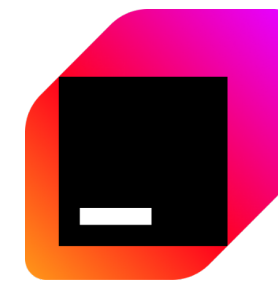


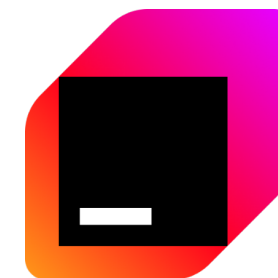
Koog your
own AI!
Delicious
recipes



JETBRAINS



About us



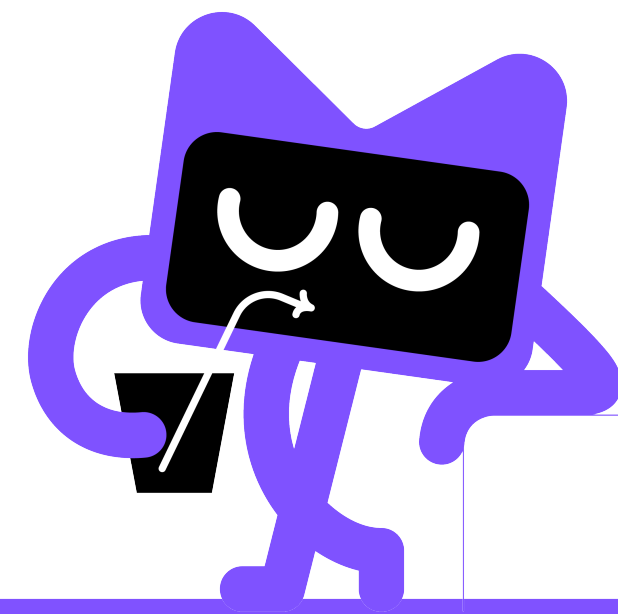
JETBRAINS



Simon Vergauwen
Developer Advocate | ex Principal Engineer

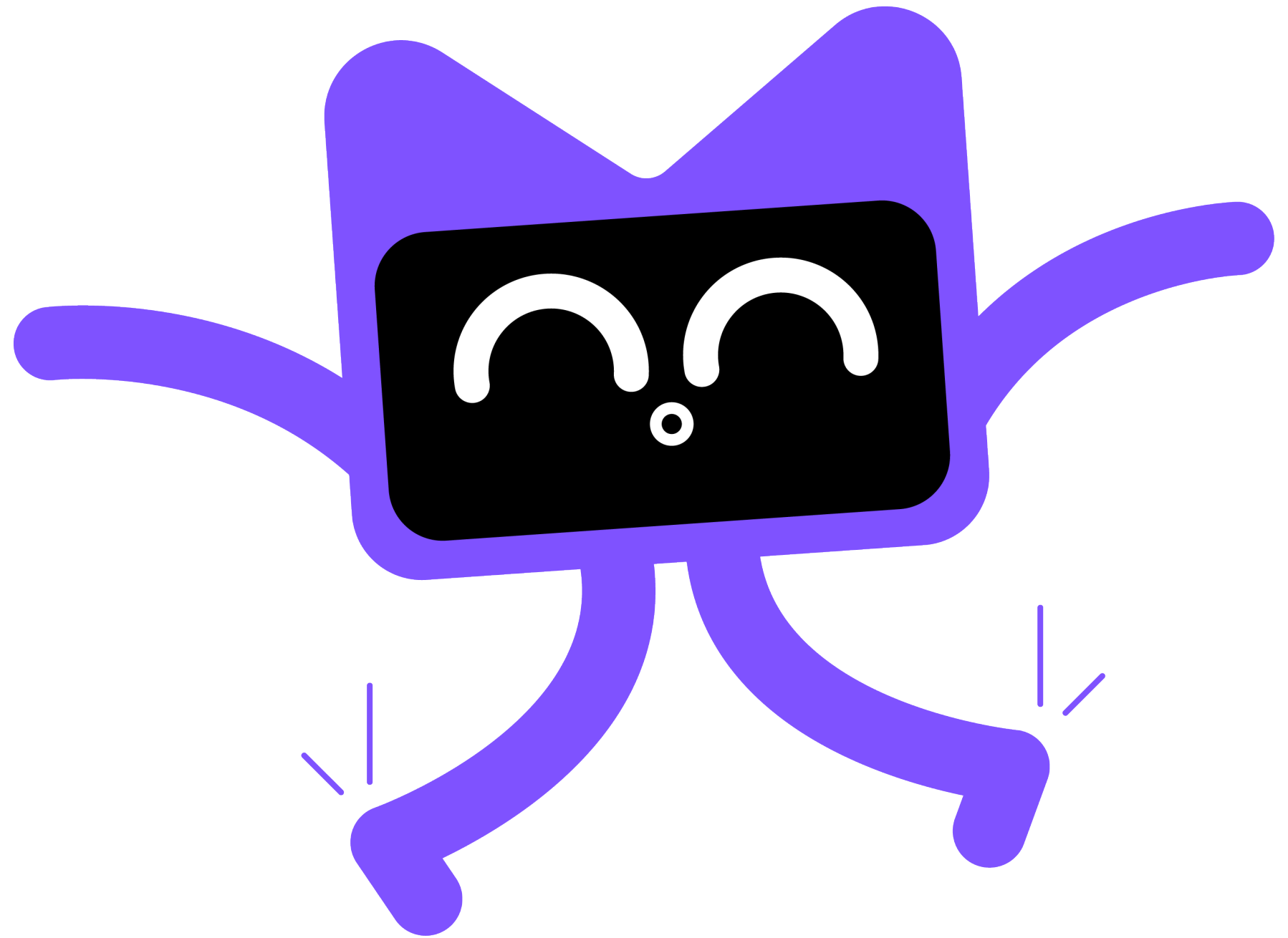


Maria Tigina
Developer @ Koog | ex AI Researcher



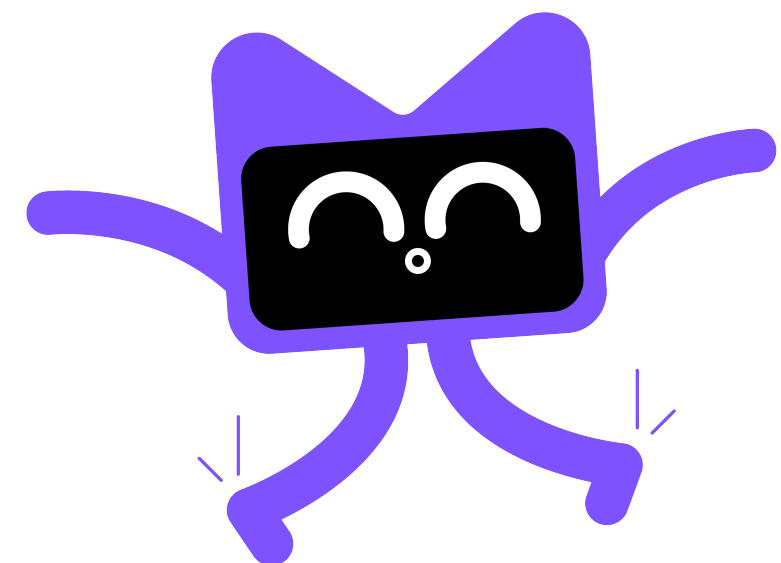
Koog

- DSL: High- and low-level API
- Tools, MCP and Structured Output
- Agent Graph Workflows
- Memory, Storage and Persistency
- Observability and tracing
- Spring & Ktor Support
- Kotlin Multiplatform



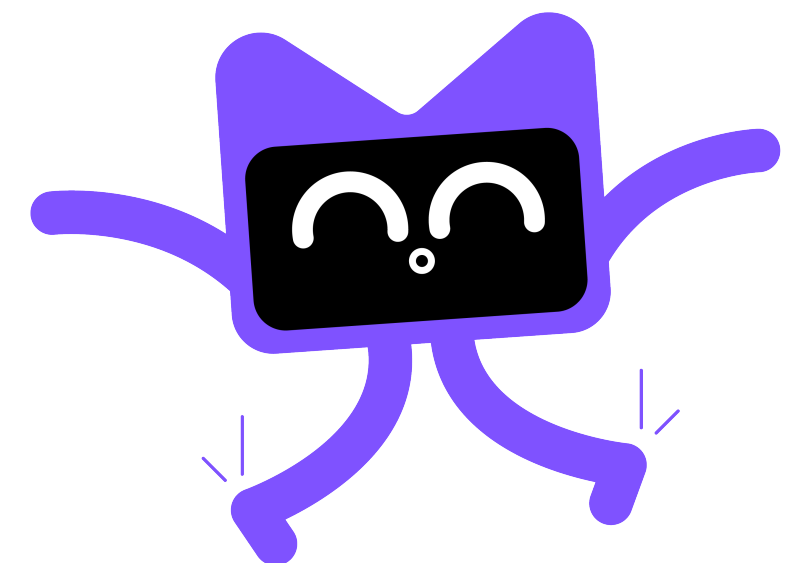
Koog

- DSL: High -and low level API
- Custom Tools & MCP
- Agent Graph Workflows
- Memory, Storage and Persistency
- Observability, and tracing
- Spring & Ktor Support
- Kotlin Multiplatform



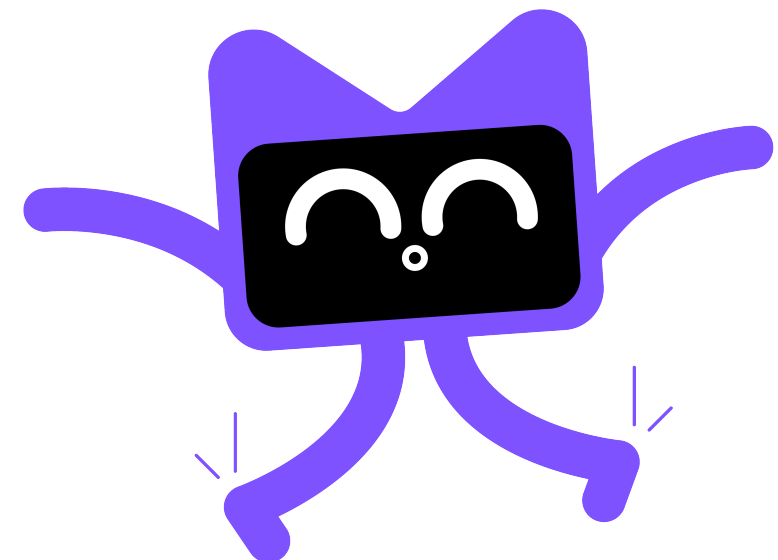
LLMClient

```
val openAI = OpenAILLMClient(apiKey = "key")
```



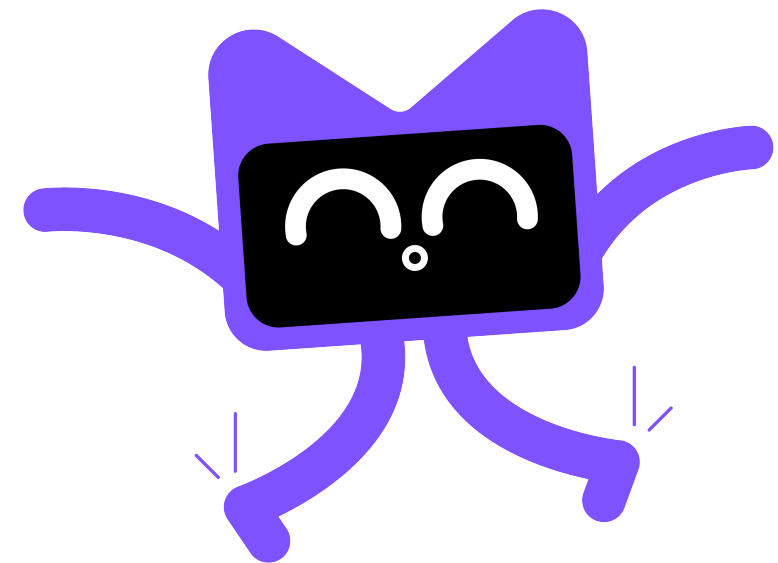
LLMClient

```
val openAIClient = OpenAILLMClient(  
    apiKey = "key",  
    settings = OpenAIClientSettings(  
        baseUrl = "https://api.openai.com",  
        chatCompletionsPath = "v1/chat/completions",  
        embeddingsPath = "v1/embeddings",  
        moderationsPath = "v1/moderations",  
    ),  
    baseClient = HttpClient()  
)
```

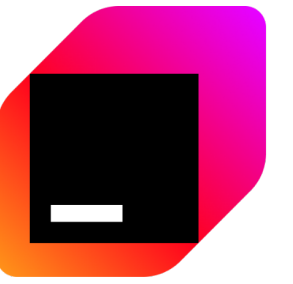


Executing

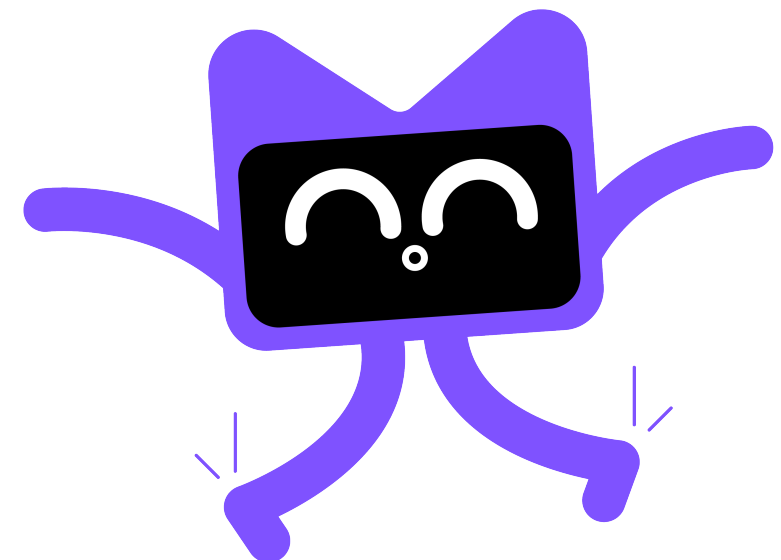
```
openAI.execute(prompt("my-prompt") {  
    system("You're a helpful assistant.")  
    user("Tell me a Kotlin Joke!")  
}, OpenAIModels.Chat.GPT5)
```



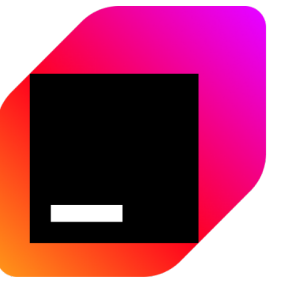
Responses



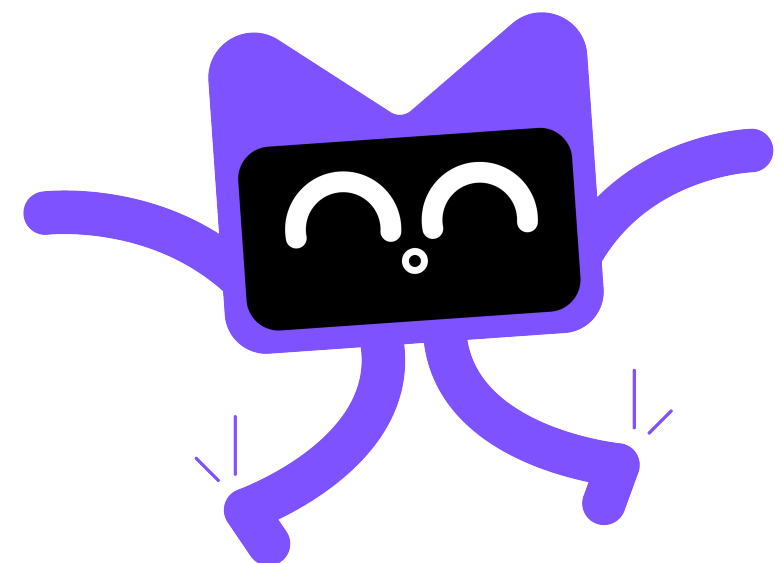
```
val responses: List<Message.Response> =  
    openAI.execute(prompt("my-prompt") {  
        system("You're a helpful assistant.")  
        user("Tell me a Kotlin Joke!")  
    }, OpenAIModels.Chat.GPT5)
```



Response

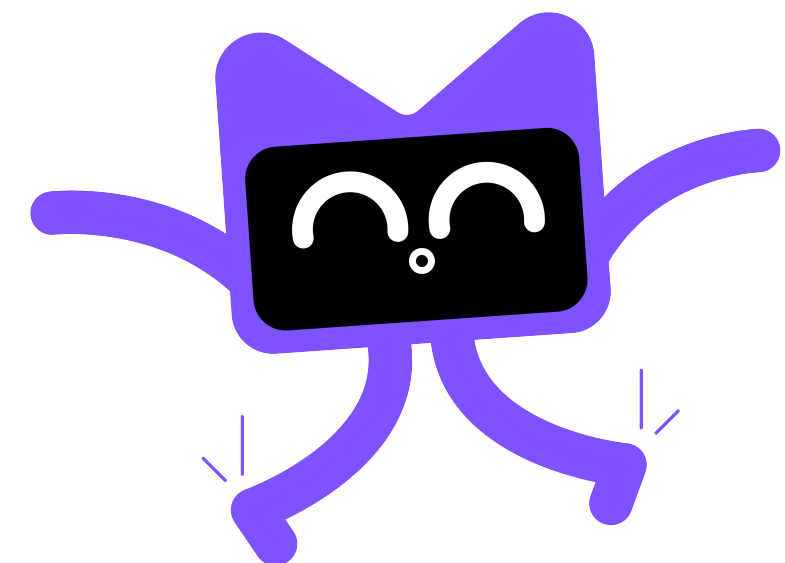


```
for(response in responses) {  
    when(response) {  
        is Message.Assistant → TODO()  
        is Message.Tool.Call → TODO()  
    }  
}
```



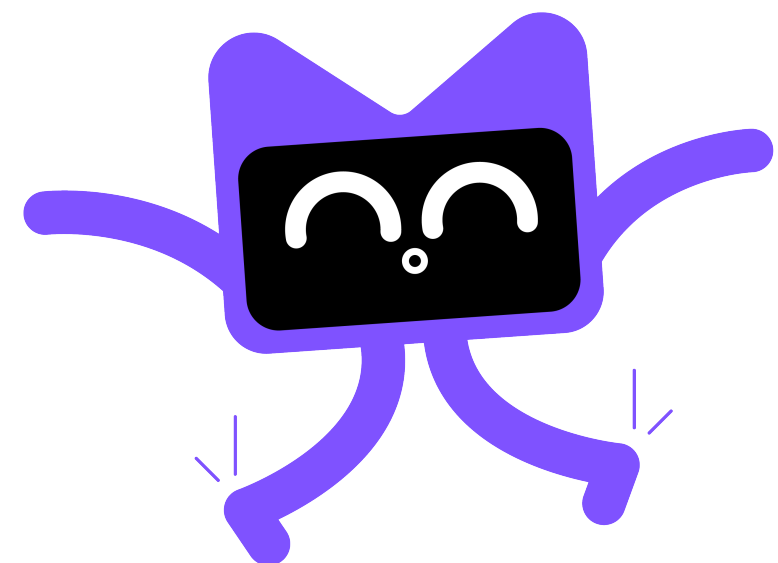
Prompt

```
val prompt: Prompt = prompt("my-prompt") {  
    system("You're a helpful assistant.")  
    user("Tell me a Kotlin Joke!")  
}
```



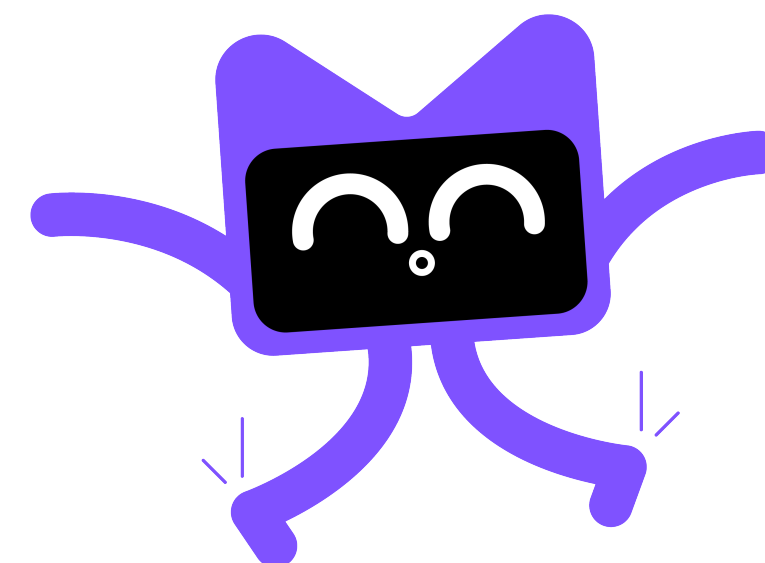
LLMParams

```
prompt("my-prompt", params = LLMParams(  
    temperature = 0.7,  
    maxTokens = 1000,  
    toolChoice = LLMParams.ToolChoice.Auto,  
    // ..  
)) {  
    system("You're a helpful assistant.")  
    user("Tell me a Kotlin Joke!")  
}
```



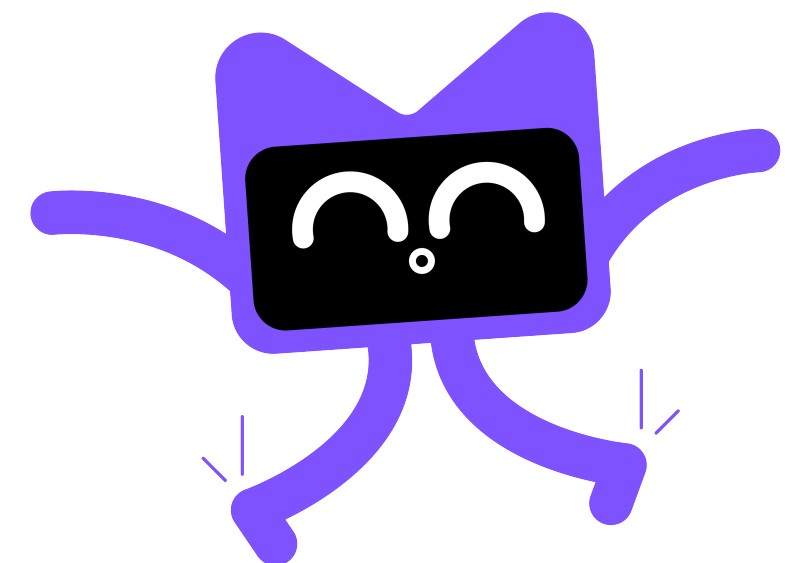
Prompt

```
val prompt = prompt("my-prompt") {  
  system {  
  
  }  
}
```



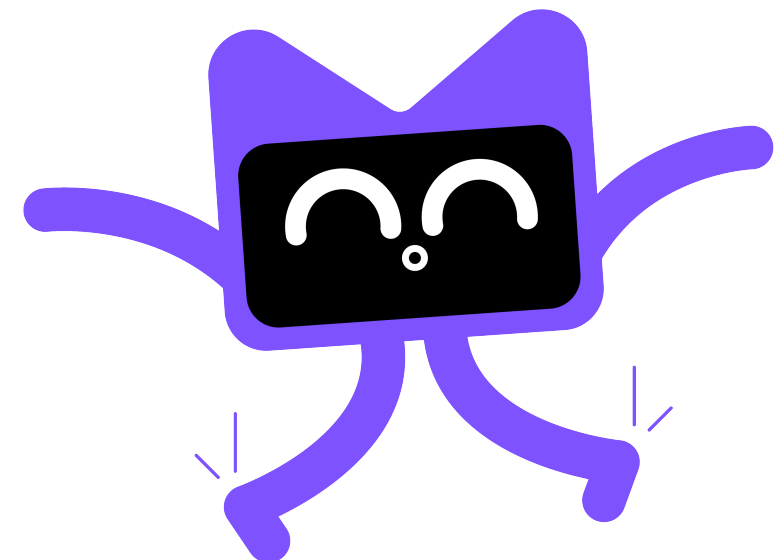
Prompt

```
val prompt = prompt("my-prompt") {  
    system {  
        markdown {  
            header(1, "Role and Capabilities")  
            text("You're a helpful assistant.")  
        }  
    }  
}
```



