JAKARTA EE

Jakarta Agentic AI Specification

Jakarta Agentic AI Project Team, https://projects.eclipse.org/projects/ee4j.agentic-ai

1.0, July 30, 2026: Draft

Table of Contents

License

```
Specification: Jakarta Agentic AI Specification

Version: 1.0

Status: Draft

Release: July 30, 2026
```

Copyright

Eclipse Foundation Specification License - v2.0

This license (the "`License`") governs the use, modification and distribution of a specification generated pursuant to the Eclipse Foundation Specification Process.

By using the file or document that incorporates this License (the "`Document`"), you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy and distribute the contents of the Document, in any medium for any purpose, and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Document that you copy or distribute:

- link or URL to the Document
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "`Copyright (C) [date-of-document] Eclipse Foundation AISBL [URL to this license] This document includes material copied from [title and URL of the Eclipse Foundation specification document].`"

In addition to the license granted above, you are granted the right to create modifications or derivatives of the Document:

- in software that implements the Document, in supporting materials accompanying such software, and in documentation of such software (collectively the "`Works`"), such Works to be made available under terms of your choice PROVIDED that all such Works include a notice of the form: "`This [software/document (choose as applicable)] includes material derived from [title and URL of the Eclipse Foundation specification document].`"; and
- in materials that demonstrate or describe conformance with the Document PROVIDED that all such materials include a notice of the form: "`This material demonstrates conformance with the [title and URL of the Eclipse Foundation specification document].`"

NOTWITHSTANDING THE FOREGOING, the creation or publication of derivative works of the Document or portions of the Document for use as a specification or standard is expressly prohibited.

Disclaimers

THE DOCUMENT IS PROVIDED "AS IS," AND TO THE EXTENT PERMITTED BY APPLICABLE LAW THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION AISBL MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

Chapter 1. Introduction

Jakarta Agentic AI provides vendor-neutral APIs for building, deploying, and running AI agents on Jakarta EE runtimes. The specification standardizes agent lifecycles, workflow composition, and integration with foundational AI technologies such as Large Language Models.

1.1. Scope

- Defines common usage patterns and lifecycles for AI agents on Jakarta EE runtimes.
- Provides a minimal facade for foundational AI capabilities (e.g., LLMs), with pluggable access to existing APIs.
- A mechanism to define dynamic agent workflows using a fluent Java API.
- Integrates with other Jakarta EE APIs (Validation, RESTful Web Services, JSON Binding, Persistence, Data, Transactions, NoSQL, Concurrency, Security, Messaging).
- Aims to utilize Jakarta Config and allows implementations to use MicroProfile Config.
- Implementations may provide OpenTelemetry integration.
- Designed for Jakarta EE runtimes, but potentially usable in Quarkus, Micronaut, and Spring Boot.

The current version of Jakarta Agentic AI targets Java SE 17 or higher and Jakarta EE Platform 10 or higher. See the Bibliography for referenced specifications.

1.2. Motivation

Modern enterprise applications increasingly require automation capabilities that:

- Monitor systems and respond to events
- Make context-aware decisions using AI/ML models
- Execute dynamic workflows based on LLM-generated insights
- Integrate with existing enterprise infrastructure

While AI and LLM technologies have rapidly matured, there is no standard way to build AI agents in Jakarta EE applications. Developers must rely on vendor-specific frameworks or integrate multiple disparate technologies themselves, leading to:

- Lack of portability between Jakarta EE implementations
- Inconsistent programming models
- Difficulty integrating with CDI and other Jakarta EE technologies
- Custom boilerplate code for agent orchestration, even for common patterns

Jakarta Agentic AI addresses these challenges by providing a standard API that feels natural to Java and Jakarta EE developers, similar to how Jakarta Servlet, Jakarta RESTful Web Services, and Jakarta Batch standardized their respective domains.

1.3. Goals

The primary goals of Jakarta Agentic AI are:

Annotation-driven agent development

Define agent workflows using familiar Jakarta EE annotations, marking methods as triggers, decisions, actions, outcomes, etc.

CDI integration

Agents are CDI beans with full support for dependency injection, interceptors, scoping, etc.

Simple LLM abstraction

Provide a lightweight facade for accessing Large Language Models without standardizing the LLM APIs themselves. Vendors plug in different LLM backends while applications enjoy a consistent basic interface.

Workflow orchestration

Enable declarative and programmatic workflows where the runtime manages execution flow between agent lifecycle phases.

Vendor neutrality

Allow implementations across Jakarta EE compatible products and other CDI/Java-based runtimes.

Jakarta EE alignment

Follow familiar Jakarta EE patterns and conventions, ensuring consistency with specifications like Jakarta Persistence, Jakarta RESTful Web Services, and Jakarta Batch.

1.4. Non-goals

The following are explicitly outside the scope of this specification:

Standardizing LLM APIs

Jakarta Agentic AI provides a simple facade for accessing LLMs but does not standardize the underlying LLM APIs. LLM provider APIs vary significantly and evolve rapidly, making standardization impractical at the current moment. The facade provides ready access to the underlying provider APIs through `unwrap()`. A separate Jakarta EE LLM API may be appropriate in the future.

Machine learning frameworks

This specification focuses on agent orchestration and lifecycle management, not ML framework standardization.

Training and fine-tuning

Model training, fine-tuning, and ML model lifecycle management are outside the scope of agent orchestration.

1.5. How this Specification is Organized

The chapters that follow take the reader from concept to normative behavior to reference material:

- Programming Model introduces the agent and workflow model and shows a complete example.
- Annotations and Lifecycle is the normative chapter that defines `@Agent`, `@Trigger`, `@Decision`, `@Action`, `@Outcome`, and `@HandleException`, along with parameter injection, validation, error handling, and workflow termination.
- CDI Integration covers scopes, the `@WorkflowScoped` custom scope, event observation rules, interceptors, and lifecycle callbacks.
- Large Language Model Integration defines the `LargeLanguageModel` facade, `LLMException`, and usage guidance.
- Examples illustrates common agent patterns end to end.

