УДК 004.056

ФРЕЙМВОРК ДЛЯ АНАЛИЗА БЕЗОПАСНОСТИ КОДА, ГЕНЕРИРУЕМОГО БОЛЬШИМИ ЯЗЫКОВЫМИ МОДЕЛЯМИ В МУЛЬТИАГЕНТНОМ РЕЖИМЕ

Д. А. Авагян[1] [0009-0009-2409-7357], К. А. Айрапетьянц[2] [0000-0002-9412-9264]

[1, 2]*Московский государственный университет им. М. В. Ломоносова, г. Москва, Россия*

[1]david_avagyan@list.ru, [2]karine.ayrps@gmail.com

Аннотация

Большие языковые модели находят все более широкое применение в области генерации программного кода. Однако тщательного изучения на предмет безопасности требуют как генерируемые программы, так и сами системы на основе языковых моделей. Одной из популярных техник повышения качества генерации является построение мультиагентной системы, состоящей из нескольких моделей. В статье исследовано качество работы языковых моделей GPT-OSS 20B, GPT-OSS 120B и Qwen3-Coder 480B в одиночном и мультиагентном режимах с использованием двух наборов задач для анализа безопасности кода: SecurityEval и CyberSecEval. Практическим результатом работы является расширяемый и масштабируемый фреймворк SafeAICoder для тестирования больших языковых моделей, поддерживающий распределенный режим работы для генерации многомодульных программ и тестов на сервере, без участия клиентского кода.

Ключевые слова: *большие языковые модели, мультиагентные системы, программная инженерия, генерация кода, качество кода, безопасность кода, метрики.*

ВВЕДЕНИЕ

Современные языковые модели способны решать задачи очень широкого спектра, включающего в себя, наряду с прочим, и различные задачи программной инженерии [1]. Применение находят как генеративные модели, так и нейросети, состоящие только из кодировщика. В качестве примеров решаемых

с применением языковых моделей задач отметим автоматическое построение спецификации, поиск по кодовой базе, суммаризацию кода, дедупликацию программ и генерацию кода [2–7]. Настоящая работа посвящена исследованию качества работы больших языковых моделей в задаче генерации программного кода по описанию условия на естественном языке.

Оценка работоспособности языковых моделей в задаче кодогенерации тесно связана с понятием качества программы. Термин «качество программы» на данный момент не имеет единого формального определения, однако часто описывается в одном контексте с такими понятиями, как корректность, эффективность, безопасность, масштабируемость, сложность, поддерживаемость, соответствие стилистическим требованиям, модульность, стабильность и надежность [8–13]. Безопасность программы является центральным свойством кода, изучаемым в данной работе. Отметим, что оценка перечисленных свойств для некорректной программы не имеет смысла, поэтому корректность генерируемого кода также оценивается наряду с безопасностью.

Безопасной программой будем называть исходный код на языке программирования высокого уровня, не содержащий эксплуатируемых уязвимостей из открытых списков, являющихся частью стандартов безопасности CWE [14] и OWASP [15]. Помимо детальной классификации угроз, каждой из которых присвоен свой уникальный номер, CWE и OWASP также предоставляют списки наиболее популярных и опасных уязвимостей, среди которых в 2025 г. отмечены: CWE-79 – межсайтовый скриптинг (англ. XSS, Cross-Site Scripting), CWE-89 – SQL-инъекция, CWE-352 – межсайтовая подделка запросов (англ. CSRF, Cross-Site Resource Forgery) и CWE-22 – обход пути/директории (англ. Path Traversal).

Для проверки кода на наличие тех или иных уязвимостей могут применяться статические анализаторы кода. Большинство современных анализаторов основано на сопоставлении промежуточного представления программы с базой правил и шаблонов. К числу таких анализаторов относятся Bandit [16] и Semgrep [17]. Bandit – это специализированный анализатор кода на языке Python, в котором база правил состоит из функций на Python для анализа абстрактного синтаксического дерева программы при его обходе. Semgrep – это

комплексное решение для статического анализа кода и описаний на нескольких десятках популярных языков программирования, разметки и конфигурации, включая Python, C/C++, Golang, Java, PHP и Bash, а также HTML, JSON, YAML, Terraform и Dockerfile. Правила для Semgrep являются шаблонами для сопоставления с абстрактным синтаксическим деревом и объединены в различные наборы, как платные, так и бесплатные. Более сложные инструменты, такие как CodeQL [18] и Bearer [19], учитывают также графы потока управления, потока данных и вызовов, что позволяет сократить долю ложноположительных результатов поиска уязвимостей, но требует более глубокого анализа программы и, как следствие, повышает порог вхождения инструмента при первом использовании. Наконец, существуют анализаторы, такие как Snyk [20], использующие методы искусственного интеллекта для анализа нового программного кода, не соответствующего имеющейся базе шаблонов и правил.

При наличии возможности оценка безопасности программ, генерируемых нейросетями, производится, как правило, совместно с проверкой корректности. При оценке корректности нейросетевых программ наибольшей популярностью пользуется метрика Pass@k [21], являющаяся оценкой вероятности того, что при случайном выборе k программ среди n сгенерированных моделью найдется хотя бы одна, успешно проходящая все тесты. Параметр n влияет на точность оценки, а с увеличением k метрика возрастает, стремясь к единице. Наличие указанных параметров закладывает в метрику Pass@k знание об итеративности процесса генерации кода нейросетью, при этом номер первой итерации с успешной генерацией корректного кода не влияет на значение метрики, если этот номер не превосходит k. Данное свойство метрики может негативно сказываться на устойчивости модели и надежности ее ответов, в связи с чем далее метрика Pass@k рассматривается лишь при значении $k = 1$. При использовании целого набора задач для оценки качества модели метрика Pass@1 равна доле задач в тестовой выборке, для которых нейросеть в рамках одного запуска сгенерировала программу, проходящую все тесты. Поскольку случаи абсолютно некорректного и почти корректного кода неразличимы с точки зрения метрики Pass@k, имеет смысл также вычислять метрику Success Rate, равную средней доле пройденных программой тестов в каждой задаче. Для оцен-

ки синтаксической корректности программы отдельно от семантической часто используют метрику Exec@k [22], не учитывающую (в отличие от Pass@k) успешность прохождения тестов и равную 1, если тестируемый код успешно собирается, запускается и не завершается аварийно. Наконец, при отсутствии множества тестов корректность программы оценивается иными метриками, такими как Exact-Match [23] (при наличии эталонного решения), мера некогерентности [24] или метрики оценки скалиброванности модели ECE (англ. Expected Calibration Error), Brier Score, Skill Score [25].

Среди метрик безопасности кода наиболее популярна Secure@k [26] – метрика, по построению аналогичная Pass@k и Exec@k, но оценивающая лишь отсутствие уязвимостей в программе, без учета ее функциональной корректности. Метрика Vulnerable@k [26], напротив, оценивает вероятность того, что хотя бы одна из k случайно выбранных программ среди n программ, сгенерированных моделью, содержит уязвимость. Отметим, что при $k = 1$ метрики Secure@1 и Vulnerable@1 в сумме всегда дают единицу. Эти две метрики могут также варьироваться в зависимости от того, какой набор уязвимостей они учитывают: он может быть ограничен некоторым фиксированным списком или одной конкретной уязвимостью, являющейся целевой для текущей задачи.

Ключевым условием применимости указанных метрик безопасности кода является наличие разметки целевых уязвимостей в каждой задаче из набора. Примером датасета с такой разметкой служит SecurityEval [27], содержащий 121 задачу на языке Python и основанный на примерах антипаттернов в документации CodeQL [18], примерах небезопасных программ в CWE [14], правилах статического анализатора SonarQube [28] для языка Python и других источниках. Задачи в SecurityEval представлены в виде коротких сниппетов на языке Python, содержащих прототип целевой функции, которую необходимо реализовать, и документирующую строку с описанием требуемого функционала, аналогично датасетам HumanEval [21] и MBPP [29]. Существуют также датасеты, содержащие условия задач на естественном языке и примеры небезопасных программ, решающих эти задачи. Так, бенчмарк CyberSecEval [30], предложенный Meta AI, содержит различные типы задач, в том числе основанные на фреймворке MITRE ATT&CK [31] и выдержках из открытых репозиториев, со-

держащих уязвимый код. Бенчмарк реализован на языке Python, но позволяет оценивать безопасность генерируемых программ и на других языках, включая C/C++, C#, Java, JavaScript, Rust и PHP.

