Dans Nexpose, ce sont les vulnérabilités que le scanner a pu *vérifier* explicitement [8]. Dans notre modèle, cela se traduit par le stockage d'une preuve irréfutable de l'existence de la faille.

- **Niveau de confiance (confidence)** : En complément du statut qualitatif ci-dessus, chaque vulnérabilité peut se voir attribuer un score ou un label de confiance. Par défaut, on peut associer **CONFIRMED = confiance 100%**, **LIKELY = confiance ~70%**, **THEORETICAL = confiance ~30%** (valeurs indicatives) ou utiliser des qualificatifs (High/Medium/Low). Ce score permet de nuancer au sein d'un même statut. Par exemple, deux vulnérabilités Likely pourraient avoir l'une 80% (indices très forts) et l'autre 60% (indices modérés). Le calcul ou l'assignation de ce score peut être simple (mapping fixe par type de test) ou plus fin (par l'agent curator, en fonction du contexte, du nombre de preuves différentes convergentes, etc.). L'important est de fournir au **PLANNER** ou à l'analyste humain un moyen de prioriser : une vulnérabilité Likely à 90% mérite plus d'attention qu'une autre à 50%. Ce champ *confidence* sera stocké dans le nœud de vulnérabilité du graphe.

- **Preuve (evidence)** : Pour chaque vulnérabilité qui n'est pas purement théorique, une preuve tangible est attachée. La preuve peut prendre plusieurs formes en fonction du type de vulnérabilité :

- *Extrait de réponse HTTP* : par exemple, quelques lignes d'un fichier sensible obtenu, une partie du contenu d'une page d'erreur, ou un élément d'API. On veillera à ne pas stocker d'information trop volumineuse ou hautement sensible en clair. Souvent, un extrait non critique suffit (p.ex., le début du fichier `.env` sans dévoiler la clé complète).
- *Hash cryptographique d'une ressource* : dans le cas où la donnée est trop sensible pour être même partiellement stockée, on peut calculer un hash (SHA-256 par ex.) du contenu récupéré. Ainsi on prouve que l'on a obtenu quelque chose (le hash pourra être recalculé lors d'une validation manuelle) sans exposer le contenu. Ce procédé est parfois utilisé par des scanners qui exploitent des failles de manière sûre – par exemple Invicti (Netsparker) qui, pour éviter les faux positifs, exploite la faille de façon non destructive et souvent prouve la réussite en lisant juste un *unique token/empreinte* [10]. On peut s'inspirer de cela en insérant un marqueur de test et en vérifiant sa présence (voir ci-dessous).
- *Metadonnées de réponse* : code HTTP obtenu, en-têtes spécifiques, temps de réponse, etc., si ceux-ci constituent en eux-mêmes la preuve. Par exemple, pour prouver un redirectionnel ouvert (*open redirect*), on peut montrer l'en-tête `Location` renvoyé reflétant un domaine externe.
- *Capture technique* : dans certains cas complexes, la preuve peut être un fichier attaché hors graphe (ex: capture d'écran pour une XSS dans l'interface, ou un pcap si réseau). Mais idéalement, tout texte/valeur sera encodé directement dans le graphe pour exploitation automatique.

Chaque nœud *VULNERABILITY* possèdera un champ ou un sous-objet `evidence` contenant la preuve. Le format peut être par exemple :

```
"evidence": {
    "type": "snippet",
    "value": "Database password: ****",
    "note": "Sha256(content)=abcdef123456..."
}
```

Ici on indique qu'on fournit un snippet et un hash du contenu complet masqué. Pour une exploitation directe de type commande, la preuve pourrait être "commande exécutée avec succès, résultat = X". Pour une injection SQL en lecture, la preuve pourrait être "extraction de la version DB: 5.7.34".
L'**evidence_curator_agent** est chargé de structurer ces preuves de manière cohérente et uniforme lors de l'insertion dans le graphe.

