# jOpenAgent

Java Object-Oriented Agents

# Table des matières

# 1. Introduction et vue d'ensemble

jOpenAgent est un framework de création d'agents IA.

Un agent n'est rien de plus qu'une classe Java ordinaire.
Les champs sont l'état, les méthodes ordinaires sont des capacités déterministes, et les méthodes annotées @Generative sont déléguées à un modèle de langage (LLM) à l'exécution. Il n'y a pas de modèle d'objet spécifique au framework à apprendre au-delà de l'héritage Java classique, ni de langage de configuration séparé, un agent n'est qu'une classe, et son comportement n'est que du code.
jOpenAgent s'inspire en partie de NVIDIA Object Oriented Agents (NOOA).

## 1.1 L'exemple canonique

Le plus petit agent jOpenAgent complet est une classe étendant org.jopenagent.core.Agent, portant une annotation @SystemPrompt au niveau de la classe et une méthode @Generative dont le corps se contente d'appeler generate(...) :

```
@SystemPrompt("You are an agent specializing in analyzing customer feedback.")
public class FeedbackAgent extends Agent {

    public FeedbackAgent(AgentConfig config) {
        super(config);
    }

    @Generative("Analyze customer feedback for sentiment and key topics in one
sentence.")
    public String analyzeFeedback(String text) {
        return generate(text); // le point d'interception
    }
}
```

Il s'utilise comme n'importe quel autre objet Java :

```
FeedbackAgent agent = new
FeedbackAgent(AgentConfig.builder().llmClient(llmClient).build());
String result = agent.analyzeFeedback("Great product, but shipping was slow");
System.out.println(result);
```

Modifier le texte @Generative de analyzeFeedback change son comportement, la chaîne d'instruction est le prompt.

## 1.2 Ce que couvre ce manuel

Ce manuel est organisé pour amener un lecteur de zéro à opérationnel, puis servir de référence permanente :

Les chapitres 2 à 14 couvrent successivement : la mise en place du projet, les mécanismes fondamentaux, les quatre stratégies de génération, l'intégration des outils et du MCP, la mémoire,

les compétences (skills), l'historique de conversation, les blocs de contexte, le contenu multimodal, la traçabilité et l'observabilité, le bac à sable d'exécution de code, le harnais d'évaluation, et les deux clients LLM fournis. Le chapitre 15 parcourt les dix-neuf exemples pédagogiques fournis, dans l'ordre. Le chapitre 16 rassemble en un seul endroit les consignes de sécurité du projet. Les annexes fournissent un inventaire complet des paquets et classes (annexe A), une référence d'API détaillée pour les classes les plus centrales du framework (annexe B) et (pour les mainteneurs et contributeurs) les raisons de conception expliquant où et pourquoi jOpenAgent diverge de NOOA (annexe C).

### 1.3 Parité fonctionnelle avec NOOA, en un coup d'œil

jOpenAgent implémente l'intégralité du mécanisme central, les quatre stratégies de génération (dont l'une, TOOL_CALLING, n'a pas d'équivalent direct dans NOOA), la sortie structurée, la divulgation progressive, les blocs de contexte, la synthèse de l'historique, les compétences, l'intégration d'outils MCP, la mémoire inter-sessions, le contenu multimodal, l'export de traces ATIF, un trio de visualiseurs de traces (web/Swing/terminal), deux clients LLM, un bac à sable d'exécution de code avec des moteurs en processus et hors processus, et un harnais d'évaluation.

## 2. Prise en main

### 2.1 Structure du projet

jOpenAgent est une librairie Java 21 (ou supérieure) sans dépendances, aucun JAR externe n'est requis pour le framework lui-même :

| Fonctionnalités | Implémentation |
|---|---|
| HTTP | java.net.http.HttpClient uniquement |
| JSON | Le parseur org.jopenkit.json fourni, étendu par org.jopenagent.json |
| Bac à sable | jdk.jshell |
| Serveur web embarqué | com.sun.net.httpserver (jdk.httpserver) |
| Processus serveur MCP | java.lang.ProcessBuilder |
| Tests uniquement | JUnit 5 |

### 2.4 Configurer un client LLM

Chaque programme d'exemple de org.jopenagent.examples récupère son client LLM à partir de variables d'environnement, via org.jopenagent.examples.ExampleSetup, vérifiées dans cet ordre de priorité :

| Variable(s) | Effet |
|---|---|
| `ANTHROPIC_API_KEY` | Avec JOPENAGENT_MODEL (ex. claude-sonnet-5), construit un AnthropicClient contre l'API Anthropic hébergée. |
| `OPENAI_API_KEY` | Avec JOPENAGENT_MODEL (ex. gpt-4o-mini), construit un OpenAiClient contre l'API OpenAI hébergée. |
| `JOPENAGENT_LMSTUDIO_MODEL` | Construit OpenAiClient.forLmStudio(model, baseUrl), aucune clé API nécessaire. JOPENAGENT_LMSTUDIO_BASE_URL optionnelle, par défaut http://localhost:1234. |
| `JOPENAGENT_OLLAMA_MODEL` | Construit OpenAiClient.forOllama(model, baseUrl), aucune clé API nécessaire. JOPENAGENT_OLLAMA_BASE_URL optionnelle, par défaut http://localhost:11434. |
| `JOPENAGENT_EMBEDDING_MODEL` | Fait passer ExampleSetup.createEmbedder() du HashingEmbedder par défaut à un véritable OpenAiEmbedder, avec les mêmes identifiants de fournisseur que ci-dessus. |

## 2.5 Exécuter un exemple

Chaque classe de org.jopenagent.examples possède sa propre méthode main() sans aucune modification de code, seulement les variables d'environnement ci-dessus. Par exemple, avec ANTHROPIC_API_KEY et JOPENAGENT_MODEL définies :

```
$ export ANTHROPIC_API_KEY=sk-ant-...
$ export JOPENAGENT_MODEL=claude-sonnet-5
# puis exécuter org.jopenagent.examples.E01_FirstGenerationMethod en tant que Java
Application
```

Le chapitre 15 parcourt les dix-neuf exemples fournis (E01-E19) dans l'ordre où ils sont conçus pour être lus, chacun illustrant un domaine fonctionnel précis couvert plus loin dans ce manuel.

**Les exemples MCP nécessitent Node.js**

E11_McpTools requiert en plus que npx (Node) soit disponible dans le PATH, car il lance le serveur de référence officiel @modelcontextprotocol/server-everything en tant que processus fils. Sous Windows, org.jopenagent.mcp.McpClient (via org.jopenagent.util.ProcessLaunch) redirige automatiquement et de façon transparente ces commandes basées sur npx à travers cmd.exe /c, car ProcessBuilder ne peut pas lancer directement des scripts .cmd/.bat.

# 3. Concepts fondamentaux

## 3.1 Les agents sont des objets Java

Un agent jOpenAgent est une classe ordinaire étendant org.jopenagent.core.Agent.

Les champs d'instance typés constituent l'état de l'agent et visibles par réflexion via la méthode describe() et les mécanismes de bac à sable code-as-action et d'outils-via-self, à moins d'être marqués @Hidden.

Les méthodes ordinaires avec un vrai corps sont des capacités déterministes, appelables directement par du code Java tiers, par du code généré exécuté dans le bac à sable, ou en tant qu'outils natifs de function-calling.

Les méthodes annotées @Generative, se terminant par un appel à generate(...), sont déléguées au LLM.

## 3.2 La méthode generate()

Le système utilise StackWalker pour identifier son appelant à l'exécution. Il lit l'annotation @Generative de cette méthode ainsi que son type de retour déclaré par réflexion, puis délègue à la Strategy configurée. Les paramètres de la méthode destinés à être montrés au LLM doivent être passés explicitement en paramètres :

```
@Generative("Classify the ticket.")
protected Category classify(String text) {
    return generate(text);
}
```

## 3.3 Instructions au niveau classe et au niveau méthode

@SystemPrompt (au niveau classe) porte le prompt système global d'un agent.

Sa valeur est développée via un PromptTemplate. Si une sous-classe ne déclare pas son propre @SystemPrompt, celui de la superclasse annotée la plus proche est utilisé.

@Generative (au niveau méthode) porte l'instruction (prompt) propre à cette capacité précise :

- value() est le texte d'instruction
- maxRetries() surcharge le budget de nouvelles tentatives global de l'agent pour cette méthode
- strategy() sélectionne un StrategyKind pour cet appel, par défaut StrategyKind.INHERIT (qui se replie sur AgentConfig.getDefaultStrategy()).

org.jopenagent.core.Doc est le mécanisme équivalent pour les méthodes et champs ordinaires (non-@Generative).