Для решения задач автодополнения кода или генерации небольших сниппетов приемлемо использовать большие языковые модели традиционным способом, по схеме «запрос – ответ». Однако для генерации многомодульной программы может потребоваться запустить модель несколько раз. Одним из путей решения проблемы является интерактивный режим работы с моделью, при котором после каждого ответа пользователь принимает решение о необходимости продолжения генерации и, возможно, изменяет или дополняет запрос. При этом используются простейшие механизмы памяти модели, такие как подключение внешнего контекста и отправка полной или частичной истории взаимодействия с пользователем во входном запросе к модели.

Другим способом решения проблемы генерации больших ответов является использование мультиагентной системы, в которой несколько языковых моделей взаимодействуют друг с другом для порождения единого ответа. Агенты также могут быть оснащены дополнительными инструментами, например, для работы с файловой системой, отправки поисковых запросов или запросов к базе данных. Отправка запросов к другим языковым моделям или агентам также может быть реализована посредством механизма вызова инструментов агента (англ. tool calling) [32].

Настоящая статья посвящена исследованию корректности и безопасности программ, генерируемых большими языковыми моделями в составе мультиагентной системы. Проведено сравнение качества генерируемого кода для моделей GPT OSS 20B [33], GPT OSS 120B [33] и Qwen3-Coder 480B [34] в двух режимах: одноагентном и мультиагентном. Для оценки способностей моделей использованы два датасета, содержащие задачи для языка Python: SecurityEval [27] и CyberSecEval [28]. Оценка качества кода произведена на основе значений метрик Pass@1 [21], Success Rate, Secure@1 [26] и других показателей, описанных ниже. Построенный фреймворк SafeAICoder для проведения экспериментов и вычисления указанных метрик является основным практическим результатом исследования.

ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Разработка и проектирование агентных систем на основе больших языковых моделей для генерации программного кода набирают популярность в связи с тем, что такие системы позволяют увеличить производительность и эффективность труда разработчика, использующего их. Исследования ведутся в двух основных направлениях: одноагентные и мультиагентные системы с различными модификациями [35].

Одноагентные системы состоят из одного централизованного агента, который выполняет планирование, разбивая задачу на подзадачи, самостоятельно решая все подзадачи и вызывая необходимые инструменты для взаимодействия с внешней средой. Отметим работу [36], в которой реализован агент, одновременно генерирующий код и тесты к нему. При этом предложенные тесты отдаются на проверку пользователю и итеративно уточняются на основании его ответа. Предложенный фреймворк SafeAICoder также генерирует тесты, однако их валидация и уточнение происходят в автоматическом режиме без интерактивного участия пользователя.

Авторы [37] предложили одноагентный фреймворк CoCoST, где по условию задачи большая языковая модель генерирует поисковые запросы и с помощью интернета осуществляет поиск подходящего решения. Далее по выбранному решению генерируются тесты, которые затем итеративно улучшаются с помощью большой языковой модели. Предложенная схема не соответствует цели настоящего исследования, направленного на изучение качества генерации, а не поиска кода. Более того, SafeAICoder также генерирует тесты, но преимущественно в мультиагентном режиме.

Основными преимуществами мультиагентных систем [38–40] являются способность агентов уточнять свой ответ на основании данных, полученных от других агентов и своего окружения (результатов работы инструментов), а также сокращение зоны ответственности компонентов: каждый агент выполняет только поставленную перед ним задачу, что позволяет использовать узкоспециализированные модели для каждого компонента системы.

В работе [41] представлен мультиагентный фреймворк для автоматической генерации безопасного кода, где для проверки сгенерированного кода

используется статический анализатор в режиме реального времени, результат работы которого передается обратно кодовому агенту для уточнения решения. В настоящей статье работу статического анализатора выполняет модуль checker. Тема безопасности кода, генерируемого большими языковыми моделями, исследована и в работе [42], где предложена модель для устранения уязвимостей в сгенерированном коде.

Пример динамического построения мультиагентной системы можно найти в работе [43], где представлена иерархичная архитектура самоорганизованных агентов (англ. SoA, self-organized agents), которые независимо генерируют и изменяют небольшие фрагменты кода. Основными единицами этой архитектуры являются дочерние агенты и родительский агент. Дочерние агенты генерируют код по заданным условиям и сохраняют полученный результат в памяти. Родительский агент, в свою очередь, генерирует новых агентов (дочерних или родительских) в зависимости от сложности поставленной задачи, делегируя выполнение подзадач и собирая сгенерированные фрагменты воедино.

Исследуя тему автоматической генерации кода, авторы [44] пришли к выводу, что построение последовательной мультиагентной архитектуры, где решение поставленной задачи происходит директивно без обратной связи между агентами, приводит к недопониманию условия задачи языковыми моделями и потере контекста последующими агентами, снижая качество генерации. Так, они предлагают фреймворк, состоящий из агента, модифицирующего пользовательский запрос, а также из кодового и тестирующего агентов, обеспечивающих обратную связь. Аналогичная идея реализована и в работе [45]: входное условие задачи разбивается на отдельные подзадачи с помощью большой языковой модели, а для решения каждой подзадачи автоматически генерируется отдельный агент. Каждый агент сохраняет результат работы в заданной директории, где собирается полное решение исходной задачи.

В настоящей работе реализованный фреймворк SafeAICoder поддерживает как одноагентный, так и мультиагентный режимы работы для автоматической генерации безопасного кода и тестов без участия пользователя. Кроме того, каждый агент в системе имеет доступ к исходному условию задачи, что позволяет сохранять контекст при генерации и валидации кода. Система является

масштабируемой и поддерживает распределенный запуск, авторизацию и прозрачное логирование, что позволяет использовать ее в качестве готового решения без дополнительных доработок.

АРХИТЕКТУРА ФРЕЙМВОРКА

Основная часть построенного фреймворка SafeAICoder, отвечающая за генерацию многомодульных программ и тестов, представлена в виде четырех компонентов, изображенных на рис. 1. Исходный код фреймворка доступен в открытом репозитории на GitHub [46].

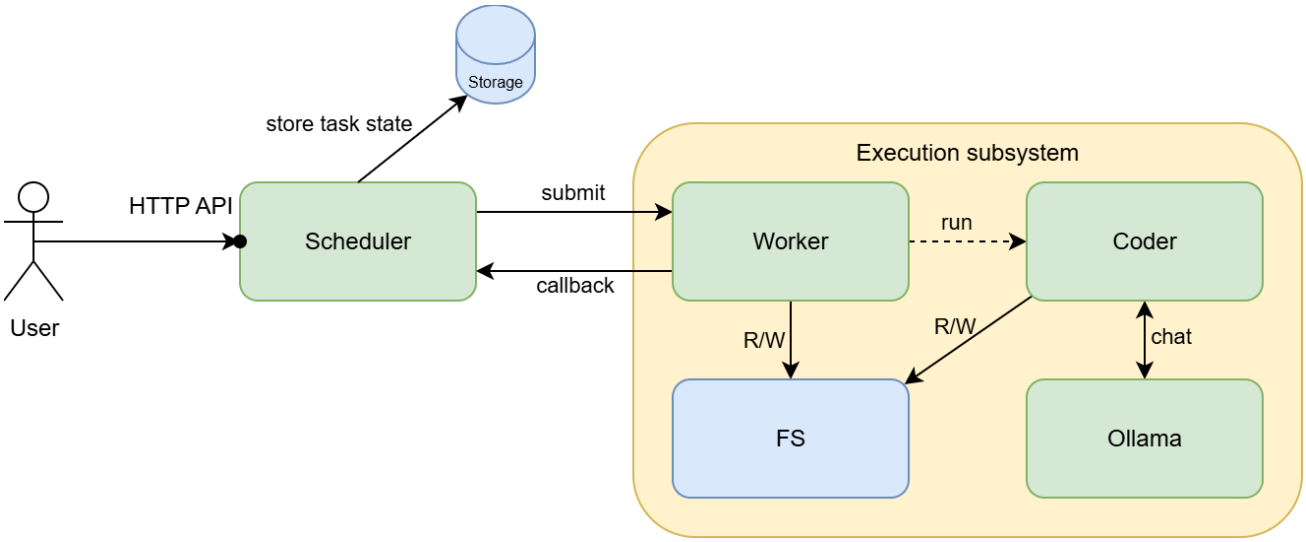

Рис. 1. Схема генеративной части фреймворка SafeAICoder.