- **Mise à jour ou création de vulnérabilité existante** : Si le scan de validation confirme une vulnérabilité qui avait été initialement marquée comme hypothétique, le même nœud peut être promu en modifiant ses propriétés. Par exemple, une *HYPOTHESIS* "Laravel .env leak possible" devient un *VULNERABILITY* confirmée : on changera son status de THEORETICAL à CONFIRMED, on ajoutera la preuve et le score de confiance, et éventuellement on changera son étiquette/type pour indiquer que ce n'est plus juste une hypothèse. Ceci préserve l'historique (c'est le même objet, mis à jour). Si en revanche un module révèle une vulnérabilité qui n'avait pas du tout été envisagée (nouvelle faille découverte spontanément), l'agent curator va créer un **nouveau** nœud *VULNERABILITY* et le lier à l'endpoint, avec directement le status Confirmé/Probable. Dans tous les cas, chaque vulnérabilité dans le graphe, à la fin du processus, aura un des trois statuts et potentiellement une preuve si elle est Confirmed/Likely. Celles restées théoriques (non reproduites) pourront être annotées différemment ou laissées avec status=THEORETICAL et confidence faible.

Ce modèle standardisé s'aligne avec les pratiques du secteur pour réduire les faux positifs et prioriser les failles avérées. Par exemple, la technique de **Proof-Based Scanning** d'Invicti consiste à automatiquement exploiter de façon safe les vulnérabilités afin de fournir une preuve indéniable, éliminant les faux positifs [10]. De même, Rapid7 Nexpose différencie clairement les vulnérabilités confirmées de celles simplement potentielles basées sur des versions logicielle, ces dernières n'étant pas prouvées [8]. En adoptant ces notions, Recon Gotham renforcera la crédibilité de ses rapports (chaque finding Confirmé est justifié par une preuve) et optimisera le travail de l'analyste humain en lui fournissant des éléments concrets à investiguer.

Spécification JSON des Modules de Validation

Afin de garantir que les modules de vérification puissent être exécutés de manière générique par le runner sandboxé, nous définissons un **schéma JSON standard** pour les décrire. Cette spécification permet à l'orchestrateur (ou au *scanner-proxy*) de comprendre comment exécuter le test sans ambiguïté. Les principaux champs proposés sont :

- `id` (string) : Identifiant unique du module de validation. Par convention, on peut utiliser un format combinant la techno et la vulnérabilité, par ex. `"wordpress_xmlrpc_ping"` ou

`"laravel_env_leak"`. Cet ID servira de référence dans les communications entre agents (p. ex. le planner réfère aux modules par leur id) et dans le graphe (un nœud vulnérabilité pourrait stocker l'id du module qui l'a confirmé). L'identifiant est aussi utile pour la traçabilité des résultats et pour éviter les doublons.

- `target` (string ou object) : La cible spécifique du test. Le plus souvent ce sera une URL ou un chemin relatif à l'endpoint. Par exemple `"target": "/.env"` ou `"target": "/wp-json/wp/v2/users"` pour un module web, à combiner avec l'URL de base de l'endpoint. Si un module doit s'appliquer sur plusieurs URLs ou paramètres, ce champ peut être un objet plus complexe (liste d'URLs, ou modèle paramétré). Dans une version simple, l'orchestrateur/runner peut concaténer l'URL de l'endpoint avec ce chemin. Il faut également prévoir la possibilité d'inclure des **placeholders** si nécessaire (par ex. `<DOMAIN>` ou `<IP>` qui seront remplacés par l'hôte cible, ou `<USER>` si le module prend un param utilisateur). Une bonne pratique est de garder le module auto-suffisant pour une cible donnée, c'est-à-dire éviter d'y mettre des placeholders dynamiques complexes. Le champ target pourrait aussi contenir un port ou un protocole si besoin, bien que souvent implicite (HTTP/HTTPS déterminé par l'endpoint).

- `method` (string) : La méthode HTTP à utiliser pour la requête du module, par exemple `"GET"`, `"POST"`, `"HEAD"`, etc. Ce champ est essentiel pour appliquer les règles d'engagement – il sera lu par le planner ou le runner pour autoriser/refuser l'exécution selon le mode. De plus, cela permet de formater correctement la requête. Si des modules non-HTTP étaient envisagés (ex: module faisant du SMB, ou du DNS), on pourrait étendre à un champ protocole, mais dans le contexte web/API actuel, `method` suffit.

