УДК 004.89

ПОДХОД К СОЗДАНИЮ СЕРВИСА ГЕНЕРАЦИИ ПРОГРАММНОГО КОДА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ

Резуник Л., Александров Д.В.

Национальный исследовательский университет "Высшая школа экономики" (НИУ ВШЭ) 109028, Москва, Покровский бульвар, д. 11

Технологии машинного обучения и различные инструменты для генерации кода в последние годы оказали значительное влияние на сферу разработки программного обеспечения. Хотя большинство существующих решений не созданы специально для генерации кода, программисты применяют их в различных задачах. Не многие из существующих решений для искусственного интеллекта хорошо работают с менее распространенными языками, такими как Kotlin или Swift, которые используются в мобильной разработке. Поэтому существующие большие языковые модели редко адаптируются в стороннем программном обеспечении для мобильных разработчиков, хотя это принесло бы пользу отрасли. Целью данной работы является создание сервиса, который использовал бы большую языковую модель для предоставления пользователям, разработчикам мобильных устройств, инструмента для эффективного программирования на вышеупомянутых языках. Разработанный сервис использует уже готовую языковую модель, которая дорабатывается на основе данных, доступных онлайн в репозиториях с открытым исходным кодом и собранных вручную. Разработанное программное обеспечение может выполнять различные задачи программирования, характерные для области мобильной разработки: написание кода для макетов экранов, компонентов пользовательского интерфейса, бизнес-логики и модульных тестов. Программное обеспечение также оценивается с помощью набора тестов HumanEval и его вариаций, а также предложенных авторами тестов, которые дают представление о качестве сгенерированного кода. Данная статья является результатом исследовательского проекта, реализуемого в рамках программы фундаментальных исследований Национального исследовательского университета "Высшая школа экономики" (НИУ ВШЭ).

Ключевые слова: генерация кода, мобильная разработка, большие языковые модели, тонкая настройка, веб-сервис.

APPROACH TO CREATING THE SERVICE FOR CODE GENERATION OF MOBILE APPLICATIONS USING THE LARGE LANGUAGE MODELS

Rezunik L., Alexandrov D.V.

National Research University Higher School of Economics (HSE University) 11 Pokrovsky Boulevard, Moscow, 109028

Machine learning technologies and various tools for code generation have had a significant impact on the field of software development in recent years. Although most of the existing solutions are not built exactly for code generation, programmers apply them in different tasks. Not many of the existing AI solutions work well with less common languages, such as Kotlin or Swift, that are used in mobile development. Therefore, existing large language models are rarely adapted in the third-party software for mobile developers, although it would benefit the industry. The goal of this work is to develop a service that would use a large language model to provide the users, mobile developers, with a tool for efficient programming in the aforementioned programming languages. The developed service utilizes an already existing language model, which is fine-tuned based on the data available online in open-source repositories and collected manually. The developed software can perform various programming tasks specific to the mobile development domain: writing code for screen layouts, UI (User Interface) components, business logic, and unit tests. The software is also evaluated against the HumanEval benchmark and its variations as well as a custom benchmark that gives an understanding of the quality of generated code. This article is the result of a research project implemented within the framework of the fundamental research program of the National Research University Higher School of Economics (HSE University).

Key Words: code generation, mobile development, large language models, fine-tuning, web-service.

1 Введение

1.1 Предпосылки исследования

LLM (Large Language Model, большая языковая модель) – это тип модели машинного обучения, которая обучается на огромном объеме текстовых данных. LLM способны "понимать" и генерировать текст, похожий на тот, что мог бы написать человек. Применять LLM в программной инженерии – не новая концепция, однако она становится все более популярной благодаря совершенствованию технологий и появлению успешных программных продуктов на основе ИИ. Такие продукты обычно направлены на то, чтобы сделать процесс разработки проще и эффективнее, так как могут применяться для решения различных задач: обзора кода, рефакторинга, создания документации и генерации кода. Генерация кода – это процесс автоматического создания исходного кода на основе описания или спецификации, что упрощает разработку программного обеспечения, избавляя от необходимости писать код вручную. Это может быть реализовано в виде «помощника», общающегося с разработчиком (например, как в инструменте GitHub Copilot [3]), призванного оптимизировать процесс разработки, автоматизируя повторяющиеся задачи, предлагая дополнения к коду и генерируя целые блоки кода [4].

У задачи генерации кода есть две разновидности: генерация «из кода в код» и «из описания в код» [7]. Первый подход, как следует из названия, ориентирован на генерацию программного кода из фрагмента другого программного кода (рис. 1). Он может быть использован для автозаполнения или предложения возможных для использования фрагментов кода. Таким образом, например, реализуется функциональность подсказок в GitHub Copilot [3].

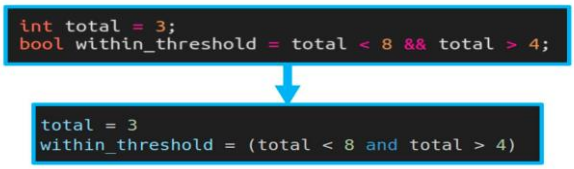

Рис. 1 – Пример генерации «из кода в код» [7]

Задача «из описания в код» – это задача, основанная на получении описания на естественном языке, также называемым prompt, и генерации программного кода на его основе. Prompt – это разновидность формата ввода текста, который используется для управления поведением модели ИИ, предоставляя модели контекст и инструкции о том, как она должна генерировать ответ. Генерация «из описания в код» была выбрана для реализуемого в рамках исследования проекта, поскольку она обеспечивает большую гибкость для конечного пользователя – разработчик может начать «с нуля», очистить контекст и модифицировать код так, как ему необходимо. Выбор такого подхода был вдохновлен чат-ассистентами ChatGPT и YandexGPT [5, 6].

1.1.1 Обзор существующих решений

Решения конкурентов представлены различными инструментами и сервисами на основе ИИ. В данном разделе представлен обзор некоторых продуктов, известных среди разработчиков и послуживших источником вдохновения для проекта: ChatGPT, Copilot, GigaCode, Replit Ghostwriter и Amazon Q Developer [8 – 10]. Эти продукты имеют схожую базовую функциональность, но отличаются целевой платформой, возможностью встраивания в интегрированную среду разработки (Integrated Development Environment, IDE) и т. д. На рис. 2 представлены особенности каждого продукта. GitHub Copilot, GigaCode и Amazon Q обладают одинаковой функциональностью – реализуют ИИ-ассистента, который способен генерировать код на основе текстового описания и контекста проекта (кодовой базы). В этих инструментах также реализована функция общения с ассистентом: пользователь может задавать различные вопросы в процессе разработки, причем они не обязательно должны быть связаны с генерацией кода. Эти вопросы касаются объяснения кода, предоставлении данных о методе или алгоритме.

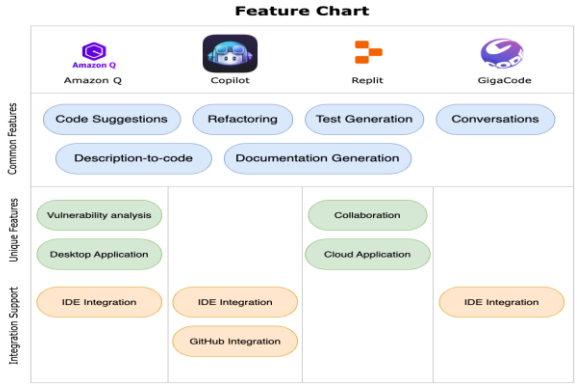

Рис. 2 – Основные функции ИИ-ассистентов конкурентов

Если Copilot и GigaCode практически идентичны по функциональности, то в Amazon Q реализован анализ безопасности кода, который помогает обнаружить потенциальные уязвимости и предлагает способы их устранения. Replit объединяет в себе уже упомянутые функции, однако сильно отличается от других ИИ-помощников, поскольку предоставляет независимую платформу для написания кода и совместной работы. Этот факт можно рассматривать и как недостаток, поскольку инструмент не может быть интегрирован в сторонние приложения, включая IDE.

