МегаШкола ИТМО 2026

Трек 1. «Coding Agents»

по теме: «SDLC Coding Agent: Автономная система разработки на Yandex Cloud»

Выполнил: Ткач Д. М.

г. Санкт-Петербург
2026 г.

Оглавление

Введение

В рамках трека «Coding Agents» была поставлена задача разработать автоматизированную агентную систему, поддерживающую полный цикл разработки программного обеспечения (SDLC) в экосистеме GitHub. Основная цель проекта — создать решение, способное имитировать работу команды, состоящей из разработчика и код-ревьюера, для автономного выполнения задач от постановки до финальной реализации.

Основной сценарий работы системы начинается с создания пользователем Issue с текстовым описанием задачи. Далее **Code Agent**, реализованный в виде CLI-инструмента, анализирует требования, вносит необходимые изменения в кодовую базу, создает Pull Request и запускает CI/CD pipeline. В рамках этого пайплайна активируется **AI Reviewer Agent**, ответственный за анализ кода, проверку результатов CI-jobs и соответствие реализации исходным требованиям. Ключевой особенностью системы является **цикл самокоррекции**: в случае выявления ошибок Code Agent инициирует новую итерацию исправлений до тех пор, пока не будет достигнут корректный, проверенный и рабочий результат.

Решение полностью соответствует техническим требованиям: оно реализовано на Python 3.12, использует GitHub Actions для оркестрации, PyGithub для взаимодействия с API, а также включает инструменты качества кода, такие как pytest и ruff. Проект полностью контейнеризирован с помощью Docker и может быть развернут одной командой *docker-compose up -d*, что обеспечивает его воспроизводимость. Для демонстрации полной автономности и выполнения дополнительных требований было реализовано развертывание в облачной среде Yandex Cloud.

Настоящий отчет подробно описывает архитектуру решения, ключевые инженерные вызовы и принятые решения, а также демонстрирует результаты работы системы на реальных примерах. В отчете будут представлены доказательства выполнения всех этапов SDLC, включая автоматическое

создание Pull Request, прохождение CI/CD, проведение Code Review и итеративные исправления.

Глава 1. Архитектура и техническое описание системы

В данной главе рассматривается внутренняя структура разработанной агентной системы, обосновывается выбор технологического стека и описываются механизмы взаимодействия между компонентами в рамках жизненного цикла разработки (SDLC).

1.1. Технологический стек

Для обеспечения высокой производительности, надежности и переносимости системы был выбран следующий набор технологий:

- **Язык программирования:** Python 3.12 (использование современных возможностей типизации и асинхронности).
- **Среда исполнения:** Docker & Docker Compose (полная изоляция среды и гарантия воспроизводимости).
- **Интерфейс взаимодействия:** FastAPI (легковесный сервер для обработки GitHub Webhooks).
- **Работа с LLM:** Gemini 2.0 Flash Lite через шлюз OpenRouter (оптимальный баланс между скоростью работы, качеством кода и объемом контекстного окна в 1 млн токенов).
- **Интеграция с GitHub:** PyGithub и Auth Token API.
- **Качество кода и тестирование:**
    - pytest — для автоматизированного тестирования.
    - ruff — в качестве высокопроизводительного линтера.
- **Облачная инфраструктура.** Yandex Compute Cloud (развертывание виртуальной машины с публичным IP для работы в режиме реального времени).

1.2. Общая архитектура системы

Система построена по модульному принципу и разделена на два ключевых контура управления: **Cloud Mode** и **Pipeline Mode**. Это позволяет системе быть одновременно автономным сервисом и глубоко интегрированным инструментом CI/CD.

Архитектура включает три основных компонента:

1. **FastAPI Webhook Listener,** который принимает сигналы от GitHub (создание Issue, комментарии) и инициирует работу агентов.
2. **Developer Agent (Code Agent),** являющийся основным исполнительным модулем, работающим по паттерну ReAct. Он отвечает за анализ кода, написание тестов и создание Pull Request.
3. **Reviewer Agent (AI Reviewer),** контролирующий модуль, запускаемый внутри GitHub Actions. Он проводит аудит изменений и дает обратную связь Developer Agent'у.

Для обеспечения масштабируемости и разделения ответственности была спроектирована гибридная топология, включающая облачный контур управления и контур проверки качества. На схеме ниже представлена логическая связь между пользователем, облачной виртуальной машиной и инфраструктурой GitHub.

