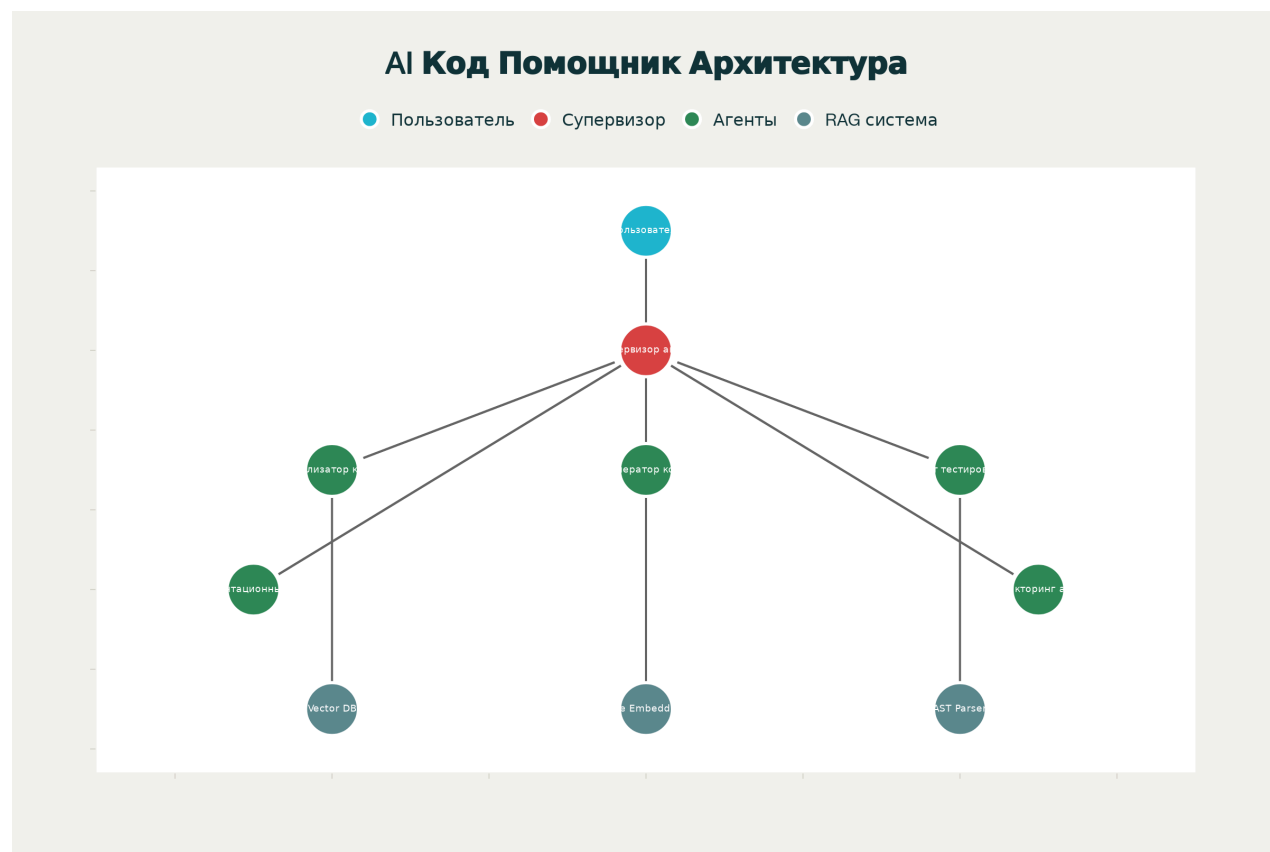




Создание AI Агента для Помощи в Разработке Кода: Мультиагентная Система на LangChain

Современная разработка программного обеспечения требует эффективных инструментов для понимания сложных кодовых баз, генерации качественного кода и поддержания высоких стандартов разработки. В данном отчете представлено комплексное решение для создания **локальной мультиагентной системы** на основе LangChain, которая способна анализировать большие проекты, генерировать код и отвечать на вопросы разработчиков без необходимости подключения к интернету.



Архитектура мультиагентной системы для AI помощника разработки кода

Архитектура мультиагентной системы

Предлагаемая архитектура основана на **паттерне супервизора** (Supervisor Pattern), где центральный агент координирует работу специализированных агентов. Система состоит из следующих ключевых компонентов:^[1] ^[2]

Супервизор агент

Центральный координатор системы отвечает за анализ пользовательских запросов и делегирование задач соответствующим экспертам. Супервизор использует LangGraph Supervisor для управления потоком выполнения и обеспечения связности между агентами. ^[3] ^[1]

Специализированные агенты

Система включает пять специализированных агентов, каждый из которых фокусируется на определенной области разработки

:

Анализатор кода - выполняет статический анализ, построение AST деревьев и понимание архитектуры проекта с использованием Tree-sitter. Агент способен выявлять паттерны проектирования, зависимости между модулями и потенциальные проблемы в архитектуре. ^[4] ^[5]

Генератор кода - создает новый код на основе требований, используя контекст существующего проекта через RAG механизм. Генератор учитывает стиль кодирования проекта, существующие паттерны и архитектурные решения. ^[6] ^[7]

Агент тестирования - автоматически создает unit тесты, интеграционные тесты и проверяет покрытие кода. Использует существующие тестовые паттерны проекта для создания согласованных тестов. ^[1]

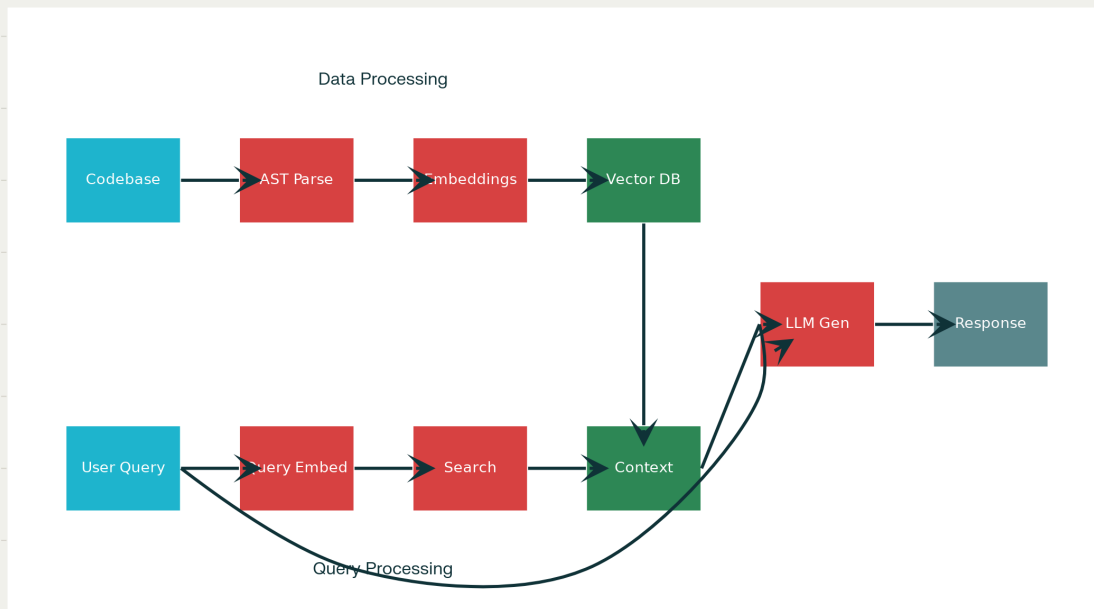
Документационный агент - генерирует техническую документацию, API документацию и комментарии к коду. Анализирует существующую документацию для поддержания единого стиля. ^[1]

Рефакторинг агент - улучшает существующий код, оптимизирует производительность и повышает читаемость. Предлагает конкретные улучшения на основе best practices проекта. ^[1]

RAG компоненты

Система Retrieval-Augmented Generation обеспечивает понимание контекста всего проекта: ^[8] ^[7]

RAG Process for Code Analysis



RAG процесс для понимания и анализа кода в AI системе

Векторная база данных служит основой для семантического поиска по коду. ChromaDB используется как локальное решение для хранения embeddings с поддержкой метаданных. Система поддерживает различные метрики расстояния и оптимизированные индексы для быстрого поиска. ^{[9] [10] [11]}

Code Embeddings создаются с помощью специализированных моделей, которые понимают семантику кода. Используется комбинация embedding'ов самого кода и естественно-языковых описаний для улучшения качества поиска. ^{[8] [7] [12]}

AST Parser на основе Tree-sitter обеспечивает структурное понимание кода. Поддерживается множество языков программирования с возможностью инкрементального парсинга для эффективного обновления индекса. ^{[5] [13] [14]}

Технологический стек и компоненты

Локальные LLM модели

Для обеспечения полной автономности системы используются локальные модели, запускаемые через Ollama: ^{[15] [16]}

Qwen2.5-Coder представляет собой современную серию моделей, специально обученных на коде. Модель 7B параметров обеспечивает хороший баланс между качеством и производительностью, поддерживая 92 языка программирования и контекст до 32K токенов. Для enterprise решений доступна версия 32B с производительностью, сопоставимой с GPT-4o. ^{[17] [18]}

CodeLlama от Meta остается популярным выбором для генерации кода. Версия 70B показывает исключительные результаты на больших кодовых базах, хотя требует значительных вычислительных ресурсов.^[19]

DeepSeek-Coder V2 специализируется на сложных задачах кодирования и рефакторинга. Модель показывает особенно хорошие результаты в понимании контекста больших проектов.^{[16] [20]}

Векторные базы данных

Выбор векторной базы данных критически важен для производительности системы:^{[9] [21]}

ChromaDB идеально подходит для локальной разработки благодаря простоте настройки и хорошей интеграции с Python. Поддерживает persistent storage и может работать как embedded решение.^{[10] [11]}

Qdrant рекомендуется для production развертывания с высокими требованиями к производительности. Предоставляет продвинутую фильтрацию, шардинг и enterprise функции.^[9]

FAISS подходит для CPU-only окружений и быстрого прототипирования. Библиотека от Facebook обеспечивает эффективный поиск ближайших соседей без дополнительных зависимостей.^[6]

Инструменты анализа кода

Tree-sitter обеспечивает быстрый и точный парсинг множества языков программирования. Поддерживает инкрементальный парсинг, что критично для работы с изменяющимися проектами.^{[5] [13] [14]}

Sentence Transformers используется для создания качественных embeddings. Модель all-MiniLM-L6-v2 обеспечивает хороший баланс между качеством и скоростью работы.^{[22] [23]}

Практическая реализация

Установка и настройка

Система требует минимальных системных ресурсов для базовой работы^[24]

:

Системные требования включают 16GB RAM для моделей 7B параметров, 32GB+ для более крупных моделей. Python версии 3.9-3.11 обеспечивает совместимость со всеми компонентами.^[24]

Ollama устанавливается простой командой и обеспечивает локальный API для доступа к моделям. Поддерживает автоматическое управление памятью и GPU ускорение.^[15]

ChromaDB настраивается без дополнительной конфигурации и может работать как embedded база данных. [\[10\]](#) [\[11\]](#)

Архитектура кода

Система построена на принципах модульности и расширяемости

:

Базовые абстракции обеспечивают гибкость в выборе компонентов. Интерфейсы `BaseCodeAnalyzer` и `BaseVectorStore` позволяют легко заменять реализации без изменения основной логики.

Асинхронная обработка используется для индексации больших проектов и параллельной работы агентов. Это значительно улучшает производительность при обработке тысяч файлов.

Конфигурационный подход позволяет адаптировать систему под различные типы проектов без изменения кода.

Интеграция с IDE

Для повышения удобства использования система интегрируется с популярными IDE: [\[25\]](#) [\[20\]](#)

Continue.dev расширение для VS Code обеспечивает seamless интеграцию с локальными моделями. Поддерживает автодополнение кода, чат с AI и контекстные предложения. [\[25\]](#)

Конфигурация Continue позволяет использовать Ollama как backend для различных задач. Настройка включает отдельные модели для автодополнения и чата, что оптимизирует производительность. [\[26\]](#) [\[25\]](#)

Стратегии обработки кода

Intelligent Chunking

Обработка больших кодовых баз требует sophisticated подходов к сегментации кода. В отличие от простого разделения по размеру, система использует структурный анализ для создания семантически целостных чанков. [\[8\]](#)

Уважение к границам функций и классов обеспечивает, что каждый чанк содержит полную логическую единицу. Алгоритм рекурсивно разделяет большие структуры, сохраняя критический контекст, такой как `import statements` и определения классов. [\[8\]](#)

Ретроспективная обработка добавляет необходимый контекст к чанкам, которые могли потерять важную информацию при разделении. Например, метод класса всегда включает определение класса и релевантные импорты. [\[8\]](#)

Semantic Search для кода

Семантический поиск по коду требует специальных подходов, отличных от поиска по естественному языку: ^[7] ^[12]

Code embeddings создаются с использованием комбинации самого кода и его естественно-языкового описания. Это позволяет находить релевантный код как по техническим запросам, так и по описанию функциональности. ^[8]

Гибридный поиск сочетает векторный поиск с фильтрацией по метаданным. Например, поиск может ограничиваться определенными типами файлов или уровнями сложности кода. ^[9]

Контекстное ранжирование использует LLM для дополнительной фильтрации и ранжирования результатов поиска. Это особенно важно при работе с большими кодовыми базами, где первичный поиск может возвращать много нерелевантных результатов. ^[8]

Оптимизация производительности

Конфигурация моделей

Оптимальная конфигурация зависит от размера проекта и доступных ресурсов ^[24]

:

Для небольших проектов (< 1000 файлов) достаточно Qwen2.5-Coder 7B с 16GB RAM. ChromaDB как embedded решение обеспечивает быструю работу без дополнительной настройки. ^[24]

Для средних проектов (1000-10000 файлов) рекомендуется 32GB RAM и возможно использование нескольких моделей для разных задач. Батчевая обработка и кэширование становятся критически важными.