## 3.4 AgentConfig

Les réglages propres à chaque agent sont injectés via AgentConfig, construit avec AgentConfig.builder()...build() :

```
AgentConfig config = AgentConfig.builder()
    .llmClient(llmClient)          // requis
    .maxRetries(3)                 // défaut 3
    .defaultStrategy(StrategyKind.PREDICT)  // défaut PREDICT
    .maxCodeActIterations(8)        // défaut 8
    .historyConfig(HistoryConfig.builder().build())
    .embedder(new HashingEmbedder())   // ou .memoryManager(...) directement
    .sandboxConfig(SandboxConfig.builder().build())
    .build();
```

build() lève une IllegalStateException si llmClient n'a jamais été défini, ou si maxRetries est inférieur à 1. Si .memoryManager(...) et .embedder(...) sont tous deux fournis, memoryManager est prioritaire ; si aucun des deux n'est fourni, le comportement par défaut construit un MemoryManager sur un InMemoryMemoryStore et un HashingEmbedder.

## 3.5 Divulgation progressive et visibilité

Agent.describe() restitue un résumé des champs et méthodes visibles d'un agent en délégant à org.jopenagent.agentdoc.Introspector, qui parcourt la hiérarchie de classes jusqu'à (sans l'inclure) Agent lui-même, en dédupliquant les membres redéfinis.

La même forme littérale {doc(self)} peut être intégrée directement dans une instruction @SystemPrompt ou @Generative, permettant au LLM de découvrir les capacités propres de l'agent au moment de la construction du prompt, plutôt que de les faire lister manuellement par le développeur.

La visibilité suit un modèle visible par défaut piloté par les modificateurs Java ordinaires. Les champs/méthodes publics sont visibles par défaut ; @Hidden exclut un membre autrement visible de doc(self), du bac à sable et des schémas d'outils ; @Shown réexpose un membre autrement caché (non public). @NoTrace exclut en plus un membre de la capture d'attributs de span de trace.

## 3.6 Sortie structurée

org.jopenagent.json.ObjectBinder est une couche JSON<->Java entièrement automatique, fondée sur la réflexion, prenant en charge les records, les POJO, List/Set/Map, Optional, les énumérations et les primitives/wrappers, sans aucune étape d'enregistrement par classe requise nulle part.

Lorsque le type de retour déclaré d'une méthode @Generative est, par exemple, un record :

```
public record ReviewSummary(int rating, String sentiment, List<String> highlights) {}

@Generative("Summarize this product review.")
public ReviewSummary summarize(String review) {
    return generate(review);
}
```

PredictStrategy (voir chapitre 4) demande au modèle un JSON correspondant à cette forme, puis ObjectBinder l'analyse, retire les éventuelles balises de code markdown, et le lie par réflexion. Cela est fait via le constructeur canonique du record et les métadonnées RecordComponent pour les records, ou un constructeur sans argument suivi d'une affectation de champs pour les POJO ordinaires.

Si la liaison échoue (champ manquant, mauvais type, JSON mal formé), une BindingException porte un message rédigé pour être renvoyé au LLM tel quel comme retour correctif. Ceci est le mécanisme derrière la boucle de nouvelles tentatives automatique et bornée de PredictStrategy. Les types de retour scalaires, énumérés et de collection sont enveloppés sous une enveloppe de premier niveau {"value": ...}.

**Nom de paramètre réservé**

GenerationEngine rejette tout paramètre de méthode @Generative littéralement nommé reasoning, ce nom étant réservé au champ de raisonnement interne du modèle dans plusieurs stratégies. Cette vérification est effectuée à chaque appel (au premier appel de generate()), et non à la définition de la classe, car Java n'a pas de mécanisme équivalent à une métaclasse permettant d'effectuer ce contrôle plus tôt.

## 4. Stratégies de génération

Chaque appel @Generative est exécuté par une Strategy par une seule méthode :

Object execute(CallContext ctx)

Quatre implémentations sont fournies avec jOpenAgent, sélectionnables par méthode via @Generative(strategy = ...), ou globalement pour l'agent via AgentConfig.getDefaultStrategy() :

| Stratégie | Ce qu'elle fait | Quand l'utiliser |
|---|---|---|
| PREDICT (par défaut) | Un seul appel à sortie structurée, avec la nouvelle tentative automatique et bornée d'ObjectBinder en cas d'échec de validation. | Le cas courant : classification, extraction, synthèse, ou toute tâche avec une correspondance directe entrée-sortie. |
| CODE_ACT | Fournit au modèle une paire d'outils execute_java / return_result et boucle (jusqu'à maxCodeActIterations) via un CodeExecutor, avec un self vivant lié afin que le code généré puisse appeler les | Calculs à plusieurs étapes que le modèle exprime mieux sous forme de code que sous forme d'une seule réponse JSON, agrégation, itération, arithmétique sur plusieurs champs. |

| Stratégie | Ce qu'elle fait | Quand l'utiliser |
|---|---|---|
| | propres méthodes de l'agent. | |
| TOOL_CALLING | Une boucle classique de function-calling sur les méthodes ordinaires visibles de l'agent (via ToolRegistry), plus un outil synthétique return_result. | Tâches naturellement décomposables en appels d'outils discrets que le modèle choisit, y compris des outils issus du MCP. |
| REFLEXION | Exécute une stratégie de base (PREDICT par défaut), puis un appel de critique structuré séparé ; relance avec le retour intégré dans l'instruction jusqu'à satisfaction ou épuisement des tentatives (3 par défaut). | Tâches où une auto-vérification améliore sensiblement la justesse : arithmétique, énigmes logiques, tout ce qui mérite un second regard. |

## 4.1 PredictStrategy

La stratégie par défaut. Elle émet une requête à sortie structurée. En cas de BindingException, elle renvoie au modèle la réponse brute ainsi que l'erreur de validation comme texte correctif, en relançant jusqu'au maxRetries résolu ; elle lève une GenerationException une fois ce budget épuisé.

## 4.2 CodeActStrategy

Le modèle reçoit des outils (execute_java et return_result) et itère via CodeExecutor, soit le CodeSandbox en processus fondé sur JShell, soit l'OutOfProcessCodeSandbox hors processus optionnel, choisi via AgentConfig.getSandboxConfig().getMode().

Voir le chapitre 12 pour le modèle complet du bac à sable.

En cas de succès, l'échange est ajouté à Agent.history et peut déclencher HistorySummarizer, comme pour TOOL_CALLING (voir chapitre 8).

```
@Generative(strategy = StrategyKind.CODE_ACT,
            value = "Compute the average order total in cents.")
public double averageOrderDollars() {
    return generate();
}
```

## 4.3 ToolCallingStrategy

Expose les méthodes ordinaires visibles de l'agent en tant qu'outils natifs de function-calling à schéma JSON via ToolRegistry, ainsi qu'un outil synthétique return_result que le modèle appelle pour terminer.

Tout champ visible de type McpClient sur l'agent est automatiquement fusionné également, chaque outil MCP étant exposé sous le nom fieldName__toolName (voir chapitre 5).

### 4.4 ReflexionStrategy

Porte directement la boucle génération->critique->nouvelle tentative.

Deux constructeurs sont disponibles :

ReflexionStrategy() sans argument (stratégie de base PREDICT, 3 tentatives)

ReflexionStrategy(StrategyKind baseKind, int maxIterations) pour une stratégie de base différente (CODE_ACT ou TOOL_CALLING sont toutes deux valides) ou un budget de tentatives différent.

Une fois que la stratégie de base a produit un résultat, un appel de sortie structurée séparé demande au modèle de le critiquer, renvoyant un ReflectionOutcome{isSatisfactory, issues, suggestions, reasoning}. Si le résultat n'est pas satisfaisant, la stratégie de base est relancée avec le retour intégré dans l'instruction via CallContext.withInstruction(...). Un échec de l'appel de critique lui-même est traité comme « satisfaisant » plutôt que de faire échouer toute la génération.

> **StrategyKind.INHERIT**
>
> La strategy() d'une méthode @Generative vaut par défaut StrategyKind.INHERIT, ce qui signifie « utiliser ce que AgentConfig.getDefaultStrategy() résout pour cet agent » (elle-même par défaut PREDICT). Ne définir strategy() explicitement sur une méthode que lorsqu'elle doit différer de la stratégie globale de l'agent.