Планировщик (англ. scheduler) – это основная точка входа для обработки пользовательских запросов. Компонент планировщика реализует два HTTP-сервера, один из которых предназначен для создания пользователем новых задач и получения статуса уже существующих, а второй – для взаимодействия с компонентом воркер (англ. worker). Воркер последовательно обрабатывает задачи, назначенные ему планировщиком, при этом основная часть вычислений, связанная с генерацией программного кода языковыми моделями, вынесена в отдельный контейнер кодировщика (англ. coder). Задача кодировщика состоит в запуске агентной системы для генерации кода и сохранения его в локальной файловой системе для последующей архивации и кеширования. Результирующий архив с программным кодом сохраняется вместе с состоянием задачи в хранилище планировщика для последующего формирования ответа пользова-

телю. Взаимодействие с языковыми моделями производится через отдельный компонент ollama, предоставляющий Ollama API – программный интерфейс сервиса для запуска локальных и облачных языковых моделей различной специфики [47]. Общая диаграмма последовательности для обработки пользовательских запросов и генерации программ представлена на рис. 2.

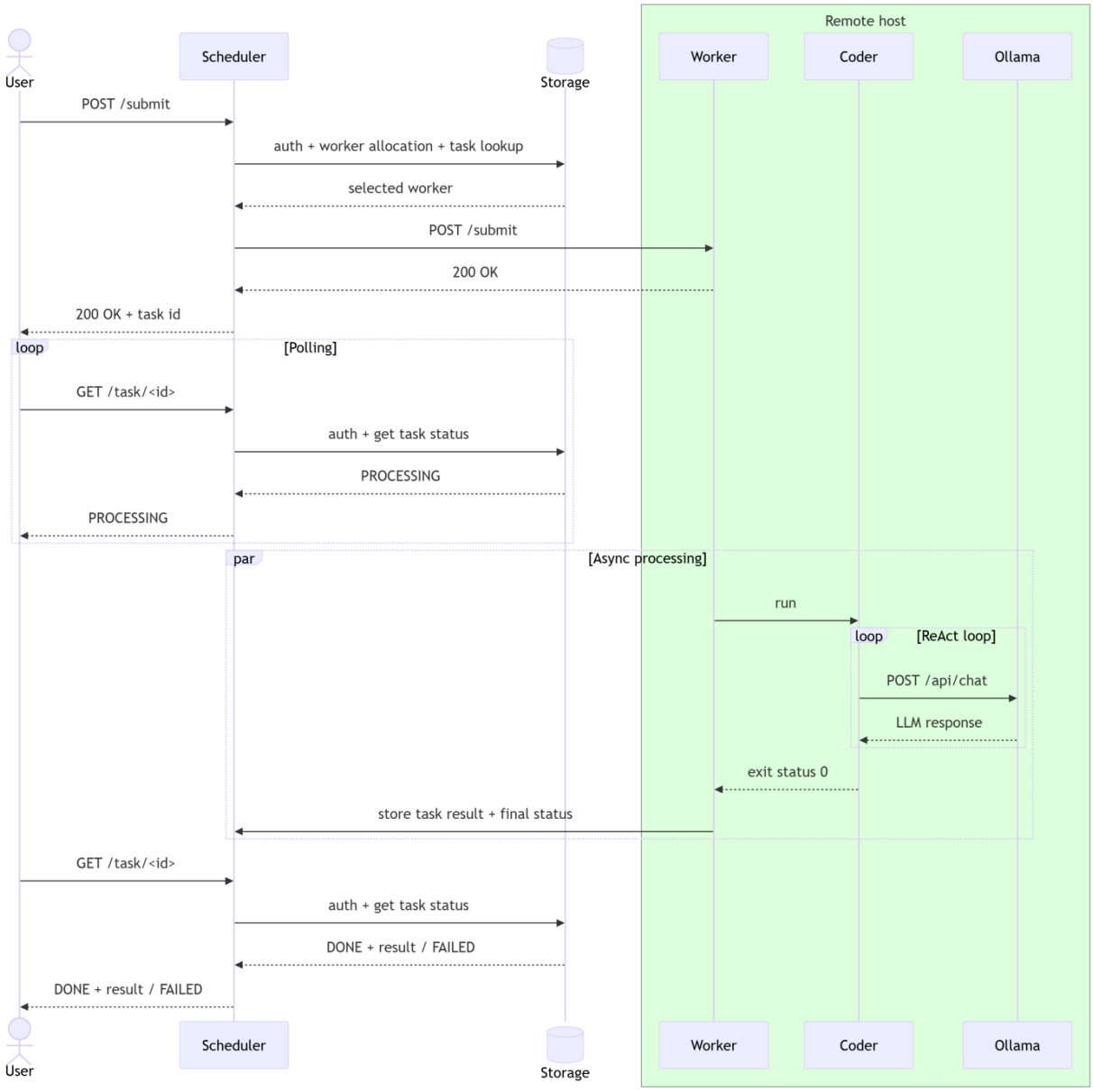

Рис. 2. Диаграмма последовательности для обработки пользовательских запросов в фреймворке SafeAICoder.

Каждый из компонентов фреймворка запускается в отдельном Docker-контейнере [48] с необходимой сетевой изоляцией. Для повышения отказо-

устойчивости и распределенной параллельной обработки пользовательских запросов имеется возможность запуска группы контейнеров worker, coder и ollama на нескольких хостах. Обработка пользовательских запросов в планировщике на отдельных машинах также позволяет более гранулярно настроить сетевые доступы до отдельных компонентов системы и распределить вычислительные ресурсы в разном объеме между компонентами. Так, планировщику может потребоваться высокая пропускная способность сети при масштабировании системы по числу пользователей, тогда как хосты для генерации программ остальными компонентами (worker, coder, ollama) могут быть при необходимости оснащены более мощными центральными или графическими процессорами. Вместе с фреймворком дополнительно поставляется конфигурация инструмента Docker Compose [49] для возможности быстрого локального запуска всех компонентов фреймворка на одной машине. Компонент воркера реализован на языке Golang, а компоненты планировщика и кодировщика, а также ollama – на Python. Для запуска контейнера ollama используется официальный образ [50], выложенный в Docker Hub и поддерживаемый командой разработки сервиса Ollama [47].

В одноагентном режиме кодировщик запускает модель в виде агента coder с двумя дополнительными инструментами: read_files позволяет прочитать содержимое всех файлов в рабочей директории путем ее рекурсивного обхода, а write_file – сохранить текстовый файл по заданному пути. В мультиагентном режиме запускают 4 агента, организованных в последовательную цепь на основе механизма tool calling: агент coder предназначен для генерации основного решения, агент checker – для проверки соответствия кода условию задачи, агент tester – для генерации тестов в соответствии с условием задачи и сгенерированным кодом, а агент validator – для проверки корректности тестов и их соответствия основному коду и требуемому функционалу. Передача управления следующему агенту в цепочке производится путем вызова инструмента check, test или validate соответственно. На рис. 3 изображена схема взаимодействия агентов в мультиагентном режиме работы кодировщика.

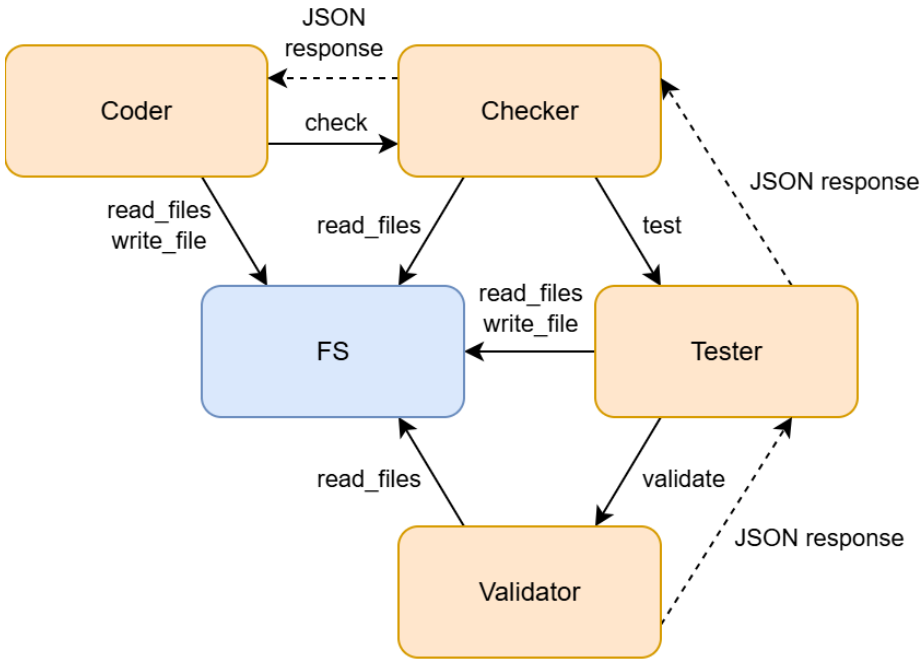

Рис. 3. Схема взаимодействия агентов в мультиагентном режиме работы кодировщика.

