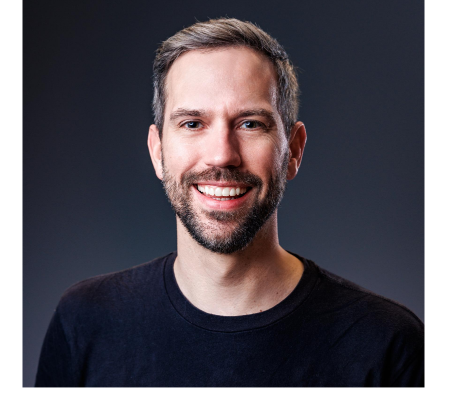
#KubeCon #CloudNativeCon

Building and Orchestrating Production-ready Agentic AI Systems

Kevin Dubois and Daniel Oh

Kevin Dubois



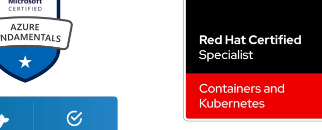
★ Sr. Principal Developer Advocate at IBM

★ Java Champion

★ Technical Lead, CNCF DevEx TAG

★ Belgian 🇧🇪 but based in the Swiss Alps 🏔️ 🇨🇭

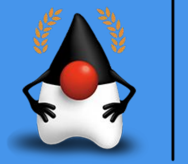
★ 🗣️ English, Dutch, French, Italian



Let's connect!

- @kevindubois
- @kevindubois.com
- youtube.com/@thekevindubois
- github.com/kdubois

Daniel Oh



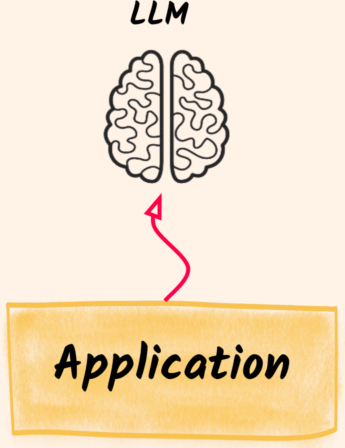
- Developer Advocate at IBM (Red Hat)
  - Cloud Native Runtimes
  - AI, GitOps, Security, and Serverless
- JAVA Champion
- CNCF Ambassador & TAG DevEx co-chair
- Microsoft MVP - Java & Developer Tools
- Keynote Speaker & Published Author
- Living in Boston, MA USA

Microsoft® MVP Most Valuable Professional

@danieloh30

It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

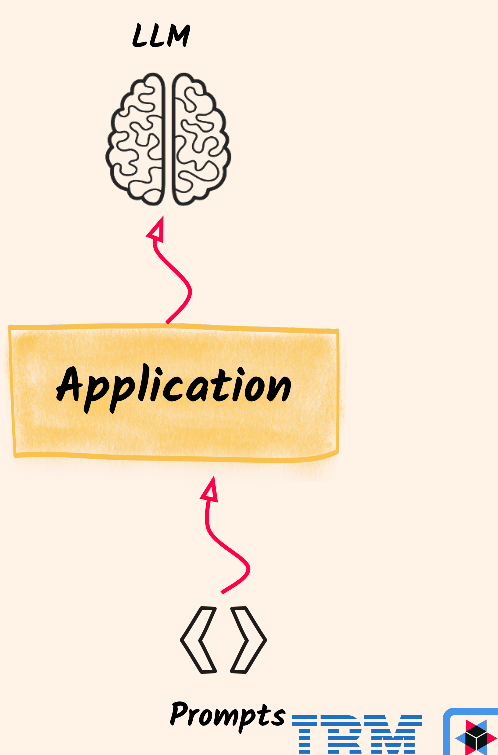


It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways

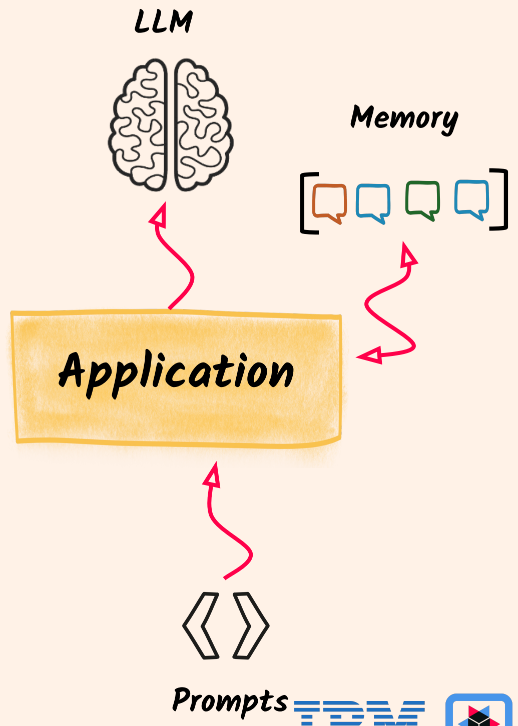


It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways
- A **chat memory** to "remember" previous interactions and make the AI service conversational

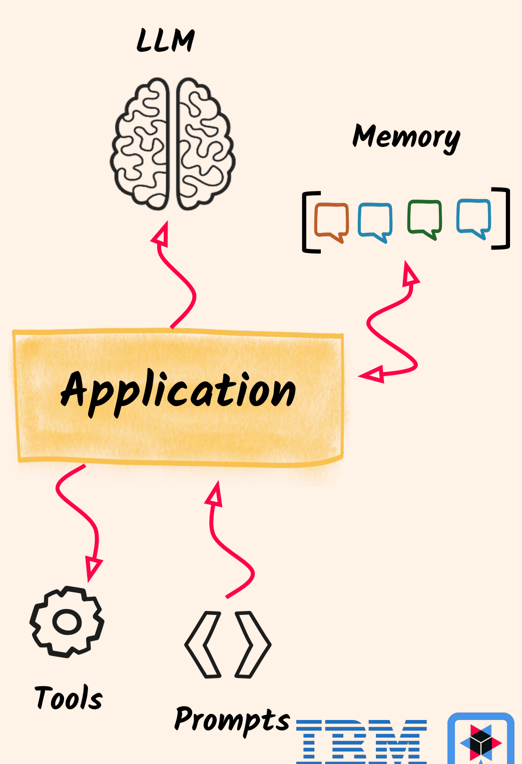


It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways
- A **chat memory** to "remember" previous interactions and make the AI service conversational
- External tools (**function calling**) expanding LLM capabilities and take responsibility for deterministic tasks where generative AI falls short