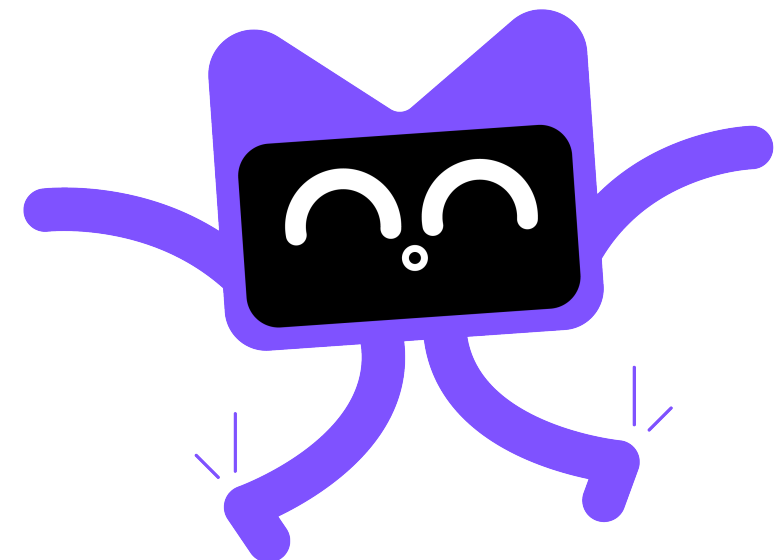
Prompt

```
val prompt = prompt("my-prompt") {  
  system {  
    markdown {  
      header(1, "Role and Capabilities")  
      text("You're a helpful assistant.")  
      bulleted {  
        item("You can summarise a document.")  
      }  
    }  
  }  
}
```



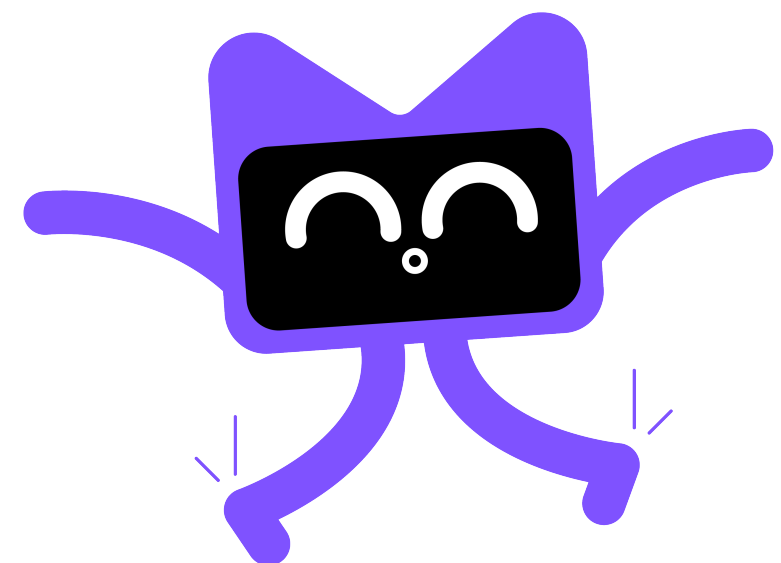
Prompt

```
val prompt = prompt("my-prompt") {  
  system {  
    markdown {  
      header(1, "Role and Capabilities")  
      text("You're a helpful assistant.")  
      bulleted {  
        item("You can summarise a document.")  
      }  
    }  
  }  
}
```



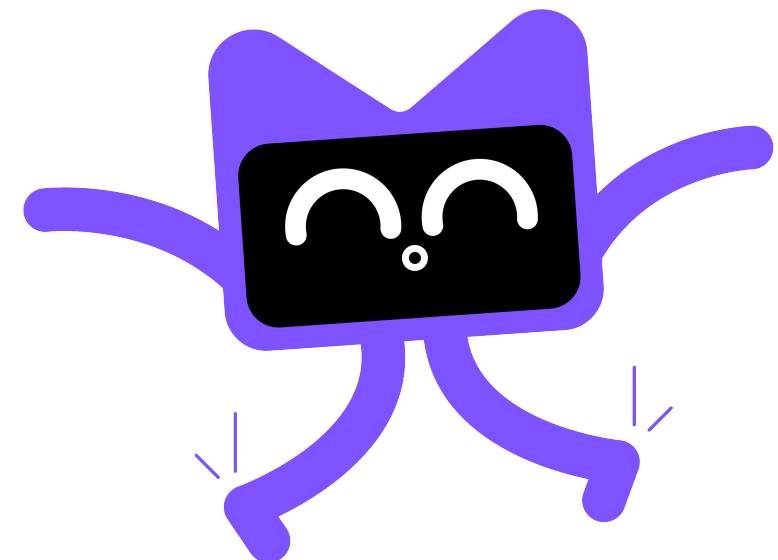
Prompt

```
val prompt = prompt(prompt, "my-prompt") {  
    user {  
        xml {  
            tag("user-question") {  
                text("Describe the Kotlin mascot in 100 words.")  
            }  
        }  
    }  
}
```



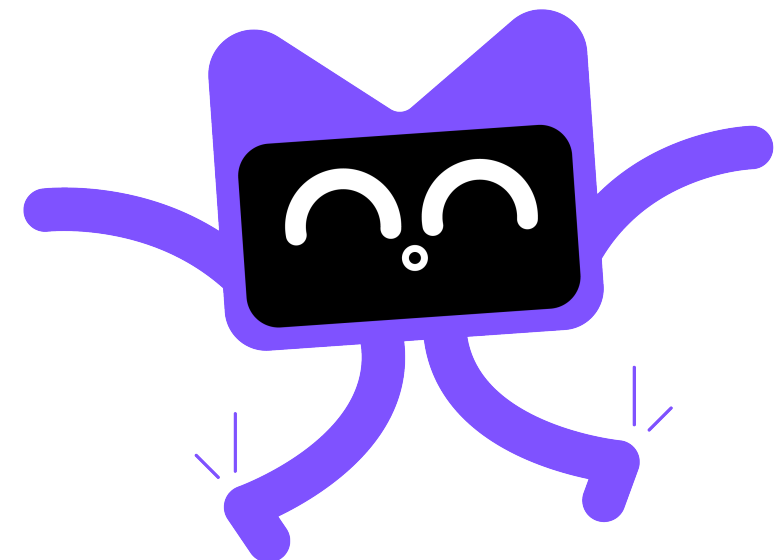
Attachments

```
val file: Prompt = prompt("attachments") {  
    user("Summarise this document:") {  
        file(  
            Attachment.File(  
                AttachmentContent.Binary.Bytes("raw".toByteArray()),  
                "format",  
                "mimeType",  
            )  
        )  
    }  
}
```



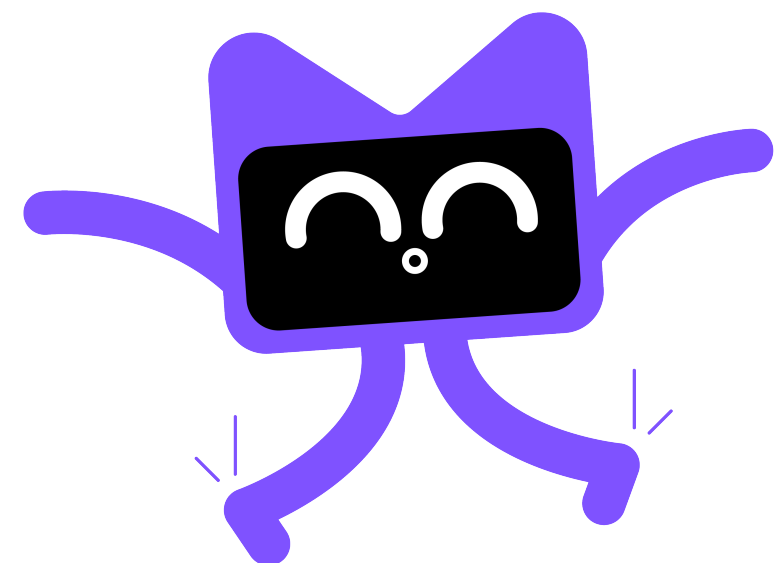
Attachments

```
prompt("files") {  
  user("Summarise this document:") {  
    textFile(Path("doc.txt"), "text/plain")  
    binaryFile(Path("blob"), "application/octet-stream")  
    file(youTrackOpenApiJson, "application/json")  
  }  
}
```



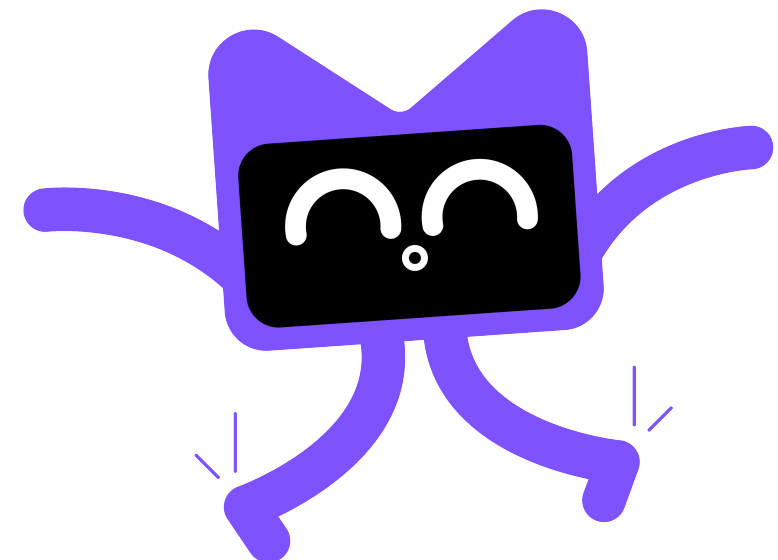
Attachments: Vision

```
prompt("vision") {  
    user("Generate a descriptions attached video and image") {  
        image(Path("kodee.png"))  
        video(bigBuckBunnyUrl)  
    }  
}
```



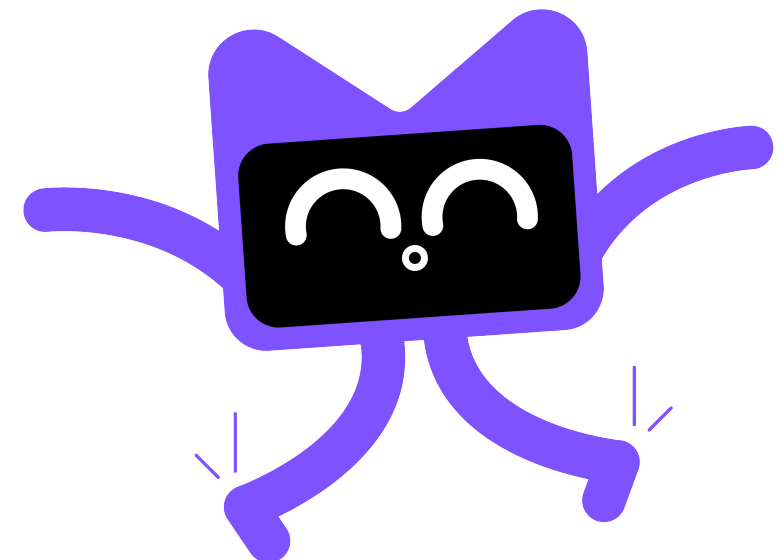
Attachments: Video

```
prompt("audio") {  
    user("Generate a SRT formatted transcript") {  
        audio(Path("script.mp3"))  
        audio(mp3Url)  
    }  
}
```



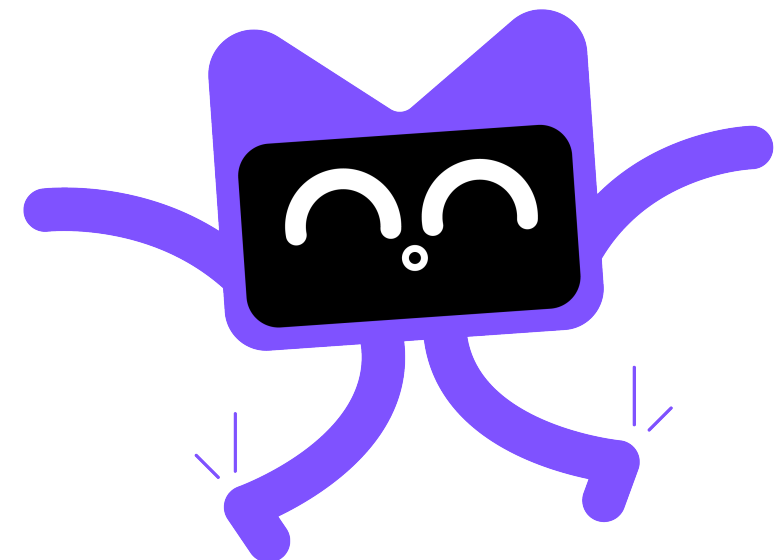
Attachments: Video

```
prompt("audio") {  
    user("Generate a SRT formatted transcript") {  
        audio(Path("script.mp3"))  
        audio(mp3Url)  
    }  
}
```



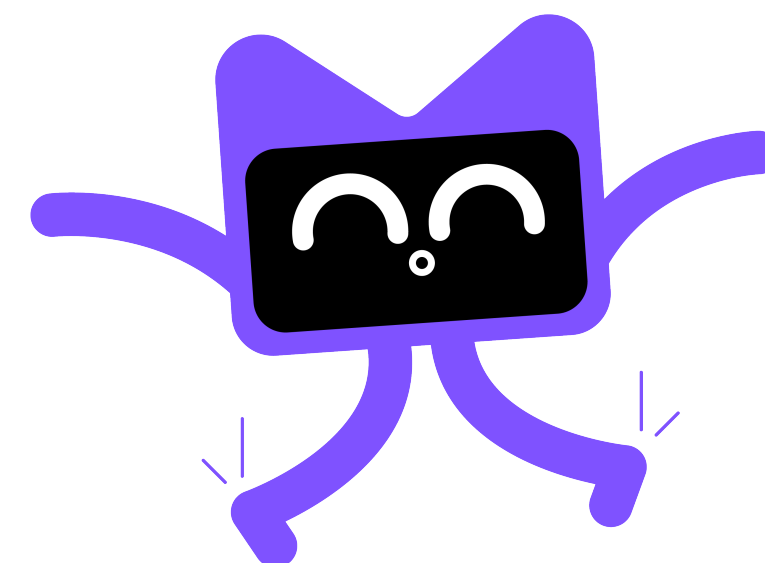
Tools

```
executor.execute(  
    prompt = prompt("weather") {  
        system("You're a helpful assistant.")  
        user("What is the weather like in Amsterdam?")  
    },  
    model = OpenAIModels.Chat.GPT5,  
    tools = listOf(WeatherTool())  
)
```



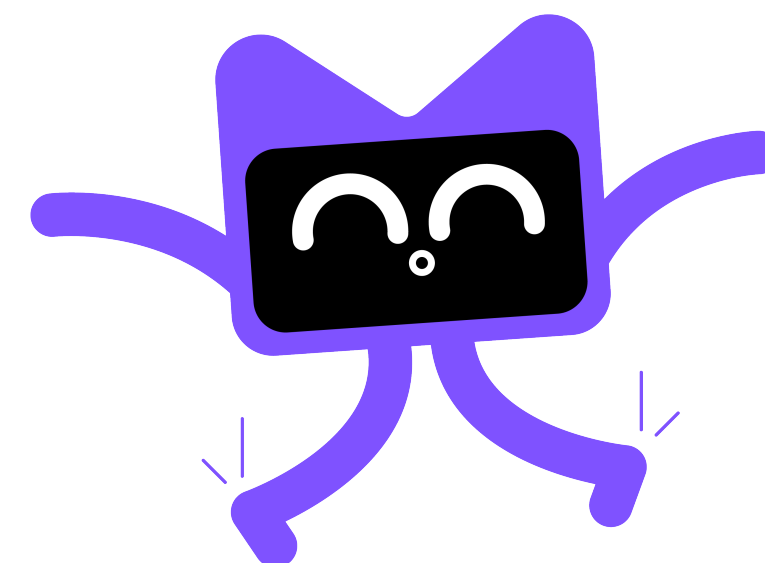
Tool

```
fun calculateDate(  
    date: String,  
    days: Int,  
    months: Int  
): String
```



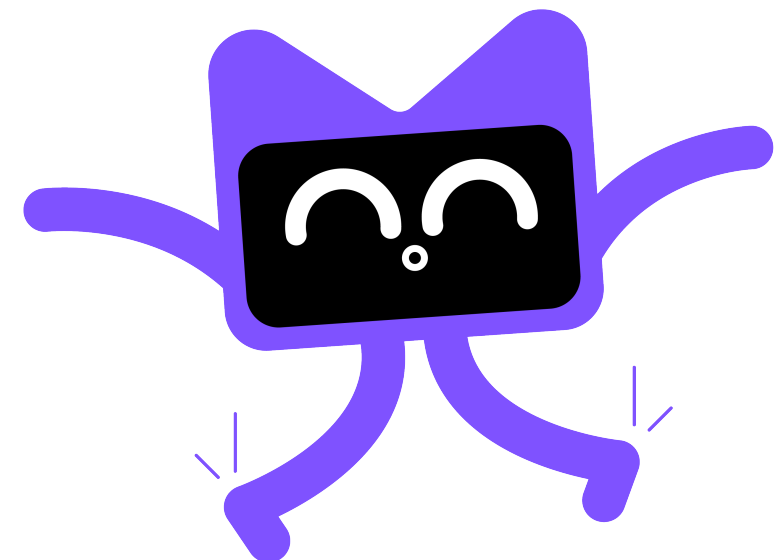
Tool

```
@Tool  
fun calculateDate(  
    date: String,  
    days: Int,  
    months: Int  
): String
```



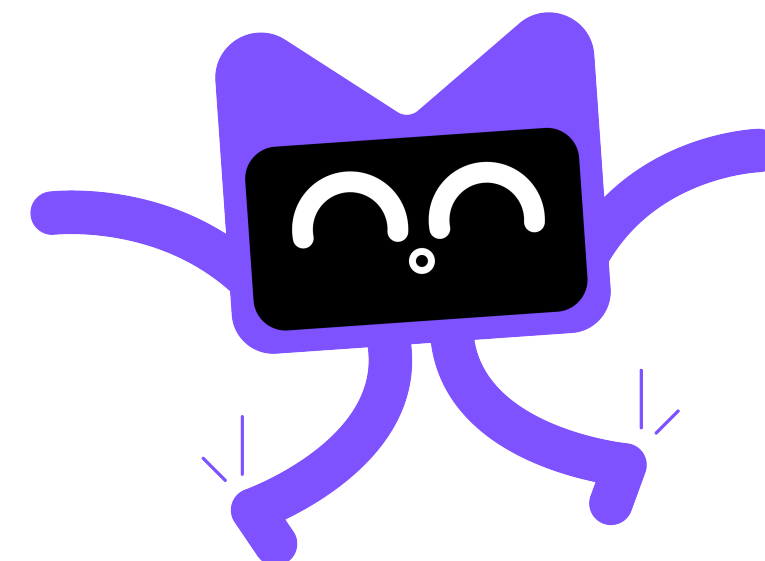
Tool

```
@Tool
@LLMDescription("Add a duration to a date." +
    "Use this tool when you need to calculate offsets, such as  
tomorrow, in two days, yesterday.")
fun addDate(
    @LLMDescription("The date to add to in ISO format, e.g. 2022-01-01")
    date: String,
    @LLMDescription("The number of days to add, defaults to 0")
    days: Int,
    @LLMDescription("The number of months to add, defaults to 0")
    months: Int
): String
```



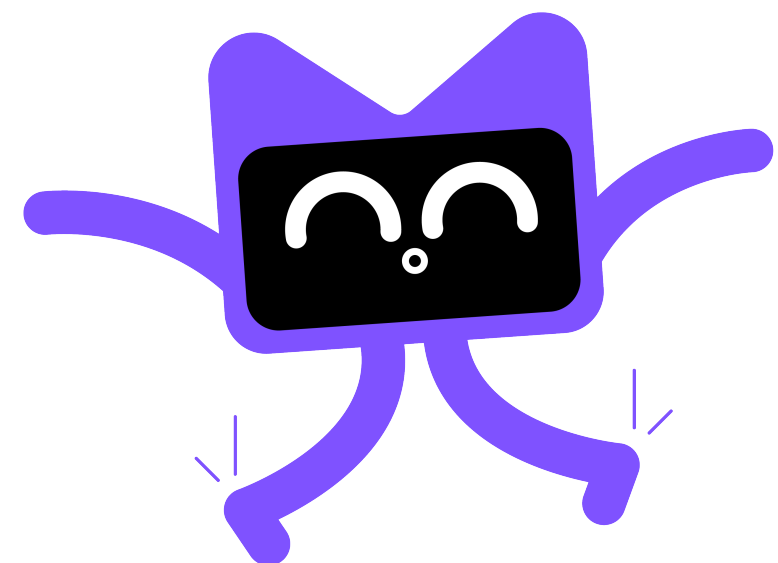
ToolRegistry

```
val registry = ToolRegistry {  
    tool(::addDate)  
}
```



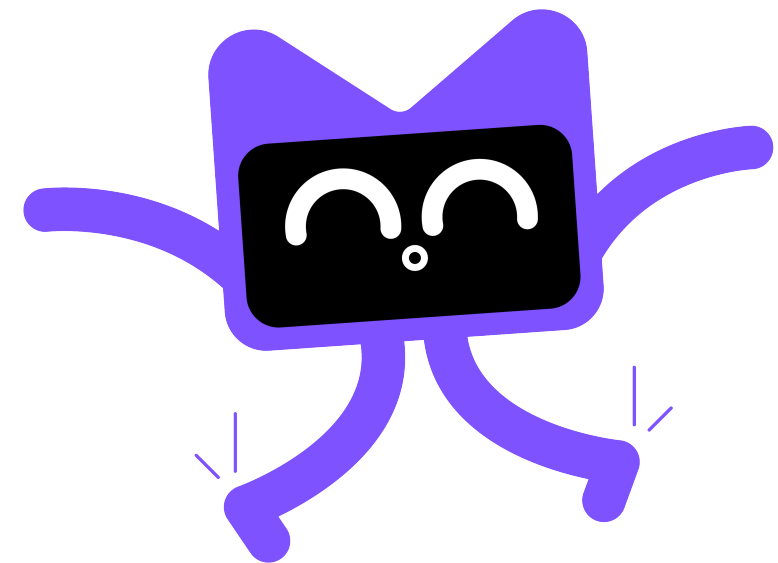
LLMPromptExecutor

```
val executor = MultiLLMPromptExecutor(  
    LLMProvider.OpenAI to openAIClient,  
    LLMProvider.Anthropic to anthropicClient,  
    LLMProvider.Google to googleClient  
)
```

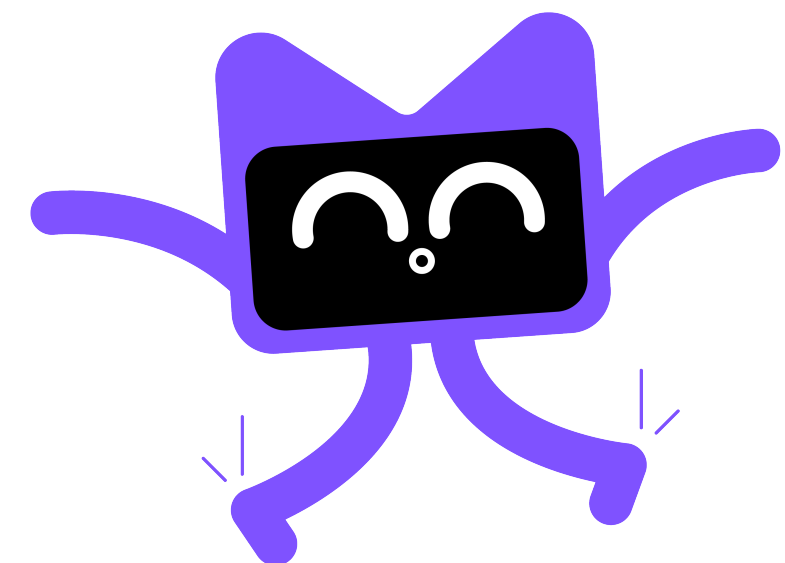
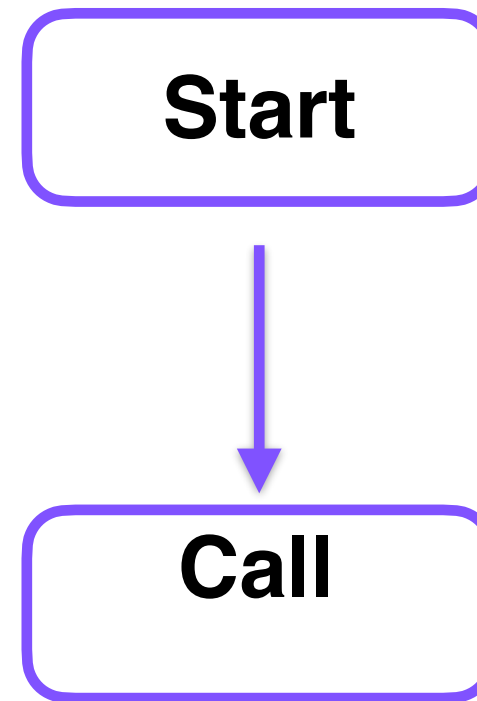


What is an Agent?