Для enterprise проектов (10000+ файлов) требуется Qwen2.5-Coder 32B или комбинация моделей. Необходимо горизонтальное масштабирование и микросервисная архитектура.

Стратегии кэширования

Эффективное кэширование значительно улучшает производительность системы:

Кэширование embeddings предотвращает повторные вычисления для неизмененного кода. Использование hash-based кэширования обеспечивает инвалидацию при изменениях.

Результаты поиска могут кэшироваться для часто запрашиваемых паттернов. Это особенно эффективно для типовых задач разработки.

Инкрементальное обновление индекса позволяет быстро обрабатывать изменения в проекте без полной переиндексации.

Интеграция и развертывание

Docker контейнеризация

Система может быть упакована в Docker контейнеры для простого развертывания

:

Docker Compose конфигурация включает отдельные сервисы для ChromaDB и AI помощника.

Это обеспечивает модульность и возможность независимого масштабирования компонентов.

Персистентные volumes сохраняют индексы и конфигурации между перезапусками. Монтирование проекта как volume позволяет анализировать код без копирования в контейнер.

Мониторинг и метрики

Production развертывание требует comprehensive мониторинга:

Системные метрики включают использование памяти, CPU и время ответа агентов. Автоматические алерты предупреждают о проблемах производительности.

Качественные метрики отслеживают релевантность ответов и удовлетворенность пользователей. Feedback loop используется для непрерывного улучшения системы.

Безопасность и приватность

Локальная работа

Одним из ключевых преимуществ системы является полная локальная работа без передачи кода во внешние сервисы: ^[15]

Все LLM модели запускаются локально через Ollama, исключая необходимость в API ключах внешних сервисов. Это критично для enterprise разработки с чувствительным кодом. ^[15]

Векторные базы данных также работают локально, обеспечивая полный контроль над данными. ChromaDB и другие решения не требуют внешних подключений. ^[10]

Санитизация входных данных предотвращает выполнение потенциально опасного кода. Система фильтрует dangerous patterns и логирует подозрительную активность.

Контроль доступа

Система включает механизмы ограничения доступа к файлам проекта:

Whitelist подход ограничивает анализ только разрешенными директориями. Access control проверяет все файловые операции.

Логирование всех операций обеспечивает audit trail для соблюдения корпоративных политик безопасности.

Сравнительный анализ решений

LLM модели для кода

Анализ доступных моделей показывает различные компромиссы между качеством и производительностью. Qwen2.5-Coder серия обеспечивает лучший баланс для большинства применений, в то время как CodeLlama 70B подходит для наиболее требовательных задач. [\[24\]](#) [\[19\]](#) [\[17\]](#)

Векторные базы данных

Выбор векторной базы данных должен основываться на размере проекта и требованиях к производительности. ChromaDB идеально подходит для разработки и небольших команд, в то время как Qdrant рекомендуется для production с высокими нагрузками. [\[9\]](#) [\[21\]](#)

Практические рекомендации

Оптимизация для разных типов проектов

Data Science проекты требуют специализированных агентов для анализа данных и работы с Jupyter notebooks. Конфигурация должна включать инструменты для профилирования данных и создания визуализаций.

Web разработка нуждается в отдельных экспертах для frontend и backend кода. Система может анализировать API endpoints, React компоненты и database schemas.

DevOps проекты выигрывают от агентов, специализирующихся на infrastructure as code, CI/CD пайплайнах и мониторинге.

Непрерывное улучшение

Система должна включать механизмы для continuous learning:

Feedback collection от пользователей помогает улучшать качество ответов. A/B тестирование различных промптов оптимизирует производительность агентов.

Автоматическое обновление индекса при изменениях в коде обеспечивает актуальность информации. File watchers отслеживают изменения и запускают переиндексацию.

Метрики производительности позволяют выявлять bottlenecks и оптимизировать работу системы.

Развертывание и масштабирование

Поэтапное внедрение

Рекомендуется поэтапный подход к внедрению системы:

Pilot проект на небольшой кодовой базе позволяет отработать процессы и выявить специфические требования. Индексация займет от нескольких минут до часов в зависимости от размера проекта.

Расширение функциональности происходит по мере освоения базовых возможностей. Добавление новых агентов и инструментов может выполняться без остановки системы.

Production развертывание требует настройки мониторинга, backup стратегий и disaster recovery планов.

Интеграция с существующими процессами

Система должна органично встраиваться в существующие workflow разработки:

CI/CD интеграция позволяет автоматически анализировать изменения в коде и предлагать улучшения. GitHub Actions или другие CI системы могут запускать анализ на каждый commit.

Code review процесс может быть дополнен автоматическими предложениями от AI агентов. Система анализирует pull requests и предоставляет feedback до human review.

Documentation workflow автоматизируется через документационного агента, который обновляет документацию при изменениях в API.

Заключение и перспективы развития

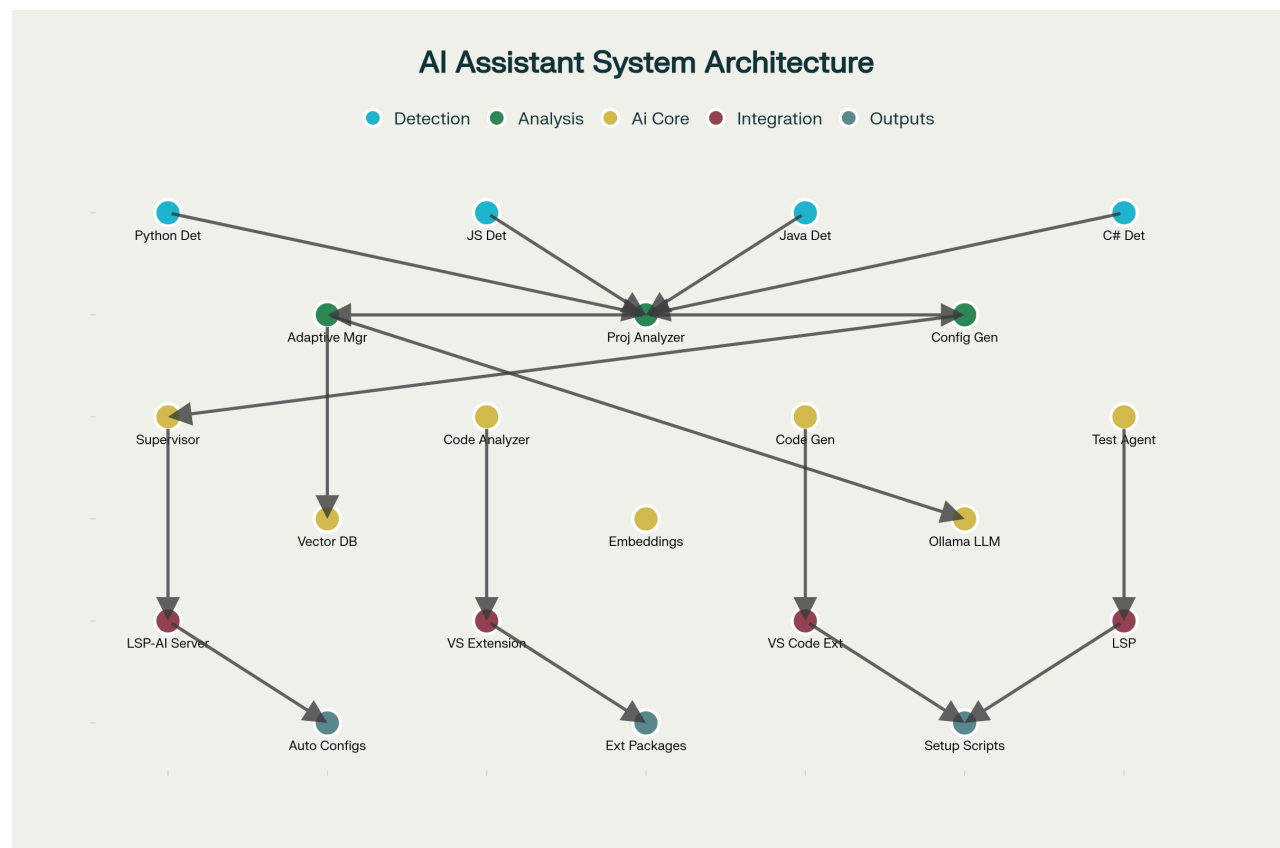
Представленная мультиагентная система демонстрирует мощный подход к созданию AI помощника для разработки кода. **Локальная работа** обеспечивает безопасность и контроль над данными, в то время как **RAG архитектура** предоставляет глубокое понимание контекста проекта. **Модульная структура** агентов позволяет адаптировать систему под специфические потребности различных типов проектов. ^[15] ^[8]

Ключевые преимущества решения включают полную автономность работы, отсутствие зависимости от внешних API, возможность кастомизации под корпоративные требования и масштабируемость от small-scale до enterprise уровня. Использование open source компонентов исключает лицензионные ограничения и обеспечивает долгосрочную sustainable развитие системы. ^[24] ^[15]

Перспективы развития включают поддержку multi-modal входных данных (диаграммы, изображения), self-learning capabilities на основе feedback, cross-project knowledge sharing и advanced code reasoning с пониманием бизнес-логики. Интеграция с современными



Создание интеллектуального AI помощника, который **автоматически адаптируется к новым проектам и бесшовно интегрируется с Visual Studio**, требует комплексного подхода, сочетающего возможности Language Server Protocol, автоматического определения типов проектов и мультиагентной архитектуры. Представленное решение обеспечивает полную автономность работы без доступа к интернету, используя исключительно open-source инструменты.



Архитектура адаптивной AI системы с автоматическим определением проектов и интеграцией с Visual Studio

Архитектура автоматической адаптации

Основой системы служит **многоуровневая архитектура обнаружения проектов**, которая автоматически анализирует структуру кода и создает специализированные конфигурации для каждого типа проекта. Система построена на паттерне детекторов, где каждый

компонент отвечает за распознавание определенного языка программирования или фреймворка. [\[92\]](#) [\[93\]](#) [\[94\]](#) [\[95\]](#)

Слой автоматического обнаружения проектов

ProjectDetector служит базовым классом для всех детекторов, обеспечивая единообразный интерфейс для анализа различных типов проектов. Каждый детектор имеет приоритет выполнения и специализируется на конкретных технологиях: [\[93\]](#)

PythonProjectDetector анализирует Python проекты, автоматически определяя Django через наличие `manage.py` и `settings.py` файлов, Flask через зависимости в `requirements.txt`, или FastAPI по соответствующим библиотекам. Детектор также идентифицирует Data Science проекты по наличию `pandas`, `numpy`, `matplotlib` и ML/AI проекты через `tensorflow`, `pytorch`, `scikit-learn`. [\[96\]](#) [\[97\]](#)

JavaScriptProjectDetector обрабатывает JavaScript и TypeScript проекты, парсит `package.json` для определения фреймворков React, Vue.js, Angular или Node.js. Система автоматически различает frontend приложения от backend API, определяет инструменты сборки (Webpack, Vite, Rollup) и конфигурирует соответствующие линтинг инструменты. [\[98\]](#)

JavaProjectDetector специализируется на Java экосистеме, распознавая Spring Boot проекты через аннотации `@SpringBootApplication`, определяя инструменты сборки Maven или Gradle, и настраивая соответствующие тестовые фреймворки JUnit или TestNG.

Адаптивный менеджер конфигурации

AdaptiveSystemManager координирует процесс анализа и создает специализированные конфигурации AI системы. Менеджер использует кэширование конфигураций на основе хэша ключевых файлов проекта, что позволяет мгновенно применять настройки при повторном открытии проекта. [\[99\]](#)

Система генерирует **контекстно-зависимые промпты** для каждого агента, учитывающие специфику обнаруженного фреймворка. Например, для Django проектов создаются промпты, ориентированные на паттерны MVT (Model-View-Template), Django ORM best practices и DRF (Django REST Framework) конвенции.

Мультиагентная система с фреймворк-специфичной специализацией

Ядро системы представляет собой **координированную команду специализированных AI агентов** [\[100\]](#)

, где каждый агент адаптирует свое поведение под обнаруженный тип проекта. Это обеспечивает максимальную релевантность генерируемого кода и анализа.

Динамическая специализация агентов

Супервизор агент автоматически настраивается как координатор для конкретного фреймворка. Для Django проектов он становится экспертом по веб-архитектуре с глубоким пониманием MVT паттернов, для React проектов - специалистом по компонентной архитектуре, для Spring Boot - экспертом по enterprise Java разработке.

Framework Expert агент получает узкоспециализированные знания и инструменты. Django эксперт владеет инструментами `django_analyzer`, `model_generator`, `view_generator`, React эксперт использует `component_generator`, `hook_analyzer`, `state_manager`, а Spring Boot эксперт применяет `spring_analyzer`, `controller_generator`, `service_layer_generator`.