## 5. Outils, MCP et capacités structurées

### 5.1 Outils via self

ToolRegistry transforme les propres méthodes ordinaires et visibles d'un agent en outils de function-calling à schéma JSON (aucune étape séparée de définition d'outil n'est nécessaire).

ToolRegistry(agent) parcourt une seule fois les méthodes ordinaires visibles de l'agent (via MemberScanner.ordinaryVisibleMethods), en construisant un ToolSpec (nom, description issue de @Doc, schéma JSON de paramètres à partir des vrais noms et types de paramètres) pour chacune. C'est ce qui alimente à la fois StrategyKind.TOOL_CALLING et la capacité du bac à sable execute_java à appeler self.method(...).

```
@Doc("Look up current stock for a SKU.")
public int getStock(String sku) {
    return inventory.getOrDefault(sku, 0);
}
```

ToolRegistry.invoke(ToolCall) lie les arguments JSON du modèle aux vrais types de paramètres via ObjectBinder, puis invoque la méthode par réflexion, en levant une ToolInvocationException en cas d'échec afin que la stratégie appelante décide comment faire remonter l'erreur au modèle.

### 5.2 Model Context Protocol (MCP)

McpClient est un client JSON-RPC 2.0 en stdio direct.

Attacher un serveur MCP à un agent revient simplement à déclarer un champ visible de type McpClient. Son constructeur appelle initialize() et bloque jusqu'à ce que le serveur soit prêt. listTools() est mis en cache pour la durée de vie du client ; callTool(name, args) concatène les blocs de contenu texte renvoyés par le serveur en une seule String.

Tout champ McpClient visible est automatiquement fusionné dans l'ensemble d'outils de ToolCallingStrategy, chaque outil de ce serveur étant exposé sous le nom fieldName__toolName.

Par exemple un champ everything exposant un outil echo devient everything__echo.

> **Frontière de confiance**
>
> Attacher un serveur MCP lui accorde un sous-processus vivant s'exécutant avec les propres permissions du système d'exploitation de la JVM. N'attachez que des serveurs MCP en lesquels vous avez confiance. McpClient n'effectue aucun bac à sable supplémentaire sur le processus du serveur lui-même. Voir le chapitre 16 pour l'ensemble des consignes de sécurité.

Sous Windows, lancer un serveur MCP fondé sur Node via npx nécessite un passage par cmd.exe, car ProcessBuilder ne peut pas invoquer directement les fichiers d'amorçage .cmd/.bat (pas de gestion d'association de fichiers de type shell sous Windows). ProcessLaunch gère cela de façon transparente pour chaque appelant, il recherche d'abord un vrai .exe correspondant, et ne se rabat sur le passage par cmd.exe /c qu'en cas de nécessité réelle, afin qu'un processus tué ne laisse pas d'enfant orphelin derrière lui.

## 6. Mémoire

Chaque agent porte un champ self.memory (MemoryManager), présent par défaut et configurable via AgentConfig.getMemoryManager()/.embedder(...).

C'est une mémoire persistante et inter-sessions avec déduplication à l'écriture fondée sur la similarité, un graphe typé d'arêtes entre les souvenirs, et une notation de rappel de style ACT-R (Adaptive Control of Thought-Rational).

### 6.1 Opérations centrales

| Méthode | Comportement |
|---|---|
| `remember(content, type, importance, tags)` | Stocke le contenu en tant que nouveau souvenir, ou augmente l'importance d'un quasi- |

| Méthode | Comportement |
|---|---|
| | doublon existant. |
| `recall(query, k)` | Combine pertinence normalisée, récence (activation de base ACT-R) et importance selon les pondérations de MemoryRetrievalConfig, puis propage l'activation à travers les arêtes du graphe de mémoire, en renvoyant les k meilleurs souvenirs. |
| `associate(fromId, toId, relationType)` | Crée une arête typée, dirigée et pondérée entre deux souvenirs. Par exemple « caused_by », « related_to ». |
| `traverse(startId, maxDepth)` | Parcours en largeur du graphe de mémoire à partir d'un souvenir de départ. |
| `update(id, newContent)` | Remplace le contenu d'un souvenir sur place. |
| `forget(id)` | Suppression douce d'un souvenir (le marque comme oublié plutôt que de le supprimer physiquement). |

```
self.memory.remember("Customer prefers email over phone contact.",
Memory.Type.PREFERENCE, 0.8, Set.of("contact"));
List<Memory> relevant = self.memory.recall("how does this customer like to be
contacted?", 5);
```

## 6.2 Stores et embedders

MemoryManager est construit au-dessus d'un MemoryStore (persistance) et d'un Embedder (similarité sémantique).

Deux stores sont fournis par défaut :

InMemoryMemoryStore (éphémère, par défaut)

JsonFileMemoryStore (un unique fichier JSON, entièrement réécrit à chaque mutation).

Deux embedders sont fournis par défaut :

HashingEmbedder (déterministe, entièrement hors ligne, hachage de caractéristiques sur 256 dimensions)

OpenAiEmbedder (un véritable appel /v1/embeddings, avec des méthodes de fabrique forOllama/forLmStudio/forLocalServer reflétant celles d'OpenAiClient, pour des serveurs d'embeddings locaux et sans clé).

Comme les deux interfaces sont réduites (Embedder : embed(String)/dimensions() ; MemoryStore : save/findById/findAll/delete), substituer un véritable modèle d'embeddings ou une couche de

persistance différente est un changement d'une ligne à la construction d'AgentConfig, sans aucun autre changement dans le code d'un agent.

# 7. Compétences (Skills)

Les compétences (skills) sont des paquets de capacités/instructions attachés à un agent via self.skills (SkillRegistry).

Le nom et la description d'une compétence sont automatiquement restitués dans le prompt système pour la divulgation progressive, tandis que ses instructions complètes ne sont chargées qu'à la demande. Cela garde le prompt de base compact tout en laissant le modèle découvrir les capacités disponibles.

## 7.1 SkillRegistry et Skill

SkillRegistry(Agent owner) expose load(Skill), discoverAndLoadAll(Path rootDir) (charge automatiquement chaque sous-répertoire contenant un SKILL.md/skill.md), ainsi que unload/get/has/all/render.

org.jopenagent.skills.Skill est la classe de base abstraite dont hérite toute compétence : attach/detach sont final et appellent les méthodes protégées onAttach/onDetach qu'une sous-classe redéfinit ; getId()/getDescription() identifient la compétence ; un contextBlock() optionnel permet à une compétence de contribuer son propre contexte dynamique (voir chapitre 9).

## 7.2 TextSkill et SKILL.md

org.jopenagent.skills.TextSkill est l'implémentation concrète et générale des compétences : TextSkill.load(Path) analyse le frontmatter d'un fichier SKILL.md (name et description sont requis) et son corps (les instructions complètes, chargées uniquement lorsque la compétence est réellement consultée). Un SKILL.md minimal :

```
---
name: refund-policy
description: How to evaluate and respond to refund requests.
---

When a customer asks for a refund:
1. Confirm the order number and purchase date.
2. Refunds are only valid within 30 days of purchase.
3. Respond politely and state the decision clearly.
```

TextSkill expose aussi readFile(relativePath) cantonné au propre répertoire de la compétence, afin qu'une compétence puisse embarquer des données de référence aux côtés de son SKILL.md et runScript(scriptName, String... args), qui lance un véritable sous-processus via org.jopenagent.skills.ScriptRunner.

L'interpréteur est détecté automatiquement à partir de l'extension du fichier de script (.sh -> bash, .py -> python3, .js -> node, .ps1 -> powershell).

Les scripts sont tués de force (Process#destroyForcibly()) après un délai par défaut de 30 secondes s'ils ne se terminent pas.

# 8. Historique de conversation et synthèse

Chaque agent porte un champ final ConversationHistory : un journal d'entrées rôle+texte, thread-safe, lu et écrit par défaut par les stratégies CODE_ACT et TOOL_CALLING (les appels de base de PREDICT et REFLEXION ne le consultent pas).

## 8.1 ConversationHistory

| Méthode | Rôle |
|---|---|
| `append(HistoryEntry)` | Ajoute une entrée rôle+texte au journal. |
| `snapshot()` | Renvoie une copie immuable des entrées actuelles. |
| `replaceRange(from, to, replacement)` | Utilisé par HistorySummarizer pour condenser une plage d'anciennes entrées en une seule entrée de synthèse. |
| `asMessages()` | Restitue le journal sous forme de List<LlmMessage>, prête à être préfixée à une LlmRequest sortante. |
| `setLastReportedPromptTokens(int)` | Enregistre le dernier nombre de jetons de prompt rapporté par le fournisseur, alimentant le déclencheur par seuil de jetons de HistorySummarizer. |

## 8.2 Synthèse automatique

HistorySummarizer est statique uniquement, piloté par deux seuils combinés configurés via HistoryConfig : shouldSummarize(history, config) ne se déclenche que lorsque le nombre d'entrées dépasse preserveRecent, et, en plus, soit le nombre de jetons de prompt rapporté par le fournisseur dépasse maxTokens, soit le nombre d'entrées dépasse maxEntries.

