jOpenAgent

Java Object-Oriented Agents

Table of Contents

1. Introduction & Overview

jOpenAgent is a framework to create AI agents.

It brings a central idea to the JVM: an agent is just a plain Java class. Fields are state, ordinary methods are deterministic capabilities, and methods annotated with @Generative are delegated to a language model (LLM) at runtime. There is no framework-specific base object model to learn beyond ordinary Java inheritance, and no separate configuration language. An agent is just a class, and its behavior is just code.

jOpenAgent draws partial inspiration from NVIDIA Object-Oriented Agents (NOOA).

1.1 The canonical example

The smallest complete jOpenAgent agent is a class extending org.jopenagent.core.Agent, carrying a class-level @SystemPrompt annotation and one @Generative method whose body simply calls 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); // the interception point
    }
}
```

It is used like any other Java object:

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

Changing analyzeFeedback's @Generative text changes its behavior. The instruction string is the prompt.

1.2 What this manual covers

Chapters 2-14 cover, in turn: project setup, core mechanisms, the four generation strategies, tools and MCP integration, memory, skills, conversation history, context blocks, multimodal content, tracing and observability, the code-as-action sandbox, the evaluation harness, and the two built-in LLM clients. Chapter 15 walks through the bundled tutorial examples. The appendices provide a complete package/class inventory (Appendix A) and a quick-reference index of the framework's most central classes (Appendix B).

1.3 Feature parity with NOOA, at a glance

jOpenAgent implements the full core mechanism, all four generation strategies (including one, TOOL_CALLING, that has no direct NOOA equivalent), structured output, progressive disclosure, context blocks, history summarization, skills, MCP tool integration, cross-session memory, multimodal content, ATIF trace export, a web/Swing/terminal trace viewer trio, two LLM clients, a code-execution sandbox with both in-process and out-of-process backends, and an evaluation harness.

2. Getting Started

2.1 Project structure

jOpenAgent is a dependency-free Java 21 (or higher) library; no external JAR is required for the framework itself:

Feature	Implementation
HTTP	java.net.http.HttpClient
JSON	The bundled org.jopenkit.json parser, extended by org.jopenagent.json
Sandbox	jdk.jshell
Embedded web server	com.sun.net.httpserver (jdk.httpserver)
MCP server processes	java.lang.ProcessBuilder
Tests only	JUnit 5

2.2 Configuring an LLM client

Each example program in org.jopenagent.examples picks up its LLM client from environment variables, via org.jopenagent.examples.ExampleSetup, checked in this priority order:

Variable(s)	Effect
`ANTHROPIC_API_KEY`	Together with JOPENAGENT_MODEL (e.g. claude-sonnet-5), builds an AnthropicClient against the hosted Anthropic API.
`OPENAI_API_KEY`	Together with JOPENAGENT_MODEL (e.g. gpt-4o-mini), builds an OpenAiClient against the hosted OpenAI API.
`JOPENAGENT_LMSTUDIO_MODEL`	Builds OpenAiClient.forLmStudio(model,

Variable(s)	Effect
	baseUrl), no API key needed. Optional JOPENAGENT_LMSTUDIO_BASE_URL, default http://localhost:1234.
`JOPENAGENT_OLLAMA_MODEL`	Builds OpenAiClient.forOllama(model, baseUrl), no API key needed. Optional JOPENAGENT_OLLAMA_BASE_URL, default http://localhost:11434.
`JOPENAGENT_EMBEDDING_MODEL`	Switches ExampleSetup.createEmbedder() from the default HashingEmbedder to a real OpenAiEmbedder, using the same provider credentials as above.

2.3 Running an example

Every class under org.jopenagent.examples has its own main(), runnable with no code edits, only the environment variables above. For instance, with ANTHROPIC_API_KEY and JOPENAGENT_MODEL set:

```
$ export ANTHROPIC_API_KEY=sk-ant-...
$ export JOPENAGENT_MODEL=claude-sonnet-5
# then run org.jopenagent.examples.E01_FirstGenerationMethod as a Java Application
```

Chapter 15 walks through the bundled examples (E01-E19) in the order they're meant to be read, each demonstrating one specific feature area covered later in this manual.

> **MCP examples need Node.js**
>
> E11_McpTools additionally requires Node's npx on PATH, since it launches the official @modelcontextprotocol/server-everything reference server as a child process. On Windows, org.jopenagent.mcp.McpClient (via org.jopenagent.util.ProcessLaunch) automatically and transparently routes such npx-based commands through cmd.exe /c, since ProcessBuilder cannot launch .cmd/.bat scripts directly.

3. Core Concepts

3.1 Agents are Java objects

A jOpenAgent agent is an ordinary class extending org.jopenagent.core.Agent.

Typed instance fields are the agent's state, and are visible by reflection via the describe() method and the code-as-action sandbox and tools-via-self mechanisms, unless marked @Hidden.

Ordinary methods with real bodies are deterministic capabilities, callable directly by other Java code, by generated sandbox code, or as native function-calling tools.

Methods annotated @Generative, ending in a call to generate(...), are delegated to the LLM.

3.2 The generate() method

The system uses StackWalker to identify its caller at runtime. It reads the @Generative annotation of that method as well as its declared return type by reflection, then delegates to the configured Strategy. Parameters intended for the LLM must be passed explicitly:

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

3.3 Class-level and method-level instructions

@SystemPrompt (class-level) carries an agent's overall system prompt. Its value is expanded via a PromptTemplate. If a subclass doesn't declare its own @SystemPrompt, the nearest annotated superclass's is used.

@Generative (method-level) carries the instruction (prompt) for that specific capability:

- value() is the instruction text
- maxRetries() overrides the agent-wide retry budget for this method
- strategy() selects a StrategyKind for this call, defaulting to StrategyKind.INHERIT (which falls back to AgentConfig.getDefaultStrategy())

org.jopenagent.core.Doc is the equivalent mechanism for ordinary (non-@Generative) methods and fields.

3.4 AgentConfig

Per-agent settings are injected via AgentConfig, built with AgentConfig.builder()...build():

