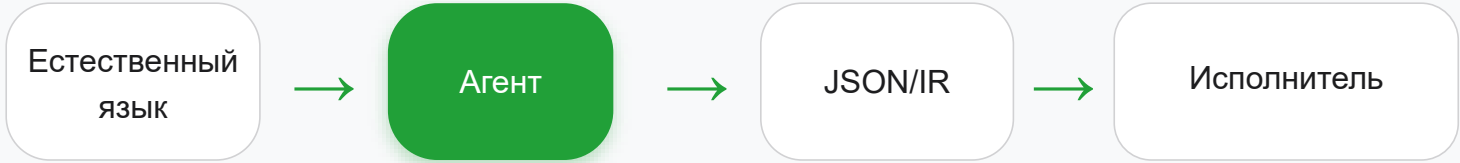
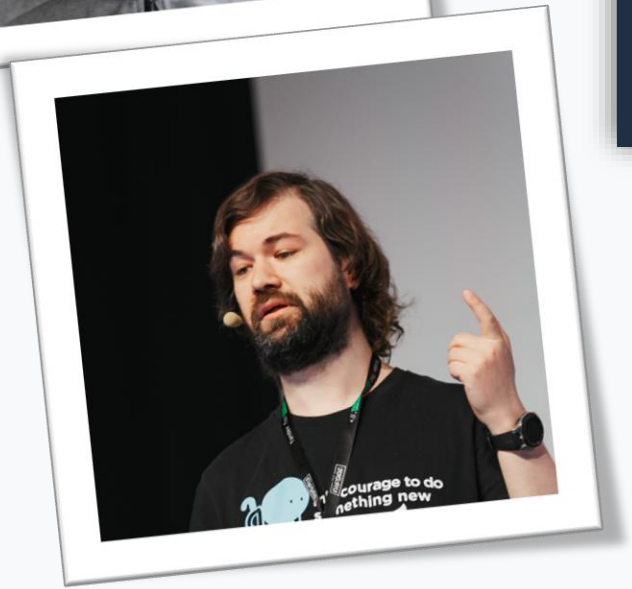
Koog твой друг: специализированный агент на Kotlin

JPoint 2026 · Александр Нозик · Максим Колпаков
при поддержке ДГР Сбера



Максим Колпаков



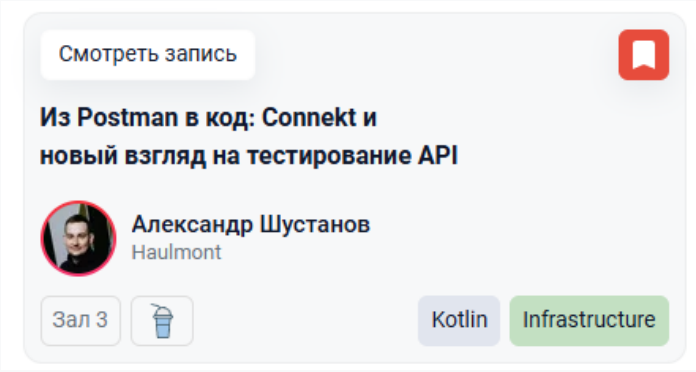
Александр Нозик

Задача. Зачем нам агент?

- Надо делать автоматические интеграционные тесты на основе сценариев и контекста.
- Тестировщики не хотят учиться программировать.
- Поставлена задача сделать это на модном агентском подходе, чтобы без программистов.

А если без агента?

- Делаем DSL/Builder, генерирующий промежуточное представление для исполняемого теста.
- Делаем простую обертку например для Kotlin Jupyter kernel.
- Обучаем людей работе с DSL и радуемся.



https://jokerconf.com/archive/2025/persons/edc93537342143bea8f1424a25158e68/

https://amplicode.ru/http-client/

Программиста подход: генерируем код

- Делаем чат-генератор кода.
- Добавляем к нему какой-нибудь MCP, чтобы можно было подкачивать данные.
- Добавляем второго агента, чтобы проверял код.

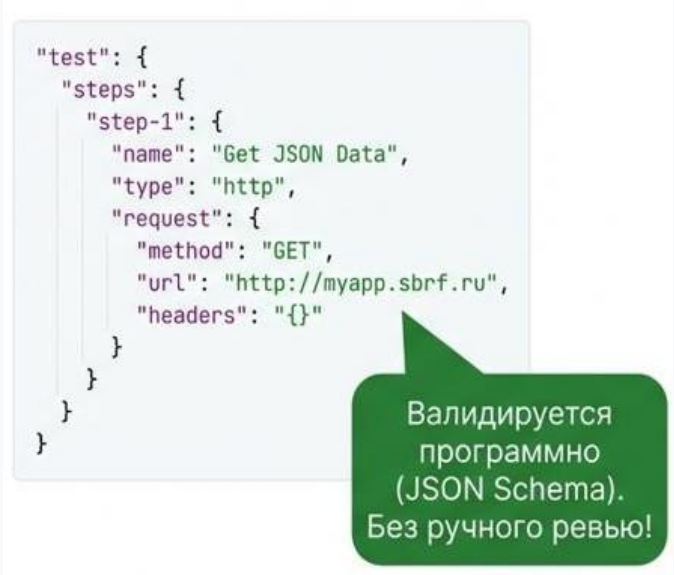
- Для работы все еще требуется уметь читать и писать код.
- Огромная дыра в безопасности (результирующий код может творить дичь).
- Плохая воспроизводимость.

Технолога подход: генерируем сценарий

Текстовая спецификация

Вводим описание теста и нажимаем «отправить». Добавляем данных если их недостаточно.

