Введение

Актуальность темы

Современный этап развития информационных технологий сопровождается не только стремительным ростом объёмов программного кода, но и усложнением архитектурных решений программных систем, а также всё более активным внедрением инструментов искусственного интеллекта (ИИ) в процессы разработки. Согласно отчёту State of AI 2024, уже более 60% IT-компаний по всему миру применяют ИИ для анализа кода, автоматической генерации функций, тестирования и поиска ошибок. Вместе с тем набирают популярность интеллектуальные ассистенты, интегрируемые в среды разработки, такие как GitHub Copilot, Amazon CodeWhisperer, Cursor и другие.

На фоне масштабной цифровизации экономики особенно актуальным становится создание умных инструментов, способных не только повышать производительность разработчика, но и упрощать освоение сложных кодовых баз, обеспечивая лучшее понимание архитектуры проекта. Значительную ценность приобретают системы, позволяющие создавать наглядные визуальные карты кода, способствующие лёгкой навигации по большим репозиториям, а также включающие возможности взаимодействия с ИИ прямо в среде разработки.

Однако, в большинстве существующих решений либо требуется обязательное подключение к облачным сервисам — что влечёт за собой риски безопасности и может быть недоступным в ряде организаций, — либо отсутствует глубокий анализ и качественная визуализация архитектуры. В этих условиях особенно важной становится задача обеспечения локального и автономного использования языковых моделей в сочетании с визуальными средствами анализа.

Таким образом, актуальность данного исследования обусловлена противоречием между запросом со стороны разработчиков на безопасные, локальные и интеллектуальные средства навигации и анализа кода и отсутствием доступных комплексных решений, способных объединить визуализацию архитектуры проекта и работу локальной языковой модели.

Степень разработанности темы

Интерес к применению языковых моделей и визуальных средств в сфере программной инженерии продолжает устойчиво расти. В научной литературе широко рассматриваются подходы к анализу кода с использованием трансформеров, методы построения графов вызовов и архитектурных диаграмм — в работах таких авторов, как J. Allamanis, M. Brockschmidt, A. Hindle и других. Однако в большинстве исследований основное внимание сосредоточено на алгоритмической составляющей — определении зависимостей, извлечении семантики, классификации элементов кода.

В индустриальной практике стремительно развиваются инструменты, включающие ИИ в процесс разработки — среди них GitHub Copilot, Tabnine, Sourcegraph Cody. Но, как правило, такие решения завязаны на облачную инфраструктуру и не предоставляют разработчику средств интерактивной визуализации архитектуры проекта. Интеграция локальных языковых моделей, включая Qwen3-MNN, в среду разработки и объединение их с визуализацией кода пока находятся преимущественно в стадии экспериментов и прототипов.

Именно поэтому рассматриваемая тема располагается на пересечении двух значимых направлений — использования локального ИИ в IDE и построения визуальной архитектуры кода, оставаясь при этом недостаточно проработанной как с научной, так и с прикладной точки зрения.

Цель и задачи исследования

Целью данной работы является создание расширения для Visual Studio Code, которое объединяет функции интерактивной визуализации кодовой базы проекта и локального взаимодействия с языковой моделью Qwen3-MNN.

Для достижения поставленной цели сформулированы следующие задачи:

Теоретические задачи:

1. Провести анализ существующих подходов к визуализации архитектуры программных проектов и построению графов кода.
2. Изучить потенциал современных языковых моделей, в частности Qwen3-MNN, в контексте их использования в офлайн-режиме в среде разработки.

Практические задачи:  
3. Разработать расширение для Visual Studio Code, обеспечивающее визуализацию кодовой базы на основе графов вызовов и зависимостей.  
4. Интегрировать в расширение чат-панель для взаимодействия с локальной языковой моделью и поддержкой типичных пользовательских сценариев.

Объект и предмет исследования

Объектом исследования выступает процесс взаимодействия программиста со средой разработки при анализе и навигации по кодовой базе.

Предмет исследования охватывает методы и технологии визуализации архитектуры программных систем, а также использование локальных языковых моделей в среде разработки (IDE).

Научная новизна и практическая значимость

Научная новизна проекта заключается в разработке методологического подхода, позволяющего объединить архитектурную визуализацию кода и взаимодействие с локальной языковой моделью в едином расширении для Visual Studio Code. Дополнительно новизна выражается в адаптации модели Qwen3-MNN к задачам анализа кода и обработке запросов разработчика в автономном режиме.

Практическая значимость определяется созданием удобного и эффективного инструмента, способного повысить продуктивность и качество работы программиста, при этом обеспечивая автономность и высокий уровень безопасности благодаря отсутствию необходимости в сетевом подключении и использовании облачных платформ.

1. Анализ предметной области и существующих решений

### 1.1. Проблемы навигации и анализа в крупных кодовых базах

#### Проблемы масштабируемости

Современные программные системы нередко насчитывают десятки, а порой и сотни миллионов строк кода, что порождает серьёзные трудности в их анализе и сопровождении. В таких условиях традиционные средства навигации — основанные на текстовых редакторах и ручном поиске — становятся малоэффективными. Так, компания Facebook разработала инструмент статического анализа Zoncolan, способный сканировать массив более чем в 100 миллионов строк кода менее чем за полчаса, выявляя потенциальные уязвимости и ошибки. Этот пример ярко демонстрирует необходимость автоматизации процессов анализа для обеспечения масштабируемости и повышения продуктивности работы с крупными кодовыми базами.

#### Сложности визуализации связей между компонентами

Визуализация архитектурной структуры и взаимодействий компонентов в больших проектах также представляет собой значительный вызов. Графовые модели, такие как деревья вызовов или диаграммы зависимостей, быстро становятся перегруженными и теряют информативность по мере увеличения масштаба системы. Например, инструмент Rigi, предназначенный для обратной разработки, испытывает сложности при визуализации графов с числом узлов свыше 500, что ограничивает его применимость в крупных проектах.

Для преодоления подобных ограничений разрабатываются альтернативные методы визуализации. Один из таких — метафора "города кода", в рамках которой программные компоненты представлены в виде зданий и улиц. Это позволяет разработчикам задействовать пространственное мышление для лучшего понимания структуры системы. Компания Galois использует данный подход для создания трёхмерных визуализаций кодовых баз, облегчая ориентацию в архитектуре сложных систем.

Инструменты визуализации становятся ключевыми помощниками в процессе анализа и навигации. Они позволяют разработчику эффективно выявлять зависимости, отслеживать архитектуру и быстрее ориентироваться в проекте. Современные решения, такие как SPViz, предоставляют возможности для построения настраиваемых визуализаций с применением доменно-специфических языков, что делает их гибкими и адаптируемыми к требованиям конкретных систем.

Существуют также инструменты, такие как CodeSee, которые автоматически создают и обновляют карты кода, визуализируя зависимости и потоки данных в приложении. Это особенно важно при работе с распределёнными системами и микросервисной архитектурой, где понимание взаимодействий между компонентами имеет решающее значение для эффективной разработки и поддержки.

Дополнительную ценность даёт интеграция статического и динамического анализа, реализованная, к примеру, в инструменте ExplorViz. Она позволяет получить комплексное представление о поведении системы и выявить потенциальные узкие места на ранних этапах жизненного цикла программного обеспечения.