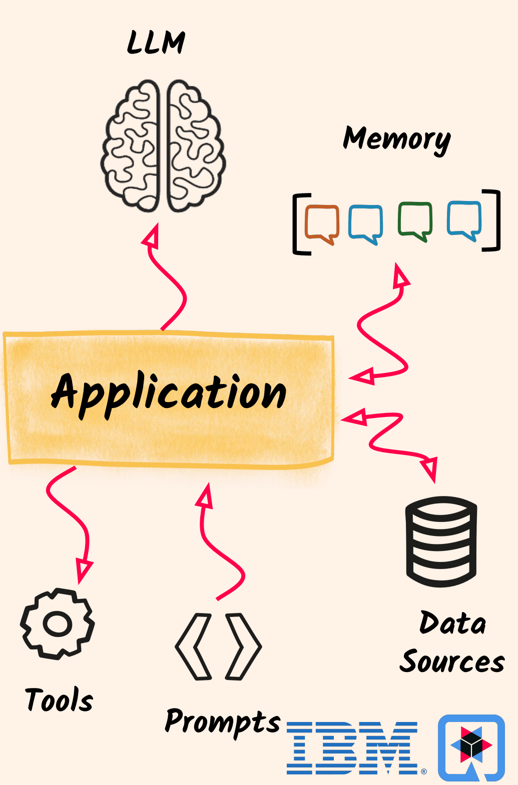


It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways
- A **chat memory** to "remember" previous interactions and make the AI service conversational
- External tools (**function calling**) expanding LLM capabilities and take responsibility for deterministic tasks where generative AI falls short
- **Data/Knowledge sources** to provide contextual information (RAG) and persist the LLM state

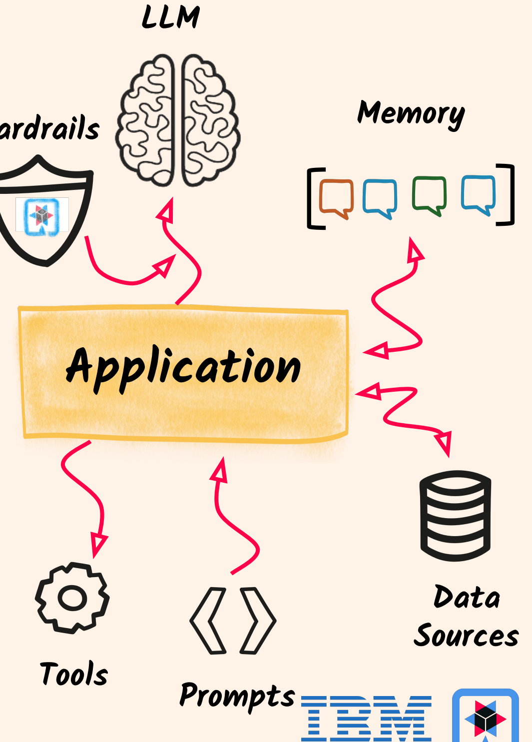


It all starts with a single AI Service

A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways
- A **chat memory** to "remember" previous interactions and make the AI service conversational
- External tools (**function calling**) expanding LLM capabilities and take responsibility for deterministic tasks where generative AI falls short
- **Data/Knowledge sources** to provide contextual information (RAG) and persist the LLM state
- **Guardrails** to prevent malicious input and block wrong or unacceptable responses