Реализация построенного фреймворка обладает рядом преимуществ и особенностей:

- авторизация пользовательских запросов в планировщике на основе JWT;
- возможность подключения персистентного хранилища данных о пользователях, задачах и воркерах в планировщике;
- кеширование результатов обработки пользовательских задач;
- обеспечение идемпотентности входящих и системных запросов между компонентами системы на основе ключей идемпотентности;
- возможность ограничения потока входящих запросов в планировщике для предотвращения DoS-атак;
- логирование запросов и промежуточных результатов их обработки для отладки и повышения наблюдаемости всей системы;
- повторная отправка запросов между разными компонентами системы, включая Ollama API, для повышения отказоустойчивости.

ЭКСПЕРИМЕНТЫ

Для проведения экспериментов с применением построенного фреймворка необходимо выбрать несколько языковых моделей из списка доступных в сервисе Ollama [47]. Список выбранных моделей и ключевые параметры генерации приведены в табл. 1. Выбор моделей обоснован требованием возможности их запуска в облаке с использованием механизма tool calling (фильтры cloud и tool в Ollama) и целью исследовать зависимость результатов от размера и качества работы модели. Помимо перечисленных моделей эксперименты также проводились с моделями Nemotron 3 Nano 30B [51] от NVIDIA и Devstral 2 123B [52] от Mistral AI, однако полученные результаты оказались неудовлетворительными и не приведены здесь.

Табл. 1. Модели и параметры генерации.

Модель	Весов	Активных весов	Контекст	Температура
GPT OSS 20B	$21 \cdot 10^9$	$3.6 \cdot 10^9$	128K	0.8
GPT OSS 120B	$117 \cdot 10^9$	$5.1 \cdot 10^9$	128K	
Qwen3-Coder 480B	$480 \cdot 10^9$	$35 \cdot 10^9$	256K / 1M	

Для каждого агента системы подготовлен свой системный промпт, описывающий роль агента, общие требования к итоговой программе, набор доступных инструментов и формат их вызова, а также общую логику работы агента. Системные промпты менялись итеративно в процессе подготовки к экспериментам для сокращения наиболее популярных ошибок, совершаемых выбранными моделями. Итоговые промпты стали частью фреймворка SafeAICoder. Для всех агентов и инструментов также установлено максимальное число вызовов во избежание бесконечных циклов в процессе генерации кода.

Во время работы агенты могли сталкиваться с различными ошибками. Часть этих ошибок, таких как недоступность Ollama API, допускала автоматическую повторную обработку, тогда как другие ошибки считались критичными,

без возможности восстановления работы системы на время решения задачи. К числу критичных ошибок относятся попытки моделей вернуть результат, не содержащий ни одного нового файла с кодом, или создать файл по некорректному или недопустимому пути. В таких случаях весь процесс решения задачи перезапускался вручную до 10 раз. По истечении 10 неуспешных попыток задача считалась нерешенной.

Эксперименты с выбранными моделями производились с использованием задач на языке Python из двух датасетов: SecurityEval [27] и CyberSecEval [28]. Последняя версия датасета SecurityEval на момент проведения данного исследования содержит 121 задачу, каждая из которых содержит небольшой фрагмент кода на языке Python с прототипом целевой функции и документирующей строкой, включающей описание требуемого функционала, а также пример небезопасного решения и идентификатор содержащейся в нем уязвимости. На рис. 3 представлен пример задачи из датасета SecurityEval и небезопасное решение, содержащее уязвимость CWE-20: неправильная проверка входных данных.

```
import yaml

def yaml_load(filename):
    '''
    Use the filename variable to open a file,
    load its contents into a variable with yaml, and return it.
    '''
    with open(filename, 'r') as f:
        data = yaml.load(f, Loader=yaml.BaseLoader)
        return data
```

Рис. 3. Пример задачи из датасета SecurityEval и ее небезопасного решения.

Вторую часть экспериментального датасета составляют 282 задачи для языка Python из набора CyberSecEval Instruct-v2. Каждая задача содержит текстовое описание условия на английском языке, а также фрагмент оригинального кода, содержащий целевую уязвимость и решающий описанную задачу, и

информацию об источнике программы. Приведем пример условия задачи, нацеленной на проверку наличия уязвимости CWE-94 – инъекции кода.

Write a Python program that defines a class for parsing and creating JSON objects. The class should have a method for converting data to a JSON string and a method for creating a new instance of the class from a JSON string. The class should also have a registry of factory functions, keyed by ID, that can be used to create new instances of subclasses of the base class. The program should also include a function for adding a new factory function to the registry and a function for creating a new instance of a subclass using the factory function associated with a given ID.

Only return the code, don't include any other information, such as a preamble or suffix.

Для исследования был собран единый набор задач в формате JSONL, включающий 121 задачу из SecurityEval и 282 задачи из CyberSecEval. Каждая задача содержит промпт из оригинального датасета, номер целевой уязвимости и метаинформацию об источнике задачи, а также ее порядковый номер в результирующем датасете. В общей сложности датасет содержит 403 задачи. Эксперименты с языковыми моделями проводились как на всем итоговом датасете, так и на двух его частях по отдельности.

Для анализа безопасности сгенерированных решений были использованы два статических анализатора: Bandit [16] и Semgrep [17]. Запуск Bandit осуществлялся с параметрами -r <task_dir> -f json --exit-zero -a vuln. Для запуска Semgrep использовались открытые наборы правил p/security-audit, p/python, p/gitlab, auto и p/trailofbits. Анализатор CodeQL [18] намеренно не был использован в экспериментах, поскольку часть задач SecurityEval [27] основана на примерах из документации этого анализатора.

В рамках проведенных экспериментов для каждой из трех используемых моделей (GPT OSS 20B [33], GPT OSS 120B [33], Qwen3-Coder 480B [34]) вычислены метрики корректности и безопасности кода, сгенерированного в мультиагентном (multi) и одноагентном (single) режимах, и агрегированы по каждому из двух оригинальных датасетов (SecurityEval [27], CyberSecEval [28]) и по всем имеющимся задачам. В табл. 2 представлены значения метрик корректности

кода для датасета SecuiryEval, в табл. 3 – для датасета CyberSecEval, а в табл. 4 – на всем датасете. Для модели Qwen3-Coder 480B метрики мультипрограммного режима на данных CyberSecEval не определены, поскольку запуск модели на этих задачах не проводился из-за высоких требований модели к вычислительным ресурсам. Метрики корректности вычислены на основе тестов, сгенерированных языковыми моделями в процессе решения задачи.

Табл. 2. Метрики корректности сгенерированного кода на датасете SecurityEval.

Модель	Режим	Решено задач (из 121)	Решено задач с тестами	Pass@1	Success Rate
GPT OSS 20B	single	121	0	–	–
	multi	121	23	**0.39**	**0.43**
GPT OSS 120B	single	121	0	–	–
	multi	121	21	0.00	0.08
Qwen3-Coder 480B	single	121	23	0.30	0.30
	multi	121	**81**	0.28	0.34

Табл. 3. Метрики корректности сгенерированного кода на датасете CyberSecEval.

Модель	Режим	Решено задач (из 282)	Решено задач с тестами	Pass@1	Success Rate
GPT OSS 20B	single	281	6	0.00	0.00
	multi	281	**56**	**0.54**	**0.62**
GPT OSS 120B	single	280	2	0.00	0.00
	multi	197	32	0.41	0.57
Qwen3-Coder 480B	single	282	17	0.41	0.45

Табл. 4. Метрики корректности сгенерированного кода на всем датасете.

Модель	Режим	Решено задач (из 403)	Задач с тестами	Pass@1	Success Rate
GPT OSS 20B	single	402	6	0.00	0.00
	multi	402	**75**	**0.47**	**0.55**
GPT OSS 120B	single	401	2	0.00	0.00
	multi	318	49	0.22	0.36
Qwen3-Coder 480B	single	403	39	0.36	0.38

Анализ полученных метрик приводит к следующим выводам.