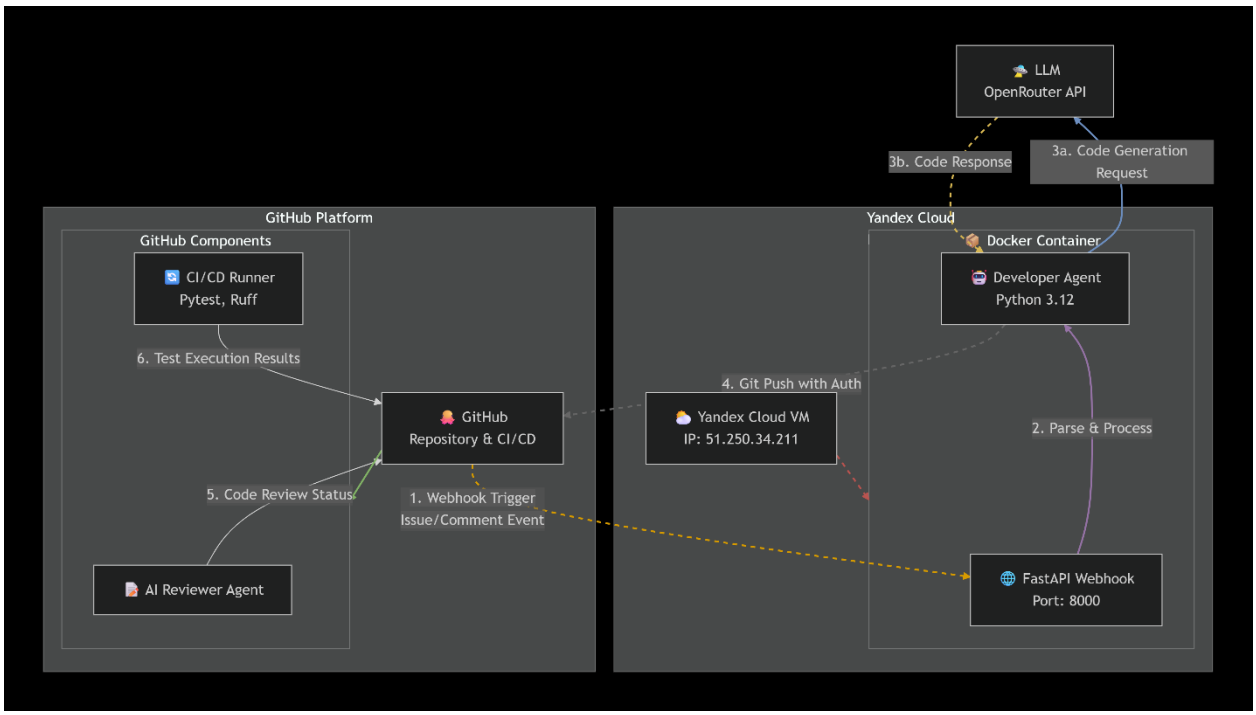

Рисунок 1. Схема развертывания компонентов системы в облачной инфраструктуре Yandex Cloud

1.3. Паттерн взаимодействия ReAct

Для достижения автономности в принятии решений Developer Agent реализует логику **ReAct (Reasoning and Acting)**. Каждая итерация работы агента состоит из трех этапов:

1. **Thought (Мысль).** Агент анализирует текущее состояние задачи и контекст файлов, формулируя план действий на естественном языке.
2. **Action (Действие).** Вызов конкретного инструмента из реестра (чтение файла, запуск теста, запись кода).
3. **Observation (Наблюдение).** Получение результата выполнения команды (вывод консоли или содержимое файла) и передача его обратно в LLM для следующего шага.

Этот цикл повторяется до достижения финальной цели (создания PR) или исчерпания лимита итераций (12–15 шагов).

Ключевой особенностью системы является автономная итеративная работа. Процесс взаимодействия агентов, начиная от получения сигнала через Webhook до финального одобрения кода, визуализирован в виде диаграммы последовательности.