генерация



Исполнение сценариев на специальном безопасном сервере

Технолога подход: генерируем сценарий

Плюсы

→ Тестировщик пишет текст, не код

→ Сценарий валидируется по схеме автоматически

→ Безопасное исполнение — агент не выйдет за рамки IR

→ Воспроизводимость — тот же JSON, то же поведение

Минусы

→ Схему IR нужно синхронизировать с исполнителем

→ Агент также может сгенерировать невалидный JSON

→ Качество всё равно зависит от полноты контекста

Схема IR — это контракт. Чем строже контракт, тем надёжнее результат.

Анатомия агента

Что такое агентский фреймворк?

Что говорит на эту тему Copilot:

Агентский фреймворк — это набор библиотек и инструментов, которые помогают:

- **Оркестрировать** работу агента: управлять шагами, задачами, ветвлениями.
- **Хранить память**: контекст, историю диалогов, знания.
- **Интегрироваться с инструментами**: API, базы данных, выполнение кода.
- **Коммуницировать**: обмен сообщениями между агентами в мультиагентных системах.
- **Масштабировать**: строить сложные цепочки действий и распределённые системы.

- Prompt — один запрос и один ответ
- Агент — целая архитектура циклов вызовов и проверок
- Как это собрать вместе? – с агентским фреймворком!

Если на складе 124 маленьких колеса и 62 больших, сколько 3-х колесных велосипедов можно собрать?.. «Ни одного, одних колёс недостаточно, чтобы собрать велосипед!»

М. Задорнов, 2002 год

А зачем агентский фреймворк?

Без Spring

→ Голые сервлеты, ручной DI

→ Сам пишешь транзакции, маппинг, жизненный цикл

→ Можно. Но зачем, если есть Spring?

Без агентского фреймворка

→ Голые вызовы LLM API

→ Сам пишешь цикл, ретраи, память, стратегию

→ Можно. Но зачем, если есть, например, Koog?

Фреймворк убирает инфраструктурный шум — остаётся только логика агента.

Какие основные агентские фреймворки (в мире JVM)?

ТУЛКИТЫ (НАБОРЫ КОМПОНЕНТОВ)

Spring AI

Spring-нативный

- Java/Kotlin JVM-ориентированный
- Набор базовых паттернов и советников
- Минимальное планирование (не планировщик)
- Основа для построения компонентов
- Еще очень свежий

LangChain4j

Популярный на Java

- Крупнейший Java-тулкит для ИИ
- Интеграция с 100+ LLM и VectorDB
- Модульный: инструменты, память, извлечение
- Строительные блоки для ИИ-систем

АГЕНТСКИЕ ФРЕЙМВОРКИ

Koog

Агентский фреймворк от JetBrains

- Kotlin и Java API (KMP)
- Три стратегии: функциональная, граф и планировщик
- Встроенная отказоустойчивость и checkpoint-ы
- Интеграция со Spring Boot и Ktor

Embabel

Агентский фреймворк от создателя Spring

- Надстройка над Spring AI с высокоуровневым API
- Планирование через GOAP — предсказуемо, без магии LLM
- Динамическая переработка плана после каждого шага
- Строгая типизация: промпты и доменная модель

Spring AI и **LangChain4j** — это мощные тулкиты для создания ИИ-компонентов. **Koog** и **Embabel** — полноценные **агентские фреймворки** с готовой оркестрацией.

А когда и что выбрать? Два основных пути у нас

Koog

- → Автономные агенты со сложной логикой
- → Графовая оркестрация, планирование целей
- → Kotlin-first
- → Мультиплатформенность: iOS, Android, браузер
- → Сохранение состояния с восстановлением
- → Межагентное взаимодействие по протоколу A2A

Spring AI

- → Встраивание ИИ в существующий Spring-сервис
- → RAG, семантический поиск, 19+ хранилищ
- → Агентные паттерны из советников
- → Java-команда с Spring-практиками
- → Коммерческая поддержка Tanzu
- → Максимально широкий выбор провайдеров

Не конкуренты, а дополнения по необходимости. Два уровня стека, одна JVM-платформа.

А что там с агентами?

Koog: каркас агента

Разработчик описывает стратегию (граф, план, функцию) — каркас сам управляет исполнением: какие узлы обойти, когда вызвать модель, когда запустить инструмент, как откатиться.

Аналогия: готовый станок — задаёшь программу, станок сам выполняет операции по порядку.

Spring AI: набор инструментов

Разработчик сам собирает агента из советников, инструментов, памяти и вызовов модели. Рекурсивные советники дают циклы. Паттерны (цепочка, маршрутизация, оркестратор) задокументированы и реализуемы.

Аналогия: верстак с деталями — сам решаешь, в каком порядке их соединить.

Можно совмещать: Koog берёт модели через Spring AI (мост-стартер koog-spring-ai-starter-model-chat).
При этом логику инструментов всегда исполняет Koog, а Spring AI передаёт модели только их описания.

https://docs.koog.ai/spring-ai-integration/

А почему всё же не Spring AI? Можно, если нужно, и на JVM!

Koog — Kotlin-нативный

- Все операции — сопрограммы (suspend)
- Типобезопасный предметный язык агентов
- Изолированные классы для when проходов
- @Serializable → автоматическая JSON-схема
- Функции-расширения для стратегий
- Kotlin Multiplatform: 5 целевых платформ

Spring AI — Java-ориентированный

- ChatClient.call() — блокирующий вызов
- Потоковый вывод через Reactor Flux
- Нет предметного языка для ИИ-функций
- Запрос #3718: поддержки сопрограмм пока нет
- Цепочечный построитель в стиле Java
- Только JVM (нет мультиплатформы)

Spring Boot 4 / Spring AI 2.0: базовая планка Kotlin 2.2, JSpecify для null-safety — но не в ИИ-части API

Ключевые возможности Koog

Сохранение состояния агентов

Полное сохранение в PostgreSQL, S3 или на диск. Откат побочных действий через RollbackToolRegistry.