Chapter 2. Programming Model

This chapter presents the Jakarta Agentic AI programming model: what an agent is, how a workflow is composed, and how the pieces fit together. The Annotations and Lifecycle chapter then defines each annotation normatively.

2.1. Agents and Workflows

An agent is a CDI bean annotated with `@Agent` that orchestrates a flexible workflow. Every workflow begins with a trigger and may include any combination of the following phases:

1. **Trigger**: detects an event or condition that initiates the workflow.
2. **Decisions**: analyze workflow state and determine whether and how to proceed.
3. **Actions**: execute operations as part of the workflow.
4. **Outcome**: produces or finalizes the result of the workflow.

Decisions and actions can be intermixed in any sequence, allowing agents to implement simple linear workflows or complex conditional branching.

In addition, an agent may declare `@HandleException` methods to recover from errors that occur during workflow execution.

2.1.1. Workflow Execution

The runtime orchestrates annotated methods using a sequential execution model:

1. The trigger fires.
2. Decisions and actions execute in order, intermixed as defined.
3. Decisions control flow: a positive return continues, negative terminates.
4. The outcome executes once the workflow completes without early termination.
5. Exception handlers execute if exceptions occur in any phase.

The phase rules and per-annotation cardinality are defined normatively in the Annotations and Lifecycle chapter.

2.1.2. Future Direction

A programmatic workflow API is planned for a subsequent release. It will wire annotated methods (the basic building blocks) into dynamic, potentially complex control flows that may be modified at runtime — enabling dynamic decision routing, conditional action execution, and workflows that adapt based on agent state or LLM interactions. An initial workflow could be defined or loaded from storage in a `@PostConstruct` method and persisted in a `@PreDestroy` method.

2.2. An Agent at a Glance

The following agent illustrates the basic model: a CDI bean with a trigger, a decision that consults an LLM, and an action.

```
@Agent
public class MonitoringAgent {

    @Inject
    private LargeLanguageModel llm;

    @Inject
    private EventLogger logger;

    @Trigger
    public void onSystemAlert(Alert alert) {
        // Triggered by an external CDI event
    }

    @Decision
    public boolean shouldEscalate(Alert alert) {
        String analysis = llm.query("Should this alert be escalated?", alert);
        return analysis.toLowerCase().contains("yes");
    }

    @Action
    public void escalateAlert(Alert alert) {
        logger.logEscalation(alert);
    }
}
```

2.3. Jakarta EE Integration

Agents are first-class Jakarta EE citizens. They inject other beans, are managed by CDI, and participate in transactions, validation, persistence, messaging, concurrency, security, and REST integration just like any other Jakarta EE component.

Validation

- Jakarta Validation — constraint annotations on agent life-cycle method parameters; see Jakarta Validation Constraints.

Data access

- Jakarta Persistence — relational databases
- Jakarta NoSQL — non-relational databases
- Jakarta Data — repositories

Transactional operations

- Jakarta Transactions — `@Transactional` agent methods

Data serialization

- Jakarta JSON Binding — used by `LargeLanguageModel` for structured prompt parameters and typed responses; see Large Language Model Integration.

Messaging and integration

- Jakarta Messaging — publish and consume messages (a future trigger source)

REST integration

- Jakarta RESTful Web Services — expose agents as REST resources or consume external services; a REST `POST` endpoint is an anticipated future trigger source

Asynchronous operations

- Jakarta Concurrency — managed executors for asynchronous work within agent logic

```
@Agent
public class DataIntegratedAgent {

    @Inject private EntityManager entityManager;            // Jakarta Persistence
    @Inject private CustomerRepository customerRepo;        // Jakarta Data
    @Inject private Template mongoTemplate;                 // Jakarta NoSQL
    @Inject private ManagedExecutorService executor;        // Jakarta Concurrency

    @Action
    @Transactional                                          // Jakarta Transactions
    public void processCustomerData(
            @Valid @NotNull CustomerId id) {                // Jakarta Validation
        Customer customer = entityManager.find(Customer.class, id);
        List<Order> orders = customerRepo.findOrdersByCustomerId(id);
        mongoTemplate.insert(new AnalyticsEvent(customer, orders));
        executor.submit(() -> generateReport(customer, orders));
    }
}
```

For CDI scoping, observer rules, interceptors, and lifecycle callbacks, see the CDI Integration chapter. For the LLM facade, see the Large Language Model Integration chapter.

Chapter 3. Annotations and Lifecycle

This chapter is the normative reference for the annotations included in the specification, including the lifecycle annotations. Each lifecycle annotation marks a method as participating in a specific phase of the agent workflow. The chapter also defines parameter resolution, validation, error handling, return type handling, and workflow termination.

3.1. Workflow Phases and Annotations

Annotations define agent behavior. They are structured per agent class, generally in order:

- `@Agent` — required, exactly one (class-level)
- `@Trigger` — required, exactly one (may be relaxed in the future)
- `@Decision` — optional, zero or more
- `@Action` — optional, zero or more
- `@Outcome` — optional, zero or one (may be relaxed in the future)
- `@HandleException` — optional, zero or more

3.2. @Agent

The `@Agent` annotation marks a class as an AI agent. Agent classes are CDI beans and follow CDI bean definition rules.

3.2.1. Semantics

- **Required**: every agent class must be annotated with `@Agent`.
- **CDI bean**: the annotated class becomes a CDI managed bean.
- **Discovery**: implementations must discover all `@Agent`-annotated classes during deployment.
- **Default scope**: if no scope is specified, `@WorkflowScoped` is the default. Agents can also be `@ApplicationScoped`. See CDI Integration for scope semantics and bean requirements.
- **Naming**: the agent name defaults to the simple class name with the first character converted to lower case when omitted.

```
@Agent
@ApplicationScoped
public class MonitoringAgent { /* longer-lived agent */ }

@Agent
@WorkflowScoped
public class WorkflowAgent { /* per-workflow lifecycle */ }

@Agent
public class DefaultScopedAgent { /* @WorkflowScoped is implied */ }
```

3.2.2. Workflow Context

Regardless of the scope, the runtime maintains a fresh workflow context for each workflow invocation. For `@WorkflowScoped` agents, that context is tied to the workflow itself. For `@ApplicationScoped` agents, the agent instance may be reused across multiple workflows, but each invocation still has an isolated workflow context that begins with a trigger and typically ends with an outcome.