Хотя ChatGPT не реализован исключительно для задач написания кода, разработчики также адаптируют его для использования в своих повседневных задачах. ChatGPT не может обнаруживать уязвимости или использоваться для эффективного рефакторинга, однако он предоставляет базовую функциональность для объяснения и генерации кода и может конкурировать с другими упомянутыми инструментами. Кроме того, было замечено, что не многие инструменты поддерживают языки программирования, которые используются в мобильной разработке (рис. 3).

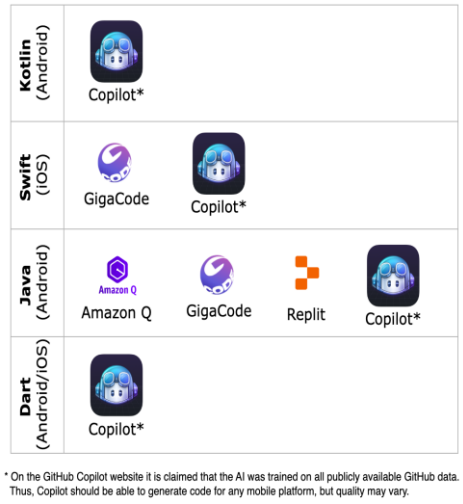

Рис. 3 – Языки программирования, поддерживаемые конкурентами

Например, утверждается, что GitHub Copilot способен генерировать код на всех языках программирования, которые использовались в проектах, размещенных на платформе GitHub, в то время как его российский конкурент GigaCode знает всего 15 языков программирования, в число которых не входит Kotlin – основной язык программирования для разработки под Android. Другие ИИ-ассистенты, такие как Replit и Amazon Q, совсем не предназначены для мобильной разработки. Важно также отметить, что хотя некоторые из этих инструментов могут быть интегрированы в Android Studio, используемую в Android-разработке, ни один из них не может быть легко применен в IDE Apple Xcode для iOS-разработчиков. Причиной тому является фактическая монополия Apple на IDE для iOS-разработки (за исключением JetBrains AppCode) и Apple App Store, что также делает добавление расширений в Xcode сложной задачей.

Таким образом, можно сделать выводы.

1. Инструменты искусственного интеллекта не так успешно находят свое применение в мобильной разработке, как в других областях программной инженерии. На рынке существует пока еще неосвоенная ниша;

2. Сервис, разработанный в рамках проекта, должен реализовать базовую функциональность общения с ИИ-ассистентом и генерации описания в код на языках программирования, используемых в мобильной разработке.

1.1.2 Применение LLM в программных проектах

Процесс разработки ИИ-ассистентов для генерации кода мало чем отличается от других программных продуктов на основе ИИ (рис. 4), так как используются те же принципы. Поскольку код представлен в виде текста, он может быть сгенерирован с помощью тех же языковых моделей, которые используются для генерации текста.

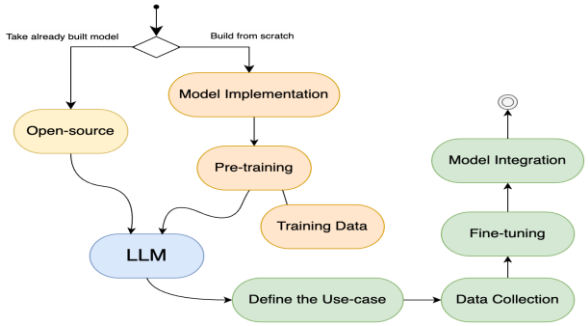

Рис. 4 – Этапы применения LLM в программном проекте

Модель, которая используется в реализации сервиса, имеет наибольшее значение, поэтому ее выбор является важным шагом. Существует два варианта: модель ИИ может быть разработана с нуля или взята из открытых источников. Хотя первый вариант дает возможность добиться наилучших результатов и настроить модель под себя, он выгоден только крупным компаниям с достаточными ресурсами. Подобная модель потребует предварительного обучения на больших объемах данных [11], которые не так просто получить, и на их обработку также потребуется время.

В настоящее время стало проще использовать модели ИИ в программном обеспечении, поскольку существуют различные открытые предварительно обученные модели. Таким образом, лучшим решением в некоторых случаях может быть использование открытой большой языковой модели в качестве основы, а затем ее тонкая настройка (fine-tuning) с помощью пользовательского набора данных, который представляет точные сценарии использования.

Реальный пример из индустрии – Copilot, при разработке которого был использован подобный подход: в качестве базовой модели была взята модель OpenAI Codex, которая дополнительно обучалась на публичных репозиториях GitHub и других общедоступных исходных кодах [12].

1.2 Постановка задачи

Инструменты и продукты ИИ для разработки программного обеспечения находят свое применение в реальных производственных условиях, а также в процессе обучения с помощью ИИ-ассистентов. Эти инструменты на базе ИИ используются для решения задач написания кода, анализа и объяснения кода, написания документации и т. д. Однако в области мобильной разработки до сих пор нет эксклюзивного решения, поскольку эти инструменты не ориентированы на какую-то одну область и стараются поддерживать как можно более разнообразный набор языков программирования. Некоторые из них не предлагают поддержку языков мобильной разработки в целом. Поэтому данный проект направлен на адаптацию существующих технологий искусственного интеллекта к потребностям мобильных разработчиков путем создания программного продукта, который можно применять на разных этапах процесса разработки для генерации кода, тем самым делая этот процесс более эффективным.

1.3 Цель и задачи

Основная цель работы – создать сервис, предоставляющий пользователям функциональность генерации кода (в частности, на языке программирования Swift), включающий в себя несколько микросервисов. Код должен генерироваться на основе текстовых описаний на естественном языке, предоставленных пользователем.

Проект включает в себя задачи:

– обзор конкурентов и сбор требований;

– сбор датасетов для тонкой настройки и оценки модели;

– fine-tuning LLM. Оценка модели на наборе тестов HumanEval, на других его вариациях, а также на собственном тестовом наборе [1]. Разработка клиентской и серверной частей приложения, интеграция LLM в приложение, его развертывание и тестирование;

– формирование инструкций по применению LLM для разработки инструмента для генерации кода, а также набора тестов для его тестирования.

1.4 Вопросы исследования

Наряду с разработкой сервиса даны ответы на вопросы.

1. Как оценить ИИ-модель для генерации кода?
2. Существует ли зависимость между качеством, эффективностью генерации и типом задачи генерации кода?
3. Как реализовать тонкую настройку модели: какой метод использовать, как собирать обучающие данные и какому формату они должны соответствовать?
4. Каково качество ответов модели и как можно добиться лучших результатов?

Ответы на первые два вопроса позволят лучше понять концепции, которые необходимо применить для разработки модуля ИИ для сервиса. Это также поможет раскрыть детали, касающиеся генерации кода большими языковыми моделями, и то, как подход к их применению в этом случае может отличаться от того, который используется для генерации текста на естественном языке.

1.5 Актуальность работы

Проект сосредоточен вокруг недавно возникшей области – применения LLM для генерации кода и исследует более конкретное направление – генерацию кода в мобильной разработке. Эта область не так популярна среди исследователей из-за своей узкой направленности и нуждается в новых и актуальных работах. Кроме того, как уже было отмечено выше, качественного программного решения для генерации кода мобильных приложений пока не существует, так как оно слишком специфично. Это еще одна причина, по которой данный проект может быть интересен. В работе предлагается не только сервис, но и рекомендации по применению LLM в инструментах генерации кода, а также набор тестов для оценки таких инструментов. Таким образом, проект может оказать определенное влияние на направление будущих исследований, касающихся более специфических областей в части генерации кода.

2 Подготовительная стадия проекта

2.1 Обзор предметной области

Для подготовки к выбору модели важно понять, какие модели ИИ существуют на данный момент, а также разобраться в терминах, которые используются в этой работе. Стоит начать с описания архитектур моделей, затем будет упомянут подход квантования модели, а также то, как эта техника предоставляет возможность запускать LLM локально с меньшими ресурсами.