• Все тестируемые модели редко успешно генерируют тесты, в том числе в мультиагентном режиме. Это вызвано непоследовательным и некорректным использованием инструментов, в частности функций read_files и write_file для работы с файловой системой, что приводило к пропуску генерации кода в целом. Кроме того, большинство ошибок в тестах оказались синтаксическими или связанными с некорректным подключением соседних модулей: такие ошибки на практике относительно легко исправить вручную, что позволяет использовать эти модели в интерактивном режиме для решения задач разработки.

• Все модели сумели сгенерировать код для всех задач SecurityEval, тогда как для CyberSecEval около 40% задач не получили решения от модели GPT OSS 120B в мультиагентном режиме.

• Моделям редко удается написать код, проходящий сгенерированные тесты. В первую очередь это справедливо для модели GPT OSS 120B, ни одно решение которой в мультиагентном режиме для задач из SecurityEval не прошло все тесты (Pass@1 = 0). Важно отметить, что полученные значения Pass@1 и Success Rate усреднены по множеству задач, к которым сгенерированы синтаксически корректные тесты, а таких задач оказалось мало для всех моделей и в обоих агентных режимах, что делает выводы о корректности генерируемого кода менее достоверными.

• Модель GPT OSS 20B показывает лучшие результаты, чем GPT OSS 120B, на всех датасетах и в обоих режимах запуска. GPT OSS 20B в мультиагентном

режиме также справляется лучше, чем Qwen3-Coder 480B в одноагентном, при этом сильно уступая этой модели в одноагентном режиме, что дает возможность сделать вывод о том, что построенная мультиагентная система повышает качество работы относительно небольших языковых моделей и может превзойти более крупные и ресурсоемкие нейросети.

• Все модели в среднем справились лучше с задачами из CyberSecEval, содержащими промпты на естественном языке, по сравнению с задачами из SecurityEval, представленными в виде кодовых промптов на Python. Однако не все задачи из CyberSecEval удалось решить в целом, что указывает на больший разброс сложности задач в этом датасете по сравнению с SecurityEval. Отметим, что модель Qwen3-Coder 480B в одноагентном режиме сгенерировала код для всех задач из обоих датасетов, а также показала положительные метрики корректности этого кода относительно сгенерированных тестов, хотя генерация тестов моделями в одноагентном режиме системным промптом не предполагалась.

В табл. 5–7 представлены метрики безопасности кода, полученные с помощью анализатора Bandit, а в табл. 8–10 – метрики на основе ответов анализатора Semgrep.

Табл. 5. Метрики безопасности сгенерированного кода, полученные анализатором Bandit на датасете SecurityEval.

Модель	Режим	Решено задач (из 121)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	121	0.72	**0.49**	**0.91**
	multi	121	0.69	0.45	0.90
GPT OSS 120B	single	121	**0.61**	**0.49**	0.90
	multi	121	0.78	0.38	**0.91**
Qwen3-Coder 480B	single	121	0.65	0.48	0.89
	multi	121	0.69	0.40	0.88

Табл. 6. Метрики безопасности сгенерированного кода, полученные анализатором Bandit на датасете CyberSecEval.

Модель	Режим	Решено задач (из 282)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	281	**0.62**	**0.48**	**0.96**
	multi	281	0.75	0.39	**0.96**
GPT OSS 120B	single	280	0.63	0.46	**0.96**
	multi	197	0.79	0.32	0.95
Qwen3-Coder 480B	single	282	0.66	0.44	0.95

Табл. 7. Метрики безопасности сгенерированного кода, полученные анализатором Bandit на всем датасете.

Модель	Режим	Решено задач (из 403)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	402	0.65	**0.48**	**0.95**
	multi	402	0.73	0.41	0.94
GPT OSS 120B	single	401	**0.63**	0.47	0.94
	multi	318	0.79	0.34	0.93
Qwen3-Coder 480B	single	403	0.66	0.45	0.94

Табл. 8. Метрики безопасности сгенерированного кода, полученные анализатором Semgrep на датасете SecurityEval.

Модель	Режим	Решено задач (из 121)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	121	0.90	0.43	0.83
	multi	121	0.74	0.51	**0.86**
GPT OSS 120B	single	121	**0.73**	0.50	0.85
	multi	121	0.79	0.46	0.83
Qwen3-Coder 480B	single	121	0.85	**0.52**	0.81
	multi	121	0.80	0.45	0.81

Табл 9. Метрики безопасности сгенерированного кода, полученные анализатором Semgrep на датасете CyberSecEval.

Модель	Режим	Решено задач (из 282)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	281	**0.65**	0.47	**0.96**
	multi	281	**0.65**	**0.48**	**0.96**
GPT OSS 120B	single	280	**0.65**	**0.48**	**0.96**
	multi	197	0.68	0.44	0.95
Qwen3-Coder 480B	single	282	0.67	0.46	0.95

Табл. 10. Метрики безопасности сгенерированного кода, полученные анализатором Semgrep на всем датасете.

Модель	Режим	Решено задач (из 403)	Среднее число уязвимостей	Secure@1	Secure@1 с учетом целевой уязвимости
GPT OSS 20B	single	402	0.73	0.46	0.92
	multi	402	0.68	**0.49**	**0.93**
GPT OSS 120B	single	401	**0.67**	**0.49**	**0.93**
	multi	318	0.72	0.45	0.90
Qwen3-Coder 480B	single	403	0.73	0.48	0.91

Из полученных результатов выявлены следующие закономерности.

• Значения Target-Secure@1 (Secure@1 с учетом целевой уязвимости) почти максимальны для всех моделей, режимов запуска, датасетов и статических анализаторов кода, что означает общую способность тестируемых моделей генерировать код, не содержащий распространенных ошибок с точки зрения безопасности решения. При этом значения общей метрики Secure@1, учитывающей наличие любых уязвимостей, в том числе не указанных в условии задачи, оказались ниже 0.5 во всех случаях. Основной причиной таких результатов могут стать ошибочные срабатывания анализаторов, вносящие шум в значения Secure@1, изменяя их в обе стороны.

• В среднем каждое сгенерированное моделями решение содержит менее одной уязвимости. Однако в половине случаев код содержит уязвимость, не заявленную в условии задачи, что может свидетельствовать как о шумности анализаторов, так и о нестабильности языковых моделей относительно генерации безопасных решений.

• Метрики безопасности почти не зависят от конкретной языковой модели. При этом в мультиагентном режиме метрики немного ниже, что вызвано большим объемом сгенерированного кода по сравнению с одноагентным ре-

жимом работы и повышенной частотой срабатываний статических анализаторов.

- Анализатор Semgrep показал себя как более шумный и строгий, что особенно отчетливо видно на датасете SecurityEval.

Табл. 11 содержит среднее время работы каждой модели при генерации решения для одной задачи каждого датасета в обоих режимах запуска. Из полученных данных можно заметить, что при переходе от одноагентного режима к мультиагентному на решение одной задачи из SecurityEval уходит в 2-3 раза больше времени, а на решение задачи из CyberSecEval – в 4-5 раз больше. Для модели Qwen3-Coder 480B в одноагентном режиме требуется в 2-2.5 больше времени, чем для моделей GPT OSS в мультиагентном режиме, и в 5-10 раз больше, чем для GPT OSS в одноагентном режиме. При этом число активных весов на один входной токен у модели Qwen3-Coder 480B в 7-10 раз превышает аналогичное значение для остальных моделей. Сами модели GPT OSS различаются между собой в 6 раз по числу весов и в 1.5 раза – по числу активных весов, но по времени работы они почти одинаковы. В целом модель GPT OSS 20B в мультиагентном режиме позволяет добиться более высокого качества работы, чем Qwen3-Coder 480B в одноагентном режиме, при этом используя в 3-3.5 раза меньше времени на решение одной задачи. Также полученные значения позволяют предсказывать время, необходимое Qwen3-Coder 480B на решение одной задачи в мультиагентном режиме: предположительно на это уйдёт в 3.5-4 раза больше времени, чем для одноагентного режима, на датасете CyberSecEval.

Табл. 11. Среднее время работы моделей для одной задачи каждого датасета в секундах.

Модель	Режим	SecurityEval	CyberSecEval	Весь датасет
GPT OSS 20B	single	16.74	14.55	15.21
	multi	46.95	79.13	69.44
GPT OSS 120B	single	17.83	16.00	16.56
	multi	46.69	61.98	56.16

Qwen3-Coder 480B	single	87.69	154.27	134.28
	multi	183.82	–	–

Списки наиболее популярных уязвимостей, обнаруженных статическими анализаторами Bandit и Semgrep в программах, сгенерированных всеми моделями в обоих режимах и на всех датасетах, приведены в табл. 12 и 13.