`LargeLanguageModel` conversational state follows the same boundaries: it is bound to the workflow context and remains isolated per workflow even when the agent is `@ApplicationScoped`. See Conversational State and Threading.

3.3. @Trigger

The `@Trigger` annotation marks the entry point of an agent workflow.

3.3.1. Cardinality

Currently there must be exactly one `@Trigger` method per agent class. This constraint will be relaxed in future releases.

3.3.2. Invocation

In the initial release, triggers are invoked by CDI events. A trigger method may use `@Observes` on the event parameter to explicitly declare the observed type. However, explicitly using `@Observes` is not required. There must be exactly one event parameter. The triggering event becomes part of the workflow state and is available for parameter resolution in later phases.

In future releases, additional trigger sources are anticipated, such as Jakarta Messaging, manual invocation through a lifecycle API, or REST `POST` requests.

CDI observer behavior depends on agent scope. See the Event Observation Rules in the CDI Integration chapter.

3.3.3. Parameters

Trigger methods follow the standard parameter resolution rules. The required triggering event is the most common parameter.

3.3.4. Return Type

Trigger methods support two return patterns:

- **Void** — initialization with side effects only; no data is passed forward.
- **Domain object** — the returned object becomes part of workflow state and is available for injection into later phases.

```
@Agent
```

```java
public class EventDrivenAgent {

    @Trigger
    public void onCustomEvent(CustomEvent event) {
        logger.info("Workflow triggered for event: " + event.getId());
    }
}

@Agent
public class AnalyzingAgent {

    @Trigger
    public EventAnalysis analyzeEvent(CustomEvent event, LargeLanguageModel llm) {
        String analysis = llm.query("Classify this event", event);
        EventAnalysis result = new EventAnalysis();
        result.setEvent(event);
        result.setClassification(analysis);
        return result;
    }
}
```

3.4. Parameter Resolution

All workflow methods (annotated with `@Trigger`, `@Decision`, `@Action`, `@Outcome`, and `@HandleException`) receive their parameters through automatic resolution based on type and the current workflow state. Each annotation uses this single set of rules.

3.4.1. Resolution Order

When a workflow method is invoked, the runtime resolves each parameter in the following order:

1. **Trigger event** — if the parameter type matches the triggering event.
2. **Previous phase results** — if the parameter type matches an object returned by an earlier step in the current workflow. When multiple steps have returned objects of the same type, the most recently returned object is used.
3. `LargeLanguageModel` — if the parameter type is `LargeLanguageModel`.
4. **Exception** — for `@HandleException` methods only, the thrown exception (or any supertype thereof) is the required first parameter.
5. **CDI injection** — if the parameter is a CDI bean.

3.4.2. Jakarta Validation Constraints

Parameters may declare Jakarta Validation constraints (`@Valid`, `@NotNull`, `@NotEmpty`, `@Size`, etc). Validation occurs before parameter injection and workflow method invocation. If validation fails, a `ConstraintViolationException` is raised; it can be caught by an `@HandleException` method or propagated to the container.

`ConstraintViolationException` is a runtime exception, so explicit handling is not required. Common scenarios include parameter constraint failures, custom constraint violations, and validation of complex domain objects passed between phases.

3.5. @Decision

The `@Decision` annotation marks a method that decides whether and how the workflow should proceed.

3.5.1. Cardinality

An agent may define zero or more `@Decision` methods.

3.5.2. Parameters

Decision methods follow the standard parameter resolution rules.

3.5.3. Return Type

Decisions support three return patterns:

Boolean

- `true` — proceed with the workflow.
- `false` — terminate the workflow.

`Result` **record**

Standardizes a decision outcome with a boolean success flag and optional details object:

```
public record Result(boolean success, Object details) {}
```

- `success == true` proceeds; `success == false` terminates.
- `details` (if non-`null`) becomes part of workflow state and is available for injection by its runtime type into later steps.

Domain object

- Non-`null` — proceed; the returned object becomes part of workflow state.
- `null` — terminate the workflow.

```
@Agent
public class DecisionExamples {

    @Inject
    private LargeLanguageModel llm;

    // Boolean
    @Decision
    public boolean shouldProcess(Event event) {
```

```
        String analysis = llm.query("Should this event be processed?", event);
        return analysis.toLowerCase().contains("yes");
    }

    // Result record
    @Decision
    public Result evaluateEvent(Event event) {
        String analysis = llm.query("Analyze this event", event);
        boolean proceed = requiresAction(analysis);
        AnalysisDetails details = proceed ? parseDetails(analysis) : null;
        return new Result(proceed, details);
    }

    // Domain object
    @Decision
    public AnalysisResult analyzeEvent(Event event) {
        String analysis = llm.query("Analyze this event", event);
        return requiresAction(analysis) ? new AnalysisResult(analysis) : null;
    }
}
```

3.6. @Action

The `@Action` annotation marks a method that performs work as part of the workflow.

3.6.1. Cardinality

An agent may define zero or more `@Action` methods. All actions complete before the outcome phase runs.

3.6.2. Parameters

Action methods follow the standard parameter resolution rules.