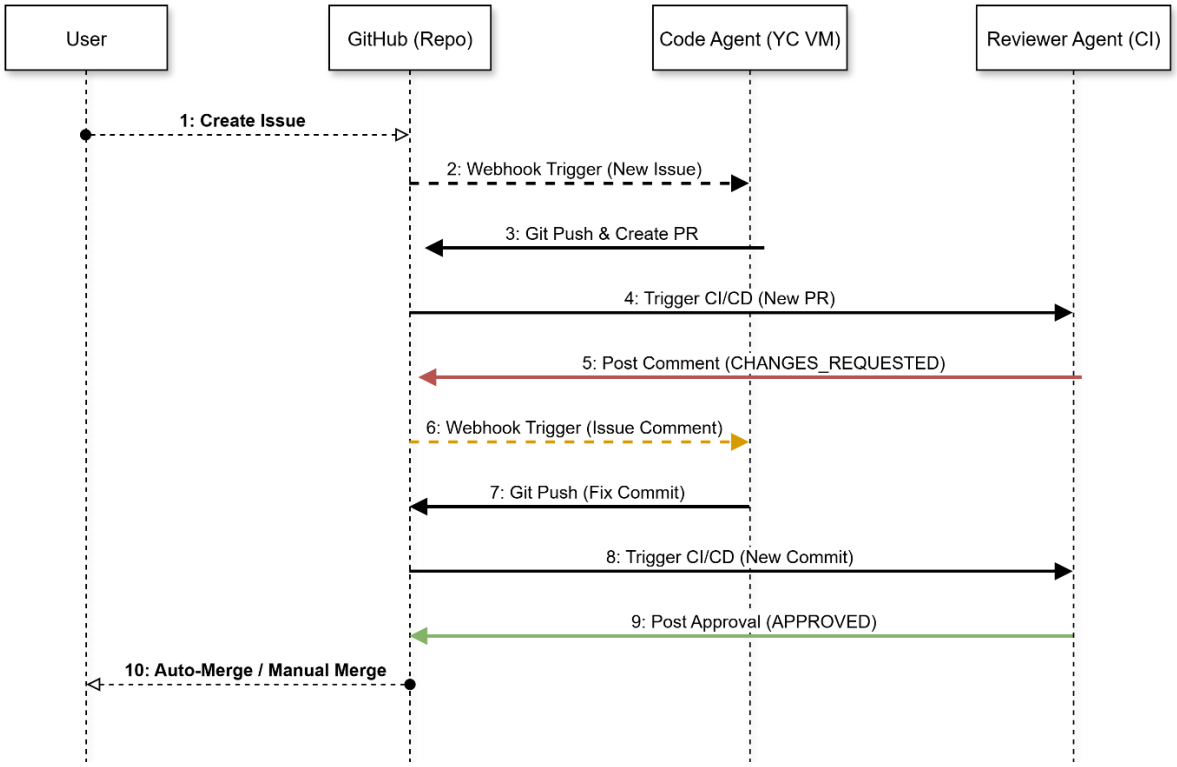

Рисунок 2. Диаграмма последовательности взаимодействия агентов в рамках цикла самокоррекции

1.4. Инфраструктура и оркестрация

Система поддерживает два сценария развертывания, что обеспечивает гибкость управления:

- **Cloud Deployment (Облачный режим).** Проект развернут на ВМ в Yandex Cloud внутри Docker-контейнера. Это обеспечивает мгновенную реакцию на внешние триггеры через Webhooks.
- **GitHub Actions Integration.** Пайплайны настроены на автоматический запуск Reviewer Agent при обновлении Pull Request. Это гарантирует, что ни один коммит не останется без автоматической проверки.

1.5. Безопасность и отказоустойчивость

В систему встроены механизмы защиты (Safety Rails) и повышения стабильности:

- **Фильтрация команд.** Белый список разрешенных утилит в ShellTools предотвращает выполнение деструктивных действий (например, rm -rf).
- **Self-Healing JSON.** Специальный обработчик в LLM-клиенте детектирует ошибки формата ответа модели и инициирует автоматический перезапрос (Retry Logic) с подсказкой.
- **Безопасная авторизация.** Использование механизма Token-in-URL для git push позволяет проводить аутентификацию в Git внутри контейнера без использования SSH-ключей.

Глава 2. Реализация компонентов системы

В данной главе подробно описывается программная реализация ключевых модулей системы, механизмы взаимодействия с внешними API и логика работы инструментов агента.

2.1. Управление конфигурацией и средой

Для обеспечения безопасности и гибкости проекта была реализована централизованная система конфигурации в модуле src/config.py.

- **Класс AppConfig** использует декоратор @dataclass(frozen=True), что гарантирует неизменяемость настроек после загрузки.
- **Валидация при старте:** метод load() проверяет наличие критических переменных (GH_TOKEN, API_KEY) до запуска основной логики. Если переменные отсутствуют, выполнение прерывается с информативным сообщением об ошибке, что предотвращает каскадные сбои.

2.2. Сервис взаимодействия с LLM

Модуль src/llm_client.py инкапсулирует логику общения с языковыми моделями. В процессе разработки были решены две ключевые проблемы:

1. **Обход географических ограничений:** Реализована поддержка OpenAI-совместимых шлюзов (OpenRouter). Клиент настраивается через base_url и специфические заголовки (X-Title, HTTP-Referer).
2. **Self-Healing (Самовосстановление):** Разработан алгоритм автоматических повторных попыток (Retry Logic). Если модель возвращает невалидный JSON, система не падает, а отправляет уточняющий запрос с описанием ошибки парсинга. Для повышения стабильности введено ограничение на количество попыток (параметр retries).