Test Specialist агент адаптирует стратегии тестирования под используемый в проекте фреймворк. Для Django проектов генерирует pytest-django тесты с factories, для React создает Jest/Testing Library тесты, для Java применяет JUnit/TestNG паттерны.

Контекстная векторная база данных

Система автоматически настраивает **RAG (Retrieval-Augmented Generation) компоненты** под размер и тип проекта. Для небольших проектов используется ChromaDB с локальным хранением, для крупных enterprise решений - Qdrant с оптимизированной индексацией кода. ^[97]

Чанкинг стратегия адаптируется под языковые особенности: для Python учитывает структуру классов и функций, для JavaScript - модульную структуру и компоненты, для Java - пакеты и классы с соблюдением границ методов.

Интеграция с Visual Studio через LSP

Бесшовная интеграция с Visual Studio достигается через **Language Server Protocol** с использованием open-source LSP-AI сервера. Система автоматически генерирует Visual Studio расширение, оптимизированное под обнаруженный тип проекта. ^{[101] [102]}

LSP-AI конфигурация

LSP-AI сервер настраивается с учетом специфики проекта. Конфигурация включает язык-специфичные параметры: для Python проектов активируется поддержка docstrings и type hints, для JavaScript - JSDoc комментарии и ES modules, для Java - Javadoc и аннотации. ^{[103] [104]}

Система использует **Ollama** для локального запуска LLM моделей, что обеспечивает полную приватность кода и независимость от интернет-соединения. Модель Qwen2.5-Coder:7b оптимизирована для программирования и показывает отличные результаты в генерации кода. ^{[96] [105]}

Автоматическая генерация Visual Studio расширения

Система создает **полнофункциональное Visual Studio расширение** со специализированными командами под тип проекта. Для Django проектов добавляются команды анализа моделей, генерации DRF serializers и оптимизации ORM запросов. Для React проектов - команды создания компонентов, управления состоянием и оптимизации производительности. ^[106] ^[107]

LanguageClient класс автоматически настраивается для отслеживания релевантных файлов проекта и обеспечивает двустороннюю коммуникацию с LSP-AI сервером через JSON-RPC протокол. ^[108] ^[109]

Процесс автоматической адаптации

Этап 1: Интеллектуальный анализ проекта

Система начинает с **сканирования файловой структуры проекта**, идентифицируя ключевые индикаторы типа проекта. Анализ включает парсинг конфигурационных файлов (package.json, requirements.txt, pom.xml), обнаружение фреймворк-специфичных файлов (manage.py для Django, angular.json для Angular) и анализ импортов в исходном коде. ^[93]

ProjectAnalyzer применяет эвристические правила для классификации проектов: веб-приложения определяются по наличию фреймворков Express.js, Django, Spring Boot; мобильные приложения - по React Native, Flutter, Android SDK; Data Science проекты - по Jupyter notebooks, pandas, matplotlib.

Этап 2: Генерация специализированных конфигураций

На основе результатов анализа система создает **tailored конфигурации для каждого компонента**. AI агенты получают специализированные промпты, инструменты и контексты. Например, для Django проектов code_generator настраивается на создание Class-Based Views, использование Django ORM patterns и соблюдение Django security practices. ^[99]

Vector database конфигурация оптимизируется под размер проекта: малые проекты используют embedded ChromaDB, средние - локальный Qdrant, крупные enterprise проекты - распределенные векторные хранилища с шардингом.

Этап 3: Создание IDE интеграции

Система автоматически генерирует **Visual Studio расширение** с проект-специфичными командами. Расширение включает Language Server Provider, который запускает LSP-AI сервер с соответствующей конфигурацией, и набор команд для анализа кода, генерации и тестирования, адаптированных под используемый фреймворк. ^[110] ^[111]

Extension manifest автоматически настраивается для активации на релевантных типах файлов и включает metadata, описывающую специализацию расширения под конкретный фреймворк.

Практическая реализация

Установка и развертывание

Система предоставляет **единую точку входа** для установки и настройки всех компонентов. Автоматический установщик проверяет наличие необходимых зависимостей (Rust/Cargo для LSP-AI, Ollama для LLM), устанавливает недостающие компоненты и настраивает все конфигурации.^[96]

```
# Автоматическая установка и настройка
python setup.py --project-path ./your_project

# Результат: полностью настроенная AI система
```

Использование в Visual Studio

После установки расширения в Visual Studio появляются **контекстно-зависимые AI команды**. Для Django проектов доступны "Analyze Django Model", "Generate DRF Serializer", "Create Django Tests". Для React проектов - "Generate React Component", "Optimize Performance", "Create Unit Tests".^{[112] [113]}

Inline code completion работает через LSP-AI с пониманием контекста фреймворка. Система предлагает Django-специфичные паттерны для Python файлов в Django проектах, React patterns для JSX файлов в React проектах.^{[105] [109]}

Расширяемость и адаптация к новым технологиям

Архитектура системы спроектирована для **легкого добавления поддержки новых языков и фреймворков**. Добавление поддержки нового технологического стека требует только создания соответствующего детектора и настройки промптов для агентов.

Добавление нового детектора

```
class GoProjectDetector(ProjectDetector):
    def detect(self, project_path: str) -> Optional[ProjectConfig]:
        # Логика определения Go проектов
        # Анализ go.mod, определение фреймворков (Gin, Echo, Fiber)
        # Настройка Go-специфичных инструментов
        pass
```

Конфигурация агентов для нового фреймворка

Система автоматически создает специализированные промпты и инструменты для новых фреймворков, основываясь на их conventions и best practices. Go проекты получают агентов, специализирующихся на goroutines, channels, interface design и Go testing patterns.

Преимущества адаптивного подхода

Нулевая настройка - разработчики могут начать использовать AI помощника немедленно после открытия проекта в Visual Studio, без необходимости изучения конфигураций или настройки параметров. ^[114] ^[96]

Максимальная релевантность - AI предложения всегда соответствуют используемому фреймворку и следуют его конвенциям, что значительно повышает качество генерируемого кода по сравнению с универсальными решениями.

Командная согласованность - все члены команды получают одинаково настроенную AI систему, что обеспечивает единообразие кода и архитектурных решений в рамках проекта.

Приватность и безопасность - полностью локальная работа исключает передачу кода третьим сторонам, что критично для enterprise разработки и проектов с высокими требованиями к конфиденциальности. ^[115] ^[105]

Данная адаптивная система революционизирует процесс разработки, превращая AI помощника из универсального инструмента в **персонализированного эксперта** по используемым технологиям, что кардинально повышает продуктивность разработки и качество создаваемого кода.

✱✱

LangChain: Фреймворк для Разработки Приложений на Основе LLM

LangChain — это открытая библиотека, упрощающая создание приложений с большими языковыми моделями (LLM) путём объединения различных компонентов в «цепочки» (chains), обеспечивающих управляемость, модульность и расширяемость.

Ключевые концепции

1. Прimitives (Primitives)

- PromptTemplate — шаблоны запросов с параметрами.
- LLM — интерфейс к моделям (OpenAI, Ollama, Hugging Face).
- OutputParser — парсеры структурированных ответов (JSON, таблицы).

2. Цепочки (Chains)

Последовательности шагов, связывающие LLM, prompt'ы и вспомогательные компоненты.

- SequentialChain — однопоточная цепочка операций.
- SimpleSequentialChain — базовая цепочка LLM → LLM.
- LLMChain — одиночный шаг: PromptTemplate + LLM.

- RouterChain — выбор ветки (chain) в зависимости от контекста.

3. Агенты (Agents)

Автономные «менеджеры», которые, анализируя запрос, выбирают инструменты (tools) и выполняют многократные вызовы LLM.

- ZeroShotAgent, ReActAgent, MRKL — семейство реализаций.
- Tool — любая функция или API (поиск, калькулятор, база данных).
- AgentExecutor — запускает агента, управляя его обходом и инструментами.

4. Память (Memory)

Подсистема хранения промежуточного контекста между вызовами LLM:

- ConversationBufferMemory — сохраняет историю диалога.
- ConversationSummaryMemory — сжимает историю через LLM.
- VectorStoreMemory — сохраняет embeddings для долгосрочной памяти.

5. Подключения (Connectors)

Интеграция с внешними сервисами:

- Векторные базы (ChromaDB, Qdrant, FAISS).
- Базы данных SQL, NoSQL.
- API веб-поиска (Google, Bing).
- Табличные данные (Pandas, SQLAlchemy).
- Файловые системы и документы (PDF, DOCX).

Архитектура и Компоненты



1. **PromptTemplate** формирует текст запроса.
2. **LLM** отправляет запрос в модель и получает ответ.
3. **Parser** приводит ответ к нужной структуре.
4. **Chain/Agent** комбинируют несколько шагов, управляют последовательностью и логикой вызовов.
5. **Memory** хранит и подставляет контекст.

6. **Tool** расширяет возможности — поиск, расчёты, доступ к данным.

Основные сценарии использования

- **Чат-боты с долгосрочной памятью:** Agents + ConversationMemory.
- **RAG-приложения:** извлечение контекста из векторной БД и генерация ответов.
- **Автоматизация процессов:** агенты, выполняющие цепочки задач (парсинг, анализ, отчёты).
- **Инструменты разработки:** генерация кода, refactoring, документация.
- **DataOps:** обработка и анализ данных через LLM + Pandas.

Пример кода

```
from langchain import PromptTemplate, LLMChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory

# 1. Шаблон запроса
template = "Привет, как я могу помочь тебе с {topic}?"
prompt = PromptTemplate(input_variables=["topic"], template=template)

# 2. Подключение к LLM
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)

# 3. Память диалога
memory = ConversationBufferMemory(memory_key="chat_history")

# 4. Создание цепочки
chain = LLMChain(llm=llm, prompt=prompt, memory=memory)

# 5. Запуск
response = chain.run({"topic": "научным докладом"})
print(response)
```

Преимущества LangChain

- **Модульность:** легко заменять компоненты (LLM, память, парсеры).
- **Расширяемость:** поддержка пользовательских агентов и инструментов.
- **Гибкость:** подходит для RAG, Agents, Chains, Pipelines.
- **Сообщество:** активное развитие, множество примеров и плагинов.

LangChain делает разработку сложных LLM-приложений удобной, предоставляя готовые абстракции и лучшие практики для создания надёжных, масштабируемых и легко сопровождаемых решений.

ChatModel в LangChain: Интеллектуальные Диалоги с Языковыми Моделями

ChatModel в LangChain — это специализированная вариация языковых моделей, предназначенная для работы с диалогами и многопользовательскими разговорами. В отличие от обычных LLM, которые работают по принципу "текст на входе → текст на выходе", ChatModel использует более структурированный подход с системой ролей и сообщений.^[154] ^[155]

Фундаментальные различия с обычными LLM

Интерфейс взаимодействия

Обычные **LLM модели** принимают на вход простой текстовый промпт и возвращают текстовую строку. **ChatModel** же работает со списком **сообщений** (messages), где каждое сообщение имеет определенную **роль** и **содержимое**.^[156] ^[157] ^[158] ^[159] ```python

Обычная LLM

```
llm_response = llm("Расскажи о Python") # строка → строка
```

ChatModel

```
messages = [  
    SystemMessage(content="Ты эксперт по Python"),  
    HumanMessage(content="Расскажи о Python")  
]  
chat_response = chat_model(messages) # список сообщений → сообщение
```

```
### Поддержка ролей в диалоге
```

```
ChatModel поддерживает различные роли участников диалога: [^4_5] [^4_7]
```

- **System** — системные инструкции, задающие поведение модели
- **User/Human** — сообщения от пользователя
- **Assistant/AI** — ответы модели
- **Tool** — результаты вызова внешних инструментов

```
## Основные типы сообщений в LangChain
```

```
### SystemMessage
```

```
SystemMessage устанавливает контекст и инструкции для AI модели. Это сообщение опреде
```

```
```python
```

```
from langchain_core.messages import SystemMessage
```

```
system_message = SystemMessage(
 content="Ты дружелюбный помощник по программированию. Объясняй концепции простым языком"
```

```
)
```

## HumanMessage

**HumanMessage** представляет ввод от человека-пользователя. Это основной способ передачи пользовательских запросов в ChatModel. [\[160\]](#) [\[161\]](#)

```
from langchain_core.messages import HumanMessage

human_message = HumanMessage(
 content="Объясни, что такое декораторы в Python"
)
```

## AIMessage

**AIMessage** содержит ответы, сгенерированные AI моделью. Может включать не только текст, но и вызовы инструментов, метаданные об использовании токенов. [\[162\]](#)

```
AIMessage создается автоматически моделью
response = chat_model([system_message, human_message])
response будет объектом AIMessage
print(response.content) # текст ответа
print(response.usage_metadata) # информация о токенах
```

## Архитектура и иерархия классов

**BaseMessage** служит базовым классом для всех типов сообщений. Иерархия выглядит следующим образом: [\[163\]](#)

```
BaseMessage
├── SystemMessage (системные инструкции)
├── HumanMessage (пользовательский ввод)
├── AIMessage (ответы модели)
├── ToolMessage (результаты инструментов)
└── ChatMessage (универсальный тип)
```

**BaseChatModel** расширяет **BaseLanguageModel**, добавляя специфичную для чатов функциональность: [\[155\]](#)