3.6.3. Return Type

Actions support two return patterns:

- **Void** — side effects only (e.g., sending alerts, updating databases).
- **Domain object** — the returned object becomes part of workflow state and is available for injection into later phases.

```
@Agent
public class ActionExamples {

    // Side effects
    @Action
    public void handleFraud(Fraud fraud, BankTransaction transaction) {
```

```
        if (fraud.isSerious()) {
            alertBankSecurity(fraud);
        }
        Customer customer = getCustomer(transaction);
        alertCustomer(fraud, transaction, customer);
    }

    // Pass result forward
    @Action
    public FraudReport processFraudCase(Fraud fraud, BankTransaction transaction) {
        FraudReport report = new FraudReport();
        report.setFraudType(fraud.getType());
        report.setTransaction(transaction);
        report.setTimestamp(System.currentTimeMillis());
        persistReport(report);
        return report;
    }

    @Outcome
    public void recordOutcome(FraudReport report) {
        auditLog("Fraud case processed: " + report.getId());
    }
}
```

You can apply or use Jakarta Transactions in lifecycle methods including Actions:

```
@Agent
public class TransactionalAgent {

    @Action
    @Transactional
    public void updateRecords(UpdatePlan plan) {
        database.updateRecords(plan.getRecords());
    }
}
```

3.7. Execution Order

The execution order of `@Action` and `@Decision` methods can be intermixed as needed. The runtime determines their order using the following rules, in decreasing precedence:

1. **`@Priority` on the method** — `jakarta.annotation.Priority` value is the sort key; lower values execute first.
2. **`order` attribute on the `@Action` and `@Decision` annotations** — used as the sort key when `@Priority` is absent; lower values execute first.
3. **Source declaration order** — fallback when no `@Action` or `@Decision` method in the agent declares `@Priority` or an explicit `order`.

Java SE does not guarantee that reflection returns methods in source declaration order. However, all major implementations do so in practice. Portable applications requiring strict ordering must use `@Priority` or `order`.

3.7.1. Consistency Requirement

If any `@Action` or `@Decision` method in an agent declares an explicit `order` or `@Priority`, every `@Action` and `@Decision` method in that agent must do the same. Mixing explicitly ordered and unordered methods is a deployment error.

3.8. @Outcome

The `@Outcome` annotation marks a method that finalizes the workflow. It produces the final workflow result, runs completion logic, and marks the end of successful workflow execution.

3.8.1. Cardinality

Currently, an agent may declare zero or one `@Outcome` method. This constraint may be relaxed in a future release.

3.8.2. Parameters

Outcome methods follow the standard parameter resolution rules. Any object produced by an earlier phase is eligible for injection.

3.8.3. Return Type

Outcome methods must return `void`. The outcome phase is intended for finalization and side effects, not for producing further data. Future releases may relax this for downstream system integration.

3.9. @HandleException

The `@HandleException` annotation marks a method that handles exceptions occurring during workflow execution. For handler matching and dispatch semantics, see Error Handling.

3.9.1. Cardinality

An agent may define zero or more `@HandleException` methods.

3.9.2. Parameters

Exception handler methods follow the standard parameter resolution rules. The thrown exception (or any supertype) is the required first parameter.

3.9.3. Return Type

Exception handler methods must return `void`. They are intended for error recovery, logging, and

cleanup.

3.9.4. Examples

```
@Agent
public class ExceptionHandlingExamples {

    // Recoverable - returns normally, workflow continues
    @HandleException
    public void handleRecoverable(IOException ex, BankTransaction transaction) {
        logger.warn("I/O error, retrying: " + transaction.getId(), ex);
        retryQueue.add(transaction);
    }

    // Fatal - re-throws, workflow stops
    @HandleException
    public void handleFatal(SecurityException ex, BankTransaction transaction) {
        logger.error("Security violation", ex);
        auditService.logSecurityBreach(transaction);
        throw ex;
    }

    // Conditional recovery
    @HandleException
    public void handleWithFallback(Exception ex, BankTransaction transaction) {
        if (isRecoverable(ex)) {
            performRecovery(transaction);
        } else {
            throw new WorkflowFailureException("Unrecoverable", ex);
        }
    }
}
```

3.10. Error Handling

When an exception occurs in any workflow phase (`@Trigger`, `@Decision`, `@Action`, or `@Outcome`), the runtime searches for a matching `@HandleException` method. The most specific exception type match wins, following Java's exception hierarchy.

- If the handler returns normally, the workflow continues with the next step.
- If the handler re-throws or throws a new exception, the workflow stops and the exception propagates to the container.
- If no handler is defined, the exception propagates to the container.
- Exceptions thrown from handler methods are not handled recursively.

```
@Agent
public class ResilientAgent {
```

```
    @Trigger
    public void onEvent(Event event) {
        if (event == null) {
            throw new IllegalArgumentException("Event cannot be null");
        }
    }

    @Decision
    public boolean analyze(Event event) {
        return analyzeEvent(event); // may throw IOException
    }

    @HandleException
    public void handleIOException(IOException e, Event event) {
        logger.warn("I/O error, will retry", e);
        scheduleRetry(event);
    }

    @HandleException
    public void handleGenericException(Exception e, Event event) {
        logger.error("Unexpected error", e);
        logFailure(event, e);
    }
}
```

3.11. Workflow Termination

A workflow terminates when any of the following occurs:

1. **Normal completion** — the outcome phase completes successfully (or, if no outcome is defined, the last phase completes).
2. **Decision stop** — a decision phase returns a negative result.
3. **Unhandled exception** — an exception is raised and no matching handler is defined.
4. **Handler exception** — an exception handler itself throws.

Any runtime-managed state associated with the current workflow is discarded once the workflow terminates.

3.11.1. One Phase per Method

Each lifecycle annotation marks a method as participating in a single workflow phase. A method must not declare more than one of `@Trigger`, `@Decision`, `@Action`, `@Outcome`, or `@HandleException`; combining two or more of them on the same method is a deployment error.

3.11.2. Inherited Methods

Phase methods declared on a superclass are inherited by an agent subclass and participate in the

workflow as if declared on the subclass itself. A method that overrides an inherited phase method participates only when the overriding method is itself annotated; following Java semantics, phase annotations are not implicitly carried onto an unannotated override.

3.12. Workflow Patterns

Agents are not constrained to a rigid template — only `@Trigger` is required. The following patterns illustrate some common shapes.