- `headers` (object) : Un dictionnaire des en-têtes HTTP à inclure dans la requête. Par exemple :

```
"headers": {
    "Content-Type": "application/json",
    "Accept": "application/json"
}
```

On peut aussi y prévoir des entêtes conditionnels, comme l'**Authorization** ou des **Cookie** pour les tests authentifiés. Dans ce cas, soit on stocke dans le module un placeholder (ex: `"Authorization": "Bearer <TOKEN>"`) que le runner remplacera par le vrai token fourni dans la configuration de la cible, soit on laisse vide et le runner ajoute lui-même les credentials s'il en a en contexte. Ce champ permet en tout cas de reproduire fidèlement des requêtes précises nécessaires à certains tests (ex: content-type XML pour XMLRPC, header spécifique d'API, etc.).

- `body` (string ou object) : Le corps de la requête HTTP, pour les méthodes comme POST/PUT. Ce champ peut contenir une chaîne (typiquement du JSON, XML ou formulaire URL-encodé) ou éventuellement une structure (ex: un objet JSON qui sera sérialisé). Si le body est binaire ou très long, on préférera peut-être que le module référence un payload externe, mais idéalement tout tient dans le JSON. On pourra utiliser des placeholders ici aussi si nécessaire. Par exemple, un module de brute forcing léger pourrait avoir `"body": "username=admin&password=<PASSWORD>"` et un mécanisme de wordlist associé (voir plus bas). **Important** : tous les bodies des modules de notre

matrice seront choisis pour être **non destructifs** et sans effet de bord significatif, conformément aux ROE. Souvent, il s'agit de requêtes de lecture (dump d'une info, test d'accès) ou d'exploitation passive (extraction de données). Pas de modification de données côté cible, pas de charges dangereuses.

- `stop_conditions` (object) : Ce champ décrit les conditions d'arrêt ou d'achèvement du module. Il s'agit en quelque sorte de la logique de contrôle de la boucle de test si le module en nécessite une (notamment dans les cas de fuzzing ou de tests itératifs). Quelques exemples de sous-champs possibles :

- `"max_tries": 5` – Nombre maximum de tentatives (par ex. essayer 5 inputs différents puis arrêter).
- `"stop_on_match": true` – Indique que dès qu'une condition de succès est rencontrée (voir expected_proof), on arrête les itérations supplémentaires. Utile pour ne pas continuer à fuzz une fois qu'on a trouvé un résultat positif.
- `"stop_on_fail": ...` – On pourrait définir une condition d'arrêt sur échec, par ex. si on reçoit 5 réponses d'affilée avec le même code d'erreur, on arrête car probablement ça ne passe pas.
- `"time_limit": 30` – en secondes, si un module ne doit pas courir plus de X secondes indépendamment du nombre d'essais.
- `"concurrency": 1` – pour limiter le nombre de requêtes en parallèle dans ce module (certains fuzzings pourraient être parallélisés, d'autres pas).

Ces conditions d'arrêt sont surtout pertinentes pour les modules complexes. Pour un module simple (une requête unique), on peut mettre `max_tries: 1` et `stop_on_match: true` par défaut. Pour un module de fuzzing (par ex. tester une liste de chemins pour trouver des backups), on indiquerait la taille de la wordlist et `stop_on_match`. **Remarque** : la wordlist ou le set d'inputs en lui-même pourrait être référencé dans le module spec soit directement (ex: `"payloads": ["admin.zip", "backup.zip", ...]`), soit indirectement (ex: `"wordlist": "small-backups.txt"` si le runner a accès à des fichiers de wordlists locales). On pourrait inclure cela dans `stop_conditions` ou dans un champ séparé. Pour garder simple, on peut imaginer que si un module nécessite une wordlist ou plusieurs tentatives, le script Python du module gère cela en interne, et `stop_conditions` sert à informer l'orchestrateur des limites pour supervision.