Une fois déclenché, summarizeIfNeeded(history, llm, config) condense les entrées les plus anciennes en une seule HistoryEntry de type « summary » via un appel LLM dédié.

```
HistoryConfig config = HistoryConfig.builder()
    .maxEntries(40)       // défaut 40
    .preserveRecent(10)   // défaut 10, jamais synthétisées
    .maxTokens(8000)      // défaut 8000
    .targetChars(1000)    // défaut 1000, taille cible de l'entrée de synthèse
    .build();
```

**Seules CODE_ACT et TOOL_CALLING utilisent l'historique par défaut**

Comme les appels de base de PredictStrategy et ReflexionStrategy sont typiquement des appels uniques à sortie structurée plutôt que des boucles d'outils multi-tours, ils ne lisent ni n'écrivent dans l'historique par défaut. Si une mémoire conversationnelle multi-tours est nécessaire quelle que soit la stratégie, un agent peut toujours lire/ajouter à la mémoire directement depuis ses propres méthodes.

# 9. Blocs de contexte

ContextApi permet à un agent de contribuer des blocs de texte nommés supplémentaires à son propre prompt système, restitués automatiquement sous une section « ## Context » à chaque construction de prompt. Deux types de blocs sont pris en charge :

## 9.1 Blocs statiques

put(key, value) stocke une valeur fixe une seule fois, mise en cache pour la durée de vie de l'agent, approprié pour un contenu qui ne change pas d'un appel à l'autre, comme un cahier des charges de projet ou un plan de sprint défini à la construction :

```
context.put("sprint_goal", "Ship the new checkout flow by Friday.");
```

## 9.2 Blocs dynamiques

setDynamic(key, selfExpression) stocke à la place une chaîne d'expression auto-référente, réévaluée via org.jopenagent.core.PromptTemplate à chaque construction de prompt. C'est à utiliser pour un contenu qui change d'un appel à l'autre, comme un résumé de backlog en direct :

```
context.setDynamic("backlog", "{self.formatBacklog()}");
```

PromptTemplate.expand prend en charge exactement {self.field}, {self.method()}, les chaînes pointées, et la forme littérale {doc(self)}, un sous-ensemble délibérément restreint, et non une évaluation de code arbitraire.

Tout ce qui sort de cette grammaire lève une IllegalArgumentException avec un message clair, plutôt que de ne rien faire silencieusement.

Autres opérations de ContextApi : get(key), remove(key), keys(), et render() (assemble tous les blocs actuellement enregistrés dans la section de prompt « ## Context » restituée).

**Les compétences peuvent aussi contribuer des blocs de contexte**

Le contextBlock() optionnel d'une Skill (voir chapitre 7) est connecté au même mécanisme ContextApi lors du chargement de la compétence. Une compétence ne se limite pas à des instructions statiques, elle peut aussi contribuer un bloc de contexte dynamique et auto-

référent, exactement comme le propre code d'un agent.

## 10. Contenu multimodal

org.jopenagent.llm.media.Media/Image/Audio/Video/FileAttachment permettent à une méthode @Generative d'accepter un média comme un véritable paramètre typé (plutôt que d'exiger que l'appelant l'encode en base64) dans le prompt texte :

```
Image circle = Image.fromBytes(pngBytes, "image/png");

@Generative("Describe what shape and color is in this image.")
public String describeImage(Image image) {
    return generate(image);
}
```

org.jopenagent.core.PromptBuilder.collectMedia(ctx) extrait les paramètres liés de type Media du rendu textuel normal et les attache en tant que véritables blocs de contenu multimodal sur la LlmRequest sortante.

Chaque client LLM concret est ensuite responsable de son propre mappage spécifique au fournisseur vers le format de câble de ce fournisseur (voir chapitre 14).

| Type | Remarques |
|---|---|
| Image | Le chemin entièrement vérifié, utilisé directement par AnthropicClient et OpenAiClient comme contenu image natif. |
| Audio / Vidéo | Anthropic n'a pas de type de bloc de contenu audio/vidéo natif, donc AnthropicClient se rabat sur l'envoi d'un bloc en forme d'image pour ces types. |
| FileAttachment | Contenu de fichier générique, mappé vers le type de bloc document/fichier de chaque fournisseur. |

## 11. Traçabilité et observabilité

Chaque agent est tracé par défaut : chaque appel LLM, exécution de code, exécution d'outil et appel de génération est automatiquement enveloppé dans un Span, qu'un exportateur soit attaché ou non.

org.jopenagent.trace.Tracer maintient une pile ThreadLocal<Deque<Span>> par thread.

Tracer.start(kind, name) rattache un nouveau span à ce qui se trouve actuellement en haut de la pile du thread appelant ; Tracer.end(span) le dépile et, une fois l'arbre parvenu à sa racine, transmet l'arbre de spans complet à chaque TraceExporter enregistré.

## 11.1 Span et le contrat d'exportateur

Un Span porte un id/parentId, un kind, un nom, des horodatages de début/fin, un statut, un message d'erreur optionnel, une carte d'attributs, et ses enfants.

TraceExporter n'a qu'une seule méthode (void export(Span rootSpan)) appelée exactement une fois par span racine terminé, une fois que tout son sous-arbre a déjà été construit.

org.jopenagent.trace.Traced est un wrapper de confort (begin/finish/fail) tolérant aux valeurs nulles de bout en bout, devenant un no-op dès que le traçage est désactivé.

## 11.2 Formats d'export

| Exportateur | Format / destination |
|---|---|
| JsonTraceExporter | JSON brut de l'arbre de spans, éventuellement ajouté à un fichier JSONL. C'est le format le plus simple, utile pour une inspection locale rapide ou pour alimenter un outillage personnalisé. |
| AtifExporter | ATIF-v1.7 (agent-trace-interchange-format) : aplatit les enfants d'un span racine en étapes, un enfant llm_call devient une étape source « agent », un enfant tool_execution/code_execution devient une étape « observation », et un span generation imbriqué devient une entrée subagent_trajectories[]. |
| OtlpHttpExporter | JSON OTLP/HTTP standard (ExportTraceServiceRequest), avec le traceId dérivé de l'UUID du span racine. |
| LangfuseExporter | Étend OtlpHttpExporter avec l'authentification Basic et le chemin d'ingestion spécifique de Langfuse. |

```
Tracer.addExporter(new JsonTraceExporter(Paths.get("jooa-trace.jsonl")));
agent.writeHaiku("autumn");
// tous les spans produits par cet appel sont désormais capturés par l'exportateur
```

### 11.3 Visualiseurs de traces

Trois façons d'inspecter les traces capturées, toutes construites entièrement sur des dépendances de bibliothèque standard :

| Visualiseur | Base |
|---|---|
| TraceViewerServer (org.jopenagent.trace.viewer) | Une application web embarquée reposant uniquement sur com.sun.net.httpserver. Une unique page HTML/JavaScript autonome liste les traces, parcourt la mémoire, prend en charge les annotations, et inclut un onglet « Experiments » pour le harnais d'évaluation (voir chapitre 13). |
| TraceViewerSwingApp (org.jopenagent.trace.viewer.swing) | Une application de bureau native Swing, sans dépendance externe, utilisable soit en direct (enregistrée directement comme TraceExporter, si bien que les traces apparaissent à mesure que chaque appel se termine), soit en mode d'interrogation à distance contre un TraceViewerServer exécuté séparément. |
| TerminalTraceExplorer (org.jopenagent.trace.explorer) | Un afficheur d'arbre de spans fondé sur PrintStream, pour une inspection console rapide, sans serveur ni fenêtre du tout. |

## 12. Bac à sable code-as-action

CodeExecutor (une interface AutoCloseable : execute(String)/close()) est l'abstraction d'exécution derrière StrategyKind.CODE_ACT.

Deux implémentations sont disponibles (voir AgentConfig.getSandboxConfig().getMode()).

