

COURSE OUTLINE

Java 25 for AI Engineering — Build Reliable RAG and Agentic Applications

An intensive, lab-driven program for Java and Spring developers building useful, safe LLM-powered services

DURATION

FORMAT

5 days

Instructor-led, lab-driven, cumulative capstone project

DELIVERY

TRAINER

Online Instructor Led Workshop · Onsite Program for your Teams

Muhammed Shakir

Why this course exists

Enterprise Java teams already know how to build APIs, integrate systems, manage data, and run reliable services. The missing piece is not a chatbot SDK: it is the ability to decide when an LLM should retrieve information, call a bounded business tool, ask for clarification, or stop—and to prove that the resulting system behaves safely under real operational conditions.

This program turns familiar Java and Spring skills into an engineering toolkit for source-backed RAG and bounded agentic applications. Participants build one inspectable Financial Guidance Assistant: a Spring Boot service that retrieves approved policy documents, accesses a simulated account system through safe tools, returns evidence with its answers, and leaves a trace that can be tested and reviewed.

By the end of five days, participants will have built and evaluated a working Java 25 application with model integration, retrieval, citations, validated tool calls, bounded workflows, and production-minded controls. They leave able to judge where an LLM belongs in a Java system—not merely how to add one.

What makes this different

Most AI courses treat Java as a wrapper around an API call, or present agents as autonomous magic.

This is an engineering room: every AI capability has an explicit contract, a testable boundary, and a running implementation. You will:

- **Build from a dependable Java baseline** — start with a Spring Boot service, typed model outputs, configuration discipline, and observable request handling before introducing retrieval or agents.

- **Earn each layer of autonomy** — establish cited retrieval first, then add a small, allow-listed tool set and a bounded workflow with identity propagation, confirmation points, and safe exits.
- **Measure behavior rather than trust demos** — maintain an evaluation set, inspect retrieval and tool traces, inject failures and hostile inputs, and compare quality, latency, cost, and failure modes.

The course favours small, explainable systems and deliberate engineering trade-offs over framework tours and promises of fully autonomous enterprise agents.

Skills You'll Walk Away With

- Build a Java 25 and Spring Boot LLM service with portable model configuration, streaming responses, structured outputs, and safe secret handling.
- Design a RAG pipeline that preserves document provenance, permissions, citations, answer boundaries, and abstention behavior.
- Implement semantic retrieval, hybrid retrieval, and re-ranking, then compare them against a version-controlled evaluation set.
- Expose Java business capabilities as narrowly scoped, validated tools with user context, authorization, and auditable outcomes.
- Integrate selected MCP tools and expose a controlled Java capability through MCP without granting the model direct infrastructure access.
- Orchestrate bounded multi-step AI workflows using virtual threads, timeouts, budgets, retries, circuit breakers, and human approval for consequential actions.
- Test, observe, secure, package, and explain an AI service using traces, redaction, adversarial cases, latency and cost measurements, and operational runbooks.

Who should attend

This program is for experienced Java professionals who need to turn AI capabilities into software their teams can understand, test, and operate.

- **Java and Spring Boot developers** — adding grounded LLM capabilities to existing services and APIs.
- **Backend and platform engineers** — designing integration, concurrency, resilience, and observability boundaries for AI workloads.
- **Solution and enterprise architects** — deciding where RAG, tools, workflow orchestration, and conventional software each belong.
- **Technical leads and engineering managers** — reviewing AI implementation choices, safety controls, and delivery risks.
- **QA and automation engineers** — building repeatable checks for retrieval quality, structured output, tool behavior, and adversarial failure modes.

Prerequisites: confidence with Java, Spring Boot, REST APIs, Git, tests, and SQL or document-database basics. Familiarity with asynchronous programming is useful. Prior machine-learning experience is not required; LLM concepts are introduced in plain engineering language.

Course Outline

Day 1 — Establish the AI Service Foundation

Make the model call a well-engineered part of a Java service before asking it to retrieve information or use tools.

Session 1 — Design the System Before the Prompt

Define the Financial Guidance Assistant's supported decisions, trust boundaries, and test cases, then make a minimal Java 25 service observable and safe to run.