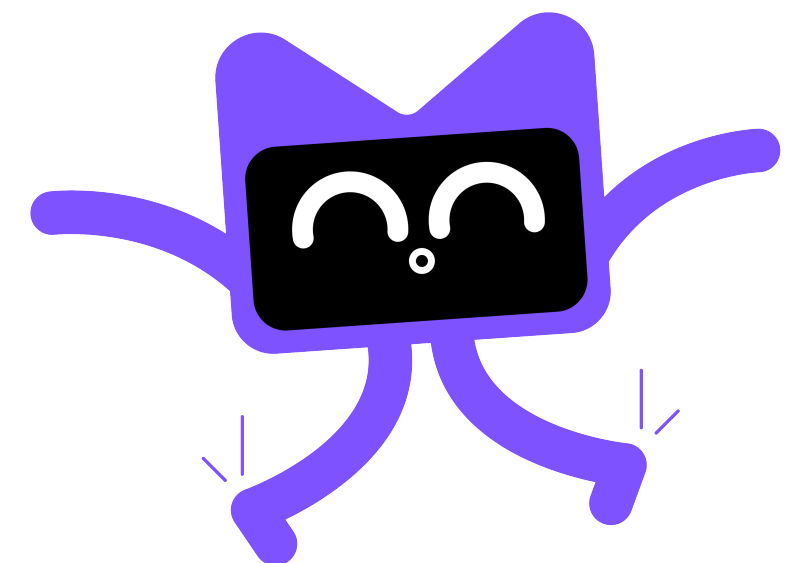
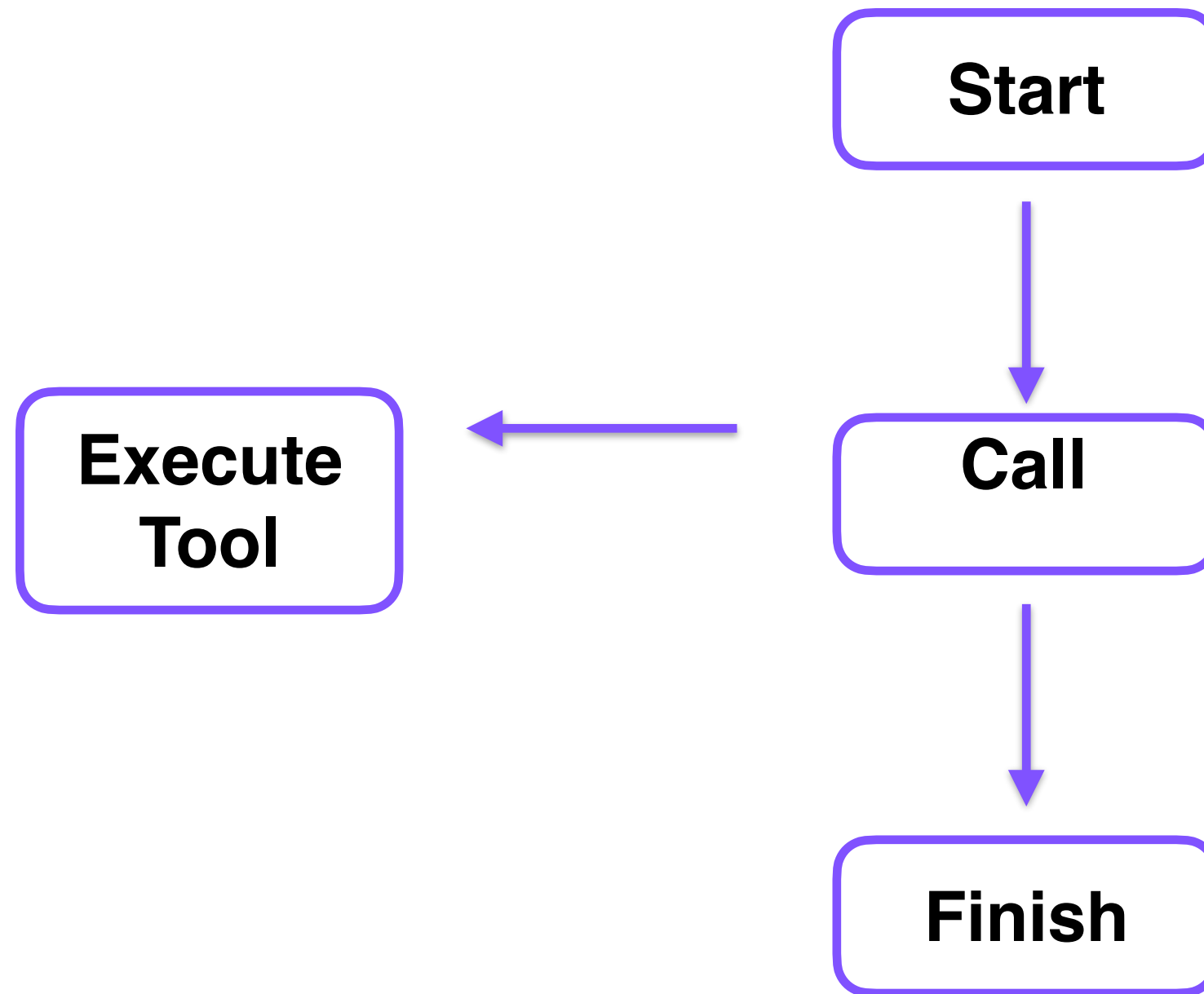
Start



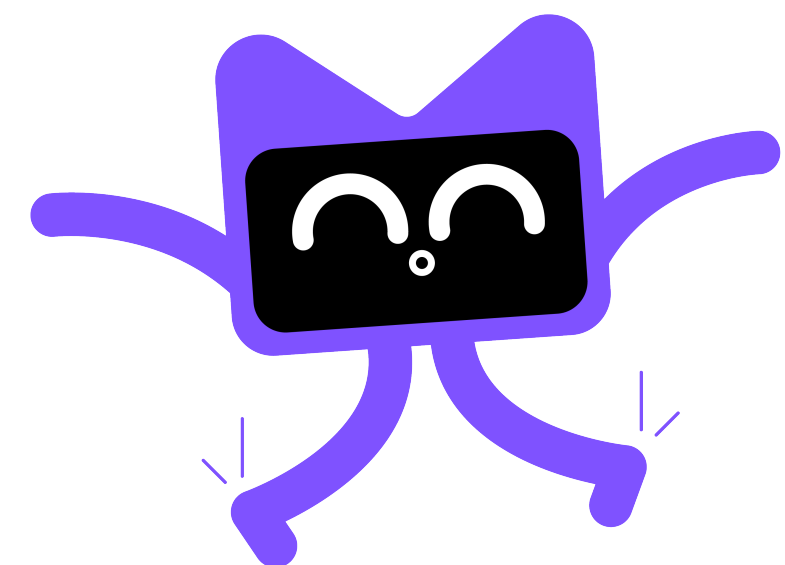
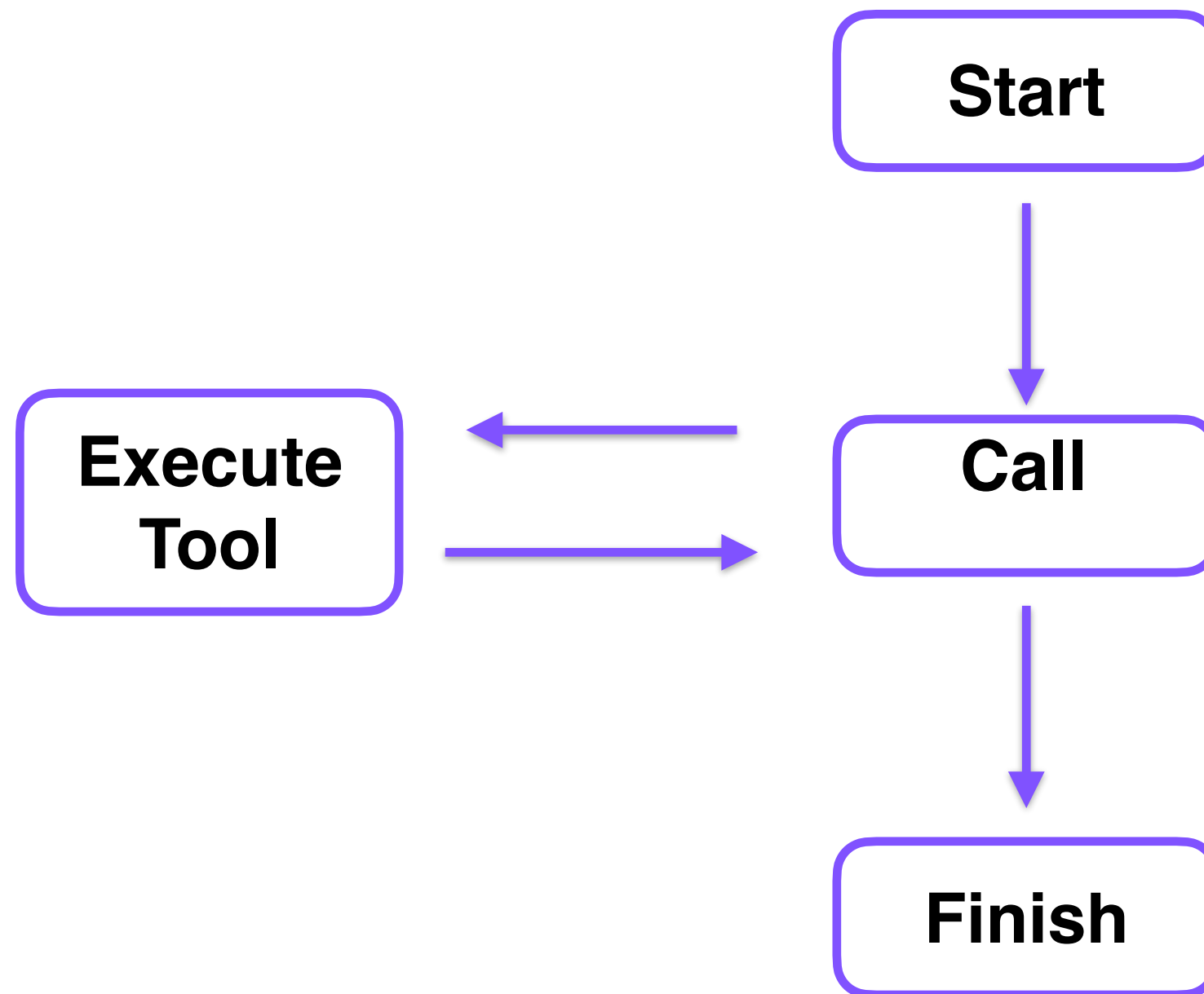
What is an Agent?



What is an Agent?

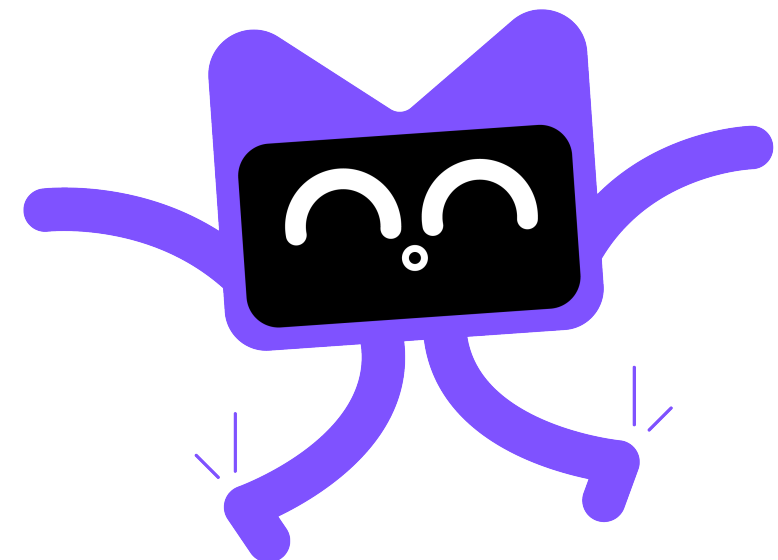


What is an Agent?



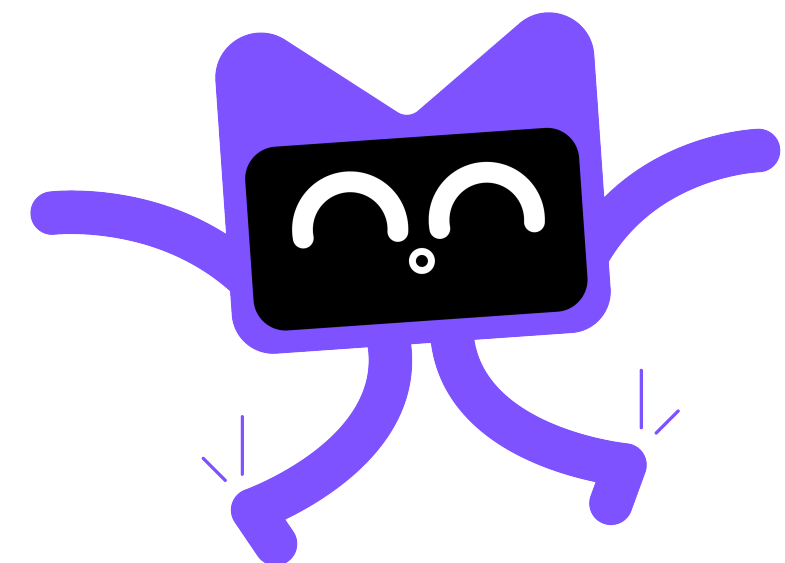
First agent!

```
AIAgent(  
    executor = executor,  
    llmModel = OpenAIModels.Chat.GPT5,  
    systemPrompt = "You're a helpful assistant.",  
    toolRegistry = ToolRegistry { tool(WeatherTool()) },  
    strategy = singleRunStrategy(),  
).run("What is the weather like in Amsterdam?")
```



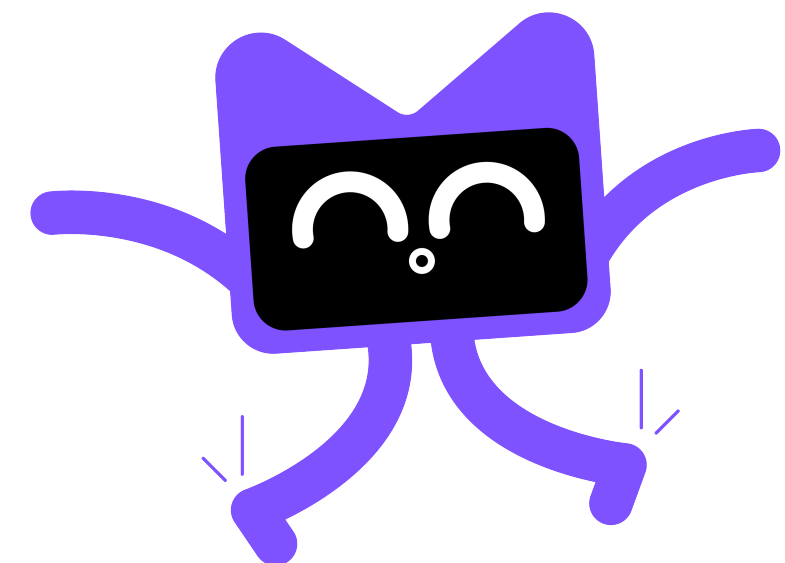
Environment aware tools

```
class WeatherTool(  
    private val client: HttpClient,  
    private val apiUrl: String  
)
```



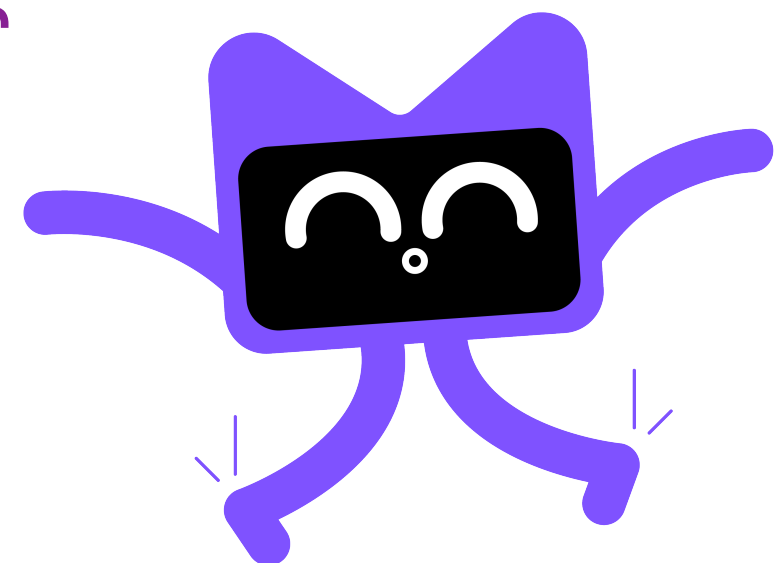
ToolSet

```
class WeatherTool(  
    private val client: HttpClient,  
    private val apiUrl: String  
): ToolSet {  
    @Tool  
    suspend fun weatherForLocation(  
        latitude: Double,  
        longitude: Double  
    ): WeatherToolResult = TODO()  
}
```



weatherForLocation

```
@Tool
suspend fun weatherForLocation(
    latitude: Double,
    longitude: Double
): WeatherToolResult = client.get(apiUrl) {
    parameter("latitude", latitude)
    parameter("longitude", longitude)
    parameter("current_weather", true)
    parameter("timezone", "auto")
}.body<WeatherApiResponse>().currentWeather
```



Error Handling?

Exception in thread "main" java.lang.RuntimeException: boom!

at

com.example.ApplicationKt\$main\$1.invokeSuspend\$lambda\$2(Application.kt:55)

at

kotlin.sequences.FlatteningSequence\$iterator\$1.ensureItemIterator(Sequences.kt:330)

at

kotlin.sequences.FlatteningSequence\$iterator\$1.hasNext(Sequences.kt:318)

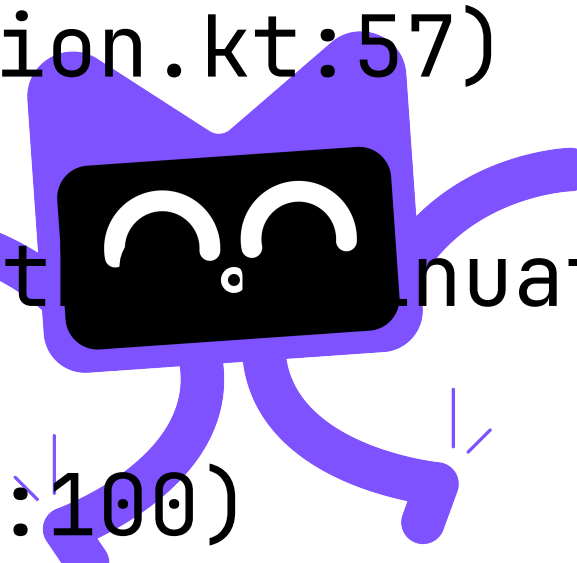
at kotlin.sequences.SequencesKt___SequencesKt.toList(_Sequences.kt:824)

at com.example.ApplicationKt\$main\$1.invokeSuspend(Application.kt:57)

at

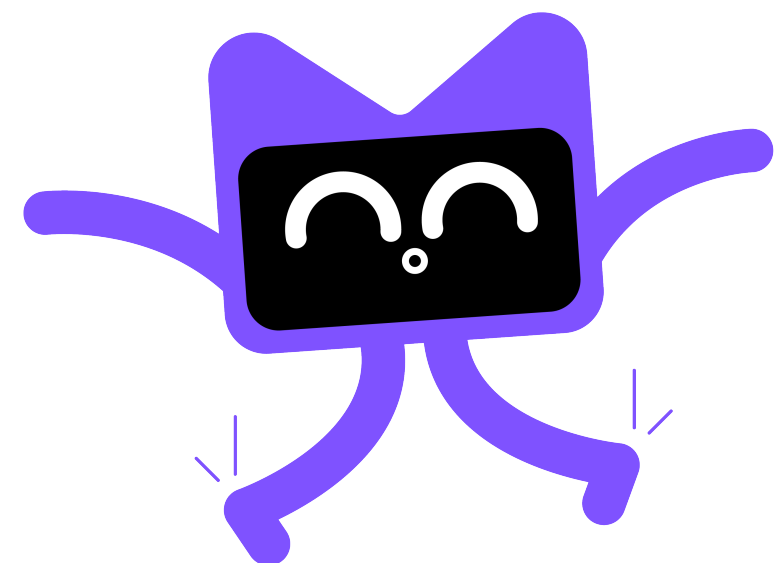
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)

at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:100)



Tailoring tool responses

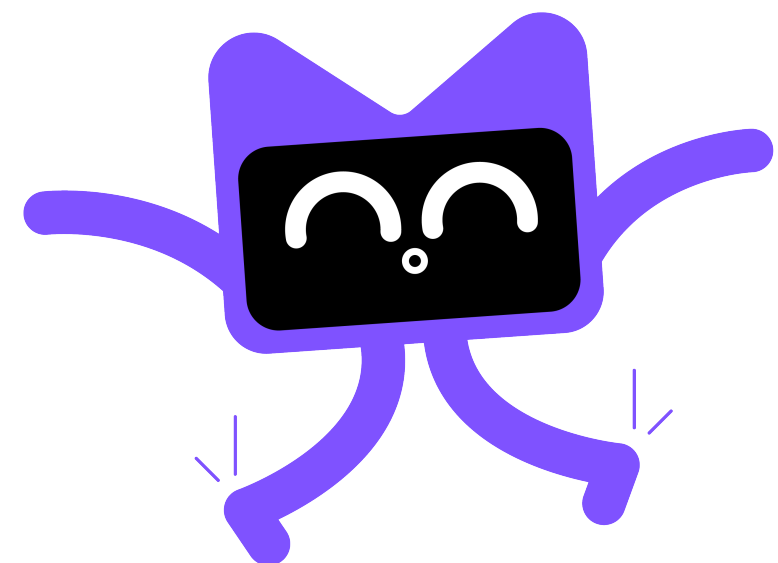
```
if (response.status.isSuccess()) {  
    response.body<WeatherApiResponse>().currentWeather  
} else {  
    val error = response.bodyAsText()  
    Text("An error occurred with the API request. error=$error")  
}
```