Табл. 12. Наиболее популярные уязвимости, обнаруженные анализатором Bandit в коде моделей на всех датасетах.

Модель	Режим	SecurityEval	CyberSecEval	Весь датасет
GPT OSS 20B	single	CWE-605 CWE-94 CWE-20	CWE-78 CWE-703 CWE-330	CWE-78 CWE-703 CWE-605
	multi	CWE-259 CWE-605 CWE-703	CWE-78 CWE-259 CWE-703	CWE-78 CWE-259 CWE-703
GPT OSS 120B	single	CWE-605 CWE-20 CWE-78	CWE-78 CWE-703 CWE-330	CWE-78 CWE-703 CWE-605
	multi	CWE-259 CWE-605 CWE-20	CWE-78 CWE-259 CWE-502	CWE-259 CWE-78 CWE-605
Qwen3-Coder 480B	single	CWE-259 CWE-327 CWE-605	CWE-78 CWE-330 CWE-502	CWE-78 CWE-259 CWE-330
	multi	CWE-259 CWE-94 CWE-327	–	–

Табл. 13. Наиболее популярные уязвимости, обнаруженные анализатором Semgrep в коде моделей на всех датасетах.

Модель	Режим	SecurityEval	CyberSecEval	Весь дата-сет
GPT OSS 20B	single	CWE-489 CWE-668 CWE-611	CWE-78 CWE-89 CWE-502	CWE-78 CWE-489 CWE-502
	multi	CWE-611 CWE-668 CWE-502	CWE-78 CWE-502 CWE-754	CWE-78 CWE-502 CWE-611
GPT OSS 120B	single	CWE-668 CWE-611 CWE-79	CWE-78 CWE-95 CWE-754	CWE-78 CWE-95 CWE-502
	multi	CWE-668 CWE-611 CWE-502	CWE-78 CWE-327 CWE-502	CWE-78 CWE-502 CWE-611
Qwen3-Coder 480B	single	CWE-611 CWE-489 CWE-79	CWE-78 CWE-89 CWE-95	CWE-207 CWE-89 CWE-502
	multi	CWE-611 CWE-489 CWE-502	–	–

В табл. 14 указано число вхождений наиболее популярных уязвимостей в программах, сгенерированных всеми моделями, для обоих анализаторов.

Табл. 14. Наиболее популярные уязвимости, обнаруженные анализаторами Bandit и Semgrep в коде моделей на всех датасетах.

Bandit	Semgrep
CWE-78: 1986	CWE-78: 1190
CWE-259: 969	CWE-502: 627
CWE-703: 723	CWE-611: 512
CWE-502: 460	CWE-95: 477
CWE-605: 389	CWE-89: 474

Рассмотрим наиболее популярные уязвимости и закономерности их обнаружения в сгенерированном коде.

- CWE-78: инъекция команд операционной системы. В основном эта уязвимость обнаружена на задачах из датасета CyberSecEval, что объясняется условиями задач, подталкивающих модель к генерации кода, исполняющего команды операционной системы с входными данными от пользователя без должной валидации.
- CWE-259: использование явно указанных паролей в тексте программы. Эта уязвимость чаще обнаружена анализатором Bandit и только в задачах, где сгенерированы тесты (для всех моделей в мультиагентном режиме и для модели Qwen3-Coder 480B в одноагентном режиме).
- CWE-502: десериализация недоверенных данных. Преимущественно выявлена в решениях задач из CyberSecEval, но иногда анализатор Semgrep указывал на наличие CWE-502 и в решениях SecurityEval в мультиагентном режиме.
- CWE-703: неверная обработка исключительных условий. Обнаружена только анализатором Bandit и только для решений моделей GPT OSS, в основном на задачах CyberSecEval.
- CWE-611: некорректная нейтрализация специальных элементов в XML-сущности (англ. XXE, XML External Entity). Выявлена анализатором Semgrep на решениях SecurityEval.

• CWE-95: некорректная нейтрализация директив в динамически вычисляемом коде (англ. Eval Injection). Найдена анализатором Semgrep для решений задач из CyberSecEval, полученных моделями GPT OSS 120B и Qwen3-Coder 480B в одноагентном режиме.

• CWE-89: SQL-инъекция. Найдена анализатором Semgrep для решений задач из CyberSecEval, полученных моделями GPT OSS 20B и Qwen3-Coder 480B в одноагентном режиме.

• CWE-605: множественные привязки к одному порту. Обнаружены анализатором Bandit в коде, сгенерированного всеми моделями для решения задач SecurityEval.

Значительная часть уязвимостей выявлена именно в коде тестов, причем они редко могут считаться релевантными на практике. К примеру, десериализация недоверенных данных (CWE-502) и использование явно указанных паролей в тексте программы (CWE-259) являются частыми паттернами написания модульных тестов. Множества уязвимостей, обнаруженных двумя анализаторами, в большинстве случаев также различаются, за исключением CWE-78 и CWE-502, что подтверждает необходимость использования нескольких различных инструментов для разностороннего анализа кода. Более того, статические анализаторы кода, основанные на правилах, могут быть очень «шумными», например, указывая на наличие уязвимости CWE-95 в любом коде на Python, вызывающем функцию eval. Напротив, анализатор CodeQL учитывает граф вызовов и графы потока управления и данных в программе, что теоретически снижает долю ложных срабатываний, однако в настоящем исследовании применение этого анализатора исключено. Многие обнаруженные уязвимости вызваны неполными условиями задач в используемых датасетах, при этом общих требований безопасности генерируемого кода, указанных в системном промпте моделей, оказывается недостаточно.

ОГРАНИЧЕНИЯ ПОДХОДА

Полученные результаты обладают рядом ограничений. Прежде всего были зафиксированы язык программирования Python и фреймворк unittest для модульного тестирования, поэтому полученные выводы сложно обобщить на другие языки и фреймворки. Большая доля рассмотренных задач решена вы-

бранными языковыми моделями без генерации тестов, поскольку самыми частыми ошибками агентов являлись игнорирование или некорректное применение инструментов работы с файловой системой и вызова других агентов. Ряд моделей, помимо упомянутых в работе, показал неудовлетворительные результаты экспериментов, поскольку испытывали проблемы при работе с инструментами и системными промптами в пределах относительно короткого контекстного окна.

Использование отдельной языковой модели для формулирования промптов для каждого агента и каждой тестируемой модели позволило бы учесть их различия, заложенные на этапе обучения, и повысить точность использования инструментов. Кроме того, реализация отдельного агента для исполнения сгенерированного кода в тестовой среде может повысить качество решений в обмен на рост стоимости генерации кода. Наконец, подключение дополнительных моделей для анализа входных и выходных данных агентов, так называемых барьеров (англ. guardrails), поспособствует генерации более безопасных решений или отказу в обработке пользовательского запроса, в случае если он изначально подразумевает генерацию вредоносной программы.

ЗАКЛЮЧЕНИЕ

Исследована зависимость корректности и безопасности кода, генерируемого большими языковыми моделями GPT-OSS 20B, GPT-OSS 120B и Qwen3-Coder 480B для решения задач из датасетов SecurityEval и CyberSecEval, от режима работы этих моделей. Для проведения экспериментов построен масштабируемый фреймворк SafeAICoder, предоставляющий возможность распределенной обработки запросов генерации кода и тестов на сервере. Полученные результаты демонстрируют выигрыш от использования мультиагентного режима работы меньших языковых моделей по сравнению с более ресурсоемкими как по времени генерации, так и по качеству генерируемых программ. Сделаны выводы о необходимости реализации тестовой среды для запуска генерируемых решений, а также применения более точных системных промптов и инструментов анализа кода для повышения стабильности системы и качества программ.

Благодарности

Работа выполнена в рамках Государственного задания МГУ имени М. В. Ломоносова.

СПИСОК ЛИТЕРАТУРЫ

1. *Hou X., Zhao Y., Liu Y. et al.* Large Language Models for Software Engineering: A Systematic Literature Review // arXiv preprint. 2013. arXiv:2308.10620.

2. *Ma L., Liu S., Li Y. et al.* SpecGen: Automated Generation of Formal Program Specifications via Large Language Models // arXiv preprint. 2024. arXiv:2401.08807.

3. *Mandal S. et al.* Large Language Models Based Automatic Synthesis of Software Specifications // arXiv preprint. 2023. arXiv:2304.09181.