- LLM mental model for Java developers: tokens, context windows, temperature, structured outputs, streaming, latency, and cost as service-level trade-offs
- Java 25 and Spring Boot foundations for AI services: LTS runtime, configuration profiles, virtual threads for blocking external calls, and when ordinary request-response code is enough
- Model-provider portability: model, embedding, and chat interfaces; comparing hosted and local development models without hard-coding application logic to a provider
- Trust boundaries: user input, model output, retrieved content, tool arguments, secrets, personal data, logs, and third-party MCP servers
- Supported versus unsupported requests: informational policy questions, account lookups, simulation-only recommendations, clarification, refusal, escalation, and actions that require human approval
- Evaluation from day one: a version-controlled set of answerable, unanswerable, ambiguous, permission-sensitive, and adversarial requests
- *Hands-on:* Establish the runnable capstone baseline.
 - Create the Spring Boot project in `/Users/shakir/learning-capstones/java-25-ai-assistant` with Java 25, a health endpoint, a local development command, configuration validation, and secret-loading boundaries.
 - Implement a provider-neutral chat adapter behind an application interface; configure at least one hosted model and one local-development option without placing credentials in source control or logs.
 - Return a typed response envelope containing request ID, answer status, model metadata, latency, token/cost fields where available, and a safe error representation.
 - Write `docs/supported-workflows.md` to define allowed financial-guidance scenarios, prohibited advice and transactions, required clarifications, and human-escalation paths.
 - Create `evaluation/v1.jsonl` with expected behavior—not only ideal answers—for policy, account, unknown, ambiguous, and restricted-user scenarios.
 -

Add automated checks for configuration validation, absent secrets in health/error responses, and the response-envelope schema.

- **Session checkpoint:** a tested Java 25 service starts locally, responds to health checks, calls a configured model through a replaceable adapter, and has a documented scope and initial evaluation set. Retrieval, tools, and agent loops are deliberately not implemented yet.

Day 2 — Build Grounded Retrieval, Not Document Chat

Turn approved documents into inspectable evidence and make the service decline answers it cannot support.

Session 2 — Ingest, Retrieve, Cite, and Evaluate

Build the source-backed policy knowledge path before adding live account data or tool calling.

- Retrieval architecture: model knowledge versus supplied evidence; document ingestion, chunks, embeddings, indexes, metadata, retrieval, context assembly, and answer generation
- Corpus engineering: cleaning, chunk boundaries, parent-child references, document versions, source hashes, updates, deletions, and permission metadata
- Semantic retrieval, keyword retrieval, hybrid fusion, and re-ranking: what each helps with, where it adds cost or latency, and how to inspect failure
- Grounded answers: context budgets, citations, source excerpts, conflict handling, unsupported-answer abstention, and the limit of prompt instructions as a safety control
- Java integration choices: Spring AI or LangChain4j abstractions, vector-store adapters, and why retrieval logic remains visible in application code and tests
- *Hands-on:* Build the assistant's policy RAG path and measure its baseline.
 - Ingest a supplied financial-product policy corpus into PostgreSQL with pgvector (or the cohort's selected supported vector store), retaining document ID, version, section, source hash, access attributes, and parent-document references.
 - Implement idempotent ingestion, chunk validation, embedding generation, semantic retrieval, and a retrieval-debug API that exposes permitted source metadata and scores.
 - Add hybrid retrieval and an optional re-ranking stage; retain candidate order and scores before and after each stage for inspection.
 - Assemble a bounded context and generate an answer only from retrieved sources. Return section-level citations and supporting excerpts; abstain or ask for clarification when evidence is missing, contradictory, or outside the approved corpus.
 - Extend the evaluation harness to run `semantic`, `hybrid`, and `hybrid-reranked` modes over the same questions and record retrieval relevance, citation support, answer quality, latency, and cost.
 - Add tests for source provenance, index idempotency, access filtering before context assembly, unsupported-answer abstention, and citation-to-evidence validity.
 - **Session checkpoint:** a working, tested cited-answer endpoint with independently selectable retrieval modes and an initial comparison report.

Day 3 — Give the Model Safe Ways to Ask the System

Expose a small set of business capabilities without allowing the model to bypass authorization or invent an API contract.

Session 3 — Tools, MCP, and Safe Integration Boundaries

Separate a model's request to use a capability from the application's authority to execute it.

