DOI: 10.15514/ISPRAS-2026-38(5)-5

JavaCapsule: Итеративная генерация и отладка Java-кода на основе структурированной обратной связи

[1] Д.В. Александров, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>
[1] В.И. Василевский, ORCID: 0009-0004-0115-7082 <vvasilevskiy@hse.ru>
[1] Л.А. Резуник, ORCID: 0009-0000-9428-4718 <lrezunik@hse.ru>
[1] Л.А. Кулигин, ORCID: 0009-0005-7892-9711 <lkuligin@hse.ru>
[1] А.В. Манушкина, ORCID: 0009-0009-5387-3546 <amanushkina@hse.ru>
[1] Н.А. Думкин, ORCID: 0009-0006-4280-8253 <ndumkin@hse.ru>
[1] М.А. Прозорский, ORCID: 0009-0006-9250-7280 <mprozorskiy@hse.ru>
[2] К.Ю. Пинигин, ORCID: 0009-0005-2106-5633 <k.pinigin@innopolis.university>
[1] Научно-учебная лаборатория облачных и мобильных технологий,
Национальный исследовательский университет "Высшая школа экономики",
Россия, 109028, г. Москва, Покровский бульвар, д. 11.
[2] Университет Иннополис,
Россия, 420500, Республика Татарстан, г. Иннополис, Университетская ул., д. 1.

Аннотация. Ограничения при выполнении задач, требующих точного извлечения информации из сверхдлинных последовательностей, приводят к снижению качества генерации и отладки кода крупных объектно-ориентированных систем, характеризующихся сложными межклассовыми зависимостями и необходимостью анализа длинных трассировок выполнения, превышающих размеры стандартных контекстных окон. Предлагается технология JavaCapsule для генерации и отладки Java-кода, реализующая итеративную отладку на основе структурированной обратной связи, включающей результаты выполнения тестов и трассировки выполнения. Для обработки сверхдлинного контекста исследуется Ассоциативный рекуррентный трансформер с памятью (Associative Recurrent Memory Transformer, ARMT), демонстрирующий точность извлечения информации до 99% на последовательностях длиной до 1 миллиона токенов. Для корректной оценки разработан JavaBench+ – модернизированный набор задач с устраненными дефектами, предоставляющий расширенный цикл обратной связи. Экспериментальные результаты показывают, что JavaCapsule обеспечивает увеличение качества генерации на 20 процентов по метрике Pass@1 по сравнению с базовой моделью Google Gemma-3-27B-it, а исследование архитектуры ARMT подтверждает возможность эффективной обработки сверхдлинных последовательностей.

Ключевые слова: генерация кода; интеллектуальная отладка; модели с длинным контекстом; трансформеры с расширенной памятью; ассоциативная память; рекуррентный трансформер ARMT; язык программирования Java; большие языковые модели.

Для цитирования: Александров Д.В., Василевский В.И., Резуник Л.А., Кулигин Л.А., Манушкина А.В., Думкин Н.А., Прозорский М.А., Пинигин К.Ю. JavaCapsule: Итеративная генерация и отладка Java-кода на основе структурированной обратной связи. Труды ИСП РАН, том 38, вып. 5, 2026 г., стр. 73–90. DOI: 10.15514/ISPRAS–2026–38(5)–5.

Благодарности: Исследование выполнено с использованием суперкомпьютерного комплекса НИУ ВШЭ.

JavaCapsule: Iterative Java Code Generation and Debugging Based on Structured Feedback

[1] D.V. Alexandrov, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>
[1] V.I. Vasilevskiy, ORCID: 0009-0004-0115-7082 <vvasilevskiy@hse.ru>
[1] L.A. Rezunik, ORCID: 0009-0000-9428-4718 <lrezunik@hse.ru>
[1] L.A. Kuligin, ORCID: 0009-0005-7892-9711 <lkuligin@hse.ru>
[1] A.V. Manushkina, ORCID: 0009-0009-5387-3546 <amanushkina@hse.ru>
[1] N.A. Dumkin, ORCID: 0009-0006-4280-8253 <ndumkin@hse.ru>
[1] M.A. Prozorskiy, ORCID: ORCID: 0009-0006-9250-7280 <mprozorskiy@hse.ru>
[2] K.Y. Pinigin, ORCID: 0009-0005-2106-5633 <k.pinigin@innopolis.university>
[1] Scientific and Educational Laboratory of Cloud and Mobile Technologies,
Higher School of Economics (HSE),
11, Pokrovsky Boulevard, Moscow, 109028, Russia.
[2] Innopolis University,
1, Universitetskaya St., Innopolis, Republic of Tatarstan, 420500, Russia.

Abstract. Large language models demonstrate limitations in tasks that require precise information retrieval from very long input sequences. This limitation affects code generation in large object-oriented systems, where correctness depends on complex inter-class dependencies and the analysis of long execution traces that exceed standard context window sizes. This paper presents JavaCapsule, a technology for automated generation and iterative debugging of Java code based on structured feedback. The approach incorporates unit test results and execution traces into an iterative refinement loop. For reliable evaluation, JavaBench+ is introduced as an improved benchmark with corrected structural inconsistencies and extended feedback that includes access to test specifications and execution logs. In addition, the Associative Recurrent Memory Transformer (ARMT) is investigated as a mechanism for accurate information retrieval from long sequences. The study is based on controlled experiments using JavaBench+ and comparative evaluation with a strong baseline model. The results show that JavaCapsule improves Pass@1 by 20 percentage points compared to Google Gemma-3-27B-it on medium-complexity tasks and demonstrates consistent improvements on larger projects. In addition, experiments with ARMT demonstrate accurate information retrieval from sequences of up to one million tokens. The obtained results characterize the capabilities of iterative debugging and long-context processing as two complementary components of the proposed approach.

Keywords: code generation; intelligent debugging; long-context models; memory-augmented transformers; ARMT; java; large language models.

For citation: Alexandrov D.V., Vasilevskiy V.I., Rezunik L.A., Kuligin L.A., Manushkina A.V., Dumkin N.A., Prozorskiy M.A., Pinigin K.Y. JavaCapsule: Iterative Java Code Generation and Debugging Based on Structured Feedback. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 5, 2026, pp. 73-90 (in Russian). DOI: 10.15514/ISPRAS-2026-38(5)-5.

Acknowledgements: This research was supported in part through computational resources of HPC facilities at HSE University.

1. Введение

Задача автоматической генерации и отладки программного кода с использованием больших языковых моделей становится все более востребованной, однако ее применение в промышленной разработке остается ограниченным. Основная трудность связана с необходимостью учитывать глобальный контекст программной системы, включающий взаимосвязи между компонентами, особенности типизации и поведение программы в процессе выполнения.