Таким образом, эффективная навигация и глубокий анализ крупных кодовых баз требуют внедрения современных визуальных и аналитических инструментов. Их применение существенно облегчает понимание архитектуры и поведения системы, ускоряет разработку и способствует повышению её качества. Продолжение развития технологий визуализации и автоматизированного анализа остаётся важным направлением для работы с масштабными программными проектами.

### 1.2. Обзор существующих решений

Развитие инструментов поддержки программной инженерии отражает возрастающую потребность в повышении продуктивности, управляемости больших кодовых баз и автоматизации рутинных задач. Особенно востребованными стали средства визуализации кода и генеративные ИИ-помощники, интегрируемые в популярные среды разработки, такие как Visual Studio Code (VS Code). Ниже приводится обзор ключевых решений в этих направлениях с акцентом на их функциональные возможности, текущие ограничения и потенциал для дальнейшего развития.

#### 1. Инструменты визуализации кода в VS Code

**CodeMap**  
Это расширение отображает структуру проекта в виде дерева или графа, позволяя получить обзор по файлам, классам и методам. Однако его возможности ограничиваются структурой внутри проекта, не охватывая вызовы между функциями, сложные зависимости или метрики связанности.

Ограничения:

* Отсутствие интерактивности — невозможность перехода к коду по узлу.
* Нет поддержки анализа межфайловых связей.
* Недостаточная глубина анализа для популярных языков, например, Python.

**GitLens**  
Популярное расширение, предназначенное для анализа истории изменений, авторства и сравнения версий файлов. Предоставляет граф коммитов и временные диаграммы, отображая эволюцию кода прямо в редакторе.

Ограничения:

* Сфокусировано на системах контроля версий, а не на архитектуре проекта.
* Не объединяет данные анализа с визуальной моделью кода.
* Может быть избыточным при работе с небольшими репозиториями.

**Import Cost**  
Анализирует вес импортируемых библиотек (в основном JS/TS) и отображает его в редакторе. Помогает следить за крупными зависимостями в фронтенд-проектах.

Ограничения:

* Ограничено фронтенд-стеком.
* Не визуализирует зависимости в графовой форме.
* Не анализирует структуру собственного кода проекта.

#### 2. Генеративные ИИ-помощники

**GitHub Copilot**  
Разработанный GitHub и OpenAI, Copilot стал одним из первых коммерчески успешных ИИ-инструментов для автодополнения кода. Он генерирует функции по описанию, предлагает улучшения и фрагменты.

Ограничения:

* Непрозрачность — сложно отследить происхождение сгенерированного кода.
* Не анализирует архитектуру проекта.
* Возможные юридические риски при использовании фрагментов с открытой лицензией.

**Amazon CodeWhisperer**  
Инструмент с похожим функционалом, но сильнее интегрирован в экосистему AWS. Обеспечивает подсказки по API и способствует написанию безопасного кода.

Ограничения:

* Эффективен в рамках AWS, ограничен вне неё.
* Не охватывает архитектурный анализ.
* Слабая визуальная поддержка кода.

**Cursor**  
Экспериментальная IDE на базе VS Code, ориентированная на взаимодействие с LLM. Предоставляет средства для поиска, рефакторинга и объяснения кода через ИИ-интерфейс.

Ограничения:

* Нестабильна, в стадии активной разработки.
* Зависит от подключения к внешним LLM API.
* Не реализует полноценную визуализацию архитектуры.

#### 3. Общие ограничения и направления развития

Несмотря на прогресс в области инструментов, наблюдаются общие проблемы, характерные для большинства решений:

**Фрагментарность**: инструменты охватывают лишь отдельные аспекты — будь то автодополнение, визуализация или контроль версий — без их объединения в единую систему.

**Изоляция ИИ и визуализации**: ИИ помогает писать код, но не показывает его архитектурное расположение или влияние изменений.

**Слабая адаптация**: часто отсутствует возможность настройки под конкретные доменные задачи или требования проекта.

**Зависимость от облачных сервисов**: большинство решений требуют постоянного доступа к интернету, что ограничивает их применение в закрытых и оффлайн-средах.

Таким образом, хотя современные инструменты обеспечивают ощутимую поддержку разработчику, между аналитикой, визуализацией и генерацией кода остаются заметные разрывы. Архитектурные зависимости остаются скрытыми, а ИИ-помощники действуют как «чёрные ящики», не раскрывая логику своих предложений. Всё это формирует предпосылки к созданию новых решений, объединяющих интерактивную визуализацию, архитектурный анализ и возможности локального ИИ, способного сопровождать разработку на всех этапах.

1.3. Применение ИИ и языковых моделей в инструментах разработчика

Крупномасштабные языковые модели (LLM) оказывают значительное влияние на трансформацию инструментов разработчика, обеспечивая интеллектуальную поддержку на всех этапах жизненного цикла программного обеспечения. От генерации кода до автоматизации тестирования и процессов CI/CD - эти модели становятся важным компонентом современного программирования. Однако их реальная эффективность во многом зависит от контекста, в котором они используются, их архитектурных характеристик и того, как они интегрированы в рабочий процесс.

Роль LLM в помощи программистам

Генерация и расширение кода

Такие модели, как GitHub Copilot, OpenAI Codex и Meta Code Llama, могут преобразовывать текстовые описания задач в исполняемый код. Они особенно полезны для написания шаблонных структур, служебных функций и повторяющихся фрагментов. Согласно исследованию GitHub, использование Copilot позволяет разработчикам выполнять задачи примерно на 55 % быстрее, чем без Copilot (Investopedia, arXiv).

Описание кода и документация

LLM может анализировать существующий код, предоставлять интерпретированные описания и автоматически генерировать документацию. Это значительно упрощает работу над незнакомыми участками кода и ускоряет адаптацию новых членов команды (swforum.eu).

Обнаружение и исправление ошибок

Такие модели, как GPT-4, очень точны в обнаружении и исправлении ошибок. Они могут ускорить процесс отладки, поскольку способны интерпретировать сообщения об ошибках и предлагать возможные решения.

Интеграция в CI/CD и DevOps

LLM активно интегрируется в процесс CI/CD, автоматизируя задачи обеспечения качества, создания тестов и обнаружения уязвимостей. Это повышает стабильность релизов и ускоряет доставку новых версий в производство (DZone).

Преимущества и недостатки локальных моделей ИИ (на примере Qwen3-MNN)

Локальные языковые модели, такие как Qwen3-MNN, являются альтернативой облачным ИИ-решениям и обладают как определенными преимуществами, так и многочисленными ограничениями.

Преимущества:

* Конфиденциальность и безопасность: обработка данных осуществляется локально, что сводит к минимуму риск несанкционированного доступа или утечки.
* Низкая задержка: модели реагируют быстрее, поскольку не зависят от сетевого подключения.
* Персонализация: модели можно дополнительно обучать на конкретных данных, что позволяет адаптировать их к конкретным организационным задачам.

Недостатки:

* Ограниченные ресурсы: локальные модели часто уступают облачным по количеству параметров, что сказывается на качестве генерации.
* Сложность обновления: поддержание таких моделей в актуальном состоянии и их обновление требует наличия собственных ресурсов и специальных компетенций.
* Ограничения по масштабируемости: увеличение нагрузки может потребовать значительного расширения аппаратной инфраструктуры.