- Tool calling mechanics: tool descriptions and JSON schemas, model requests, application execution, tool results, and final synthesis
- Designing safe Java tools: narrow purpose, typed inputs, server-side validation, deterministic error contracts, idempotency, allow-lists, read versus write risk tiers, and confirmation requirements
-

Identity and authorization: propagate authenticated user and tenant context outside the model prompt; enforce entitlements in the service or downstream system—not in tool descriptions

- MCP in practical terms: tools, resources, prompts, transport choices, trusted server selection, capability filtering, and the difference between consuming an MCP server and exposing one
- Guarding against prompt injection and indirect prompt injection: treat model output and retrieved/tool content as untrusted; minimize tool surface, validate every argument, and protect sensitive results
- *Hands-on:* Add controlled account lookup and recommendation-simulation capabilities.
 - Create a simulated legacy Account Service with synthetic customer, account, and product-eligibility records; create two users with deliberately different permissions.
 - Implement allow-listed, read-only tools for account-summary lookup, product-eligibility lookup, and policy search. Use typed input records, server-side validation, authorization checks, correlation IDs, and redacted audit events.
 - Add tool-calling through the selected Java AI integration. Make the available tool set request-specific; do not register privileged or destructive tools as defaults.
 - Build or consume one minimal, trusted MCP capability, then filter the exposed surface to the one tool needed for the lab. Document its server identity, permissions, transport, and failure behavior.
 - Add a recommendation-simulation workflow that clearly labels results as educational and requires explicit human confirmation before any future state-changing extension could execute.
 - Test malformed tool arguments, unauthorized access, fabricated account IDs, repeated requests, prompt-injected policy content, tool timeouts, and trace redaction.
 - **Session checkpoint:** the assistant can answer policy questions with citations and retrieve authorized synthetic account data through controlled tools, while every tool call has validated inputs, identity context, and an inspectable trace.

Day 4 — Orchestrate Bounded Work Under Load

Turn a tool-calling demo into a dependable workflow with explicit limits, concurrency choices, and clear failure states.

Session 4 — Bounded Agentic Workflows and Resilience

Use the model for judgment inside a constrained control plane; keep business invariants, budgets, and recovery behavior in Java.

- From a fixed flow to a bounded agent loop: plan, select an approved tool, inspect evidence, continue only when needed, then answer, clarify, escalate, or stop
- Workflow patterns: sequential enrichment, conditional routing, bounded parallel retrieval, and supervisor-style coordination; when a normal Java workflow is simpler and better
- Virtual threads and external AI I/O: sizing concurrency, protecting downstream dependencies, cancellation, deadlines, backpressure, and avoiding unbounded fan-out
- Resilience: timeouts, retries only where safe, rate-limit handling, circuit breakers, bulkheads, fallback answers, and partial-result disclosure

- Cost and control: maximum steps, allowed tools, token and time budgets, result-size caps, per-request audit trails, and human-in-the-loop gates
- *Hands-on:* Implement a bounded guidance workflow and test its operational edges.
 - Implement a controller that may use only policy retrieval, authorized account lookup, eligibility lookup, and a calculation tool. Define maximum steps, a deadline, a maximum tool-result size, and token/cost budgets.
 - Add explicit outcome states for ANSWERED, NEEDS_CLARIFICATION, ESCALATED, UNSUPPORTED, UNAUTHORIZED, and FAILED_SAFELY; make the API and UI surface them clearly.
 - Run independent retrieval calls in structured, bounded concurrency where they are safe; use virtual threads for blocking model and service calls, with timeouts and cancellation propagated to child work.
 - Add Resilience4j protections for model and account-service dependencies, including rate-limit behavior, circuit-open behavior, and a source-backed fallback where possible.
 - Capture structured traces for each model and tool step while redacting personal and secret data. Expose a safe trace view for debugging and an operator-focused failure summary.
 - Run load and fault-injection scenarios: slow model, account-service outage, repeated tool request, exhausted budget, conflicting sources, and parallel requests from separate users.
 - **Session checkpoint:** a bounded, tested multi-step workflow that produces a useful result or an explicit safe exit within declared time, tool, and cost limits.

Day 5 — Prove, Secure, and Deliver the Service

Turn the capstone into a credible engineering deliverable with evidence of quality, known limits, and an operable deployment path.