```
AgentConfig config = AgentConfig.builder()
    .llmClient(llmClient)          // required
    .maxRetries(3)                 // default 3
    .defaultStrategy(StrategyKind.PREDICT)  // default PREDICT
    .maxCodeActIterations(8)        // default 8
    .historyConfig(HistoryConfig.builder().build())
    .embedder(new HashingEmbedder())   // or .memoryManager(...) directly
    .sandboxConfig(SandboxConfig.builder().build())
    .build();
```

build() throws IllegalStateException if llmClient was never set, or if maxRetries is less than 1. If both .memoryManager(...) and .embedder(...) are supplied, memoryManager takes precedence; if

neither is supplied, the default is a fresh MemoryManager over an InMemoryMemoryStore and a HashingEmbedder.

3.5 Progressive disclosure and visibility

Agent.describe() renders a summary of an agent's visible fields and methods by delegating to org.jopenagent.agentdoc.Introspector, which walks the class hierarchy up to (but not including) Agent itself, de-duplicating overridden members.

The same {doc(self)} literal form can be embedded directly inside a @SystemPrompt or @Generative instruction string, letting the LLM discover the agent's own capabilities at prompt-build time rather than the developer having to hand-list them.

Visibility follows a visible-by-default model driven by ordinary Java modifiers. Public fields/methods are visible by default; @Hidden excludes an otherwise-visible member from doc(self), the sandbox, and tool schemas; @Shown re-exposes an otherwise-hidden (non-public) member. @NoTrace additionally excludes a member from trace-span attribute capture.

3.6 Structured output

org.jopenagent.json.ObjectBinder is a fully automatic, reflection-based JSON<->Java layer supporting records, POJOs, List/Set/Map, Optional, enums, and primitives/wrappers, with no per-class registration step required anywhere.

When a @Generative method's declared return type is, say, a 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 (see Chapter 4) asks the model for JSON matching that shape, then ObjectBinder parses, strips markdown code fences if present, and binds it by reflection. This is done via the record's canonical constructor and RecordComponent metadata for records, or a no-arg constructor plus field assignment for ordinary POJOs.

If binding fails (missing field, wrong type, malformed JSON), a BindingException carries a message written to be re-sent to the LLM verbatim as retry feedback. This is the mechanism behind PredictStrategy's automatic, bounded retry loop. Scalar, enum, and collection return types are wrapped under a top-level {"value": ...} envelope.

> **Reserved parameter name**
>
> GenerationEngine rejects any @Generative method parameter literally named reasoning, since that name is reserved for the model's own internal reasoning field in several strategies. This is validated per-call (at the first generate() invocation), not at class-definition time, since Java has

no metaclass-equivalent hook to run that check earlier.

4. Generation Strategies

Every @Generative call is executed by a Strategy through a single method:

```
Object execute(CallContext ctx)
```

Four implementations ship with jOpenAgent, selectable per-method via @Generative(strategy = ...), or agent-wide via AgentConfig.getDefaultStrategy():

Strategy	What it does	When to reach for it
PREDICT (default)	A single structured-output call, with ObjectBinder's bounded automatic retry on validation failure.	The common case: classification, extraction, summarization, or anything with a direct input-to-output shape.
CODE_ACT	Gives the model an execute_java / return_result tool pair and loops (up to maxCodeActIterations) through a CodeExecutor, with a live self bound so generated code can call the agent's own methods.	Multi-step computation the model is better off expressing as code than as a single JSON answer, aggregation, iteration, arithmetic over several fields.
TOOL_CALLING	A classic ReAct-style function-calling loop over the agent's ordinary visible methods (via ToolRegistry) plus a synthetic return_result tool.	Tasks naturally decomposed into discrete tool calls the model chooses between, including MCP-sourced tools.
REFLEXION	Runs a base strategy (default PREDICT), then a separate structured critique call; retries with feedback folded into the instruction until satisfactory or attempts exhausted (default 3).	Tasks where a self-check meaningfully improves correctness, e.g. arithmetic, logic puzzles, anything worth a second look.

4.1 PredictStrategy

The default strategy. Issues one structured-output request. On BindingException, it feeds the raw model response plus the validation error back to the model as corrective text, retrying up to the resolved maxRetries; throws GenerationException once that budget is exhausted.

4.2 CodeActStrategy

The model receives tools (execute_java and return_result) and iterates through CodeExecutor, either the in-process JShell-based CodeSandbox, or the opt-in out-of-process OutOfProcessCodeSandbox, selected via AgentConfig.getSandboxConfig().getMode().

See Chapter 12 for the full sandbox model.

On success, the exchange is appended to Agent.history and may trigger HistorySummarizer, the same as TOOL_CALLING (see Chapter 8).

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

4.3 ToolCallingStrategy

Exposes the agent's ordinary visible methods as native JSON-schema function-calling tools via ToolRegistry, plus a synthetic return_result tool the model calls to finish.

Any visible McpClient-typed field on the agent is automatically merged in as well, with each MCP tool exposed under the name fieldName__toolName (see Chapter 5).

4.4 ReflexionStrategy

Runs the generate -> critique -> retry loop directly. Two constructors are available: ReflexionStrategy() with no arguments (base strategy PREDICT, 3 attempts), and ReflexionStrategy(StrategyKind baseKind, int maxIterations) for a different base strategy (CODE_ACT or TOOL_CALLING are both valid) or a different attempt budget.

After the base strategy produces a result, a separate structured-output call asks the model to critique it, returning a ReflectionOutcome{isSatisfactory, issues, suggestions, reasoning}. If not satisfactory, the base strategy is re-run with the feedback folded into the instruction via CallContext.withInstruction(...). A failed critique call itself is treated as "satisfactory" rather than crashing the whole generation.

StrategyKind.INHERIT

A @Generative method's strategy() defaults to StrategyKind.INHERIT, meaning "use whatever AgentConfig.getDefaultStrategy() resolves to for this agent" (itself defaulting to PREDICT). Only

set strategy() explicitly on a method when it needs to differ from the agent's overall default.

5. Tools, MCP & Structured Capabilities

5.1 Tools via self

ToolRegistry turns an agent's own ordinary, visible methods into JSON-schema function-calling tools (no separate tool-definition step is needed).

ToolRegistry(agent) scans the agent's ordinary visible methods once (via MemberScanner.ordinaryVisibleMethods), building a ToolSpec (name, description from @Doc, JSON parameter schema from real parameter names and types) for each. This is what powers both StrategyKind.TOOL_CALLING and the execute_java sandbox's ability to call self.method(...).

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

ToolRegistry.invoke(ToolCall) binds the model's JSON arguments to the real parameter types via ObjectBinder, then invokes the method reflectively, throwing ToolInvocationException on failure so the calling strategy can decide how to surface the error back to the model.

5.2 Model Context Protocol (MCP)

McpClient is a direct stdio JSON-RPC 2.0 client.

Attaching an MCP server to an agent is just declaring a visible McpClient-typed field. Its constructor calls initialize() and blocks until ready. listTools() is cached for the client's lifetime; callTool(name, args) concatenates the server's returned text content blocks into a single String.

Any visible McpClient field is automatically merged into ToolCallingStrategy's tool set, with each of that server's tools exposed under the name fieldName__toolName.

For example, an everything field exposing an echo tool becomes everything__echo.

Trust boundary

Attaching an MCP server grants it a live subprocess running with the JVM's own OS permissions. Only attach MCP servers you trust. McpClient performs no additional sandboxing of the server process itself.

On Windows, launching a Node-based MCP server via npx requires routing through cmd.exe, since ProcessBuilder cannot invoke .cmd/.bat shims directly (no shell-style file-association handling on Windows). ProcessLaunch handles this transparently for every caller, probing for a matching

real .exe first, and only falling back to cmd.exe /c routing when genuinely needed, so a killed process doesn't leave an orphaned child behind.

6. Memory

Every agent carries a self.memory field (MemoryManager), present by default and configurable via AgentConfig.getMemoryManager()/.embedder(...).

It is a persistent, cross-session memory with similarity-based dedup-on-write, a typed graph of edges between memories, and ACT-R-style (Adaptive Control of Thought-Rational) retrieval scoring.

6.1 Core operations

Method	Behavior
`remember(content, type, importance, tags)`	Stores the content as a new memory, or bumps the importance of an existing near-duplicate.
`recall(query, k)`	Combines normalized relevance, recency (ACT-R base-level activation), and importance per MemoryRetrievalConfig weights, then spreads activation across the memory graph's edges, returning the top k memories.
`associate(fromId, toId, relationType)`	Creates a typed, directed, weighted edge between two memories. For example "caused_by", "related_to".
`traverse(startId, maxDepth)`	Breadth-first walk of the memory graph from a starting memory.
`update(id, newContent)`	Replaces a memory's content in place.
`forget(id)`	Soft-deletes a memory (marks it forgotten rather than physically removing it).