3.12.1. Trigger + Action

```
@Agent
public class SimpleNotificationAgent {

    @Trigger
    public void onAlert(SystemAlert alert) { }

    @Action
    public void sendNotification(SystemAlert alert) {
        emailService.notifyAdmins(alert);
    }
}
```

3.12.2. Sequential Phases

```
@Agent
public class DataProcessingAgent {

    @Trigger
    public void onDataReceived(DataBatch data) { }

    @Action public void validateData(DataBatch data)   { validator.check(data); }
    @Action public void transformData(DataBatch data)  { transformer.process(data); }
    @Outcome public void persistData(DataBatch data)    { database.save(data); }
}
```

3.12.3. Intermixed Decisions and Actions

```
@Agent
public class OrderAgent {

    @Trigger
    public void onOrder(Order order) { }

    @Decision
    public boolean isLegitimate(Order order, LargeLanguageModel llm) {
```

```
        return llm.query("Does this order appear legitimate?", order).contains("yes");
    }

    @Decision
    public boolean inStock(Order order) {
        return inventory.available(order);
    }

    @Action
    public void reserveStock(Order order) {
        inventory.reserve(order);
    }

    @Decision
    public boolean paymentClears(Order order) {
        return payment.charge(order);
    }

    @Action
    public void shipOrder(Order order) {
        shipping.dispatch(order);
    }

    @Outcome
    public void confirmOrder(Order order) {
        notifier.confirm(order);
    }
}
```

Chapter 4. CDI Integration

This chapter describes how Jakarta Agentic AI integrates with Jakarta Contexts and Dependency Injection (CDI), including agent scoping, the `@WorkflowScoped` custom scope, event observation rules, interceptors, and lifecycle callbacks.

4.1. Agents as CDI Beans

Every agent is a CDI managed bean. This provides:

- **Dependency injection** of other beans, Jakarta EE resources, and services
- **Interceptors** for cross-cutting concerns
- **CDI events** for production and observation
- **Lifecycle callbacks** (`@PostConstruct`, `@PreDestroy`)
- **Flexible scoping**

4.1.1. Bean Requirements

An agent class:

- Must be annotated with `@Agent`
- Must satisfy CDI managed bean requirements

4.2. Agent Scopes

An agent uses one of two scopes:

- `@WorkflowScoped` (default) — a new instance per workflow context.
- `@ApplicationScoped` — a single shared instance across workflows.

If no scope is declared, `@WorkflowScoped` applies.

4.2.1. @WorkflowScoped

`@WorkflowScoped` is a custom CDI scope that ties the agent's lifecycle to the workflow context. It is a normal scope (`@NormalScope`).

When an agent is `@WorkflowScoped`:

- A new instance is created for each workflow context.
- The instance is active from trigger invocation through workflow completion.
- The instance is destroyed after the workflow completes or fails.
- Each concurrent workflow has its own instance.

4.2.2. @ApplicationScoped

When an agent is `@ApplicationScoped`, a single instance is shared across all workflows. Each invocation still runs in an isolated workflow context. Because a single instance may handle multiple concurrent workflow invocations, `@ApplicationScoped` agents must be thread-safe; the agent itself must either be stateless or guard any shared state with appropriate concurrency controls.

4.2.3. Threading and Concurrency

- **Sequential phases** — within a single workflow context, annotated methods (`@Trigger`, `@Decision`, `@Action`, `@Outcome`, `@HandleException`) execute sequentially (see the Execution Order rules in Annotations and Lifecycle).
- `@WorkflowScoped` **agents** — a workflow context typically executes on a single thread; the agent instance is therefore not exposed to concurrent invocation by the runtime.
- `@ApplicationScoped` **agents** — the same instance may serve concurrent workflows; the agent must be thread-safe (see above).
- `LargeLanguageModel` — implementations must be thread-safe within a single workflow context and must not leak conversational state across concurrent workflow contexts. See Conversational State and Threading for the full contract.

4.3. Event Observation Rules

Agents may produce CDI events freely. Event **observation** depends on agent scope:

- `@WorkflowScoped` **agents** — may observe CDI events only through their `@Trigger` method. General observer methods (`@Observes` without `@Trigger`) are not supported.
- `@ApplicationScoped` **agents** — may use both `@Trigger` methods and general CDI observer methods.

A trigger method may optionally use `@Observes` on the event parameter to explicitly declare the observed type.

4.3.1. Producing Events

```
@Agent
public class EventProducer {

    @Inject
    private Event<WorkflowCompleted> completionEvent;

    @Outcome
    public void publishCompletion(Result result) {
        completionEvent.fire(new WorkflowCompleted(result));
    }
}
```

4.3.2. Observing Events

```
@Agent
@ApplicationScoped
public class EventObserver {

    @Trigger
    public void onCustomEvent(@Observes CustomEvent event) {
        processEvent(event);
    }
}
```

4.4. Interceptors

Standard CDI interceptors apply to agent methods. Jakarta Transactions is a common case:

```
@Agent
public class TransactionalAgent {

    @Inject
    private EntityManager entityManager;

    @Action
    @Transactional
    public void updateDatabase(Data data) {
        entityManager.persist(data);
    }
}
```

4.5. Lifecycle Callbacks

Agents support standard CDI lifecycle callbacks:

```
@Agent
@WorkflowScoped
public class LifecycleAwareAgent {

    private static final Logger logger =
            Logger.getLogger(LifecycleAwareAgent.class.getName());

    @Inject
    private ResourcePool resources;

    @PostConstruct
    public void initialize() {
        logger.info("Agent initialized");
```

```
        resources.allocate();
    }

    @PreDestroy
    public void cleanup() {
        logger.info("Agent destroying");
        resources.release();
    }
}
```

4.6. LargeLanguageModel as a CDI Bean

The `LargeLanguageModel` interface is provided as a CDI bean by the implementation and is injected into agents like any other bean. Its conversational state lifecycle aligns with workflow context boundaries; see Conversational State and Threading for the full contract.