### 12.1 En processus : CodeSandbox (par défaut)

Le moteur par défaut est une session JShell persistante s'exécutant dans la même JVM, avec org.jopenagent.sandbox.SandboxBindings connectant une référence self vivante afin que le code généré puisse appeler directement les propres méthodes de l'agent.

Chaque appel est enveloppé par un thread de surveillance (watchdog) qui appelle jshell.stop() en cas de dépassement de délai, un mécanisme coopératif, par simple cession de contrôle, incapable de préempter une boucle réellement infinie telle que while (true) {}.

Avant l'exécution, org.jopenagent.sandbox.CodeValidator exécute une liste noire fondée sur la tokenisation (consciente des chaînes/commentaires) sur le code soumis, rejetant les paquets

dangereux (java.lang.reflect, java.net, java.nio.file, sun.*, etc.) et les noms de types simples (Runtime, ProcessBuilder, File, Socket, etc.).

> **CodeValidator est de la défense en profondeur uniquement, pas une frontière de sécurité**
>
> Une liste noire statique ne peut pas empêcher totalement un programme suffisamment déterminé d'atteindre le système de fichiers ou le réseau via la réflexion ou d'autres moyens indirects. Exécutez toujours les agents qui exécutent du code généré à l'intérieur d'une isolation au niveau du système d'exploitation (conteneur ou VM) ; ne comptez pas uniquement sur la validation en processus de jOpenAgent comme seule mesure de sécurité.

## 12.2 Hors processus : OutOfProcessCodeSandbox

OutOfProcessCodeSandbox est une amélioration optionnelle, activée via SandboxConfig.builder().mode(OUT_OF_PROCESS).timeoutMillis(...).heapMb(...).build().

Elle lance SandboxWorkerMain en tant que véritable JVM fille séparée (avec son propre plafond de tas -Xmx), et une classe stub Java générée (SelfStubGenerator émet du texte source ordinaire, pas du bytecode) transmet les appels self.method(...) via un RpcChannel bidirectionnel sur socket loopback à RemoteSelfDispatch, qui les répartit côté hôte via le même ToolRegistry qui alimente TOOL_CALLING.

Un dépassement de délai sous ce mode est un véritable arrêt forcé Process.destroyForcibly(), et non une simple demande coopérative stop().

```
SandboxConfig config = SandboxConfig.builder()
    .mode(SandboxMode.OUT_OF_PROCESS)
    .timeoutMillis(5000)
    .heapMb(256)
    .build();
```

Un worker mort est relancé de façon transparente à l'appel suivant.

## 12.3 Choisir un mode

| Mode | Garanties | Coût |
|---|---|---|
| IN_PROCESS (défaut) | Délai coopératif uniquement (jshell.stop()). | Aucun surcoût de lancement de processus |
| OUT_OF_PROCESS | Véritable délai à arrêt forcé. | Coût de démarrage d'une JVM fille par session |

> **OUT_OF_PROCESS n'améliore que la maîtrise du délai**
>
> Passer à SandboxMode.OUT_OF_PROCESS n'ajoute pas d'isolation système de fichiers ou réseau, cela garantit uniquement qu'une exécution incontrôlée est véritablement tuée plutôt que simplement invitée à s'arrêter.

# 13. Harnais d'évaluation

org.jopenagent.eval fournit un harnais d'évaluation compact et autonome : définir un ensemble de cas, les exécuter, noter les résultats.

Chaque appel à EvalRunner.run(...) enveloppe chaque cas dans son propre span de trace eval_case avant même l'exécution de la tâche, si bien que les spans produits par la tâche (le span generation d'un appel @Generative, ses enfants llm_call/tool_execution/code_execution imbriqués) s'y imbriquent naturellement.

L'ensemble complet des attributs eval.* (nom de l'expérience, palier, modèle, réussite/échec, score, durée) réside sur ce span, si bien qu'un TraceExporter normal capture tout ce dont le harnais d'évaluation a besoin, et l'onglet « Experiments » de TraceViewerServer reconstruit les résumés d'expériences en parcourant simplement les traces capturées à la recherche de spans de type eval_case.

## 13.1 Construire et exécuter un ensemble

```
EvalSuite<String> suite = EvalSuite.builder("ticket-triage")
    .addCase(EvalCase.builder("tc-1", () -> agent.classify("My order never arrived"))
        .displayName("Missing order")
        .expected("shipping")
        .tier(Tier.STABLE)
        .build())
    .build();

List<EvalResult<String>> results = EvalRunner.run(suite, new ContainsScorer(), "claude-
sonnet-4-5");
```

Une exception levée par un EvalTask devient un ScoreDetail en échec plutôt que de se propager hors de EvalRunner.run(...), un mauvais cas n'interrompt pas l'ensemble de l'exécution.

## 13.2 Scorers et paliers

| Scorer | Compare |
|---|---|
| ExactMatchScorer | Correspondance equals() exacte entre l'attendu et le réel. |
| ContainsScorer | Correspondance de sous-chaîne insensible à la casse, pour les résultats String. |
| NumericToleranceScorer(tolerance) | Proximité numérique dans une tolérance fixe, pour les résultats Number. |
| LlmJudgeScorer<T> | Demande à un LLM séparé de rendre {score, passed, reasoning} selon une grille d'évaluation (par défaut ou personnalisée), lui-même capturé comme un span llm_call imbriqué ; un |

| Scorer | Compare |
|---|---|
| | échec de l'appel de jugement devient un ScoreDetail en échec plutôt que de lever une exception. |

Tier (STABLE / FRONTIER / HORIZON) étiquette la difficulté attendue d'un cas, les cas STABLE doivent réussir, les cas FRONTIER sondent la limite des capacités actuelles, les cas HORIZON sont aspirationnels permettant à une vue de résultats de regrouper et d'interpréter les échecs selon leur importance réelle.

# 14. Clients LLM

LlmClient est l'abstraction indépendante du modèle avec laquelle communique chaque stratégie : un unique aller-retour bloquant, LlmResponse generate(LlmRequest request), plus modelName().

Les clients ne conservent aucun état conversationnel propre (l'historique complet des messages est renvoyé à chaque appel) et doivent lever une LlmException plutôt qu'une exception d'E/S brute en cas d'échec. Deux implémentations concrètes sont fournies avec jOpenAgent, toutes deux construites entièrement sur java.net.http.HttpClient.

## 14.1 AnthropicClient

Cible l'API Messages d'Anthropic. Deux constructeurs : AnthropicClient(apiKey, model) pour l'API hébergée (délai de connexion de 30 secondes), et une forme à quatre arguments acceptant une URL de base et un HttpClient personnalisés pour les proxies ou les doublures de test.

Comportement de mappage de câble notable : le prompt système devient le champ system de premier niveau de la requête ; tout tour supplémentaire de rôle SYSTEM au-delà du premier est replié dans un tour user préfixé par « [system note] » ; et parce qu'Anthropic exige que tous les blocs tool_result d'un tour assistant arrivent dans un seul et même tour user suivant, les messages TOOL consécutifs sont fusionnés en un seul tour user portant plusieurs blocs tool_result. Les médias sont mappés vers des blocs de contenu image ou document ; Audio et Video se replient sur un bloc en forme d'image, Anthropic n'ayant pas de type de bloc audio/vidéo natif.

AnthropicClient ne prend pas en charge le streaming.

## 14.2 OpenAiClient

Cible l'API Chat Completions d'OpenAI, comme cette même forme d'API est aussi celle qu'exposent Ollama et LM Studio en local, OpenAiClient fait également office de client pour ces deux-là, via des méthodes de fabrique dédiées :

| Fabrique / constructeur | Cible |
|---|---|
| `new OpenAiClient(apiKey, model)` | API OpenAI hébergée (délai de connexion 30s / |

| Fabrique / constructeur | Cible |
|---|---|
| | délai de requête 120s). |
| `OpenAiClient.forOllama(model, baseUrl)` | Un serveur Ollama local |
| `OpenAiClient.forLmStudio(model, baseUrl)` | Un serveur LM Studio local (par défaut http://localhost:1234) |
| `OpenAiClient.forLocalServer(model, baseUrl)` | Tout autre serveur local compatible OpenAI, avec un délai plus long de 600 secondes pour tenir compte d'une inférence locale plus lente. |

apiKey peut être null ou vide pour les serveurs locaux sans clé.

Contrairement à AnthropicClient, OpenAiClient n'effectue aucun repliement de messages : l'API Chat Completions autorise un message de résultat d'outil par appel, corrélé à son appel d'origine via tool_call_id, si bien que chaque LlmMessage se mappe vers exactement un message au format de câble. Les médias sont mappés vers des blocs input_audio, file, ou image_url selon le sous-type Media concret.