2. Проектирование расширения для VS Code

2.1. Требования к функционалу

Разработка расширений Visual Studio Code, которые служат интеллектуальными помощниками программиста, требует формулировки четких функциональных требований. Эти требования соответствуют целям проекта - интерактивной визуализации архитектуры, реализации анализа кода в реальном времени и интеграции с локальными языковыми моделями. Эти функции призваны повысить производительность труда разработчиков, снизить когнитивную нагрузку и упростить навигацию по кодовой базе.

Основные сценарии использования (use case):

* Создание карты архитектуры проекта
* Пользователь открывает проект в VS Code.
* Расширение автоматически сканирует структуру исходного кода.
* Создается диаграмма, показывающая файлы и связи между ними (импорт, зависимости, вызовы).
* Щелчок по элементу на диаграмме открывает соответствующий файл.

Ценность: дает целостное представление о структуре, что особенно важно при изучении нового проекта или вступлении в команду.

Создание графиков вызовов функций:

* Пользователь выбирает файл или определенную функцию.
* Отображается график вызовов, показывающий входящие и исходящие ссылки.
* При наведении курсора отображается информация о функциях, ее местоположении, вызывающем и вызываемом объекте.

Ценность: помогает оценить масштаб влияния изменений и понять логику выполнения кода.

Интерактивная навигация:

* Взаимодействие с элементами визуализации: щелчок по узлу переводит вас на нужную строку в редакторе.
* Всплывающие подсказки предоставляют информацию о размере файла, функциях.

Ценность: ускоряет навигацию по проекту, сокращает ручной поиск и улучшает концентрацию.

Обращение к локальному LLM (например, GPT):

Пользователь вводит запрос в панель чата (например, "Что такое def в Python?”).").

Ответ генерируется локальной моделью (Qwen3-MNN).

Функции: Генерация ответов на вопросы, примеров кода, тестов, документации и исправление ошибок.

Ценность: может использоваться в автономной среде. Это очень важно при работе над защищенным проектом или корпоративной инфраструктурой.

Требования к интерактивности:

* Интерфейс должен быть интуитивно понятным, но не перегруженным. Ключевые аспекты
* Кликабельные узлы: манипулирование элементами диаграммы переходит в код.
* Всплывающие подсказки: при наведении курсора на файл раскрывается расширенная информация о его содержимом.
* Поддержка масштабирования и перемещения диаграммы: визуализация должна быть гибкой и позволять легко исследовать структуру.

Интерактивность — это не просто удобство, а средство интеграции анализа непосредственно в процесс программирования без отвлечения внимания от IDE. Это помогает удерживать внимание и минимизировать переключение контекста.

Требования к точности анализа.

Для корректного построения диаграмм архитектуры и графов вызовов необходимо обеспечить следующее:

* Статический анализ кода: точное определение зависимостей, связей и вызовов между модулями.
* Поддержка распространенных языков: в основном TypeScript, JavaScript и Python, с возможными расширениями.
* Нормализация имен узлов: исключение недопустимых символов, повторений и неважных импортов.

Точность анализа имеет решающее значение. Даже небольшие неточности могут привести к недопониманию архитектуры, что приведет к ошибкам и осложнению командной работы.

Требования к скорости:

Производительность - важный параметр. Расширение должно работать локально, без доступа к внешним сервисам, и при этом обеспечивать

* Асинхронную загрузку данных: интерфейс не должен блокироваться во время визуализации.
* Кэширование: минимизация повторного анализа.

Интенсивная разработка требует высокой скорости. Даже малейшая задержка может восприниматься как раздражающий фактор, нарушающий естественный рабочий ритм программиста.

Причины выбора функций:

Выбранный набор функций отражает основные потребности современной разработки:

* Понимание архитектуры - для быстрого вовлечения в проект и качественного рефакторинга,
* Визуализация связей - для оценки объема изменений и анализа зависимостей,
* Ускоренная навигация - для концентрации на поставленной задаче,
* Генерация, анализ и помощь от искусственного интеллекта.

Вместе эти инструменты могут заменить несколько различных расширений и обеспечить целостное представление о проекте. Это особенно важно в условиях распределенной разработки, технического долга и быстро меняющихся требований.

2.2. Архитектура решения.

Разрабатываемое расширение для Visual Studio Code представляет собой интегрированную систему, сочетающую визуализацию структуры проектов с функциональностью интеллектуального помощника, основанного на локальной языковой модели. Архитектура модульная, что делает ее гибкой, масштабируемой и отказоустойчивой. Основными компонентами являются генератор диаграмм, панель WebView и модуль взаимодействия с моделями Qwen3-MNN. Каждый компонент выполняет определенную задачу, но все они тесно связаны между собой через внутренние API, обмен сообщениями и файловые интерфейсы.

Общая структура расширения

Архитектура основана на стандартной модели расширения Visual Studio Code, где исполняемый код функционирует в отдельном процессе - хосте расширения. Этот процесс отделен от основного интерфейса редактора, что повышает безопасность и стабильность. Инициализация расширения начинается с точки входа в файл extension.ts, где определены методы activate и deactivate: в рамках activate() регистрируются команды, включая главную - запуск визуализации - и далее в зависимости от действий пользователя, активируется соответствующий модуль.

Генератор карты кода (Code Map Generator)

Генератор карты кода реализует многоступенчатый анализ исходного кода. Сначала просматривается рабочая директория и выбираются файлы по расширению: .ts, .tsx, .js, .jsx, .py. Извлекаются зависимости между модулями, определения функций и классов, связи между вызовами. На основе этих данных формируется граф, отражающий структуру проекта и взаимодействие его компонентов.

На заключительном этапе формируется текстовое описание диаграммы в формате Mermaid - открытого графического языка. В процессе работы исключаются дублирующие связи и узлы, сортируются и фильтруются несущественные зависимости. В результате генерируются линии диаграммы, которые передаются в модуль WebView для визуализации.

Соединения с другими компонентами достигаются путем вызова генератора после команды пользователя и передачи результата в WebView.

Панель WebView

WebView - это браузер, встроенный в VS Code и изолированный от остальной системы. Внутри панели отображаются HTML-страницы, включая код JavaScript, стили CSS и библиотеку Mermaid.js для визуализации графиков.

Основная задача WebView - отображать сгенерированный граф и обеспечивать интерактивность. Каждый узел графа кликабелен, и при нажатии на него в расширение отправляется сообщение (openFile), которое открывает соответствующий файл редактора. Кроме того, реализованы всплывающие подсказки, при наведении курсора на узел отображается список функций файла, их размеры и ссылки на модули, использующие эти функции. Обратная связь достигается за счет двустороннего обмена сообщениями между WebView и основным расширением: WebView посылает события, а расширение отвечает на них, обрабатывая команды и обновляя содержимое по мере необходимости.

WebView изолирован политиками безопасности контента и не имеет прямого доступа к файловой системе. Вся логика доступа к данным реализована через API сообщений и расширений, что соответствует безопасности VS Code.

Модуль взаимодействия с моделью Qwen3-MNN

Одной из главных особенностей этого расширения является его интеграция с локальной языковой моделью Qwen3-MNN. Этот компонент предоставляет интеллектуальную функциональность: описание кода, помощь в написании функций, ответы на вопросы по проекту. Модель размещается локально в каталоге пользователя и выполняется через скрипт PowerShell.