```
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 and embedders

MemoryManager is constructed over a MemoryStore (persistence) and an Embedder (semantic similarity).

Two stores ship by default:

InMemoryMemoryStore (ephemeral, the default)

JsonFileMemoryStore (a single JSON file, fully rewritten on each mutation).

Two embedders ship by default:

HashingEmbedder (deterministic, fully offline, 256-dimension feature hashing)

OpenAiEmbedder (a real /v1/embeddings call, with forOllama/forLmStudio/forLocalServer factory methods mirroring OpenAiClient's own, for local, keyless embedding servers).

Because both interfaces are small (Embedder: embed(String)/dimensions(); MemoryStore: save/findById/findAll/delete), swapping in a real embedding model or a different persistence layer is a one-line change at AgentConfig construction time, with zero changes elsewhere in an agent's code.

7. Skills

Skills are packaged capability/prompt bundles attached to an agent via self.skills (SkillRegistry).

A skill's name and description are rendered into the system prompt automatically for progressive disclosure, while its full instructions are only pulled in on demand. This keeps the base prompt compact while still letting the model discover what capabilities are available.

7.1 SkillRegistry and Skill

SkillRegistry(Agent owner) exposes load(Skill), discoverAndLoadAll(Path rootDir) (auto-loads every subdirectory containing a SKILL.md/skill.md), plus unload/get/has/all/render.

org.jopenagent.skills.Skill is the abstract base every skill extends: attach/detach are final and call the protected onAttach/onDetach hooks a subclass overrides; getId()/getDescription() identify the skill; an optional contextBlock() pair lets a skill contribute its own dynamic context (see Chapter 9).

7.2 TextSkill and SKILL.md

org.jopenagent.skills.TextSkill is the concrete, general-purpose skill implementation: TextSkill.load(Path) parses a SKILL.md file's frontmatter (name and description are required) and body (the full instructions, pulled in only when the skill is actually consulted). A minimal SKILL.md:

```
---
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 also exposes readFile(relativePath), sandboxed to the skill's own directory, so a skill can bundle reference data alongside its SKILL.md, and runScript(scriptName, String... args), which launches a real subprocess via ScriptRunner.

The interpreter is auto-detected from the script's file extension (.sh -> bash, .py -> python3, .js -> node, .ps1 -> powershell).

Scripts are force-killed (Process#destroyForcibly()) after a default 30-second timeout if they don't finish.

8. Conversation History & Summarization

Every agent carries a final ConversationHistory field: a thread-safe, plain role-plus-text entry log, read and written by default by the CODE_ACT and TOOL_CALLING strategies (the base calls of PREDICT and REFLEXION do not consult it).

8.1 ConversationHistory

Method	Purpose
`append(HistoryEntry)`	Adds one role+text entry to the log.
`snapshot()`	Returns an immutable copy of the current entries.
`replaceRange(from, to, replacement)`	Used by HistorySummarizer to collapse a range of old entries into one summary entry.
`asMessages()`	Renders the log as List<LlmMessage>, ready to prepend to an outgoing LlmRequest.
`setLastReportedPromptTokens(int)`	Records the provider's last reported prompt-token count, feeding HistorySummarizer's token-threshold trigger.

8.2 Automatic summarization

HistorySummarizer is static-only, driven by two combined thresholds configured via HistoryConfig: shouldSummarize(history, config) fires only once the entry count exceeds preserveRecent, and additionally either the provider-reported prompt-token count exceeds maxTokens or the entry count exceeds maxEntries.

When triggered, summarizeIfNeeded(history, llm, config) collapses the oldest entries into a single "summary" HistoryEntry via a dedicated LLM call.