2.1.1 Архитектуры моделей ИИ

Модели, которые хорошо справляются с задачами генерации текстов, называются моделями "последовательность-последовательность" (seq2seq). Одна из первых и наиболее успешных моделей seq2seq была представлена исследователями Google в 2014 году. Она представляет собой тип искусственной рекуррентной нейронной сети (RNN), состоящей из двух частей: кодера, который считывает входную последовательность и создает представление фиксированной длины, и декодера, который использует это представление для генерации выходной последовательности [13]. RNN используют предыдущие выходы и состояния сети в качестве дополнительной информации к текущему входу. Основной и наиболее важной особенностью RNN является "скрытое состояние", которое запоминает некоторую информацию о последовательности, это состояние также называют "состоянием памяти". В то время как традиционные глубокие нейронные сети предполагают, что входы и выходы независимы друг от друга, выход рекуррентных нейронных сетей зависит от предыдущих элементов в последовательности. Рекуррентная нейронная сеть состоит из нескольких блоков с фиксированной функцией активации, по одному для каждого временного шага. Каждый блок имеет внутреннее "скрытое состояние". На каждом временном шаге это скрытое состояние обновляется, что свидетельствует об изменении знаний сети о прошлом (рис. 5). Обновление скрытого состояния происходит с помощью рекуррентного соотношения:

$$h_t = f_w(h_{t-1}, x_t).$$

Для получения нового состояния h_t необходимо передать предыдущее скрытое состояние h_{t-1}и вход на текущем шаге x_t в функцию f_w, которая зависит от весов w. При передаче в модель новых данных процесс повторяется, причем функция f и веса w остаются неизменными. Однако стоит учитывать, что такая нейронная сеть становится неэффективной при обработке длинной цепи условий. Это связано с тем, что информация передается на каждом шаге, и чем длиннее цепь, тем больше вероятность потери данных.

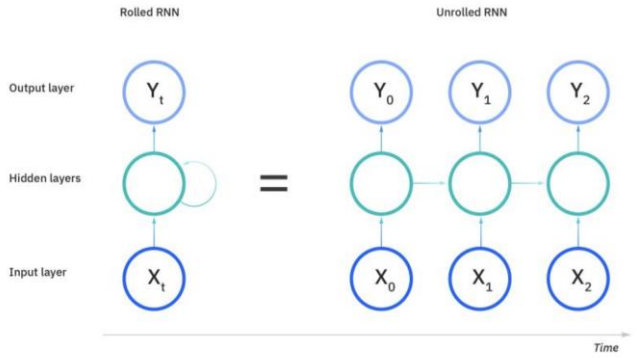

Рис. 5 – Архитектура RNN модели [14]

В процессе обучения RNN обычно сталкиваются с двумя проблемами, известными как "взрывающиеся градиенты" и "исчезающие градиенты" [15, 16]. Эти проблемы определяются величиной градиента, который представляет собой наклон функции потерь вдоль кривой ошибок. Если градиент слишком мал, он продолжает уменьшаться, обновляя весовые параметры, пока они не станут пренебрежимо малыми, т. е. близкими к 0. Когда это происходит, алгоритм прекращает обучение. Взрывные градиенты возникают, когда градиент слишком велик и создает неустойчивую модель. В этом случае веса модели становятся слишком большими и в итоге представляются как NaN (Not a Number). Для решения этих проблем придумано несколько улучшенных версий RNN: двунаправленные RNN и длительная кратковременная память (LSTM – Long Short-Term Memory).

Другой тип нейросетевой архитектуры – трансформер (рис. 6), которая в настоящее время наиболее популярна при проектировании LLM.

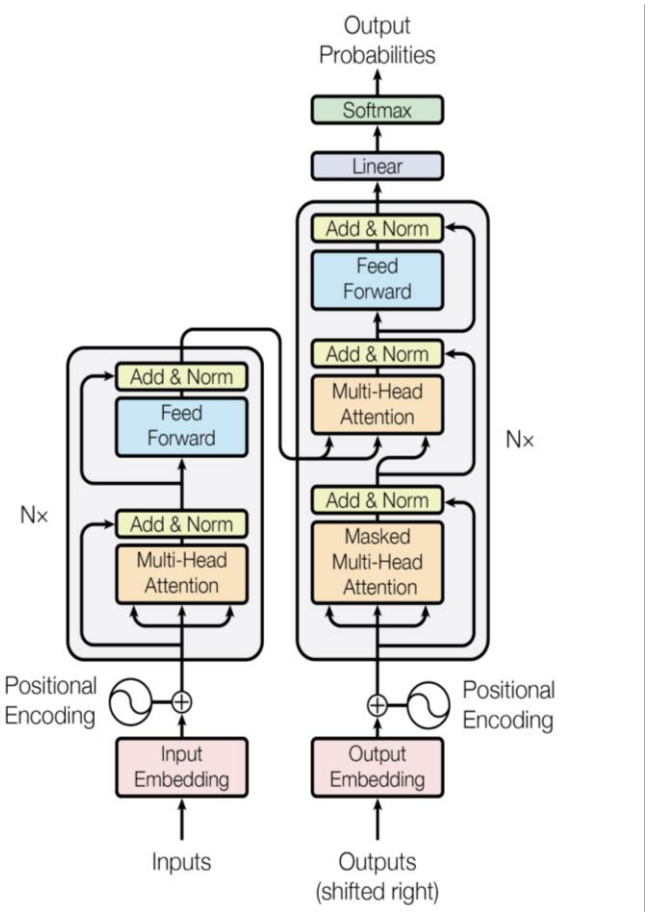

Рис. 6 – Архитектура модели типа трансформер [17]

Компания OpenAI использовала трансформер в GPT-3, а также в других версиях. Трансформер отличается от RNN наличием механизмов внимания (attention), позиционных embedding’ов, multihead attention. Все эти концепции были представлены в 2017 году в [17]. Трансформеры работают по той же логике, что и предыдущие модели. Входной текст преобразуется в вектор. Всем словам может быть присвоено векторное значение, которое

кодирует их смысл, а смысл – это их отношения с другими словами. Однако такое представление не отражает положение слов в последовательности. Для добавления этой информации используются позиционные embedding'и: они кодируют позицию каждой лексемы в последовательности. Это дает модели информацию о положении слов в последовательности, что важно для таких моделей, поскольку слова не обрабатываются последовательно. Полученная векторная последовательность преобразуется в выходной вектор, проходя через несколько слоев кодера и декодера (см. рис. 6).

Цель каждого слоя кодировщика – извлечь из последовательности признаки, а каждого слоя декодировщика – использовать эти признаки для получения выходной последовательности. Кодирование и декодирование осуществляются с помощью так называемого механизма "внимания", который помогает модели "понять" контекст, в котором используется слово. Трансформеры реализуют механизмы «внимания» (attention) и «самовнимания» (self-attention). Механизм "внимания" помогает модели анализировать различные части последовательности, генерируемой декодером. Механизм "самовнимания" работает с кодируемой последовательностью, позволяя модели оценивать степень важности различных частей входной последовательности по отношению друг к другу. Для этого вычисляется набор весов, которые указывают на степень важности каждого элемента входной последовательности по отношению к другим элементам. Это позволяет модели эффективно «улавливать» дальние зависимости во входной последовательности и учиться распознавать паттерны, охватывающие несколько элементов. Вместо того чтобы обращать внимание друг на друга только в одном измерении, трансформеры используют концепцию multihead attention. Multihead attention позволяет сети научиться нескольким способам взвешивания входной последовательности относительно самой себя. Суть этого заключается в том, что, например, во время перевода слов, каждому слову может быть уделено разное внимание в зависимости от типа задаваемого вопроса. Этот процесс может быть распараллелен, что позволяет ускорить обучение модели и глубже воспринимать контекст слов. Все это делает модели на основе архитектуры типа трансформер наиболее успешными при решении задач генерации текста, что подтверждается различными рейтингами [18, 19]. Это основная причина, по которой в данном проекте применяются и исследуются модели на основе именно архитектуры трансформер, хотя в настоящее время появляются и новые архитектуры, например Mamba, которые также могут быть интересны для применения в будущем в программных продуктах на основе ИИ [20].

2.1.2 Квантование

Квантование – это техника, используемая для уменьшения размера модели путем преобразования ее типа весов из представления с плавающей точкой высокой точности в представление с плавающей точкой низкой точности (16 бит) или даже целое число (8 бит). Помимо повышения скорости вычислений эта техника позволяет снизить требования к пропускной способности памяти [21]. Однако это означает, что точностью придется в определенной степени пожертвовать.