- `expected_proof` (object/string) : Ce champ décrit ce que le module considère comme preuve de réussite – en d'autres termes, comment déterminer que la vulnérabilité est présente à partir de la réponse de la cible. C'est équivalent au concept de *matcher* dans les templates Nuclei [3] . On peut le structurer de plusieurs façons :
- **Pattern de réponse attendue** : Par exemple, `"pattern": "root:x:0:0:"` pour détecter une ligne d'un fichier passwd dans une LFI, ou `"pattern": "WordPress"` pour repérer le mot WordPress dans un readme. Ce pattern peut être une simple sous-chaîne ou une regex. On peut prévoir des champs séparés pour cibler une partie de la réponse (headers vs body). Par exemple:

```
"expected_proof": {
    "in": "body",
    "pattern": "APP_KEY="
}
```

ou

```
"expected_proof": {
    "in": "headers",
    "name": "X-Debug-Token"
}
```

pour dire « la preuve de succès est la présence de l'en-tête X-Debug-Token dans la réponse ».

- **Condition logique** : Parfois la preuve peut être plus complexe qu'une simple recherche de texte. On pourrait avoir `"status_code": 200` et un pattern dans le body, etc. On peut donc permettre que `expected_proof` combine plusieurs critères (tous devant être remplis). Par exemple:

```
"expected_proof": {
  "status": 200,
  "patterns": ["wp-config.php", "DB_NAME"]
}
```

signifierait qu'on attend un code 200 ET les textes "wp-config.php" et "DB_NAME" dans la réponse (ce qui indiquerait qu'un dump de config WP a réussi).

- **Extraction** : en plus de détecter, on peut vouloir extraire la donnée précise à stocker. Par exemple pour la version WordPress, on pourrait mettre `"extract": "WordPress ([0-9\\.]+)"` avec une regex de capture, et le script renverrait la version trouvée. Ce n'est pas indispensable pour la logique de vérification (l'important est de savoir si vulnérable ou pas), mais c'est utile pour le reporting. On peut donc inclure des instructions d'extraction dans ce champ ou à part. Dans l'optique simple, on peut dire que si un pattern avec capture est fourni, le runner extrait la valeur capturée comme *preuve* (value).

En résumé, `expected_proof` doit décrire comment reconnaître que la vulnérabilité a été exploitée ou l'information divulguée. Le **runner** utilisera cela pour marquer le résultat comme succès ou échec. En cas de succès, il renverra aussi la donnée extraites ou l'indication correspondante qui sera passée à l'evidence_curator.

- `max_status` (string) : Ce champ indique le niveau maximal de certitude/confidentialité que ce module peut atteindre. En effet, tous les modules ne mènent pas à une vulnérabilité confirmée. Par exemple, on peut définir qu'un module de détection de version (qui ne fait que collecter la version du logiciel) ne peut au mieux rendre la vulnérabilité que *LIKELY* (car connaître une version vulnérable n'est pas une exploitation directe, c'est un indice fort). Au contraire, un module qui exfiltre une info sensible ou réalise un exploit de lecture aura `max_status: "CONFIRMED"`. Ce champ guide l'evidence_curator dans l'assignation du statut : même s'il trouve la preuve, il ne devrait pas marquer au-delà de ce niveau. Donc si `max_status` est Likely et qu'on a un résultat, on marquera la vulnérabilité "Likely" malgré la preuve, pour signifier qu'on a un indice solide mais pas une exploitation *complète*. Dans la plupart des cas, un module exploitant effectivement la faille aura max_status=CONFIRMED, tandis qu'un module purement déductif aura Likely. (Si un module n'aboutit pas, la vulnérabilité reste Theoretical ou est infirmée, suivant la logique du curator).

- `description`, `references`, `severity`... : Bien qu'ils n'aient pas été listés explicitement dans la question, on peut envisager d'autres champs informatifs dans la spec du module, similaires à ceux des templates Nuclei (nom de la vuln, description textuelle, lien CVE ou exploit DB, sévérité CVSS, tags technos, etc. [11]). Ces champs ne sont pas utilisés par le runner mais pourraient servir pour la documentation et le reporting. Par exemple `"description": "Checks for exposed Laravel .env configuration file."`. Également un champ `"severity": "high"` ou un score pourrait être utile. Ces informations peuvent rester dans la définition du module (fichier JSON ou base de modules) et ne pas forcément être transmises à chaque exécution, sauf si le runner en a besoin pour logguer. Néanmoins, pour garder notre spec focalisée, on considère ces champs comme optionnels/documentaires.