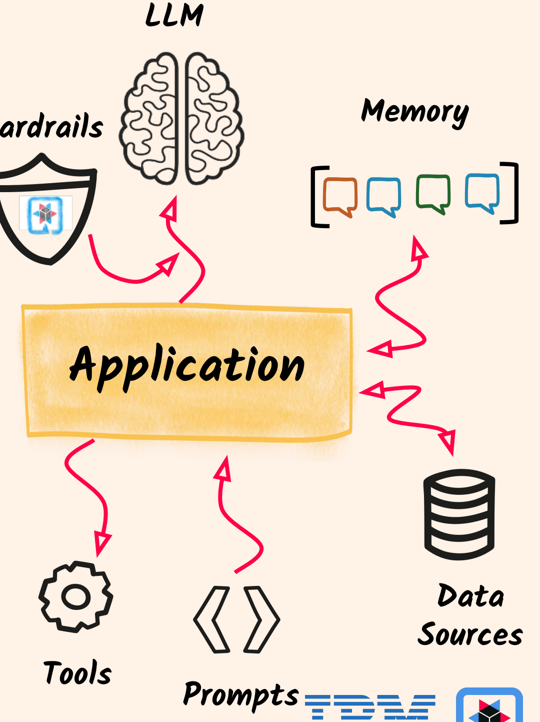


It all starts with a single AI Service

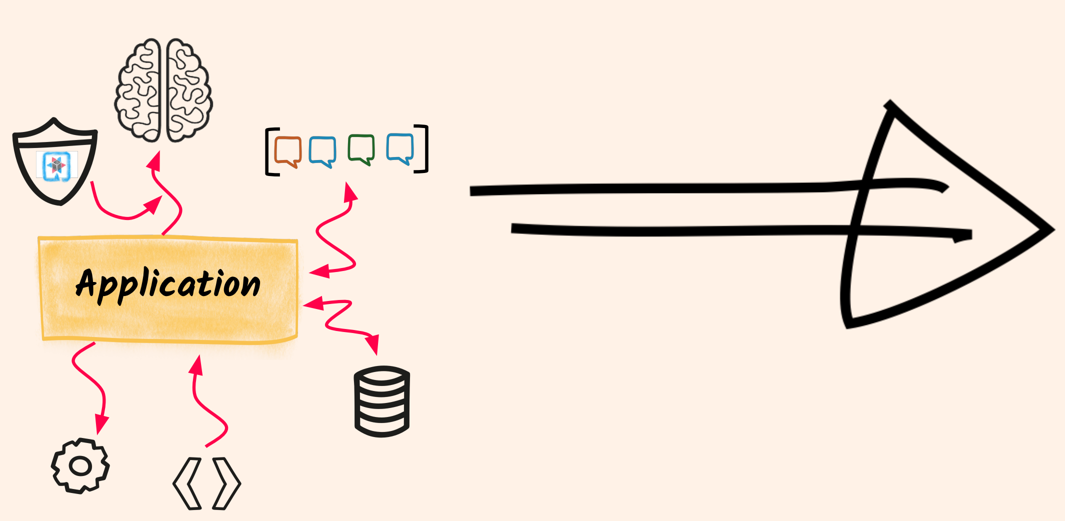
A **Large Language Model** is at the core of any **AI-Infused Application** … but this is not enough.

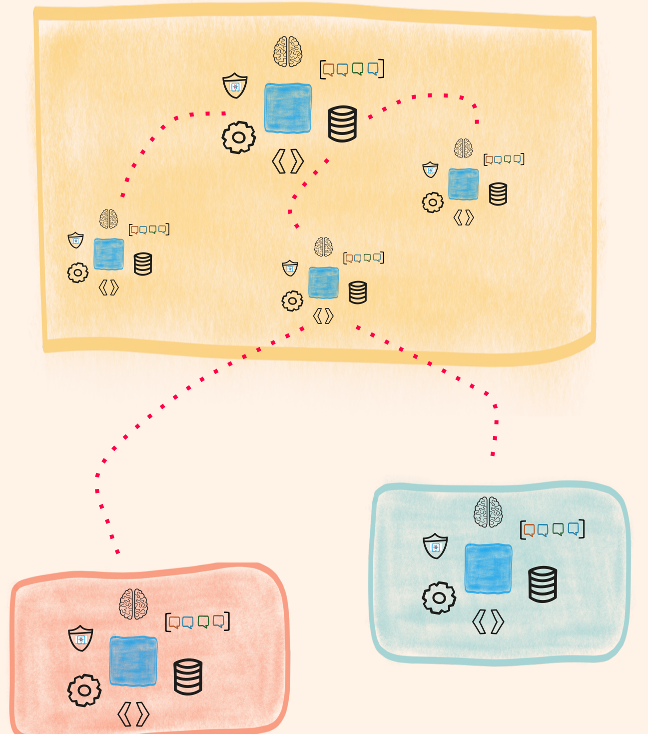
You also need:

- Well crafted **prompts** guiding the LLM in the most precise and least ambiguous possible ways
- A **chat memory** to "remember" previous interactions and make the AI service conversational
- External tools (**function calling**) expanding LLM capabilities and take responsibility for deterministic tasks where generative AI falls short
- **Data/Knowledge sources** to provide contextual information (RAG) and persist the LLM state
- **Guardrails** to prevent malicious input and block wrong or unacceptable responses
- **Production capabilities:** observability, resilience, etc



From a single AI service to Agentic Systems





From single AI Service to Agentic Systems

What turns an **AI service** into an **Agent** is the capability to **collaborate** with other Agents in order to perform more complex tasks and pursue a **common goal**

Foundation
Memory
AI Services
Function calling

Workflow & Patterns
Chaining
Parallelization
Looping

Goal-based Autonomy
Autonomous
Multi-agent
Planning

User-Defined Planning
Pluggable
Planning
Strategy

LangChain4j

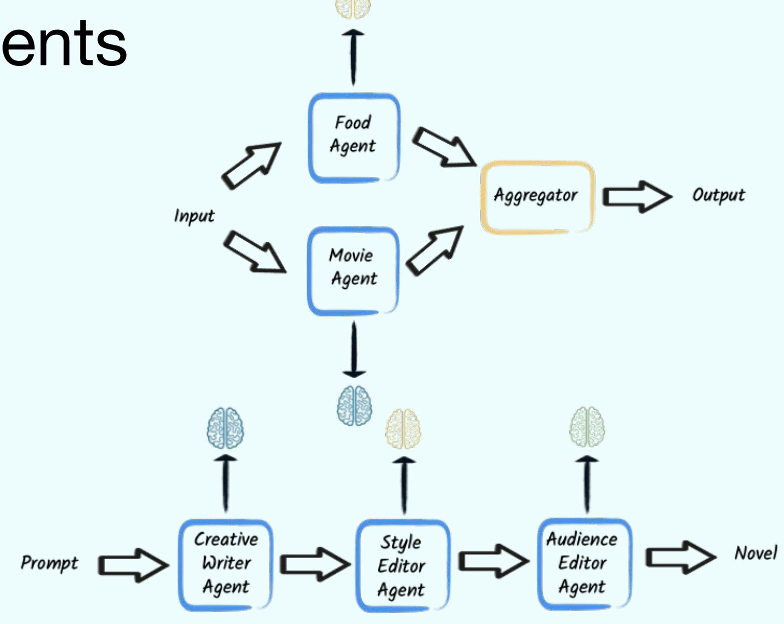
IBM

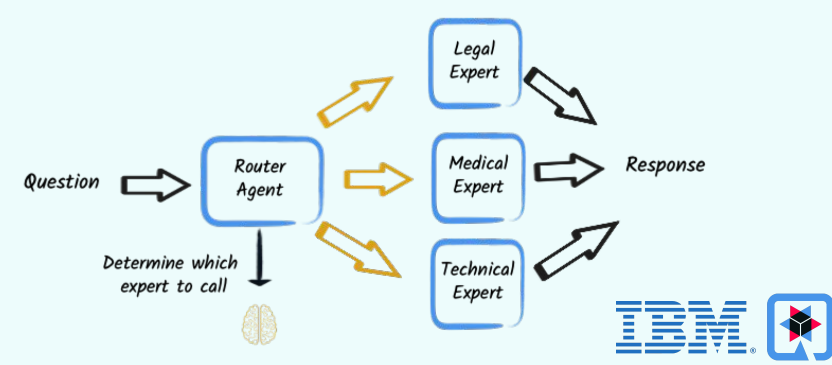
Programmatic Orchestration of Agents

The simplest way to glue agents together is programmatically orchestrating them in fixed and **predetermined workflows**