Маршрутизатор моделей

Циклическое распределение запросов между провайдерами, автоматический переход на запасного при сбоях.

Сжатие контекста

Встроенные стратегии сжатия длинных диалогов для уменьшения расхода токенов.

Долговременная память

Извлечение и сохранение фактов из бесед, обогащение промптов контекстом из прошлых разговоров.

Наблюдаемость

OpenTelemetry, адаптеры для Langfuse и W&B Weave. Маскирование чувствительных данных в событиях.

Тестирование агентов

Предметный язык для подмены ответов модели, проверка структуры графов, воспроизводимые сценарии.

Spring AI становится не менее интересным вариантом на будущее

spring-ai-agent-utils (v0.7.0) — библиотека, вдохновлённая Claude Code. Серия из 6 статей «Spring AI Agentic Patterns»

Навыки агентов (SkillsTool)

Модульные папки с инструкциями, скриптами и ресурсами.
Агент подгружает нужный навык по запросу.

Подагенты (TaskTool)

Иерархическая оркестрация: главный агент делегирует задачи специализированным подагентам.

Долговременная память

AutoMemoryTools: файловая память между сеансами.
Индекс MEMORY.md + тематические файлы.

Протокол A2A

Взаимодействие между агентами по открытому стандарту.
Обнаружение через AgentCard, передача задач.

Вопрос пользователю

AskUserQuestionTool: агент останавливается и задаёт уточняющий вопрос человеку.

Управление задачами

TodoWriteTool: структурированное отслеживание хода выполнения многошаговых задач.

Не часть ядра Spring AI, но официально поддерживается командой и работает с Spring AI 2.0.0-M2+

Архитектура агентов Koog

Базовый

Заданная стратегия.
Один цикл «запрос → ответ»
с вызовом инструментов.

Графовый

Направленный граф: узлы, рёбра и условные переходы.
Обход задаётся в предметном языке.

Функциональный

Произвольный Kotlin-код.
Каждый шаг — сопрограмма
с доступом к модели.

Планировщик

GOAP: цели, предусловия и эффекты действий.
Модель
или алгоритм строит план.

Поддерживаемые протоколы

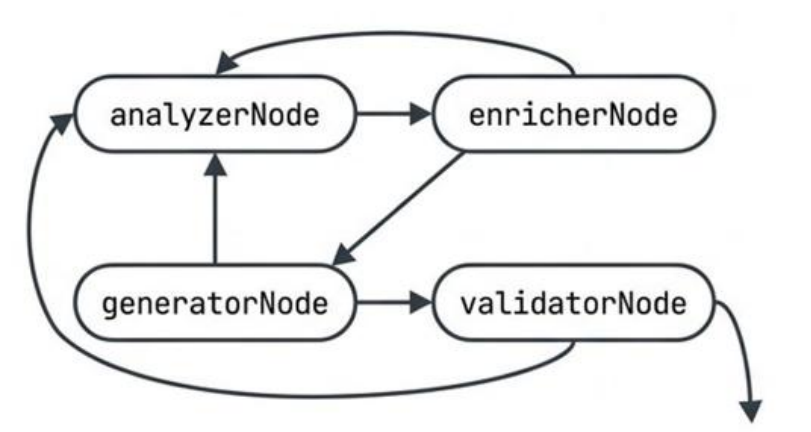
MCP подключение внешних инструментов к модели

A2A взаимодействие между агентами

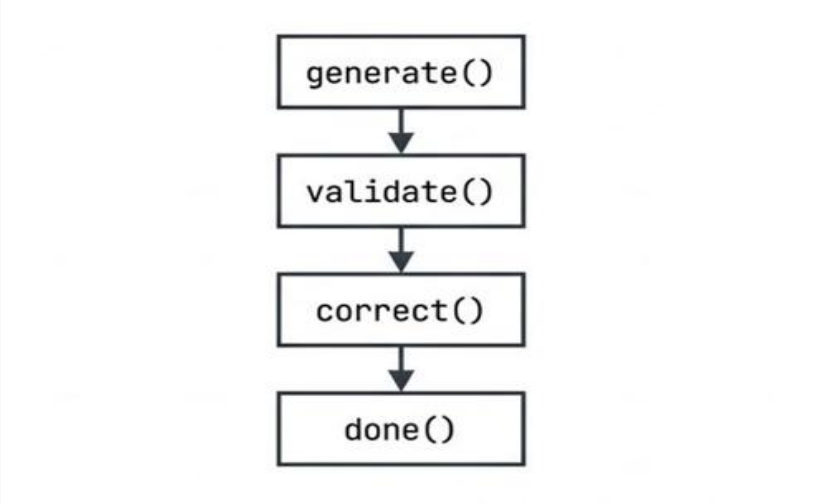
ACP встраивание в среду разработки JetBrains

Погружение в типы агентской логики

Граф



Функция



Попробовали и то, и другое, можем сравнить.

То же самое применимо не только к Koog! Но именно он позволил нам быстрее начать обкатку

Сравнение подходов

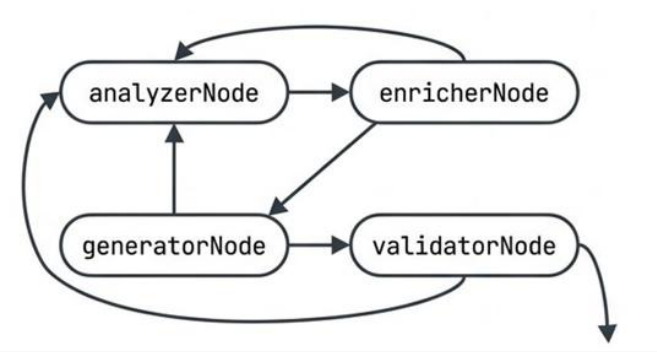
Графовая логика

- Можно разделить ответственность разных узлов.
- Можно тестировать узлы отдельно.
- Сложно делать циклы и сложные ветвления.