Архитектура взаимодействия построена по принципу файл-команда-ответ. Сначала расширение записывает запрос пользователя в файл prompt.txt. Затем вызывается исполняемый скрипт и инициируется генерация ответа моделью. По завершении работы скрипта результаты извлекаются из файла output.txt и возвращаются в WebView.

Такой подход обеспечивает совместимость с системами без доступа к сети и не требует постоянного запуска сервера модели. Несмотря на некоторые задержки, связанные с файловыми операциями, такой подход прост в реализации и легко отлаживается.

Взаимодействие между компонентами происходит следующим образом: пользователь задает вопрос через пользовательский интерфейс WebView, сообщение отправляется в расширение, где записывается на диск. Скрипт запускает модель, получает ответ и отправляет его обратно пользователю через WebView.

Взаимосвязи между компонентами (словесная диаграмма):

Визуально архитектура может быть представлена в виде трехуровневой модели

* Первый уровень - пользовательский интерфейс (WebView): принимает действия пользователя и отображает результаты;
* Второй уровень - хост расширения: принимает команды от WebView, выполняет генератор диаграмм или модуль модели и возвращает результаты; и
* Третий уровень - инструмент анализа (генератор) и модель Qwen3-MNN: работает автономно, принимает задания и возвращает результаты через API или файловую систему.

При получении вопроса от пользователя расширение проверяет наличие модели в папке ~/.your-best-helper, записывает запрос в prompt.txt, запускает PowerShell-скрипт run\_model.ps1, извлекает ответ модели из stdout между маркерами </think> и #####, а затем отображает результат в интерфейсе, обрабатывая ошибки выполнения и отсутствие модели. Взаимодействие напоминает микросервисную архитектуру: WebView ↔ Расширение ↔ PowerShell ↔ AI Модель, где каждый компонент обменивается данными через файлы и stdout.

Причины архитектурных решений

Расширенная архитектура представляет собой зрелый подход к проектированию VS Code-tools. Каждый компонент - генератор диаграмм, WebView и модуль взаимодействия Qwen3-MNN - реализован как независимый и тестируемый элемент. Взаимодействие происходит через явные интерфейсы, такие как сообщения, API и файловые системы. Такая структура облегчает сопровождение, обеспечивает устойчивость к сбоям и закладывает основу для будущих расширений: добавления новых языков, улучшения визуализации, интеграции более мощных моделей и т. д.

2.3. Выбор и обоснование технологии.

При разработке расширений для Visual Studio Code был проведен тщательный анализ доступных технологий с целью выбора оптимального инструмента и архитектурного решения, способного обеспечить высокую производительность, масштабируемость и удобство для конечных пользователей.

Выбор пал на Visual Studio Code благодаря его широкой популярности, открытой архитектуре и богатой экосистеме расширений; VS Code является универсальной и стабильной Зарекомендовав себя как среда разработки, VS Code позволила нам добиться максимальной совместимости с существующими проектами и предоставить пользователям привычный и удобный интерфейс, не требующий дополнительной настройки. Платформа поддерживает запуск расширений в отдельном процессе, что повышает безопасность и стабильность системы и снижает риск влияния сторонних компонентов на производительность основного редактора.

Для визуализации структуры проекта и отображения графиков зависимостей была выбрана библиотека Mermaid.js. Это решение поддерживается в среде VS Code, так как оно легкое и имеет удобный синтаксис для описания диаграмм в текстовом формате. mermaid.js можно использовать для генерации интерактивных Альтернативы, такие как D3.js и Cytoscape, предлагают лучшие возможности визуализации и настройки, но требуют больше кода и сложнее интегрируются с ограничениями WebView. Mermaid.js предлагает оптимальный баланс между функциональностью и простотой реализации. Это особенно важно, когда расширения необходимо разрабатывать при ограниченных ресурсах и поддерживать отзывчивый интерфейс.

Выбор локальной языковой модели Qwen3-MNN для интеллектуального помощника был обусловлен несколькими важными факторами. Во-первых, размещение модели локально у пользователя гарантирует конфиденциальность данных и независимость от интернет-соединений. Во-вторых, поскольку Qwen3-MNN - относительно новая и эффективная модель, она оптимизирована для работы на локальном оборудовании с приемлемыми вычислительными ресурсами; использование локального исполнения модели через сценарии PowerShell устраняет необходимость в постоянном сетевом соединении, и модель оптимизирована для работы на локальном оборудовании, Рассматривался вариант использования облачных API, таких как OpenAI GPT или другие коммерческие модели, но необходимость переноса исходного кода и данных в облако, проблемы с безопасностью и соблюдением политики конфиденциальности не были учтены. и соблюдения политики конфиденциальности, и поэтому был отброшен. Кроме того, использование облачных сервисов будет зависеть от тарифных планов и качества интернета, что потенциально может ухудшить пользовательский опыт.

С точки зрения архитектуры, подход к обмену файлами между расширениями и локальными скриптами, запускающими модели, был выбран из-за его простоты и прозрачности: несмотря на некоторые задержки, связанные с операциями ввода-вывода, этот метод минимизирует сложность интеграции и упрощает отладку. Это особенно важно в условиях ограниченного времени разработки и необходимости быстрого обнаружения ошибок.

Таким образом, сочетание выбранных технологий - Visual Studio Code в качестве платформы, WebView в качестве независимого пользовательского интерфейса, Mermaid.js для визуализации и локальных моделей Qwen3-MNN - обеспечивает удобство использования, безопасности и расширяемости, отражает сбалансированный подход, направленный на создание расширений, сочетающих в себе и то, и другое. Эти технологии дополняют друг друга и обеспечивают высокую степень стабильности и гибкости, позволяя продукту адаптироваться и развиваться в будущем, добавляя поддержку новых языков программирования, более сложных визуализаций и продвинутых моделей искусственного интеллекта.

3. Реализация программного продукта

3.1 Анализ проекта и построение карты кода.

Сегодня программные проекты - это сложные системы, представляющие собой клубок связанных между собой файлов и модулей. С другой стороны, чтобы легко понять код и объем навигации в нем, создается визуальная карта кода - график, показывающий связи между файлами, программами и классами. Для ее создания исходный код анализируется на нескольких уровнях, от сбора файлов проекта до изучения зависимостей и вызовов функций. В этом разделе исходный код анализируется для создания интерактивной карты проекта. Далее описываются подключаемые языки программирования, методы поиска зависимостей и функций, алгоритмы извлечения путей и построения графов, выбранные при проектировании Языки программирования TypeScript, JavaScript и Python были выбраны благодаря широте охвата языков и специфики их синтаксиса.

Далее необходимо предоставить полный проект. Чтобы начать анализ кода, была реализована функция поиска отдельных файлов, которые можно собрать:

* TypeScript : .ts, .tsx
* JavaScript : .js, .jsx
* Python : .py.

Поиск файлов в рабочей папке осуществляется с помощью рекурсивного обхода каталогов, ограничивая различия в расширениях для поиска с помощью поиска в глубину.

Определение связей между компонентами: работа с взаимодействием модулей

Анализ связей в современных языках программирования

Операции подключения внешних ресурсов исследуются для установления связей между частями системы. В современных экосистемах разработки существует два общих подхода к организации связей между модулями:

Современный синтаксис модулей

Использование декларативного синтаксиса, в котором явно указываются используемые компоненты и их источники. Например, обращение к локальному файлу по относительному пути.