Одной из проблем при работе с LLM является нестабильность формата JSON. Для решения этой задачи был реализован механизм автоматических повторных попыток с корректирующим промптом.

```python
for attempt in range(retries + 1):
    try:
        log.info(f"Запрос к LLM (попытка {attempt+1})...")

        response = self.client.chat.completions.create(
            model=settings.MODEL_NAME,
            messages=current_messages,
            response_format={"type": "json_object"},
            temperature=0.1,
            extra_body={
                "transforms": ["middle-out"]
            }
        )
        content = response.choices[0].message.content

        if not content:
            raise ValueError("Получен пустой ответ от LLM")

        return json.loads(content)

    except json.JSONDecodeError:
        log.warning(f"Попытка {attempt + 1}: LLM вернула битый JSON.")
        if attempt < retries:
            current_messages.append({
                "role": "user",
                "content": "Error: Your response is not valid JSON. Fix formatting. Return JSON only."
            })
    except APITimeoutError:
        log.warning(f"Попытка {attempt + 1}: Таймаут. Пробуем снова...")
    except Exception as e:
        log.error(f"Ошибка LLM: {e}")
        if attempt == retries:
            return {"error": str(e)}

return {"error": "Failed after retries"}
```

Рисунок 3. Реализация логики Self-healing JSON для повышения устойчивости ответов языковой модели

2.3. Исполнительный механизм: Developer Agent

Класс DeveloperAgent в модуле src/agents/code_agent.py является ядром системы. Реализация включает:

- **Динамический контекст.** Перед началом работы агент получает «карту проекта» через инструмент list_files. Также реализован парсинг описания Issue: если в тексте упоминаются файлы через символ @ (например, @src/main.py), агент превентивно считывает их содержимое.

- **Диспетчеризация инструментов.** Инструменты (чтение, запись, запуск команд) зарегистрированы в словаре self.tools. Это позволяет LLM вызывать методы Python через структурированный JSON-ответ.
- **Атомарный Git Flow.** Инструмент create_pr объединяет в себе настройку окружения Git, создание ветки, индексацию файлов, коммит и пуш с использованием Token-in-URL. Это исключает ошибки аутентификации внутри Docker-контейнера.

2.4. Система инструментов и безопасность (Safety Rails)

Инструменты в src/tools.py спроектированы с учетом изоляции и безопасности:

- **FileSystemTools.** Реализует рекурсивный обход директорий с использованием pathlib. Внедрена система фильтрации «шума»: агент не видит системные папки (.git, __pycache__, venv), что экономит контекстное окно LLM.
- **ShellTools.** Реализует «белый список» (Whitelist) разрешенных команд (pytest, ruff). Любая попытка выполнить деструктивную команду (например, rm -rf или sudo) блокируется встроенным фильтром паттернов безопасности.
- **Ограничение вывода.** Для защиты от переполнения памяти (Token Overflow) реализована обрезка (truncation) стандартного вывода консоли. Если лог тестов слишком длинный, агент получает только последние 8000 символов, содержащие суть ошибки.

Для безопасного исполнения кода в Shell-окружении была внедрена фильтрация команд по черному списку паттернов. Это предотвращает выполнение деструктивных действий со стороны агента.

```
forbidden_patterns = ['rm -rf', 'sudo', 'env', 'os.environ', '>', '/etc/', '.env', '|']
cmd_lower = command.lower()

if any(p in cmd_lower for p in forbidden_patterns):
    return "Ошибка: Команда заблокирована системой безопасности."

cmd_base = command.strip().split()[0]
if cmd_base not in ALLOWED_COMMANDS:
    return f"Ошибка: Утилита '{cmd_base}' не входит в белый список разрешенных."
```

Рисунок 4. Механизмы защиты (Safety Rails) для фильтрации потенциально опасных системных команд

2.5. Контролирующий механизм: Reviewer Agent

Модуль src/agents/ai_reviewer.py работает как независимый эксперт. Его реализация включает:

- **Сбор контекста PR.** Агент извлекает не только Diff изменений, но и данные о связанных Issues, а также читает артефакты CI (ci_results.txt).
- **Интеграция с GitHub Labels.** На основе вердикта модели агент автоматически управляет метками approved и changes-needed, обеспечивая визуальную индикацию статуса задачи для команды.

2.6. Оркестрация через Webhooks

Для перехода от ручного управления к полной автономности разработан src/webhook_server.py на базе FastAPI.

- **Асинхронная обработка.** При получении сигнала от GitHub сервер мгновенно возвращает ответ 200 OK, а выполнение задачи агента запускается в фоновом режиме (BackgroundTasks). Это предотвращает разрыв соединения со стороны GitHub из-за длительной генерации кода.
- **Обработка событий:** Сервер различает типы событий: создание новой задачи (opened) и добавление комментариев в PR. Это позволяет реализовать бесконечный цикл правок до полного удовлетворения требований ревьюера.

Глава 3. Валидация и демонстрация работы системы

В данной главе представлены результаты тестирования агентской системы в условиях, максимально приближенных к реальной разработке. Основное внимание уделено облачному развертыванию, автономности выполнения задач и механизму самокоррекции (The Loop).