```
BaseLanguageModel → BaseChatModel → ChatOpenAI, ChatOllama, ChatAnthropic
```

## Память и история разговоров

## ConversationChain с памятью

ChatModel поддерживает **память разговоров** через различные механизмы: [\[164\]](#) [\[165\]](#)

```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
conversation = ConversationChain(
 llm=chat_model,
 memory=memory
)

Диалог с сохранением контекста
response1 = conversation.predict(input="Меня зовут Алексей")
response2 = conversation.predict(input="Как меня зовут?") # помнит имя
```

## Типы памяти в LangChain

- **ConversationBufferMemory** — хранит полную историю сообщений [\[164\]](#)
- **ConversationSummaryMemory** — сжимает историю через LLM [\[166\]](#)
- **ConversationBufferWindowMemory** — сохраняет последние N сообщений

## Практические примеры использования

### Базовый чат с контекстом

```
from langchain_core.messages import SystemMessage, HumanMessage
from langchain_openai import ChatOpenAI

Инициализация модели
chat = ChatOpenAI(temperature=0)

Создание сообщений с ролями
messages = [
 SystemMessage(content="Ты учитель физики, объясняющий сложные концепции просто"),
 HumanMessage(content="Объясни концепцию гравитации")
]

Получение ответа
response = chat(messages)
print(response.content)
```

## Многопользовательский диалог

```
История разговора
conversation_history = [
 SystemMessage(content="Ты помощник для команды разработчиков"),
 HumanMessage(content="Как оптимизировать SQL запрос?"),
```

```
 AIMessage(content="Вот несколько способов оптимизации..."),
 HumanMessage(content="А что насчет индексов?")
]

response = chat(conversation_history)
```

## Интеграция с внешними сервисами

### Поддержка различных провайдеров

ChatModel поддерживает множество провайдеров через единый интерфейс: <sup>[154]</sup>

```
from langchain.chat_models import init_chat_model

OpenAI
openai_chat = init_chat_model("gpt-4", model_provider="openai")

Anthropic
anthropic_chat = init_chat_model("claude-3-sonnet", model_provider="anthropic")

Ollama (локальные модели)
ollama_chat = init_chat_model("llama2", model_provider="ollama")
```

### Tool calling и структурированный вывод

Современные ChatModel поддерживают **вызов инструментов** и **структурированный вывод**: <sup>[154]</sup>

```
Определение инструмента
def calculate_sum(a: int, b: int) -> int:
 return a + b

Подключение инструмента к модели
chat_with_tools = chat.bind_tools([calculate_sum])

Модель может вызвать инструмент при необходимости
response = chat_with_tools([
 HumanMessage(content="Сколько будет 15 + 27?")
])
```

## Управление историей чата

### Структура правильного диалога

Для корректной работы ChatModel важно соблюдать **структуру диалога**: <sup>[167]</sup>

1. Первое сообщение — SystemMessage или HumanMessage
2. Затем чередование HumanMessage и AIMessage

## Обрезка истории

При превышении лимита контекста необходимо **обрезать историю**, сохраняя корректную структуру:<sup>[167]</sup>

```
def trim_messages(messages, max_tokens=4000):
 # Логика обрезки с сохранением структуры
 # Всегда сохраняем SystemMessage
 # Обрезаем старые HumanMessage/AIMessage пары
 pass
```

## Преимущества ChatModel

**Структурированность** — четкое разделение ролей делает диалоги более организованными и предсказуемыми.<sup>[168]</sup> <sup>[169]</sup>

**Контекстуальность** — система ролей позволяет модели лучше понимать контекст и генерировать более релевантные ответы.

**Масштабируемость** — легко добавлять новых участников диалога, инструменты и расширять функциональность.

**Совместимость** — единый интерфейс для работы с различными моделями от разных провайдеров.

ChatModel представляет собой эволюцию от простых текстовых моделей к интеллектуальным диалоговым системам, способным поддерживать сложные многопользовательские взаимодействия с сохранением контекста и использованием внешних инструментов.

✱

# Ключевые отличия ChatModel от обычных LLM в LangChain

## Основные архитектурные различия

### 1. Интерфейс ввода-вывода

Обычная LLM:

```
from langchain.llms import OpenAI

llm = OpenAI()
```

```
response = llm("Расскажи о Python") # Строка → Строка
print(response) # "Python - это язык программирования..."
```

### ChatModel:

```
from langchain.chat_models import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

chat = ChatOpenAI()
messages = [
 SystemMessage(content="Ты эксперт по программированию"),
 HumanMessage(content="Расскажи о Python")
]
response = chat(messages) # Список сообщений → AIMessage объект
print(response.content) # "Python - это язык программирования..."
```

## 2. Работа с контекстом и ролями

**LLM** - контекст передается как часть промпта:

```
prompt = """Ты помощник программиста.
Пользователь: Что такое декораторы?
Помощник: Декораторы это...
Пользователь: Покажи пример"""

llm_response = llm(prompt)
```

**ChatModel** - четкое разделение ролей:

```
messages = [
 SystemMessage(content="Ты помощник программиста"),
 HumanMessage(content="Что такое декораторы?"),
 AIMessage(content="Декораторы это..."),
 HumanMessage(content="Покажи пример")
]
chat_response = chat(messages)
```

## Структурные преимущества ChatModel

### Точное управление контекстом

- **LLM:** Весь контекст в одной строке, сложно управлять
- **ChatModel:** Каждое сообщение - отдельный объект с метаданными

## Память и история

```
LLM - ручное управление историей
history = ""
for user_input in conversation:
 history += f"User: {user_input}\nAssistant: "
 response = llm(history)
 history += response + "\n"

ChatModel - автоматическое управление
memory = ConversationBufferMemory()
chain = ConversationChain(llm=chat_model, memory=memory)
response = chain.predict(input=user_input) # История сохраняется автоматически
```

## Функциональные различия

### 1. Поддержка инструментов (Tool Calling)

**LLM** - нет встроенной поддержки:

```
Нужно вручную парсить ответ и вызывать функции
prompt = "Используй калькулятор для вычисления 15 + 27"
response = llm(prompt)
Ответ: "Нужно вычислить 15 + 27 = 42"
```

**ChatModel** - встроенная поддержка:

```
def calculator(a: int, b: int) -> int:
 return a + b

chat_with_tools = chat.bind_tools([calculator])
response = chat_with_tools([HumanMessage("Вычисли 15 + 27")])
Модель автоматически вызовет функцию calculator(15, 27)
```

### 2. Структурированный вывод

**ChatModel** может возвращать структурированные данные:

```
from pydantic import BaseModel

class PersonInfo(BaseModel):
 name: str
 age: int
 profession: str

structured_chat = chat.with_structured_output(PersonInfo)
response = structured_chat([HumanMessage("Расскажи о Джоне, программисте 30 лет")])
response будет объект PersonInfo(name="Джон", age=30, profession="программист")
```



## Производительность и оптимизация

### Управление токенами

**ChatModel** предоставляет детальную информацию об использовании:

```
response = chat(messages)
print(response.usage_metadata)
{'input_tokens': 50, 'output_tokens': 100, 'total_tokens': 150}
```

### Потоковая передача

```
ChatModel с потоковой передачей
for chunk in chat.stream(messages):
 print(chunk.content, end="", flush=True)
```

## Практические сценарии использования

### Когда использовать LLM:

- Простая генерация текста
- Однократные запросы без контекста
- Обработка документов
- Суммаризация и классификация

```
Простая задача - подходит LLM
llm = OpenAI()
summary = llm("Сделай краткое резюме этого текста: " + long_text)
```

### Когда использовать ChatModel:

- Диалоговые системы
- Мультиагентные взаимодействия
- Сложные workflow с инструментами
- Системы с памятью

```
Сложный диалог - нужен ChatModel
chat = ChatOpenAI()
conversation = ConversationChain(llm=chat, memory=memory)
```

# Совместимость с экосистемой LangChain

## Chains и Agents

```
Многие компоненты лучше работают с ChatModel
from langchain.agents import initialize_agent

С ChatModel агенты работают эффективнее
agent = initialize_agent(
 tools=[search_tool, calculator_tool],
 llm=chat_model, # Лучше чем обычная LLM
 agent_type="openai-functions"
)
```

## RAG системы