В объектно-ориентированных языках программирования, таких как Java, корректность программы определяется не только логикой отдельных фрагментов кода, но и сложной структурой межклассовых зависимостей. Диагностика ошибок в таких системах требует анализа больших объемов данных, включая трассировки выполнения и журналы сборки, значительно превышающих стандартные размеры контекстных окон больших языковых моделей.

Классические трансформеры обеспечивают высокое качество генерации кода, однако большинство таких архитектур основано на механизме полного внимания, что ограничивает их применение в задачах с длинными входными последовательностями. Альтернативные подходы, использующие компактные внутренние представления последовательности, такие как модели пространства состояний (State Space Models, SSMs), демонстрируют лучшую масштабируемость, но имеют ограничения при обращении к конкретным фактам, необходимым для анализа программных ошибок.

В данной работе рассматривается задача генерации и отладки Java-кода в длинном контексте. Предлагаемый подход основан на разделении функций генерации и анализа: синтез кода осуществляется с использованием языковой модели, тогда как обработка длинных последовательностей и извлечение информации выполняются специализированным модулем памяти. Для реализации этого подхода разработана методология JavaCapsule, использующая итеративный процесс уточнения кода с опорой на результаты выполнения тестов и структурированные данные об ошибках.

Для проведения экспериментов сформирован обновленный набор задач JavaBench+, в котором устранены несогласованности исходных данных и обеспечен доступ к дополнительной информации о тестах и результатах их выполнения.

На основе проведенного анализа рассматривается возможность построения гибридной архитектуры, объединяющей генеративную модель для синтеза программного кода и модуль памяти для обработки длинных трассировок выполнения и кодовых баз. Такое разделение позволяет учитывать как требования к качеству генерации, так и необходимость точного анализа информации в условиях ограничений контекстного окна.

Основные результаты работы включают повышение качества генерации Java-кода и демонстрацию возможности обработки контекста, существенно превышающего стандартные ограничения языковых моделей, что подтверждает применимость предложенного подхода к задачам генерации и анализа программного кода в условиях длинного контекста.

2. Адаптация наборов оценочных задач для генерации и отладки кода в длинном контексте

Корректная оценка способности моделей решать задачи генерации кода в объектно-ориентированных языках требует использования наборов оценочных задач, отражающих структуру реальных программных систем. Задачи, ограниченные простыми сценариями ввода-вывода, не позволяют выявить способность моделей учитывать межклассовые зависимости, а также анализировать поведение программы на основе трассировок выполнения и журналов сборки. В связи с этим в работе используется набор взаимодополняющих инструментов, обеспечивающих оценку как генерации кода, так и обработки длинного контекста.

2.1 Модификация тестового набора JavaBench

В качестве основного предметно-ориентированного инструмента был выбран тестовый набор JavaBench [1], ориентированный на задачи генерации кода в рамках Java-проектов с реалистичной структурой взаимодействия и зависимостей классов. Однако детальный анализ выявил ряд систематических структурных дефектов, влияющих на корректность оценки:

- **Несогласованность условий задачи и тестовой среды**

  В ряде задач наблюдается расхождение между интерфейсом, предоставленным модели, и фактическими требованиями тестов (рис. 1). Например, элементы, объявленные как доступные извне, в тестовой среде используются с более строгими ограничениями доступа. В результате корректная с точки зрения спецификации реализация может приводить к ошибкам компиляции, связанным с нарушением правил доступа.

```java
/**
 * Map of the game.
 */
public class Map {

    private final int rows;
    private final int cols;
    public final Cell[][] cells;
```

Контекст
Context

```java
/**
 * Map of the game.
 */
public class Map {

    private final int rows;
    private final int cols;
    @NotNull
    final Cell[][] cells;
```

Тестовые данные
Test

Рис. 1. Пример структурного несоответствия в JavaBench.
Fig. 1. Example of a structural inconsistency in JavaBench.

- **Неполнота функциональных спецификаций**

  Javadoc-описания методов не всегда отражают полный набор требований, проверяемых тестами (рис. 2). В частности, часть логики, связанная с изменением внутреннего состояния системы или побочными эффектами, остается неявной. Это приводит к ситуации, при которой модель вынуждена восстанавливать поведение программы на основе косвенных признаков, не имея доступа к полной спецификации.

```java
/**
 * Processes a Move action performed by the player.
 *
 * @param direction The direction the player wants to move to.
 * @return An instance pf {@link MoveResult} indicating the result of the action.
 */
public MoveResult processMove(@NotNull final Direction direction) {
    Objects.requireNonNull(direction);

    final var result = this.gameState.getGameBoardController().makeMove(direction);

    if (result instanceof MoveResult.Valid v) {
        this.gameState.incrementNumMoves();

        if (v instanceof MoveResult.Valid.Alive va) {
            this.gameState.increaseNumLives(va.collectedExtraLives.size());

            this.gameState.getMoveStack().push(va);
        } else if (v instanceof MoveResult.Valid.Dead) {
            this.gameState.increaseNumDeaths();
            this.gameState.decrementNumLives();
        }
    }

    return result;
}
```

Рис. 2. Пример скрытых требований к побочным эффектам в тестовых данных JavaBench.
Fig. 2. Example of hidden side effect requirements in JavaBench test data.

- **Ограниченность стратегий формирования контекста**

  Используемые в исходном тестовом наборе подходы к отбору контекста не обеспечивают охват всех зависимостей в задачах с большим числом взаимосвязанных классов (рис. 3). При передаче лишь части кодовой базы модели недоступна информация, необходимая для корректного воспроизведения логики взаимодействия компонентов.

Для устранения указанных проблем был разработан расширенный вариант JavaBench – JavaBench+. В рамках данной версии выполнена полная переработка исходных кодов, позволившая устранить противоречия и обеспечить согласованность между описанием задач и условиями их проверки. Дополнительно реализована поддержка передачи всей кодовой базы проекта в контекстное окно, тела непройденных тестов и полной трассировки стека ошибок, что позволяет использовать результаты тестирования как источник формализованной обратной связи.