Функциональная логика

- Проще видеть всю логику в одном месте.
- Можно делать сложную передачу управления и ветвления.
- Можно делать произвольные циклы (в том числе вложенные).

Логика на графах, суть процесса.



- Анализатор разбирает входные данные и создает план работы.
- Обогатитель проходится по пунктам плана и корректирует их, запрашивая у пользователя дополнительную информацию.
- Генератор генерирует результат на основе поправленного плана.
- Валидатор проверяет, что результат адекватный.

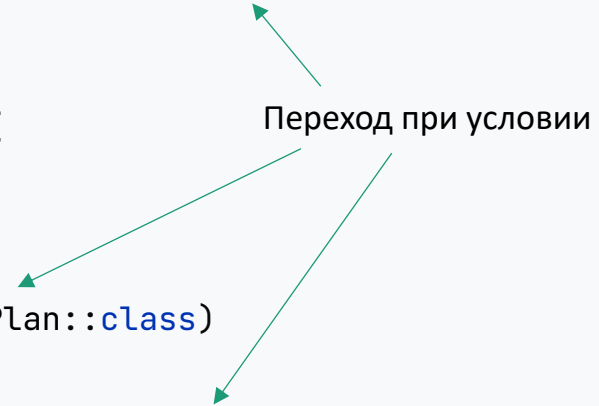
Логика на графах. Общий вид.

```
edge(nodeStart forwardTo analyzer)
edge(analyzer forwardTo generator onIsInstance CompletePlan::class)
edge(analyzer forwardTo enricher onIsInstance IncompletePlan::class)
edge(enricher forwardTo userCommunicationTool onIsInstance PlanNeedsInput::class)

// выдает ошибку, если запрос неуспешный
edge(
  userCommunicationTool forwardTo validator transformed {
    it.asSuccessful().result
  }
)

edge(validator forwardTo generator onIsInstance CompletePlan::class)

// вернуться на этап обогащения
edge(validator forwardTo enricher onIsInstance IncompletePlan::class)
edge(generator forwardTo nodeFinish)
```



Логика на графах. Узел.

Ресивер для узла и скоуп для вызовов

```
internal fun AIAgentGraphStrategyBuilder<*, *>.analyzerNode() = node<ScenarioData, AnalyzedPlan>(
    "analyzer"
) { data ->

    val schemaType = detectSpecificationType(data, logCollector) ?: TestType.UI
    val schemaString = loadSchema(schemaType)

    val llmResult: JsonObject = callAnalyzerLlmOrNull(data.specification, schemaString) as? JsonObject
        ?: error("LLM response is not a valid JSON object")

    val structuredPlan = llmResult["structured_plan"] as? JsonArray
        ?: error("LLM response does not contain a structured plan")
    val missingInfo = llmResult["missing_info"] as? JsonArray
        ?: error("LLM response does not contain missing info")

    logger.info {
        "Plan generated (schema: $schemaType). Steps: ${structuredPlan.size}. Missing info: $missingInfo"
    }

    AnalyzedPlan(data, structuredPlan, missingInfo)
}
```

Вызов в LLM (контекст из скоупа)

Типизированный результат

Функциональная логика

Точка входа:

```
val agent = AIAgent<ScenarioData, ScenarioValidationResult>(
    promptExecutor = executor,
    llmModel = model,
    // toolRegistry = toolRegistry, // Disabled
    strategy = functionalStrategy(messageFlow, correctionCycles),
    systemPrompt = "You are build automation engineer. Use Russian for all responses."
)
```

Точка определения стратегии:

```
internal fun functionalStrategy(
    log: FlowCollector<AgentMessage>,
    correctionCycles: Int,
    jsonParser: Json = Json {…}
) = functionalStrategy { input: ScenarioData -> … }
```

Функциональная логика

Выполняем запрос
и добавляем его в «промпт»

Выполняем тулы
если LLM их поддерживает

Извлекаем результат

```
// Send the user input to the LLM
var responses = requestLLMMultiple(initialPrompt)

// Only loop while the LLM requests tools
while (responses.containsToolCalls()) {
    // Extract tool calls from the response
    val pendingCalls = extractToolCalls(responses)
    // Execute the tools and return the results
    val results = executeMultipleTools(pendingCalls)
    // Send the tool results back to the LLM.
    //The LLM may call more tools or return a final output
    responses = sendMultipleToolResults(results)
}
// When no tool calls remain,
//extract and return the assistant message content from the response
val assistantMessage = responses.single().asAssistantMessage().content
log.emit(AgentMessage.MessageFromLlm(assistantMessage))

val sanitizedMessage = sanitizeJsonResponse(assistantMessage)
log.status("Получен ответ от LLM. Идет валидация результата...")
```

Работа с «промптом» (контекстным окном)

Диалог

- Каждый запрос и ответ добавляется в «промпт».
- Агент «помнит» всю историю общения и помнит, что его просили… и не просили.
- Для воспроизведения результата на других данных надо повторить всю историю общения.

Спецификация

- Даем входные данные и отрабатываем на них.
- Если надо что-то поменять, то меняем входные данные и «забываем» историю.
- Если даешь правильную вводную, то с большой вероятностью получает правильный вывод за один шаг.

Работа с «промптом» (контекстным окном)

- Чат с историей хорош для уникальных задач.
- Чат плох для повторяющихся задач, в которых варьируются только данные.
- Можно дополнительно ограничить типы и формат контекстной информации и информировать LLM о том, какую информацию можно запросить.