MCP SDK

```
val GATEWAY_URL = System.getenv("GATEWAY_URL")

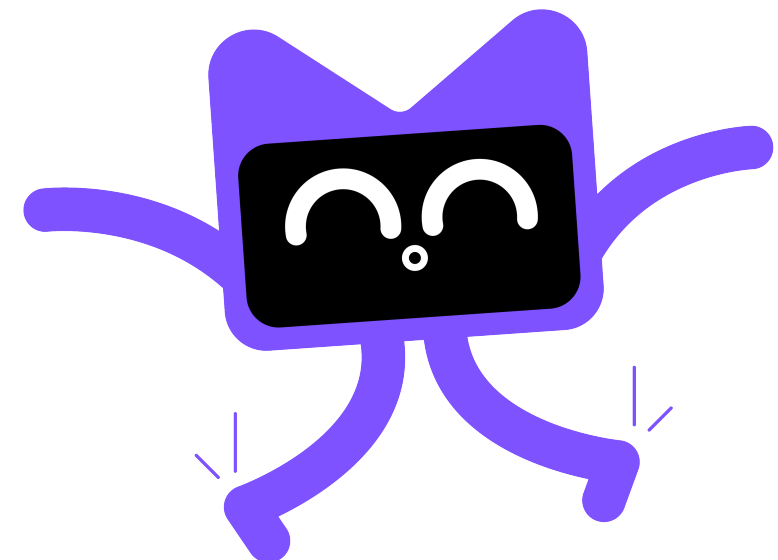
val transport = SseClientTransport(
    client = HttpClient(CIO) { install(SSE) },
    urlString = GATEWAY_URL
)
```



Koog MCP

```
val GATEWAY_URL = System.getenv("GATEWAY_URL")
```

```
val transport = McpToolRegistryProvider  
    .defaultSseTransport(GATEWAY_URL)
```

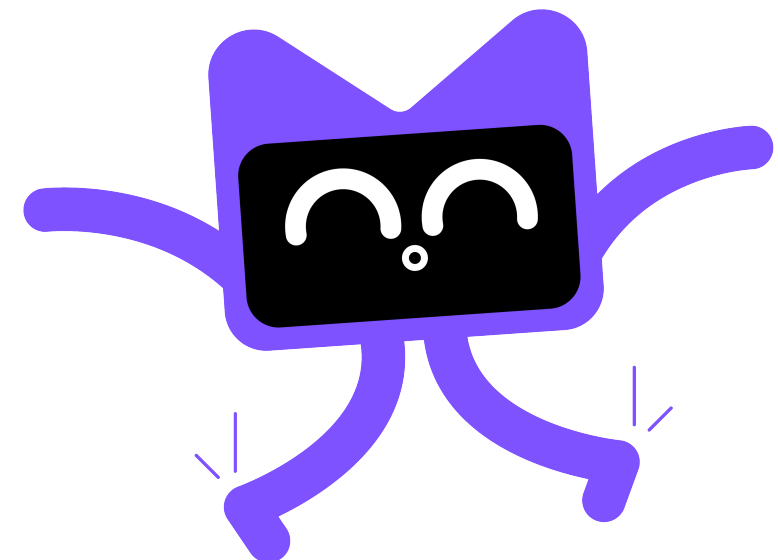


Koog MCP ToolRegistry

```
val GATEWAY_URL = System.getenv("GATEWAY_URL")
```

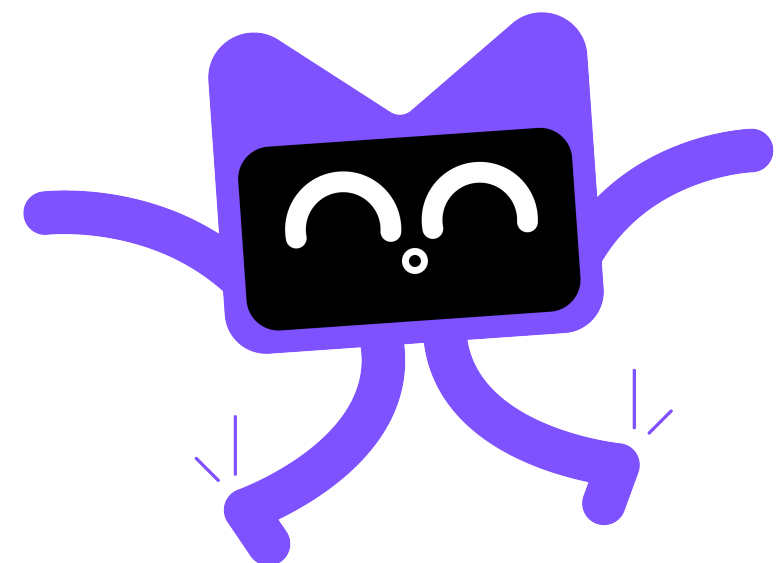
```
val transport = McpToolRegistryProvider  
    .defaultSseTransport(GATEWAY_URL)
```

```
suspend fun toolRegistry(transport: Transport): ToolRegistry =  
    McpToolRegistryProvider.fromTransport(transport)
```

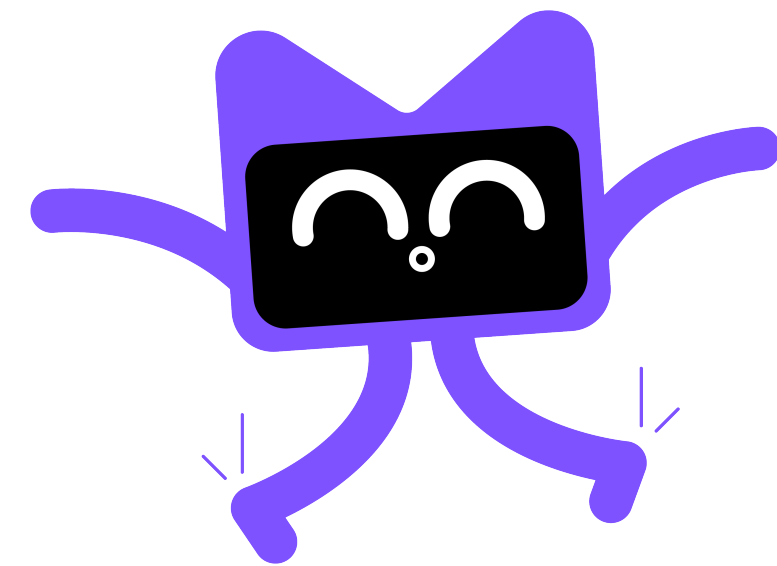


Calling tools!

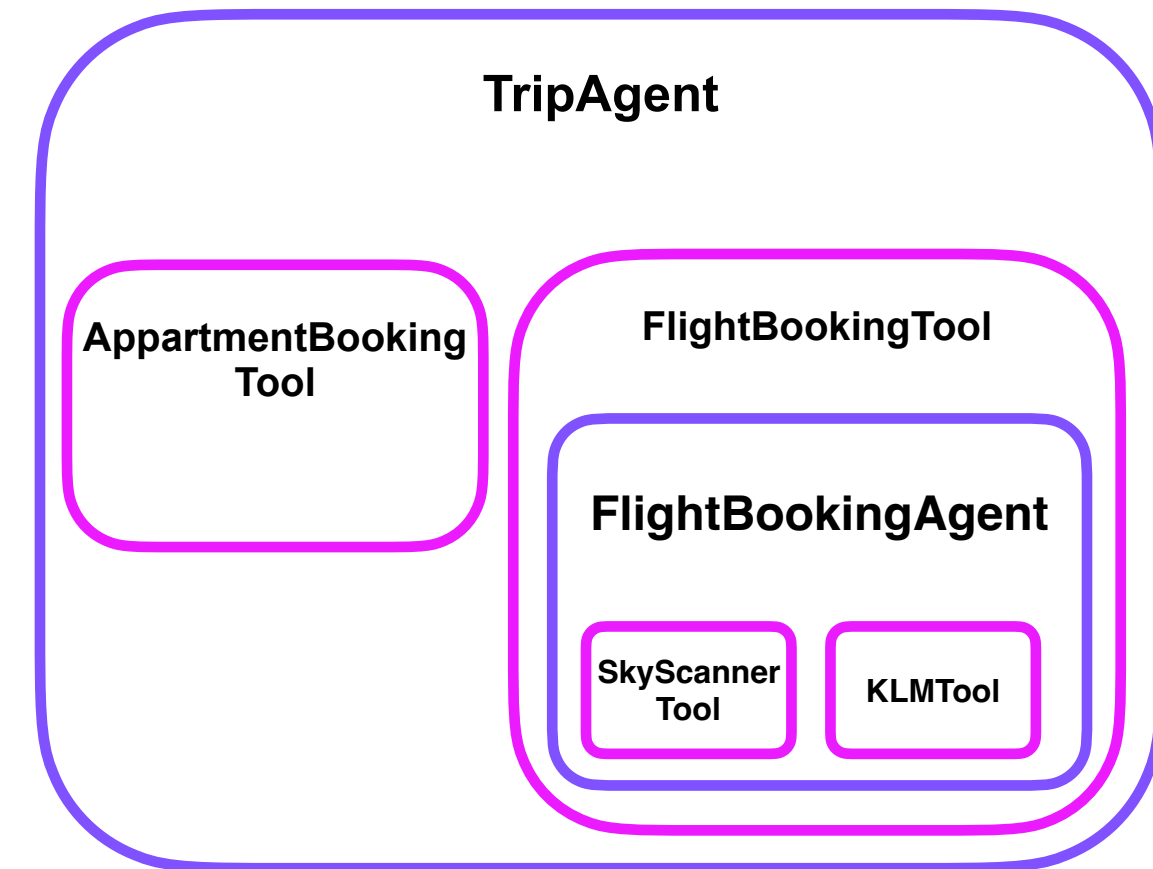
```
val result = AIAgent(  
    executor = executor,  
    llmModel = OpenAIModels.Chat.GPT4o,  
    toolRegistry = allTools  
).run(prompt)
```



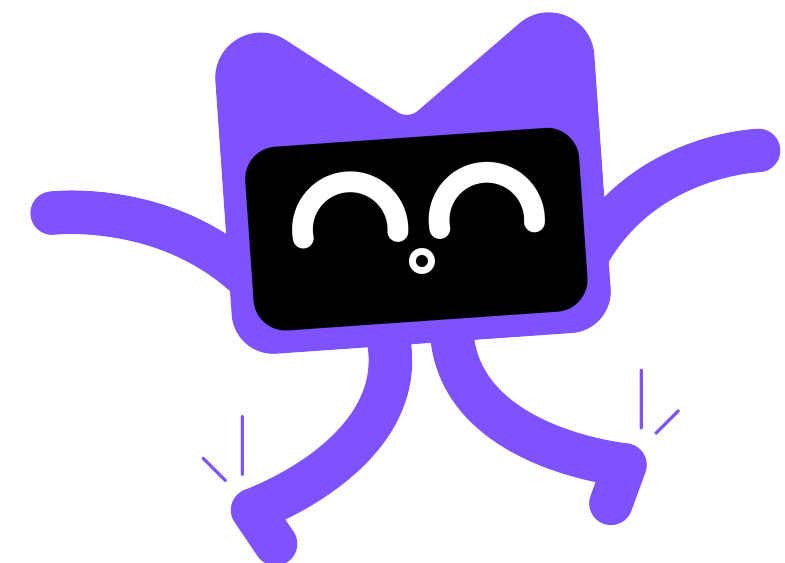
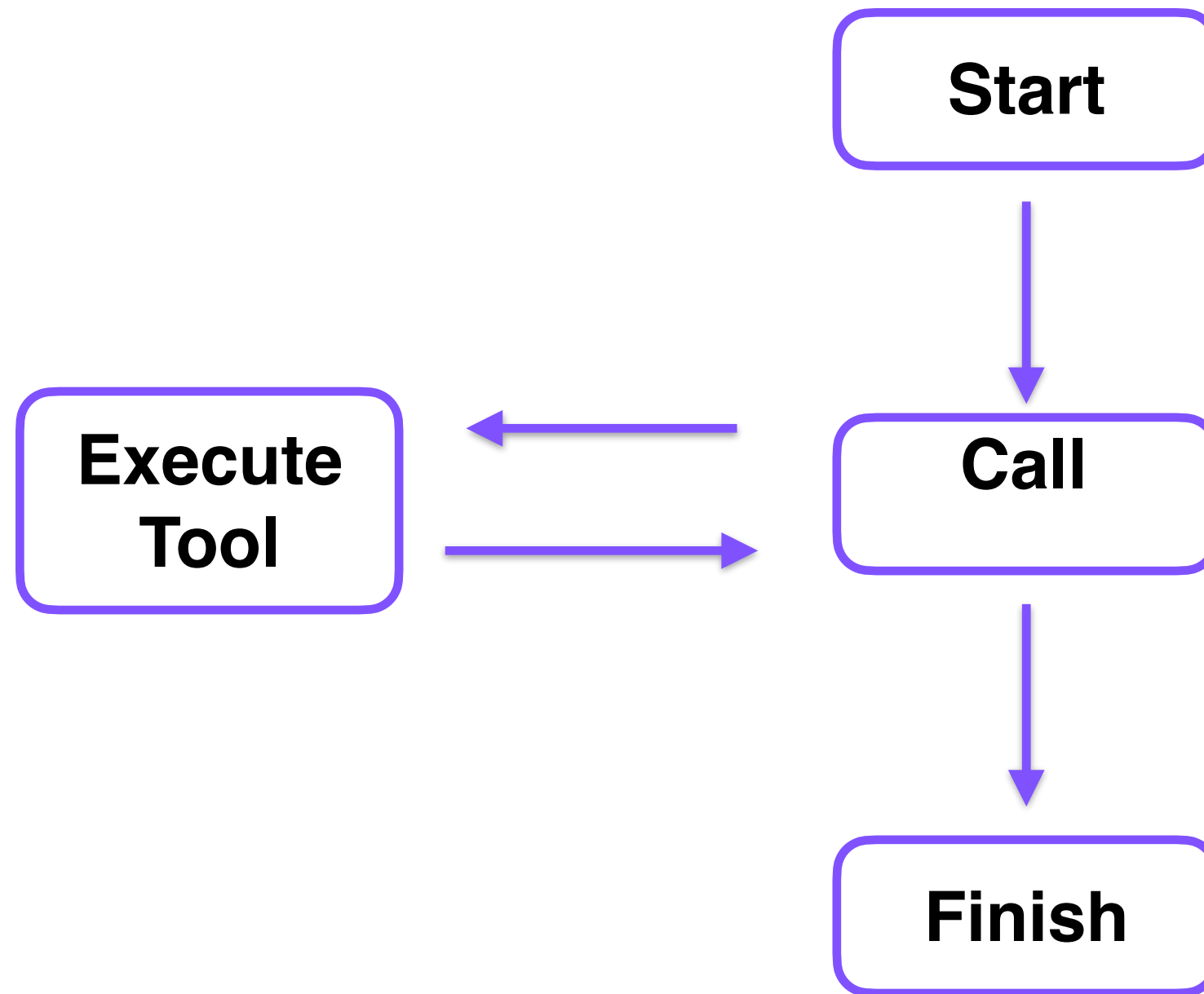
Agent as Tool (A2A*)



```
val flightBookingAgent = AIAgent(  
    executor = executor,  
    llmModel = OpenAIModels.Chat.GPT4o,  
    systemPrompt = "You are a helpful assistant for flight booking.",  
    strategy = strategy("flight-booking") { /* Some smart strategy here */ }  
    toolRegistry = ToolRegistry { tools(SkyScannerTool(), KLMTTool()) }  
)  
  
val flightBookingTool = flightsBookingAgent.asTool(  
    agentName = "book_flight",  
    agentDescription = "Books the most suitable flight for the requested trip",  
    inputDescriptor = ToolParameterDescriptor(  
        name = "request",  
        description = "Trip description",  
        type = ToolParameterType.String  
    )  
)  
  
val tripAgent = AIAgent(  
    executor = executor,  
    llmModel = OpenAIModels.Chat.GPT4o,  
    systemPrompt = "You are a helpful assistant for trip planning.",  
    toolRegistry = ToolRegistry { tool(flightBookingTool), tool(appartmentBookingTool) }  
    strategy = strategy("trip") { /* Some smart strategy here */ }  
)
```



Strategy

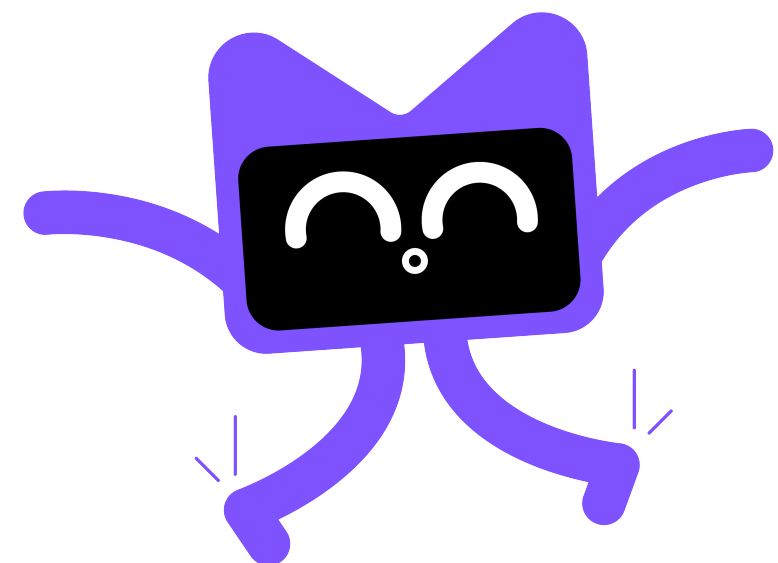


Strategy

Start

```
strategy<String, String>("single-run") {  
}
```

Finish



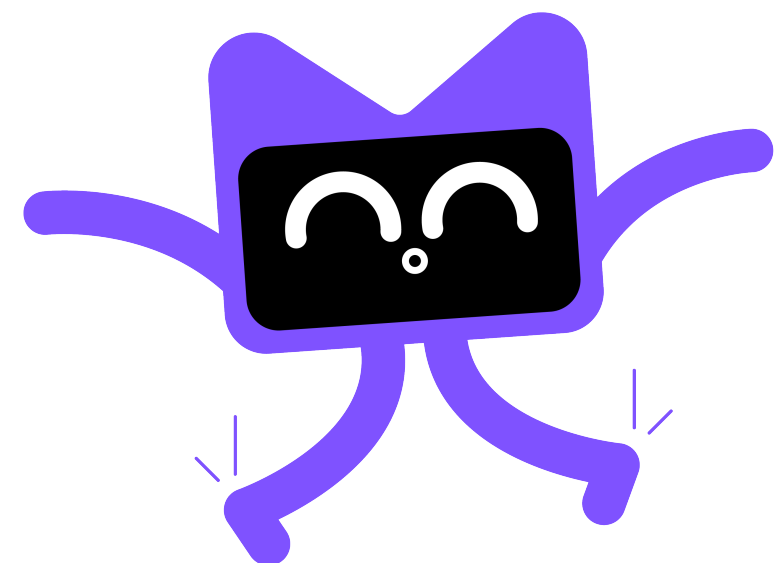
Strategy

```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
}
```

Start

Call

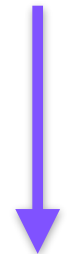
Finish



Strategy

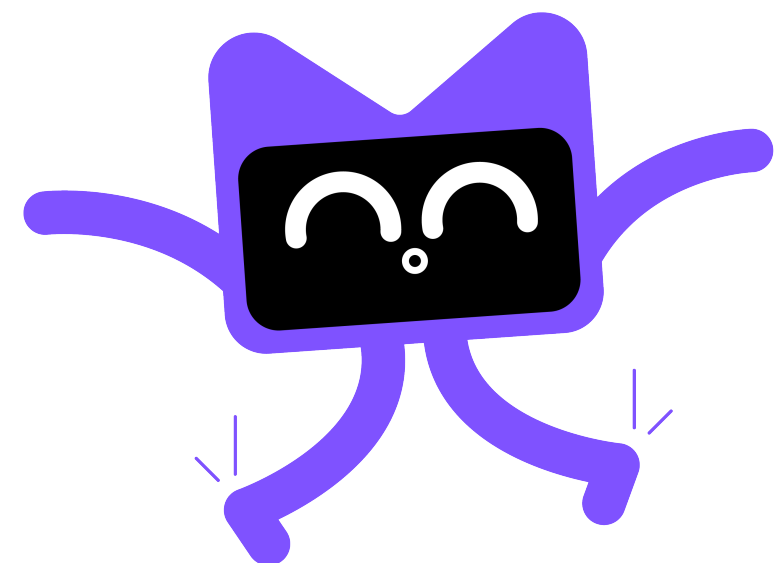
```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
  
    nodeStart then nodeCallLLM  
}
```

Start



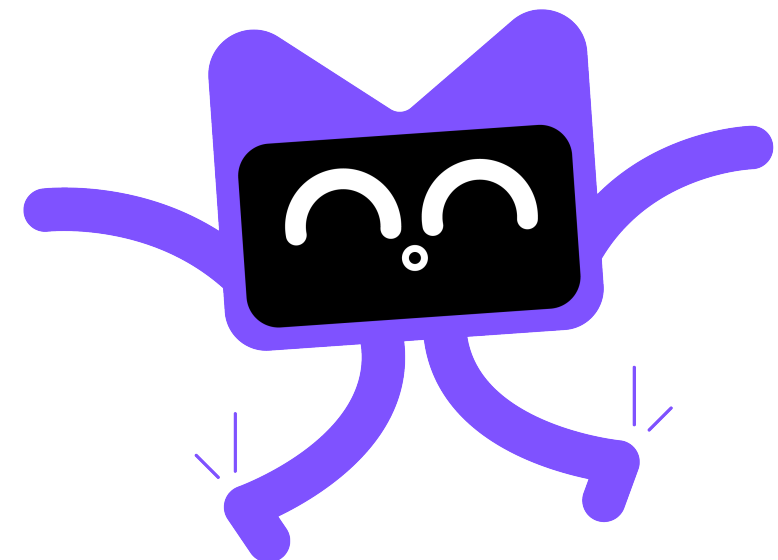
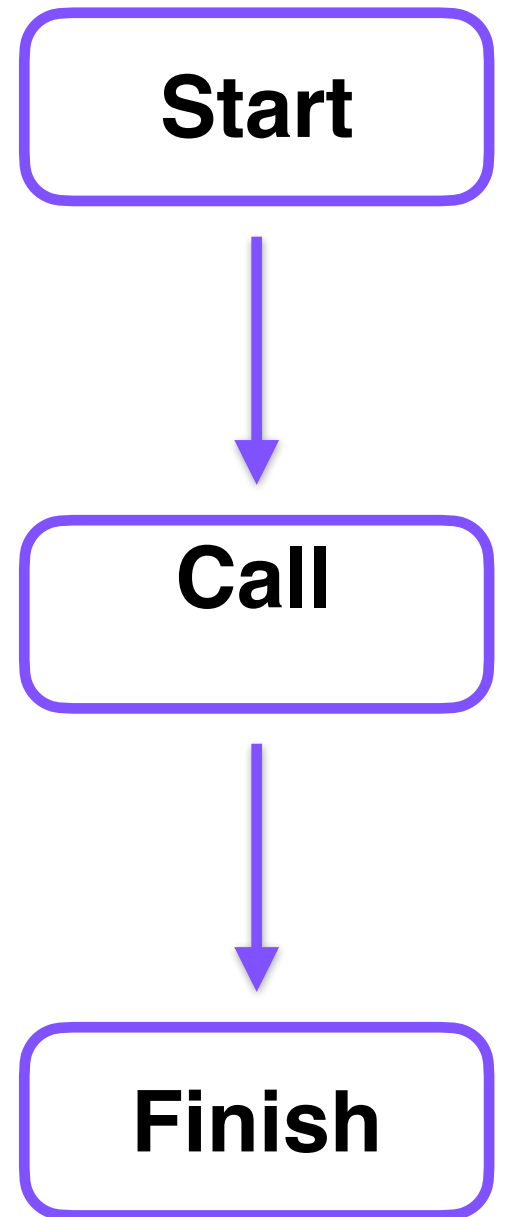
Call

Finish



Strategy

```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
  
    nodeStart then nodeCallLLM  
    edge(nodeCallLLM forwardTo nodeFinish onAssistantMessage { true })  
}
```