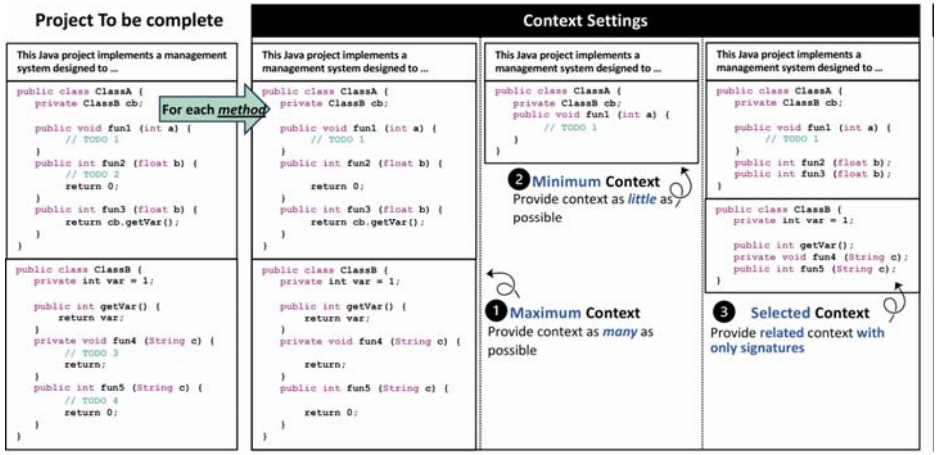

Рис. 3. Демонстрация необходимости стратегии контекста "Maximum" для охвата всех зависимостей.
Fig. 3. Demonstrating the need for the "Maximum" context strategy to cover all dependencies.

2.2 Модернизация инфраструктуры системы оценки Needle-in-a-Haystack

Стандартным инструментом для оценки точности извлечения фактов при наличии большого объема нерелевантной информации является задача «needle in a haystack» (NiAH).

Стандартные реализации NiAH [2] ориентированы на использование внешних API и ограничены устаревшими зависимостями, что затрудняет их воспроизводимость и масштабирование. В рамках данной работы инфраструктура тестирования была переработана с учетом требований автономного выполнения на локальных вычислительных ресурсах. Для проверки корректности извлеченного ответа была реализована автоматическая оценка на базе Qwen-Evaluator с использованием подхода LLM-as-a-Judge. Кроме того, среда выполнения была пересобрана с использованием менеджера uv, что позволило сократить время развертывания и обеспечить стабильную работу с последовательностями длиной до 128 000 токенов. Эти изменения обеспечивают контролируемое и воспроизводимое тестирование моделей в условиях длинного контекста.

2.3 Изоморфизм задач BABILong и отладки кода

Для оценки способности моделей выполнять последовательные рассуждения используется набор задач BABILong [3], предназначенный для анализа работы с длинными текстовыми последовательностями. Выбор данного инструмента обусловлен структурным сходством между задачами обработки текста и задачами отладки программ.

В рамках BABILong модель должна отслеживать изменения состояния объектов на основе последовательности событий, например перемещения сущностей или изменения их свойств (рис. 4). Аналогичная задача возникает при анализе трассировок выполнения программ, где требуется отслеживать изменения значений переменных и восстанавливать причинно-следственные связи, приводящие к возникновению ошибки.

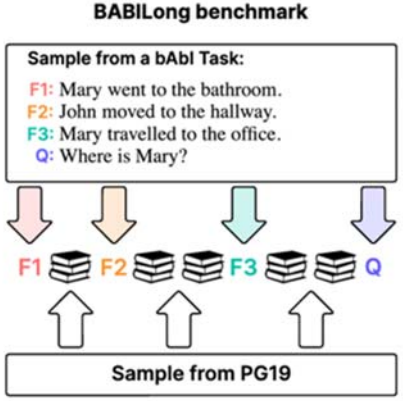

Рис. 4. Структура задач набора BABILong.
Fig. 4. BABILong benchmark task structure.

Таким образом, между задачами рассуждения в BABILong и процессом отладки кода наблюдается изоморфизм: в обоих случаях требуется последовательное обновление состояния и извлечение релевантной информации из цепочки зависимых шагов. Способность модели успешно решать задачи данного типа рассматривается как индикатор ее применимости для локализации ошибок в условиях длинного контекста.

Совместное использование JavaBench+, NiAH и BABILong формирует многоуровневую схему оценки, охватывающую генерацию кода, точность извлечения информации и способность к последовательному анализу состояния.

3. JavaCapsule

Предлагаемый в работе подход основан на развитии идей итеративной отладки [4], ранее реализованных для Python, в частности, в рамках фреймворка PyCapsule [5]. В отличие от динамических языков программирования, применение аналогичных методов к Java сопряжено с дополнительными трудностями, обусловленными строгой типизацией, необходимостью компиляции и сложной структурой зависимостей между компонентами программы.

В данной работе предлагается методология JavaCapsule, адаптирующая подход итеративной отладки к условиям объектно-ориентированной среды Java. Основная цель заключается в повышении качества генерации программного кода за счет интеграции механизма последовательного исправления ошибок на основе результатов выполнения.

3.1 Архитектура и цикл работы

JavaCapsule по аналогии с PyCapsule (рис. 5) реализует двухкомпонентную схему взаимодействия, включающую генерацию кода и его выполнение в изолированной среде. Процесс организован в виде итеративного цикла, в котором сгенерированное решение последовательно уточняется на основе результатов его выполнения.

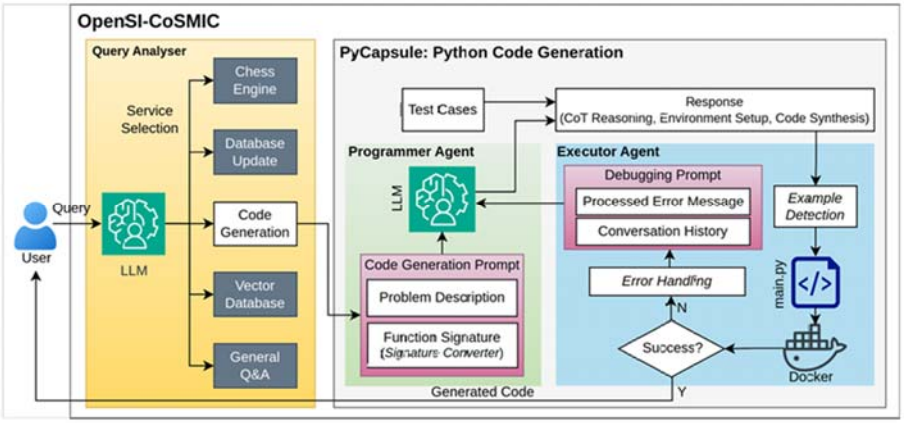

Рис. 5. Архитектура итеративной генерации и выполнения кода PyCapsule.
Fig. 5. PyCapsule's iterative code generation and execution architecture.

Выполнение кода осуществляется в контейнеризованной среде, обеспечивающей воспроизводимость экспериментов и изоляцию зависимостей. В качестве инструмента сборки используется Gradle, а проверка корректности реализуется с помощью тестов JUnit, что соответствует стандартной практике разработки Java-приложений.