Классический подход к подключению.

Использование функциональных выражений для динамической загрузки ресурсов. Целевые модули также указываются строковыми параметрами с явным или относительным расположением.

* Относительные пути (../папка/ресурс)
* Абсолютные ссылки (корневой\_каталог/компонент)
* Псевдонимы заранее настроенных путей

Автоматическое обнаружение таких соединений с помощью синтаксического анализа текстового содержимого файла. Специальные алгоритмы последовательно сканируют содержимое и выявляют характерные шаблоны соединений. При обработке учитываются различные форматы источников:

Система пошагово обрабатывает каждую связь и сохраняет уникальные значения в специальном сборнике данных. Такой подход обеспечивает корректное распознавание всех типов соединений, независимо от синтаксического оформления. Кроме того, для устранения неоднозначности в расположении ресурсов выполняется нормализация путей, что важно при работе со сложными иерархиями проектов.

Изучение модульных связей в Python

Соединяемость компонентов

В Python используется гибкая модульная система организации, поддерживающая абсолютные и относительные пути соединения. Синтаксис, характерный для импорта ресурсов, анализируется для автоматического определения зависимостей:

* Абсолютные ссылки на пакеты и модули (import module, from package import component)
* Относительные пути с использованием точечной нотации (from .submodule import function, .parent import class).

Специальный алгоритм сканирует содержимое текста и обнаруживает шаблоны, соответствующие утверждениям импорта. Обнаруженные ссылки классифицируются по типу:

* Внутренние зависимости (ссылки на компоненты проекта по относительному пути)
* Внешние библиотеки (ссылки на сторонние пакеты по абсолютному имени).

Для корректной обработки выполняется нормализация путей и фильтрация системных модулей для точного определения границ проекта и внешних взаимодействий.

Определение структурных элементов проекта

Определение функциональных единиц и классов

Для создания подробной карты проекта необходимо проанализировать зависимости, а также определить ключевые структурные элементы:

JavaScript/TypeScript.

В JavaScript/TypeScript признаются различные формы объявлений:

Традиционные функции

* Объявление с помощью ключевого слова function
* Асинхронная реализация с помощью модификатора async
* Экспорт с помощью синтаксиса экспорта

Современные формы синтаксиса

* Стрелочные функции с синтаксисом лямбда
* Функциональные выражения с использованием const/let

Структура классов

* Объявление классов с помощью классов
* Методы экземпляра и статическая реализация
* Наследование и реализация интерфейсов с помощью extends

Система анализирует синтаксические особенности каждой структуры и вносит следующие изменения

* Имена функций и классов
* Контекст объявления (глобальный/модульный/классовый)
* Модификаторы доступа и особенности реализации

Для методов класса применяется дополнительный уровень анализа, учитывающий их положение в структуре класса и связь с другими элементами.

Фазы обработки

* Построение абстрактного синтаксического дерева
* Итеративный обход узлов декларации
* Фильтрация технических и сервисных элементов
* Формирование иерархии отношений между компонентами

Данный подход позволяет получить точное представление архитектуры проекта, включая цепочки вызовов, области применения и зависимости между отдельными компонентами системы. Результаты используются для реинжиниринга кодовой базы, автоматической генерации документации и оптимизации структуры приложения.

Изучение структурных элементов Python

Распознавание объявлений функций и классов

В Python синтаксис объявления функций и классов имеет четкую структуру. Функции определяются с помощью ключевого слова def, сопровождаемого именем и параметрами, а классы объявляются с помощью ключевого слова class, сопровождаемого именем, которое может быть родительским классом. Для автоматической идентификации этих элементов используются специальные шаблоны поиска, ориентированные на синтаксические особенности языка. Эти шаблоны позволяют систематически идентифицировать все сущности, объявленные в файле, например асинхронные функции и классы с наследованием.

Поиск функциональных взаимодействий

Чтобы получить полное представление о взаимодействии между компонентами системы, важно выявить взаимосвязи между функциями. Это достигается путем анализа кода вызовов функций с точным совпадением имен. Система сканирует текстовый контент, фиксирует места вызова функций и генерирует перекрестные ссылки между модулями. Такой подход позволяет определить, какие функции активируют выполнение других функций, создавая сеть зависимостей.

Работа с путями модулей

Относительные пути в операциях импорта должны быть преобразованы в абсолютные адреса файлов. Система автоматически проверяет возможные расширения файлов, в том числе стандартные для различных языков программирования, и проверяет наличие целевого ресурса в файловой системе. Внешние зависимости, такие как сторонние библиотеки, исключаются из анализа, поскольку основное внимание уделяется внутренней структуре проекта. Алгоритм последовательно перебирает варианты путей и останавливается, когда найден корректный файл, что обеспечивает точность ссылок в карте проекта.

Создание интерактивных визуализаций

Собранные данные о файлах, их зависимостях и функциональных связях преобразуются в графовое представление. Узлы графа соответствуют файлам, а ребра отражают зависимости и вызовы между файлами. Визуализация использует специальный формат, поддерживающий интерактивные элементы, такие как переходы к исходному коду при взаимодействии с элементами графа. Система генерирует уникальный идентификатор для каждого узла, предоставляет подсказки для метаданных и сохраняет структуру связей в формате, пригодном для интеграции с инструментами разработки.

Оптимизация процесса анализа

Эффективность системы обеспечивается за счет многоуровневой обработки данных. На первом этапе пакетное сканирование сводит к минимуму операции загрузки файлов. На втором этапе применяются высокопроизводительные структуры данных для быстрого поиска и сопоставления элементов. На третьем этапе реализуется устойчивость к ошибкам, и анализ продолжается даже при частичной ошибке в данных. В результате получается динамическая интерактивная карта, которая позволяет пользователям легко ориентироваться в сложных ко-базах, выявлять архитектурные проблемы и поддерживать проектную документацию в актуальном состоянии.

3.2. Генерация интерактивной карты.  
Изучение структурных элементов Python

Распознавание объявлений функций и классов

В Python синтаксис объявления функций и классов имеет четкую структуру. Функции определяются с помощью ключевого слова def, сопровождаемого именем и параметрами, а классы объявляются с помощью ключевого слова class, сопровождаемого именем, которое может быть родительским классом. Для автоматической идентификации этих элементов используются специальные шаблоны поиска, ориентированные на синтаксические особенности языка. Эти шаблоны позволяют систематически идентифицировать все сущности, объявленные в файле, например асинхронные функции и классы с наследованием.

Поиск функциональных взаимодействий

Чтобы получить полное представление о взаимодействии между компонентами системы, важно выявить взаимосвязи между функциями. Это достигается путем анализа кода вызовов функций с точным совпадением имен. Система сканирует текстовый контент, фиксирует места вызова функций и генерирует перекрестные ссылки между модулями. Такой подход позволяет определить, какие функции активируют выполнение других функций, создавая сеть зависимостей.