OpenAiClient ne prend pas en charge le streaming.

> **La configuration programmatique reste toujours disponible**
>
> Toute la configuration pilotée par variables d'environnement présentée au chapitre 2 (via ExampleSetup) n'est qu'une commodité au-dessus de constructeurs et méthodes de fabrique ordinaires. N'importe quelle application peut construire AnthropicClient ou OpenAiClient directement, avec des identifiants, des URL de base, des délais ou un HttpClient personnalisés explicites, sans dépendre du tout de variables d'environnement.

# 15. Parcours des exemples

org.jopenagent.examples contient 19 programmes exemples (E01 à E19).

## 15.1 E01_FirstGenerationMethod

Le plus petit agent possible : une seule méthode @Generative, greet(String name), illustrant de bout en bout le motif @SystemPrompt / @Generative / trampoline generate() avec la stratégie PREDICT par défaut. Un bon point de départ.

## 15.2 E02_StructuredOutput

Introduit un type de retour record, ReviewSummary(int rating, String sentiment, List<String> highlights). Montre ObjectBinder analysant, validant et liant automatiquement le JSON du LLM sur le record, avec une nouvelle tentative alimentée par l'erreur de validation si la première tentative ne se lie pas proprement.

## 15.3 E03_ToolsViaSelf

Un assistant d'inventaire utilisant StrategyKind.TOOL_CALLING : des méthodes ordinaires annotées @Doc (getStock, knownItems) sont exposées en tant qu'outils JSON natifs via ToolRegistry, et le modèle décide lesquels appeler avant de répondre.

## 15.4 E04_Strategies

La même tâche de tarification/classification résolue de trois façons différentes côte à côte : PREDICT (un seul appel à sortie structurée), CODE_ACT (le modèle écrit du Java appelant self.unitPriceCents(...) dans le bac à sable), et TOOL_CALLING (function-calling natif contre la même méthode), une comparaison directe et concrète des trois stratégies non-Reflexion.

## 15.5 E05_ProgressiveDisclosure

Illustre Agent.describe() et le modèle visible par défaut : un champ public apparaît automatiquement, un champ marqué @Hidden n'apparaît pas, et la forme de gabarit de prompt {doc(self)} permet au LLM de découvrir les propres capacités de l'agent à l'exécution.

## 15.6 E06_Tracing

Enregistre un JsonTraceExporter avant d'exécuter un agent rédigeant des haïkus, montrant que chaque appel de génération est automatiquement enveloppé dans un Span, qu'un exportateur soit attaché ou non, puis affiche l'arbre de spans parent-enfant capturé en JSON.

## 15.7 E07_ContextBlocks

Un assistant de planification de sprint utilisant self.context : un bloc statique (put, défini une fois le plan de sprint) et un bloc dynamique (setDynamic, réévalué à chaque construction de prompt via une expression self.formatBacklog()), tous deux restitués automatiquement dans le prompt système.

## 15.8 E08_CodeAsAction

Le mode CODE_ACT, le modèle écrit et exécute du Java appelant orderTotalsCents() à l'intérieur de CodeSandbox pour calculer une moyenne, en itérant via execute_java/return_result jusqu'à ce que ce soit fait.

## 15.9 E09_HistorySummarization

Un assistant de journalisation de fin de service pour le support client utilisant TOOL_CALLING, configuré avec un HistoryConfig artificiellement réduit afin que la synthèse se déclenche après seulement quelques appels. Une démonstration compacte de HistorySummarizer condensant les anciennes entrées en une seule entrée Summary/Topics/Outcomes/State.

## 15.10 E10_Skills

Attache une TextSkill construite à partir d'un SKILL.md en mémoire (frontmatter name/description plus un corps de style maison) à un agent de support, montrant le nom et la description de la compétence restitués pour la divulgation progressive, et ses instructions complètes chargées à la demande, la génération de réponse suivant implicitement les consignes de la compétence.

## 15.11 E11_McpTools

Attache le véritable serveur MCP de référence officiel @modelcontextprotocol/server-everything (lancé via npx) en tant que simple champ McpClient. Illustre son outil echo accessible via l'outil everything__echo auto-fusionné de TOOL_CALLING, et directement depuis du code généré par CODE_ACT (self.everything.callTool(...)). Requiert que npx (Node) soit disponible dans le PATH.

## 15.12 E12_Memory

Un agent de support utilisant self.memory (présent par défaut sur chaque agent) : des méthodes enveloppes minces (rememberFact, recallFacts) délèguent à MemoryManager.remember/recall, appelées par le modèle via TOOL_CALLING au fil de plusieurs tours de conversation. Illustre aussi le remplacement par un véritable OpenAiEmbedder via la variable d'environnement JOPENAGENT_EMBEDDING_MODEL, sans aucun autre changement de code.

## 15.13 E13_Multimodal

Génère un PNG en forme de cercle rouge et le transmet en tant que paramètre @Generative typé Image, démontrant que le média est attaché comme un véritable contenu multimodal.

## 15.14 E14_AtifTrajectory

Un assistant d'entrepôt utilisant TOOL_CALLING ; enregistre un AtifExporter et affiche le document de trajectoire ATIF-v1.7 résultant pour un appel de génération.

## 15.15 E15_Reflexion

Un résolveur de problèmes arithmétiques en texte utilisant StrategyKind.REFLEXION. Il illustre la boucle d'auto-critique et de nouvelle tentative (jusqu'à 3 tentatives) sur un problème délibérément choisi pour tester si le modèle obtient le bon calcul après avoir reçu sa propre critique en retour.

## 15.16 E16_TraceViewer

Démarre un TraceViewerServer embarqué, exécute quelques appels de génération de blagues, puis attend que l'utilisateur ouvre un navigateur et inspecte la liste des traces et l'arbre de spans avant d'appuyer sur Entrée pour arrêter le serveur.

### 15.17 E17_TraceViewerSwing

L'alternative de bureau native à E16 : enregistre TraceViewerSwingApp directement comme TraceExporter afin que les traces apparaissent en direct dans une interface Swing à mesure que chaque appel se termine.

### 15.18 E18_OutOfProcessSandbox

La même tâche de moyenne CODE_ACT que E08, mais configurée avec SandboxMode.OUT_OF_PROCESS. Sa méthode main() prouve ensuite le bénéfice concret directement (en contournant totalement le LLM) en soumettant une boucle serrée while (true) {} à un OutOfProcessCodeSandbox brut et en montrant qu'elle est tuée de force après le délai configuré plutôt que de rester bloquée indéfiniment, contrairement au moteur en processus par défaut.

### 15.19 E19_EvalHarness

Un classificateur de tri de tickets de support évalué via EvalSuite/EvalCase/EvalRunner avec un ContainsScorer sur quatre cas (dont un cas délibérément ambigu étiqueté Tier.FRONTIER). Démarre aussi un TraceViewerServer afin que les résultats puissent être parcourus, réussite/échec, par modèle et par palier, dans son onglet « Experiments ».

### 15.20 ExampleSetup

Configuration partagée que chaque exemple appelle. createLlmClient() choisit un fournisseur à partir des variables d'environnement, par ordre de priorité (Anthropic, puis OpenAI, puis LM Studio, puis Ollama), en levant une IllegalStateException aux instructions claires si aucune n'est définie. createEmbedder() choisit de même HashingEmbedder (par défaut) ou un véritable OpenAiEmbedder selon JOPENAGENT_EMBEDDING_MODEL et le fournisseur déjà configuré.

> **Taille du modèle**
>
> Il a été observé, lors des tests, que de petits modèles locaux (environ 4 milliards de paramètres) peinaient avec les conventions CODE_ACT et TOOL_CALLING multi-tours, inventant des données, ou n'appelant jamais return_result. Un modèle de 30 milliards de paramètres spécialisé dans le code a traité correctement chaque cas. Il s'agit d'une limitation de capacité du modèle, et non d'un défaut de jOpenAgent ; le framework lève une GenerationException plutôt que de rester bloqué lorsqu'un modèle ne converge jamais.