```
HistoryConfig config = HistoryConfig.builder()
    .maxEntries(40)       // default 40
```

```
    .preserveRecent(10)   // default 10, never summarized away
    .maxTokens(8000)      // default 8000
    .targetChars(1000)    // default 1000, target size of the summary entry
    .build();
```

Only CODE_ACT and TOOL_CALLING use history by default

Since PredictStrategy and ReflexionStrategy's base calls are typically single-shot structured output rather than multi-turn tool loops, they don't read from or write to history by default. If a multi-turn conversational memory is needed regardless of strategy, an agent can still read/append to memory directly from its own methods.

9. Context Blocks

ContextApi lets an agent contribute additional, named blocks of text into its own system prompt, automatically rendered under a "## Context" section every time a prompt is built. Two kinds of block are supported:

9.1 Static blocks

put(key, value) stores a fixed value once, cached for the agent's lifetime, appropriate for content that doesn't change between calls, such as a project brief or a sprint plan set up at construction time:

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

9.2 Dynamic blocks

setDynamic(key, selfExpression) instead stores a self-referencing expression string, re-evaluated via org.jopenagent.core.PromptTemplate every time a prompt is built. Use this for content that changes between calls, such as a live backlog summary:

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

PromptTemplate.expand supports exactly {self.field}, {self.method()}, dotted chains, and the literal {doc(self)} form, a deliberately restricted subset, not arbitrary code evaluation.

Anything outside that grammar throws IllegalArgumentException with a clear message rather than silently doing nothing.

Other ContextApi operations: get(key), remove(key), keys(), and render() (assembles all currently-registered blocks into the rendered "## Context" prompt section).

Skills can contribute context blocks too

A Skill's optional contextBlock() (see Chapter 7) is wired into the same ContextApi mechanism when the skill is loaded. A skill isn't limited to static instructions. It can also contribute a dynamic, self-referencing context block exactly like an agent's own code can.

10. Multimodal Content

org.jopenagent.llm.media.Media/Image/Audio/Video/FileAttachment let a @Generative method accept media as a real, typed parameter (rather than requiring the caller to base64-encode it) in the text prompt:

```
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) pulls any Media-typed bound parameters out of normal text rendering and attaches them as genuine multimodal content blocks on the outgoing LlmRequest.

Each concrete LLM client is then responsible for its own provider-specific mapping onto that provider's wire format (see Chapter 14).

Type	Notes
Image	The fully verified path, used directly by both AnthropicClient and OpenAiClient as native image content.
Audio / Video	Anthropic has no native audio/video content-block type, so AnthropicClient falls back to sending an image-shaped block for these.
FileAttachment	General-purpose file content, mapped to each provider's document/file block type.

11. Tracing & Observability

Every agent is traced by default: every LLM call, code execution, tool execution, and generation call is automatically wrapped in a Span, whether or not any exporter is attached.

org.jopenagent.trace.Tracer maintains a per-thread ThreadLocal<Deque<Span>> stack.

Tracer.start(kind, name) parents a new span to whatever is currently on top of the calling thread's stack; Tracer.end(span) pops it and, once the tree reaches its root, hands the complete span tree to every registered TraceExporter.

11.1 Span and the exporter contract

A Span carries an id/parentId, kind, name, start/end timestamps, status, an optional error message, an attribute map, and its children.

TraceExporter has one method (void export(Span rootSpan)) called exactly once per completed root span, after its entire subtree has already been built.

org.jopenagent.trace.Traced is a convenience wrapper (begin/finish/fail) that is null-tolerant throughout, becoming a no-op whenever tracing is disabled.

11.2 Export formats

Exporter	Format / destination
JsonTraceExporter	Plain span-tree JSON, optionally appended to a JSONL file. This is the simplest format, useful for quick local inspection or feeding into custom tooling.
AtifExporter	ATIF-v1.7 (agent-trace-interchange-format): flattens a root span's children into steps. An llm_call child becomes an agent-source step, a tool_execution/code_execution child becomes an observation step, and a nested generation span becomes a subagent_trajectories[] entry.
OtlpHttpExporter	Standard OTLP/HTTP JSON (ExportTraceServiceRequest), with the traceId derived from the root span's UUID.
LangfuseExporter	Extends OtlpHttpExporter with Basic authentication and Langfuse's specific ingestion path.

```
Tracer.addExporter(new JsonTraceExporter(Paths.get("jooa-trace.jsonl")));
agent.writeHaiku("autumn");
// every span produced by that call is now captured by the exporter
```

11.3 Trace viewers

Three ways to inspect captured traces, all built entirely on standard-library dependencies:

Viewer	Basis
TraceViewerServer (org.jopenagent.trace.viewer)	An embedded web app on com.sun.net.httpserver. A single self-contained HTML/vanilla-JS page lists traces, browses memory, supports annotations, and includes an eval-harness "Experiments" tab (see Chapter 13).
TraceViewerSwingApp (org.jopenagent.trace.viewer.swing)	A native desktop Swing application, zero external dependency, usable either live (registered directly as a TraceExporter, so traces appear as each call completes) or in a remote polling mode against a separately-running TraceViewerServer.
TerminalTraceExplorer (org.jopenagent.trace.explorer)	A PrintStream-based span-tree pretty-printer for quick console inspection with no server or window at all.

12. Code-as-Action Sandbox

CodeExecutor (an AutoCloseable interface: execute(String)/close()) is the execution abstraction behind StrategyKind.CODE_ACT.

Two implementations are available (see AgentConfig.getSandboxConfig().getMode()).

12.1 In-process: CodeSandbox (default)

The default backend is a persistent JShell session running inside the same JVM, with org.jopenagent.sandbox.SandboxBindings wiring a live self reference so generated code can call the agent's own methods directly.

Each call is wrapped by a watchdog thread that calls jshell.stop() on timeout, a cooperative, yield-only mechanism that cannot preempt a tight, genuinely infinite loop such as while (true) {}.

Before execution, org.jopenagent.sandbox.CodeValidator runs a tokenizer-based (string/comment-aware) denylist over the submitted code, rejecting dangerous packages (java.lang.reflect, java.net, java.nio.file, sun.*, etc.) and simple type names (Runtime, ProcessBuilder, File, Socket, etc.).

> **CodeValidator is defense-in-depth only, not a security boundary**
>
> A static denylist cannot fully prevent a sufficiently determined program from reaching the filesystem or network via reflection or other indirection. Always run agents that execute generated code inside OS-level isolation (a container or VM); do not rely on jOpenAgent's in-process validation alone as your only safety measure.

12.2 Out-of-process: OutOfProcessCodeSandbox

OutOfProcessCodeSandbox is an opt-in upgrade, enabled via SandboxConfig.builder().mode(OUT_OF_PROCESS).timeoutMillis(...).heapMb(...).build().