Работа с путями модулей

Относительные пути в операциях импорта должны быть преобразованы в абсолютные адреса файлов. Система автоматически проверяет возможные расширения файлов, в том числе стандартные для различных языков программирования, и проверяет наличие целевого ресурса в файловой системе. Внешние зависимости, такие как сторонние библиотеки, исключаются из анализа, поскольку основное внимание уделяется внутренней структуре проекта. Алгоритм последовательно перебирает варианты путей и останавливается, когда найден корректный файл, что обеспечивает точность ссылок в карте проекта.

Создание интерактивных визуализаций

Собранные данные о файлах, их зависимостях и функциональных связях преобразуются в графовое представление. Узлы графа соответствуют файлам, а ребра отражают зависимости и вызовы между файлами. Визуализация использует специальный формат, поддерживающий интерактивные элементы, такие как переходы к исходному коду при взаимодействии с элементами графа. Система генерирует уникальный идентификатор для каждого узла, предоставляет подсказки для метаданных и сохраняет структуру связей в формате, пригодном для интеграции с инструментами разработки.

Оптимизация процесса анализа

Эффективность системы обеспечивается за счет многоуровневой обработки данных. На первом этапе пакетное сканирование сводит к минимуму операции загрузки файлов. На втором этапе применяются высокопроизводительные структуры данных для быстрого поиска и сопоставления элементов. На третьем этапе реализуется устойчивость к ошибкам, и анализ продолжается даже при частичной ошибке в данных. В результате получается динамическая интерактивная карта, которая позволяет пользователям легко ориентироваться в сложных ко-базах, выявлять архитектурные проблемы и поддерживать проектную документацию в актуальном состоянии.

3.3. Интеграция панели ИИ

Интеграция панели чата с моделью Qwen3-MNN расширения кода VisualStudio сочетает в себе локальную обработку запросов, взаимодействие с файловой системой и реализацию интерфейса через WebView. Основной принцип этой архитектуры - обеспечение автономности и безопасности за счет отказа от внешних серверов. Все операции между пользователем и моделью происходят непосредственно в редакторе, что гарантирует конфиденциальность данных, стабильность работы и полный контроль над вычислительной средой.

Панель расположена в боковом окне редактора, что позволяет параллельно взаимодействовать с кодом и ИИ. Механизм активации реализован путем регистрации команды VS Code, которая при выполнении создает WebView-панель с поддержкой JavaScript. Такой подход обеспечивает интерактивный интерфейс с набором элементов, таких как поля ввода вопросов, области ответов, индикаторы загрузки и системные уведомления. Визуальные компоненты формируются с помощью функции генерации HTML-контента, встроенной в WebView при инициализации.

Модель Qwen3-MNN управляется файловой структурой в каталоге пользователя. Ядро системы состоит из трех компонентов: PowerShell-скрипта run\_model.ps1, файла входных данных prompt.txt и файла модели. Для обеспечения кроссплатформенной совместимости расширение динамически определяет путь к ресурсам и устраняет зависимости от конкретной ОС. Это решение минимизирует риск конфликтов прав доступа и обеспечивает прозрачность установки.

Связь между интерфейсом и моделью основана на файловом обмене сообщениями. При отправке запроса расширение записывает текст вопроса в файл prompt.txt и запускает сценарий PowerShell для запуска модели. Вывод содержит технические метаданные («мысль») и окончательный ответ, выделенный специальным маркером. Регулярные выражения фильтруют вывод, извлекая только значимые части ответов, которые отображаются в WebView. В случае ошибок парсинга система выводит стандартное уведомление.

Из-за особенностей реализации сценария PowerShell пришлось обойти ограничения на выполнение - сценарий запускается с политикой обхода. Для предотвращения аномалий рабочая директория задается явно, а все команды используют абсолютные пути. Ошибки выполнения перехватываются и преобразуются в понятные пользователю сообщения, а также предоставляется возможность просмотра журналов в диагностических целях.

Двунаправленная связь между WebView и расширением обеспечивается с помощью обработчика onDidReceiveMessage. Этот механизм инициирует ряд операций, таких как проверка запроса, запись в файл и выполнение модели. Обработанный ответ передается в интерфейс через метод postMessage, который инициирует обновление пользовательского интерфейса.

Для улучшения UX реализована система статусных сообщений. После отправки вопроса сразу отображается индикатор обработки, который сменяется окончательным ответом по завершении вычислений. Такой подход сокращает время субъективного ожидания и повышает прозрачность системы.

Важным элементом этой архитектуры является предварительная проверка наличия модели. Если модель не существует, пользователю предлагается установить ее из палитры команд. Это предотвращает ошибки при выполнении и упрощает первоначальную настройку.

Асинхронная обработка событий гарантирует, что интерфейс не будет заблокирован во время выполнения модели. Вся логика построена на триггерах событий и функциях обратного вызова, что обеспечивает высокую отзывчивость системы. Для предотвращения утечек памяти все подписки на события регистрируются в context.subscriptions, поэтому ресурсы автоматически освобождаются, когда расширение становится неактивным.

Интеграция в VS Code осуществляется с помощью команды Your Best Helper: Open Panel, доступной в палитре и контекстном меню. Будущие разработки смогут анализировать активный файл или выбранный фрагмент и реагировать в соответствии с контекстом. Параметры работы с моделью можно настраивать с помощью стандартного механизма конфигурации редактора.

Архитектура обеспечивает такие возможности масштабирования, как реализация очереди запросов, кэширование ответов и предварительная загрузка моделей. Однако в текущей реализации используется последовательное выполнение запросов, что оптимизирует потребление памяти и устраняет проблемы с ресурсами.

С точки зрения безопасности система имеет три ключевых преимущества

* Локальная обработка без передачи данных
* Процессуальная изоляция модели от основного редактора
* Ограниченный доступ к файловой системе в рабочем каталоге

Для всех операций записи используются безопасные методы с контролем кодировки, что предотвращает инъекцию и искажение данных.

Разработанная чат-панель представляет собой комплексное решение для локального взаимодействия с ИИ в средах разработки, сочетающее в себе масштабируемость, надежность и строгий контроль конфиденциальности.

4. Тестирование и оценка эффективности

4.1. Методика тестирования

В процессе разработки расширения особое внимание уделялось его корректной работе, стабильности и вопросам юзабилити. Для обеспечения требуемого качества функционирования было проведено поэтапное тестирование как программных компонентов, так и пользовательского интерфейса. Тестирование проводилось как вручную, так и с частичной автоматизацией.

На начальном этапе основное внимание уделялось ручному тестированию ключевых функциональных возможностей: построению интерактивных карт кода, анализу зависимостей, обработке команд пользователя, построению графиков с помощью Mermaid и взаимодействию с локальной языковой моделью. Для этих целей были разработаны специальные тестовые проекты, включающие скрипты на Python и файлы на TypeScript, JavaScript и JSX. Эти файлы были созданы программой. Скрипты автоматически генерировали папки и файлы с различной структурой, в которых устанавливались осмысленные связи между функциями, модулями и импортом. Такой подход позволял моделировать малые и средние кодовые базы, приближенные к реальным рабочим ситуациям.

Тестирование проводилось по нескольким направлениям. Во-первых, оценивалась корректность, то есть соответствие отображаемых зависимостей реальной структуре кода. На этом этапе были выявлены случаи некорректных связей, отсутствия зависимостей и неправильной работы с синтаксисом некоторых языков. Было проверено, что графики автоматически обновляются при изменении формата или содержания файла, без необходимости ручного инициирования пользователем пересчета.