## Annexe A : inventaire complet des paquets et classes

| Paquet | Objet | Classes |
|---|---|---|
| `org.jopenagent.agentdoc` | Le mécanisme d'introspection | Introspector, MemberScanner, |

| Paquet | Objet | Classes |
|---|---|---|
| | équivalent à doc(self) (Introspector, MemberScanner, Visibility). | Visibility |
| org.jopenagent.core | La classe de base Agent, les annotations @SystemPrompt/@Generative/@Hidden/@Shown/@NoTrace/@Doc, le trampoline generate(), GenerationEngine, PromptTemplate/PromptBuilder, AgentConfig et GenerationException. | Agent, AgentConfig, CallContext, Doc, GenerationEngine, GenerationException, Generative, Hidden, NoTrace, PromptBuilder, PromptTemplate, Shown, SystemPrompt |
| org.jopenagent.core.context | self.context — les blocs de contexte statiques et dynamiques restitués dans le prompt système. | ContextApi |
| org.jopenagent.core.history | self.history — le journal de conversation persistant inter-appels et son HistorySummarizer. | ConversationHistory, HistoryConfig, HistoryEntry, HistorySummarizer |
| org.jopenagent.core.strategy | L'interface Strategy et ses quatre implémentations (PredictStrategy, CodeActStrategy, ToolCallingStrategy, ReflexionStrategy) ainsi que StrategyKind. | CodeActStrategy, PredictStrategy, ReflectionOutcome, ReflexionStrategy, Strategy, StrategyKind, ToolCallingStrategy |
| org.jopenagent.eval | Le harnais d'évaluation : EvalCase/EvalSuite/EvalRunner/EvalTask/EvalResult/Tier, ainsi que l'interface Scorer et ses implémentations ExactMatchScorer/ContainsScorer/NumericToleranceScorer/LlmJudgeScorer. | ContainsScorer, EvalCase, EvalResult, EvalRunner, EvalSuite, EvalTask, EvalTraceView, ExactMatchScorer, LlmJudgeScorer, NumericToleranceScorer, ScoreDetail, Scorer, Tier |
| org.jopenagent.examples | E01..E19 ainsi qu'ExampleSetup — un tutoriel progressif reflétant nooa/examples/quickstart. | E01_FirstGenerationMethod, E02_StructuredOutput, E03_ToolsViaSelf, E04_Strategies, E05_ProgressiveDisclosure, |

| Paquet | Objet | Classes |
|---|---|---|
| | | E06_Tracing, E07_ContextBlocks, E08_CodeAsAction, E09_HistorySummarization, E10_Skills, E11_McpTools, E12_Memory, E13_Multimodal, E14_AtifTrajectory, E15_Reflexion, E16_TraceViewer, E17_TraceViewerSwing, E18_OutOfProcessSandbox, E19_EvalHarness, ExampleSetup |
| `org.jopenagent.json` | ObjectBinder — liaison JSON <-> record/POJO/List/Map/Optional/enum fondée sur la réflexion — ainsi que SchemaGenerator et TypeUtils, construits sur org.jopenkit.json. | BindingException, ObjectBinder, SchemaGenerator, TypeUtils |
| `org.jopenagent.llm` | Les abstractions indépendantes du modèle LlmClient/LlmRequest/LlmResponse/LlmMessage/ToolSpec/ToolCall. | LlmClient, LlmException, LlmMessage, LlmRequest, LlmResponse, ToolCall, ToolSpec |
| `org.jopenagent.llm.anthropic` | AnthropicClient — le client concret de l'API Messages d'Anthropic, construit uniquement sur java.net.http. | AnthropicClient |
| `org.jopenagent.llm.media` | Media/Image/Audio/Video/FileAttachment — le contenu de message multimodal. | Audio, FileAttachment, Image, Media, Video |
| `org.jopenagent.llm.openai` | OpenAiClient — le client concret de l'API Chat Completions d'OpenAI (OpenAI réel, Ollama, LM Studio), également uniquement sur java.net.http. | OpenAiClient |

| Paquet | Objet | Classes |
|---|---|---|
| `org.jopenagent.mcp` | McpClient (JSON-RPC en stdio), McpServerConfig, McpToolDefinition. | McpClient, McpException, McpServerConfig, McpToolDefinition |
| `org.jopenagent.memory` | Memory, MemoryManager, MemoryStore (InMemory/JsonFile) et Embedder (HashingEmbedder par défaut ; OpenAiEmbedder pour des embeddings réels ou en serveur local). | Embedder, HashingEmbedder, InMemoryMemoryStore, JsonFileMemoryStore, Memory, MemoryEdge, MemoryException, MemoryManager, MemoryRetrievalConfig, MemoryStore, OpenAiEmbedder, VectorMath |
| `org.jopenagent.sandbox` | L'abstraction CodeExecutor, le CodeSandbox en processus (JShell), la liste noire de CodeValidator, SandboxResult et SandboxConfig/SandboxMode. | CodeExecutor, CodeSandbox, CodeValidator, JShellScriptRunner, SandboxBindings, SandboxConfig, SandboxException, SandboxMode, SandboxResult |
| `org.jopenagent.sandbox.remote` | OutOfProcessCodeSandbox — le travailleur JVM fille optionnel, RpcChannel (JSON-RPC bidirectionnel sur un socket loopback), et SelfStubGenerator/RemoteSelfDispatch. | JavaTypeSource, OutOfProcessCodeSandbox, RemoteSelfDispatch, RpcChannel, RpcException, SandboxWorkerMain, SelfStubGenerator |
| `org.jopenagent.skills` | Skill, TextSkill (incluant runScript), SkillRegistry, ScriptRunner — les paquets de capacités/instructions. | ScriptResult, ScriptRunner, Skill, SkillException, SkillRegistry, TextSkill |
| `org.jopenagent.tools` | ToolRegistry — expose les méthodes ordinaires d'un agent et tout outil MCP attaché en tant qu'outils à schéma JSON pour le function-calling natif. | ToolInvocationException, ToolRegistry |

| Paquet | Objet | Classes |
|---|---|---|
| `org.jopenagent.trace` | Span, Tracer (la pile de spans parent-enfant par thread), le contrat TraceExporter, et JsonTraceExporter. | JsonTraceExporter, Span, TraceException, TraceExporter, Traced, Tracer |
| `org.jopenagent.trace.atif` | AtifExporter — export au format ATIF-v1.7 (agent-trace-interchange-format). | AtifExporter |
| `org.jopenagent.trace.explorer` | TerminalTraceExplorer — un afficheur d'arbre de spans pour la console. | TerminalTraceExplorer |
| `org.jopenagent.trace.otlp` | OtlpHttpExporter (JSON OTLP/HTTP standard) et LangfuseExporter. | LangfuseExporter, OtlpHttpExporter |
| `org.jopenagent.trace.viewer` | TraceViewerServer — l'application web embarquée de visualisation de traces (com.sun.net.httpserver uniquement), incluant la navigation en mémoire, les annotations et les routes « Experiments » du harnais d'évaluation. | TraceViewerPage, TraceViewerServer |
| `org.jopenagent.trace.viewer.swing` | TraceViewerSwingApp — le visualiseur de traces natif de bureau (Swing, sans dépendance). | JsonTraceNode, TraceViewerSwingApp |
| `org.jopenagent.util` | ProcessLaunch — l'utilitaire partagé de lancement de processus sûr sous Windows (.cmd), utilisé par McpClient et les scripts de compétences. | ProcessLaunch |
| `org.jopenkit.json` | Le parseur JSON préexistant, développé maison et non modifié (JSONObject/JSONArray/JSONTokener), sur lequel s'appuie la couche d'analyse/arbre de org.jopenagent.json. | JSON, JSONAble, JSONArray, JSONException, JSONObject, JSONStringer, JSONTokener, TypeAdapter, TypeAdapters, WithClassTypeAdapter |

# Annexe B : référence des classes centrales

## B.1 Modèle central de l'agent

### *Agent (org.jopenagent.core)*

Classe de base pour les agents jOpenAgent.

### *AgentConfig (org.jopenagent.core)*

Réglages propres à chaque agent.

### *Generative (org.jopenagent.core)*

Marque une méthode dont le corps délègue au LLM via Agent.generate(Object...).

### *SystemPrompt (org.jopenagent.core)*

Prompt système au niveau classe pour un Agent.

### *Doc (org.jopenagent.core)*

Texte de description pour une méthode ou un champ ordinaire (non-@Generative), se substituant au Javadoc (non réflectable à l'exécution).

### *Hidden (org.jopenagent.core)*

Masque un champ ou une méthode ordinaire du rendu de doc(self), du périmètre d'exécution du bac à sable code-as-action, et de la génération de schéma d'outils-via-self.

### *Shown (org.jopenagent.core)*

Réexpose un champ ou une méthode non publique qui serait autrement masqué par défaut (reflète l'option @spec(hidden=False) de NOOA pour les méthodes de style _privé).