It launches SandboxWorkerMain as a genuinely separate child JVM (with its own -Xmx heap cap), and a plain generated Java stub class (SelfStubGenerator emits ordinary source text, not bytecode) forwards self.method(...) calls over a bidirectional loopback-socket RpcChannel to RemoteSelfDispatch, which dispatches them host-side via the same ToolRegistry that backs TOOL_CALLING.

A timeout under this mode is a real Process.destroyForcibly() kill, not a cooperative stop() request.

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

A dead worker is transparently relaunched on the next call.

12.3 Choosing a mode

Mode	Guarantees	Cost
IN_PROCESS (default)	Cooperative timeout (jshell.stop()) only.	No process-launch overhead
OUT_OF_PROCESS	Real hard-kill timeout.	Child-JVM startup cost per session

OUT_OF_PROCESS upgrades timeout containment only

Switching to SandboxMode.OUT_OF_PROCESS does not add filesystem or network isolation. It only guarantees that a runaway execution is genuinely killed rather than merely asked to stop.

13. Evaluation Harness

org.jopenagent.eval provides a small, self-contained evaluation harness: define a suite of cases, run them, score the results.

Each EvalRunner.run(...) call wraps every case in its own eval_case trace span before the task itself runs, so any spans the task produces (the generation span from a @Generative call, its nested llm_call/tool_execution/code_execution children) nest naturally inside it.

The full eval.* attribute set (experiment name, tier, model, pass/fail, score, duration) lives on that span, so a normal TraceExporter captures everything the eval harness needs, and

TraceViewerServer's "Experiments" tab rebuilds experiment summaries purely by scanning captured traces for eval_case-kind spans.

13.1 Building and running a suite

```
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");
```

A thrown exception from an EvalTask becomes a failing ScoreDetail rather than propagating out of EvalRunner.run(...), one bad case does not abort the whole run.

13.2 Scorers and tiers

Scorer	Compares
ExactMatchScorer	Exact equals() match between expected and actual.
ContainsScorer	Case-insensitive substring match, for String results.
NumericToleranceScorer(tolerance)	Numeric closeness within a fixed tolerance, for Number results.
LlmJudgeScorer<T>	Asks a separate LLM to render {score, passed, reasoning} against a rubric (default or custom), itself captured as a nested llm_call span; a judge-call failure becomes a failing ScoreDetail rather than throwing.

Tier (STABLE / FRONTIER / HORIZON) tags a case's expected difficulty, STABLE cases must pass, FRONTIER cases probe the edge of current capability, HORIZON cases are aspirational, letting a results view group and interpret failures by how much they should matter.

14. LLM Clients

LlmClient is the model-agnostic abstraction every strategy talks to: a single blocking round trip, LlmResponse generate(LlmRequest request), plus modelName().

Clients hold no conversational state of their own (the full message history is re-sent on every call) and must throw LlmException rather than a raw I/O exception on failure. Two concrete implementations ship with jOpenAgent, both built entirely on java.net.http.HttpClient.

14.1 AnthropicClient

Targets the Anthropic Messages API. Two constructors: AnthropicClient(apiKey, model) for the hosted API (30-second connect timeout), and a four-argument form accepting a custom base URL and HttpClient for proxies or test doubles.

Notable wire-mapping behavior: the system prompt becomes the request's top-level system field; any extra SYSTEM-role turns beyond the first are folded into a "[system note]"-prefixed user turn; and because Anthropic requires all tool_result blocks for one assistant turn to arrive inside a single following user message, consecutive TOOL messages are merged into one user turn carrying multiple tool_result blocks. Media maps to image or document content blocks; Audio and Video fall back to an image-shaped block, since Anthropic has no native audio/video block type.

AnthropicClient does not stream.

14.2 OpenAiClient

Targets the OpenAI Chat Completions API, and since that same API shape is also what Ollama and LM Studio expose locally, OpenAiClient doubles as the client for both of those, via dedicated factory methods:

Factory / constructor	Target
`new OpenAiClient(apiKey, model)`	Hosted OpenAI API (30s connect / 120s request timeout).
`OpenAiClient.forOllama(model, baseUrl)`	A local Ollama server.
`OpenAiClient.forLmStudio(model, baseUrl)`	A local LM Studio server (default http://localhost:1234).
`OpenAiClient.forLocalServer(model, baseUrl)`	Any other local OpenAI-compatible server, with a longer 600-second timeout to accommodate slower local inference.

apiKey may be null or empty for keyless local servers.

Unlike AnthropicClient, OpenAiClient performs no message folding: the Chat Completions API allows one tool-result message per call, correlated back to its originating call via tool_call_id, so each LlmMessage maps to exactly one wire-format message. Media maps to input_audio, file, or image_url blocks depending on the concrete Media subtype.

OpenAiClient does not stream.

Programmatic configuration is always available

Every environment-variable-driven setup shown in Chapter 2 (via ExampleSetup) is a convenience on top of ordinary constructors and factory methods. Any application can construct AnthropicClient or OpenAiClient directly, with explicit credentials, base URLs, timeouts, or a custom HttpClient, without relying on environment variables at all.

15. Tutorial Walkthrough

org.jopenagent.examples contains 19 example programs (E01 to E19).

15.1 E01_FirstGenerationMethod

The smallest possible agent: one @Generative method, greet(String name), illustrating the @SystemPrompt / @Generative / generate() pattern end to end with the default PREDICT strategy. A good place to start.

15.2 E02_StructuredOutput

Introduces a record return type, ReviewSummary(int rating, String sentiment, List<String> highlights). Shows ObjectBinder automatically parsing, validating, and binding the LLM's JSON onto the record, retrying with the validation error fed back to the model if the first attempt doesn't bind cleanly.

15.3 E03_ToolsViaSelf

An inventory assistant using StrategyKind.TOOL_CALLING: ordinary @Doc-annotated methods (getStock, knownItems) are exposed as native JSON tools via ToolRegistry, and the model decides which to call before answering.

15.4 E04_Strategies

The same pricing/classification task solved three different ways side by side: PREDICT (a single structured-output call), CODE_ACT (the model writes Java calling self.unitPriceCents(...) inside the sandbox), and TOOL_CALLING (native function-calling against the same method), a direct, practical comparison of the three non-Reflexion strategies.

15.5 E05_ProgressiveDisclosure

Illustrates Agent.describe() and the visible-by-default model: a public field shows up automatically, a field marked @Hidden does not, and the {doc(self)} prompt-template form lets the LLM discover the agent's own capabilities at runtime.

