DOI: 10.15514/ISPRAS-2025-37(5)-13

Generating and Debugging Java Code Using LLMs Based on Associative Recurrent Memory

V.I. Vasilevskiy, ORCID: 0009-0004-0115-7082 <vivasilevskiy_1@edu.hse.ru>
D.V. Alexandrov, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>
HSE University,
11, Pokrovsky blvd, Moscow, 109028, Russia.

Abstract. Automatic code generation by large language models (LLMs) has achieved significant success, yet it still faces challenges when dealing with complex and large codebases, especially in languages like Java. The limitations of LLM context windows and the complexity of debugging generated code are key obstacles. This paper presents an approach aimed at improving Java code generation and debugging. We propose using the Associative Recurrent Memory Transformer (ARMT) model, which extends the context window and has enhanced memory capabilities, to address two tasks: 1) selecting the most relevant snippets from the existing codebase for generating new code; 2) selecting the most significant parts of stack traces and runtime data for iterative debugging. This approach is integrated with an iterative debugging loop, embodied in our developing system "JavaCapsule" (inspired by PyCapsule for Python), which includes compilation and test execution in a controlled Docker environment using Gradle. It is expected that the proposed method will enhance the accuracy and relevance of generated Java code, particularly in the context of large projects, and improve the automated debugging process. Such benchmarks like JavaBench further underscore the need for such focused advancements. This paper is an output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE University).

Keywords: code generation; java; large language models; code debugging; associative recurrent memory transformer; recurrent memory transformer; long context; context selection; iterative debugging; javabench.

For citation: Vasilevskiy V.I., Alexandrov D.V. Generating and Debugging Java Code using LLMs based on Associative Recurrent Memory. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025, pp. 173-182. DOI: 10.15514/ISPRAS-2025-37(5)-13.

Acknowledgements. This research is conducted at the Cloud and Mobile Technologies Laboratory of the Software Engineering Department at HSE University.

Генерация и отладка Java-кода с использованием больших языковых моделей на основе ассоциативной рекуррентной памяти

В.И. Василевский, ORCID: 0009-0004-0115-7082 <vivasilevskiy_1@edu.hse.ru>
Д.В. Александров, ORCID: 0000-0002-9759-8787 <dvalexandrov@hse.ru>
НИУ ВШЭ,
Россия, 109028, г. Москва, Покровский б-р, д. 11.

Аннотация. Автоматическая генерация кода большими языковыми моделями (LLM) достигла значительных успехов, однако все еще сталкивается с проблемами при работе со сложными и объемными кодовыми базами, особенно на таких языках, как Java. Ограничения контекстного окна LLM и сложность отладки сгенерированного кода являются ключевыми препятствиями. В данной статье представлен подход, направленный на улучшение генерации и отладки Java-кода. Мы предлагаем использовать модель Associative Recurrent Memory Transformer (ARMT), обладающую расширенным контекстным окном и улучшенными возможностями памяти, для решения двух задач: 1) выбора наиболее релевантных фрагментов из существующей кодовой базы для генерации нового кода; 2) выбора наиболее значимых частей стектрейсов и рантаймданных для итеративной отладки. Этот подход интегрирован в итеративный цикл отладки, реализованный в нашей разрабатываемой системе «JavaCapsule» (по аналогии с PyCapsule для Python), которая включает компиляцию и выполнение тестов в контролируемой среде Docker с использованием Gradle. Ожидается, что предложенный метод повысит точность и релевантность генерируемого Java-кода, особенно в контексте крупных проектов, и улучшит процесс автоматизированной отладки. Бенчмарки, такие как JavaBench, дополнительно подчеркивают необходимость подобных целенаправленных усовершенствований.

Ключевые слова: генерация кода; java; большие языковые модели; отладка кода; преобразователь ассоциативной рекуррентной памяти; преобразователь рекуррентной памяти; длинный контекст; выбор контекста; итеративная отладка; оценка моделей javabench.

Для цитирования: Василевский В.И., Александров Д.В. Генерация и отладка Java-кода с использованием больших языковых моделей на основе ассоциативной рекуррентной памяти. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 173–182 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)–13.

Благодарности. Данное исследование ведется в Лаборатории Облачных и Мобильных технологий Департамента Программной Инженерии НИУ ВШЭ.

1. Introduction

Large Language Models (LLMs) demonstrate impressive results in the field of automatic code generation [7]. However, applying these models to complex object-oriented languages like Java presents several difficulties. Java projects are often characterized by large code volumes, complex dependencies, and strong typing, requiring models to have a deep understanding of the context. The JavaBench benchmark [1] has highlighted these challenges, particularly in object-oriented programming (OOP) features and project-level code generation, and underscores the relevance of research in this area.

One of the key problems is the limited context window size of modern LLMs. When generating or modifying a code snippet in a large project, the model must access relevant parts of the existing codebase (other classes, methods, interfaces), which often exceeds the standard context limit. Furthermore, debugging the generated code remains a complex task. Approaches based on analyzing code execution block by block and providing the model with runtime data [3] are quite promising but are limited by the same context window, preventing the transmission of the full stack trace or the history of variable value changes needed to find and fix complex errors.

To address the limited context problem, architectures such as the Recurrent Memory Transformer (RMT) [4] have been proposed, using recurrent mechanisms to process long sequences. A further development of this idea, the Associative Recurrent Memory Transformer (ARMT) [5], adds associative memory, significantly improving memory usage efficiency and performance on long-context tasks, such as BABILong [6].

On the other hand, iterative debugging approaches, where code is executed in a controlled environment (e.g., a container), and the execution results (success/failure, test output) are used for the next generation iteration, have shown effectiveness for Python [2]. We are developing a similar system for Java, named ”JavaCapsule”.