4 basic patterns that can be used as building blocks to create more complex interactions

- Sequence / Prompt chaining
- Loop / Reflection
- Parallelization
- Conditional / Routing





IBM

Defining Agents

```java
public interface StoryGenerator {

   @Agent("Generate a story based on the given topic,
          for a specific audience and in a specific style")
   String generateStory(String topic, String audience, String style);
}
```

Agent System Interface (API):

```java
var story = storyGenerator.generateStory(
                "dragons and wizards", "young adults", "fantasy");
```

From single agents…

```java
public interface CreativeWriter {
   @UserMessage("""
      You are a creative writer.
      Generate a draft of a story long no more
      than 3 sentence around the given topic.
      The topic is {topic}.""")
   @Agent("Generate a story based on the given topic")
   String generateStory(String topic);
}
```

```java
public interface AudienceEditor {
   @UserMessage("""
  You are a professional editor.
  Analyze and rewrite the following story to
  better align with the target audience of {audience}.
  The story is "{story}".""")
   @Agent("Edit a story to fit a given audience")
   String editStory(String story, String audience);
}
```

```java
public interface StyleEditor {
   @UserMessage("""
      You are a professional editor.
      Analyze and rewrite the following story to better
      fit and be more coherent with the {{style}} style.
      The story is "{story}".""")
   @Agent("Edit a story to better fit a given style")
   String editStory(String story, String style);
```

…to Agentic Systems

Topic,
Audience,
Style

```
public interface CreativeWriter {
   @UserMessage("""
      You are a creative writer.
      Generate a draft of a story long no more
      than 3 sentence around the given topic.
      The topic is {topic}.""")
   @Agent("Generate a story based on the given topic")
   String generateStory(String topic);
}
```

```
public interface AudienceEditor {
   @UserMessage("""
  You are a professional editor.
  Analyze and rewrite the following story to
  better align with the target audience of {audience}.
  The story is "{story}".""")
   @Agent("Edit a story to fit a given audience")
   String editStory(String story, String audience);
}
```

```
public interface StyleEditor {
   @UserMessage("""
      You are a professional editor.
      Analyze and rewrite the following story to better
      fit and be more coherent with the {{style}} style.
      The story is "{story}".""")
   @Agent("Edit a story to better fit a given style")
   String editStory(String story, String style);
```

Sequence Workflow - Composing Agents *Declaratively*

```
public interface StoryGenerator {
    @SequenceAgent(
        outputKey = "story",
        subAgents = {CreativeWriter.class, AudienceEditor.class, StyleEditor.class})
    String generateStory(String topic, String audience, String style);
}


@Inject StoryGenerator storyGenerator;

String story = storyGenerator.generateStory(topic, audience, style);
```

Sequence Workflow - Composing Agents *Programmatically*

```
var creativeWriter =
            AgenticServices.agentBuilder(CreativeWriter. class)
                .chatModel(myModel).outputKey( "story")
                .build();

var audienceEditor = agentBuilder(AudienceEditor. class)
        .chatModel(myModel).outputKey( "story").build();

var styleEditor = agentBuilder(StyleEditor. class)
        .chatModel(myModel).outputKey( "story").build();

var storyGenerator = AgenticServices.sequenceBuilder( StoryGenerator.class)
        .subAgents(creativeWriter, audienceEditor, styleEditor)
        .outputKey("story").build();
```

Invoke the system using the StoryGenerator API

Defining *Agents* *Programmatically*

```
var creativeWriter =
            AgenticServices.agentBuilder(CreativeWriter. class)
                .chatModel(myModel).outputKey( "story")
                .build();

var audienceEditor = agentBuilder(AudienceEditor. class)
        .chatModel(myModel).outputKey( "story").build();

var styleEditor = agentBuilder(StyleEditor. class)
        .chatModel(myModel).outputKey( "story").build();
```

Passing Arguments

```
public interface StoryGenerator {
    @Agent("...")
    String generateStory(String topic, String audience, String style);
}
var writer = agentBuilder(CreativeWriter. class)
             .chatModel(myModel).outputKey( "story")
             .build();
var editor = agentBuilder(AudienceEditor. class)
             .chatModel(myModel).outputKey( "story")
             .build();
var style = agentBuilder(StyleEditor. class)
             .chatModel(myModel).outputKey( "story")
             .build();
var storyGenerator = sequenceBuilder( StoryGenerator.class)
        .subAgents( writer, editor, style).outputKey("story").build();
```

Keeping Track of State

```
public interface StoryGenerator {
   @Agent("...")
   String generateStory(String topic, String audience, String style);
}
var writer = agentBuilder(CreativeWriter. class)
                .chatModel(myModel).outputKey( "story")
                .build();
var editor = agentBuilder(AudienceEditor. class)
                .chatModel(myModel).outputKey( "story")
                .build();
var style = agentBuilder(StyleEditor. class)
                .chatModel(myModel).outputKey( "story")
                .build();
var storyGenerator = sequenceBuilder(StoryGenerator.class)
        .subAgents(writer, editor, style).outputKey("story").build();
```

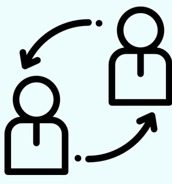
State

- topic
- audience
- style
- story

The AgenticScope

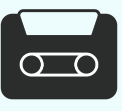
A collection of data shared among the agents participating in the same agentic system