Контексты, тулы и прочие детальки в Koog

Инструмент своими руками, строго и прозрачно

```
Объясни текстом в 2-3 предложениях
  следующую ошибку, полученную при
  попытке преобразования
  спецификации в сценарий теста.

  {errorMessage}

  Предложи до трех способов
  исправления ошибки. Для
  исправления ошибки могут быть
  использованы два способа:
  * Изменение текста спецификации
  * Добавление элементов контекста
  из следующего списка:

  {missingKeys}
```

- Создаем типизированные ключи контекста с описанным содержанием.
- Даем возможность подгружать только их.
- При наличии ошибки предлагаем LLM выбрать какого ключа из списка ей не хватает.

Инструмент и немного кода

```
@Serializable(ScenarioContextSerializer::class)
public class ScenarioContext(
    val data: Map<Key<*>, Any?>
) {

    @Serializable
    sealed interface Key<T> {
        val name: String
        val header: String
        val description: String? get() = null
        val serializer: KSerializer<T>
        fun toPrompt(value: T): String
        fun toJson(value: T): JsonElement = Json.encodeToJsonElement(serializer, value)
        fun fromJson(json: JsonElement): T = Json.decodeFromJsonElement(serializer, json)
    }

    @Suppress("UNCHECKED_CAST")
    operator fun <T> get(key: Key<T>): T? = data[key] as? T
    operator fun contains(key: Key<*>) = key in data
    operator fun plus(other: ScenarioContext): ScenarioContext = ScenarioContext(data + other.data)
    fun <T> withKey(key: Key<T>, value: T): ScenarioContext = ScenarioContext(data + (key to value))
    fun withoutKey(key: Key<*>): ScenarioContext = ScenarioContext(data - key)
}
```

Весь контекст сценария кроме запроса

Трансформация контента в текст

Промты как контракт

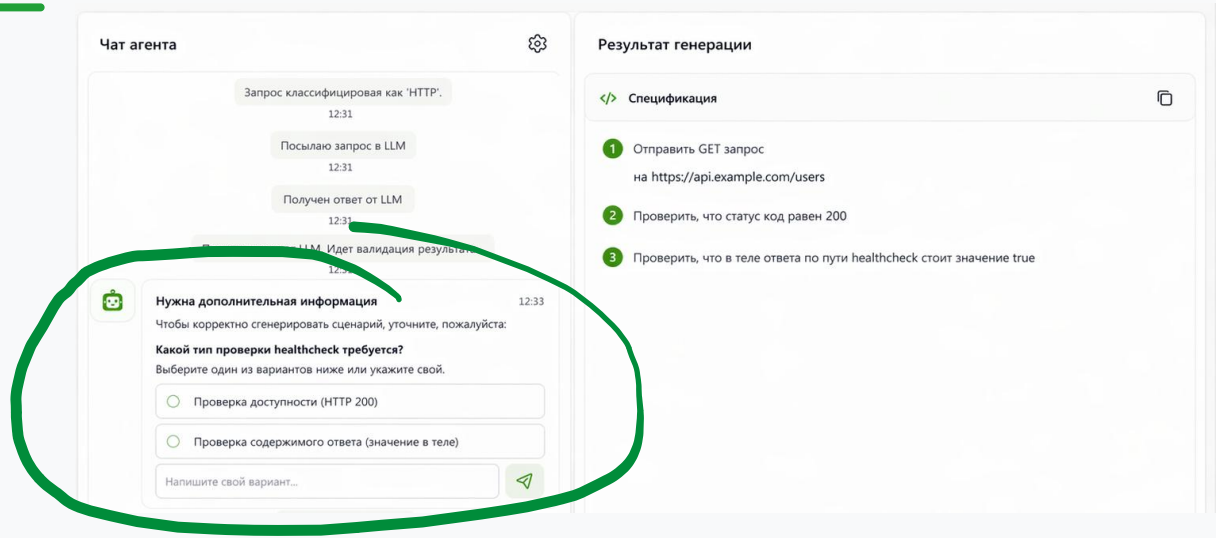
В ответе приводи только получившийся JSON без комментариев и дополнительных символов.

Если информации не хватает, помести null.

Никогда не придумывай значения для селекторов, URL и технических параметров.

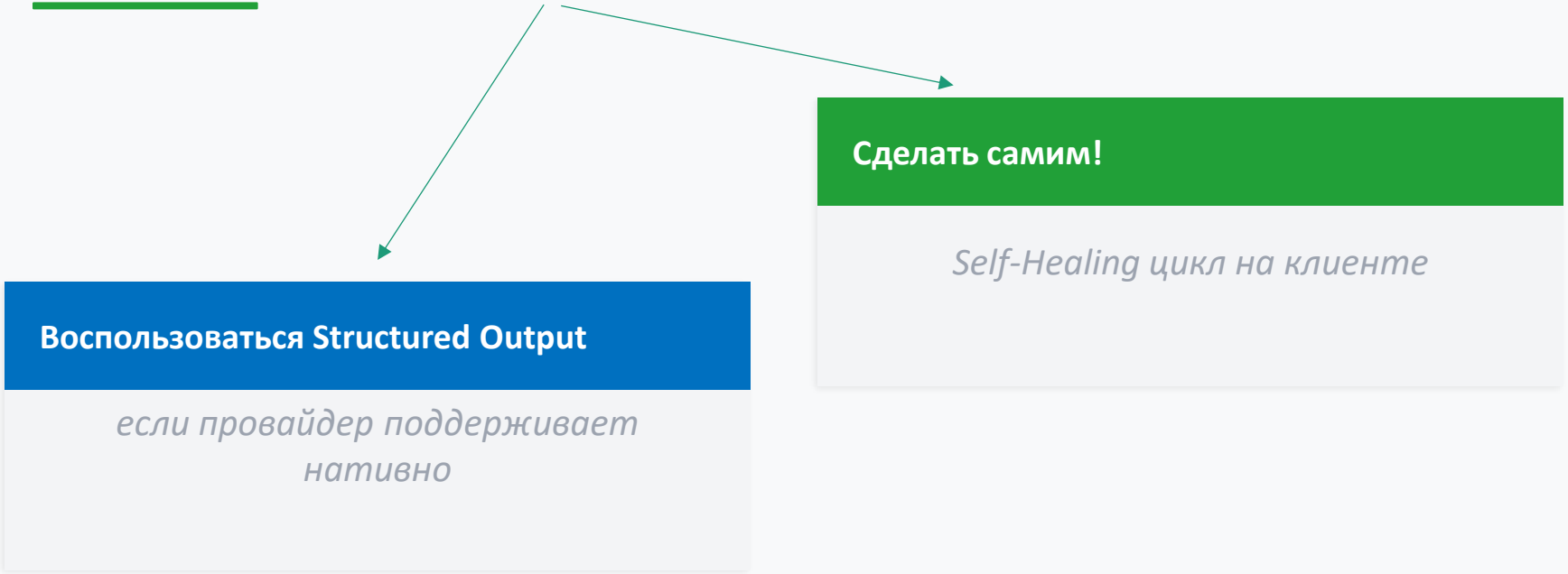
Отдельный prompt переводит ошибку на человеческий язык после исчерпания попыток