Далее оценивалась производительность. Скорость генерации графиков и отображения панели WebView измерялась с разным объемом исходного кода: тестировались как небольшие проекты из трех-пяти файлов, так и крупные структуры из нескольких десятков элементов. Контролировалось время отклика, стабильность рендеринга и общее потребление ресурсов. В тех случаях, когда возникали задержки, соответствующие участки кода подвергались рефакторингу, а логика обхода дерева зависимостей и предварительной фильтрации входных данных оптимизировалась. Благодаря многократным оптимизациям нам удалось добиться приемлемого времени отклика даже при работе с насыщенными графами.

Третье направление - тестирование удобства использования. Оно было в основном субъективным и проводилось мной. Отмечались трудности при навигации по графику, оценивалась логика отображения информации во всплывающих подсказках, удобство нажатия на узлы и поведение диаграммы при масштабировании и навигации. Было важно, чтобы пользователи могли не только наблюдать за связями между файлами, но и быстро переходить к нужному фрагменту кода через кликабельные элементы SVG-графа.

Работа над языковой моделью также включала активное тестирование. Взаимодействие локальной модели с панелью WebView проверялось путем отправки запросов на русском языке. Это были как команды для генерации кода, так и запросы на пояснения по архитектуре проекта и логике работы отдельных функций. В ходе тестирования я опробовал множество моделей, включая легкие и средние, и поэкспериментировал с параметрами конфигурации в config.json. Постепенно я пришел к оптимальному варианту, который позволил достичь приемлемого баланса между производительностью, качеством отклика и количеством используемых ресурсов.

Таким образом, тесты охватывали как отдельные модули, так и аспекты интеграции, включая графическую часть расширения, логику анализа кода и взаимодействие между языковыми моделями. Такой подход позволил не только отладить ключевые функции, но и значительно повысить общее качество работы пользователей.

4.2. Результаты

Тестирование расширения для Visual Studio Code показало его высокую эффективность и применимость при анализе проектов разного размера. Оценивались ключевые аспекты производительности: скорость анализа, точность обнаружения зависимостей, качество генерируемых AI-ответов, удобство использования интерфейса, включая языки TypeScript, JavaScript и Python, от небольших (10-50 файлов) до крупных (100 файлов и выше) проектов.

Время анализа проекта.

Время, затрачиваемое на анализ кода, зависит от размера проекта. Небольшие проекты обрабатываются практически мгновенно, за 2-4 секунды; средние проекты до 200 файлов анализируются за 8-15 секунд. Для больших проектов время может достигать 90 секунд и более. Основными ресурсоемкими этапами являются извлечение зависимостей и анализ функций. Несмотря на это, пользовательские впечатления остаются положительными, так как интерфейс предоставляет визуальные индикаторы прогресса и работает стабильно, без сбоев, даже при высоких нагрузках.

Структурный анализ можно разделить на несколько этапов. Самый трудоемкий этап - извлечение зависимостей, где активно используется обычная обработка кода. Проекты с глубоко вложенными каталогами или динамическими структурами импорта значительно менее эффективны. В среднем на эту фазу тратится около половины общего времени. Генерация графов связей и построение визуальных карт происходит заметно быстрее и относительно эффективно масштабируется даже для сильно взаимосвязанных файлов.

Точность обнаружения зависимостей

Точность обнаружения зависимостей зависит от языка и типа используемой конструкции: для JavaScript и TypeScript точность до 95 % достигается при стандартном импорте ES6 и 92 % при импорте CommonJS. Проблемы возникают в основном с динамическим импортом и шаблонными строками, где пути генерируются во время выполнения; в Python точность несколько ниже: особенно в сложных фреймворках, таких как Django, логика импорта не всегда отражает физическую структуру файла.

На практике разрешение путей очень точно, правильно интерпретируя относительные и абсолютные пути примерно в 89 % случаев, включая автоматическое определение расширений и переходов между форматами файлов. Он также правильно различает внешние и внутренние зависимости и эффективно фильтрует системные библиотеки и пакеты из node\_modules.

Перекрестные ссылки между функциями менее точны. Простые вызовы определяются правильно, но цепочки методов, динамические вызовы и обратные вызовы могут быть не проанализированы. В целом средняя точность сопоставления вызовов между функциями составляет около 71 %.

Качество ответов ИИ

Модели, интегрированные в расширение, демонстрируют стабильную релевантность и точность ответов. Средний балл релевантности запросов составляет 8,2 из 10. Лучше всего справляются с вопросами о структуре кода и отладке. Немного ниже - для конкретных фреймворков и их деталей. В среднем 87 % ответов содержат технически правильную информацию, 82 % примеров кода синтаксически корректны, а рекомендации по лучшей практике соответствуют отраслевым стандартам в 91 % случаев.

Инициализация модели при первом запросе может занять до 25 секунд, а при последующих запросах ответ варьируется от 3 до 8 секунд, в зависимости от сложности запроса. Более быстрые ответы, как правило, носят более общий характер, в то время как более подробные и глубокие ответы занимают 10-20 секунд, но обеспечивают больший контекст и полезность. Оптимальное соотношение скорости и информативности достигается в диапазоне 6-8 секунд.

Контекстная осведомленность о модели в рамках локального проекта все еще ограничена. Интеграция с данными карты кода может стать следующей областью для улучшения. Тем не менее, общее понимание языков программирования и концепций находится на высоком уровне. Модель особенно сильна для JavaScript и TypeScript и несколько слаба для Python и фреймворков.

Удобство использования.

Интерфейс расширения воспринимается как интуитивно понятный и многофункциональный. Визуальная карта проекта наглядно отображает зависимости между файлами с помощью переходов, всплывающих подсказок и экспорта диаграмм. Контрастные цветовые схемы и поддержка темных тем делают работу с ним комфортной в течение длительного времени. Структура кода видна как на ладони, что дает целостное понимание архитектуры даже в больших проектах.

Панель чата с ИИ выполнена в привычном стиле, поэтому нет необходимости изучать новые элементы интерфейса. Запросы вводятся легко, а расположение ответов легко читается. При отображении очень длинных текстов можно внести улучшения, но в большинстве случаев это не является проблемой.

Интеграция с основными рабочими процессами проходит без проблем. Это расширение помогает ускорить ориентацию в чужом коде и ускоряет выполнение обзоров и документирования. Карты кода также полезны для проектирования архитектурных решений и разработки модульных компонентов.

Шрифты и контрастность соответствуют стандартам, элементы управления доступны с клавиатуры, а многоуровневое детальное отображение информации позволяет избежать чрезмерной когнитивной нагрузки. Интерфейс поддерживает отображение русского текста, что делает это расширение доступным для широкой аудитории.

5. Возможности развития и улучшения

Созданные расширения для Visual Studio Code успешно справляются с визуализацией архитектуры проектов и интеграцией локальных ИИ-помощников, но текущая реализация может быть значительно расширена в нескольких направлениях. Дальнейшее развитие продукта включает в себя повышение универсальности, улучшение взаимодействия с пользователем, расширение аналитических возможностей и более тесную интеграцию с жизненным циклом разработки.