15.6 E06_Tracing

Registers a JsonTraceExporter before running a haiku-writing agent, showing that every generation call is automatically wrapped in a Span regardless of whether an exporter is attached, then prints the captured parent-child span tree as JSON.

15.7 E07_ContextBlocks

A sprint-planning assistant using self.context: a static block (put, defined once, the sprint plan) and a dynamic block (setDynamic, re-evaluated on every prompt build via a self.formatBacklog() expression), both auto-rendered into the system prompt.

15.8 E08_CodeAsAction

The CODE_ACT mode: the model writes and executes Java that calls orderTotalsCents() inside CodeSandbox to compute an average, iterating via execute_java/return_result until done.

15.9 E09_HistorySummarization

A customer-support shift-logging assistant using TOOL_CALLING, configured with an artificially small HistoryConfig so summarization triggers after only a few calls. A compact demonstration of HistorySummarizer collapsing older entries into a single Summary/Topics/Outcomes/State entry.

15.10 E10_Skills

Attaches a TextSkill built from an in-memory SKILL.md (frontmatter name/description plus a house-style body) to a support agent, showing the skill's name and description rendered for progressive disclosure, and its full instructions pulled in on demand as reply generation implicitly follows the skill's guidance.

15.11 E11_McpTools

Attaches the official @modelcontextprotocol/server-everything reference MCP server (launched via npx) as a plain McpClient field. Illustrates its echo tool reachable via TOOL_CALLING's auto-merged everything__echo tool, and directly from CODE_ACT-generated code (self.everything.callTool(...)). Requires Node's npx on PATH.

15.12 E12_Memory

A support agent using self.memory (present by default on every agent): thin wrapper methods (rememberFact, recallFacts) delegate to MemoryManager.remember/recall, called by the model via TOOL_CALLING across multiple conversational turns.

15.13 E13_Multimodal

Generates a red-circle PNG and passes it as an @Generative parameter typed Image, demonstrating that media is attached as real multimodal content.

15.14 E14_AtifTrajectory

A warehouse assistant using TOOL_CALLING; registers an AtifExporter and prints the resulting ATIF-v1.7 trajectory document for one generation call.

15.15 E15_Reflexion

An arithmetic word-problem solver using StrategyKind.REFLEXION. It illustrates the self-critique-and-retry loop (up to 3 attempts) on a problem deliberately chosen to test whether the model gets the arithmetic right after receiving its own critique as feedback.

15.16 E16_TraceViewer

Starts an embedded TraceViewerServer, runs a couple of joke-generation calls, then waits for the user to open a browser and inspect the trace list and span tree before pressing Enter to stop the server.

15.17 E17_TraceViewerSwing

The native-desktop alternative to E16: registers TraceViewerSwingApp directly as a TraceExporter so traces appear live in a Swing interface as each call completes.

15.18 E18_OutOfProcessSandbox

The same CODE_ACT averaging task as E08, but configured with SandboxMode.OUT_OF_PROCESS. Its main() then proves the concrete benefit directly (bypassing the LLM entirely) by handing a tight while (true) {} loop to a raw OutOfProcessCodeSandbox and showing it gets force-killed after the configured timeout rather than hanging forever, unlike the in-process default.

15.19 E19_EvalHarness

A support-ticket-triage classifier evaluated via EvalSuite/EvalCase/EvalRunner with a ContainsScorer across four cases (including one deliberately ambiguous case tagged Tier.FRONTIER).

15.20 ExampleSetup

Shared setup that each example calls. createLlmClient() picks a provider from environment variables in priority order (Anthropic, then OpenAI, then LM Studio, then Ollama), throwing a clear instructional IllegalStateException if none are set. createEmbedder() similarly picks HashingEmbedder (default) or a real OpenAiEmbedder based on JOPENAGENT_EMBEDDING_MODEL plus whichever provider is already configured.

Model size

Small (roughly 4-billion-parameter) local models were observed to struggle with CODE_ACT and multi-turn TOOL_CALLING conventions during testing, inventing data, or never calling return_result. A 30-billion-parameter code-specialized model handled every case correctly. This is a model-capability limitation, not a jOpenAgent defect; the framework raises GenerationException rather than hanging when a model never converges.

Appendix A: Full Package & Class Inventory

The complete, source-scanned inventory of every package and class under org.jopenagent (plus the pre-existing org.jopenkit.json parser it builds on): 26 packages, 144 classes/interfaces/enums/annotations in total. This table is generated directly from the source tree (docs/docgen/scan-source.js) rather than hand-transcribed, so it stays accurate as the codebase grows.

Package	Purpose	Classes
`org.jopenagent.agentdoc`	The doc(self)-equivalent introspection machinery (Introspector, MemberScanner, Visibility).	Introspector, MemberScanner, Visibility
`org.jopenagent.core`	Agent base class, the @SystemPrompt/@Generative/@Hidden/@Shown/@NoTrace/@Doc annotations, the generate() trampoline, GenerationEngine, PromptTemplate/PromptBuilder, AgentConfig, and GenerationException.	Agent, AgentConfig, CallContext, Doc, GenerationEngine, GenerationException, Generative, Hidden, NoTrace, PromptBuilder, PromptTemplate, Shown, SystemPrompt
`org.jopenagent.core.context`	self.context, static and dynamic context blocks rendered into the system prompt.	ContextApi
`org.jopenagent.core.history`	self.history. The persistent cross-call conversation log and its HistorySummarizer.	ConversationHistory, HistoryConfig, HistoryEntry, HistorySummarizer
`org.jopenagent.core.strategy`	The Strategy interface and its four implementations	CodeActStrategy, PredictStrategy,

Package	Purpose	Classes
	(PredictStrategy, CodeActStrategy, ToolCallingStrategy, ReflexionStrategy) plus StrategyKind.	ReflectionOutcome, ReflexionStrategy, Strategy, StrategyKind, ToolCallingStrategy
`org.jopenagent.eval`	The evaluation harness: EvalCase/EvalSuite/EvalRunner /EvalTask/EvalResult/Tier, and the Scorer interface with its ExactMatchScorer/ContainsSc orer/NumericToleranceScorer/ LlmJudgeScorer implementations.	ContainsScorer, EvalCase, EvalResult, EvalRunner, EvalSuite, EvalTask, EvalTraceView, ExactMatchScorer, LlmJudgeScorer, NumericToleranceScorer, ScoreDetail, Scorer, Tier
`org.jopenagent.examples`	E01..E19 plus ExampleSetup, a progressive tutorial mirroring nooa/examples/quickstart.	E01_FirstGenerationMethod, E02_StructuredOutput, E03_ToolsViaSelf, E04_Strategies, E05_ProgressiveDisclosure, 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 (reflection-based JSON <-> record/POJO/List/Map/Option al/enum binding) plus SchemaGenerator and TypeUtils, built on org.jopenkit.json.	BindingException, ObjectBinder, SchemaGenerator, TypeUtils
`org.jopenagent.llm`	The model-agnostic LlmClient/LlmRequest/LlmRes	LlmClient, LlmException, LlmMessage, LlmRequest,