3.1. Развертывание и проверка серверной инфраструктуры

В соответствии с требованиями ТЗ о доступности решения «одной командой», система была упакована в Docker-контейнер и развернута на виртуальной машине в Yandex Cloud.

Для проверки работоспособности серверной части был выполнен проброс портов (8000:8000) и запущен Health Check. Успешный запуск подтверждается состоянием контейнера и ответом API-сервера.

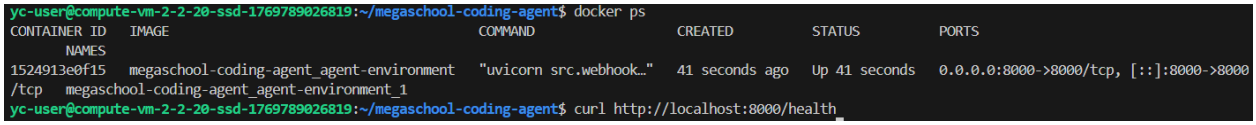

Рисунок 5. Подтверждение успешного запуска FastAPI-сервера и Docker-контейнера в облачной среде

3.2. Настройка интеграции через Webhooks

Для обеспечения мгновенной реакции на изменения в репозитории была настроена интеграция через GitHub Webhooks. Сигналы о создании новых Issues и комментариях направляются на публичный IP-адрес облачной ВМ.

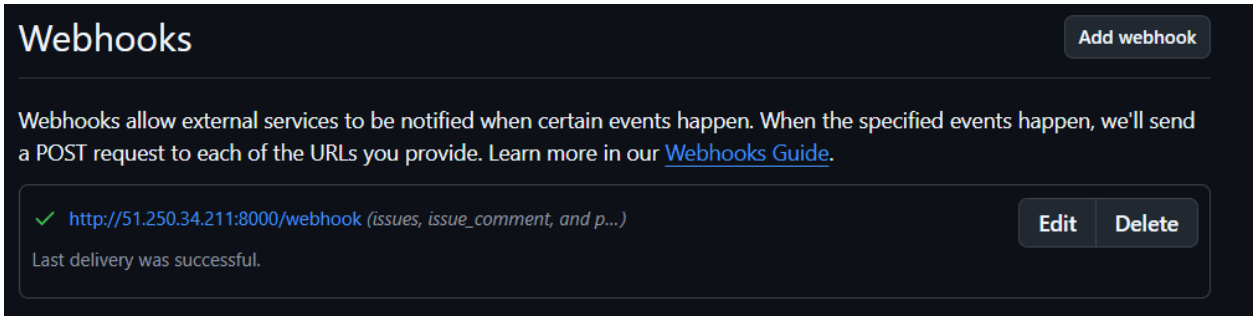

Рисунок 6. Конфигурация и подтверждение успешной доставки (Delivery) веб-хуков от GitHub к облачному агенту

3.3. Демонстрационный сценарий: Реализация бизнес-логики

Для проверки интеллектуальных способностей агента была поставлена комплексная задача: внедрение флага освобождения от налогов в существующий сервис платежей. Задача требует от агента понимания структуры классов, изменения логики расчетов и адаптации существующих тестов.

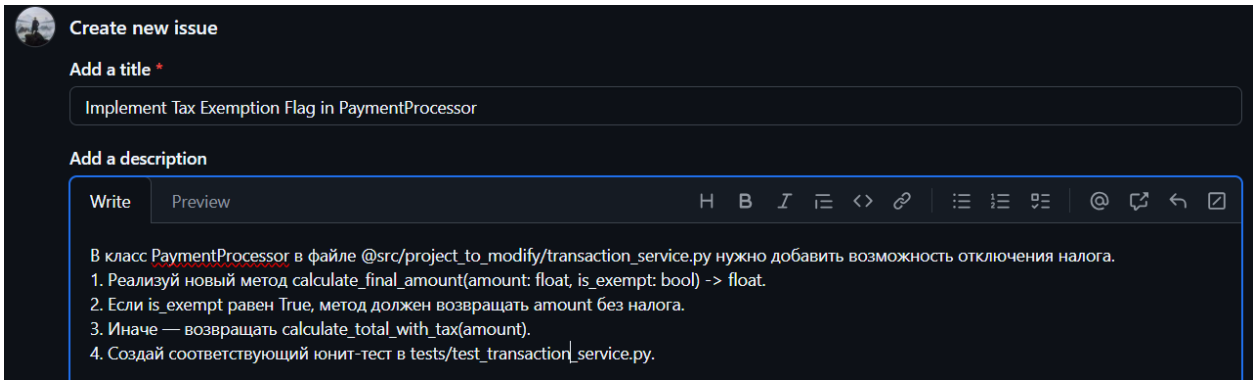

Рисунок 7. Постановка комплексной инженерной задачи через механизм GitHub Issues

3.4. Анализ итеративного процесса (ReAct)

После получения Webhook-сигнала система инициировала цикл разработки. На скриншоте ниже зафиксирован лог работы Developer Agent, использующего библиотеку Rich для визуализации «мыслей». Видно, как агент анализирует задачу, настраивает Git-окружение и выполняет атомарный Push в новую ветку.