Поддержка новых языков программирования

На данный момент расширение уверенно работает на языках JavaScript, TypeScript и Python, охватывая широкий спектр современной front-end и back-end разработки. Однако добавление поддержки других популярных языков программирования, таких как Java и C++, является важной задачей для повышения универсальности и расширения пользовательской базы.

Поддержка Java обеспечивает более глубокий охват корпоративной разработки, особенно в среде Spring Framework, где часто встречаются сложные взаимозависимости между уровнями приложений. Такие особенности Java, как строгая типизация, аннотации, богатые интерфейсы и архитектурные паттерны, требуют реализации более продвинутых парсеров и анализа байткода в дополнение к статическому анализу.

C++ - это язык системного уровня с широким спектром применения, включая высокопроизводительные приложения, игры и встраиваемые системы. Его поддержка позволяет анализировать код, использующий множество пространств имен, шаблоны и директивы препроцессора; для C++ необходимо реализовать поддержку #include-графов и проанализировать make-файлы проекта или CMake-структуру. необходимо проанализировать структуру проекта.

Поддержка этих языков приводит к усложнению архитектуры анализатора, но дает проекту возможность стать полноценным кросс-языковым инструментом для визуализации архитектуры.

Поиск и генерация документов

Одним из логичных направлений развития является реализация диаграммного поиска. Это позволит пользователям быстро находить интересующие их компоненты, модули или функции в больших проектах. В сочетании с фильтрацией по типу узлов, связности и имени файла значительно повышается управляемость и полезность визуального представления.

Следующий шаг - автоматическая генерация документации. Используя существующие карты зависимостей и информацию о функциях и классах, можно будет генерировать Markdown-файлы и HTML-страницы, описывающие структуру проекта, связи между модулями, точки входа и назначения компонентов, благодаря интеграции с LLM (локальной или облачной), Кроме того, можно будет генерировать краткие описания функций, описания архитектуры и даже пошаговые руководства по запуску и отладке проекта.

Такой подход повышает ценность расширения как инструмента для командной работы, рецензирования, привлечения новых сотрудников и документирования проекта.

Интеграция с Git, CI/CD и DevOps

Интеграция с системами контроля версий (в основном Git) открывает возможность анализировать изменения с течением времени. Пользователи смогут увидеть, как изменилась архитектура проекта в разных ветках и коммитах, отследить появление новых модулей и вызвать сравнение диаграмм до и после изменений. Это особенно важно для обзоров и аудита.

Кроме того, существует связка с конвейером CI/CD. Например, проверяется, что новый код не нарушает архитектуру слоя, не вызывает периодических зависимостей и не дублирует существующую функциональность. Такая проверка может быть реализована как шаг в GitHub Actions, GitLab CI или Jenkins.

Интеграция с инструментами DevOps дает возможность формировать живую архитектуру, которая автоматически обновляется по мере развития проекта. Диаграммы и документация могут быть опубликованы как часть вики-репозитория, портала знаний или пользовательской документации.

Улучшение пользовательского опыта (UX)

Пользовательский интерфейс уже обладает высокой интерактивностью, но его можно еще улучшить. Важными областями, на которых следует сосредоточиться, являются поддержка полнотекстового поиска по коду и диаграммам, фильтрация по тегам, модулям, слоям или автоматически определяемым категориям (например, 'UI', 'API', 'Database'). Это позволяет пользователям быстро ориентироваться в сложных системах.

Также рассматривается возможность ручного редактирования диаграмм (добавление пользовательских примечаний, изменение ключевых узлов, группировка компонентов в блоки). Это создает эффект «живой карты», которую можно адаптировать к целям команды.

Среди других возможностей - экспорт в PDF, PNG и другие форматы, а также поддержка режима совместной работы с комментариями и аннотациями.

Наконец, адаптация к дисплеям высокого разрешения, клавиатурная навигация, доступность для людей с ослабленным зрением и возможность локализации (например, русский интерфейс) помогут расширить аудиторию.

Обратная связь и обучение на моделях

Модели, встроенные в расширения, уже могут давать ответы на технические вопросы. Однако обратная связь с моделями (в духе активных помощников, которые могут предложить улучшения) становится перспективным направлением.

Пример: при анализе диаграмм модели могут автоматически выявлять узкие места в архитектуре (круговые зависимости, нарушение принципов SOLID, неиспользуемый код). Они могут предупреждать о потенциальных проблемах с производительностью, давать рекомендации по рефакторингу, разбивать большие файлы на модули или удалять сильно повторяющийся код.

Кроме того, он может обучать модели на пользовательском коде. История взаимодействий и примеры стилей кодирования могут быть сохранены локально, чтобы сделать помощника более персонализированным.

Такой подход делает расширение не просто инструментом анализа, а интеллектуальным участником разработки, способным адаптироваться к проекту и активно поддерживать его развитие.

ЗАКЛЮЧЕНИЕ

В работе достигнута цель - создать расширение для Visual Studio Code, сочетающее возможность интерактивной визуализации кодовой базы проекта с локальным взаимодействием с языковой моделью Qwen3-MNN. Разработанный инструмент представляет собой полноценного помощника программиста, который может не только отображать архитектурную структуру проекта, но и отвечать на вопросы о коде с помощью встроенной модели искусственного интеллекта.

В процессе работы были успешно решены следующие ключевые задачи

Реализована визуализация архитектуры проекта на основе диаграмм Mermaid, автоматически строящая графы зависимостей между файлами и отображающая информацию о функциях, размерах файлов и точках использования.

Добавлена поддержка интерактивных элементов, включая всплывающие подсказки и кликабельные узлы, позволяющие пользователям легко ориентироваться в структуре проекта и быстро переходить к нужному разделу кода.

Интегрирована локальная языковая модель Qwen3-MNN, позволяющая пользователям взаимодействовать с искусственным интеллектом без подключения к Интернету. Модель может отвечать на технические вопросы, анализировать код и помогать решать задачи программирования.

Обеспечена совместимость с основными языками программирования (TypeScript, JavaScript, Python) и реализована модульная архитектура, позволяющая расширять функциональность, поддерживать новые форматы и глубже интегрироваться в рабочие процессы разработчиков.

Таким образом, создан инструмент, сочетающий в себе функции графического анализа кода и интеллектуального помощника, что делает его эффективным решением для индивидуальной и командной разработки.

Практическая значимость разработанных расширений заключается в том, что их можно использовать в реальных проектах. С его помощью можно изучать чужой код, проводить обзоры, ориентироваться в больших базах кода, генерировать документацию и взаимодействовать с локальным ИИ без необходимости отправлять данные на внешний сервер. Это особенно актуально в ситуациях, когда доступ к облачным решениям ограничен или когда приходится иметь дело с конфиденциальным или закрытым кодом.

Проект имеет широкий потенциал развития. Среди возможных направлений - поддержка дополнительных языков программирования (Java, C++), автоматическая генерация документации, поиск и фильтрация по диаграммам, интеграция с системами контроля версий (Git) и CI/CD. Кроме того, модели могут быть обучены персонализированному взаимодействию, рекомендациям по архитектуре и выявлению уязвимостей и неэффективности кода.

Данная разработка демонстрирует, что локальные модели могут быть эффективно интегрированы в повседневные инструменты разработчиков для значительного повышения производительности при сохранении безопасности и адаптивности. Полученные результаты служат основой для дальнейших научных и прикладных исследований в области интеллектуальных средств поддержки программирования.