Пример запроса контекста



Агент запрашивает дополнительный контекст на основе списка возможных ключей.

Как нам сгенерировать JSON?



Structured Output: кто поддерживает

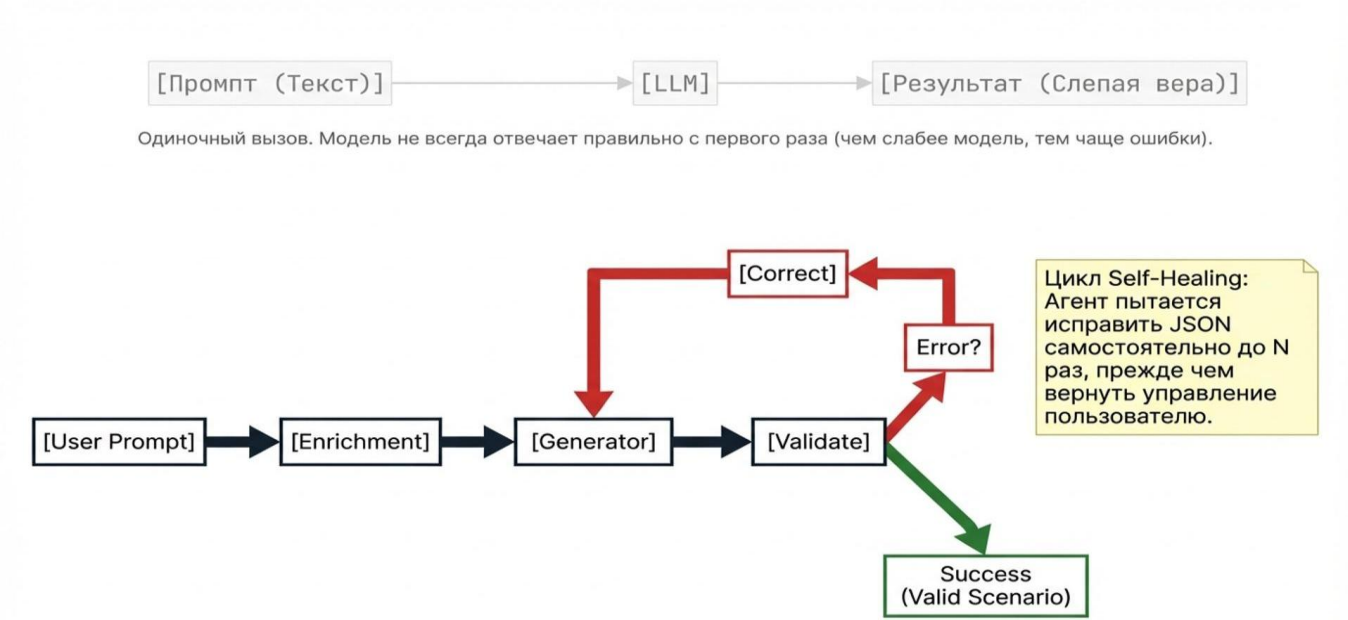
Не «верни JSON» в промпте, а схема как контракт на уровне API.

Нативная — гарантия соответствия схеме через constrained decoding.

OpenAI (GPT-4o+)	**Нативная**	С августа 2024, JSON Schema Strict Mode
Anthropic (Claude)	**Нативная**	С начала 2026
Google Gemini	**Нативная**	JSON Schema с января 2026
DeepSeek V3	**JSON Mode**	Валидный JSON, без строгой схемы
DeepSeek R1	**Частично**	R1-0528. Вроде есть, но нельзя принудить вызвать инструмент, нет гарантий
GigaChat	**Эмуляция**	Через function calling, 1 вызов/запрос, костыли
Ollama / vLLM	**Локально**	Grammar-based constrained decoding

Самовосстанавливающийся цикл для агента

- генерация → очистка → парсинг → проверка → коррекция
- 1 генерация + 3 коррекции по умолчанию



Код – Structured output своими руками

```
val json = try {
    jsonParser.parseToJsonElement(sanitizedMessage) as JsonObject
} catch (e: SerializationException) {

    if (iteration >= totalIterations - 1) {

        intermediateResult = ScenarioValidationResult.Invalid(
            input,
            null,
            "Ответ модели не является валидным JSON.
        )
        return@repeat
    }

    val requestString = makeCorrectionPrompt(
      "Invalid JSON format: ${e.message}«
    )

    responses = requestLLMMultiple(requestString)
    return@repeat //continue to the next iteration
}
```