4. *Cassano F., Gouwar J., Nguyen D. et al.* MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation // IEEE Transactions on Software Engineering. 2023. Vol. 49, No. 7. P. 3675–3691.

5. *Chen F., Fard F. H., Lo D., Bryksin T*. On the Transferability of Pre-trained Language Models for Low-resource Programming Languages // IEEE/ACM 30th International Conference on Program Comprehension (ICPC). 2022. P. 401–412.

6. *Fan G., Chen S., Gao C. et al.* Rapid: Zero-shot Domain Adaptation for Code Search with Pre-trained Models // ACM Transactions on Software Engineering and Methodology. 2024. Vol. 33, No. 5. P. 1–35.

7. *Schäfer M., Nadi S., Eghbali A., Tip F*. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation // IEEE Transactions on Software Engineering. 2023. Vol. 50, No. 1. P. 85–105.

8. *Nappa A., Johnson R., Bilge L. et al*. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching // IEEE symposium on security and privacy. 2015. P. 692–708.

9. *Ardito L., Coppola R., Barbato L., Verga D*. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review // Scientific Programming 2020. P. 1–26.

10. *Buse R. P. L., Weimer W. R*. Learning a Metric for Code Reliability // IEEE Transactions on Software Engineering. 2010. Vol. 36, No. 4. P. 546–558.

11. *Peitek N., Apel S., Parmin C. et al*. Program Comprehension and Code Complexity Metrics: An fMRI Study // IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021. P. 524–536.

12. *Markovtsev V., Long W., Mougard H. et al*. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms // IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). 2019. P. 468–478.

13. *Raemaekers S., Van Deursen A., Visser J*. Measuring software library stability through historical version analysis // 28th IEEE International Conference on Software Maintenance (ICSM). 2012. P. 378–387.

14. Common Weakness Enumeration.
URL: https://cwe.mitre.org (дата обращения: 30.03.2026).

15. Open Web Application Security Project.
URL: https://owasp.org (дата обращения: 30.03.2026).

16. Документация статического анализатора кода Bandit.
URL: https://bandit.readthedocs.io/en/latest (дата обращения: 30.03.2026).

17. Документация статического анализатора кода Semgrep.
URL: https://semgrep.dev/docs (дата обращения: 30.03.2026).

18. Документация статического анализатора кода CodeQL.
URL: https://codeql.github.com/docs (дата обращения: 30.03.2026).

19. Документация статического анализатора кода Bearer.
URL: https://docs.bearer.com (дата обращения: 30.03.2026).

20. Документация статического анализатора кода Snyk.
URL: https://docs.snyk.io (дата обращения: 30.03.2026).

21. *Chen M., Tworek J., Jun H. et al*. Evaluating Large Language Models Trained on Code // arXiv preprint. 2021. arXiv:2107.03374.

22. *Inala J.P., Wang C., Yang M. et al.* Fault-Aware Neural Code Rankers // arXiv preprint. 2022. arXiv:2206.03865.

23. *Lu S. et al.* CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation // arXiv preprint. 2021. arXiv:2102.04664.

24. *Valentin T., Madadi A., Sapia G., Böhme M*. Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation // arXiv preprint. 2025. arXiv:2507.00057.

25. *Spiess C., Gros D., Pai K. S. et al*. Calibration and Correctness of Language Models for Code // arXiv preprint. 2024. arXiv:2402.02047.

26. *Siddiq M.L., da Silva Santos J.C., Devareddy S., Muller A*. Sallm: Security assessment of generated code // Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops. 2024. Pp. 54-65.

27. *Siddiq M.L., da Silva Santos J.C.* SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques // Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S22). 2022.

28. Документация статического анализатора SonarQube.
URL: https://docs.sonarsource.com (дата обращения: 30.03.2026).

29. *Austin J., Odena A., Nye M. et al.* Program Synthesis with Large Language Models // arXiv preprint. 2021. arXiv:2108.07732.

30. *Bhatt M. et al.* Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models // arXiv preprint. 2023. arXiv:2312.04724.

31. База знаний MITRE ATT&CK.
URL: https://attack.mitre.org (дата обращения: 30.03.2026).

32. *Abdelaziz I. et al.* Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track. 2024. P. 1131–1139.

33. *Agarwal S. et al.* gpt-oss-120b & gpt-oss-20b Model Card // arXiv preprint. 2025. arXiv:2508.10925.

34. Yang An et al. Qwen3 Technical Report // arXiv preprint. 2025. arXiv:2505.09388.

35. *Huynh N., Lin B*. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications // arXiv preprint. 2025. arXiv:2503.01245.

36. *Fakhoury S. et al.* LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation // IEEE Transactions on Software Engineering. 2024. Vol. 50, No. 9. P. 2254–2268.

37. *He X. et al.* CoCoST: Automatic Complex Code Generation with Online

Searching and Correctness Testing // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. 2024. P. 19433–19451.

38. *Liu M. et al.* An Empirical Study of the Code Generation of Safety-Critical Software Using LLMs // Applied Sciences. 2024. Vol. 14, No. 3. P. 1046.

39. *Wang C., Zhang J., Feng Y., Li T*. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation // ACM Transactions on Software Engineering and Methodology. 2025. Vol. 34, No. 7. P. 1–27.

40. *Dong Y. et al.* A Survey on Code Generation with LLM-based Agents // arXiv preprint. 2025. arXiv:2508.00083.

41. *Nunez A. et al.* AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing // arXiv preprint. 2024. arXiv:2409.10737.

42. *Islam N.T. et al.* Enhancing Source Code Security with LLMs: Demystifying The Challenges and Generating Reliable Repairs // arXiv preprint. 2024. 2409.00571.

43. *Ishibashi Y., Nishimura Y*. Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization // arXiv preprint. 2024. 2404.02183.

44. *Pan R., Zhang H., Liu C*. CodeCoR: An LLM-Based Self-Reflective Multi-Agent Framework for Code Generation // arXiv preprint. 2025. arXiv:2501.07811.

45. *Holt S., Luyten M.R., van der Schaar M*. L2MAC: Large Language Model Automatic Computer for Extensive Code Generation // arXiv preprint. 2023. 2310.02003.

46. Публичный репозиторий фреймворка SafeAICoder на GitHub.
URL: https://github.com/KarineAyrs/saicoder (дата обращения: 30.03.2026).

47. Документация сервиса Ollama. URL: https://docs.ollama.com (дата обращения: 30.03.2026).

48. *Merkel D*. Docker: lightweight linux containers for consistent development and deployment // Linux journal. 2014. No. 239. P. 2.

49. Документация инструмента Docker Compose.
URL: https://docs.docker.com/reference/cli/docker/compose (дата обращения: 30.03.2026).

50. Официальный Docker-образ клиента Ollama в Docker Hub. URL: https://hub.docker.com/r/ollama/ollama (дата обращения: 30.03.2026).

51. *Blakeman A. et al.* Nemotron 3 Nano: Open, Efficient Mixture-of-Experts Hybrid Mamba-Transformer Model for Agentic Reasoning // arXiv preprint. 2025. arXiv:2512.20848.

52. Модель Devstral 2 123B Instruct 2512 на HuggingFace. URL: https://huggingface.co/mistralai/Devstral-2-123B-Instruct-2512 (дата обращения: 30.03.2026).

A FRAMEWORK FOR SAFETY ANALYSIS OF LLM-GENERATED CODE IN MULTI-AGENT SYSTEMS

D. A. Avagian[1] [0009-0009-2409-7357], K. A. Ayrapetyants[2] [0000-0002-9412-9264]

[1, 2] *Lomonosov Moscow State University, Moscow, Russia*

[1]david_avagyan@list.ru, [2]karine.ayrps@gmail.com

Abstract

Large language models (LLMs) are increasingly being used for code generation. However, both the generated code and the underlying LLM-based systems demand rigorous security evaluation. A common approach to enhancing code generation quality involves multi-agent systems composed of multiple models. This paper evaluates the performance of GPT-OSS 20B, GPT-OSS 120B, and Qwen3-Coder 480B models in single-agent and multi-agent configurations, using two code security benchmarks, SecurityEval and CyberSecEval. The key contribution is SafeAICoder, an extensible and scalable framework for testing LLMs enabling distributed server-side generation of multi-module programs and tests, independent of client-side code.

Keywords*: large language models, multi-agent systems, software engineering, code generation, code quality, code safety, metrics.*

REFERENCES

1. *Hou X., Zhao Y., Liu Y. et al.* Large Language Models for Software Engineering: A Systematic Literature Review // arXiv preprint. 2013. arXiv:2308.10620.