Процесс организован в виде итеративного цикла, состоящего из следующих этапов:

1) Генерация начальной версии решения на основе текстового описания задачи и доступного контекста проекта.
2) Компиляция и выполнение сгенерированного кода с использованием системы сборки и тестирования.
3) Сбор информации об ошибках, включая сообщения компилятора, результаты тестов и трассировки выполнения.
4) Формирование структурированного представления ошибок, пригодного для последующей обработки моделью.
5) Повторная генерация кода с учетом полученной обратной связи.

Данный цикл повторяется до получения корректного решения либо достижения ограничения по числу итераций. Использование реальной среды выполнения позволяет учитывать не только синтаксическую корректность, но и фактическое поведение программы.

3.2 Адаптация к особенностям Java

Прямое применение существующих подходов итеративной отладки, разработанных для динамических языков, оказывается недостаточным для Java в силу особенностей его экосистемы. В рамках JavaCapsule реализован ряд адаптаций, направленных на учет этих особенностей.

- **Предоставление кода тестов**

  В отличие от стандартных подходов, где модель получает только сообщения об ошибках, в контекст дополнительно включаются тексты не пройденных тестов. Это позволяет рассматривать тесты как формализованное описание требуемого поведения программы. В результате исправление ошибок опирается на явно заданные условия, а не на косвенные сигналы, что снижает неопределенность при генерации.

- **Отказ от цепочек рассуждений при генерации кода**

  Использование промежуточных текстовых рассуждений увеличивает нагрузку на контекстное окно и может приводить к генерации некорректных или несуществующих конструкций. В условиях строгой типизации и объемного синтаксиса Java это снижает качество итогового кода. Прямая генерация без явных рассуждений позволяет повысить синтаксическую корректность и стабильность компиляции.

- **Структурирование журналов выполнения**

  Сообщения компилятора и результаты тестирования часто содержат элементы форматирования, затрудняющие их интерпретацию моделью. В рамках подхода реализована предварительная обработка журналов, преобразующая их в структурированное текстовое представление. Это обеспечивает выделение семантически значимой информации и упрощает использование ошибок в качестве входных данных для последующих итераций.

3.3 Экспериментальная оценка

Эффективность предложенного подхода оценивалась на задачах набора JavaBench+ с использованием Google Gemma-3-27B-it в качестве базовой генеративной модели. Результаты показывают, что внедрение итеративной отладки приводит к существенному улучшению качества генерации по метрике Pass@1.

На проектах средней сложности (P19, 10 взаимосвязанных классов) наблюдается прирост на 20 процентных пунктов по сравнению с базовой моделью без механизма отладки. Для более крупных проектов, включающих большее число взаимосвязанных компонентов (P20, P21, 14 классов), сохраняется стабильное улучшение на 7-8 процентных пунктов (рис. 6).

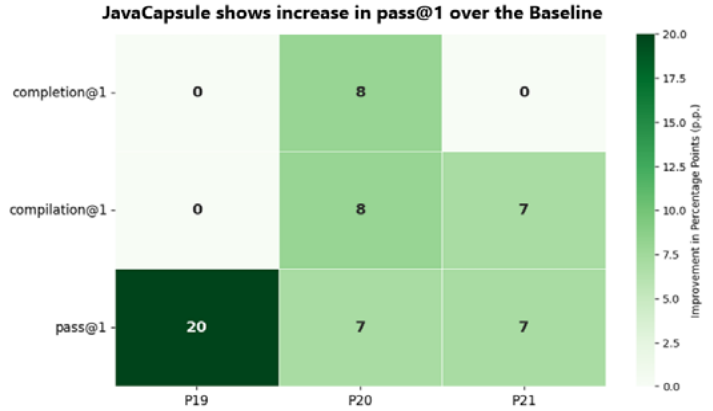

Рис. 6. Показатели улучшения качества генерации с использованием методологии JavaCapsule.
Fig. 6. Quantitative indicators of improvement in the quality of generation using the JavaCapsule methodology.

Полученные результаты подтверждают, что использование структурированной обратной связи и динамического анализа выполнения позволяет существенно повысить корректность генерируемого кода без увеличения сложности базовой модели.

4. Исследование архитектуры ARMT для работы с длинным контекстом

Задачи анализа кодовых баз и трассировок выполнения в крупных промышленных Java-проектах требуют обработки последовательностей, длина которых может существенно превышать стандартные ограничения контекстного окна языковых моделей. В связи с этим наряду с исследованием методологии JavaCapsule в работе рассматривается архитектура Ассоциативного рекуррентного трансформера с памятью (ARMT) [6], основанного на подходах интеграции внешней памяти в трансформерные модели [7], предназначенная для обработки сверхдлинных последовательностей. Целью данного исследования является оценка возможностей архитектуры ARMT при решении задач извлечения информации и анализа причинно-следственных зависимостей в длинном контексте.

В отличие от классических моделей с полным вниманием [8], где вычислительная сложность возрастает квадратично с увеличением длины последовательности, ARMT использует фиксированный по размеру модуль памяти, в который последовательно агрегируется информация из входных данных (рис. 7). Такой подход позволяет обрабатывать длинные последовательности без необходимости хранения полного контекста в механизме внимания.

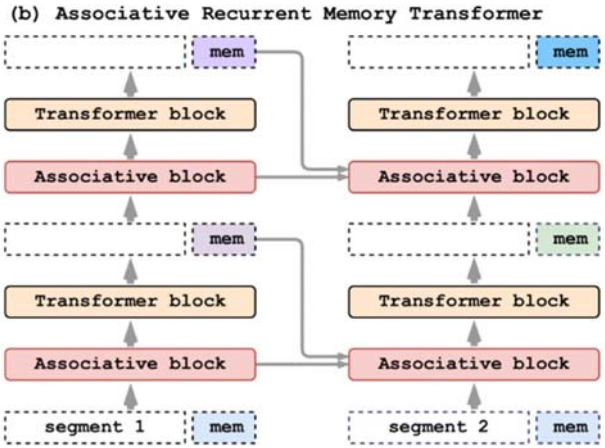

Рис. 7. Архитектура ARMT.
Fig. 7. ARMT architecture.

4.1 Экспериментальная валидация

Для анализа свойств архитектуры проведено экспериментальное исследование на моделях различного масштаба. На начальном этапе рассматривалась базовая языковая модель GPT-2 (137 миллионов параметров), дополненная слоями ARMT (10 токенов памяти, размерность 64), обучение которой осуществлялось с постепенным увеличением количества сегментов.