Voici à quoi pourrait ressembler un **exemple de définition JSON** pour un module (simplifié pour illustrer) :

```
{
  "id": "laravel_env_leak",
  "target": "/.env",
  "method": "GET",
  "headers": {},
  "body": null,
  "stop_conditions": {
    "max_tries": 1,
    "stop_on_match": true,
    "time_limit": 10
  },
  "expected_proof": {
    "status": 200,
    "pattern": "APP_KEY="
  },
  "max_status": "CONFIRMED",
  "description": "Expose Laravel .env file containing sensitive config",
  "severity": "critical"
}
```

Dans cet exemple, le module laravel_env_leak est bien identifié, effectue une unique requête GET, attend un code 200 et la présence de `APP_KEY=` dans la réponse, et s'il trouve cela il pourra confirmer la vulnérabilité. Le runner lira cette spec et saura exactement quoi faire et quoi chercher.

Un autre exemple pour un module de fuzzing de backup WordPress pourrait être :

```
{
  "id": "wordpress_backup_fuzz",
  "target": "/",
  "method": "GET",
  "headers": {},
  "body": null,
```

```
  "stop_conditions": {
    "wordlist": "wordlists/wp_backups.txt",
    "stop_on_match": true,
    "max_tries": 50,
    "concurrency": 2
  },
  "expected_proof": {
    "status": 200,
    "pattern": "WordPress"
  },
  "max_status": "CONFIRMED"
}
```

Ici, on indiquerait au runner de tenter jusqu'à 50 chemins (tirés du fichier de wordlist) sur l'URL de base (par ex. `backup.zip`, `site.bak`, etc.), de lancer 2 requêtes à la fois, et de s'arrêter dès qu'une réponse 200 contenant "WordPress" est trouvée (signe qu'on a récupéré un zip du site WordPress par exemple). La preuve attendue serait la présence du mot WordPress (ou autre signature) dans l'archive récupérée ou la page listant le contenu. Le runner renverrait celui qui a matché.

Intégration avec le runner (scanner-proxy) : Cette spec JSON est conçue pour être transmise au service d'exécution isolé (scanner-proxy). En pratique, le `validation_planner_agent` ou l'orchestrateur va envoyer une requête à scanner-proxy du style : `POST /run_check` avec ce JSON en payload, ou mettre ces infos dans un job queue que le runner consomme. Le scanner-proxy, côté implémentation, saura interpréter ce JSON : il exécutera la requête HTTP correspondante (via un outil HTTP client ou un script Python prédéfini), appliquera éventuellement la logique de fuzz/itération si demandée, et évaluera les conditions de succès. Ensuite, il renverra un résultat structuré, par exemple :

```
{
  "id": "laravel_env_leak",
  "target": "https://target-app.com/.env",
  "status": "SUCCESS",
  "evidence": "APP_KEY=base64:abcd... (truncated)",
  "proof_hash": "71b1a3f..."}
```

ou `"status": "FAIL"` si rien n'a matché. Le champ `status` ici n'est pas directement CONFIRMED/LIKELY, c'est plutôt succès/échec du check. Le mapping vers CONFIRMED/LIKELY se fera côté evidence_curator en comparant avec `max_status` du module. Par exemple, si `status=SUCCESS` et `max_status=CONFIRMED`, on marque vulnérabilité confirmée; si `max_status=LIKELY`, on marque seulement probable même en cas de succès (car c'est tout ce qu'on peut conclure). En cas d'échec du check, soit la vulnérabilité reste Théorique (si on avait une hypothèse initiale), soit on peut la rétrograder ou marquer "non reproduite" – c'est une décision de l'analyste ou du planner selon les cas (typiquement, on ne supprimera pas forcément l'hypothèse, on la laissera en théorique avec faible confiance, car absence de preuve != preuve d'absence).