Package	Purpose	Classes
	ponse/LlmMessage/ToolSpec/ ToolCall abstractions.	LlmResponse, ToolCall, ToolSpec
`org.jopenagent.llm.anthropic`	AnthropicClient. The Anthropic Messages API concrete client, built on java.net.http only.	AnthropicClient
`org.jopenagent.llm.media`	Media/Image/Audio/Video/ FileAttachment, multimodal message content.	Audio, FileAttachment, Image, Media, Video
`org.jopenagent.llm.openai`	OpenAiClient. The OpenAI Chat Completions API concrete client (real OpenAI, Ollama, LM Studio), also java.net.http only.	OpenAiClient
`org.jopenagent.mcp`	McpClient (stdio JSON-RPC 2.0), McpServerConfig, McpToolDefinition.	McpClient, McpException, McpServerConfig, McpToolDefinition
`org.jopenagent.memory`	Memory, MemoryManager, MemoryStore (InMemory/JsonFile) and Embedder (HashingEmbedder by default; OpenAiEmbedder for real or local-server embeddings).	Embedder, HashingEmbedder, InMemoryMemoryStore, JsonFileMemoryStore, Memory, MemoryEdge, MemoryException, MemoryManager, MemoryRetrievalConfig, MemoryStore, OpenAiEmbedder, VectorMath
`org.jopenagent.sandbox`	The CodeExecutor abstraction, the in-process CodeSandbox (JShell), CodeValidator's denylist, SandboxResult and SandboxConfig/SandboxMode.	CodeExecutor, CodeSandbox, CodeValidator, JShellScriptRunner, SandboxBindings, SandboxConfig, SandboxException, SandboxMode, SandboxResult
`org.jopenagent.sandbox.remote`	OutOfProcessCodeSandbox, the opt-in child-JVM worker, RpcChannel (bidirectional JSON-RPC over a loopback socket), and SelfStubGenerator/RemoteSelf Dispatch.	JavaTypeSource, OutOfProcessCodeSandbox, RemoteSelfDispatch, RpcChannel, RpcException, SandboxWorkerMain, SelfStubGenerator
`org.jopenagent.skills`	Skill, TextSkill (including	ScriptResult, ScriptRunner,

Package	Purpose	Classes
	runScript), SkillRegistry, ScriptRunner, packaged capability/prompt bundles.	Skill, SkillException, SkillRegistry, TextSkill
org.jopenagent.tools	ToolRegistry, exposes an agent's ordinary methods and any attached MCP tools as JSON-schema tools for native function calling.	ToolInvocationException, ToolRegistry
org.jopenagent.trace	Span, Tracer (the per-thread parent-child span stack), the TraceExporter contract, and JsonTraceExporter.	JsonTraceExporter, Span, TraceException, TraceExporter, Traced, Tracer
org.jopenagent.trace.atif	AtifExporter, ATIF-v1.7 agent-trace-interchange-format export.	AtifExporter
org.jopenagent.trace.explorer	TerminalTraceExplorer, a span-tree pretty-printer for the console.	TerminalTraceExplorer
org.jopenagent.trace.otlp	OtlpHttpExporter (standard OTLP/HTTP JSON) and LangfuseExporter.	LangfuseExporter, OtlpHttpExporter
org.jopenagent.trace.viewer	TraceViewerServer, the embedded trace-viewer web app (jdk.httpserver only), including memory-browsing, annotations and the eval-harness "Experiments" routes.	TraceViewerPage, TraceViewerServer
org.jopenagent.trace.viewer.swing	TraceViewerSwingApp, the native desktop trace viewer (Swing, zero dependency).	JsonTraceNode, TraceViewerSwingApp
org.jopenagent.util	ProcessLaunch. The shared Windows-.cmd-safe process launch helper used by McpClient and skill scripts.	ProcessLaunch
org.jopenkit.json	The pre-existing, untouched hand-rolled JSON parser (JSONObject/JSONArray/JSONTokener) that	JSON, JSONAble, JSONArray, JSONException, JSONObject, JSONStringer, JSONTokener, TypeAdapter, TypeAdapters,

Package	Purpose	Classes
	org.jopenagent.json's parsing/tree layer builds on.	WithClassTypeAdapter

Appendix B: Central Class Reference

A quick-reference index of the classes and interfaces most central to using jOpenAgent day to day, grouped by the feature area each belongs to (matching Chapters 3-13). For the complete inventory of all classes in the framework, see Appendix A.

B.1 Core Agent Model

Agent (org.jopenagent.core)

Base class for jOpenAgent agents.

AgentConfig (org.jopenagent.core)

Per-agent settings.

Generative (org.jopenagent.core)

Marks a method whose body delegates to the LLM via Agent#generate(Object...).

SystemPrompt (org.jopenagent.core)

Class-level system prompt for an Agent.

Doc (org.jopenagent.core)

Description text for an ordinary (non-@Generative) method or field, substituting for javadoc (not reflectable at runtime).

Hidden (org.jopenagent.core)

Hides a field or ordinary method from doc(self) rendering, the code-as-action sandbox's exec scope, and tools-via-self schema generation.

Shown (org.jopenagent.core)

Re-exposes a non-public field or method that would otherwise be hidden by default (mirrors NOOA's @spec(hidden=False) opt-in for _private-style methods).

GenerationEngine (org.jopenagent.core)

Orchestrates one @Generative call: resolves the strategy, wraps it in a trace span, runs it.

B.2 Generation Strategies

Strategy (org.jopenagent.core.strategy)

Executes one @Generative method call end-to-end and returns the bound result.

PredictStrategy (org.jopenagent.core.strategy)

Single-shot structured output with retry: ask the LLM for one JSON object matching the return type's schema; on a JSON-parse or validation failure, feed the raw response and the error back to the LLM and retry, up to CallContext.getMaxRetries() attempts.