Session 5 — Evaluation, Security, and Production Readiness

Separate model, retrieval, tool, and workflow failures; remediate the highest-value gaps and present the system honestly.

- AI-service evaluation: retrieval relevance, citation correctness, answer support, structured-output validity, tool-selection accuracy, task completion, latency, and cost
- Testing strategy: deterministic unit tests around business logic, integration tests with containers and model stubs, controlled live-model checks, contract tests for tools/MCP, and end-to-end scenarios
- Security review: prompt injection, malicious retrieved content, data leakage, over-broad tool access, confused-deputy authorization, sensitive logs, supply-chain risk, and rate/cost abuse
- Operational readiness: health and readiness checks, metrics, traces, alerts, configuration, secrets, database migration and backup strategy, runbooks, and rollback boundaries
- Delivery review: explain architecture choices, show evidence, document remaining risks, and distinguish a demonstration from a supported production workflow
- *Hands-on:* Harden, evaluate, package, and demonstrate the completed assistant.
 -

Expand the evaluation suite with retrieval, account, workflow, refusal, permission, and adversarial cases; set documented thresholds appropriate to the supplied corpus and model configuration.

- Add unit, integration, API, and end-to-end tests. Use Testcontainers for PostgreSQL/pgvector and dependent services where feasible; use deterministic fixtures for tool and model-edge tests.
- Perform a focused threat-model review and implement mitigations for the highest-risk findings, including prompt-injected document content, cross-user account leakage, unsafe tool arguments, and sensitive trace fields.
- Produce a comparison report covering retrieval modes and fixed-flow versus bounded-workflow behavior, with quality, latency, cost, known limitations, and cases that must escalate.
- Package the service with reproducible local setup, database provisioning, migrations, health checks, configuration examples with no secrets, an operations guide, and a short architecture decision record.
- Demonstrate the capstone end to end: a cited policy answer, an authorized account-aware guidance request, an ambiguous request that asks for clarification, an unauthorized request that fails safely, and a dependency failure with a clear user outcome.
- **Session checkpoint:** a tested, containerized Java AI service with evaluated behavior, safe failure states, security controls, and the operational documentation another engineer needs to run and review it.

Capstone — Financial Guidance Assistant

The five days build one coherent application rather than a set of disconnected demos. The assistant supports source-backed explanations of approved financial-product policies and authorized, synthetic account-aware guidance. It does not execute transactions, make regulated financial decisions, or present its output as personalized financial advice.

The capstone contains a Spring Boot API, Java 25 runtime configuration, provider-neutral model adapter, curated policy corpus, PostgreSQL/pgvector retrieval, citation validation, simulated account service, allow-listed tools, a selected MCP integration, a bounded workflow controller, evaluation harness, redacted traces, tests, and containerized local delivery. Each day adds a layer only after its prerequisite is observable and tested.

Day	Capstone increment	Evidence of completion
1	Service and evaluation foundation	Health endpoint, model adapter, safe configuration, supported-scope document, and version-controlled cases
2	Cited RAG	Repeatable ingestion, selectable retrieval modes, source citations, abstention tests, and comparison report

3	Safe tools and MCP	Validated read-only tools, authorization checks, audit trace, filtered MCP capability, and adversarial tool tests
4	Bounded workflow	Step/time/cost/tool limits, resilience behavior, safe exit states, concurrency and fault-injection evidence
5	Production-minded delivery	Evaluation thresholds, security review, automated tests, containerized setup, runbook, and live demonstration

A note on delivery

Each full day balances instructor-led explanation, guided implementation, code review, troubleshooting, and a working checkpoint. Participants arrive with Java 25, Docker, Git, and an IDE installed; the cohort receives a prepared Spring Boot scaffold, synthetic data, policy documents, model-access instructions, and a starting evaluation set so environment setup does not consume the labs.

The course makes its dependencies explicit: a supported use case and secure service baseline precede model integration; source-backed retrieval precedes account-aware tools; validated tools precede a bounded workflow; evaluation and security precede delivery. Framework APIs may change, but the durable outcome is the engineering judgment and evidence trail needed to make an AI capability safe to own.

Ready to build AI services your Java team can test, operate, and trust?

Enquire about a private cohort or a tailored delivery for your engineering team.