2.2 Этапы выбора LLM

Как уже говорилось выше, модель ИИ может быть взята из открытых источников или реализована с нуля. В данном проекте использовалась существующая модель. Причина этого в том, что для обучения "сырой" модели требуется много данных, времени и вычислительных мощностей, чтобы достичь результатов моделей конкурентов. Поэтому было решено, что наилучшим вариантом будет доработка уже предварительно обученной модели. Учитывая, что существует множество LLM, выбор модели – важный шаг, поскольку от этого зависят результаты проекта: качество, производительность генерации кода и сам процесс разработки. Несмотря на схожесть методик выбора моделей в различных исследованиях, процесс уникален для каждого случая использования. Сначала необходимо сформировать критерии для первоначальной оценки моделей, сформировать список подходящих моделей, оценить их на предмет эффективности и качества сгенерированных фрагментов кода. В работе предложен следующий список критериев.

1. Модель должна быть открытой и предварительно обучена на задачах генерации кода. Таким образом, ее будет легче настраивать (с точки зрения требуемых объемов данных и времени).

2. Описание модели должно содержать результаты оценки на бенчмарке типа HumanEval или других. Бенчмарк – это стандарт или эталон, по которому можно измерить производительность чего-либо [2].

3. Модель должна быть представлена в вариациях с разным числом параметров: решено остановиться на модели, содержащей около семи миллиардов параметров, чтобы построить тестовую версию сервиса. Затем сервис можно будет масштабировать и выбрать более объемную модель.

Поиск модели осуществлялся на онлайн-платформе HuggingFace, которая является наиболее популярной среди исследователей и разработчиков в области машинного обучения. Рассматривались модели на основе архитектуры трансформер, которые занимали высокие места в списках лидеров данной платформы [18, 19].

Кроме того, для сравнения моделей использовалась метрика HumanEval, поскольку она отражает способность модели понимать задачи программирования. Хотя HumanEval пригодилась на начальном этапе отбора моделей, в дальнейшем был применен собственный бенчмарк (причины описаны в следующем разделе). Для дальнейшего анализа были выбраны высокорейтинговые LLM из списка лидеров HuggingFace, имеющие вариации с семью миллиардами параметров (и более крупные версии): CodeQwen [23], Magicoder [24], Codellama [25], DeppMagic Coder [26], Mistral [27]. Хотя каждая модель была предварительно обучена на задачах генерации кода, CodeQwen и Mistral обучались не только на них.

2.2.1 Методы оценки моделей

Для дальнейшей оценки моделей необходимо было сформировать дополнительные критерии. Для создания сервиса важно как качество генерации программного кода, так и эффективность с точки зрения скорости генерации. Таким образом, в данном разделе рассматривается процессы оценки качества и эффективности больших языковых моделей. Качество ответов модели можно измерить, например, с помощью метрик машинного перевода, таких как BLEU, в том числе метрик, разработанных для сравнения фрагментов кода, таких как RUBY или CodeBLEU, а также с помощью бенчмаркинга или ручного тестирования сгенерированного кода. Недавнее исследование, проведенное в компании Jetbrains [20], показывает, что задача поиска метрики для генерации кода, которая бы совпадала с человеческой оценкой, все еще актуальна, учитывая, что широко используемые метрики, такие как BLUE и CodeBLUE, не подходят для этой задачи. Таким образом, для оценки качества генерации кода в данном проекте было решено использовать бенчмаркинг и ручное тестирование.

Существуют показательные и требовательные бенчмарки, которые используются для оценки моделей seq2seq, такие как MMLU, GPQA и т. д. Однако эти бенчмарки были специально разработаны для оценки ИИ-ассистентов, для которых важно "понимать" различные темы. Поскольку задача генерации кода специфична, использовать эти бенчмарки не стоит. Одним из самых популярных бенчмарков для генерации кода является HumanEval, созданный исследователями OpenAI, который можно увидеть во многих отчетах о тестировании, например, в недавно опубликованном Llama 3 [1, 21, 22]. Хотя HumanEval может быть полезен для определения того, насколько хорошо LLM «понимает» задачи написания кода, этот бенчмарк содержит 164 вопроса по программированию только с примерами кода на языке Python. Таким образом, нельзя полностью полагаться на HumanEval при оценке моделей, поскольку разработанный сервис ориентирован в первую очередь на язык программирования Swift. Следует отметить, что существуют и другие менее известные бенчмарки, однако на данный момент не удалось найти подходящий с исходным кодом на Swift, поэтому было решено предложить свой собственный бенчмарк.

2.3. Сбор данных для бенчмарка

Для создания бенчмарка было необходимо собрать тестовые примеры, каждый из которых содержит задаваемый модели промпт (prompt) и пример ответа или юнит-тест для проверки вывода. Примеры можно разделить на пять категории: реализация алгоритма, работа с сетью, генерация верстки, генерация компонентов пользовательского интерфейса (UI-компонентов) и генерация анимации. Для каждой из категорий составлено по десять тестовых примеров (всего 50). Лишь некоторые из них сопровождаются юнит-тестами, поскольку верстка и UI-компоненты, а также анимации могут быть сгенерированы множеством различных способов, в связи с этим результаты можно проверить только вручную. Промпт для тестового примера содержит инструкцию и входные данные. Инструкция описывает общий контекст задачи (например, что модель генерирует код только на Swift и не добавляет к нему текстовых пояснений). Входные данные – это сама задача (листинг 1).

```
Instruction: Your task is to generate code in Swift for an iOS application following the given description.
Input: Write SwiftUI code for a circular progress bar that fills based on a percentage value.
```

Листинг 1 – Пример промпта для тестового случая

Пример кода приводится вместе с промптом для ручного тестирования (листинг 2). Бенчмарк доступен в формате проекта Xcode playground [28].

```
import SwiftUI
struct ScalableButtonView: View {
    @State private var scale: CGFloat = 1.0
    var body: some View {
        Button(action: {
            withAnimation(.easeInOut(duration: 0.3)) {
                scale = scale == 1.0 ? 1.2 : 1.0
```

```
            }
        }) {
            Text("Tap me")
                .padding()
                .background(Color.blue)
                .foregroundColor(.white)
                .cornerRadius(8)
        }.scaleEffect(scale)
    }
}
```

Листинг 2 – Пример кода для тестового случая

2.4 Выбор модели

После того как были отобраны пять моделей, перечисленных в предыдущем разделе, и собран набор данных для бенчмарка, все эти модели были оценены с помощью собранного бенчмарка. Модели тестировались локально. Для запуска локального сервера для тестирования использовалась программа LM Studio [29]. Порядок тестирования для каждой модели был следующим:

1. Проверить окружение: должно быть доступно не менее 8 ГБ оперативной памяти, все фоновые процессы (кроме системных) должны быть остановлены;
2. Сервер должен быть запущен с помощью команды *lms server* из командной строки;
3. Предварительно скачанная модель должна быть полностью загружена в оперативную память с помощью команды *lms load* (во время тестирования выгрузка на GPU не применялась, но при необходимости она возможна);
4. Загруженная модель должна быть проверена командой *lms ps* (листинг 3);
5. После прохождения всех проверок необходимо запустить тестовый клиент с помощью команды *npm start*.

```
% lms ps
  LOADED MODELS
Identifier: TheBloke/deepseek-coder-6.7B-instruct-GGUF/deepseek-coder-6.7b-instruct.Q8_0.gguf
 • Type:  LLM
 • Path: TheBloke/deepseek-coder-6.7B-instruct-GGUF/deepseek-coder-6.7b-instruct.Q8_0.gguf
 • Size: 7.16 GB
 • Architecture: Llama
```

Листинг 3 – Вывод команды для проверки текущей загруженной модели

Тестовый клиент – это приложение на Node.js, которое было реализовано в рамках данного проекта [28]. Оно запускает все тест-кейсы, вызывая тестовый сервер с помощью LM Studio SDK. Клиент реализует следующий алгоритм:

1. Загрузка промпта каждого тест-кейса из файла CSV;
2. Вызов любой загруженной в данный момент модели с помощью запроса, содержащего системный промпт (одинаковый для каждого тестового случая) и пользовательский промпт –задачу, которую должна выполнить модель (листинг 4);
3. Сбор ответов модели;
4. Запись ответов в файл CSV, при этом каждая запись содержит текст ответа и значение tokensPerSecond (количество генерируемых токенов в секунду).

```
const prediction = anyModel.respond(
    [
      {
       role: "system",
       content:
       "Your task is to generate code in Swift for an iOS application following the given description. Don't give
explanations of the generated code.",
      },
      {
       role: "user",
```

```
        content: prompts[i],
      },
    ],
    {
      contextOverflowPolicy: "stopAtLimit",
      maxPredictedTokens: 2048,
      temperature: 0.7,
      inputPrefix: "Q: ",
      inputSuffix: "\nA:",
    }
  );
  const result = await prediction;
```

Листинг 4 – Фрагмент кода, использованного для вызова сервера тестирования

После того как ответы на тесты были собраны, они были оценены (в основном вручную). Для измерения качества модели применялась та же метрика "pass@", что и в HumanEval. Эта метрика отражает долю задач, для которых первая попытка ответа модели (pass@1) или любая из первых N попыток (pass@N) была правильной. Таким образом, pass@1 измеряет процент решений, которые были правильными с первой попытки. Метрику "pass@" можно определить следующим образом:

$$pass@k = 1 - \frac{\binom{n-c}{k}}{\frac{n}{k}},$$

где n – общее число тестовых случаев, c – число успешно пройденных тестов, k – число попыток.

Помимо качества сгенерированного кода, можно также оценить эффективность модели в контексте скорости генерации. Для ее измерения использована метрика t/s (tokens per second). После запуска тестового клиента получены отчеты о тестировании по бенчмарку для каждой из моделей, в результате чего собрано пять CSV-файлов [28], содержащих сгенерированные ответы и скорость их генерации в t/s. Проанализировав эти результаты, обнаружена модель наилучшего качества – CodeQwen, которая генерировала код без ошибок (рис. 7). Кроме того, для оценки среднего значения качества сгенерированного кода было решено выяснить, зависит ли качество генерации от типа поставленной перед моделью задачи (рис. 8).

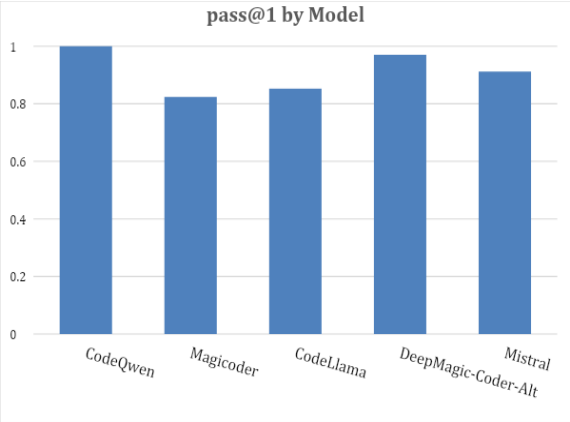

Рис. 7 – Значение метрики pass@1 для каждой из моделей

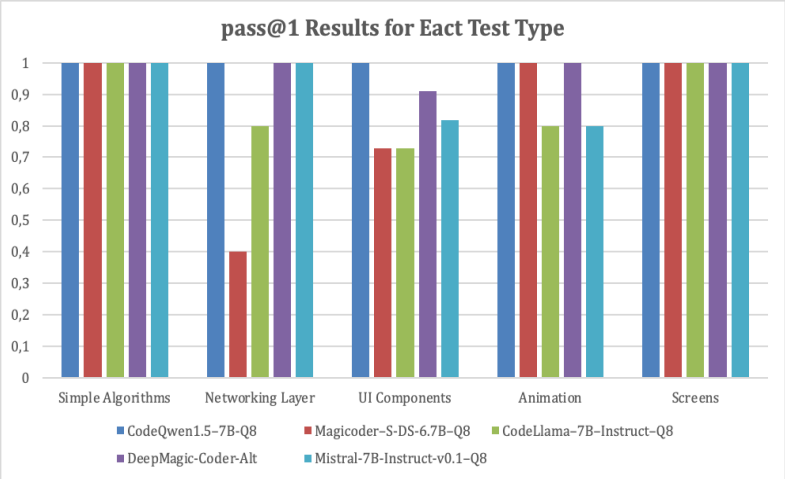

Рис. 8 – Качество генерации кода в зависимости от типа задачи

В итоге сделан вывод о том, что качество генерации для модели может варьироваться в зависимости от типа задачи написания кода. Каждая из моделей справилась со всеми заданиями по генерации простых алгоритмов на Swift. Больше всего ошибок выявлено при генерации кода отдельных компонентов пользовательского интерфейса и анимаций, хотя генерация верстки целого экрана не вызвала сложностей. Кроме того, замечено, что некоторые модели хуже справляются с определенными типами задач. Например, Magicoder оказался менее способным к решению задачи генерации сетевых запросов по сравнению, например, с задачами верстки или анимаций. Стоит отметить, что больше всего ошибок при генерации среди моделей было связано с ошибками компиляции и избыточными ответами, содержащими нерелевантные данные. Кроме того, для каждой из моделей была измерена скорость генерации, а затем рассчитаны минимальная, средняя и максимальная скорости (рис. 9).

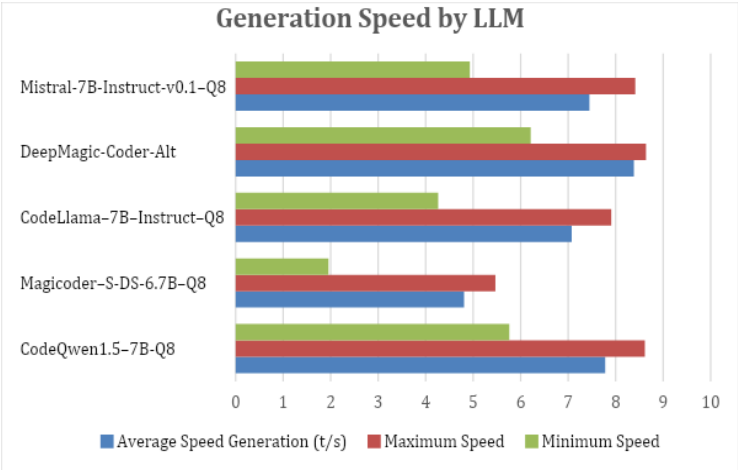

Рис. 9 – Скорость генерации для каждой из моделей

Модели по скорости работали примерно одинаково, и только одна модель – Magicoder оказалась значительно медленнее других LLM (она же генерировала ответы самого низкого качества). Кроме измерения скорости генерации, была также поставлена задача выяснить, существует ли зависимость между скоростью генерации и типом задачи (рис. 10).

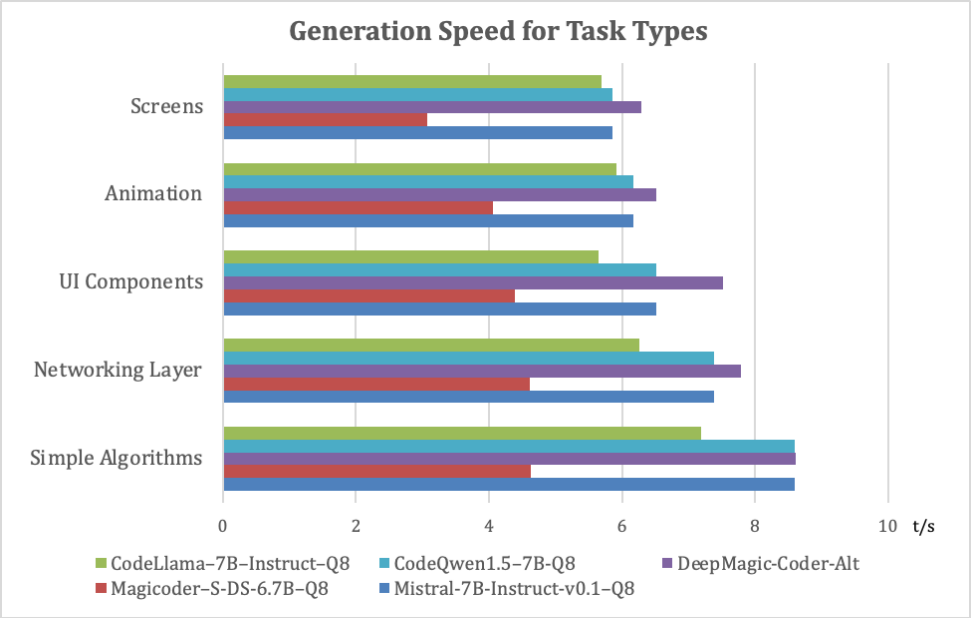

Рис. 10 – Скорость генерации в зависимости от задачи

Действительно, на скорость генерации влияла категория задания (а не количество лексем в вопросе). Можно заметить, что генерация макета была самой трудоемкой задачей, и это можно объяснить сложностью и длиной генерируемого ответа: были задания, которые требовали генерации нескольких компонентов. Отсюда можно сделать вывод, что LLM лучше справляются с генерацией бизнес-логики и алгоритмов для мобильного приложения, их также можно использовать для помощи в верстке, но не стоит ожидать высокой скорости и качества ответа.