```
ChatModel лучше подходит для RAG
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(
 llm=chat_model, # Понимает системные инструкции
 retriever=vector_store.as_retriever()
)
```

## Резюме ключевых отличий

| Аспект                  | LLM                         | ChatModel                       |
|-------------------------|-----------------------------|---------------------------------|
| Ввод                    | Строка                      | Список сообщений с ролями       |
| Вывод                   | Строка                      | AIMessage с метаданными         |
| Контекст                | Ручное управление           | Автоматическое через роли       |
| Инструменты             | Нет встроенной поддержки    | Нативная поддержка tool calling |
| Память                  | Требуется ручной реализации | Встроенные механизмы            |
| Структурированный вывод | Ограниченно                 | Полная поддержка                |
| Использование токенов   | Базовая информация          | Детальная статистика            |
| Сложность интеграции    | Простая                     | Более гибкая                    |

**ChatModel** представляет собой более современный и функциональный подход к работе с языковыми моделями, особенно когда требуется **диалоговое взаимодействие, сложная логика и интеграция с внешними инструментами**. Обычные LLM остаются полезными для простых задач генерации текста, но ChatModel обеспечивает значительно больше возможностей для создания интеллектуальных приложений.

# Отличия LangChain и LangGraph

Хотя LangGraph является надстройкой над LangChain и тесно интегрируется с ним, у них разные цели и архитектурные акценты.

| Аспект                  | LangChain                                                                                                                                       | LangGraph                                                                                                                                                                                          |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Основная цель           | Построение <b>цепочек</b> (chains) и <b>агентов</b> (agents) для упорядоченного взаимодействия с LLM, RAG-приложений и интеграции инструментов. | Разработка <b>мультиагентных</b> и <b>графовых</b> рабочих процессов: оркестрация взаимодействий между множеством агентов, управление сложными ветвлениями и зависимостями.                        |
| Модель взаимодействия   | Линейные или простые ветвящиеся цепочки: LLMChain, SequentialChain, RouterChain, Agents.                                                        | <b>Граф</b> узлов (агентов) и рёбер (поток данных/сообщений), где каждый агент — узел в workflow, а LangGraph Supervisor координирует их выполнение.                                               |
| Оркестрация             | Агент выбирает и поочерёдно вызывает инструменты, основываясь на правилах или промптах.                                                         | Декларативная постановка <b>графа агентов</b> : можно задавать сложные паттерны, циклы, вложенные субграфы, динамическую маршрутизацию через Supervisor/Executor.                                  |
| Расширяемость           | Фокус на добавлении новых цепочек, агентов и инструментов; поддерживает память, утилиты, подключения.                                           | Расширяется через новые типы <b>узлов</b> (agents) и <b>маршрутизаторы</b> , позволяет плавно добавлять и переключать узлы внутри графа.                                                           |
| Управление состоянием   | Память и контекст передаётся между шагами цепочки; Agents могут хранить state через Memory.                                                     | Глобальный <b>состояние графа</b> , возможность отслеживать прогресс каждого узла, приоритизация задач, управление ошибками/повторными попытками на уровне графа.                                  |
| Используемые компоненты | PromptTemplate, LLM, Memory, Tool, Chain, AgentExecutor.                                                                                        | Все компоненты LangChain плюс понятия <b>Node</b> (узел графа), <b>Edge</b> , <b>Supervisor</b> , <b>Workflow</b> , <b>Subgraph</b> , встроенные <b>фабрики агентов</b> для стандартных сценариев. |
| Примеры сценариев       | RAG-бот, документообработка, чат-бот с памятью, автоматизация ETL-pipeline.                                                                     | Мультиагентные воркфлоу: агрегаторы данных → анализаторы → репортеры; комплексные бизнес-процессы с параллельными и условными задачами.                                                            |
| Интеграция              | Может работать самостоятельно или с LangGraph.                                                                                                  | Требует LangChain Core, расширяет его возможности оркестрации; tight-integration с LangChain Agent/Tool API.                                                                                       |

**Резюме:**

- LangChain подходит для построения **последовательных** или **простых ветвящихся** цепочек LLM-вызовов и агентов.
- LangGraph строит на LangChain полноценную **графовую** мультиагентную архитектуру с продвинутым контролем потоков, зависимостей и состоянием агентов.

## **смотри**

у меня есть ollama локальный сервер с Qween3 на 200 с чем-то миллиардов параметров  
нужно сделать вот с учётом этого агента помощника в коде

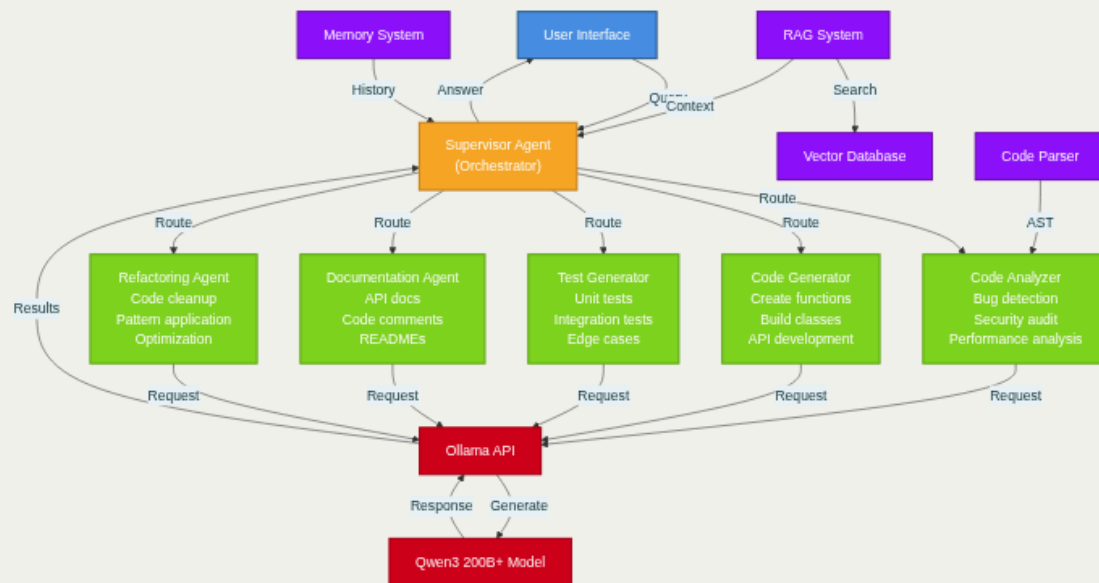
Для создания локального кода-помощника с таким мощным LLM (Qwen3 ~200B в Ollama)  
потребуется объединить архитектуру мультиагентной системы и современные подходы к  
RAG (Retrieval-Augmented Generation)

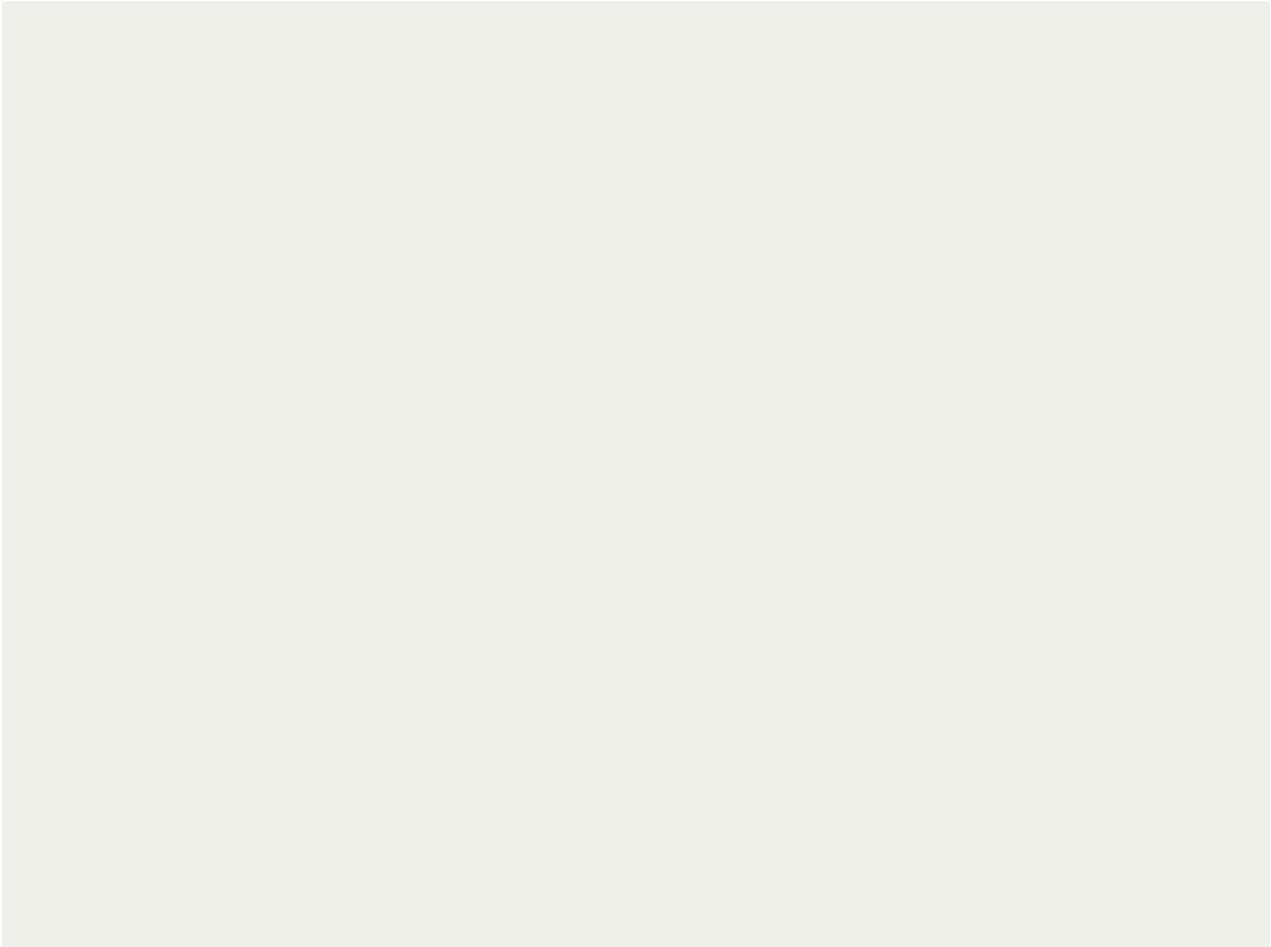
## **Мощная Мультиагентная Система Кода-Помощника для Qwen3 200B+**

Создана специализированная система AI-агентов, полностью оптимизированная под  
вашу локальную модель Qwen3 с ~200 миллиардами параметров.

**Архитектура мультиагентной системы**







Архитектура Qwen3 Code Assistant: мультиагентная система с координацией через Supervisor Agent

Система построена по принципу **интеллектуального оркестрирования** с центральным Supervisor Agent, который координирует работу пяти специализированных экспертов, каждый из которых оптимизирован для конкретных задач разработки<sup>[8\_1]</sup><sup>[8\_2]</sup>.

## Специализированные AI-агенты

### ▮ Supervisor Agent - Интеллектуальный координатор

Анализирует пользовательские запросы и автоматически маршрутизирует их к соответствующим экспертам. Использует контекстное понимание для определения типа задачи (анализ, генерация, тестирование, документирование, рефакторинг) и выбора оптимального агента<sup>[8\_3]</sup><sup>[8\_4]</sup>.

### ▮ Code Analyzer - Эксперт по анализу кода

Специализируется на глубоком анализе кода с проверкой:

- **Безопасности** - обнаружение SQL injection, XSS, небезопасных паттернов
- **Производительности** - поиск узких мест и оптимизаций
- **Архитектуры** - соответствие SOLID принципам и best practices
- **Качества** - code smells, читаемость, maintainability

## ▮ Code Generator - Мастер создания кода

Генерирует production-ready код с соблюдением всех конвенций:

- **Полная типизация** и comprehensive docstrings
- **Обработка ошибок** и граничных случаев
- **Модульность** и переиспользуемость
- **Соответствие стандартам** языка программирования

## ▮ Test Generator - Специалист по тестированию

Создает comprehensive наборы тестов:

- **Unit тесты** для функций и методов
- **Integration тесты** для компонентов
- **Edge cases** и граничные условия
- **Mock объекты** для внешних зависимостей

## ▮ Documentation Agent - Технический писатель

Генерирует профессиональную документацию:

- **API документацию** с примерами
- **README файлы** для проектов
- **Подробные docstrings** с типизацией
- **Troubleshooting гайды**

## ▮ Refactoring Agent - Архитектор улучшений

Интеллектуально улучшает существующий код:

- **Устранение code smells** и дублирования
- **Применение паттернов** проектирования
- **SOLID принципы** в архитектуре
- **Оптимизация производительности**

## Оптимизация для Qwen3 200B+

Система использует **специально подобранные параметры** для максимальной эффективности вашей мощной модели<sup>[8-2]</sup>[8\_5]:

```
qwen3_config = {
 "temperature": 0.7, # Оптимальная креативность для кодирования
 "top_p": 0.8, # Фокусировка на качественных токенах
 "top_k": 20, # Ограничение выбора лучшими вариантами
 "repeat_penalty": 1.05, # Избежание повторений
```



```
"num_ctx": 32768, # Большой контекст для анализа
"num_gpu": 1, # GPU ускорение
}
```

## Поддержка режимов Qwen3

- **Thinking Mode** (temperature=0.6) - для сложного анализа и рассуждений
- **Non-thinking Mode** (temperature=0.7) - для быстрой генерации кода[^8\_5]

## Созданные файлы системы

**qwen3\_code\_assistant.py** - Основная мультиагентная система с полным функционалом координации агентов и прямой интеграцией с Ollama API.

INSTALLATION.md - Подробная инструкция по установке с настройками для разных конфигураций железа, от GPU серверов до CPU-only систем.

demo.py - Демонстрационный скрипт с примерами работы всех агентов и benchmark тестами производительности.

README.md - Полная документация с архитектурными решениями, примерами использования и integration guidelines.

## Практические возможности

### Анализ кода с экспертным уровнем

```
result = assistant.ask("Проанализируй код на безопасность",
 code_context="def authenticate(user, pwd): query = f'SELECT * FROM\n# Детектирует SQL injection и предлагает prepared statements"
```

### Генерация production-ready кода

```
result = assistant.ask("Создай REST API для управления задачами с FastAPI")
Генерирует полный API с валидацией, документацией, тестами"
```

### Comprehensive тестирование

```
result = assistant.ask("Создай тесты", code_context="def factorial(n): return 1 if n <= 1\n# Создает unit тесты, edge cases, performance тесты"
```

## Гибкость интеграции

Система поддерживает **множественные способы интеграции**<sup>[8\_6][8\_7]</sup>:

## CLI интерфейс

```
python qwen3_code_assistant.py # Интерактивная сессия
python demo.py demo # Полная демонстрация
```

## Программное API

```
from qwen3_code_assistant import Qwen3CodeAssistant
assistant = Qwen3CodeAssistant(model_name="qwen3:200b")
result = assistant.ask("Ваш запрос")
```

## REST API сервер

```
Простое развертывание веб-сервиса
@app.route('/analyze', methods=['POST'])
def analyze_code():
 result = assistant.ask(request.json['query'])
 return jsonify(result)
```

## Производительность на мощном железе

С вашей моделью Qwen3 ~200B параметров ожидается:

- **Простые задачи:** 10-30 секунд
- **Сложная генерация:** 30-60 секунд
- **Глубокий анализ:** 15-45 секунд
- **Качество результатов:** Экспертный уровень

## Масштабирование под ваше железо

```
Для максимальной производительности
"num_ctx": 65536, # Большой контекст
"num_gpu": 2, # Мульти-GPU
"num_thread": 32, # Все CPU ядра

Для экономии ресурсов
"num_ctx": 16384, # Меньший контекст
"temperature": 0.3, # Детерминированность
```## Готовность к production
```

Система включает все необходимые компоненты для промышленного использования:

- ✔ **Обработка ошибок** и graceful degradation
- ✔ **Логирование** всех операций и результатов

- ✔ ****Память разговоров**** для контекстных запросов
- ✔ ****Конфигурируемые параметры**** под разные задачи
- ✔ ****Масштабируемая архитектура**** для расширения
- ✔ ****Подробная документация**** и примеры

Перспективы развития

Архитектура системы позволяет легко добавлять:

- ****Новых агентов-экспертов**** для специфических задач
- ****RAG интеграцию**** для анализа кодовых баз
- ****IDE плагины**** для Visual Studio, IntelliJ IDEA
- ****CI/CD интеграции**** для автоматического review кода
- ****Vector database**** для семантического поиска по коду