Chapter 5. Large Language Model Integration

This chapter defines the `LargeLanguageModel` facade, the `LLMException` runtime exception, and provides usage guidance grounded in the contract.

5.1. Why a Facade

Jakarta Agentic AI uses a facade pattern for LLM access. The `LargeLanguageModel` interface provides a small, consistent portability layer while allowing implementations to integrate any LLM backend.

Abstraction

Applications depend on the facade, not on a specific LLM implementation.

Vendor flexibility

Implementations can integrate any LLM provider or underlying SDK.

Unwrapping

The `unwrap()` method, defined in the `LargeLanguageModel` interface, allows access to vendor-specific APIs when needed, following the `EntityManager.unwrap()` pattern from Jakarta Persistence.

Evolution

LLM technologies evolve rapidly; the facade insulates applications from those changes. The facade could be evolved into a separate specification in the future.

5.2. LargeLanguageModel Interface

The `jakarta.ai.agent.LargeLanguageModel` interface defines a focused set of operations:

- `query(...)` — an overloaded method that sends a prompt to the model. Overloads cover two independent choices: whether to receive the raw `String` response or a value deserialized to a caller-specified type, and whether to supply positional parameters substituted into `{}` placeholders in the prompt (or, when no placeholder is present, attached as a single piece of structured context). Jakarta JSON Binding is used to serialize prompt parameters and deserialize typed responses, letting domain objects flow in and out without application-specific serialization code.
- `unwrap(...)` — return the underlying vendor-specific implementation so callers can reach provider APIs or features not exposed by the facade.

See the API Javadoc for the full method signatures, parameter semantics, and per-overload behavioral requirements.

Implementations are provided as CDI beans and injected into agents like any other bean; see LargeLanguageModel as a CDI Bean for the injection contract.

5.2.1. Conversational State and Threading

Implementations of the `LargeLanguageModel` interface maintain conversational state for the current workflow context across calls to the `query(...)` method, so that successive calls within a workflow can build on prior exchanges. The lifecycle of that state follows the workflow context boundaries defined in CDI Integration.

- **`@WorkflowScoped` agents** — conversational state is bound to the workflow context and must end when the workflow context ends.
- **`@ApplicationScoped` agents** — an instance of the `LargeLanguageModel` interface may be shared across concurrent workflows, but conversational state must remain isolated per workflow context and must not leak across concurrent invocations.
- **Thread safety** — implementations must be thread-safe within a single workflow context.

5.3. Exceptions

All `query(...)` methods may throw:

- `IllegalArgumentException` — for null or invalid parameters, placeholder or parameter-count mismatches, or conversion failures (input parameters that cannot be converted).
- `LLMException` — for LLM service errors or conversion failures (responses that cannot be converted to the requested type). See below for details.

`unwrap()` throws an `IllegalArgumentException` if the underlying implementation cannot be unwrapped to the requested type.

5.3.1. LLMException

The `LLMException` class is a runtime exception thrown when LLM operations fail. As a runtime exception, it does not require explicit handling but may be caught by methods annotated with the `@HandleException` annotation in agent workflows. Common failure scenarios include:

- Communication failures with the LLM service
- Invalid or malformed responses from the LLM (truncated output, error payloads, protocol-level failures)
- Rate limiting or quota exceeded
- Model unavailability or timeout
- Response deserialization failures when a typed result is requested and the LLM response cannot be converted to the expected Java type

5.4. Usage Guidance

5.4.1. Prompt Construction

When using the varargs versions of the `query(...)` overloaded methods, use the exact token `{}` for

positional substitution.

- When a prompt contains one or more {} placeholders, the number of supplied parameters must exactly match the number of placeholders.
- When a prompt contains no placeholder, at most one supplied parameter may be provided as structured context.

```
@Decision
public boolean goodPrompt(Issue issue) {
    String response = llm.query(
        """
        Analyze this issue and respond with YES or NO:
        Title: {}
        Description: {}

        Should this issue be escalated to senior engineers?
        Respond with only: YES or NO
        """,
        issue.getTitle(),
        issue.getDescription()
    );
    return response.trim().equalsIgnoreCase("YES");
}
```

```
@Decision
public String classifyEvent(MyEvent event) {
    return llm.query("Classify this event", event); // structured context
}
```

5.4.2. Typed Responses

Prefer typed responses over manual string parsing. Implementations deserialize the LLM response from JSON to the requested type via Jakarta JSON Binding.

```
@Decision
public FraudAnalysis analyzeTransaction(BankTransaction transaction) {
    String prompt = """
        Analyze this transaction for fraud.
        Respond with JSON matching this structure:
        {
          "isFraudulent": true/false,
          "confidence": 0.0-1.0,
          "reason": "explanation",
          "riskLevel": "LOW|MEDIUM|HIGH|CRITICAL"
        }
        """;
    try {
```

```
            return llm.query(prompt, FraudAnalysis.class, transaction);
        } catch (LLMException e) {
            logger.warn("Failed to deserialize fraud analysis", e);
            return FraudAnalysis.unknown();
        }
    }

    public record FraudAnalysis(
        boolean isFraudulent,
        double confidence,
        String reason,
        RiskLevel riskLevel
    ) {
        public static FraudAnalysis unknown() {
            return new FraudAnalysis(false, 0.0, "Analysis failed", RiskLevel.UNKNOWN);
        }
    }
```

When string parsing is unavoidable, parse defensively:

```
@Decision
public Priority parsePriority(Issue issue) {
    String response = llm.query(
        "Assess priority (CRITICAL, HIGH, MEDIUM, LOW) for this issue",
        issue
    );
    String normalized = response.trim().toUpperCase();
    for (Priority p : Priority.values()) {
        if (normalized.contains(p.name())) {
            return p;
        }
    }
    logger.warn("Could not parse priority from: " + response);
    return Priority.MEDIUM;
}
```