Stores **shared variables**



written by an agent to communicate the results it produced

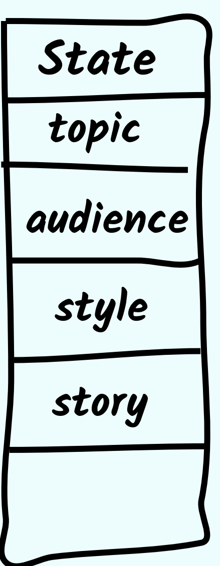
read by another agent to retrieve the necessary to perform its task



Records the sequence of **invocations of all agents** with their responses

Provides **agentic system wide context** to an agent based on former agent executions

Persistable via a pluggable SPI

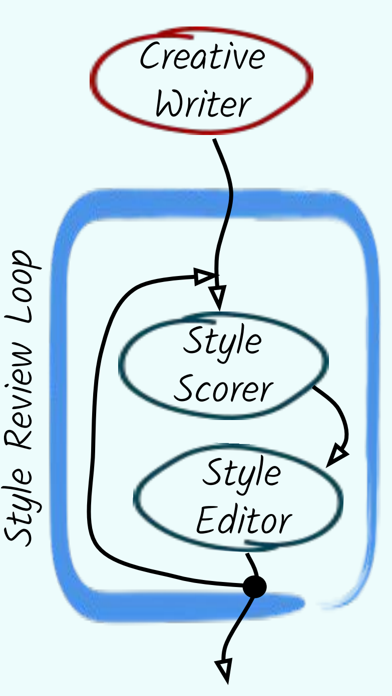


Loop Workflow

```java
public interface StyleScorer {

    @UserMessage("""
        You are a critical reviewer.
        Give a review score between 0.0 and 1.0 for the following story
        based on how well it aligns with the style '{style}'.
        Return only the score and nothing else.
        The story is: "{story}"
        """)
    @Agent("Score a story based on how well it aligns with a given style" )
    double scoreStyle(String story, String style);
}
```

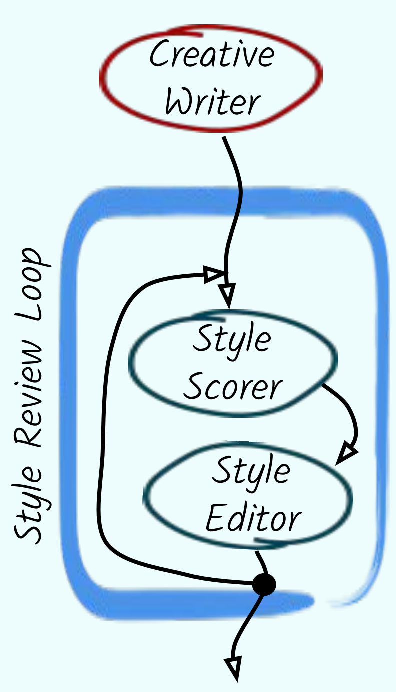
Loop Workflow

```
var styleScorer = agentBuilder(StyleScorer.class)
    .chatModel(myModel).outputKey( "score").build();

UntypedAgent styleReviewLoop = loopBuilder()
    .subAgents(styleScorer, styleEditor)
    .maxIterations(5)
    .exitCondition(scope -> scope.readState("score", 0.0) >= 0.8)
    .build();

var storyGenerator = sequenceBuilder(StoryGenerator. class)
    .subAgents(creativeWriter, styleReviewLoop)
    .outputKey("story").build();
```



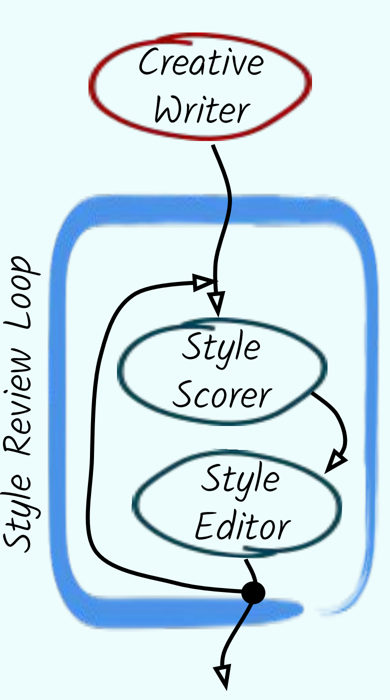
Accessing the AgenticScope

```
var styleScorer = agentBuilder(StyleScorer.class)
    .chatModel(myModel).outputKey( "score").build();

UntypedAgent styleReviewLoop = loopBuilder()
    .subAgents(styleScorer, styleEditor)
    .maxIterations(5)
    .exitCondition(scope -> scope.readState("score", 0.0) >= 0.8)
    .build();

var storyGenerator = sequenceBuilder(StoryGenerator. class)
    .subAgents(creativeWriter, styleReviewLoop)
    .outputKey("story").build();
```



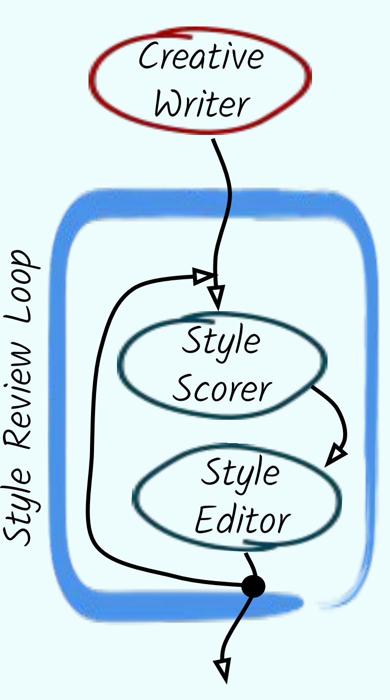
The Untyped Agent

```
var styleScorer = agentBuilder(StyleScorer.class)
    .chatModel(myModel).outputKey( "score").build();


UntypedAgent styleReviewLoop = loopBuilder()
    .subAgents(styleScorer, styleEditor)
    .maxIterations(5)
    .exitCondition(scope -> scope.readState("score", 0.0) >=
0.8)
    .build();


var storyGenerator = sequenceBuilder(StoryGenerator. class)
    .subAgents(creativeWriter, styleReviewLoop)
    .outputKey("story").build();
```