### *GenerationEngine (org.jopenagent.core)*

Orchestre un appel @Generative : résout la stratégie, l'enveloppe dans un span de trace, l'exécute.

## B.2 Stratégies de génération

### *Strategy (org.jopenagent.core.strategy)*

Exécute un appel de méthode @Generative de bout en bout et renvoie le résultat lié.

### *PredictStrategy (org.jopenagent.core.strategy)*

Sortie structurée en un seul appel avec nouvelle tentative : demander au LLM un seul objet JSON correspondant au schéma du type de retour ; en cas d'échec d'analyse JSON ou de validation, renvoyer la réponse brute et l'erreur au LLM et relancer, jusqu'à CallContext.getMaxRetries() tentatives.

### *CodeActStrategy (org.jopenagent.core.strategy)*

Code-as-action : fournit au LLM deux outils, execute_java et return_result, et boucle jusqu'à ce qu'il appelle return_result.

### *ToolCallingStrategy (org.jopenagent.core.strategy)*

Boucle classique de function-calling : le LLM appelle les méthodes ordinaires de l'agent (exposées comme outils à schéma JSON par ToolRegistry) jusqu'à appeler l'outil synthétique return_result avec sa réponse finale, validée contre le type de retour exactement comme le fait PredictStrategy avec le même comportement de nouvelle tentative alimentée par l'erreur.

### *ReflexionStrategy (org.jopenagent.core.strategy)*

Boucle génération -> réflexion -> amélioration : exécute une stratégie de base, demande via un appel LLM structuré séparé si le résultat est satisfaisant, et sinon relance la stratégie de base avec les problèmes/suggestions de la critique intégrés dans l'instruction — jusqu'à maxIterations tentatives, en renvoyant le meilleur résultat obtenu s'il ne devient jamais satisfaisant.

### *StrategyKind (org.jopenagent.core.strategy)*

Sélectionne quelle Strategy traite une méthode @Generative.

## B.3 Sortie structurée

### *ObjectBinder (org.jopenagent.json)*

Liaison fondée sur la réflexion entre des arbres org.jopenkit.json (JSONObject/JSONArray/scalaires) et des types Java arbitraires : records, POJO ordinaires (constructeur sans argument + champs), List/Set/tableau, Map, Optional, énumérations et primitives/wrappers/String.

## B.4 Clients LLM et outils

### *LlmClient (org.jopenagent.llm)*

Abstraction de client LLM indépendante du modèle.

### *AnthropicClient (org.jopenagent.llm.anthropic)*

Client de l'API Messages d'Anthropic (POST /v1/messages).

### *OpenAiClient (org.jopenagent.llm.openai)*

Client compatible Chat Completions (POST /v1/chat/completions)

### *ToolRegistry (org.jopenagent.tools)*

Expose les méthodes ordinaires (non-@Generative, visibles) d'un agent comme des outils à schéma JSON directement invocables par l'API native de function-calling d'un fournisseur.

## B.5 Historique et contexte

### *ConversationHistory (org.jopenagent.core.history)*

Le journal de conversation persistant et inter-appels d'un agent.

### *HistorySummarizer (org.jopenagent.core.history)*

Condense la plus ancienne série d'une ConversationHistory en une seule entrée de synthèse une fois qu'un déclencheur configuré se déclenche, via un appel LLM dédié.

### *ContextApi (org.jopenagent.core.context)*

Contexte du Llm.

## B.6 Mémoire

### *MemoryManager (org.jopenagent.memory)*

API remember/recall/update/forget/associate/traverse : mémoire persistante et inter-sessions avec déduplication à l'écriture fondée sur la similarité, un graphe typé d'arêtes entre souvenirs et la véritable notation de récupération de style ACT-R, activation de base tirée de l'historique d'accès de chaque souvenir, mélangée à la pertinence et à l'importance, puis propagée à travers les arêtes du graphe.

### *InMemoryMemoryStore (org.jopenagent.memory)*

Store de mémoire éphémère, valable le temps du processus (aucune persistance) et celui qu'utilise le propre quickstart de NOOA via path=":memory:".

### *JsonFileMemoryStore (org.jopenagent.memory)*

Persistance inter-sessions pour les enregistrements Memory, adossée à un unique fichier JSON.

### *Embedder (org.jopenagent.memory)*

Transforme du texte en un vecteur d'embedding de taille fixe, normalisé L2, pour le rappel fondé sur la similarité.

### *HashingEmbedder (org.jopenagent.memory)*

Un embedder déterministe, hors ligne, sans réseau — hachage de caractéristiques (sac de jetons hachés) dans un vecteur de taille fixe, puis normalisé L2.

### *OpenAiEmbedder (org.jopenagent.memory)*

Véritables embeddings via /v1/embeddings, construits entièrement sur java.net.http.HttpClient.

## B.7 Compétences (Skills)

### *SkillRegistry (org.jopenagent.skills)*

Cycle de vie des compétences par agent : chargement/déchargement/recherche, plus découverte d'un répertoire de paquets SKILL.md.

### *Skill (org.jopenagent.skills)*

Un paquet de capacités/instructions attaché à un agent.

### *TextSkill (org.jopenagent.skills)*

Charge un répertoire SKILL.md/skill.md.

## B.8 Traçabilité et observabilité

### *Tracer (org.jopenagent.trace)*

Suivi des spans parent-enfant.

### *Span (org.jopenagent.trace)*

Une unité de travail tracée : un appel de génération LLM, une exécution de code, une exécution d'outil, ou une invocation de méthode ordinaire tracée.

### *TraceExporter (org.jopenagent.trace)*

Reçoit de Tracer les spans racines terminés (un arbre d'appels complet).

### *JsonTraceExporter (org.jopenagent.trace)*

Exporte un arbre de spans terminé en JSON, une ligne (JSONL) par span racine si un fichier est fourni, sinon conservé en mémoire pour inspection.

### *AtifExporter (org.jopenagent.trace.atif)*

Exporte un arbre de spans terminé sous forme de document JSON de trajectoire ATIF-v1.7

### *OtlpHttpExporter (org.jopenagent.trace.otlp)*

Exporte un arbre de spans sous forme de ExportTraceServiceRequest JSON OTLP/HTTP.

### *LangfuseExporter (org.jopenagent.trace.otlp)*

OtlpHttpExporter préconfiguré pour le point d'ingestion OTLP de Langfuse.

## B.9 Bac à sable

### *CodeSandbox (org.jopenagent.sandbox)*

Exécution code-as-action : une session JShell REPL persistante avec une variable self vivante liée à l'instance de l'agent, afin que le code généré puisse appeler self.uneMethode(...) directement.

### *CodeExecutor (org.jopenagent.sandbox)*

Forme commune du backend d'exécution de code de CODE_ACT, afin que org.jopenagent.core.strategy.CodeActStrategy n'ait pas besoin de savoir s'il dialogue avec le CodeSandbox en processus par défaut ou l'org.jopenagent.sandbox.remote.OutOfProcessCodeSandbox optionnel.

### *OutOfProcessCodeSandbox (org.jopenagent.sandbox.remote)*

Backend CODE_ACT hors processus : lance SandboxWorkerMain en tant que véritable JVM fille séparée (son propre tas, via -Xmx), et y exécute le code généré au lieu du processus hôte.

## B.10 Harnais d'évaluation

### *EvalRunner (org.jopenagent.eval)*

Exécute une EvalSuite et note chaque cas.

### *EvalSuite (org.jopenagent.eval)*

Un groupe nommé d'EvalCase, le concept d'« expérience » de jOpenAgent.

### *Scorer (org.jopenagent.eval)*

Compare la sortie réelle d'un cas d'évaluation à sa valeur attendue et produit un ScoreDetail.

### *ExactMatchScorer (org.jopenagent.eval)*

Réussit lorsque actual.equals(expected) (ou que les deux sont null) ; le scorer le plus simple possible.

### *ContainsScorer (org.jopenagent.eval)*

Réussit lorsque actual contient (indépendamment de la casse) expected comme sous-chaîne, utile pour une sortie générative en texte libre.

### *NumericToleranceScorer (org.jopenagent.eval)*

Réussit lorsque |actual - expected| <= tolerance — pour les sorties numériques où une correspondance exacte serait trop stricte.

### *LlmJudgeScorer (org.jopenagent.eval)*

Note un cas en demandant à un LLM de juger si actual satisfait expected selon une grille d'évaluation, plutôt qu'une comparaison exacte/de sous-chaîne/numérique, pour une sortie en texte libre où il n'existe pas de chaîne correcte unique.