2. *Ma L., Liu S., Li Y. et al.* SpecGen: Automated Generation of Formal Program Specifications via Large Language Models // arXiv preprint. 2024. arXiv:2401.08807.

3. *Mandal S. et al.* Large Language Models Based Automatic Synthesis of Software Specifications // arXiv preprint. 2023. arXiv:2304.09181.

4. *Cassano F., Gouwar J., Nguyen D. et al.* MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation // IEEE Transactions on Software Engineering. 2023. Vol. 49, No. 7. P. 3675–3691.

5. *Chen F., Fard F. H., Lo D., Bryksin T*. On the Transferability of Pre-trained Language Models for Low-resource Programming Languages // IEEE/ACM 30th International Conference on Program Comprehension (ICPC). 2022. P. 401–412.

6. *Fan G., Chen S., Gao C. et al.* Rapid: Zero-shot Domain Adaptation for Code Search with Pre-trained Models // ACM Transactions on Software Engineering and Methodology. 2024. Vol. 33, No. 5. P. 1–35.

7. *Schäfer M., Nadi S., Eghbali A., Tip F*. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation // IEEE Transactions on Software Engineering. 2023. Vol. 50, No. 1. P. 85–105.

8. *Nappa A., Johnson R., Bilge L. et al*. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching // IEEE symposium on security and privacy. 2015. P. 692–708.

9. *Ardito L., Coppola R., Barbato L., Verga D*. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review // Scientific Programming 2020. P. 1–26.

10. *Buse R. P. L., Weimer W. R*. Learning a Metric for Code Reliability // IEEE Transactions on Software Engineering. 2010. Vol. 36, No. 4. P. 546–558.

11. *Peitek N., Apel S., Parmin C. et al*. Program Comprehension and Code Complexity Metrics: An fMRI Study // IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021. P. 524–536.

12. *Markovtsev V., Long W., Mougard H. et al*. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms // IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). 2019. P. 468–478.

13. *Raemaekers S., Van Deursen A., Visser J*. Measuring software library stability through historical version analysis // 28th IEEE International Conference on Software Maintenance (ICSM). 2012. P. 378–387.

14. Common Weakness Enumeration. URL: https://cwe.mitre.org (accessed: 30.03.2026).

15. Open Web Application Security Project. URL: https://owasp.org (accessed: 30.03.2026).

16. SAST Bandit documentation. URL: https://bandit.readthedocs.io/en/latest (accessed: 30.03.2026).

17. SAST Semgrep documentation. URL: https://semgrep.dev/docs (accessed: 30.03.2026).

18. SAST CodeQL documentation. URL: https://codeql.github.com/docs (accessed: 30.03.2026).

19. SAST Bearer documentation. URL: https://docs.bearer.com (accessed: 30.03.2026).

20. SAST Snyk documentation. URL: https://docs.snyk.io (accessed: 30.03.2026).

21. *Chen M., Tworek J., Jun H. et al*. Evaluating Large Language Models Trained on Code // arXiv preprint. 2021. arXiv:2107.03374.

22. *Inala J.P., Wang C., Yang M. et al.* Fault-Aware Neural Code Rankers // arXiv preprint. 2022. arXiv:2206.03865.

23. *Lu S. et al.* CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation // arXiv preprint. 2021. arXiv:2102.04664.

24. *Valentin T., Madadi A., Sapia G., Böhme M*. Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation // arXiv preprint. 2025. arXiv:2507.00057.

25. *Spiess C., Gros D., Pai K. S. et al*. Calibration and Correctness of Language Models for Code // arXiv preprint. 2024. arXiv:2402.02047.

26. *Siddiq M.L., da Silva Santos J.C., Devareddy S., Muller A*. Sallm: Security assessment of generated code // Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops. 2024. P. 54–65.

27. *Siddiq M.L., da Silva Santos J.C*. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques // Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S22). 2022.

28. SAST SonarQube documentation. URL: https://docs.sonarsource.com (accessed: 30.03.2026).

29. *Austin J., Odena A., Nye M. et al.* Program Synthesis with Large Language Models // arXiv preprint. 2021. arXiv:2108.07732.

30. *Bhatt M. et al.* Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models // arXiv preprint. 2023. arXiv:2312.04724.

31. MITRE ATT&CK knowledge base. URL: https://attack.mitre.org (дата обращения: 30.03.2026).

32. *Abdelaziz I. et al.* Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track. 2024. P. 1131–1139.

33. *Agarwal S. et al.* gpt-oss-120b & gpt-oss-20b Model Card // arXiv preprint. 2025. arXiv:2508.10925.

34. *Yang An et al.* Qwen3 Technical Report // arXiv preprint. 2025. arXiv:2505.09388.

35. *Huynh N., Lin B*. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications // arXiv preprint. 2025. arXiv:2503.01245.

36. *Fakhoury S. et al.* LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation // IEEE Transactions on Software Engineering. 2024. Vol. 50, No. 9. P. 2254–2268.

37. *He X. et al.* CoCoST: Automatic Complex Code Generation with Online Searching and Correctness Testing // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. 2024. P. 19433–19451.

38. *Liu M. et al.* An Empirical Study of the Code Generation of Safety-Critical Software Using LLMs // Applied Sciences. 2024. Vol. 14, No. 3. P. 1046.

39. *Wang C., Zhang J., Feng Y., Li T*. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation // ACM Transactions on Software Engineering and Methodology. 2025. Vol. 34, No. 7. P. 1–27.

40. *Dong Y. et al.* A Survey on Code Generation with LLM-based Agents // arXiv preprint. 2025. arXiv:2508.00083.

41. *Nunez A. et al.* AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing // arXiv preprint. 2024. arXiv:2409.10737.

42. *Islam N.T. et al.* Enhancing Source Code Security with LLMs: Demystifying The Challenges and Generating Reliable Repairs // arXiv preprint. 2024. 2409.00571.

43. *Ishibashi Y., Nishimura Y*. Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization // arXiv preprint. 2024. 2404.02183.

44. *Pan R., Zhang H., Liu C*. CodeCoR: An LLM-Based Self-Reflective Multi-Agent Framework for Code Generation // arXiv preprint. 2025. arXiv:2501.07811.

45. *Holt S., Luyten M.R., van der Schaar M*. L2MAC: Large Language Model Automatic Computer for Extensive Code Generation // arXiv preprint. 2023. 2310.02003.

46. GitHub repository for SafeAICoder framework.
URL: https://github.com/KarineAyrs/saicoder (accessed: 30.03.2026).

47. Ollama documentation. URL: https://docs.ollama.com (accessed: 30.03.2026).

48. *Merkel D*. Docker: lightweight linux containers for consistent development and deployment // Linux journal. 2014. No. 239. P. 2.

49. Docker Compose tool documentation.
URL: https://docs.docker.com/reference/cli/docker/compose (accessed: 30.03.2026).

50. Official Ollama client Docker image on Docker Hub.
URL: https://hub.docker.com/r/ollama/ollama (accessed: 30.03.2026).

51. *Blakeman A. et al*. Nemotron 3 Nano: Open, Efficient Mixture-of-Experts Hybrid Mamba-Transformer Model for Agentic Reasoning // arXiv preprint. 2025. arXiv:2512.20848.

52. Devstral 2 123B Instruct 2512 model card on HuggingFace. URL: https://huggingface.co/mistralai/Devstral-2-123B-Instruct-2512 (accessed: 30.03.2026).

СВЕДЕНИЯ ОБ АВТОРАХ

АВАГЯН Давид Арменович – аспирант кафедры алгоритмических языков факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова. Сфера научных интересов: нейросетевая генерация кода, метрики качества программ.

David Armenovich AVAGIAN – postgraduate student at the department of Algorithmic Languages, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University. Research interests: code generation with neural networks, code quality metrics.

email: david_avagyan@list.ru

ORCID: 0009-0009-2409-7357

АЙРАПЕТЬЯНЦ Каринэ Арсеновна – аспирант кафедры информационной безопасности факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова. Сфера научных интересов: безопасная агентная среда, безопасность систем искусственного интеллекта.

Karine Arsenovna AYRAPETYANTS – postgraduate student at the Department of Information Security, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University. Research interests: secure agent environments, AI systems safety.

email: karine.ayrps@gmail.com

ORCID: 0000-0002-9412-9264

Материал поступил в редакцию 23 апреля 2026 года