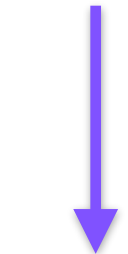


Strategy

```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
    val nodeExecuteTool by nodeExecuteTool()  
  
    nodeStart then nodeCallLLM  
    edge(nodeCallLLM forwardTo nodeFinish onAssistantMessage { true })  
}
```

**Execute
Tool**

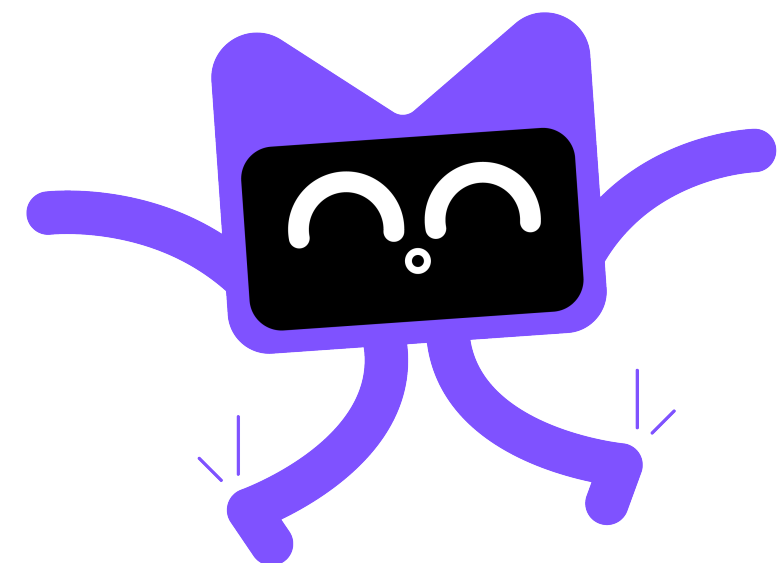
Start



Call

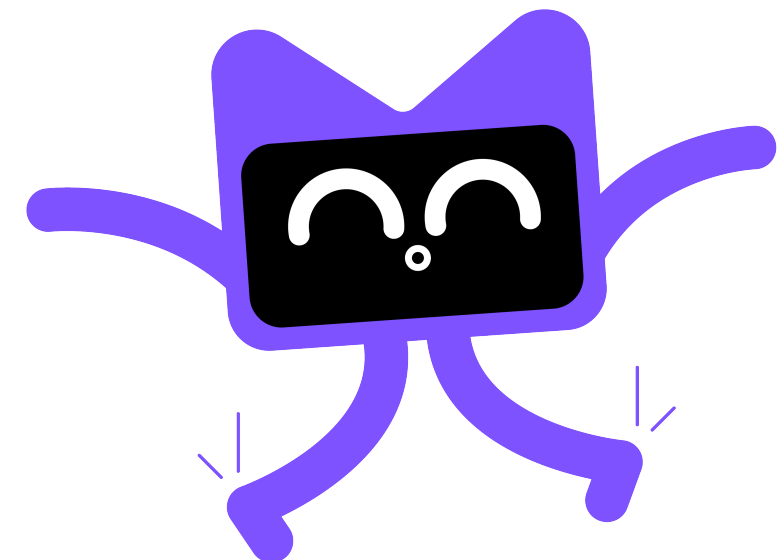
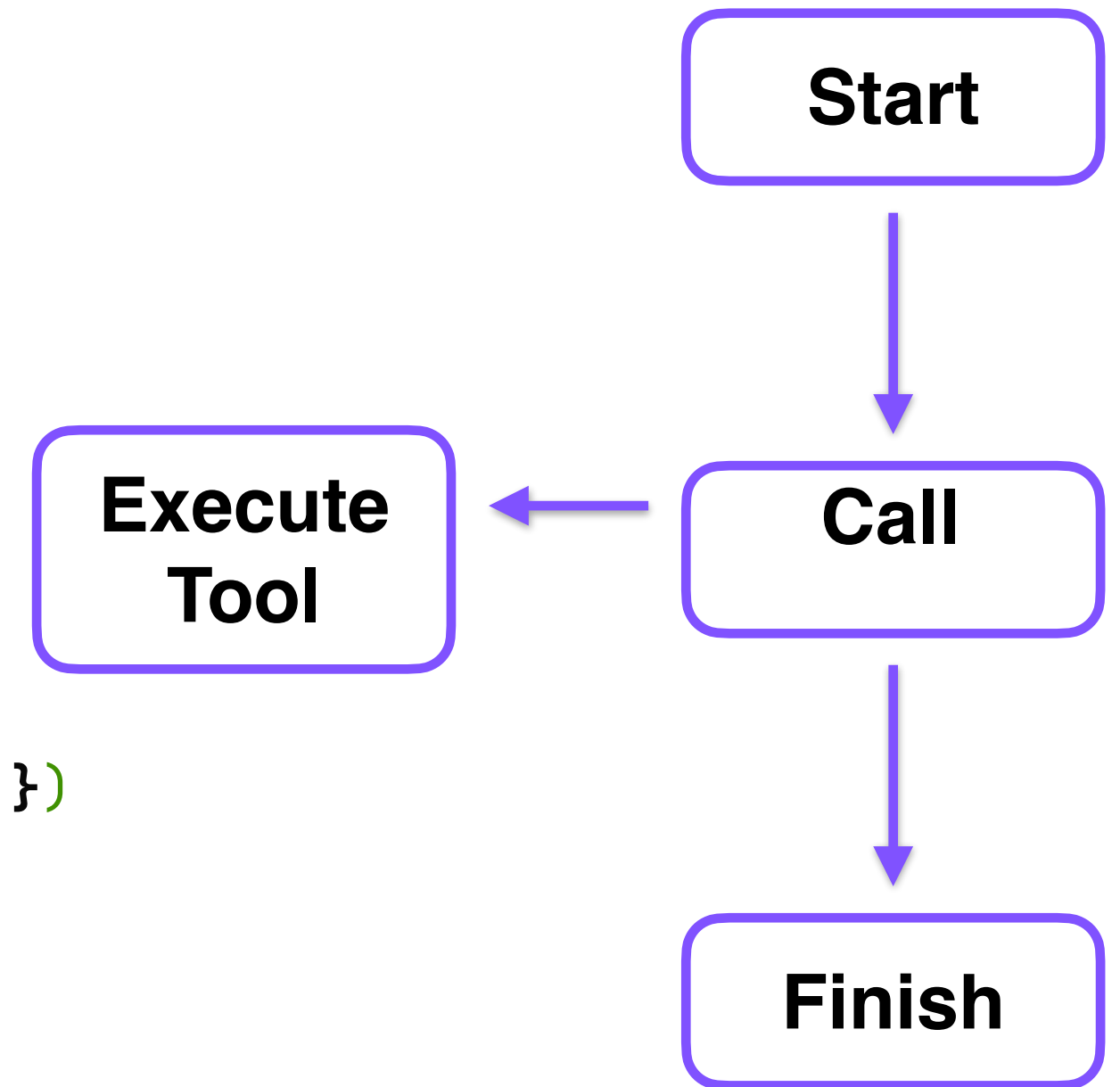


Finish



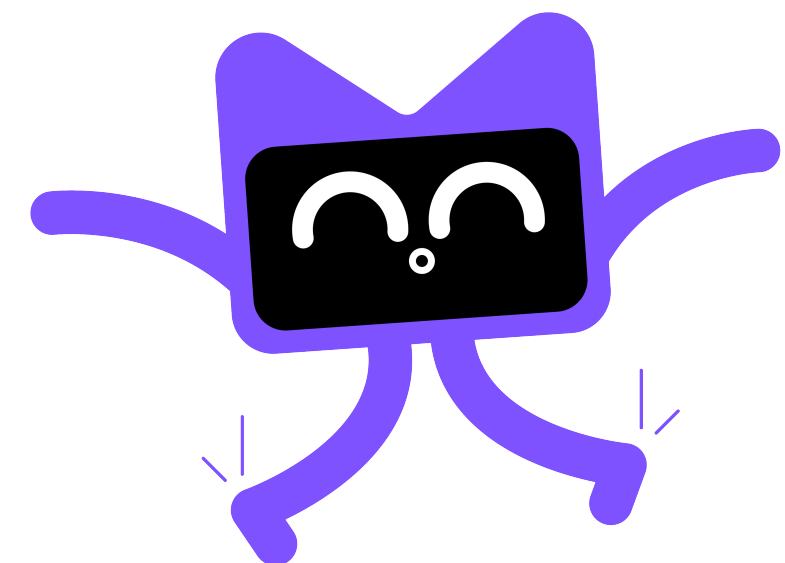
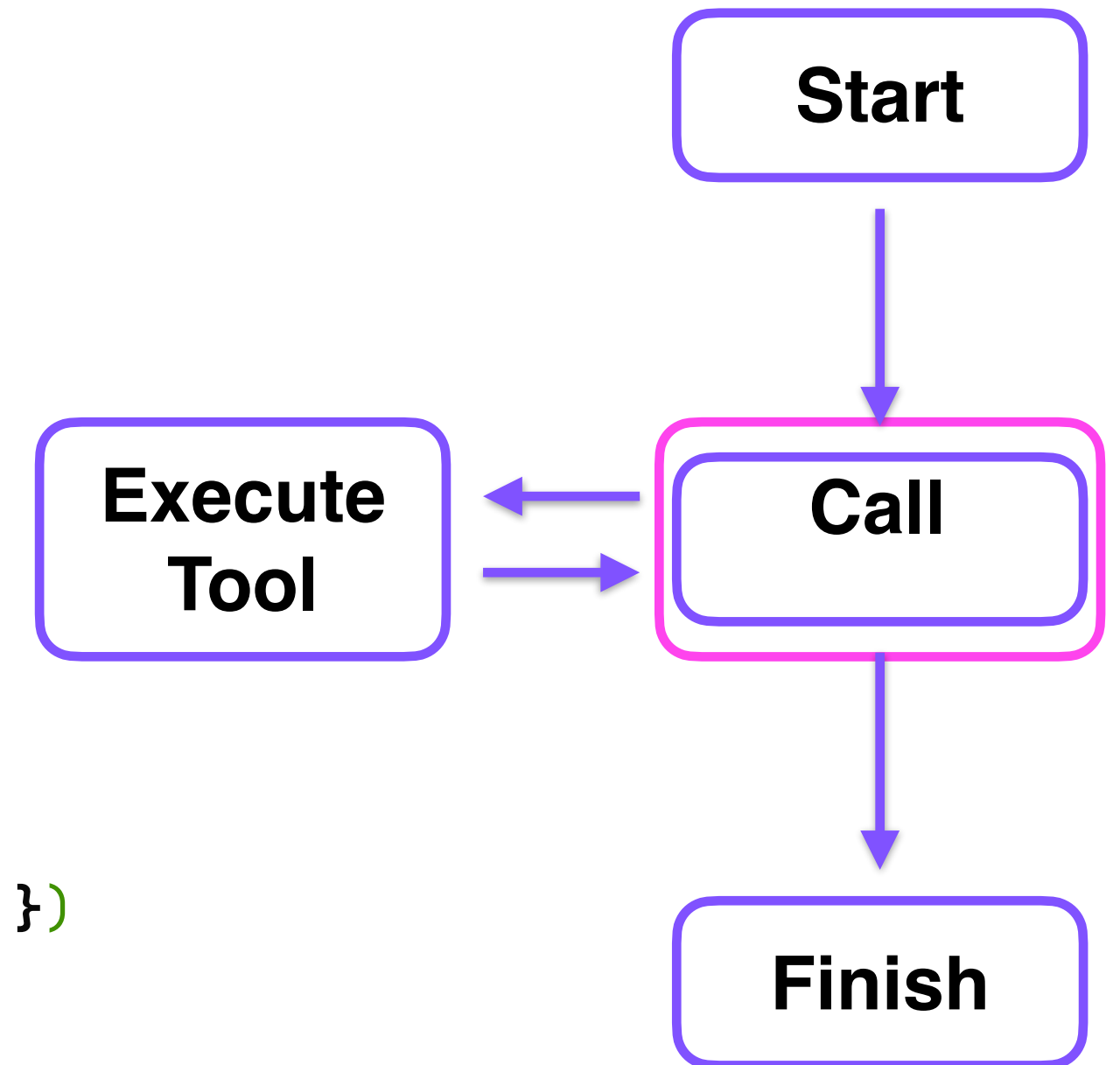
Strategy

```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
    val nodeExecuteTool by nodeExecuteTool()  
  
    nodeStart then nodeCallLLM  
    edge(nodeCallLLM forwardTo nodeFinish onAssistantMessage { true })  
  
    edge(nodeCallLLM forwardTo nodeExecuteTool onToolCall { true })  
}
```



Strategy

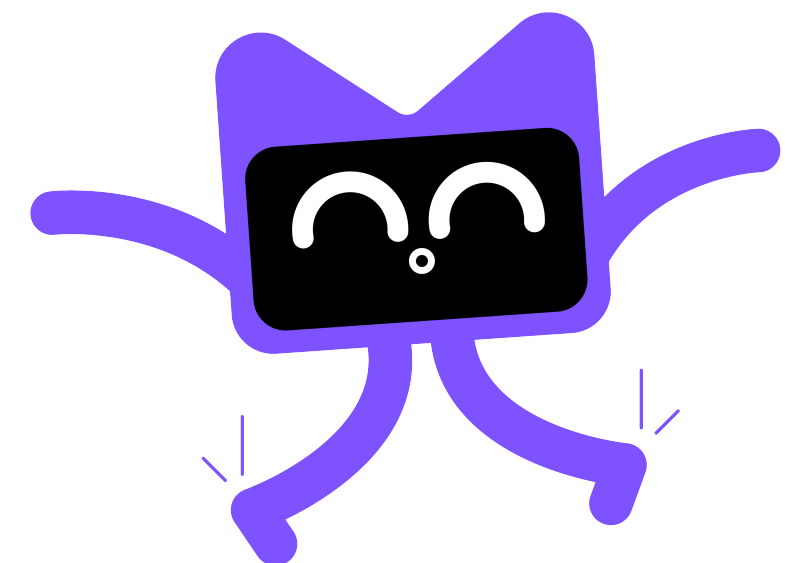
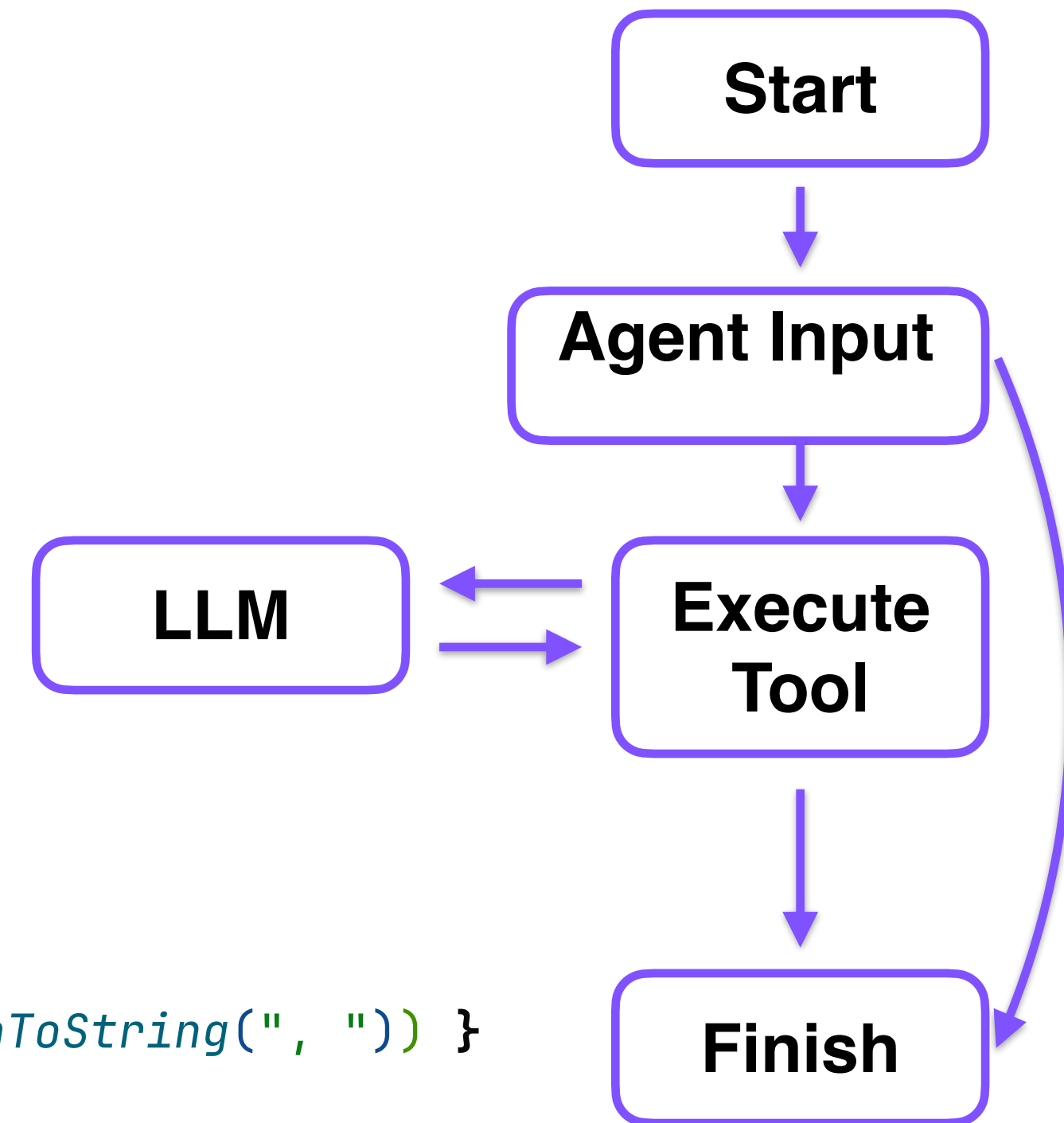
```
strategy<String, String>("single-run") {  
    val nodeCallLLM by nodeLLMRequest()  
    val nodeExecuteTool by nodeExecuteTool()  
    val nodeSendToolResult by nodeLLMSendToolResult()  
  
    nodeStart then nodeCallLLM  
  
    edge(nodeCallLLM forwardTo nodeFinish onAssistantMessage { true })  
    edge(nodeCallLLM forwardTo nodeExecuteTool onToolCall { true })  
  
    nodeExecuteTool then nodeSendToolResult  
    edge(nodeSendToolResult forwardTo nodeFinish onAssistantMessage { true })  
    edge(nodeSendToolResult forwardTo nodeExecuteTool onToolCall { true })  
}
```



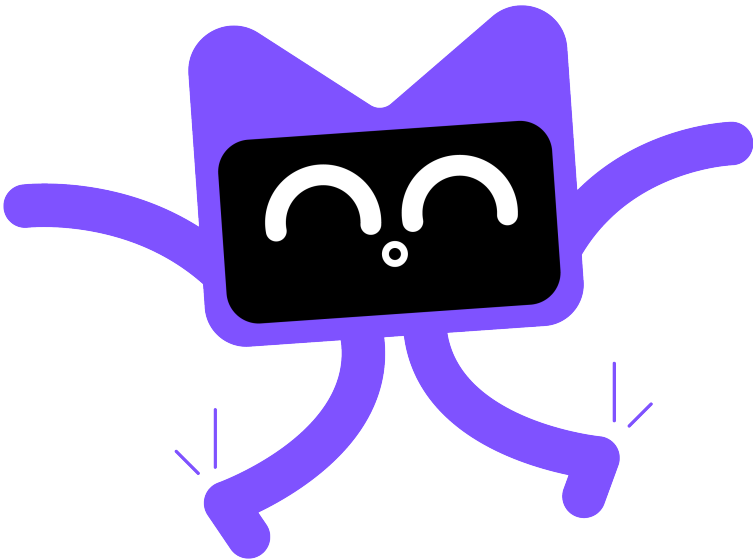
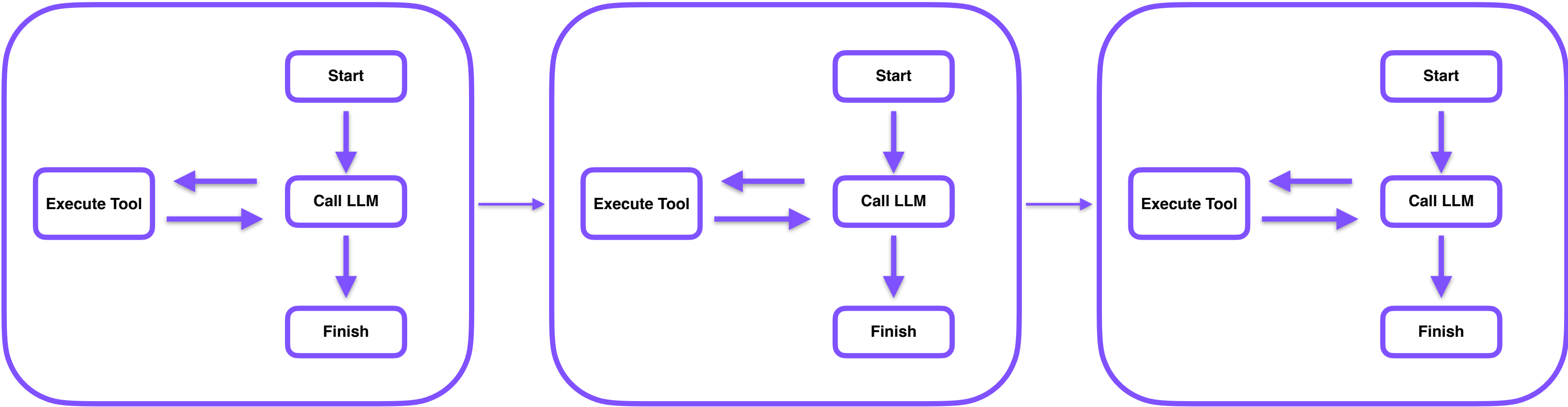
Custom strategy & Node

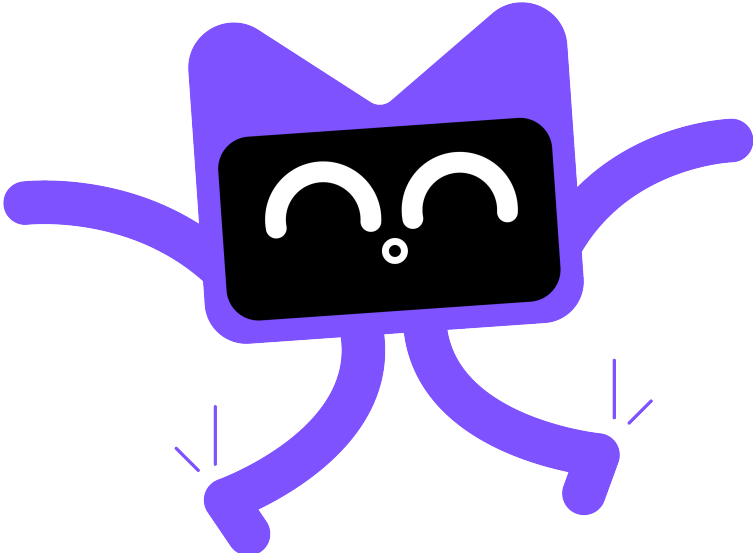
```
data class Form(  
    val name: String,  
    val preferences: List<String>,  
    val request: String  
)  
  
strategy<Form, String>("plan") {  
    val input by node<Form, Message.Response> { form →  
        llm.writeSession {  
            xml {  
                tag("user-question") {  
                    tag("name") { text(form.name) }  
                    tag("preferences") { text(form.preferences.joinToString(", ")) }  
                    tag("request") { text(form.request) }  
                }  
            }  
        }  
        requestLLM()  
    }  
}
```