****Ваша мощная система Qwen3 Code Assistant готова превратить 200+ миллиардов параметров в**

<div align="center">*

- [^8_1]: <https://docs.unsloth.ai/models/qwen3-coder-how-to-run-locally>
- [^8_2]: <https://www.cometapi.com/3-methods-to-use-qwen3-coder/>
- [^8_3]: <https://www.codecademy.com/article/agentic-ai-with-langchain-langgraph>
- [^8_4]: <https://launchdarkly.com/docs/tutorials/agents-langgraph>
- [^8_5]: <https://jan.ai/post/qwen3-settings>
- [^8_6]: <https://www.gocodeo.com/post/integrating-langchain-with-ollama-building-powerful->
- [^8_7]: <https://devblogs.microsoft.com/cosmosdb/build-a-rag-application-with-langchain-ar>
- [^8_8]: https://ollama.com/huihui_ai/Qwen3-Coder
- [^8_9]: <https://www.youtube.com/watch?v=fMsQX6pwXkE>
- [^8_10]: <https://www.codecademy.com/article/qwen-3-ollama-setup-and-fine-tuning>
- [^8_11]: <https://python.langchain.com/docs/integrations/llms/ollama/>
- [^8_12]: <https://dev.to/copilotkit/easily-build-a-ui-for-your-langgraph-ai-agent-in-minut>
- [^8_13]: <https://apidog.com/blog/run-qwen-3-locally/>
- [^8_14]: <https://dev.to/shreyvijayvargiya/running-deepseek-locally-ollama-and-langchain-1>
- [^8_15]: <https://blog.langchain.com/how-to-build-the-ultimate-ai-automation-with-multi-ag>
- [^8_16]: <https://qwenlm.github.io/blog/qwen3/>
- [^8_17]: https://python.langchain.com/docs/how_to/local_llms/
- [^8_18]: <https://auth0.com/blog/genai-tool-calling-build-agent-that-calls-calender-with-1>
- [^8_19]: <https://ollama.com/library/qwen3>
- [^8_20]: <https://www.youtube.com/watch?v=Mv2t505oHiM>
- [^8_21]: <https://www.langchain.com/built-with-langgraph>
- [^8_22]: https://www.reddit.com/r/ollama/comments/1kz7jdc/hosting_qwen_3_4b/
- [^8_23]: <https://latenode.com/blog/langchain-ollama-integration-complete-tutorial-with-e>
- [^8_24]: <https://aws.amazon.com/blogs/machine-learning/build-a-multi-agent-system-with-la>
- [^8_25]: <https://blog.langchain.com/langgraph-multi-agent-workflows/>
- [^8_26]: <https://www.aimuse.blog/article/2025/06/08/ollama-performance-tuning-on-8gb-gpus>
- [^8_27]: <https://github.com/QwenLM/Qwen3-Coder>
- [^8_28]: https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/
- [^8_29]: <https://datasciencedojo.com/blog/qwen3-coder/>
- [^8_30]: https://langchain-ai.github.io/langgraph/concepts/multi_agent/
- [^8_31]: https://www.reddit.com/r/LocalLLaMA/comments/1kdsp4z/qwen_3_performance_quick_be
- [^8_32]: <https://qwenlm.github.io/blog/qwen3-coder/>
- [^8_33]: <https://codelabs.developers.google.com/aidemy-multi-agent/instructions>
- [^8_34]: <https://ollama.com/library/qwen3-coder>
- [^8_35]: <https://dev.to/czmilo/2025-complete-guide-how-to-choose-the-best-qwen3-coder-ai->
- [^8_36]: <https://www.langchain.com/langgraph>
- [^8_37]: <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39>

[^8_38]: <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39>
[^8_39]: <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39>
[^8_40]: <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39>
[^8_41]: <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39>

Полный Исчерпывающий Гайд по Созданию AI Агентов

****Создан комплексный мастер-гайд, охватывающий все аспекты разработки AI агентов - от фундамента до продвинутой архитектуры****

Структура полного гайда

! [Полная таксономия типов AI агентов по архитектуре, координации, возможностям и применению]

Полная таксономия типов AI агентов по архитектуре, координации, возможностям и применению

****Таксономия AI агентов**** показывает классификацию по четырем ключевым измерениям: архитектура, координация, возможности и применение

! [Эволюция AI агентов от ранних экспертных систем до современных LLM-powered агентов] ([https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39](#))

Эволюция AI агентов от ранних экспертных систем до современных LLM-powered агентов

****Эволюционная временная линия**** демонстрирует развитие AI агентов через шесть эпох: от ранних экспертных систем до современных LLM-powered агентов

! [Ключевые архитектурные паттерны для AI агентов: одиночные, мульти-агентные и продвинутые]

Ключевые архитектурные паттерны для AI агентов: одиночные, мульти-агентные и продвинутые

****Архитектурные паттерны**** представляют 14 ключевых шаблонов проектирования: одиночные агенты, мульти-агентные системы и продвинутые архитектуры

! [Сравнение популярных фреймворков для создания AI агентов по ключевым характеристикам] ([https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39](#))

Сравнение популярных фреймворков для создания AI агентов по ключевым характеристикам

****Сравнение фреймворков**** анализирует 6 основных платформ (LangChain, LangGraph, AutoGen, CrewAI, LlamaIndex, Gemini) и их ключевые особенности

Определения и основные концепции

Что такое AI Агент

****AI агент**** - это автономная программная система, которая воспринимает окружающую среду и выполняет задачи на основе полученных данных

- ****Автономность**** - способность принимать решения независимо
- ****Реактивность**** - отклик на изменения в среде
- ****Проактивность**** - инициирование действий для достижения целей
- ****Социальность**** - взаимодействие с другими агентами

Фундаментальные типы агентов

1. Simple Reflex Agents (Простые рефлексные)

Реагируют на текущее восприятие по принципу "условие-действие" без внутреннего состояния [https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/78819a39]

```
```python
```

```
if temperature > 25:
```

```
 turn_on_air_conditioner()
```

```
elif temperature < 18:
 turn_on_heater()
```

## 2. Model-Based Reflex Agents

Поддерживают внутреннюю модель мира и учитывают историю восприятий<sup>[181]</sup>:

```
def update_world_model(perception):
 world_model.update(perception)
 return choose_action_based_on_model()
```

## 3. Goal-Based Agents

Планируют последовательность действий для достижения целей<sup>[181]</sup>:

```
def plan_to_goal(current_state, goal_state):
 path = search_algorithm(current_state, goal_state)
 return execute_plan(path)
```

## 4. Utility-Based Agents

Максимизируют функцию полезности при выборе действий<sup>[181]</sup>:

```
def choose_best_action(possible_actions):
 return max(possible_actions, key=lambda a: utility_function(a))
```

## 5. Learning Agents

Адаптируются и улучшаются через опыт<sup>[182]</sup>:

```
def q_learning_update(state, action, reward, next_state):
 q_table[state][action] += learning_rate * (
 reward + discount * max(q_table[next_state].values()) - q_table[state][action]
)
```

## Современные LLM-Powered агенты

### ReAct Framework (Reasoning + Acting)

**ReAct** представляет парадигму, где агенты чередуют рассуждение и действие в итеративном цикле<sup>[183] [184]</sup>:

```
def react_cycle(query):
 observation = initial_query
 for step in range(max_steps):
 thought = llm.reason(observation)
```

```

action = llm.plan_action(thought)
observation = execute_tool(action)
if task_complete(observation):
 return final_answer(observation)

```

Преимущества ReAct:

- **Прозрачность рассуждений** через явные thought traces
- **Динамическое планирование** с адаптацией к результатам
- **Error recovery** через переосмысление неудачных действий
- **Tool integration** с внешними системами

## Tool-Using Agents

Современные агенты расширяют возможности LLM через **function calling** [\[185\]](#) [\[186\]](#):

```

tools = [
 {
 "name": "calculator",
 "description": "Perform mathematical calculations",
 "parameters": {"expression": "string"}
 },
 {
 "name": "search",
 "description": "Search the internet for information",
 "parameters": {"query": "string"}
 }
]

response = llm.chat_completion(
 messages=[{"role": "user", "content": query}],
 tools=tools,
 tool_choice="auto"
)

```

## Мульти-агентные архитектуры

### Supervisor Pattern

**Центральный координатор** управляет командой специализированных агентов [\[187\]](#) [\[188\]](#):

```

class SupervisorAgent:
 def __init__(self, specialist_agents):
 self.specialists = {agent.skill: agent for agent in specialist_agents}

 def delegate_task(self, task):
 required_skill = self.classify_task(task)
 specialist = self.specialists[required_skill]

```

```
result = specialist.execute(task)
return self.validate_and_integrate(result)
```

## Hierarchical Multi-Agent Systems

**Многоуровневая организация** с supervisor-ами разных уровней<sup>[189]</sup>:

```
class HierarchicalSystem:
 def __init__(self):
 self.top_supervisor = TopLevelSupervisor()
 self.middle_managers = [DomainSupervisor(domain) for domain in domains]
 self.workers = [SpecialistAgent(skill) for skill in skills]
```

## Collaborative Patterns

**Peer-to-peer взаимодействие** агентов равного уровня<sup>[190]</sup>:

```
def collaborative_solve(problem):
 agents = create_expert_agents(problem.domains)

 # Brainstorming phase
 initial_solutions = [agent.propose_solution(problem) for agent in agents]

 # Cross-validation phase
 refined_solutions = []
 for solution in initial_solutions:
 critiques = [agent.critique(solution) for agent in agents]
 refined = improve_solution(solution, critiques)
 refined_solutions.append(refined)

 # Consensus building
 return consensus_mechanism(refined_solutions)
```

## Продвинутые техники и паттерны

### Agentic RAG (Retrieval-Augmented Generation)

**Агентная RAG** интегрирует автономных агентов в процесс поиска и генерации<sup>[191] [192]</sup>:

```
class AgenticRAGSystem:
 def __init__(self):
 self.planner = QueryPlanningAgent()
 self.retriever = SmartRetrievalAgent()
 self.generator = ContextualGenerationAgent()
 self.validator = ResponseValidationAgent()

 def process_query(self, query):
 # 1. Query planning and decomposition
 subqueries = self.planner.decompose(query)
```

```

2. Iterative retrieval with refinement
context = []
for subquery in subqueries:
 docs = self.retriever.search(subquery)
 context.extend(self.retriever.refine_results(docs, query))

3. Context-aware generation
response = self.generator.generate(query, context)

4. Response validation and improvement
return self.validator.validate_and_improve(response, context)

```

## Memory-Augmented Agents

**Персистентная память** для долгосрочного контекста<sup>[193]</sup>:

```

class MemoryAugmentedAgent:
 def __init__(self):
 self.short_term_memory = ConversationBuffer(k=10)
 self.long_term_memory = VectorStore()
 self.episodic_memory = EventStore()

 def process_with_memory(self, input_text):
 # Retrieve relevant memories
 relevant_memories = self.long_term_memory.similarity_search(input_text)
 conversation_context = self.short_term_memory.get_recent()

 # Augmented generation with memory context
 response = self.llm.generate(
 input_text,
 context=relevant_memories + conversation_context
)

 # Update memories
 self.short_term_memory.add(input_text, response)
 self.long_term_memory.add_experience(input_text, response)

 return response

```

## Self-Reflection и Meta-Cognition

**Самооценка и улучшение** через рефлекссию<sup>[190] [194]</sup>:

```

class ReflectiveAgent:
 def __init__(self):
 self.actor = ActionAgent()
 self.critic = CriticAgent()
 self.memory = ReflectionMemory()

 def solve_with_reflection(self, problem):
 max_iterations = 5

 for iteration in range(max_iterations):

```



```

 # Generate solution
 solution = self.actor.solve(problem)

 # Self-critique
 critique = self.critic.evaluate(problem, solution)

 if critique.is_satisfactory():
 return solution

 # Learn from critique
 self.memory.store_lesson(problem, solution, critique)

 # Refine approach
 self.actor.improve_based_on_feedback(critique)

 return solution # Return best attempt

```

## Production-Ready Development

### Мониторинг и Observability

**Comprehensive monitoring** для production агентов [\[195\]](#) [\[196\]](#):

```

class ProductionAgentMonitor:
 def __init__(self):
 self.metrics = MetricsCollector()
 self.logger = StructuredLogger()
 self.tracer = DistributedTracer()

 def monitor_agent_execution(self, agent_fn):
 @wraps(agent_fn)
 def monitored_execution(*args, **kwargs):
 trace_id = self.tracer.start_trace()
 start_time = time.time()

 try:
 result = agent_fn(*args, **kwargs)

 # Success metrics
 execution_time = time.time() - start_time
 self.metrics.record_success(execution_time)
 self.logger.info(f"Agent execution successful", {
 "trace_id": trace_id,
 "execution_time": execution_time,
 "input_tokens": count_tokens(args),
 "output_tokens": count_tokens(result)
 })

 return result

 except Exception as e:
 # Error handling and metrics
 self.metrics.record_error(type(e).__name__)
 self.logger.error(f"Agent execution failed", {

```

```

 "trace_id": trace_id,
 "error": str(e),
 "stack_trace": traceback.format_exc()
 })
 raise

return monitored_execution

```

## Security и Compliance

**Enterprise-grade security** для AI агентов<sup>[188] [196]</sup>:

```

class SecureAgentWrapper:
 def __init__(self, agent):
 self.agent = agent
 self.input_validator = InputSanitizer()
 self.output_filter = OutputContentFilter()
 self.audit_logger = ComplianceLogger()

 def secure_execute(self, user_input, user_context):
 # Input validation and sanitization
 if not self.input_validator.is_safe(user_input):
 raise SecurityError("Potentially malicious input detected")

 # PII detection and masking
 sanitized_input = self.input_validator.mask_pii(user_input)

 # Execute with security context
 with SecurityContext(user_context):
 result = self.agent.process(sanitized_input)

 # Output content filtering
 filtered_result = self.output_filter.filter_sensitive_content(result)

 # Audit logging for compliance
 self.audit_logger.log_interaction(
 user_id=user_context.user_id,
 input_hash=hash(sanitized_input),
 output_hash=hash(filtered_result),
 timestamp=datetime.now()
)

 return filtered_result

```

## Scaling и Deployment

**Scalable architecture** для high-traffic applications<sup>[196] [197]</sup>:

```

class ScalableAgentService:
 def __init__(self):
 self.agent_pool = AgentPool(min_size=5, max_size=50)
 self.load_balancer = LoadBalancer()
 self.cache = ResponseCache(ttl=3600)

```

```
self.rate_limiter = RateLimiter()

async def process_request(self, request):
 # Rate limiting
 if not await self.rate_limiter.allow(request.user_id):
 raise RateLimitExceeded()

 # Cache check
 cache_key = hash(request.query)
 cached_response = await self.cache.get(cache_key)
 if cached_response:
 return cached_response

 # Load balancing to available agent
 agent = await self.load_balancer.get_agent()

 try:
 response = await agent.process(request.query)