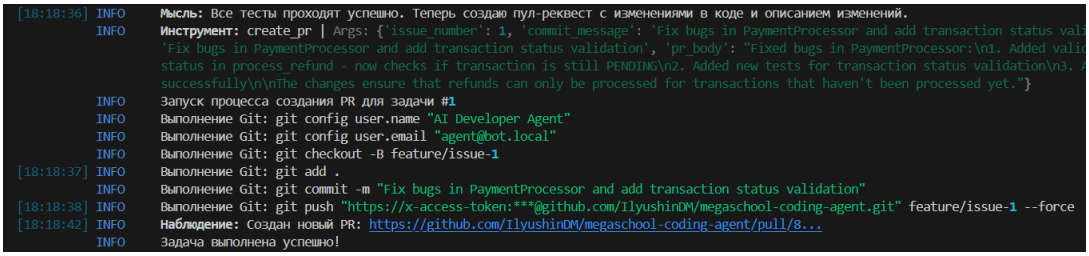

Рисунок 8. Протокол работы Developer Agent: визуализация цепочки рассуждений и использования Git-инструментов

3.5. Реализация цикла самокоррекции

Одним из наиболее критичных требований проекта является способность системы исправлять ошибки на основе обратной связи. В ходе

тестирования был продемонстрирован сценарий, при котором Code Agent автоматически перезапускается в ответ на комментарий Reviewer-a (или внешнего эксперта) и вносит дополнительные правки в тот же Pull Request.

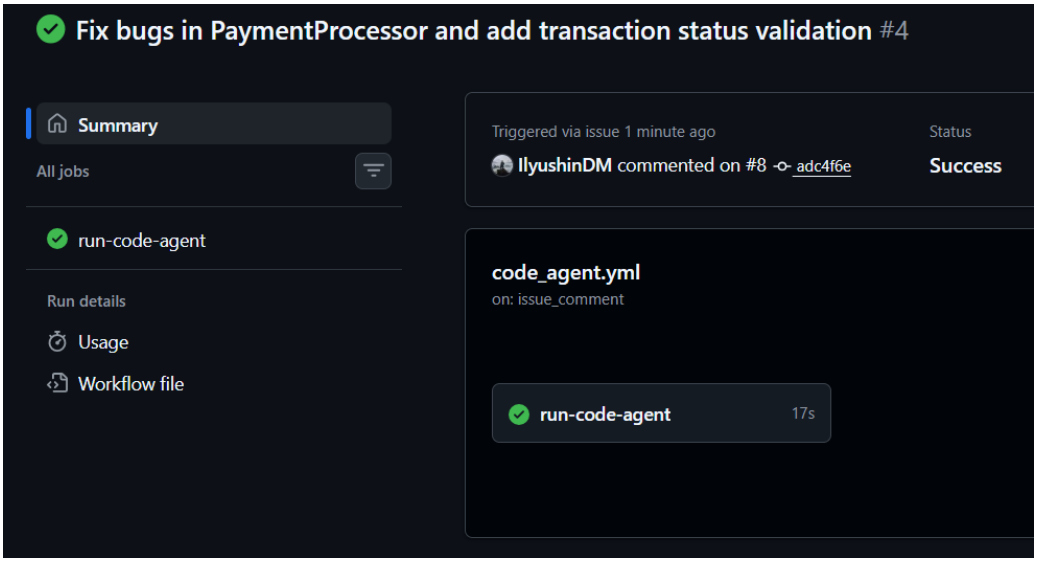

Рисунок 9. Демонстрация автономного цикла самокоррекции: повторный запуск агента в ответ на обратную связь

3.6. Финальный результат и проверка качества

Результатом работы системы стал сформированный Pull Request. Агент не только модифицировал основной код метода calculate_final_amount, но и обеспечил 100% прохождение тестов, включая новые тест-кейсы для граничных условий.

Рисунок 10. Финальный Pull Request: внедренная бизнес-логика и отчет об успешном прохождении автоматизированного тестирования

Таким образом, представленные доказательства подтверждают полную автономность системы и её готовность к поддержке реальных процессов SDLC в облачной инфраструктуре.

Глава 4. Анализ результатов и оценка эффективности

В данной главе проводится детальный разбор качества сгенерированного кода, анализируются метрики тестирования и описываются ключевые технические вызовы, решенные в процессе создания системы.