Очистка от мусора

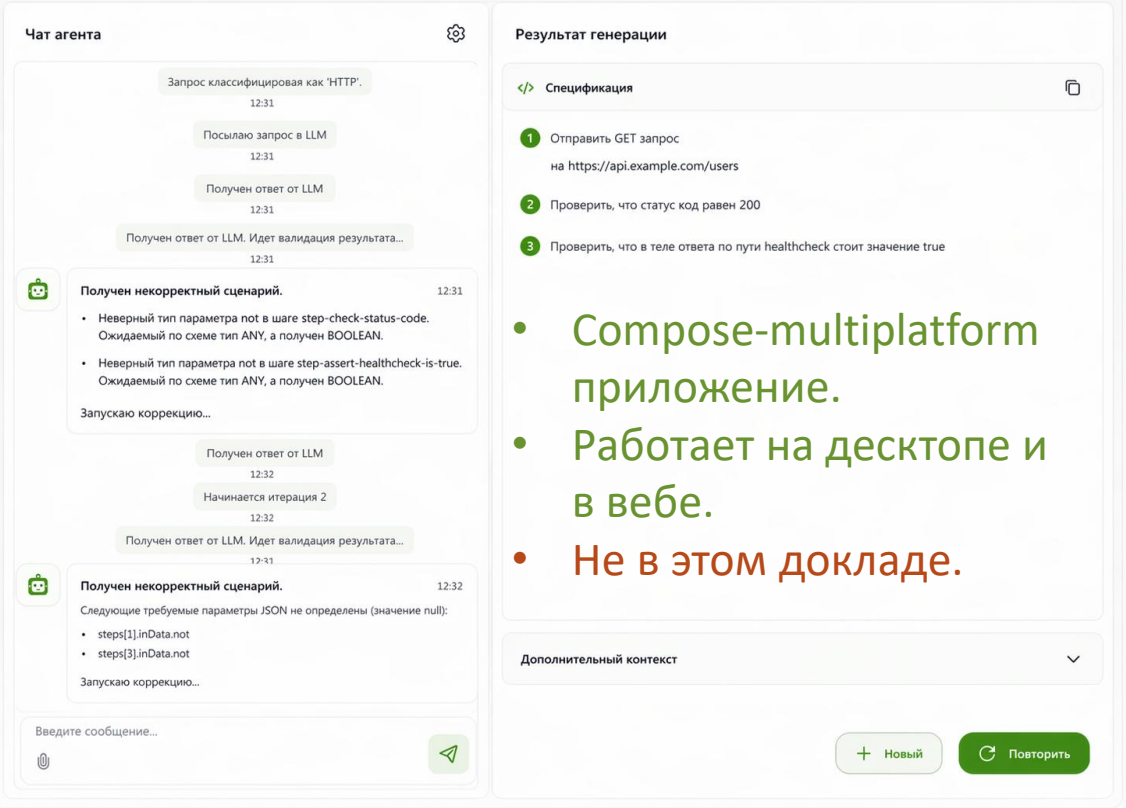
Парсинг

Возврат ошибки обратно в LLM

Повтор цикла по итерациям

Kotlinx-serialization как источник истины

Пример работы цикла



Как это работает?

- Задаем спецификацию.
- Получаем невалидный результат.
- Запрашиваем коррекцию.
- Повторяем.

Где заканчивается DSL у Koog?

```
val agent = AIAgent(
    promptExecutor = executor,
    llmModel = model,
    strategy = functionalStrategy(
        messageFlow,
        correctionCycles,
        schemaContext,
        exampleContext
    ),
    systemPrompt = SYSTEM_PROMPT
)

val result = agent.run(data)
```

Koog дает API
и взаимодействие компонент

Мы даем логику
и кастомные интеграции

Как запустить? Уже ведь можно, правда? (Нет)

Вариант здорового человека

- Покупаем подписку на OpenRouter (поддерживается Koog сразу).
- Тестируем и подбираем нужную модель.
- Покупаем продакшен подписку на нужную модель.
- Ну или запускаем у себя Qwen.

Вариант корпоративный

- Месяц получаем доступы к Гигачату.
- У него свой API, который мы не можем нормально протестировать.
- Еще у него свои сертификаты.
- А еще у пользователей своих токенов не будет, надо как-то все это зашить в код.
- Поэтому нужны еще несколько шагов…

Значит пишем кастомный провайдер!
Может ли это наш друг?

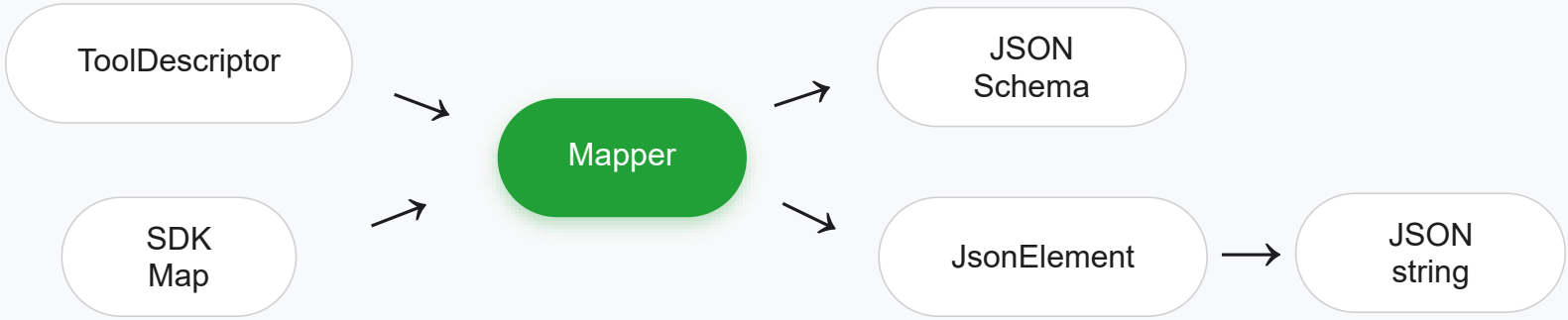
1) Пишем свой PromptExecutor – Koog не знает про российских провайдеров LLM

```
interface PromptExecutor {
    suspend fun execute(...)
    fun executeStreaming(...): Flow<StreamFrame>
}
```