По результатам анализа выбрана модель CodeQwen по причинам: отсутствия ошибок в сформированных ответах; это вторая по средней скорости генерации модель (после DeepMagic-Coder-Alt).

Другой альтернативой для будущего использования может стать DeepMagic-Coder-Alt.

Хотя ответы модели были корректными, некоторые из них содержали конструкции, которые не следует использовать (например, изменение цвета фона через объект Spacer), и зависимости от других (сторонних) библиотек (например, Alamofire) в тех случаях, где бизнес-логика может быть реализована полностью нативно.

3 Аспекты программной реализации

3.1 Архитектура сервиса

Сервис разработан на основе микросервисной архитектуры (рис. 11). Система в целом соответствует архитектуре клиент-сервер и разделена на клиентскую часть (веб-приложение) и серверную часть, которая включает в себя:

– API (Application Programming Interface) Gateway: точка входа в сервис;

– Authorization Service (сервис авторизации): сервис, отвечающий за аутентификацию и авторизацию пользователей, основанную на токенах;

– Chat Service (сервис чатов): сервис, отвечающий за хранение истории чатов пользователей;

– AI Module (ИИ-модуль): сервис с запущенной LLM.

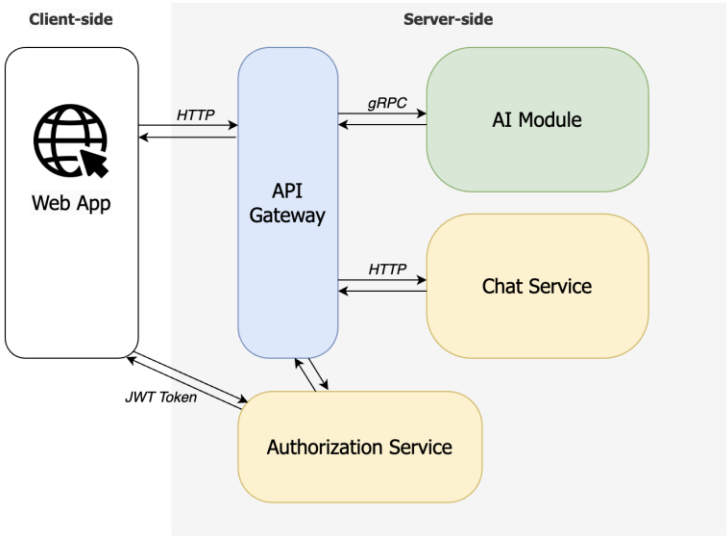

Рис. 11 – Архитектура системы

Данная архитектура была выбрана по нескольким причинам:

– разделение отсветственности: пользователи и сторонние приложения могут иметь возможность использовать лишь разработанный ИИ-модуль, не имея доступа к функционалу чата;

– безопасность: отдельный сервис авторизации обеспечивает необходимый уровень безопасности для каждого микросервиса;

– масштабируемость: ИИ-модулю потребуется больше ресурсов, чем другим микросервисам (например, GPU, что не требуется другим сервисам), т. е. важна возможность масштабировать сервисы независимо.

3.2 Реализация ИИ-модуля и тонкая настройка модели

Реализация модуля искусственного интеллекта включает в себя сбор данных и тонкую настройку. В данном проекте важно было сэкономить ресурсы, поэтому вариант с полным fine-tuning’ом, когда настраиваются все параметры модели, не рассматривался. Основное внимание было уделено подходу PEFT (Parameter-Efficient Fine-Tuning), который эффективно адаптирует предварительно обученные языковые модели без необходимости настройки всех параметров модели. PEFT используется для настройки выборки параметров, т. е. небольшого числа параметров в последних слоях модели, и позволяет значительно сократить время вычислений. Такой подход дает практически те же результаты, что и настройка всех параметров, благодаря архитектуре языковых моделей [17]. Каждый слой модели обрабатывает входные данные и выполняет определенные вычисления, после чего передает результат следующему слою. Как правило, первые слои отвечают за изучение более общей информации, такой как структура предложения, в то время как последние слои занимаются более специфическими задачами, такими как изучение значений слов. Таким образом, в настройку последних, более специфических слоев, заложено больше смысла.

LoRA (Low-Rank Adaptation) стала эффективной методикой для решения этой же задачи, сфокусированной на уменьшении числа обучаемых параметров при сохранении или повышении производительности модели. В частности, QLoRA, расширение LoRA, предлагает дальнейшую оптимизацию, что делает его рациональным выбором для тонкой настройки LLM. LoRA работает путем декомпозиции весовых матриц нейронных сетей в матрицы с низким рангом. Такой подход значительно сокращает число параметров, которые необходимо настраивать в процессе обучения. Обновляя лишь небольшое подмножество параметров модели, LoRA не только ускоряет процесс настройки, но и сокращает объем занимаемой памяти, что делает возможным тонкую настройку очень больших моделей на оборудовании с ограниченными ресурсами. QLoRA развивает эту идею, интегрируя в процесс адаптации методы, учитывающие квантование. Квантование [21] уменьшает точность весов модели, что значительно снижает требования к памяти и вычислительные затраты без существенного ущерба для точности. Комбинируя низкоранговую адаптацию с квантованием, QLoRA достигает еще большего снижения потребления ресурсов, получая при этом устойчивое повышение производительности, характерное для LoRA. По этой причине данный метод выбран для тонкой настройки.

Данные для fine-tuning’а собраны вручную из разных источников: с сайтов со статьями и учебниками, посвященными iOS-разработке [30, 31], и частично сгенерированы с помощью ChatGPT. Набор данных содержит три колонки: инструкция, ввод и вывод, каждая из которых включена в промпт для настройки (листинг 5).

```
prompt = """Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.
### Instruction:
{}
### Input:
{}
### Response:
{}
"""
```

Листинг 5 – Пример структуры prompt'а для тонкой настройки

Для fine-tuning’а модели написан скрипт на Python для обучения, описанного в предыдущем разделе: осуществляет загрузку набора данных для fine-tuning’а, затем загружает модель с конфигурацией QLoRA и обучает ее за одну эпоху. После тонкой настройки модель может быть загружена в HuggingFace для дальнейшего использования. Хотя для тестирования можно запускать модель локально, для fine-tuning’а требуются дополнительные вычислительные ресурсы, ввиду чего применена платформа Yandex DataSphere. Версия модели, использованная в реализованном прототипе, представлена на HuggingFace [32].

3.3 Серверное приложение

Как было указано выше, серверная часть содержит четыре сервиса: модуль искусственного интеллекта, сервис чата, сервис авторизации и API Gateway.

API Gateway выступает в роли прокси-сервера, обеспечивая авторизацию всех запросов к другим сервисам. Он взаимодействует с сервисом авторизации, который отвечает за предоставление JWT-токенов, используемых клиентами для выполнения запросов. Получив запрос, API Gateway проверяет токен и перенаправляет запросы к другим сервисам, если они успешно прошли все проверки. Таким образом, другим сервисам не нужно реализовывать никаких дополнительных проверок безопасности.

Сервис авторизации отвечает за генерацию и проверку JWT-токенов, а также предоставляет конечные точки CRUD для работы с данными пользователя: реализует регистрацию, вход и выход. Сервис хранит только минимально необходимую информацию: email, хэш пароля и URL-адрес изображения аватара пользователя.