В ходе обучения наблюдается постепенный переход от режима, в котором модель преимущественно опирается на локальные зависимости (рис. 8), к режиму использования механизма памяти для извлечения информации из ранее обработанных сегментов (рис. 9).

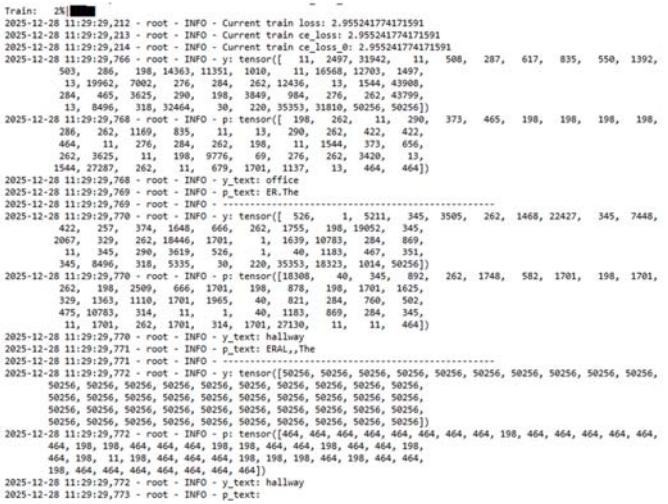

Рис. 8. Ранние итерации обучения без использования механизма памяти.
Fig. 8. Early iterations of learning without using a memory mechanism.

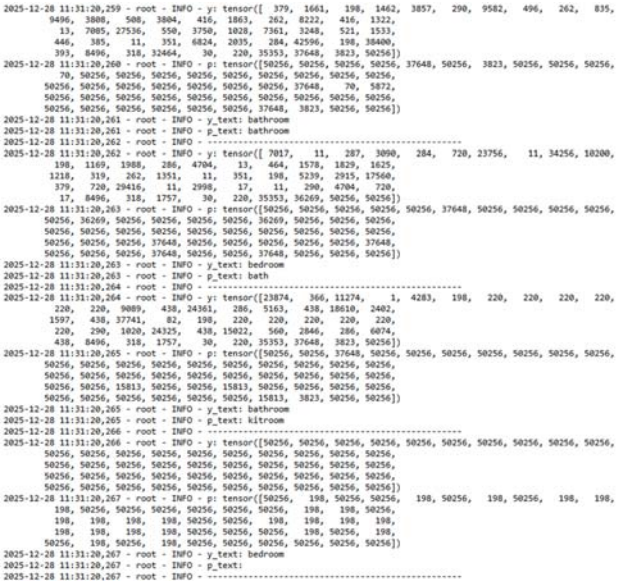

Рис. 9. Фазовый переход к использованию ассоциативной памяти.
Fig. 9. Phase transition to the use of associative memory.

Ключевым результатом является способность модели, обученной на коротких контекстах (до 32 тысяч токенов), корректно обрабатывать сверхдлинные последовательности без дообучения. В процессе обучения наблюдается быстрое увеличение точности на валидационной выборке, достигающее примерно 88% уже к 200-й итерации (рис. 10).

При тестировании на последовательностях длиной до одного миллиона токенов достигается точность извлечения информации порядка 99% (рис. 11).

Полученные результаты свидетельствуют о том, что модель осваивает устойчивый механизм работы с памятью, обеспечивающий как эффективное извлечение информации, так и обобщение на последовательности существенно большей длины.

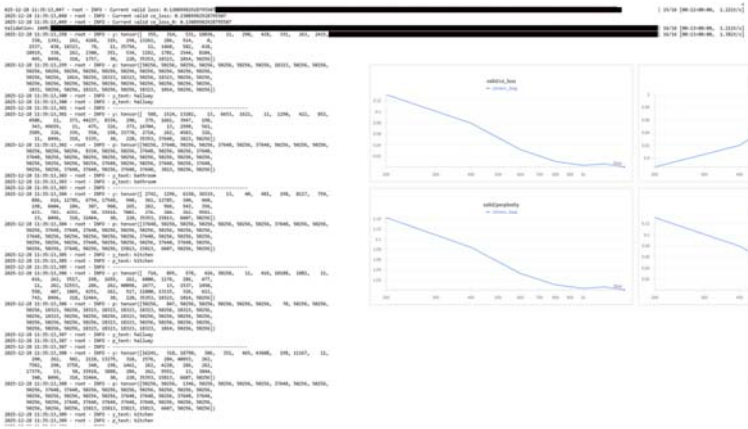

Рис. 10. Быстрая сходимость точности на валидационной выборке для модуля ARMT.
Fig. 10. Fast convergence of accuracy on the validation set for the ARMT module.

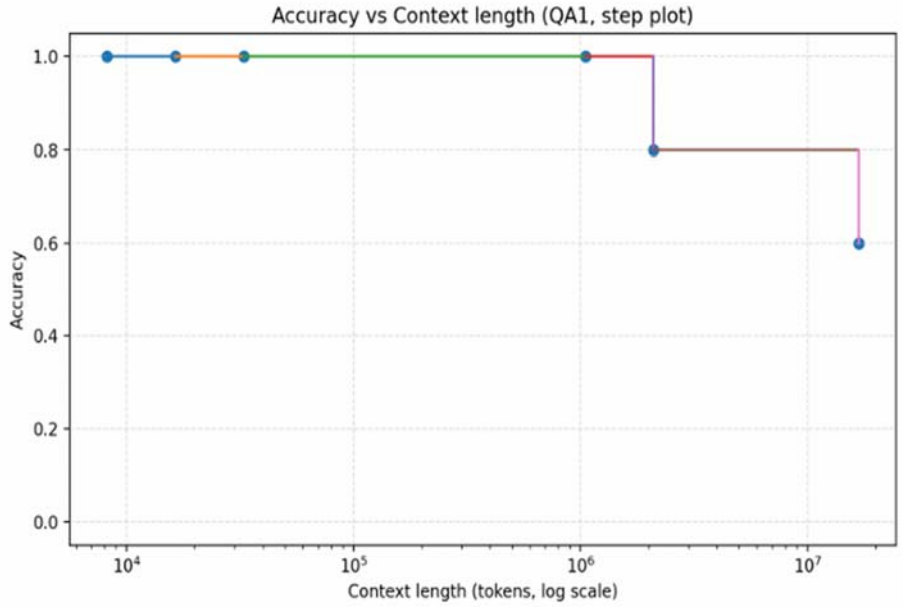

Рис. 11. Зависимость точности извлечения от длины контекста.
Fig. 11. Dependence of extraction accuracy on context length.