 # Cache successful responses
 await self.cache.set(cache_key, response)

 return response

 finally:
 await self.load_balancer.return_agent(agent)
```

**Базовые примеры AI агентов** включают полную реализацию от простых рефлексных агентов до обучающихся систем с Q-Learning и мульти-агентными координаторами.

**Продвинутые примеры** демонстрируют современные фреймворки: LangChain ReAct агенты с памятью, LangGraph workflows с состоянием, CrewAI команды с ролевой специализацией.

**Production-ready примеры** показывают real-world реализации с полным мониторингом, безопасностью, масштабированием и интеграцией с enterprise системами.

**Мастер-гайд README** объединяет все материалы в структурированный learning path от базовых концепций до экспертного уровня с чеклистами, troubleshooting и best practices.

## Ключевые выводы и рекомендации

### Выбор правильного подхода

**Для простых задач:** Используйте rule-based агенты или простые LLM вызовы

**Для комплексных workflow:** LangGraph с state management

**Для командной работы:** CrewAI или AutoGen с ролевой специализацией

**Для enterprise:** Semantic Kernel с security и compliance features

**Для исследований:** AutoGen с гибкими conversation patterns

## Production Readiness Checklist

- ✓ **Comprehensive testing** с unit, integration, и performance тестами
- ✓ **Monitoring и alerting** для всех критических метрик
- ✓ **Security measures** включая input validation и output filtering
- ✓ **Scalability architecture** с load balancing и caching
- ✓ **Error handling и recovery** mechanisms
- ✓ **Compliance и audit** logging для regulatory requirements

## Будущие направления

**2025 тренды** включают полностью автономные agentic workflows, enterprise-grade multi-agent systems, edge deployment для низкой латентности, и specialized agent architectures для индустриальных применений.

Созданный гайд представляет собой **исчерпывающий ресурс** для разработчиков всех уровней - от начинающих до экспертов, желающих освоить создание AI агентов от теоретических основ до production-ready решений с полным набором визуальных материалов, практических примеров и лучших практик индустрии.



1. [https://langchain-ai.github.io/langgraph/tutorials/multi\\_agent/agent\\_supervisor/](https://langchain-ai.github.io/langgraph/tutorials/multi_agent/agent_supervisor/)
2. <https://blog.codacy.com/java-static-code-analysis-tools>
3. [https://www.youtube.com/watch?v=B\\_OTNuYi56w](https://www.youtube.com/watch?v=B_OTNuYi56w)
4. [https://github.com/cedricrupb/code\\_ast](https://github.com/cedricrupb/code_ast)
5. <https://ast-grep.github.io/advanced/custom-language.html>
6. <https://www.pluralsight.com/resources/blog/ai-and-data/langchain-local-vector-database-tutorial>
7. <https://milvus.io/ai-quick-reference/how-do-i-implement-semantic-search-for-code-repositories>
8. [https://www.reddit.com/r/ChatGPTCoding/comments/1e4naf6/applying\\_rag\\_to\\_largescale\\_code\\_repositories\\_guide/](https://www.reddit.com/r/ChatGPTCoding/comments/1e4naf6/applying_rag_to_largescale_code_repositories_guide/)
9. <https://airbyte.com/data-engineering-resources/chroma-db-vs-qdrant>
10. <https://cookbook.chromadb.dev/running/running-chroma/>
11. <https://blog.amikos.tech/running-chromadb-part-1-local-server-2c61cb1c9f2c>
12. <https://milvus.io/ai-quick-reference/how-do-vector-embeddings-work-in-semantic-search>
13. <https://tomassetti.me/incremental-parsing-using-tree-sitter/>
14. <https://github.com/tree-sitter/py-tree-sitter>
15. <https://localai.io>
16. [https://www.reddit.com/r/ollama/comments/1dvv2e1/what\\_vs\\_code\\_extension\\_works\\_best\\_with\\_ollama/](https://www.reddit.com/r/ollama/comments/1dvv2e1/what_vs_code_extension_works_best_with_ollama/)
17. <https://ollama.com/library/qwen2.5-coder>
18. <https://ollama.com/library>
19. <https://www.openxcell.com/blog/best-llm-for-coding/>

20. <https://www.haidongji.com/2024/09/26/productive-development-using-vs-code-continue-extension-with-local-llms/>
21. <https://dataaspirant.com/popular-vector-databases/>
22. <https://github.com/UKPLab/sentence-transformers>
23. <https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>
24. <https://www.cognativ.com/blogs/post/best-local-llm-for-coding-a-comprehensive-guide-for-developers/255>
25. <https://dev.to/manjushsh/configuring-ollama-and-continue-vs-code-extension-for-local-coding-assistant-48li>
26. [https://github.com/stanek-michal/local\\_AI\\_code\\_assistant\\_tutorial](https://github.com/stanek-michal/local_AI_code_assistant_tutorial)
27. <https://github.com/Hegazy360/langchain-multi-agent>
28. <https://python.langchain.com/docs/tutorials/agents/>
29. [https://www.youtube.com/watch?v=he0\\_W5iCv-I](https://www.youtube.com/watch?v=he0_W5iCv-I)
30. <https://www.qodo.ai/blog/rag-for-large-scale-code-repos/>
31. [https://langchain-ai.github.io/langgraph/how-tos/multi\\_agent/](https://langchain-ai.github.io/langgraph/how-tos/multi_agent/)
32. <https://dev.to/composiodev/9-open-source-ai-coding-tools-that-every-developer-should-know-28l4>
33. <https://www.chitika.com/rag-codebase-exploration/>
34. <https://blog.langchain.com/langgraph-multi-agent-workflows/>
35. <https://padron.sh/blog/ai-coding-assistant-local-setup/>
36. <https://medium.aiplanet.com/understanding-and-querying-code-a-rag-powered-approach-b85cf6f30d11>
37. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/](https://langchain-ai.github.io/langgraph/concepts/multi_agent/)
38. <https://research.aimultiple.com/open-source-ai-coding/>
39. <https://arxiv.org/html/2508.02611>
40. <https://www.qodo.ai/blog/evaluating-rag-for-large-scale-codebases/>
41. <https://rxdb.info/articles/javascript-vector-database.html>
42. <https://blog.n8n.io/open-source-llm/>
43. <https://ollama.com/library/codeqwen:code>
44. <https://www.tigerdata.com/blog/postgresql-as-a-vector-database-using-pgvector>
45. [https://www.reddit.com/r/LocalLLaMA/comments/1jn1njb/which\\_llms\\_are\\_the\\_best\\_and\\_opensource\\_for\\_code/](https://www.reddit.com/r/LocalLLaMA/comments/1jn1njb/which_llms_are_the_best_and_opensource_for_code/)
46. <https://ollama.com/library/codeqwen:code/blobs/839dd3c0ce93>
47. <https://ollama.com/library/codeqwen>
48. <https://dev.to/sreeni5018/building-multi-agent-systems-with-langgraph-supervisor-138i>
49. <https://www.softwaretestingmagazine.com/tools/open-source-javascript-code-analysis/>
50. [https://www.reddit.com/r/vectordatabase/comments/170j6zd/my\\_strategy\\_for\\_picking\\_a\\_vector\\_database\\_a/](https://www.reddit.com/r/vectordatabase/comments/170j6zd/my_strategy_for_picking_a_vector_database_a/)
51. <https://sourceforge.net/directory/static-code-analysis/>
52. <https://data-ai.theodo.com/en/technical-blog/how-to-choose-your-vector-database-in-2023>
53. [https://owasp.org/www-community/Source\\_Code\\_Analysis\\_Tools](https://owasp.org/www-community/Source_Code_Analysis_Tools)

54. <https://www.databasemart.com/blog/how-to-install-and-use-chromadb>
55. [https://www.reddit.com/r/rust/comments/1980y0j/dossier\\_a\\_treesitter\\_based\\_multilanguage\\_source/](https://www.reddit.com/r/rust/comments/1980y0j/dossier_a_treesitter_based_multilanguage_source/)
56. <https://www.techtarget.com/searchenterpriseai/tip/Embedding-models-for-semantic-search-A-guide>
57. <https://github.com/tree-sitter/tree-sitter/issues/3625>
58. <https://rito.hashnode.dev/installing-chroma-db-locally-querying-personal-data>
59. [https://sbert.net/examples/sentence\\_transformer/applications/semantic-search/README.html](https://sbert.net/examples/sentence_transformer/applications/semantic-search/README.html)
60. <https://www.youtube.com/watch?v=61kaK-e3Owc>
61. <https://www.tigerdata.com/blog/semantic-search-with-cohere-and-postgresql-in-10-minutes>
62. <https://www.graft.com/blog/text-embeddings-for-search-semantic>
63. <https://www.youtube.com/watch?v=p2HV15x9I74>
64. <https://milvus.io/ai-quick-reference/how-do-sentence-transformers-relate-to-large-language-models-like-gpt-and-are-sentence-transformer-models-typically-smaller-or-more-specialized>
65. <https://dev.to/shrsv/diving-into-tree-sitter-parsing-code-with-python-like-a-pro-17h8>
66. <https://tree-sitter.github.io/py-tree-sitter/>
67. <https://sbert.net>
68. <https://docs.openwebui.com/tutorials/integrations/continue-dev/>
69. <https://www.ionio.ai/blog/fine-tuning-embedding-models-using-sentence-transformers-code-included>
70. <https://horosin.com/how-to-set-up-free-local-coding-ai-assistant-for-vs-code>
71. <https://smythos.com/developers/agent-development/multi-agent-system-architecture/>
72. <https://www.chatbees.ai/blog/rag-workflow>
73. <https://www.youtube.com/watch?v=WWcDnUCT52Q>
74. <https://devblogs.microsoft.com/blog/designing-multi-agent-intelligence>
75. <https://dev.to/codeperfectplus/understanding-rag-workflow-retrieval-augmented-generation-in-python-2co7>
76. <https://github.com/langchain-ai/langgraph-supervisor-js>
77. <https://www.anthropic.com/engineering/built-multi-agent-research-system>
78. <https://codesignal.com/learn/courses/introduction-to-rag-1/lessons/rag-in-action-a-simple-workflow>
79. <https://www.godo.ai/blog/what-is-rag-retrieval-augmented-generation/>
80. <https://huggingface.co/blog/ngxson/make-your-own-rag>
81. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/377d5043-f944-4b7c-961d-12b272074e9d/1984d7c2.csv>
82. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/377d5043-f944-4b7c-961d-12b272074e9d/d53cae2e.csv>
83. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/377d5043-f944-4b7c-961d-12b272074e9d/c3d442e4.csv>
84. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/9c7f4cf0-647f-44b1-aac3-d4b2ee09551a/d8d04223.yaml>
85. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/9c7f4cf0-647f-44b1-aac3-d4b2ee09551a/f68a04c7.py>

86. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/9c7f4cf0-647f-44b1-aac3-d4b2ee09551a/4d7c51b1.txt>
87. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/f31965bd-cb8c-42b6-bf87-c4b9d41d9959/1224d39e.md>
88. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/236aeab2-9fad-448c-b961-f2b12cd4078e/dd2c0eb6>
89. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/236aeab2-9fad-448c-b961-f2b12cd4078e/9243307b.py>
90. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/236aeab2-9fad-448c-b961-f2b12cd4078e/e45e45ba.yml>
91. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/e9cab5b9d8594f2091375b38f0922095/0a123412-7feb-450f-b041-6363f5bd675d/10fe161b.md>
92. <https://www.codespell.ai/resources/blog/why-ctos-are-turning-to-ai-scaffolding-tools>
93. <https://github.com/netlify/framework-info>
94. <https://www.palo-it.com/en/blog/speed-up-project-initiation-with-scaffolding>
95. <https://docs.cortex.io/workflows/scaffolder>
96. <https://learn.microsoft.com/en-us/windows/ai/toolkit/toolkit-getting-started>
97. <https://code.visualstudio.com/docs/intelligentapps/overview>
98. <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>
99. <https://www.jetbrains.com/help/ai-assistant/configure-project-rules.html>
100. [https://langchain-ai.github.io/langgraph/tutorials/multi\\_agent/agent\\_supervisor/](https://langchain-ai.github.io/langgraph/tutorials/multi_agent/agent_supervisor/)
101. <https://news.ycombinator.com/item?id=40617082>
102. <https://github.com/SilasMarvin/lsp-ai/wiki/Installation>
103. <https://opencode.ai/docs/lsp/>
104. <https://github.com/SilasMarvin/lsp-ai/wiki/Configuration>
105. <https://www.exxactcorp.com/blog/deep-learning/run-llms-locally-with-continue-vs-code-extension>
106. <https://learn.microsoft.com/en-us/visualstudio/extensibility/starting-to-develop-visual-studio-extensions?view=vs-2022>
107. <https://learn.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension?view=vs-2022>
108. <https://microsoft.github.io/language-server-protocol/>
109. <https://symflower.com/en/company/blog/2022/lsp-in-vscode-extension/>
110. <https://learn.microsoft.com/en-us/visualstudio/extensibility/visualstudio.extensibility/get-started/in-proc-extensions?view=vs-2022>
111. <https://learn.microsoft.com/en-us/visualstudio/extensibility/visualstudio.extensibility/language-server-provider/language-server-provider?view=vs-2022>