// Almost same as before



Subgraphs



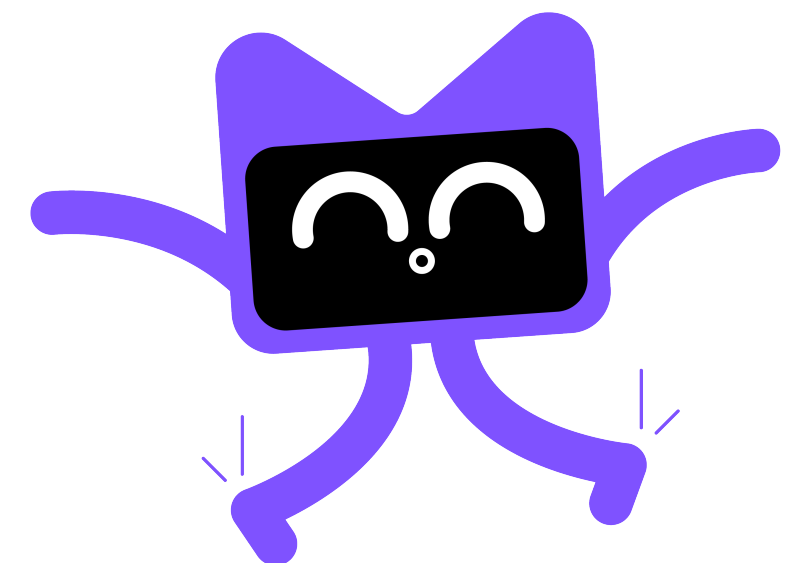


Custom strategy & Node

```
strategy<Form, String>("other") {
    val plan by subgraph<Form, String>(
        toolSelectionStrategy = googleMapsAndWeather(),
        llmModel = GoogleModels.Gemini2_5Pro,
    ) {
        val input by node<Form, Message.Response> { form →
            llm.writeSession {
                xml {
                    tag("user-question") {
                        tag("name") { text(form.name) }
                        tag("preferences") { text(form.preferences.joinToString(", ")) }
                        tag("request") { text(form.request) }
                    }
                }
                requestLLM()
            }
        }

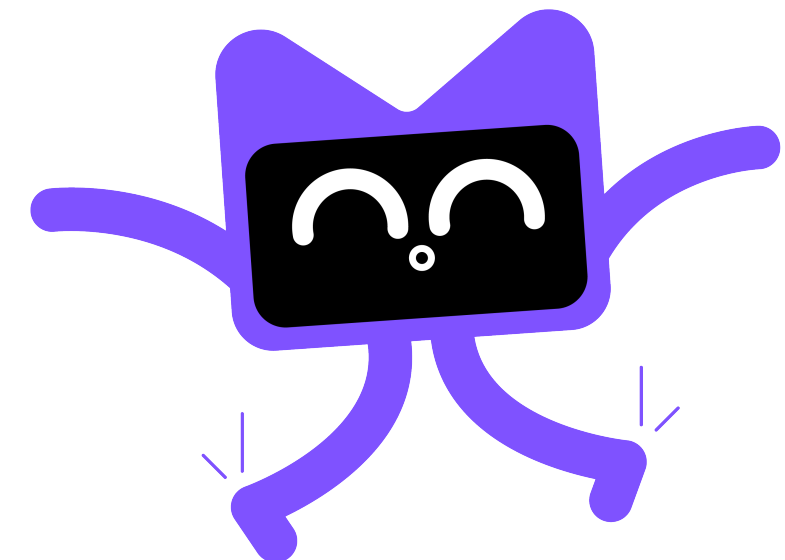
        val nodeExecuteTool by nodeExecuteTool()
        val nodeSendToolResult by nodeLLMSendToolResult()

        nodeStart then input
    }
}
```



Custom strategy & Node

```
strategy<Form, String>("big-strategy") {  
    val plan by subgraphWithTask<Form>(  
        toolSelectionStrategy = googleMapsAndWeather(),  
        llmModel = GoogleModels.Gemini2_5Pro,  
    ) { form →  
        xml {  
            tag("user-question") {  
                tag("name") { text(form.name) }  
                tag("preferences") { text(form.preferences.joinToString(", ")) }  
                tag("request") { text(form.request) }  
            }  
        }  
    }  
}  
  
nodeStart then plan  
edge(plan forwardTo nodeFinish transformed { it.result })  
}
```

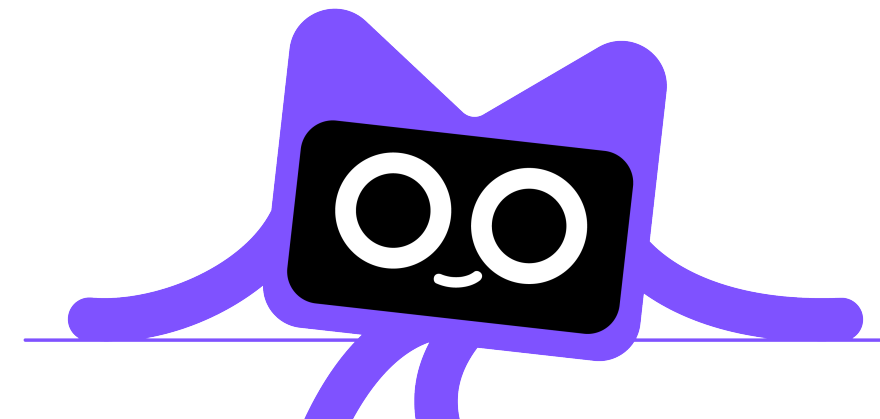
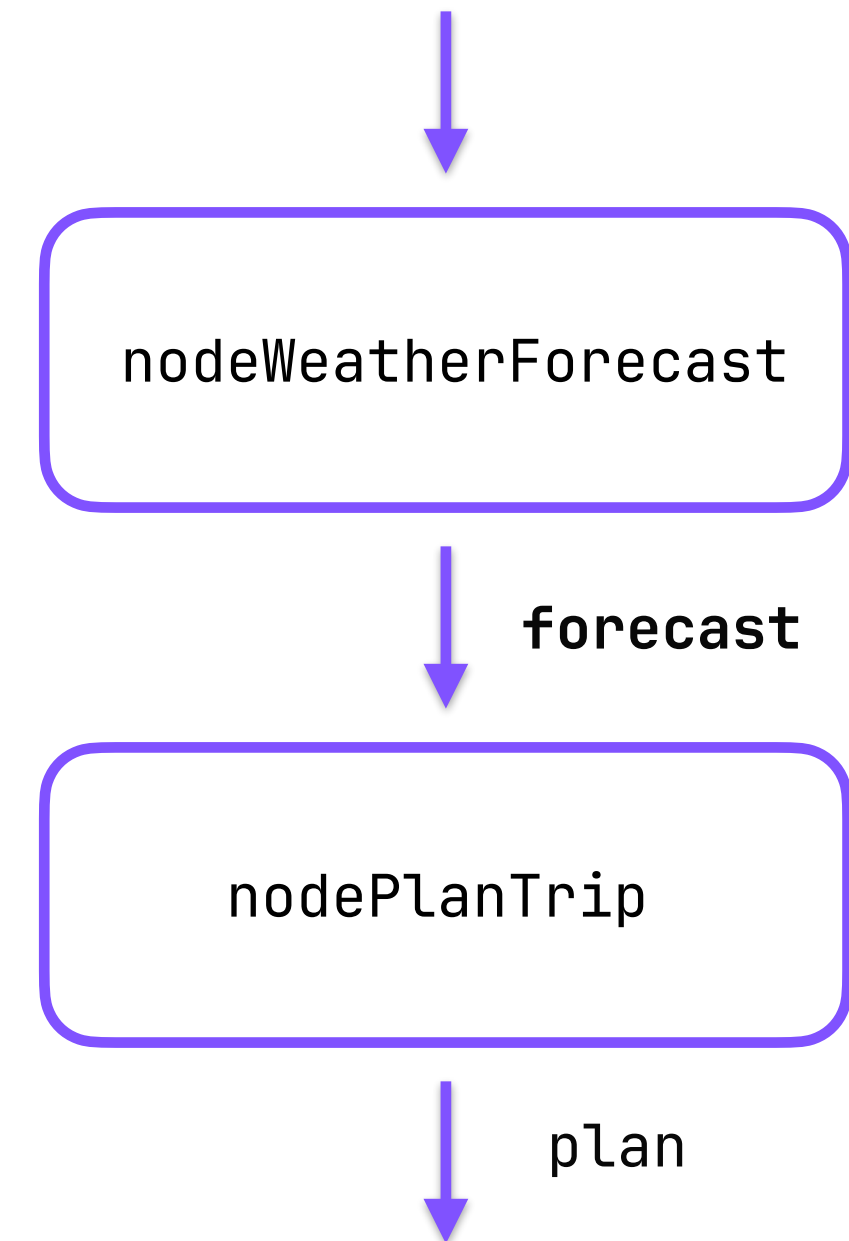


Node IO

```
val nodeGetLocalWeather by node<Unit, Weather> {  
    val forecast = weatherService.getWeekForecast()  
    forecast  
}
```

```
val nodePlanTrip by node<Weather, String> { forecast →  
    val tripPlan = llm.request(forecast)  
    tripPlan  
}
```

```
edge(nodeGetLocalWeather forwardTo nodePlanTrip)
```



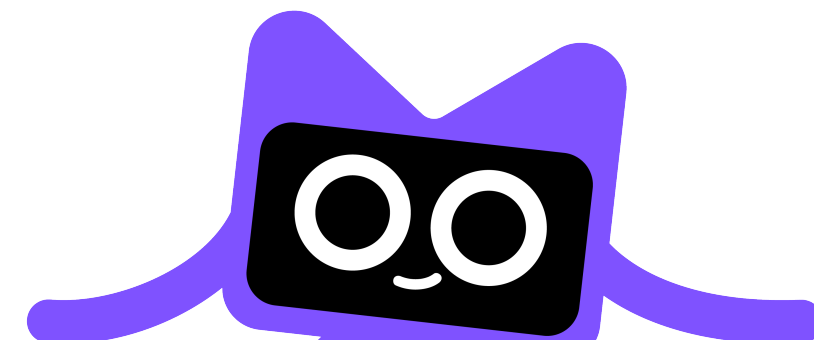
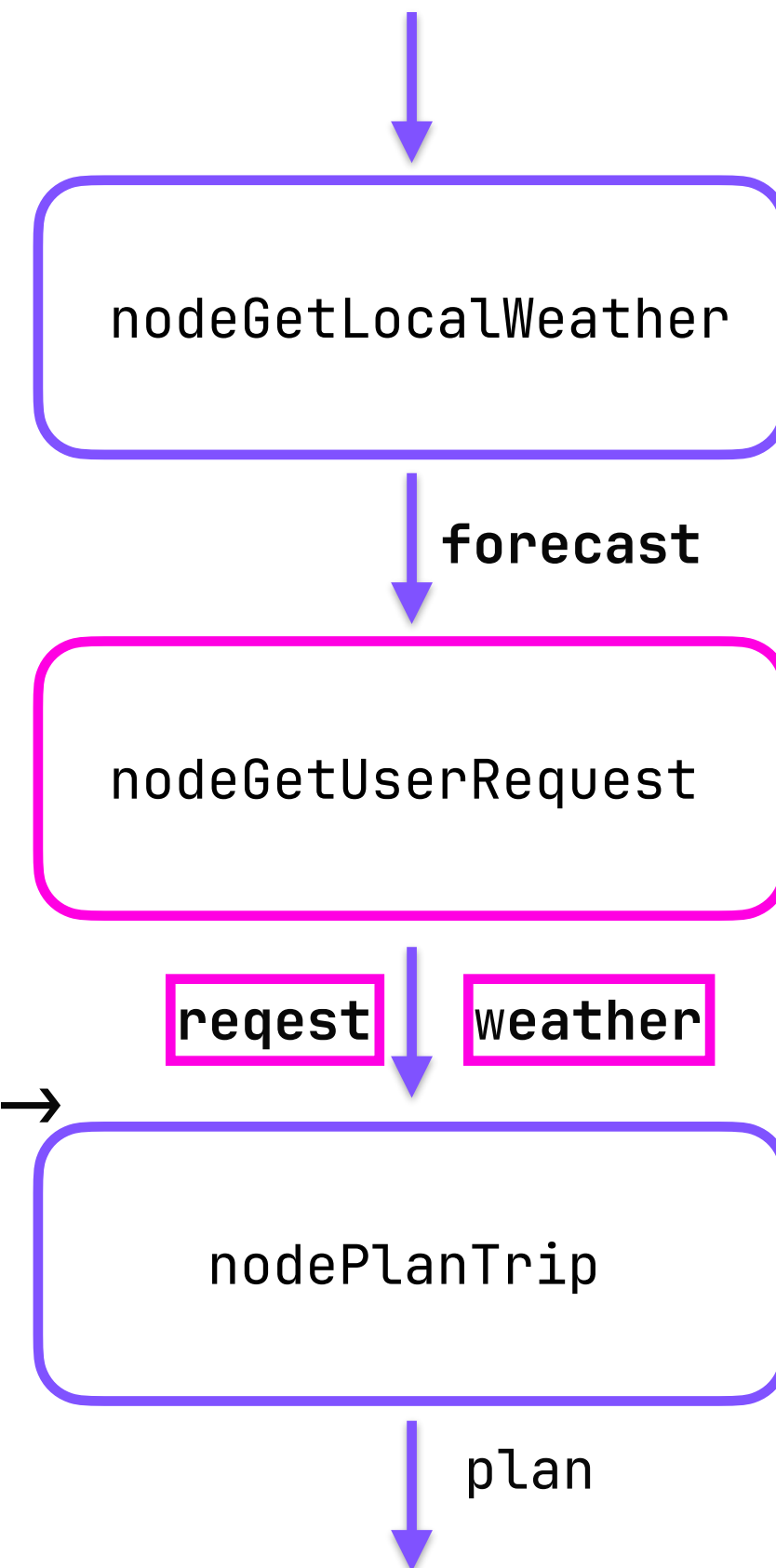
Node IO

```
val nodeGetLocalWeather by node<Unit, Weather> {  
    val weather = weatherService.getLocalForecast()  
    weather  
}
```

```
val nodeGetUserRequest by node<Weather, Pair<String, Weather>> { weather →  
    val request = readln()  
    request to weather  
}
```

```
val nodePlanTrip by node<Pair<String, Weather>, String> { (request, weather) →  
    val tripPlan = llm.writeSession {  
        updatePrompt { user(request + weather.toString()) }  
        requestLLM()  
    }  
    tripPlan.content  
}
```

```
edge(nodeGetLocalWeather forwardTo nodeGetUserRequest)  
edge(nodeGetUserRequest forwardTo nodePlanTrip)
```



Key-value storage

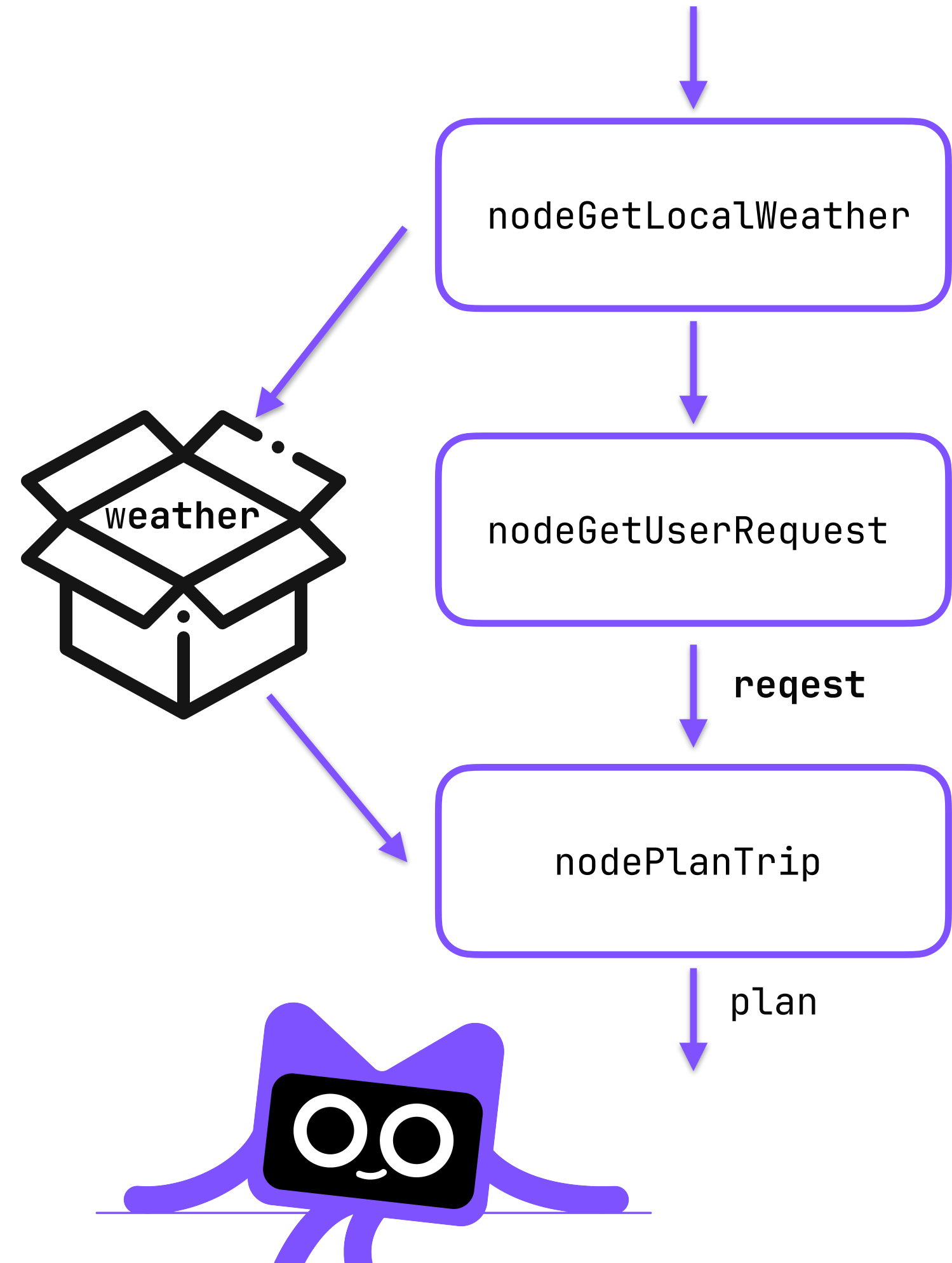
```
val weatherKey = createStorageKey<Weather>("weather")

val nodeGetLocalWeather by node<Unit, Unit> {
    val weather = weatherService.getLocalForecast()
    storage.set(weatherKey, weather)
}

val nodeGetUserRequest by node<Unit, String> {
    val request = readln()
    request
}

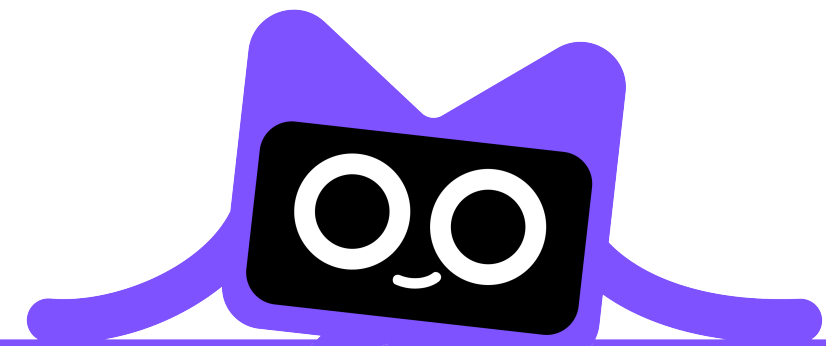
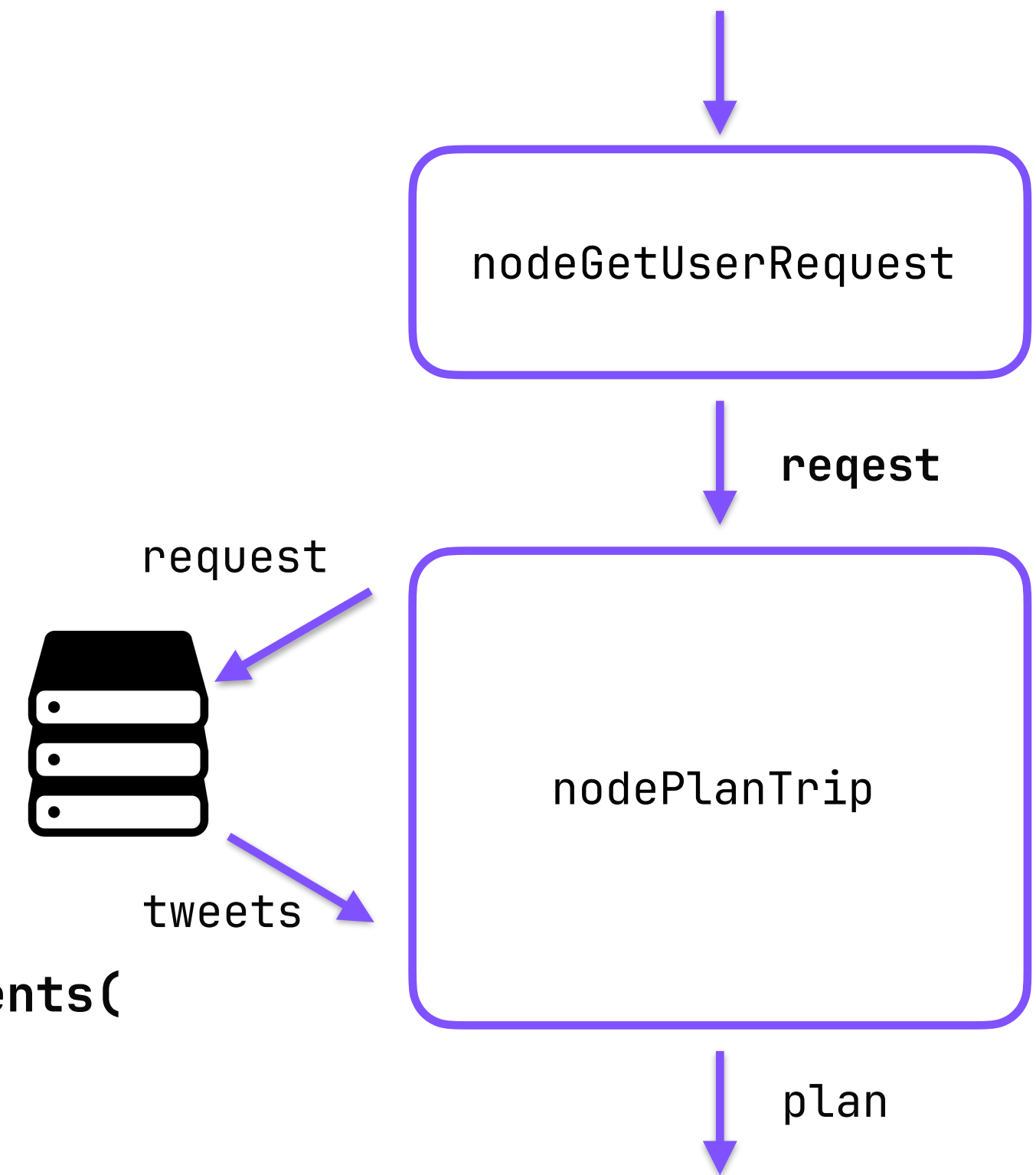
val nodePlanTrip by node<String, String> { request →
    val weather = storage.get(weatherKey)
    val tripPlan = llm.writeSession {
        updatePrompt { user(request + weather.toString()) }
        requestLLM()
    }
    tripPlan.content
}

edge(nodeGetLocalWeather forwardTo nodeGetUserRequest)
edge(nodeGetUserRequest forwardTo nodePlanTrip)
```