Intégration Locale et Exécution Isolée

Tous les éléments ci-dessus s'intègrent dans Recon Gotham **sans modification fondamentale de l'architecture** : on ajoute des composants, on n'en retire pas. L'orchestrateur CrewAI existant reste le chef d'orchestre de l'enchaînement. Chaque nouveau agent (triage, policy, planner, curator) peut être implémenté soit en tant que nouvelles microservices containerisés (par exemple un service Python pour chacun, que l'orchestrateur appelle), soit en tant que nouveaux rôles au sein du CrewAI orchestrator lui-même. L'important est qu'ils interagissent via les interfaces prévues : accès au graphe commun pour lire/écrire les informations, et messages de coordination via l'orchestrateur.

Un point clé est l'**externalisation de l'exécution des scripts de vérification**. Auparavant, l'exécuteur de scripts Python était probablement intégré ou sans isolation poussée. Désormais, nous introduisons un **runner isolé (scanner-proxy)** pour lancer les modules de check. Ce runner sera très probablement un conteneur (Docker) distinct, dans lequel seront montés les outils nécessaires (par ex. un Python avec les dépendances requests, etc., ou même des utilitaires comme `curl` / `nmap` / `nuclei` si certains modules en ont besoin). L'utilisation de conteneurs pour chaque tâche garantit une isolation et un nettoyage facile des tâches de scan [12]. Chaque requête de check aboutira à l'exécution d'un script dans un environnement confiné : - **Isolation réseau** : Le conteneur du runner sera configuré pour n'autoriser que les connexions vers la portée de l'audit (par ex. filtrage egress pour ne sortir que vers l'IP/domaines de la cible). Aucune connexion externe (ex: internet général) ne sera possible, empêchant toute fuite de données ou usage hors scope. - **Limitations de ressources** : Via Docker/OS, on allouera des quotas CPU/Mémoire au runner ou à chaque job, afin qu'un module mal conçu ne monopolise pas toute la machine. L'orchestrateur peut aussi imposer un **timeout** global par module (ex: si pas fini en 2 minutes, kill) [7]. De plus, le `stop_conditions.time_limit` mentionné dans la spec sera appliqué en interne pour arrêter la tâche à temps. - **Aucunes dépendances externes** : Tous les outils ou scripts nécessaires aux modules doivent être présents localement dans l'image du runner. Par exemple, si on décide d'utiliser *nuclei* en interne pour certains checks, l'outil et les templates seront embarqués dans le conteneur. Idéalement toutefois, on privilégie des modules implémentés en Python natif ou requêtes directes pour garder le contrôle. Aucun appel à un service cloud tiers ne sera effectué (conformément aux exigences de ne pas dépendre de LLM externes ou autres). Toute intelligence embarquée (CrewAI) reste locale et maîtrisée [1]. - **Logging et traçabilité** : Le scanner-proxy renverra les résultats structurés, mais on peut aussi conserver les logs bruts de ce qui a été envoyé/reçu (au moins pour audit interne). L'orchestrateur pourrait logguer chaque action d'agent et chaque module exécuté (avec timestamp), conformément à ce qui était envisagé dans l'architecture initiale [13]. Cela facilite le debug et le reporting (p.ex. pouvoir montrer la requête envoyée et la réponse en cas de doute). - **Sécurité** : Exécuter les modules en sandbox protège la plate-forme Recon Gotham elle-même. Si un script était compromis ou exploitait une faille, il n'affecterait que le conteneur isolé, pas l'orchestrateur ni le système hôte. De plus, en interdisant les instructions potentiellement dangereuses (pas de `DELETE` sans allowlist, etc.), on se prémunit contre des erreurs de scripts. L'orchestrateur agit comme garde-fou en validant que chaque module lancé est autorisé (grâce aux champs method/flags dans la spec).

En termes de communication, voici un scénario d'exécution end-to-end pour illustrer l'intégration : 1. **Triage & Policy** : Supposons qu'après Endpoint Intel, 10 endpoints et 3 hypothèses de vulnérabilités soient identifiés. `vuln_triage_agent` en sélectionne 2 critiques (ex: un WordPress et une API Laravel suspecte). `stack_policy_agent` voit "WordPress" → modules WP, *"Laravel" → modules L*. Il liste par exemple 3 modules WP et 2 modules Laravel à exécuter. 2. **Planification** : Le `validation_planner_agent` décide de lancer d'abord les modules les moins intrusifs en parallèle – par ex, lancer