CodeActStrategy (org.jopenagent.core.strategy)

Code-as-action: gives the LLM two tools, execute_java and return_result, and loops until it calls return_result.

ToolCallingStrategy (org.jopenagent.core.strategy)

Classic function-calling (ReAct-style) loop: the LLM calls the agent's ordinary methods (exposed as JSON-schema tools by ToolRegistry) until it calls the synthetic return_result tool with its final answer, which is validated against the return type exactly like PredictStrategy does, with the same feed-the-error-back retry behavior.

ReflexionStrategy (org.jopenagent.core.strategy)

Generate -> reflect -> improve loop: run a base strategy, ask a separate structured LLM call whether the result is satisfactory, and if not, retry the base strategy with the critique's issues/suggestions folded into the instruction — up to maxIterations attempts, returning the best-effort result if it never becomes satisfactory.

StrategyKind (org.jopenagent.core.strategy)

Selects which Strategy handles a @Generative method.

B.3 Structured Output

ObjectBinder (org.jopenagent.json)

Reflection-based binding between org.jopenkit.json trees (JSONObject/JSONArray/scalars) and arbitrary Java types: records, plain POJOs (no-arg constructor + fields), List/Set/array, Map, Optional, enums and primitives/wrappers/String.

B.4 LLM Clients & Tools

LlmClient (org.jopenagent.llm)

Model-agnostic LLM client abstraction.

AnthropicClient (org.jopenagent.llm.anthropic)

Anthropic Messages API (POST /v1/messages) client.

OpenAiClient (org.jopenagent.llm.openai)

Chat-Completions-compatible client (POST /v1/chat/completions) built entirely on java.net.http.HttpClient.

ToolRegistry (org.jopenagent.tools)

Exposes an agent's ordinary (non-@Generative, visible) methods as JSON-schema tools a provider's native function-calling API can invoke directly.

B.5 History & Context

ConversationHistory (org.jopenagent.core.history)

An agent's persistent, cross-call conversation log.

HistorySummarizer (org.jopenagent.core.history)

Compresses the oldest run of a ConversationHistory into a single summary entry once a configured trigger fires, via a dedicated LLM call.

ContextApi (org.jopenagent.core.context)

self.context equivalent.

B.6 Memory

MemoryManager (org.jopenagent.memory)

remember/recall/update/forget/associate/traverse API: persistent, cross-session memory with similarity-based dedup-on-write, a typed graph of edges between memories, and ACT-R-style retrieval scoring.

InMemoryMemoryStore (org.jopenagent.memory)

Ephemeral, process-lifetime memory store (no persistence) — the default.

JsonFileMemoryStore (org.jopenagent.memory)

Cross-session persistence for Memory records, backed by a single JSON file.

Embedder (org.jopenagent.memory)

Turns text into a fixed-size, L2-normalized embedding vector for similarity-based recall.

HashingEmbedder (org.jopenagent.memory)

A deterministic, offline, no-network embedder, feature-hashing (bag-of-hashed-tokens) into a fixed-size vector, then L2-normalized.

OpenAiEmbedder (org.jopenagent.memory)

Real embeddings over /v1/embeddings, built entirely on java.net.http.HttpClient.

B.7 Skills

SkillRegistry (org.jopenagent.skills)

Per-agent skill lifecycle: load/unload/lookup, plus discovery of a directory of SKILL.md bundles.

Skill (org.jopenagent.skills)

A packaged capability/prompt bundle attached to an agent.

TextSkill (org.jopenagent.skills)

Loads a SKILL.md/skill.md directory.

B.8 Tracing & Observability

Tracer (org.jopenagent.trace)

Parent-child span tracking.

Span (org.jopenagent.trace)

One traced unit of work: an LLM generation call, a code execution, a tool execution, or an ordinary traced method invocation.

TraceExporter (org.jopenagent.trace)

Receives completed root spans (a full call tree) from Tracer.

JsonTraceExporter (org.jopenagent.trace)

Exports a completed span tree to JSON: one line (JSONL) per root span if a file is given, otherwise kept in memory for inspection (useful in tests, or for a future viewer UI to poll).

AtifExporter (org.jopenagent.trace.atif)

Exports a completed span tree as an ATIF-v1.7 trajectory JSON document.

OtlpHttpExporter (org.jopenagent.trace.otlp)

Exports a span tree as an OTLP/HTTP JSON ExportTraceServiceRequest.

LangfuseExporter (org.jopenagent.trace.otlp)

Pre-configured OtlpHttpExporter for Langfuse's OTLP ingestion endpoint.

B.9 Sandbox

CodeSandbox (org.jopenagent.sandbox)

Code-as-action execution: a persistent JShell REPL session with a live self variable bound to the agent instance, so generated code can call self.someMethod(...) directly.

CodeExecutor (org.jopenagent.sandbox)

Common shape for CODE_ACT's code-execution backend, so org.jopenagent.core.strategy.CodeActStrategy doesn't need to know whether it's talking to the default in-process CodeSandbox or the opt-in org.jopenagent.sandbox.remote.OutOfProcessCodeSandbox.

OutOfProcessCodeSandbox (org.jopenagent.sandbox.remote)

Out-of-process CODE_ACT backend: launches SandboxWorkerMain as a genuinely separate child JVM (its own heap, via -Xmx), and executes generated code there instead of in the host process.

B.10 Evaluation Harness

EvalRunner (org.jopenagent.eval)

Runs an EvalSuite and scores each case.

EvalSuite (org.jopenagent.eval)

A named group of EvalCases — jOpenAgent's "experiment" concept.

Scorer (org.jopenagent.eval)

Compares an eval case's actual output against its expected value and produces a ScoreDetail.

ExactMatchScorer (org.jopenagent.eval)

Passes when actual.equals(expected) (or both are null); the simplest possible scorer.

ContainsScorer (org.jopenagent.eval)

Passes when actual (case-insensitively) contains expected as a substring, useful for free-text generative output.

NumericToleranceScorer (org.jopenagent.eval)

Passes when |actual - expected| <= tolerance, for numeric outputs where an exact match is too strict.

LlmJudgeScorer (org.jopenagent.eval)

Scores a case by asking an LLM to judge whether actual satisfies expected against a rubric, instead of an exact/substring/numeric comparison, for free-text output where there's no single correct string (e.g.