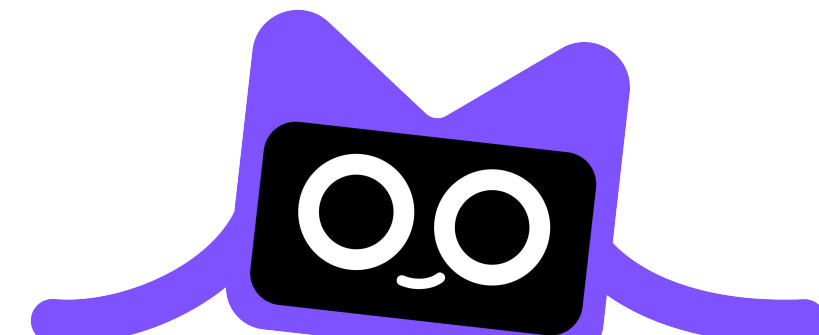
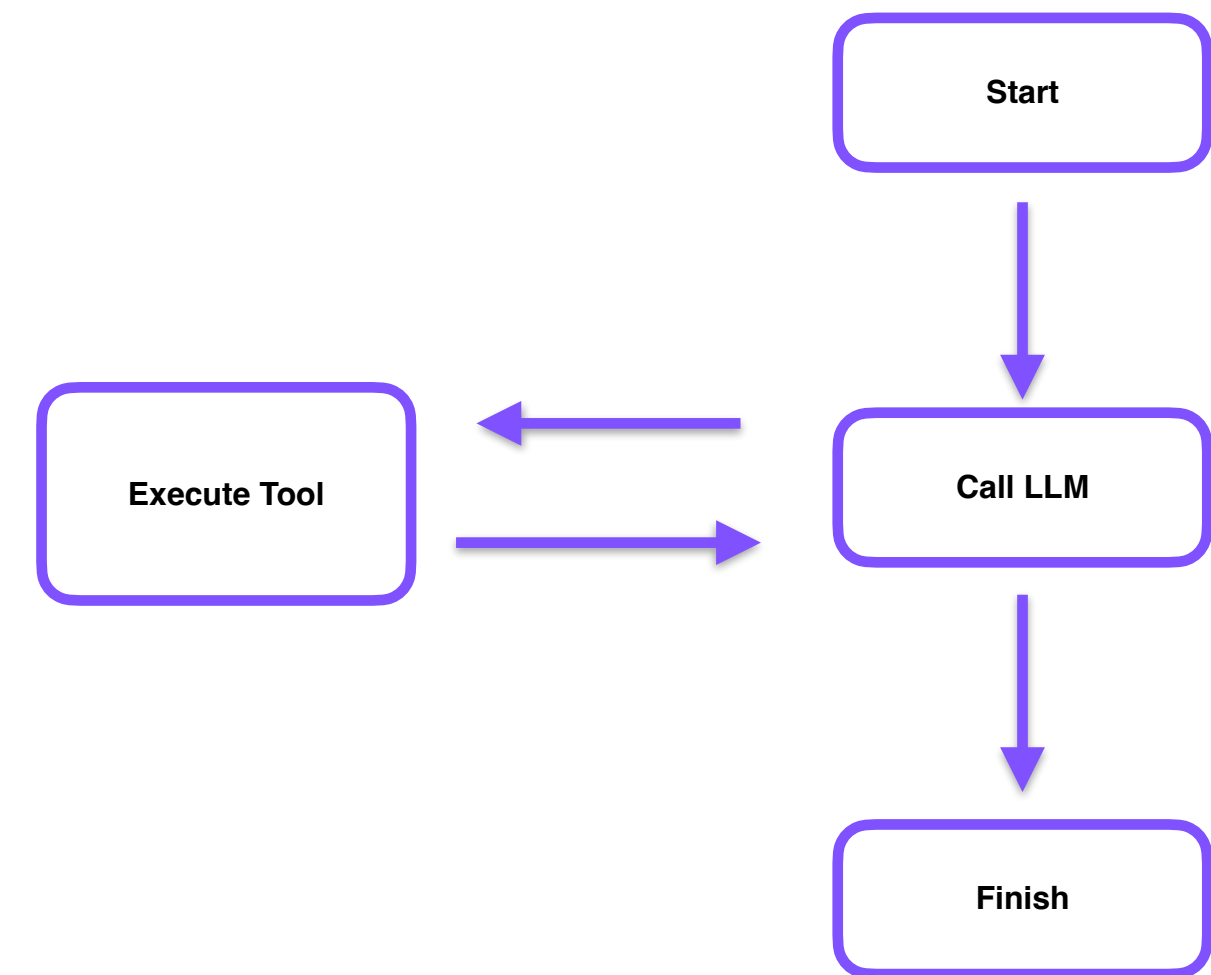
Ranked Document Storage

```
val embedder = LLMEmbedder(  
    OpenAILLMClient("API_KEY"),  
    OpenAIModels.Embeddings.TextEmbedding3Large  
)  
val docEmbedder = JVMTextDocumentEmbedder(embedder)  
val rankedDocStorage = EmbeddingBasedDocumentStorage(  
    docEmbedder,  
    InMemoryVectorStorage()  
)  
...  
  
val nodePlanTrip2 by node<String, String> { request →  
    val relevantTweets = rankedDocStorage.mostRelevantDocuments(  
        query = request,  
        count = 3,  
        similarityThreshold = 0.9  
    ).toList()  
    ...  
}
```



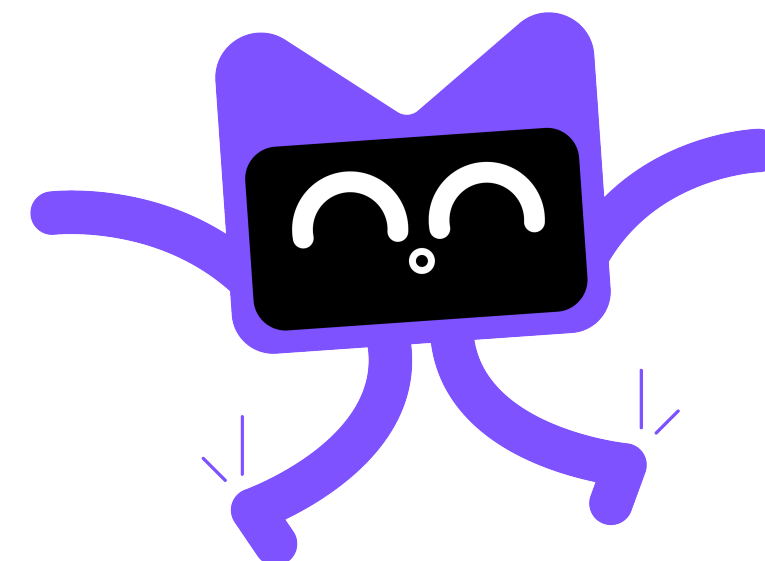
Non-Graph API (0.5.0)

```
functionalAIAgent<String, String>(  
    prompt = "You're a trip planning assistant.",  
    promptExecutor = executor,  
    model = OpenAIModels.Chat.GPT4o,  
    toolRegistry = ToolRegistry { WeatherTool() }  
) { input →  
    var responses = requestLLMMultiple(input)  
  
    while (responses.containsToolCalls()) {  
        val pending = extractToolCalls(responses)  
        val results = executeMultipleTools(pending)  
        responses = sendMultipleToolResults(results)  
    }  
  
    responses.single().asAssistantMessage().content  
}.run("What is the weather like in Amsterdam?")
```



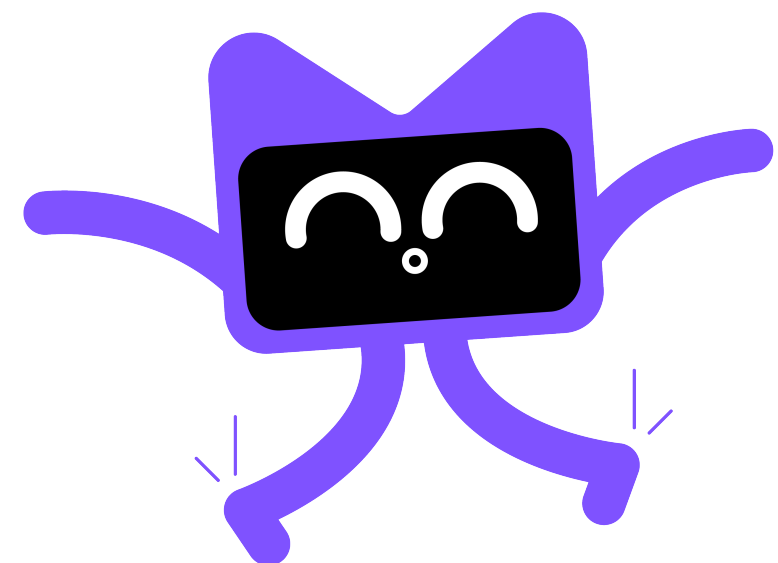
History Compression

```
val compressHistory by nodeLLMCompressHistory<Unit>()
```



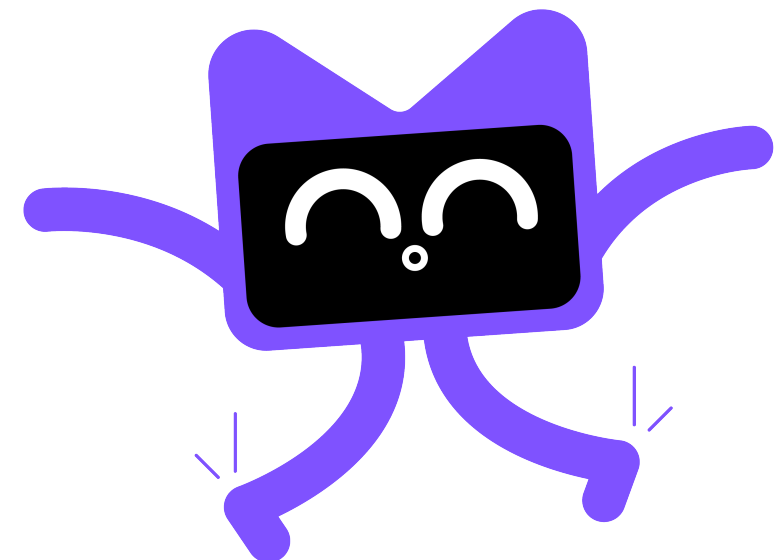
History Compression

```
val compressHistory by nodeLLMCompressHistory<Unit>(
    strategy = HistoryCompressionStrategy.WholeHistory
)
```



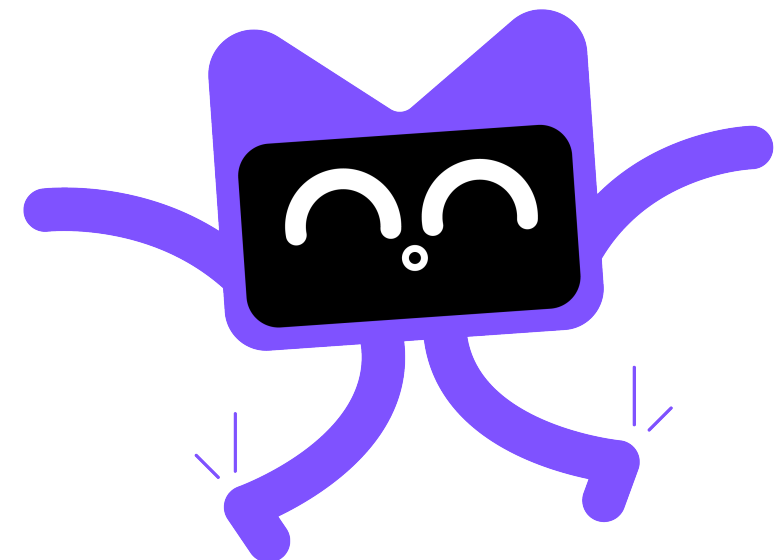
History Compression

```
val compressHistory by nodeLLMCompressHistory<Unit>(
    strategy = HistoryCompressionStrategy.Chunked(chunkSize = 20)
)
```



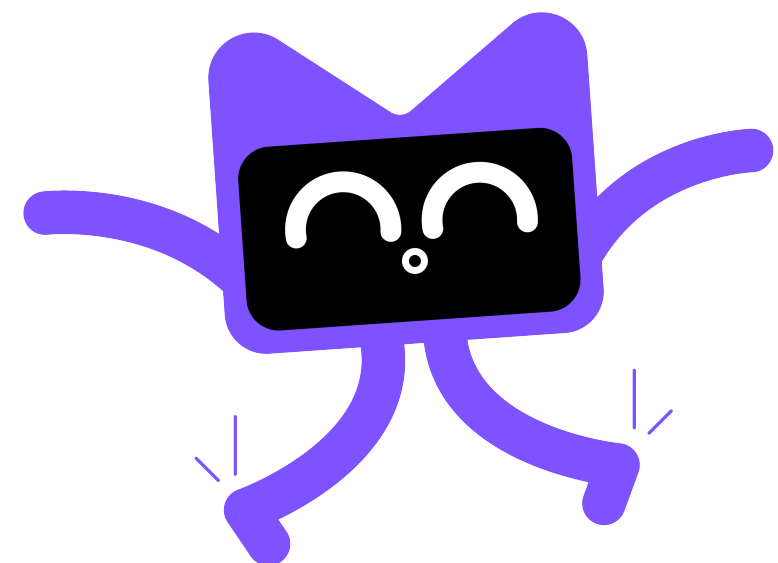
History Compression

```
val compressHistory by nodeLLMCompressHistory<Unit>(
    strategy = object : HistoryCompressionStrategy(){
        override suspend fun compress(
            llmSession: AIAgentLLMWriteSession,
            preserveMemory: Boolean,
            memoryMessages: List<Message>
        ) {
            TODO("Not yet implemented")
        }
    })
```



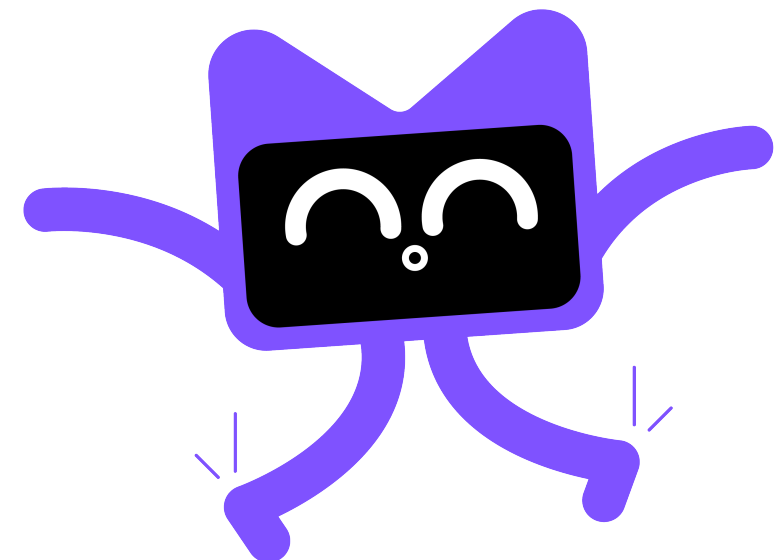
Features

```
val result = AIAgent(  
    executor = executor,  
    llmModel = AnthropicModels.Sonnet_4,  
) { // FeatureContext.() → Unit  
  
}.run("Tell me a Kotlin Joke!")
```



EventHandler

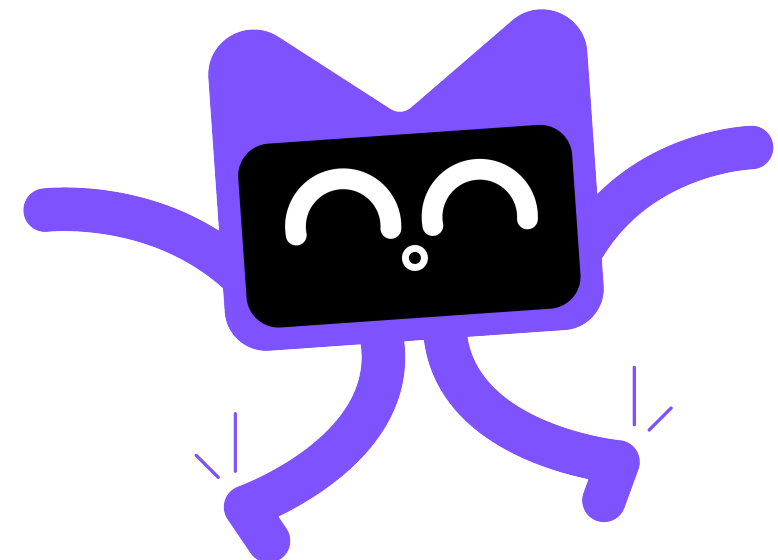
```
install(EventHandler) {  
  onToolCall { ctx →  
    logger.debug {  
      "Tool ${ctx.tool.name}, args ${  
        ctx.toolArgs.toString().replace('\n', ' ').take(100)}  
      } ..."  
    }  
  }  
  onToolCallFailure { ctx →  
    logger.error(ctx.throwable) { "Tool ${ctx.tool.name} failed with ${ctx.throwable.message}" }  
  }  
  onToolCallResult { ctx → logger.debug { "Tool ${ctx.tool.name} returned ${ctx.result}" } }  
}
```



Tracing

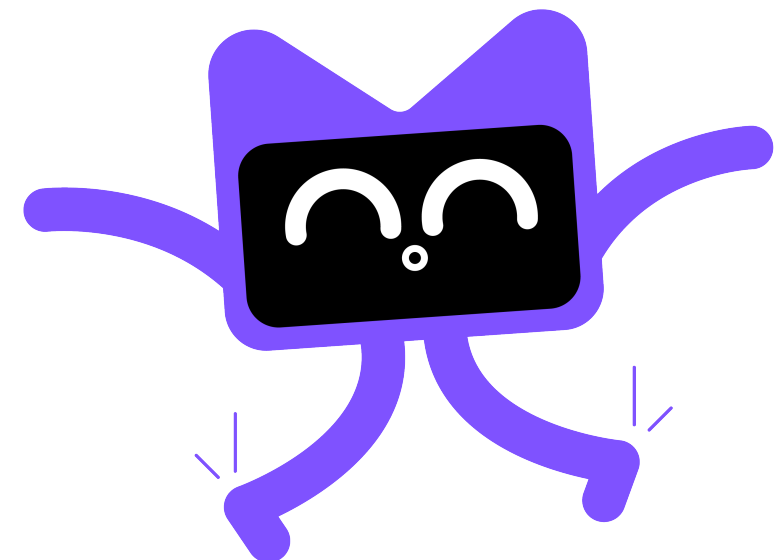
```
val logger = KotlinLogging.logger { }

val result = AIAgent(
    executor = executor,
    llmModel = AnthropicModels.Sonnet_4,
) {
    install(Tracing) {
        addMessageProcessor(TraceFeatureMessageLogWriter(logger))
    }
}.run("Tell me a Kotlin Joke!")
```



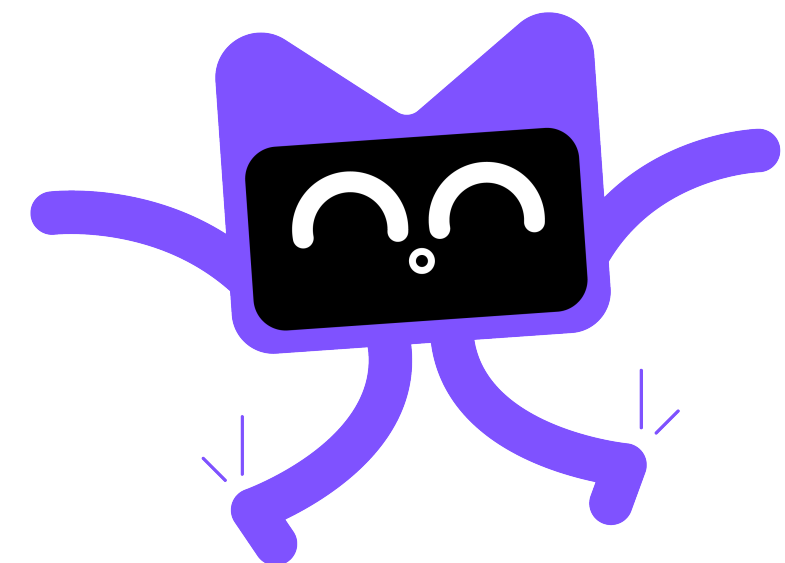
OpenTelemetry

```
val result = AIAgent(  
    executor = executor,  
    llmModel = AnthropicModels.Sonnet_4,  
) {  
    install(OpenTelemetry) {  
        setServiceInfo("my-agent-service", "1.0.0")  
        addSpanExporter(LoggingSpanExporter())  
    }  
}.run("Tell me a Kotlin Joke!")
```



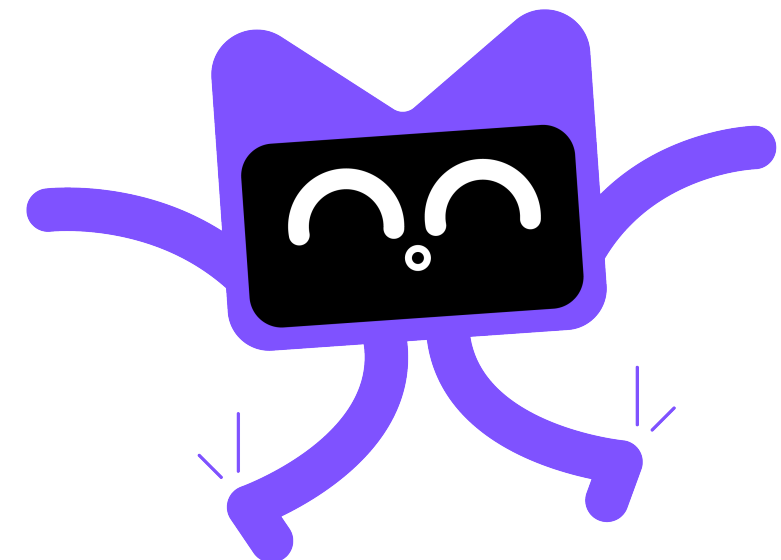
OpenTelemetry: Langfuse

```
val result = AIAgent(  
    executor = executor,  
    llmModel = AnthropicModels.Sonnet_4,  
) {  
    install(OpenTelemetry) {  
        setServiceInfo("my-agent-service", "1.0.0")  
        addLangfuseExporter(  
            "https://cloud.langfuse.com",  
            System.getenv("LANGFUSE_API_KEY"),  
            System.getenv("LANGFUSE_PROJECT_ID")  
        )  
    }  
}.run("Tell me a Kotlin Joke!")
```



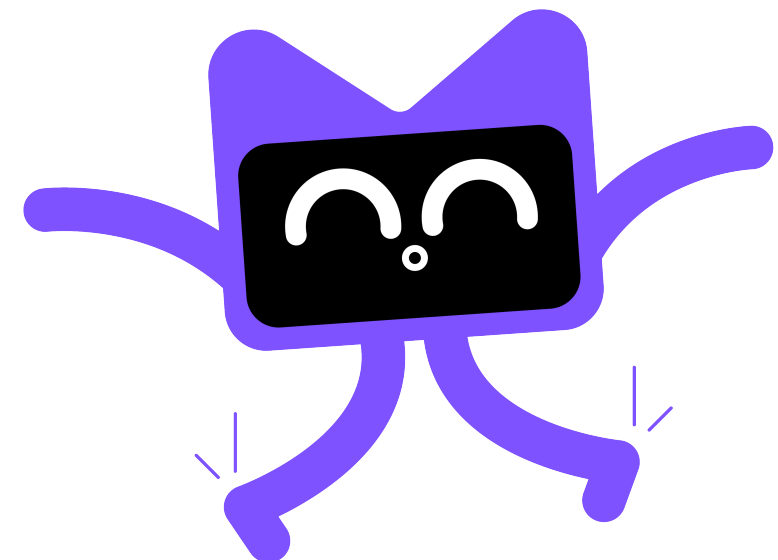
OpenTelemetry: Weave

```
val result = AIAgent(  
    executor = executor,  
    llmModel = AnthropicModels.Sonnet_4,  
) {  
    install(OpenTelemetry) {  
        setServiceInfo("my-agent-service", "1.0.0")  
        addWeaveExporter(  
            weaveEntity = "<your-entity>",  
            weaveApiKey = "<your-api-key>",  
            weaveProjectName = "koog-tracing"  
        )  
    }  
}.run("Tell me a Kotlin Joke!")
```