Чат-сервис поддерживает весь функционал чата, доступный на стороне клиента, а именно: создание и удаление чатов, получение чатов и историю сообщений.

Все сервисы реализованы с использованием Kotlin в качестве основного языка программирования и Spring Framework для реализации API. Сервисы чатов и авторизации используют отдельные базы данных PostgreSQL. API задокументирован в формате Swagger Collection. Исходный код вместе с документацией Swagger доступен на GitHub [33].

3.4 Клиентское приложение

Клиентское приложение реализовано с использованием библиотеки React.JS и соответствует компонентной архитектуре, в которой все элементы пользовательского интерфейса сгруппированы в многократно используемые компоненты. Приложение содержит несколько экранов: главный экран, экран регистрации, экран входа и экран так называемой песочницы.

Экран песочницы предоставляет конечному пользователю всю функциональность чата. Пользователи могут отправлять новые сообщения в чат и получать ответы от ИИ-модуля. Пользователь получает визуальное уведомление о том, был ли запрос успешным или нет. Ко всем ответам применяется форматирование кода и подсветка синтаксиса (рис. 12).

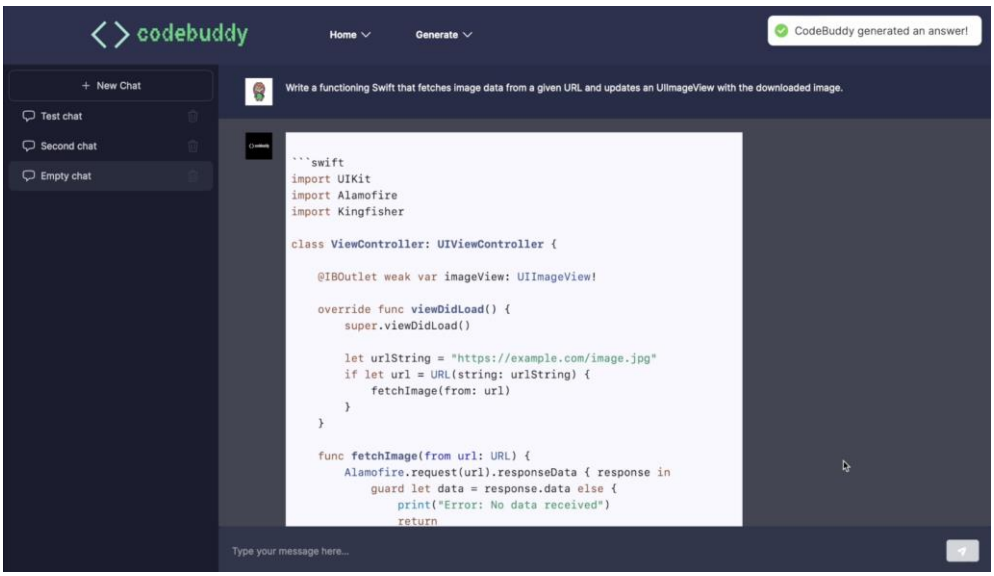

Рис. 12 – Пример ответа ИИ-ассистента

Кроме того, вся история сообщений сохраняется в базе данных на стороне сервера. Пользователи могут удалять чаты или создавать новые с помощью боковой панели.

4 Заключение

В качестве результата проекта реализован сервис для генерации кода мобильных приложений. Система предоставляет пользователю ожидаемую функциональность через клиентское приложение и отдельные микросервисы. Проект также включает в себя модуль искусственного интеллекта, который был создан на базе существующего LLM CodeQwen.

Подготовительный этап проекта включал в себя: исследование возможных применений генерации кода, обзор существующих архитектур LLM, методы fine-tuning’а и т.д. LLM, использованный в качестве базового для модуля искусственного интеллекта, был выбран на основе собственных критериев и собранного нами тестового набора данных (бенчмарка), который был разработан специально для оценки задач генерации кода на языке программирования Swift. Кроме того, чтобы в некоторой степени автоматизировать процесс оценки, был реализован тестовый клиент, который использовался для сбора ответов моделей, прошедших тестирование.

В процессе анализа результатов оценки моделей были выявлены задачи, которые влияют на эффективность и качество вывода модели. Также был собран набор данных, содержащий наиболее сложные задачи для fine-tuning’а модели.

Реализованный сервис использует опубликованную версию CodeQwen LLM, которая прошла fine-tuning. Реализован веб-клиент для конечного пользователя, а также серверная часть, содержащая три отдельных микросервиса и API Gateway.

Данная работа является результатом исследовательского проекта, реализуемого в рамках программы фундаментальных исследований Национального исследовательского университета "Высшая школа экономики" (НИУ ВШЭ). Реализованный проект может быть использован в образовательных целях и для тестирования предложеной концепции в качестве MVP.

Список источников

1. M. Chen et al., “Evaluating Large Language Models Trained on Code,” Jul. 2021. https://arxiv.org/pdf/2107.03374 (accessed May 08, 2024).

2. Merriam-Webster, “Benchmark.” Merriam-Webster Dictionary, https://www.merriam-webster.com/dictionary/benchmark (accessed May 17, 2024).

3. GitHub, “GitHub Copilot. Your AI pair programmer,” 2023. https://github.com/features/copilot (accessed May 08, 2024).

4. M. Anees, “Pros & Cons of AI Coding Assistants: A Boon for Developers or a Future Threat?”, Medium, Mar. 15, 2024. https://medium.com/@muhammad.anees_86442/pros-cons-of-ai-coding-assistants-a-boon-for-developers-or-a-future-threat-08f2d1651a23 (accessed May 08, 2024).

5. OpenAI, “ChatGPT.” https://openai.com/chatgpt/ (accessed May 08, 2024).

6. Yandex, “YandexGPT 2 — генеративная языковая модель Яндекса.” https://ya.ru/ai/gpt-2 (accessed May 08, 2024).

7. E. Dehaerne, D. Bappaditya, S. Halder, S. Gendt, W. Meert. Code Generation Using Machine Learning: A Systematic Review , 2022, IEEE Access. 10. 1-1. 10.1109/ACCESS.2022.3196347.

8. GigaCode, “AI-ассистент разработчика, который ускоряет создание ПО,” 2024. https://gigacode.ru/#/ (accessed May 08, 2024).

9. Replit, “Replit AI - Code faster with AI.” https://replit.com/ai (accessed May 08, 2024).

10. Amazon, “AI Coding Assistant - Amazon Q Developer,” Amazon Web Services, Inc. https://aws.amazon.com/q/developer/ (accessed May 08, 2024).

11. S. Johri, “The making of ChatGPT: From data to dialogue,” Science in the News, Jun. 06, 2023. https://sitn.hms.harvard.edu/flash/2023/the-making-of-chatgpt-from-data-to-dialogue/ (accessed May 08, 2024).

12. P. Krill, “OpenAI offers API for GitHub Copilot AI model,” InfoWorld, Aug. 12, 2021. https://www.infoworld.com/article/3629469/openai-offers-api-for-github-copilot-ai-model.html (accessed May 10, 2024).

13. I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” arXiv.org, 2014. https://arxiv.org/abs/1409.3215 (accessed May 08, 2024).

14. IBM, “What are Recurrent Neural Networks?”, IBM, 2023. https://www.ibm.com/topics/recurrent-neural-networks (accessed May 08, 2024).

15. A. Ceni, “Random orthogonal additive filters: a solution to the vanishing/exploding gradient of deep neural networks.”, arXiv.org. https://arxiv.org/pdf/2210.01245 (accessed: May 08, 2024).

16. Z. Lipton, J. Berkowitz, and C. Elkan, “A Critical Review of Recurrent Neural Networks for Sequence Learning,” 2015. https://arxiv.org/pdf/1506.00019 (accessed: May 08, 2024).

17. A. Vaswani et al., “Attention Is All You Need,” Jun. 2017. https://arxiv.org/pdf/1706.03762 (accessed: May 08, 2024).

18. mike-ravkine, “Can Ai Code Results - a Hugging Face Space”. https://huggingface.co/spaces/mike-ravkine/can-ai-code-results (accessed: May 22, 2024).