Embedding Workflows, in Other Workflows

```
var styleScorer = agentBuilder(StyleScorer.class)
    .chatModel(myModel).outputKey( "score").build();

UntypedAgent styleReviewLoop = loopBuilder()
    .subAgents(styleScorer, styleEditor)
    .maxIterations( 5)
    .exitCondition( scope -> scope.readState("score", 0.0) >= 0.8)
    .build();

var storyGenerator = sequenceBuilder(StoryGenerator. class)
    .subAgents(creativeWriter, styleReviewLoop)
    .outputKey("story").build();
```



Parallel Workflow

```java
public interface EveningPlannerAgent {
    @Agent
    List<EveningPlan> plan(@V("mood") String mood);
}
```

```java
public interface FoodExpert {
    @UserMessage("""
        You are a great evening planner.
        Propose a list of 3 meals matching the given mood.
        The mood is {{mood}}.
        For each meal, just give the name of the meal.
        Provide a list with the 3 items and nothing else.
        """)
    @Agent
    List<String> findMeal(@V("mood") String mood);
}

public interface MovieExpert {
    @UserMessage("""
        You are a great evening planner.
        Propose a list of 3 movies matching the given mood.
        The mood is {{mood}}.
        Provide a list with the 3 items and nothing else.
        """)
    @Agent
    List<String> findMovie(@V("mood") String mood);
}
```

```java
EveningPlannerAgent eveningPlannerAgent = AgenticServices
        .parallelBuilder(EveningPlannerAgent.class)
        .subAgents(foodAgent, movieAgent)
        .outputKey("plans")
        .output(agenticScope -> {
            List<String> movies = agenticScope.readState("movies");
            List<String> meals = agenticScope.readState("meals");
            List<EveningPlan> moviesAndMeals = new ArrayList<>();
            for (int i = 0; i < movies.size(); i++) {
                if (i >= meals.size()) { break; }
                moviesAndMeals.add(new EveningPlan(movies.get(i),
                                                   meals.get(i)));
            }
            return moviesAndMeals;
        });

List<EveningPlan> plans = eveningPlannerAgent.plan("romantic");
```

Food Agent
Movie Agent
Input
Aggregator
Output

IBM

Routing

```java
public interface ExpertRouterAgent {

    @Agent
    String ask(@V("request") String request);
}
```

Question → Router Agent → Legal Expert / Medical Expert / Technical Expert → Response

Determine which expert to call

```java
public enum RequestCategory {
    LEGAL, MEDICAL, TECHNICAL, UNKNOWN
}

public interface RouterAgent {

    @UserMessage("""
        Analyze the user request and categorize
        it as 'legal', 'medical' or 'technical',
        The user request is: '{{request}}'.
        """)
    @Agent
    String askToExpert(@V("request") String request);
}
```

```java
public interface MedicalExpert {

    @UserMessage("""
        You are a medical expert.
        Analyze the user request under a medical point
        of view and provide the best possible answer.
        The user request is {{request}}.
        """)
    @Agent("A medical expert")
    String medical(@V("request") String request);
}
```

```java
RouterAgent routerAgent =
        AgenticServices.agentBuilder(RouterAgent.class)
            .chatModel(myModel).outputKey("category").build();

MedicalExpert medicalExpert = AgenticServices
        .agentBuilder(MedicalExpert.class)
        .chatModel(myModel).outputKey("response").build());

LegalExpert legalExpert = ...
TechnicalExpert techExpert =

UntypedAgent expertsAgent = AgenticServices.conditionalBuilder()
    .subAgents(scope ->
        scope.readState("category",UNKNOWN) == MEDICAL, medicalExpert)
    .subAgents(scope ->
        scope.readState("category",UNKNOWN) == LEGAL, legalExpert)
    .subAgents(scope ->
        scope.readState("category",UNKNOWN) == TECHNICAL, techExpert)
    .build();

ExpertRouterAgent expertRouterAgent = AgenticServices
        .sequenceBuilder(ExpertRouterAgent.class)
        .subAgents(routerAgent, expertsAgent)
        .outputKey("response").build();

expertRouterAgent.ask("I broke my leg what should I do")
```

Memory and Context Engineering

- Agents are meant to be **stateless**, meaning that they **do not maintain any context** or memory of previous interactions

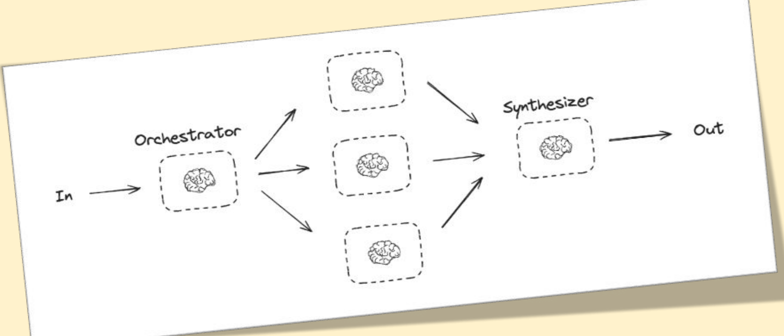
- AI Services can be provided with a **ChatMemory**, but this is **local** to the single agent, so in many cases **not enough** in a complex agentic system

- In general an agent requires a **broader context**, carrying information about everything that happened in the agentic system before its invocation