`wordpress_version_disclosure` (GET) et `laravel_env_leak` (GET) en même temps, car deux cibles différentes. Ensuite lancer `wordpress_xmlrpc_ping` (POST) car en mode Balanced c'est ok, puis `wordpress_user_enum` etc. Il envoie chaque module au runner via scanner-proxy, ou bien en lots si parallélisé. 3. **Exécution isolée** : Le scanner-proxy reçoit par ex. la requête pour `laravel_env_leak` sur *target.com*. Il exécute le GET, obtient une réponse 200 avec contenu `.env`. Il trouve `APP_KEY=` dans le body, donc succès. Il retourne au planner (ou orchestrateur) un objet résultat avec evidence "APP_KEY=...". 4. **Curation** : Une fois tous les modules terminés, l'orchestrateur appelle `evidence_curator_agent` avec l'ensemble des résultats. Pour chaque, le curator met à jour le graphe. Le WordPress version disclosure a réussi, ça correspond à une vulnérabilité "WordPress outdated version" qui était hypothétique – on la passe en *LIKELY* avec la version trouvée en preuve (ex: "WordPress 5.2, vulnérable à..."). Le laravel_env_leak a réussi, ça correspond à une nouvelle vulnérabilité (si on ne l'avait pas déjà en hypothèse) – on crée un nœud VULNERABILITY "Exposed .env file" relié à l'endpoint Laravel, statut *CONFIRMED*, preuve = hash du .env et extraits des clés. Etc. S'il y avait des modules échoués, on pourrait laisser les hypothèses correspondantes en THEORETICAL (voire diminuer leur confiance), car l'exploitation n'a pas abouti. 5. **Reporting** : La phase REPORTING peut ensuite puiser dans le graphe final et formater un rapport où les vulnérabilités confirmées sont mises en avant (avec leurs preuves en annexe), les likely sont notées comme à vérifier manuellement ou à corriger préventivement, et les théoriques éventuellement listées à titre d'information (ou filtrées si trop faible confiance). Le gain, c'est que le rapport final se base sur des faits avérés pour l'essentiel, réduisant le bruit des faux positifs.

En conclusion, cette évolution de Recon Gotham apporte une couche de **vérification active et déterministe** des failles par des modules spécialisés, tout en restant **alignée avec l'architecture existante** et les **règles d'engagement**. L'utilisation d'agents CrewAI dédiés s'inscrit dans la logique modulable de pipelines red team modernes [4], et l'**approche evidence-based** s'inspire des meilleures pratiques du domaine (preuve à l'appui pour éliminer les faux positifs [10], classification Confirmé/Probable/Théorique comme chez les scanners pros [8]). Le tout fonctionne localement, orchestré de façon sécurisée (conteneurs isolés, pas de dépendances tierces), assurant que l'outil peut être déployé on-premise et opéré en toute confidentialité. Cette mise à niveau fera de Recon Gotham un outil de Red Teaming **profond**, capable non seulement de trouver des pistes de vulnérabilités, mais de les valider concrètement et de fournir les éléments de preuve nécessaires à leur exploitation ou correction.

Sources: Les concepts présentés s'appuient sur des approches éprouvées en industrie, telles que l'orchestration d'agents spécialisés via CrewAI [4], l'isolation des tâches de scan dans des conteneurs dédiés [12], l'automatisation des exploitations non destructives pour confirmation de vulnérabilités [10], et la catégorisation des failles par niveau de confirmation [8] [9]. Ces références confirment la pertinence de l'architecture proposée dans un contexte d'audit de sécurité Red Team moderne et efficace.

---

[1] [2] [4] [5] [6] [7] [12] [13] Red Team Reconnaissance Pipeline Plan.pdf
file://file-ARpYafShGEP7LdEghWCjmf

[3] [11] Nuclei Template Structure - ProjectDiscovery Documentation
https://docs.projectdiscovery.io/templates/structure

[8] [9] Vulnerabilities FAQs
https://help.rapid7.com/nexpose/en-us/Files/Vulnerabilities_FAQ.html

[10] Avoid False Positives with Proof-Based Scanning | Invicti
https://www.invicti.com/features/proof-based-scanning