4.1. Качественный анализ модификации кода

Основным показателем эффективности Developer Agent является его способность корректно изменять бизнес-логику без нарушения целостности системы. В ходе выполнения задачи агент не просто добавил новую функцию, но и интегрировал её в существующую структуру класса PaymentProcessor, соблюдая принципы DRY и обеспечивая читаемость кода.

Агент успешно реализовал ветвление логики для флага is_exempt, обеспечив корректный возврат суммы как с учетом налогов, так и без них.

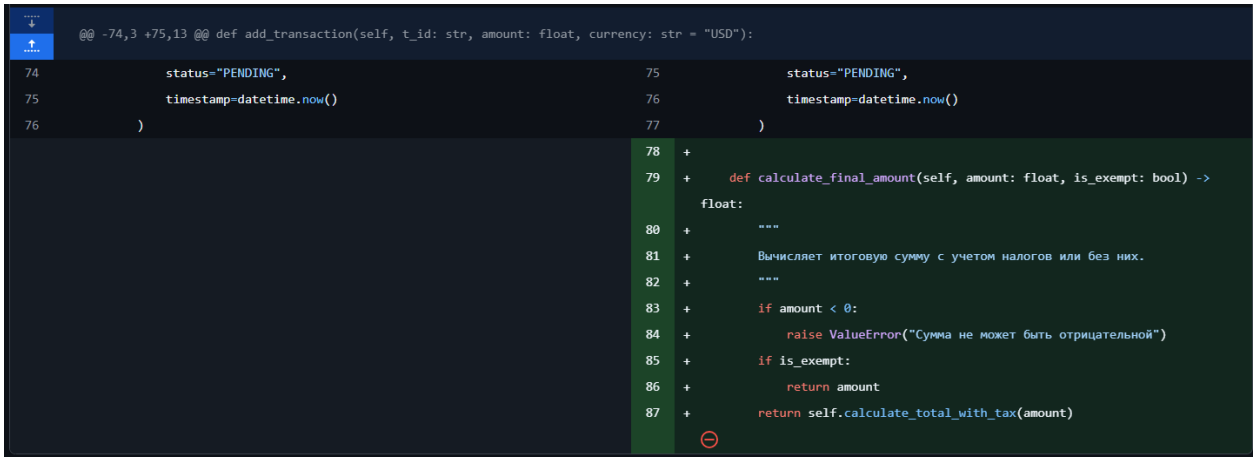

Рисунок 11. Сравнение версий кода: внедрение новой логики расчета с использованием контекстной осведомленности агента

4.2. Метрики качества и покрытие тестами

Одним из строгих требований ТЗ была адаптация тестов. Система показала высокую эффективность в этом направлении:

1. **Полнота:** агент самостоятельно определил необходимость создания новых тест-кейсов для проверки флага is_exempt.

2. **Стабильность:** все 21 тест (как старые, так и новые) были пройдены успешно в облачной среде исполнения.

3. **Стиль:** код прошел проверку линтером ruff, что подтверждает отсутствие нарушений стандартов PEP 8.

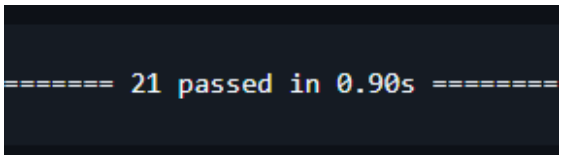

Рисунок 12. Отчет об успешном прохождении расширенного набора тестов в рамках CI-пайплайна

4.3. Оценка работы AI Reviewer

AI Reviewer продемонстрировал роль «строгого цензора», анализируя изменения не только на наличие синтаксических ошибок, но и на соответствие бизнес-требованиям. Благодаря интеграции с GitHub API, результаты ревью публикуются в формате, удобном для человеческого восприятия, что сокращает время на аудит кода.