- That's another task for the **AgenticScope**

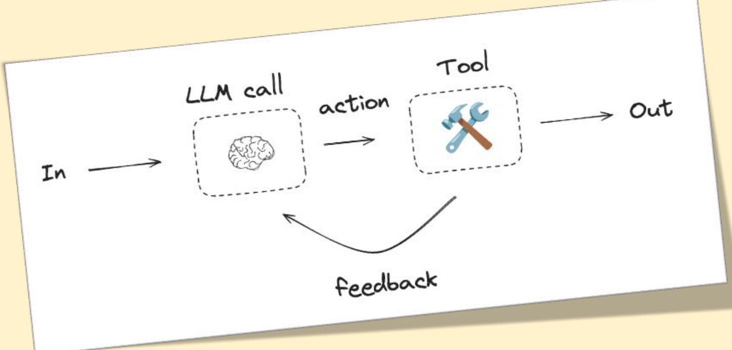
From AI Orchestration to Autonomous Agentic AI

Workflow



LLMs and tools are **programmatically orchestrated** through predefined code paths and workflows

Autonomous Agents



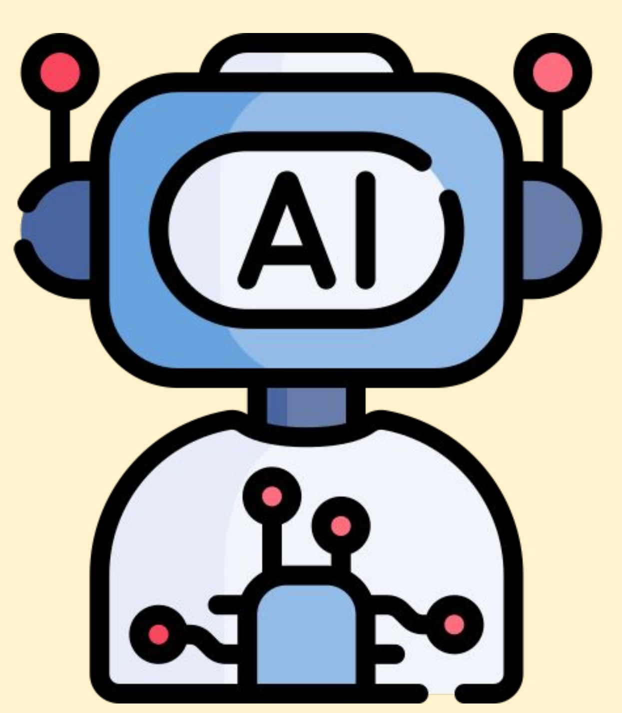
LLMs dynamically direct their own processes and tool usage, **maintaining control** over how they execute tasks

An Autonomous Agentic AI Case Study – The **Supervisor** Pattern

- All agentic systems explored so far orchestrated agents programmatically in a **fully deterministic** way

- In many cases agentic systems have to be more **flexible and adaptive**

- An Autonomous Agentic AI system
  - Takes **autonomous** decisions
  - Decides **iteratively** which agent has to be invoked next
  - Uses the result of previous interactions to determine **if it is done** and achieved its final goal
  - Uses the context and state to generate the **arguments** to be passed to the selected agent



Supervisor pattern

Input

Determine if
done or next
invocation

Supervisor

Done

Response

Select and invoke
(Agent Invocation)

Agent result +
State

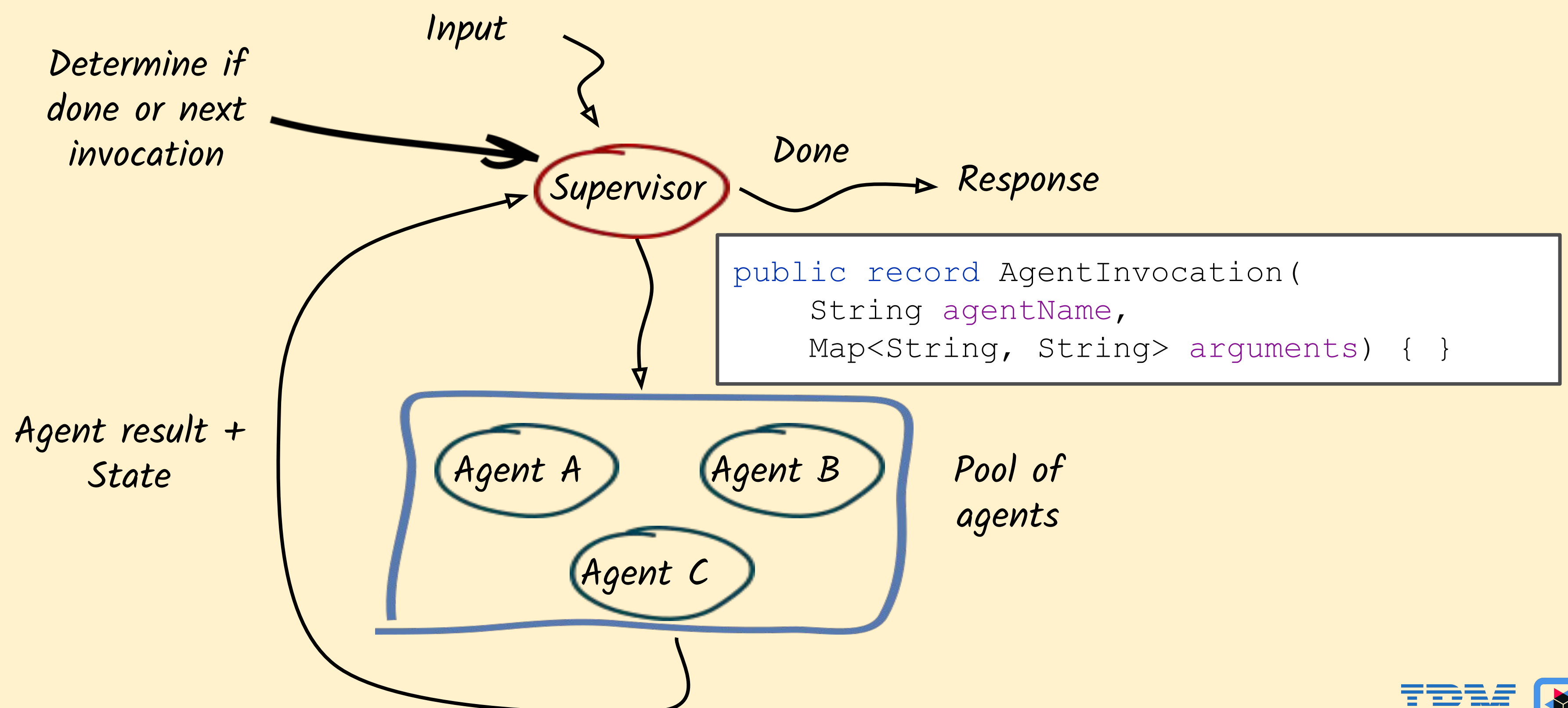
Agent A

Agent B

Agent C

Pool of
agents

Supervisor Pattern: Keeping Track of Invocations




```java
public record AgentInvocation(
    String agentName,
    Map<String, String> arguments) { }
```

DEMO TIME !!!