5.4.3. Error Handling

Handle LLM failures inline with try-catch, or workflow-wide with `@HandleException`. See Error Handling in Annotations and Lifecycle for handler matching rules and the behavior when a handler returns normally vs. re-throws.

```
@Decision
public boolean robustEvaluation(Task task) {
  try {
    String response = llm.query("Evaluate this task", task);
    return parseResponse(response);
```

```
  } catch (LLMException e) {
    logger.warn("LLM service error, using fallback", e);
    return task.getPriority() == Priority.HIGH;
  } catch (IllegalArgumentException e) {
    logger.error("Invalid parameter to LLM", e);
    return false;
  }
}
```

```
@HandleException
public void handleLLMError(LLMException e, Task task) {
  logger.error("LLM failed for task: " + task.getId(), e);
  task.setStatus(TaskStatus.NEEDS_MANUAL_REVIEW);
  entityManager.merge(task);
  // Workflow continues after this method returns
}
```

5.4.4. Vendor-Specific Access

Use the `unwrap()` method to reach vendor-specific features when portability is not required:

```
@Agent
public class AdvancedAgent {

    @Inject
    private LargeLanguageModel llm;

    @Decision
    public AdvancedResult useVendorFeatures(Data data) {
        String basicResult = llm.query("Basic analysis", data);
        try {
            SomeLlmApi vendorApi = llm.unwrap(SomeLlmApi.class);
            return vendorApi.advancedAnalysis(data);
        } catch (IllegalArgumentException e) {
            return parseBasicResult(basicResult);
        }
    }
}
```

5.5. Provider Configuration

5.5.1. Initial Release

Implementations are free to support whichever LLM libraries and APIs they choose (e.g., Spring AI, LangChain4j, or vendor SDKs). Configuration of the LLM provider, if supported, is implementation-specific (properties, environment variables, or other mechanisms).

5.5.2. Future Releases

Future releases may introduce standardized configuration for provider selection and common parameters through Jakarta Config — analogous to how Jakarta Persistence handles multiple database providers — covering shared settings such as endpoints, authentication, model selection, temperature, and maximum output tokens.

For details on the current implementation, refer to your Jakarta Agentic AI implementation's documentation.

Chapter 6. Examples

This chapter provides informative end-to-end examples demonstrating common agent patterns and use cases. The examples focus on composition and integration style rather than production-ready completeness.

6.1. Simple Fraud Detection Agent

A basic agent that detects fraudulent transactions:

```
@Agent(description="Detects bank fraud transactions")
public class FraudDetectionAgent {

    @Inject
    private LargeLanguageModel model;

    @Inject
    private EntityManager entityManager;

    @Inject
    private AlertService alertService;

    @Trigger
    public void processTransaction(@Valid BankTransaction transaction) {
        logger.info("Processing transaction: " + transaction.getId());
    }

    @Decision
    public Result checkFraud(BankTransaction transaction) {
        FraudAssessment assessment = model.query(
            """
            Analyze this transaction for fraud:
            Amount: {}
            Merchant: {}
            Location: {}

            Respond with JSON: {"isFraud": true/false, "confidence": 0.0-1.0, "reason": "..."}
            """,
            FraudAssessment.class,
            transaction.getAmount(),
            transaction.getMerchant(),
            transaction.getLocation()
        );

        Fraud details = assessment.isFraud()
            ? new Fraud(assessment.confidence(), assessment.reason())
            : null;
        return new Result(assessment.isFraud(), details);
    }

    @Action
    public void alertBankSecurity(Fraud fraud) {
```

```
        if (fraud.isSerious()) {
            alertService.alertBankSecurity(fraud);
        }
    }

    @Action
    public void alertCustomer(Fraud fraud, BankTransaction transaction) {
        Customer customer = getCustomer(transaction);
        alertService.alertCustomer(fraud, transaction, customer);
    }

    @Outcome
    @Transactional
    public void finalizeTransaction(BankTransaction transaction, Fraud fraud) {
        transaction.setStatus(TransactionStatus.SUSPECT);
        entityManager.merge(transaction);
        logger.info("Transaction marked as suspect: " + transaction.getId());
    }

    @HandleException
    @Transactional
    public void handleModelUnavailable(LLMException e, BankTransaction transaction) {
        logger.warn("Fraud model unavailable for transaction: " + transaction.getId(), e);
        // Compensate safely: route to manual review when model calls fail
        transaction.setStatus(TransactionStatus.NEEDS_REVIEW);
        entityManager.merge(transaction);
    }
}
```

6.2. Documentation Generation Agent

An agent that monitors pull requests and generates documentation:

```
@Agent
@ApplicationScoped
public class DocumentationAgent {

    @Inject
    private LargeLanguageModel llm;

    @Inject
    private GitService gitService;

    @Trigger
    public void onPullRequest(PullRequestEvent prEvent) {
        logger.info("Processing PR #" + prEvent.getNumber());
    }

    @Decision
    public DocumentationPlan shouldGenerateDocs(PullRequestEvent prEvent) {
        PullRequest pr = prEvent.getPullRequest();
```

```
        DocumentationPlan plan = llm.query(
            """
            Analyze this pull request and determine if documentation is needed.

            Title: {}
            Description: {}
            Changed Files: {}

            Respond with JSON:
            {"needsDocumentation": true/false, "suggestedFiles": ["file1.md"]}
            """,
            DocumentationPlan.class,
            pr.getTitle(),
            pr.getDescription(),
            String.join(", ", pr.getChangedFiles())
        );

        if (!plan.needsDocumentation()) {
            logger.info("No documentation needed for PR #" + pr.getNumber());
            return null; // Stop workflow
        }

        return plan;
    }

    @Action
    public DocumentationFiles generateDocumentation(
        PullRequestEvent prEvent,
        DocumentationPlan plan
    ) {
        PullRequest pr = prEvent.getPullRequest();
        DocumentationFiles files = new DocumentationFiles();

        for (String filename : plan.getSuggestedFiles()) {
            String content = llm.query(
                "Generate documentation for: {}. Diff: {}",
                filename,
                pr.getDiff()
            );
            files.addFile(filename, content);
        }

        return files;
    }

    @Outcome
    public void createDocumentationPR(
        PullRequestEvent originalPr,
        DocumentationFiles files
    ) {
```

```
        String branchName = "docs/pr-" + originalPr.getNumber();
        gitService.createBranch(branchName);
        gitService.commitFiles(branchName, files);

        PullRequest docPr = gitService.createPullRequest(
            branchName,
            "main",
            "Documentation for PR #" + originalPr.getNumber()
        );

        logger.info("Created documentation PR #" + docPr.getNumber());
    }

    @HandleException
    public void handleGenerationFailure(LLMException e, PullRequestEvent prEvent) {
        logger.warn("Documentation generation unavailable for PR #" +
            prEvent.getNumber(), e);

        gitService.addComment(
            prEvent.getNumber(),
            "Automatic documentation is temporarily unavailable. " +
            "Please update docs manually for this PR."
        );
    }
}
```