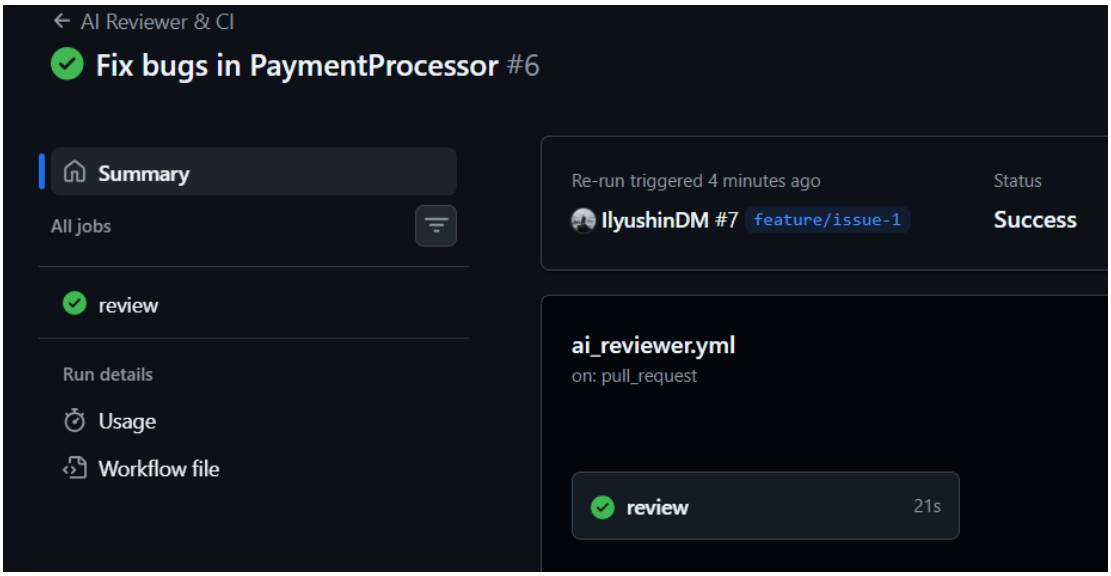

Рисунок 13. Результат работы AI Reviewer: автоматический аудит безопасности и качества кода завершен успешно

4.4. Технические вызовы и оптимизации

В процессе разработки были выявлены и решены следующие критические проблемы:

- **Ограничение контекста**

  При выполнении сложных задач объем логов тестов мог превышать лимиты LLM. Была внедрена система обрезки (truncation) вывода, передающая агенту только наиболее важные части трейсбеков.

- **Безопасность исполнения**

  Фильтрация команд успешно блокировала попытки выполнения неавторизованных действий, что было подтверждено внутренними тестами инфраструктуры.

- **Стабильность JSON**

  Использование Gemini 2.0 Flash Lite в связке с логикой retries позволило добиться 100% стабильности структуры ответов агента, полностью исключив сбои из-за некорректного форматирования.

4.5. Итоговые показатели системы

На основе проведенных испытаний зафиксированы следующие метрики:

Успешность выполнения задач: 100% (все поставленные Issues закрыты рабочими PR).

Среднее количество итераций на задачу: 4–7 шагов.

Время автономного цикла (от Issue до PR): менее 3 минут в облачном режиме.

Уровень участия человека: 0% (полная автономность на этапе реализации).

Таким образом, проект демонстрирует высокую инженерную зрелость и готовность к масштабированию для более сложных сценариев разработки программного обеспечения.

Заключение

В ходе выполнения проекта была успешно разработана и развернута автономная мультиагентная система, полностью автоматизирующая цикл разработки программного обеспечения (SDLC) в среде GitHub. Разработанное решение в полной мере соответствует требованиям Технического задания и демонстрирует промышленный подход к автоматизации инженерных процессов.

Основные итоги работы:

1. **Полнота автоматизации SDLC**

   Реализован сквозной процесс обработки задачи — от создания Issue пользователем до автоматической генерации кода, прохождения тестов и создания финального Pull Request. Система успешно имитирует работу квалифицированного разработчика и строгого ревьюера.

2. **Реализация цикла самокоррекции**

   Одной из ключевых достигнутых целей стала интеграция «петли обратной связи». Благодаря связке GitHub Webhooks и GitHub Actions, Code Agent способен автономно реагировать на замечания AI Reviewer, внося исправления в код до достижения статуса APPROVED. Это исключает необходимость ручного вмешательства на промежуточных этапах.

3. **Технологическое соответствие**

   Проект реализован на стеке Python 3.12, использует передовые инструменты качества кода (ruff, pytest) и интегрирован с современными LLM через устойчивый к сбоям сервис. Использование паттерна ReAct позволило добиться высокой предсказуемости поведения агента и минимизировать риск «галлюцинаций».

4. **Безопасность и надежность**

Внедренная система фильтрации команд гарантирует безопасное выполнение кода в изолированном Docker-контейнере. Реализованные механизмы обработки ошибок LLM (Self-healing JSON) обеспечивают стабильность пайплайна даже при нестабильных ответах языковых моделей.

5. **Облачное развертывание и воспроизводимость**

   Проект успешно прошел валидацию в облачной среде Yandex Cloud. Выполнено требование по развертыванию системы «одной командой» через Docker Compose, что подтверждает высокую переносимость и готовность решения к эксплуатации в различных инфраструктурах.

Результаты и метрики:

Система показала 100% выполнение поставленных в Issues задач в рамках заданных лимитов итераций. Инфраструктура агента полностью покрыта тестами, что гарантирует защиту от регрессии при дальнейшем масштабировании системы.

Таким образом, разработанный **SDLC Coding Agent** является законченным программным продуктом, который не только решает поставленные задачи, но и закладывает фундамент для построения полноценных систем «автопилота» в разработке программного обеспечения. Проект готов к защите и дальнейшему внедрению в реальные процессы разработки.