4.2 Масштабирование архитектуры

Для подтверждения архитектурной масштабируемости было проведено дополнительное исследование, заключающееся в применении ARMT к языковой модели Qwen-1.5 (0.5 миллиардов параметров).

Результаты показывают, что обучение характеризуется быстрой сходимостью: значение функции потерь стабилизируется уже на ранних этапах обучения (менее 100 итераций), а точность извлечения информации достигает высоких значений без длительной донастройки (рис. 12).

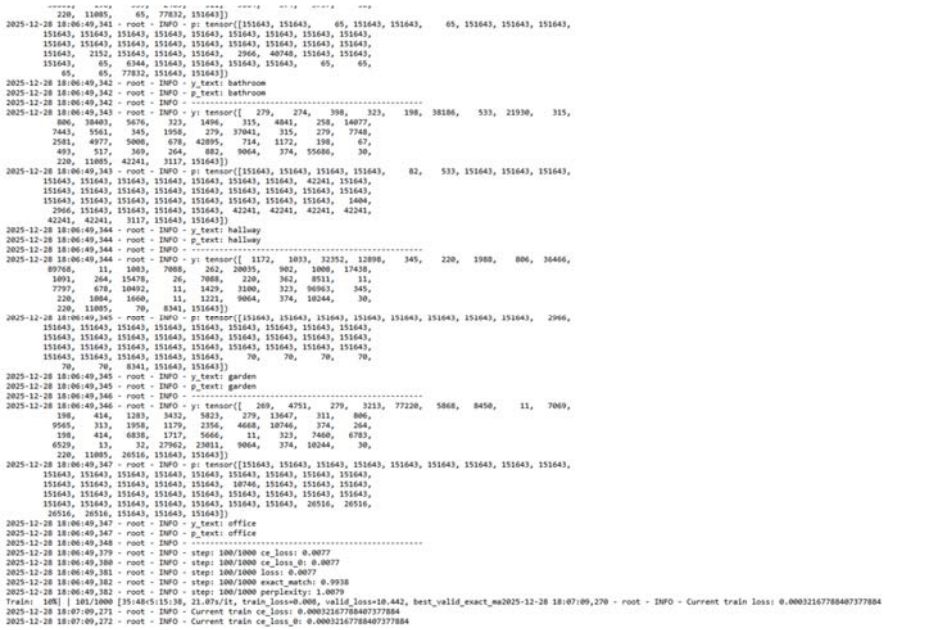

Рис. 12. Демонстрация быстрой сходимости конфигурации Qwen-0.5B ARMT.
Fig. 12. Demonstration of fast convergence of the Qwen-0.5B ARMT configuration.

Это указывает на возможность масштабирования архитектуры ARMT без существенного увеличения вычислительных затрат и на ее применимость в сочетании с современными языковыми моделями, при сохранении высокой эффективности обучения.

Дополнительно была проведена оптимизация вычислительной реализации архитектуры ARMT. За счет изменения порядка выполнения матричных операций удалось исключить формирование промежуточных тензоров большого размера, возникающих при обновлении состояния памяти. В результате пиковое потребление видеопамяти сократилось приблизительно со 110-130 ГБ до 30 ГБ, что позволило увеличить размер пакета с 32 до 128. Полученные результаты демонстрируют, что оптимизация реализации ARMT позволяет существенно снизить требования к вычислительным ресурсам без изменения архитектуры модели.

4.3 Применимость к задачам анализа кода

Рассматриваемая архитектура ориентирована на точное извлечение дискретной информации из длинных последовательностей, что соответствует требованиям задач анализа журналов

выполнения и кодовых баз. В отличие от моделей, ориентированных преимущественно на генерацию текста, ARMT обеспечивает сохранение и извлечение фактов, распределенных по контексту. Полученные результаты подтверждают применимость ARMT для использования в составе гибридных систем, где функции генерации и анализа разделяются между различными компонентами.

5. Сравнительный анализ архитектур с длинным контекстом

Для выбора архитектурной основы, подходящей для задач генерации и отладки кода, проведено сравнение современных моделей, ориентированных на обработку длинных последовательностей. Анализ включает как модели с линейной сложностью, так и гибридные подходы, сочетающие различные механизмы обработки контекста.

5.1 Модели пространства состояний

Модели пространства состояний, включая модели семейства Mamba [9], обеспечивают линейное масштабирование по длине последовательности и характеризуются низкими требованиями к памяти. Это делает их эффективными с точки зрения вычислительных затрат при работе с длинными входами.

Однако экспериментальные результаты показывают, что такие модели испытывают трудности при точном извлечении дискретной информации из середины длинных последовательностей. Наблюдается эффект постепенной деградации точности, при котором отдельные идентификаторы и значения теряются или искажаются по мере увеличения длины контекста. Это ограничивает их применимость в задачах, где требуется точное восстановление состояния программы.

5.2 Гибридные архитектуры

Гибридные модели, такие как Zamba2 [10] и Nemotron [11], комбинируют механизмы линейной обработки последовательностей с элементами внимания. Такой подход позволяет снизить требования к памяти при сохранении доступа к глобальному контексту.

Практические измерения показывают, что данные архитектуры обеспечивают высокую пропускную способность и стабильную работу при больших размерах входа. Однако в задачах генерации кода, требующих строгого соблюдения синтаксических и семантических ограничений, они уступают классическим моделям-декодерам, что связано с особенностями их архитектуры.

5.3 Модели с разреженным вниманием

Модель RWKV-X [12], использующая механизмы разреженного внимания, демонстрирует конкурентоспособные результаты на задачах извлечения информации при умеренных длинах последовательностей. Однако при увеличении сложности задач, требующих строгого следования инструкциям и форматам вывода, наблюдаются отклонения от заданных требований.

В частности, в задачах, требующих точного соблюдения формата ответа, модель может игнорировать ограничения, что снижает ее применимость в автоматизированных конвейерах генерации и отладки кода.

5.4 Обобщающий анализ

Проведенное сравнение показывает, что различные архитектурные подходы обладают взаимодополняющими свойствами. Модели с линейной сложностью обеспечивают эффективную обработку длинных последовательностей, но уступают в точности извлечения

информации. Гибридные архитектуры демонстрируют компромисс между качеством и производительностью, однако не достигают уровня генеративных моделей в задачах синтеза кода.

Рекуррентные архитектуры с памятью, такие как ARMT, обеспечивают точное извлечение информации из длинного контекста и сохраняют дискретные факты, критически важные для анализа программных систем. Это делает их предпочтительным выбором для задач, связанных с обработкой журналов выполнения и кодовых баз, где требуется высокая точность восстановления состояния.