19. bigcode, “Big Code Models Leaderboard - a Hugging Face Space”. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard (accessed: May 22, 2024).

20. A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces,” May 2024. https://arxiv.org/pdf/2312.00752 (accessed: Jun. 12, 2024).

21. Deci. “The Ultimate Guide to Deep Learning Model Quantization and Quantization-Aware Training,” https://deci.ai/quantization-and-quantization-aware-training/ (accessed May 10, 2024).

22. Meta, “Meta Llama 3.” https://llama.meta.com/llama3/ (accessed May 12, 2024).

23. Qwen, “CodeQwen1.5-7B-Chat-AWQ”. Apr. 16, 2024. https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat-AWQ/blob/main/LICENSE (accessed: May 20, 2024).

24. ise-uiuc, “Magicoder-S-DS-6.7B”. Dec. 28, 2023. https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B (accessed: May 20, 2024).

25. codellama, “CodeLlama-70b-hf”. https://huggingface.co/codellama/CodeLlama-70b-hf (accessed: May 20, 2024).

26. rombodawg, “DeepMagic-Coder-7b-Alt”. https://huggingface.co/rombodawg/DeepMagic-Coder-7b-Alt (accessed: May 20, 2024).

27. mistralai, “Mistral-7B-v0.1”. Sep. 28, 2023. https://huggingface.co/mistralai/Mistral-7B-v0.1/discussions/13 (accessed: May 20, 2024).

28. L. Rezunik, “LucyRez/Swift-LLM-Benchmark,” GitHub. https://github.com/LucyRez/Swift-LLM-Benchmark (accessed: Jun. 20, 2024).

29. lmstudio.ai, “LM Studio.” https://lmstudio.ai/docs/welcome (accessed: May 20, 2024).

30. P. Hudson, “Hacking with Swift – learn to code iPhone and iPad apps with free Swift 5.4 tutorials,” Hacking with Swift. https://www.hackingwithswift.com/ (accessed: Jun. 20, 2024).

31. Kodeco, “Elevate your dev skills,” www.kodeco.com. https://www.kodeco.com/ (accessed: Jun. 20, 2024).

32. L. Rezunik, “LucyRez/CodeQwen-Swift”. https://huggingface.co/LucyRez/CodeQwen-Swift (accessed: Jun. 20, 2024).

33. L. Rezunik, “LucyRez/CodeBuddy”. https://github.com/LucyRez/CodeBuddy (accessed: Jun. 20, 2024).

References

1. M. Chen et al., “Evaluating Large Language Models Trained on Code,” Jul. 2021. https://arxiv.org/pdf/2107.03374 (accessed May 08, 2024).

2. Merriam-Webster, “Benchmark.” Merriam-Webster Dictionary, https://www.merriam-webster.com/dictionary/benchmark (accessed May 17, 2024).

3. GitHub, “GitHub Copilot. Your AI pair programmer,” 2023. https://github.com/features/copilot (accessed May 08, 2024).

4. M. Anees, “Pros & Cons of AI Coding Assistants: A Boon for Developers or a Future Threat?”, Medium, Mar. 15, 2024. https://medium.com/@muhammad.anees_86442/pros-cons-of-ai-coding-assistants-a-boon-for-developers-or-a-future-threat-08f2d1651a23 (accessed May 08, 2024).

5. OpenAI, “ChatGPT.” https://openai.com/chatgpt/ (accessed May 08, 2024).

6. Yandex, “YandexGPT 2 — Yandex generative language model.” https://ya.ru/ai/gpt-2 (accessed May 08, 2024).

7. E. Dehaerne, D. Bappaditya, S. Halder, S. Gendt, W. Meert. Code Generation Using Machine Learning: A Systematic Review, 2022, IEEE Access. 10. 1-1. 10.1109/ACCESS.2022.3196347.

8. GigaCode, “AI-ассистент разработчика, который ускоряет создание ПО,” 2024. https://gigacode.ru/#/ (accessed May 08, 2024).

9. Replit, “Replit AI - Code faster with AI.” https://replit.com/ai (accessed May 08, 2024).

10. Amazon, “AI Coding Assistant - Amazon Q Developer,” Amazon Web Services, Inc. https://aws.amazon.com/q/developer/ (accessed May 08, 2024).

11. S. Johri, “The making of ChatGPT: From data to dialogue,” Science in the News, Jun. 06, 2023. https://sitn.hms.harvard.edu/flash/2023/the-making-of-chatgpt-from-data-to-dialogue/ (accessed May 08, 2024).

12. P. Krill, “OpenAI offers API for GitHub Copilot AI model,” InfoWorld, Aug. 12, 2021. https://www.infoworld.com/article/3629469/openai-offers-api-for-github-copilot-ai-model.html (accessed May 10, 2024).

13. I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” arXiv.org, 2014. https://arxiv.org/abs/1409.3215 (accessed May 08, 2024).

14. IBM, “What are Recurrent Neural Networks?”, IBM, 2023. https://www.ibm.com/topics/recurrent-neural-networks (accessed May 08, 2024).

15. A. Ceni, “Random orthogonal additive filters: a solution to the vanishing/exploding gradient of deep neural networks.”, arXiv.org. https://arxiv.org/pdf/2210.01245 (accessed: May 08, 2024).

16. Z. Lipton, J. Berkowitz, and C. Elkan, “A Critical Review of Recurrent Neural Networks for Sequence Learning,” 2015. https://arxiv.org/pdf/1506.00019 (accessed: May 08, 2024).

17. A. Vaswani et al., “Attention Is All You Need,” Jun. 2017. https://arxiv.org/pdf/1706.03762 (accessed: May 08, 2024).

18. mike-ravkine, “Can Ai Code Results - a Hugging Face Space”. https://huggingface.co/spaces/mike-ravkine/can-ai-code-results (accessed: May 22, 2024).

19. bigcode, “Big Code Models Leaderboard - a Hugging Face Space”. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard (accessed: May 22, 2024).

20. A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces,” May 2024. https://arxiv.org/pdf/2312.00752 (accessed: Jun. 12, 2024).

21. Deci. “The Ultimate Guide to Deep Learning Model Quantization and Quantization-Aware Training,” https://deci.ai/quantization-and-quantization-aware-training/ (accessed May 10, 2024).

22. Meta, “Meta Llama 3.” https://llama.meta.com/llama3/ (accessed May 12, 2024).

23. Qwen, “CodeQwen1.5-7B-Chat-AWQ”. Apr. 16, 2024. https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat-AWQ/blob/main/LICENSE (accessed: May 20, 2024).

24. ise-uiuc, “Magicoder-S-DS-6.7B”. Dec. 28, 2023. https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B (accessed: May 20, 2024).

25. codellama, “CodeLlama-70b-hf”. https://huggingface.co/codellama/CodeLlama-70b-hf (accessed: May 20, 2024).

26. rombodawg, “DeepMagic-Coder-7b-Alt”. https://huggingface.co/rombodawg/DeepMagic-Coder-7b-Alt (accessed: May 20, 2024).

27. mistralai, “Mistral-7B-v0.1”. Sep. 28, 2023. https://huggingface.co/mistralai/Mistral-7B-v0.1/discussions/13 (accessed: May 20, 2024).

28. L. Rezunik, “LucyRez/Swift-LLM-Benchmark,” GitHub. https://github.com/LucyRez/Swift-LLM-Benchmark (accessed: Jun. 20, 2024).

29. lmstudio.ai, “LM Studio.” https://lmstudio.ai/docs/welcome (accessed: May 20, 2024).

30. P. Hudson, “Hacking with Swift – learn to code iPhone and iPad apps with free Swift 5.4 tutorials,” Hacking with Swift. https://www.hackingwithswift.com/ (accessed: Jun. 20, 2024).

31. Kodeco, “Elevate your dev skills,” www.kodeco.com. https://www.kodeco.com/ (accessed: Jun. 20, 2024).

32. L. Rezunik, “LucyRez/CodeQwen-Swift”. https://huggingface.co/LucyRez/CodeQwen-Swift (accessed: Jun. 20, 2024).

33. L. Rezunik, “LucyRez/CodeBuddy”. https://github.com/LucyRez/CodeBuddy (accessed: Jun. 20, 2024).