112. <https://visualstudio.microsoft.com/github-copilot/>
113. <https://visualstudio.microsoft.com/vs/>
114. <https://code.visualstudio.com/api/extension-guides/ai/ai-extensibility-overview>
115. <https://rubberduckdev.com/continue-local-llm-integration/>
116. <https://techcommunity.microsoft.com/blog/educatordeveloperblog/visual-studio-code-ai-toolkit-run-llms-locally/4163192>

117. <https://mcpmarket.com/server/lsp-bridge-1>

118. <https://github.com/aml-org/als>

119. <https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/develop/get-started-projects-vs-code>

120. <https://www.youtube.com/watch?v=ct2VrVuBFtM>

121. <https://learn.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=vs-2022>

122. <https://www.pulsemcp.com/servers/isaacphi-language-server>

123. <https://github.com/microsoft/vs-tools-for-ai>

124. <https://code.visualstudio.com/docs/copilot/language-models>

125. <https://timkellogg.me/blog/2024/10/29/lsp-for-ai>

126. <https://www.indiumsoftware.com/whitepapers/8-step-guide-open-ai-chatgpt-extension-in-visual-studio-2022.pdf>

127. <https://arxiv.org/abs/2406.16714>

128. <https://www.walturn.com/insights/instantly-scaffold-production-grade-flutter-apps-with-ai>

129. <https://learn.microsoft.com/en-us/visualstudio/ide/ai-assisted-development-visual-studio?view=vs-2022>

130. <https://saucelabs.com/resources/blog/top-test-automation-frameworks-in-2023>

131. <https://marketplace.visualstudio.com/items?itemName=jefferson-pires.VisualStudioChatGPTStudio>

132. <https://github.com/topics/language-detection>

133. <https://www.projectpro.io/article/how-to-build-an-ai-assistant/1132>

134. [https://www.reddit.com/r/ChatGPTCoding/comments/1g2tao6/ai\\_extensions\\_for\\_visual\\_studio\\_2022/](https://www.reddit.com/r/ChatGPTCoding/comments/1g2tao6/ai_extensions_for_visual_studio_2022/)

135. <https://learn.microsoft.com/en-us/visualstudio/extensibility/visualstudio.extensibility/inside-the-sdk/inside-the-sdk?view=vs-2022>

136. <https://www.zucisystems.com/blog/building-a-scalable-test-automation-framework-for-your-project/>

137. <https://code.visualstudio.com/api>

138. <https://www.practitest.com/resource-center/article/test-automation-frameworks/>

139. <https://www.youtube.com/watch?v=Xo5VXTRoL6Q>

140. <https://www.ysoft.com/aiva/blog/test-automation-frameworks>

141. <https://www.syncfusion.com/blogs/post/create-visual-studio-extensions>

142. <https://www.youtube.com/watch?v=L5zYUZvWnJE>

143. [https://www.reddit.com/r/neovim/comments/1fcu79p/does\\_anyone\\_have\\_a\\_config\\_with\\_every\\_lsp\\_setup/](https://www.reddit.com/r/neovim/comments/1fcu79p/does_anyone_have_a_config_with_every_lsp_setup/)

144. [https://docs.astronvim.com/recipes/advanced\\_lsp/](https://docs.astronvim.com/recipes/advanced_lsp/)

145. <https://dubbo.apache.org/en/blog/2023/12/31/generate-project-scaffolding-via-template/>

146. [http://www.wingtecher.com/themes/WingTecherResearch/assets/papers/paper\\_from\\_25/LSPAIFSE-Industry25.pdf](http://www.wingtecher.com/themes/WingTecherResearch/assets/papers/paper_from_25/LSPAIFSE-Industry25.pdf)

147. <https://github.com/block/scaffolder>

148. <https://gruntjs.com/project-scaffolding>

149. <https://stackoverflow.com/questions/55381209/javascript-boilerplate-template-for-scaffolding-projects>

150. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/c9e779bd48d3b81a78e8954822cf66b8/3da7d451-ed4a-4bfe-8534-e0a54455163d/78321775.py>



151. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/c9e779bd48d3b81a78e8954822cf66b8/b4546507-2afe-447e-9fd2-deee80b6f63f/3f62a6b3.py>
152. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/c9e779bd48d3b81a78e8954822cf66b8/55ab3485-423c-43ce-8406-0fe530b4743e/caff9869.json>
153. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/c9e779bd48d3b81a78e8954822cf66b8/55ab3485-423c-43ce-8406-0fe530b4743e/d695ccff.md>
154. [https://python.langchain.com/docs/concepts/chat\\_models/](https://python.langchain.com/docs/concepts/chat_models/)
155. [https://python.langchain.com/api\\_reference/langchain/chat\\_models.html](https://python.langchain.com/api_reference/langchain/chat_models.html)
156. <https://python.langchain.com/docs/integrations/chat/>
157. <https://stackoverflow.com/questions/78916056/difference-between-llm-and-chat-llm-in-langchain>
158. <https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/chatgpt>
159. [https://www.youtube.com/watch?v=v\\_HCmfTS8HA](https://www.youtube.com/watch?v=v_HCmfTS8HA)
160. <https://lunary.ai/blog/langchain-humanmessage>
161. [https://python.langchain.com/api\\_reference/core/messages/langchain\\_core.messages.human.HumanMessage.html](https://python.langchain.com/api_reference/core/messages/langchain_core.messages.human.HumanMessage.html)
162. <https://python.langchain.com/docs/concepts/messages/>
163. <https://aiproduct.engineer/tutorials/langgraph-tutorial-working-with-langchain-messages-unit-11-exercise-2>
164. [https://help.ovhcloud.com/csm/en-gb-public-cloud-ai-endpoints-chatbot-memory-langchain?id=kb\\_article\\_view&sysparm\\_article=KB0067364](https://help.ovhcloud.com/csm/en-gb-public-cloud-ai-endpoints-chatbot-memory-langchain?id=kb_article_view&sysparm_article=KB0067364)
165. <https://www.pinecone.io/learn/series/langchain/langchain-conversational-memory/>
166. [https://python.langchain.com/api\\_reference/langchain/memory/langchain.memory.summary.ConversationSummaryMemory.html](https://python.langchain.com/api_reference/langchain/memory/langchain.memory.summary.ConversationSummaryMemory.html)
167. [https://python.langchain.com/docs/concepts/chat\\_history/](https://python.langchain.com/docs/concepts/chat_history/)
168. <https://www.byteplus.com/en/topic/449744>
169. <https://milvus.io/ai-quick-reference/what-is-the-difference-between-langchain-and-other-llm-frameworks>
170. <https://notes.kodekloud.com/docs/LangChain/Interacting-with-LLMs/Messages-in-ChatModel-Demo>
171. <https://stackoverflow.com/questions/78566807/how-do-you-format-the-messages-parameters-in-openai-using-model-gpt-4-using-pyth>
172. [https://cookbook.openai.com/examples/how\\_to\\_format\\_inputs\\_to\\_chatgpt\\_models](https://cookbook.openai.com/examples/how_to_format_inputs_to_chatgpt_models)
173. <https://github.com/langchain-ai/langchain/discussions/9385>
174. <https://www.youtube.com/watch?v=jVG0A7PHuBI>
175. <https://platform.openai.com/docs/guides/text-generation>
176. <https://langchain-ai.github.io/langgraph/agents/models/>
177. <https://hexdocs.pm/langchain/LangChain.Message.html>
178. <https://mirascope.com/blog/langchain-prompt-template>
179. [https://www.reddit.com/r/Langchaindev/comments/1ajaj12/is\\_there\\_an\\_alternative\\_for\\_system\\_user\\_and/](https://www.reddit.com/r/Langchaindev/comments/1ajaj12/is_there_an_alternative_for_system_user_and/)
180. <https://www.youtube.com/watch?v=uz3BxCAubH4>
181. <https://www.leanware.co/insights/ai-agent-architecture>
182. <https://www.ibm.com/think/topics/ai-agent-types>

183. <https://langchain-ai.github.io/langgraph/how-tos/react-agent-from-scratch/>
184. [https://api.python.langchain.com/en/latest/agents/langchain.agents.react.agent.create\\_react\\_agent.html](https://api.python.langchain.com/en/latest/agents/langchain.agents.react.agent.create_react_agent.html)
185. <https://www.rohan-paul.com/p/openai-function-calling-strategy>
186. <https://mirascope.com/blog/openai-function-calling>
187. <https://www.speakeasy.com/mcp/ai-agents/architecture-patterns>
188. <https://beam.ai/agent-ic-insights/production-ready-ai-agents-the-design-principles-that-actually-work>
189. <https://arxiv.org/html/2506.12508v1>
190. <https://arxiv.org/html/2405.10467v4>
191. <https://www.computer.org/publications/tech-news/trends/agent-ic-rag/>
192. <https://arxiv.org/html/2501.09136v1>
193. <https://labelyourdata.com/articles/agent-ic-rag>
194. <https://www.deeplearning.ai/the-batch/agent-ic-design-patterns-part-2-reflection/>
195. <https://uptimerobot.com/knowledge-hub/monitoring/ai-agent-monitoring-best-practices-tools-and-metrics/>
196. <https://ardor.cloud/blog/7-best-practices-for-deploying-ai-agents-in-production>
197. <https://research.aimultiple.com/agent-deployment/>
198. <https://research.aimultiple.com/agent-ic-frameworks/>
199. <https://gravity.cx/blog/ai-agent-design-guide>
200. <https://www.lyzr.ai/blog/best-ai-agent-frameworks/>
201. <https://userjot.com/blog/best-practices-building-agent-ic-ai-systems>
202. <https://blog.n8n.io/ai-agent-frameworks/>
203. <https://www.productcompass.pm/p/building-ai-agents-best-practices>
204. <https://ardor.cloud/blog/comparison-of-scalable-agent-frameworks>
205. [https://langchain-ai.github.io/langgraph/concepts/agent-ic\\_concepts/](https://langchain-ai.github.io/langgraph/concepts/agent-ic_concepts/)
206. [https://www.reddit.com/r/LangChain/comments/1fld63q/comparison\\_between\\_the\\_top\\_rag\\_frameworks\\_2024/](https://www.reddit.com/r/LangChain/comments/1fld63q/comparison_between_the_top_rag_frameworks_2024/)
207. <https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf>
208. <https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns>
209. <https://blog.langchain.com/how-to-think-about-agent-frameworks/>
210. <https://www.anthropic.com/research/building-effective-agents>
211. <https://www.shakudo.io/blog/top-9-ai-agent-frameworks>
212. <https://airbyte.com/data-engineering-resources/using-langchain-react-agents>
213. <https://www.philschmid.de/agent-ic-pattern>
214. <https://towardsdatascience.com/create-an-agent-with-openai-function-calling-capabilities-ad52122c3d12/>
215. <https://github.com/langchain-ai/react-agent>
216. <https://platform.openai.com/docs/guides/function-calling>
217. [https://python.langchain.com/api\\_reference/\\_modules/langchain/agents/react/agent.html](https://python.langchain.com/api_reference/_modules/langchain/agents/react/agent.html)

218. [https://aiengineering.academy/Agents/patterns/multiagent\\_pattern/](https://aiengineering.academy/Agents/patterns/multiagent_pattern/)
219. <https://www.youtube.com/watch?v=JGGE9n-mTlc>
220. <https://latenode.com/blog/langchain-react-agent-complete-implementation-guide-working-examples-2025>
221. <https://docs.databricks.com/aws/en/generative-ai/guide/agent-system-design-patterns>
222. <https://community.openai.com/t/how-to-make-agent-calling-function-by-himself/1065019>
223. <https://www.youtube.com/watch?v=u772oTNStmU>
224. <https://kodexolabs.com/agentic-rag-with-ai-agents/>
225. <https://galileo.ai/blog/ai-agent-architecture>
226. <https://azure.microsoft.com/en-us/blog/agent-factory-top-5-agent-observability-best-practices-for-reliable-ai/>
227. [https://www.reddit.com/r/Al\\_Agents/comments/1gxokte/how\\_are\\_you\\_monitoringdeploying\\_your\\_ai\\_agents\\_in/](https://www.reddit.com/r/Al_Agents/comments/1gxokte/how_are_you_monitoringdeploying_your_ai_agents_in/)
228. <https://www.ibm.com/think/topics/agentic-architecture>
229. <https://everworker.ai/blog/what-is-retrieval-augmented-generation-a-guide-to-rag>
230. <https://microsoft.github.io/ai-agents-for-beginners/10-ai-agents-production/>
231. <https://aws.amazon.com/what-is/ai-agents/>
232. <https://python.langchain.com/docs/tutorials/rag/>
233. <https://github.com/asinghcsu/AgenticRAG-Survey>
234. <https://www.zenml.io/llmops-database/best-practices-for-ai-agent-development-and-deployment>
235. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/cbd45bebb2b618c861f866f82cb31a07/978e2bdb-cd68-432b-835e-1afc53812670/fd35fe1f.py>
236. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/cbd45bebb2b618c861f866f82cb31a07/ea4f3960-7a95-4d58-954f-ae53a822fcd2/84ffa2de.py>
237. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/cbd45bebb2b618c861f866f82cb31a07/aeacb0e8-904a-4f2b-80bf-e0034f75a997/f6ddcdec.py>
238. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/cbd45bebb2b618c861f866f82cb31a07/8529a730-d969-42b7-8063-4be75be6bf83/b3356305.md>
239. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/cbd45bebb2b618c861f866f82cb31a07/8529a730-d969-42b7-8063-4be75be6bf83/819087f7.json>