Car Return

CarProcessingWorkflow
Sequential

Step 1:
FeedbackAnalysisWorkflow
Parallel Mapper

FeedbackTask.cleaning()

FeedbackTask.maintenance()

FeedbackTask.disposition()

FeedbackAnalysisAgent

FeedbackAnalysisResults

Step 2:
FleetSupervisorAgent
Autonomous Orchestration

Disposition Required?

Yes

PricingAgent
Estimate Value

Value > $15k?

Yes - HIGH VALUE

DispositionProposalAgent
Create Proposal

HumanApprovalAgent
@HumanInTheLoop

No - LOW VALUE

No

RETURN

VELOCITY
RENT-A-CAR

Production Deployment

- Agentic apps are mostly just ‘regular apps’ when it comes to production
- All the same production considerations apply:
    - **observability** (including token usage)
    - **AuthN** & **AuthZ** (MCP, A2A)
    - **health endpoints**
    - **scaling**
    - …

Custom Agentic Patterns

Agentic Scope

State

Request

Pluggable Planner

Workflow
Supervisor
GOAP
P2P ...

Result

Action

Execution Layer

Customizable by the framework (Quarkus)

Invoke

Agent A

Agent B

Agent C

Other langchain4j-agentic features

➢ Error handling and recovery strategies

```
UntypedAgent novelCreator = AgenticServices.sequenceBuilder()
        .subAgents(creativeWriter, audienceEditor, styleEditor)
        .errorHandler(errorContext -> {
            if (errorContext.agentName().equals("generateStory") &&
                    errorContext.exception()instanceof MissingArgumentException mEx &&
                    mEx.argumentName().equals("topic")) {
                errorContext.agenticScope().writeState("topic", "dragons and wizards");
                return ErrorRecoveryResult.retry();
            }
            return ErrorRecoveryResult.throwException();
        })
        .outputKey("story")
        .build();
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic **non-AI** agents

```java
public class ExchangeOperator {

    @Agent("A money exchanger that converts a given amount of money from the original to the target currency")
    public Double exchange(@V("originalCurrency") String originalCurrency,
                           @V("amount") Double amount,
                           @V("targetCurrency") String targetCurrency) {

        // invoke the REST API to perform the currency exchange
    }
}
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop

```
HumanInTheLoop humanInTheLoop = AgenticServices.humanInTheLoopBuilder()
        .description("An agent that asks the audience for the story)"
        .inputName("topic")
        .outputKey("audience")
        .requestWriter(topic -> {
            System.out.println("Which audience for topic " + topic + "?");
            System.out.print("> ");
        })
        .responseReader(() -> System.console().readLine())
        .build();
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop
- Asynchronous agents

```
FoodExpert foodExpert = AgenticServices
                .agentBuilder(FoodExpert.class)
                .chatModel(baseModel())
                .async(true)
                .outputKey("meals")
                .build();
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop
- Asynchronous agents
- A2A integration

```
CreativeWriter creativeWriter = AgenticServices.a2aBuilder(A2A_SERVER_URL, CreativeWriter.class)
                                                .outputKey("story")
                                                .build();
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop
- Asynchronous agents
- A2A integration
- Comprehensive Declarative API

```
public interface StyleReviewLoopAgent {

    @LoopAgent(
        description = "Review the story for the given style",
        outputKey = "story", maxIterations = 5,
        subAgents = { StyleScorer.class, StyleEditor.class }
    )
    String write(@V("story") String story);

    @ExitCondition
    static boolean exit(@V("score") double score) {
        return score >= 0.8;
    }
}
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop
- Asynchronous agents
- A2A integration
- Comprehensive Declarative API
- CDI support

```
public interface StoryCreator {

    @SequenceAgent(outputKey = "story", subAgents = {
            CreativeWriter.class,
            AudienceEditor.class,
            StyleEditor.class
    })
    String write(@V("topic") String topic,
                 @V("style") String style,
                 @V("audience") String audience);
}

@Inject
StoryCreator storyCreator;
```

Other langchain4j-agentic features

- Error handling and recovery strategies
- Programmatic non-AI agents
- Human-in-the-loop
- Asynchronous agents
- A2A integration
- Comprehensive Declarative API
- CDI support
- Dynamic Model Selection

```
public static class DynamicModelSelector {

    @Inject
    @ModelName("baseModel")
    ChatModel baseModel;

    @Inject
    @ModelName("enhancedModel")
    ChatModel enhancedModel;

    ChatModel select(CritiqueResult critique) {
        return critique.score() > 7.8 ? enhancedModel :
baseModel;
    }
}
```

What did we see and didn't see?

FUNCTION CALLING AGENT
AGENTIC WORKFLOW
SCOPE MEMORY MODELS CONTEXT
STATE QUARKUS SUPERVISOR
WORKFLOW SPI LANGCHAIN4J
RESILIENCE DECLARATIVE API
MCP PROGRAMMATIC AGENT
HUMAN-IN-THE-LOOP A2A
PLANNER SPI OBSERVABILITY

https://docs.langchain4j.dev/tutorials/agents

https://docs.quarkiverse.io/quarkus-langchain4j

https://quarkus.io/quarkus-workshop-langchain4j/

Slides