Persistency: Checkpoints

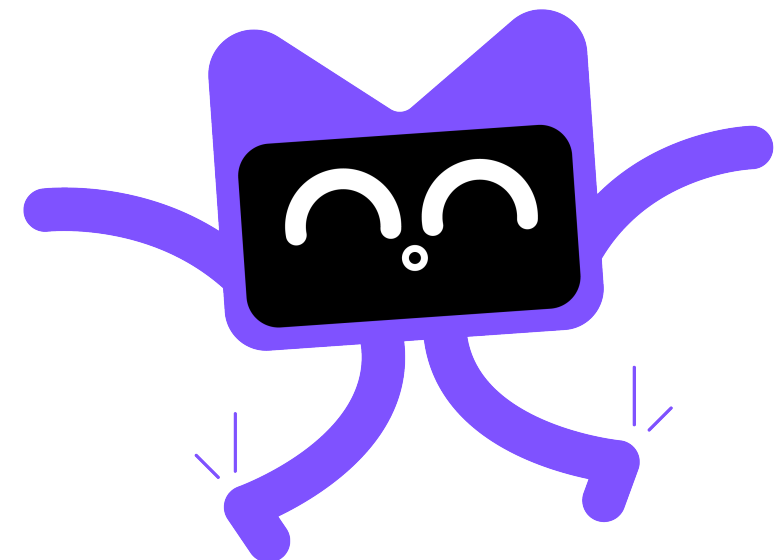
- Message history (all interactions between user, system, assistant, and tools)
- Current node being executed
- Input data for the current node
- Timestamp of creation



Persistency

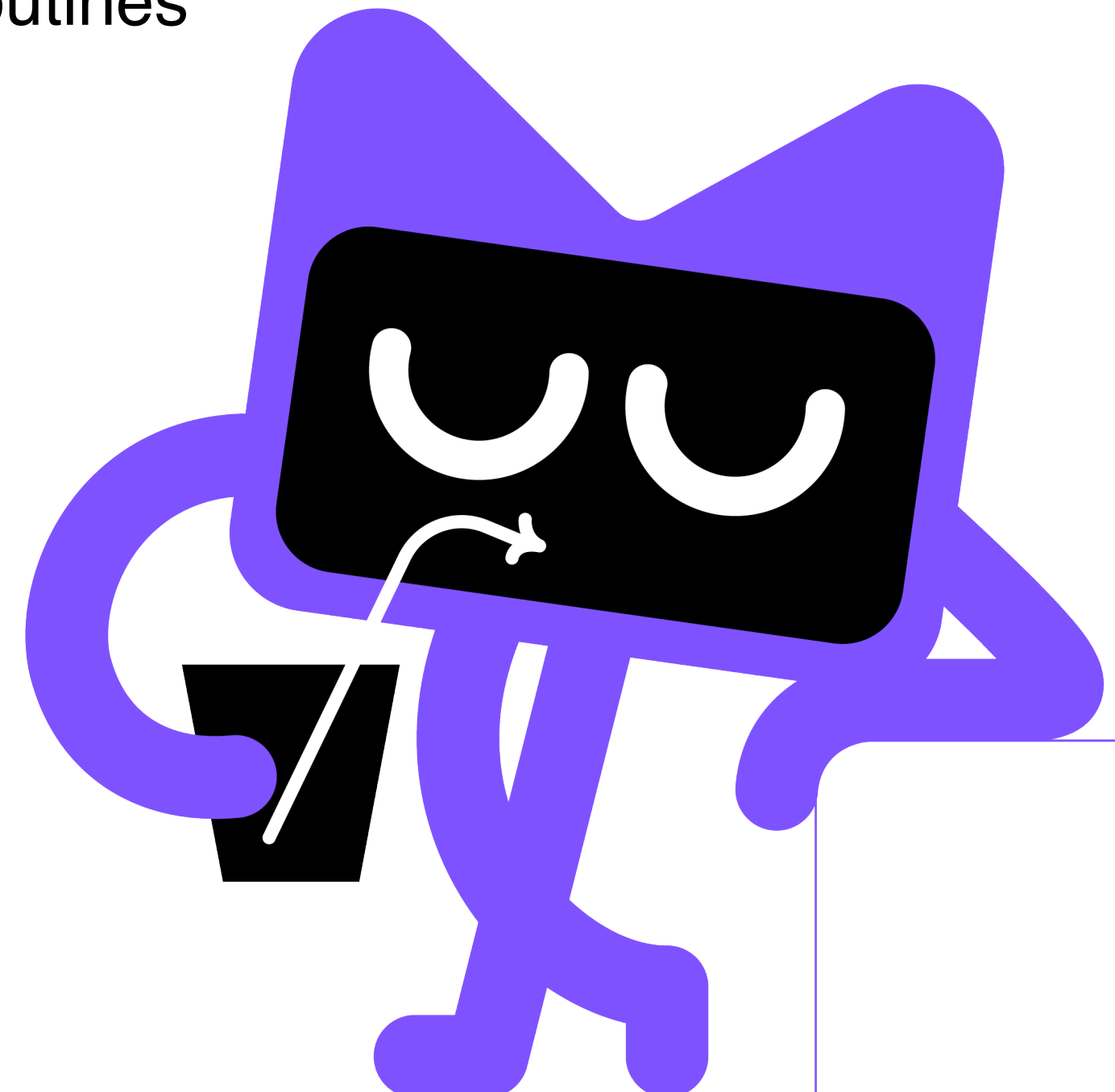
```
val persistenceId = Uuid.random().toString()

val result = AIAgent(
    executor = executor,
    llmModel = AnthropicModels.Sonnet_4,
) { // FeatureContext() → Unit
    install(Persistency) {
        storage = InMemoryPersistencyStorageProvider(persistenceId)
        enableAutomaticPersistency = true
    }
}.run("Tell me a Kotlin Joke!")
```

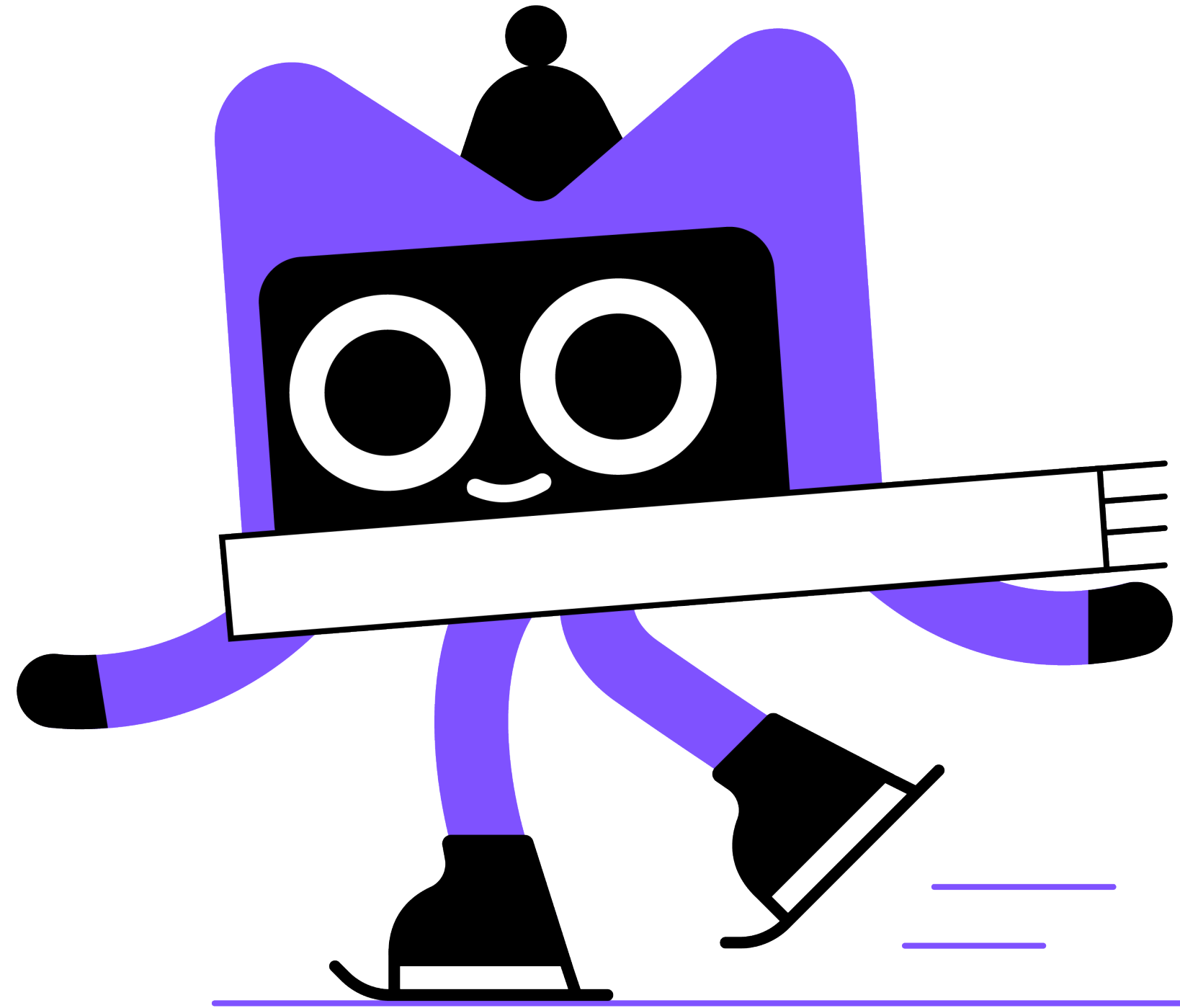


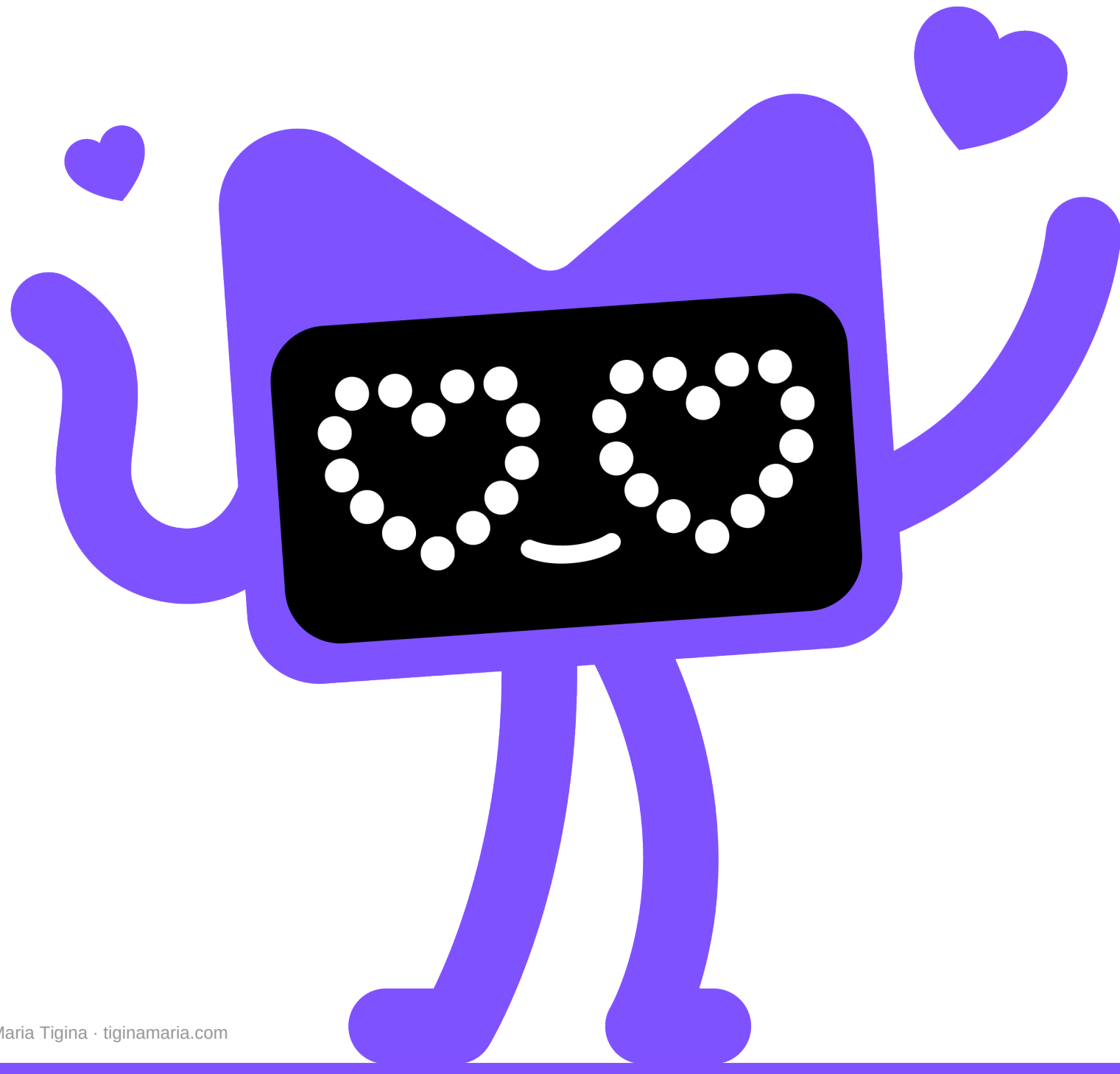
Summary

- Seamless concurrency & streaming thanks to Kotlin Coroutines
- Typesafe DSL for scalability and confidence
- Plugins allow compose-able and scalable architectures
- Efficient, lightweight and highly performant



Demo time!





Embabel by Rod Johnson (KotlinConf)

- Kotlin and Spring: The modern server side stack

OpenAI Java SDK & Anthropic Java SDK

- 100% written in Kotlin

Today's example:

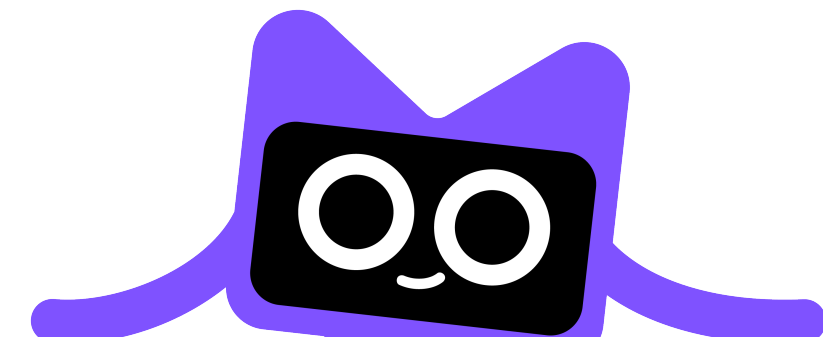
- github.com/nomisRev/ktor-koog-example

src/main/resources/application.yaml

```
app:  
  apiKey: $OPENAI_API_KEY  
  host: "$HOST:0.0.0.0"  
  port: "$PORT:8080"
```

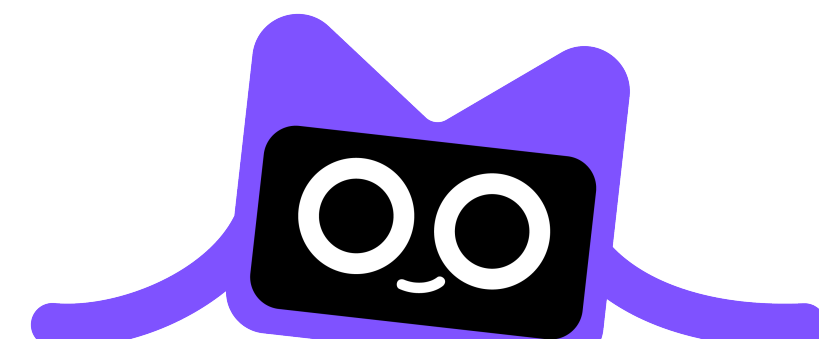
Environment
Variable

Fallback Value



AppConfig

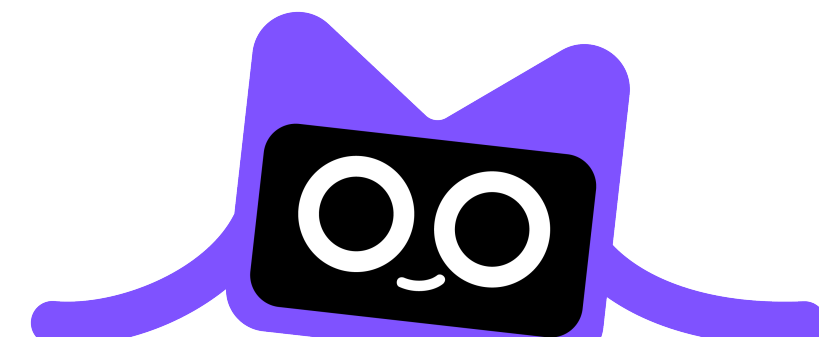
```
@Serializable  
data class AppConfig(  
    val host: String,  
    val port: Int,  
    val apiKey: String,  
)
```



AppConfig

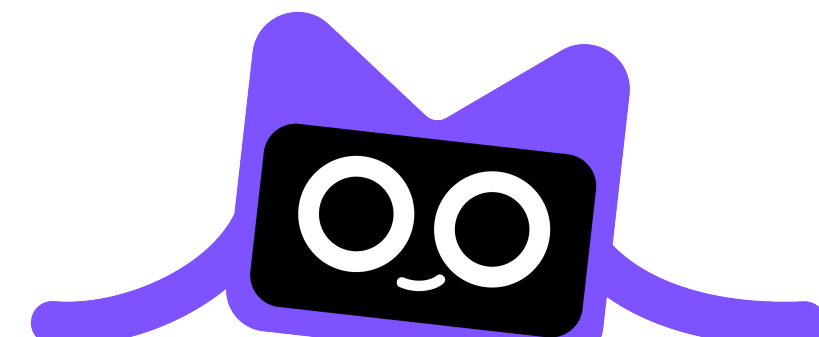
```
@Serializable
data class AppConfig(
    val host: String,
    val port: Int,
    val apiKey: String,
)

val config = ApplicationConfig("application.yaml")
    .property("app")
    .getAs<AppConfig>()
```



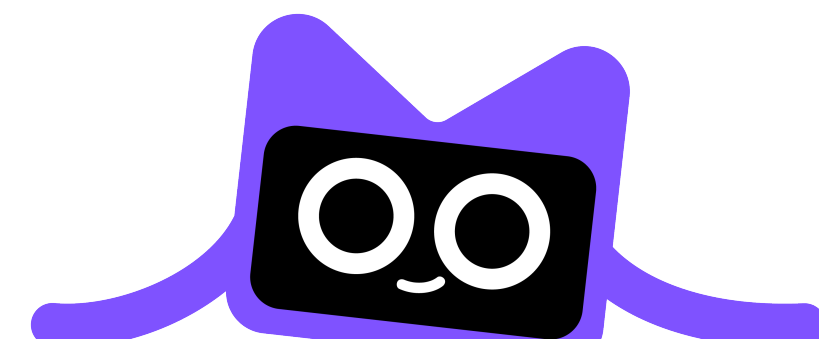
EmbeddedServer

```
embeddedServer(Netty, host = "0.0.0.0", port = 8080) {  
    // suspend Application.() → Unit  
}.start(wait = true)
```



Application.kt

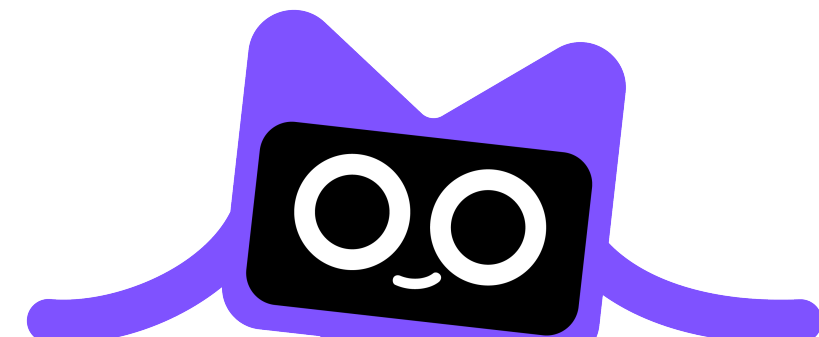
```
fun main() {  
    val config = ApplicationConfig("application.yaml")  
        .property("app")  
        .getAs<AppConfig>()  
  
    embeddedServer(Netty, host = config.host, port = config.port) {  
        // suspend Application.() → Unit  
    }.start(wait = true)  
}
```



Application.kt

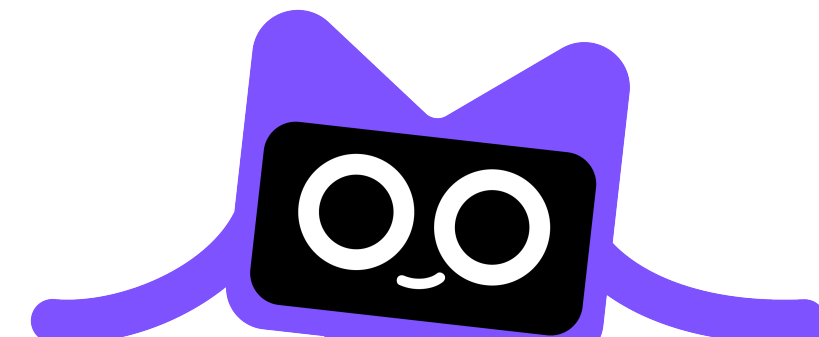
```
fun main() {  
    val config = ApplicationConfig("application.yaml")  
        .property("app")  
        .getAs<AppConfig>()  
  
    embeddedServer(Netty, host = config.host, port = config.port) {  
        // suspend Application.() → Unit  
    }.start(wait = true)  
}
```

Yaml & HOCON Support



Application.kt

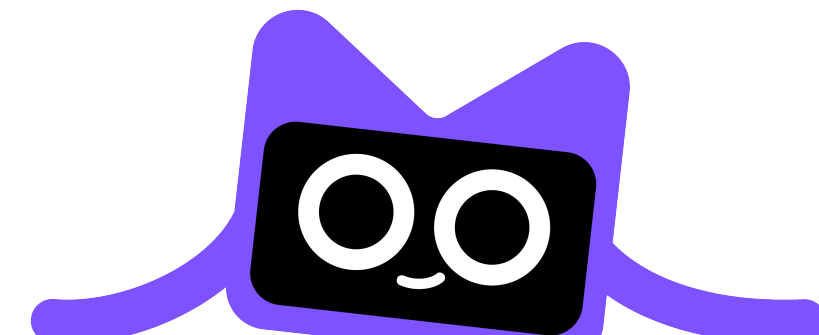
```
fun main() {  
    val config = ApplicationConfig("application.yaml")  
        .property("app")  
        .getAs<AppConfig>()  
  
    embeddedServer(Netty, host = config.host, port = config.port) {  
        app(config)  
    }.start(wait = true)  
}  
  
suspend fun Application.app(config: AppConfig) { }
```



TestApplication

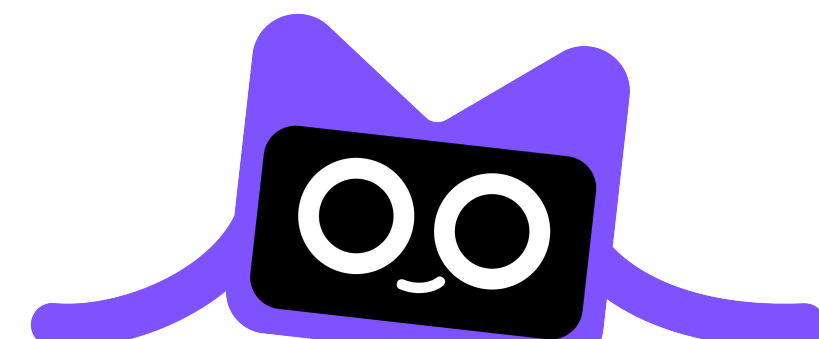
@Test

```
fun test() = testApplication {  
    application {  
        app(AppConfig(host = "not-used-in-test", port = 9999))  
    }  
    val response = client.get("/call-some-endpoint")  
    assertEquals(HttpStatusCode.OK, response.status)  
}
```



Application.kt

```
suspend fun Application.app(config: AppConfig) {  
    install(SSE)  
}
```



Application.kt

```
suspend fun Application.app(config: AppConfig) {  
    install(SSE)  
    sse("/hello") {  
        send(ServerSentEvent("world"))  
    }  
}
```