6. Гибридная архитектура генерации и отладки кода

Результаты, полученные в предыдущих разделах, указывают на ограниченность использования единой архитектуры для решения задач генерации и анализа кода в условиях длинного контекста. Генеративные модели обеспечивают высокое качество синтеза программного кода, однако ограничены размером контекстного окна. В то же время рекуррентные архитектуры с памятью демонстрируют высокую точность извлечения информации, но не оптимизированы для генерации синтаксически корректного кода.

Для объединения указанных свойств предлагается гибридная архитектура, в которой функции генерации и анализа распределяются между специализированными компонентами.

6.1 Архитектурная схема

По результатам исследования, мы решили объединить возможности стандартных LLM к генерации кода с итеративной отладкой и способности ARMT-модели работать с миллионами токенов контекста и предлагаем гибридное решение (рис. 13). ARMT-модель анализирует многомиллионный контекст и передает большой кодогенерирующей модели результат анализа (отобранные фрагменты кода или подсказка о том, как исправить ошибку).

В основе ARMT-модели лежит гибридная архитектура типа энкодер-декодер, в которой обработка контекста и генерация ответа разделены на независимые этапы.

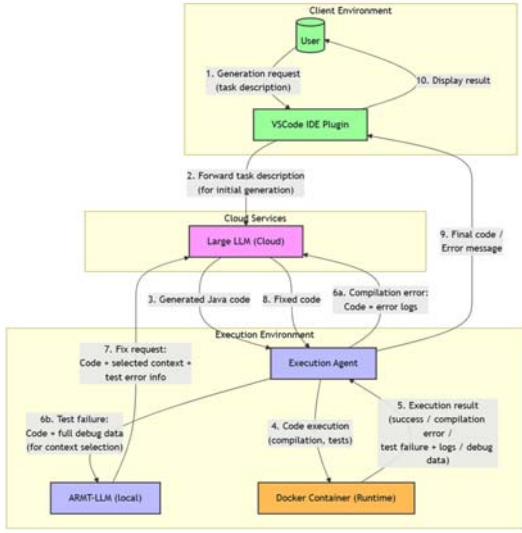

Рис. 13. Общая схема потока данных предлагаемой гибридной архитектуры JavaCapsule.
Fig. 13. General data flow diagram of the proposed JavaCapsule hybrid architecture.

- **Энкодер с модулем памяти**

  Энкодер расширен механизмом памяти на базе ARMT, что позволяет обрабатывать входные данные значительной длины, включая полные кодовые базы и трассировки выполнения. В процессе работы информация последовательно агрегируется в компактное представление, сохраняющее структуру программы, сигнатуры методов и изменения состояния.

- **Стандартный декодер**

  Декодер реализует генерацию ответа на основе представления, сформированного энкодером. Используется стандартная трансформерная архитектура, оптимизированная под взаимодействие с энкодером и на выдачу ответов в правильном формате.

- **Механизм взаимодействия компонентов**

  Связь между энкодером и декодером осуществляется через механизм перекрестного внимания, обеспечивающий доступ генеративной модели к информации, сохраненной в модуле памяти. Такая организация соответствует предлагаемой архитектуре типа T5+ARMT, в которой кодер отвечает за обработку сверхдлинного контекста, а декодер – за качественную генерацию ответа на основе сжатого представления, а также прямого доступа к своим предыдущим токенам (рис. 14).

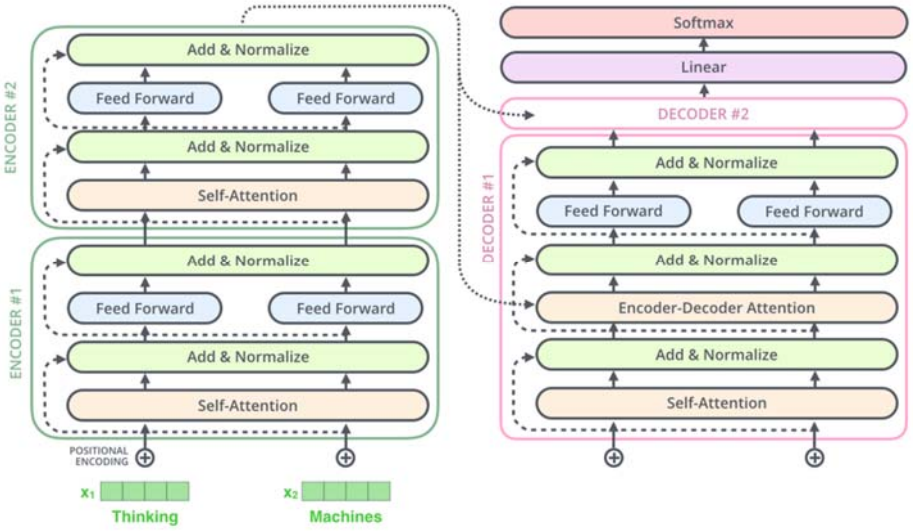

Рис. 14. Детальная организация взаимодействия кодера с модулем памяти и генеративного декодера.
Fig. 14. Detailed organization of the interaction between the encoder with memory module and the generative decoder.

6.2 Поток обработки данных

Процесс работы системы включает следующие этапы:

1) Поступление входных данных, включающих описание задачи, кодовую базу проекта и результаты предыдущих запусков.

2) Обработка входа кодером с модулем памяти и формирование компактного представления контекста.
3) Генерация решения декодером на основе полученного представления.
4) Выполнение сгенерированного кода в изолированной среде и получение результатов тестирования.
5) Передача информации об ошибках обратно в модуль памяти для последующего анализа.

Таким образом, система реализует замкнутый цикл, объединяющий генерацию и анализ, что позволяет учитывать результаты выполнения при последующих итерациях.

6.3 Преимущества подхода

Предлагаемая архитектура обеспечивает разделение задач генерации и обработки контекста, что позволяет избежать ограничений, характерных для отдельных классов моделей. Использование внешней памяти обеспечивает доступ к информации, недоступной в рамках стандартного контекстного окна, тогда как специализированный декодер гарантирует корректность синтеза кода.

В результате достигается возможность масштабирования системы на задачи, включающие большие программные проекты и длинные трассировки выполнения, без существенного ухудшения качества генерации.

7. Заключение

В работе рассмотрена задача повышения качества генерации и отладки Java-кода в условиях длинного контекста, характерного для современных программных систем. Ограничения стандартных архитектур языковых моделей, связанные с размером контекстного окна и сложностью обработки длинных последовательностей, делают необходимым использование специализированных подходов.