6.3. Customer Support Agent

A support agent that handles customer inquiries:

```
@Agent
public class CustomerSupportAgent {

    @Inject
    private LargeLanguageModel llm;

    @Inject
    private KnowledgeBaseService knowledgeBase;

    @Inject
    private TicketService ticketService;

    @Inject
    private CustomerService customerService;

    @Inject
    private MessagingService messagingService;

    private Customer customer;
```

```
    @Trigger
    public void onCustomerMessage(CustomerMessage message) {
        customer = customerService.getCustomer(message.getCustomerId());
    }

    @Action
    public SupportResponse generateResponse(CustomerMessage message) {
        List<KnowledgeArticle> articles = knowledgeBase.search(message.getText(), 5);

        String llmResponse = llm.query(
            """
            You are a helpful customer support agent.

            Customer: {}
            Relevant Knowledge Base Articles:
            {}

            Customer's Latest Message: {}

            Provide a helpful response. Respond with JSON:
            {"response": "...", "canResolve": true/false, "needsEscalation": true/false}
            """,
            customer.getName(),
            articles.stream().map(a -> a.getSummary()).collect(Collectors.joining("\n")),
            message.getText()
        );
        return parseResponse(llmResponse);
    }

    @Action
    public void sendResponse(SupportResponse response) {
        messagingService.sendToCustomer(customer.getId(), response.getResponse());
    }

    @Outcome
    public void finalizeInteraction(SupportResponse response) {
        if (response.needsEscalation()) {
            ticketService.escalate(customer.getId(), "Agent unable to resolve");
            messagingService.sendToCustomer(
                customer.getId(),
                "I've escalated your issue to a specialist."
            );
        }
    }

    @HandleException
    public void handleModelFailure(LLMException e, CustomerMessage message) {
        logger.warn("Support response generation unavailable", e);
        ticketService.escalate(customer.getId(), "Automated response unavailable");
    }
}
```

6.4. Parameter Validation and Error Recovery Agent

An agent that validates input and recovers gracefully from validation and delivery failures:

```
@Agent
public class OnboardingAgent {

    @Inject
    private EmailService emailService;

    // Constraint violations short-circuit the workflow and are recovered by
    // @HandleException below.
    @Trigger
    public void onInvitation(@Valid @NotNull UserInvitation invitation) {
        logger.info("Processing invitation for: " + invitation.getEmail());
    }

    @Decision
    public boolean isEmailActive(UserInvitation invitation) {
        // Format was validated by @Email on UserInvitation; here we confirm
        // the address is actually reachable.
        return emailService.isActiveEmail(invitation.getEmail());
    }

    @Action
    public void sendWelcomeEmail(UserInvitation invitation) {
        User invitee = invitation.getInvitee();
        emailService.send(
            invitation.getEmail(),
            "Welcome " + invitee.getFirstName() + "!"
        );
    }

    @Outcome
    public void markAccepted(UserInvitation invitation) {
        invitation.setStatus(InvitationStatus.ACCEPTED);
    }

    @HandleException
    public void onInvalidInvitation(ConstraintViolationException e,
                                    UserInvitation invitation) {
        for (var v : e.getConstraintViolations()) {
            logger.warn("Invalid: " + v.getPropertyPath() + " - " + v.getMessage());
        }
        invitation.setStatus(InvitationStatus.INVALID);
    }

    // Compensating recovery when the mail service cannot deliver.
    @HandleException
    public void onDeliveryFailure(EmailDeliveryException e,
```

```
                                    UserInvitation invitation) {
            logger.error("Unable to deliver welcome email", e);
            invitation.setStatus(InvitationStatus.DELIVERY_FAILED);
        }
    }
```

6.5. Key Patterns Demonstrated

These examples demonstrate:

1. **Different Scopes**: default `@WorkflowScoped`, explicit `@ApplicationScoped`
2. **LLM Integration**: Using `LargeLanguageModel` for intelligent decisions
3. **Conditional Workflows**: Decisions that stop or continue workflows
4. **Multiple Actions**: Sequential action execution
5. **Error Handling**: Graceful degradation and fallback strategies
6. **CDI Integration**: Dependency injection, transactions
7. **Context Management**: Maintaining state across workflow phases
8. **Parameter Validation**: Jakarta Validation constraints with failure recovery
9. **Real-world Use Cases**: Fraud detection, documentation generation, customer support

Appendix A: Revision History

A.1. 1.0

Initial release of Jakarta Agentic AI.

Bibliography

- **Java SE 17**, https://docs.oracle.com/en/java/javase/17/
- **Jakarta EE Platform 10**, https://jakarta.ee/specifications/platform/10/
- **Jakarta Contexts and Dependency Injection 4.0**, https://jakarta.ee/specifications/cdi/4.0/
- **Jakarta Validation 3.0**, https://jakarta.ee/specifications/bean-validation/3.0/
- **Jakarta JSON Binding 3.0**, https://jakarta.ee/specifications/jsonb/3.0/