- GigaChat как пример кастомного провайдера LLM
- Нужны синхронный и и потоковый режимы
- Нужны транспорт, авторизация и свой маппинг
- Как раз здесь уместна интеграция со SpringAI и готовым spring-ai-gigachat (но это вне рамок нашего времени)

2) Дружим схемы инструментов Koog и GigaChat

- Koog работает с типизированными описаниями схем инструментов
- GigaChat ожидает структуру, похожую на JSON Schema

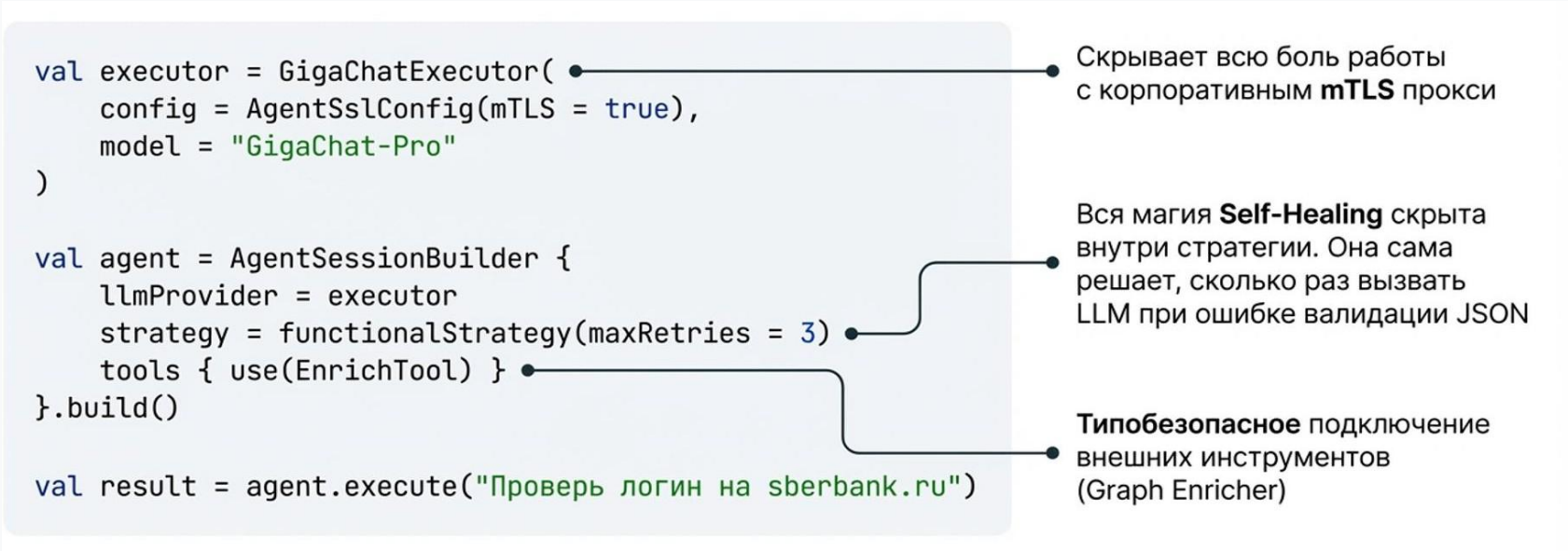


- Ответы SDK приходят как Java словари
- Нужен двусторонний рекурсивный преобразователь

3) Стыкуем потоковый вызов функций

- Имя и аргументы функции приходят частями
- Один вызов функции может быть размазан по нескольким сегментам потока
- Без накопления получаем битый JSON

Мы помогли другу с GigaChat, он помог нам.



Теперь выводы

Наш пилот уже летит...

...самолет его может быть догонит

Что получилось

- Сделали функциональную и графовую логику (от графовой пока отказались).
- Свой (более надежный) вариант структурного ответа. Kotlinx-serialization для валидации формата и свой валидатор для связей.
- Описание элементов контекста вместо тулов.
- Свой адаптер для GigaChat и его прокси.
- Кастомные настройки для безопасности в контуре банка.

Какие выводы

- Koog дает удобный функциональный интерфейс для написания агентов в виде функций.
- Специального знания LLM не требуется. Можно писать обычные функции (главное не забывать валидировать результаты).
- Агент – это не обязательно чат (хотя большинство советов в интернете именно про чаты).
- Попробовали на разных LLM и в результате запустили на GigaChat. Специализированный агент куда больше зависит от своей логики, чем от типа LLM.
- Если у вас уже есть Spring в проекте, Spring AI 1.1 — тоже разумный выбор. Мы шли от другой отправной точки: отдельное приложение без Spring в кодовой базе.

SPC Scientific Programming Centre

https://sciprog.center

Полезные ссылки:

- Kotlin комьюнити: https://t.me/kotlin_lang
- Российское Kotlin комьюнити: https://t.me/kotlin_russia
- Koog: https://docs.koog.ai
- Compose Multiplatform: https://kotlinlang.org/compose-multiplatform/