Предложенная методология JavaCapsule реализует итеративную отладку с использованием структурированной обратной связи, включающей результаты выполнения тестов и трассировки ошибок. Экспериментальные результаты показывают, что даже обычная итеративная отладка может существенно повысить качество генерации кода, обеспечивая прирост метрики Pass@1 на 20 процентных пунктов по сравнению с базовой моделью. Ожидается, что добавление модуля анализа контекста должно еще улучшить этот результат.

Проведенное исследование архитектур для работы с длинным контекстом показало, что модели с явной внешней памятью, в частности ARMT, обеспечивают высокую точность извлечения информации из длинных последовательностей и эффективно масштабируются на задачи, превышающие стандартные ограничения языковых моделей.

На основе полученных результатов предложена гибридная архитектура, объединяющая генеративную модель для синтеза кода и модуль памяти для анализа длинных трассировок выполнения и кодовых баз. Такое разделение функций позволяет одновременно учитывать требования к качеству генерации и точности обработки контекста.

Полученные результаты показывают, что итеративная отладка обеспечивает повышение качества генерации Java-кода, тогда как исследование архитектуры ARMT подтверждает возможность эффективной обработки сверхдлинных последовательностей. Тем самым работа охватывает как прикладной аспект генерации и отладки Java-кода, так и исследование специализированных механизмов памяти для анализа длинного программного контекста.

Дальнейшее развитие работы может быть направлено на масштабирование предложенной архитектуры за счет предварительного обучения, исследование более сложных структур

памяти, а также интеграцию системы в среды разработки с поддержкой взаимодействия в реальном времени.

Список литературы / References

[1]. Cao J., Chen Z., Wu J., Cheung S., Xu C. JavaBench: A Benchmark of Object-Oriented Code Generation for Evaluating Large Language Models. Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024. P. 870–882. DOI: 10.1145/3691620.3695470.

[2]. Kamradt, G. Needle in A Haystack – Pressure Testing LLMs. Available at: https://github.com/gkamradt/LLMTest_NeedleInAHaystack, accessed 29.12.2025.

[3]. Kuratov Y., Bulatov A., Anokhin P., Rodkin I., Sorokin D., Sorokin A., Burtsev M. BABILong: testing the limits of LLMs with long context reasoning-in-a-haystack. Proceedings of the 38th International Conference on Neural Information Processing Systems, 2024. Vol. 37. P. 106519–106554.

[4]. Chen X., Lin M., Schärli N., Zhou D. Teaching Large Language Models to Self-Debug. Proceedings of the 12th International Conference on Learning Representations (ICLR), 2024.

[5]. Adnan M., Xu Z., Kuhn C. C. N. Large Language Model Guided Self-Debugging Code Generation. arXiv preprint, 2025. DOI: 10.48550/arXiv.2502.02928.

[6]. Bulatov A., Kuratov Y., Burtsev M. Recurrent Memory Transformer. Advances in Neural Information Processing Systems, 2022. Vol. 35. P. 11079–11091. DOI: 10.52202/068431-0805.

[7]. Burtsev M. S., Kuratov Y., Peganov A., Sapunov G. V. Memory Transformer. arXiv preprint, 2020. DOI: 10.48550/arXiv.2006.11527.

[8]. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A.N., Kaiser L., Polosukhin I. Attention Is All You Need. Advances in Neural Information Processing Systems, 2017. Vol. 30. P. 5998–6008.

[9]. Lahoti A., Li K., Chen B., Wang C., Bick A., Kolter J.Z., Dao T., Gu A. Mamba-3: Improved Sequence Modeling using State Space Principles. Proceedings of the 14th International Conference on Learning Representations (ICLR), 2026.

[10]. Glorioso P., Anthony Q., Tokpanov Y., Golubeva A., Shyam V., Whittington J., Pilault J., Millidge B. The Zamba2 Suite: Technical Report. arXiv preprint, 2024. DOI: 10.48550/arXiv.2411.15242.

[11]. NVIDIA, Blakeman A., Basant A., Khattar A., Renduchintala A., Bercovich A., Ficek A., et al. Nemotron-H: A Family of Accurate and Efficient Hybrid Mamba-Transformer Models. arXiv preprint, 2025. DOI: 10.48550/arXiv.2504.03624.

[12]. Hou H., Huang Z., Tan K., Lu R., Yu F. R. RWKV-X: A Linear Complexity Hybrid Language Model. arXiv preprint, 2025. DOI: 10.48550/arXiv.2504.21463.

Информация об авторах / Information about authors

Дмитрий Владимирович Александров – доктор технических наук, профессор, заведующий научно-учебной лабораторией облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии, методы и технологии искусственного интеллекта.

Dmitry Vladimirovich ALEXANDROV – Dr. Sci. (Tech.), Prof., Head at the Educational and Research Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include cloud technologies, methods and techniques of artificial intelligence.

Владимир Игоревич ВАСИЛЕВСКИЙ является стажером-исследователем лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии и кодогенерацию.

Vladimir Igorevich VASILEVSKIY is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include cloud technologies and code generation.

Людмила Александровна РЕЗУНИК является младшим научным сотрудником лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера ее научных интересов включает облачные технологии и мультиагентные системы.

Lyudmila Alexandrovna REZUNIK is a junior researcher at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. Her research interests include cloud technologies and multi-agent systems.

Леон Андреевич КУЛИГИН является стажером-исследователем лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии и кодогенерацию.

Leon Andreevich KULIGIN is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include cloud technologies and code generation.

Анастасия Валерьевна МАНУШКИНА является стажером-исследователем лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера ее научных интересов включает большие языковые модели, облачные технологии и кодогенерацию.

Anastasia Valerievna MANUSHKINA is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. Her research interests include LLM, cloud technologies and code generation.

Никита Алексеевич ДУМКИН – стажер-исследователь лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает программную инженерию, методы машинного обучения и разработку высокопроизводительных клиентских систем.

Nikita Alekseevich DUMKIN is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include software engineering, machine learning methods, and the development of high-performance client-side systems.

Михаил Алексеевич ПРОЗОРСКИЙ – стажер-исследователь лаборатории облачных и мобильных технологий факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Сфера его научных интересов включает облачные технологии и кодогенерацию.

Mikhail Alekseevich PROZORSKIY is a research intern at the Laboratory of Cloud and Mobile Technologies, Faculty of Computer Science, National Research University Higher School of Economics. His research interests include cloud technologies and code generation.

Кирилл Юрьевич ПИНИГИН – ведущий инженер университета Иннополис. Сфера его научных интересов включает искусственный интеллект, машинное обучение и кодогенерацию.

Kirill Yurievich PINIGIN is a Lead Engineer at Innopolis University. His research interests include artificial intelligence, machine learning, and code generation.