In this study, we propose combining the advantages of ARMT and iterative debugging to create a system for generating and fixing Java code. The main idea is to train an ARMT-like model (or use its attention mechanisms) to select the most relevant information – codebase snippets during generation and parts of the stack trace/state during debugging – which is then passed to the main LLM to perform the task. We hypothesize that such an approach will allow effective work with large Java projects and complex errors, overcoming the limitations of the context window. This research is conducted at the Cloud and Mobile Technologies Laboratory of the Software Engineering Department at HSE University in collaboration with researchers from Huawei Technologies Co. Ltd.

2. Related Work

2.1 Code Generation using LLMs

Significant progress has been made in recent years in using LLMs for code generation (e.g., StarCoder [8], Qwen-Coder [9]). Models are trained on vast code corpora and can generate code from textual descriptions in various languages. However, the quality of generation for complex languages like Java, especially within large projects, requires improvement.

Java Code Generation Benchmarks

The landscape of code generation evaluation has been historically dominated by Python. Recognizing this gap, Cao et al. introduced JavaBench [1], a project-level Java benchmark specifically designed to exercise OOP features. JavaBench comprises four Java projects with 389 methods in 106 classes, featuring high test coverage and attestation by undergraduate students. It aims to address imbalances in programming language focus, code granularity (moving beyond function/statement level), and the lack of testing for advanced OOP features (encapsulation, inheritance, polymorphism) in existing benchmarks. JavaBench’s evaluation design includes multiple context settings and synthesis strategies, providing a more nuanced understanding of LLM capabilities in Java. Its findings emphasize the need for future advancements, especially in providing relevant context like method signatures.

2.2 Long Context Processing

The context window limitation is a fundamental problem for transformers. Various architectures are being developed to address it. RMT [4] introduces recurrence at the segment level using special memory tokens. ARMT [5] improves upon RMT with an associative memory mechanism, demonstrating superiority over RMT and other models like Mamba [10] and RWKV [11] in associative retrieval and ultra-long sequence processing tasks (up to 10 million tokens) on the BABILong benchmark [6]. The BABILong benchmark is specifically designed to evaluate a model’s ability to retrieve and use information distributed across long text, making it relevant for assessing models intended to work with large codebases.

2.3 Code Debugging using LLMs

Automated debugging is another promising direction. LDB (Large Language Model Debugger) [3] proposes using code execution information (execution traces, variable values per basic block) to

identify errors. The model is provided with the execution context, based on which it localizes and fixes the bug. However, as noted, the volume of this information can exceed the context window. PyCapsule [2] implements an iterative approach for Python with two agents (programmer and executor), where code is executed in a Docker container, and test results and compilation errors are used to request corrections from the programmer agent. This approach does not require deep stack trace analysis by the model but may need many iterations. Our JavaCapsule system draws inspiration from this iterative, containerized execution model.

2.4 Relevant Context Selection

The idea of selecting relevant information is not new and is actively used in Retrieval-Augmented Generation (RAG) [12], where an external knowledge base is used to find relevant documents. In the context of code generation, this might mean searching for similar code snippets or documentation. However, initial analysis of solutions like RAG and Repository Mapping indicates they may not be sufficiently effective for selecting precise, deeply-nested contextual information required for complex generation and debugging tasks within large, existing Java codebases. A more refined mechanism is needed, capable of extracting specific dependencies or relevant parts of debugging information. The associative memory mechanisms in ARMT [5] could potentially be adapted to train a model for such selective information extraction from structured context (codebase, stack trace).

3. Proposed Method

Within this research, we are developing ”JavaCapsule”, a system for Java code generation and debugging based on the following components:

1. Context/Debugging Selection Model based on ARMT: The core of the system is a model utilizing ARMT principles, trained to perform two main functions:
   - Code Context Selection: Upon receiving a request for Java code generation or modification (e.g., description of a method, class to be changed), the model analyzes the current codebase (provided as an indexed set of files or a structural representation) and selects the most relevant snippets (imports, signatures of other methods, class fields, parent classes, interfaces) necessary for correct generation. The associative memory mechanism of ARMT [5] can be used to establish connections between the request and relevant code sections.
   - Debugging Information Selection: When a compilation error or test failure occurs, the model receives the error message and stack trace. The model’s task is to select the most informative lines or blocks from the stack trace and possibly from the execution history (if available) that indicate the cause of the problem. This allows focusing the LLM on the source of the error without overloading the context with redundant information.
2. Main LLM Generator: We have selected ‘gemma3-27b-it‘ as the base large language model capable of generating Java code. It receives the original user request as input, augmented with the relevant code context or debugging information selected by the model in step 1.
3. Iterative Debugging Loop (JavaCapsule Workflow): The generation and debugging process is iterative, adapting the idea from [2]:
   - User Request: A user, potentially through an IDE plugin, submits a task description for code generation or modification.
   - Generation: The main LLM (‘gemma3-27b-it‘) generates Java code based on the request and context selected by the ARMT-based model (if applicable for an existing project).

- Compilation and Testing (Execution Agent): The generated code (or modified project) is passed to an Execution Agent. This agent compiles the Java code using standard tools (e.g., Gradle) and runs user-provided or automatically generated unit tests (e.g., JUnit). This step is performed in an isolated Docker container for security, dependency management, and reproducibility.
- Result Processing:
  - Successful Execution: If compilation and all tests pass, the final code is returned to the user (e.g., displayed in the IDE).
  - Compilation Error: If a compilation error occurs, the error logs are captured by the Execution Agent. These logs are then processed by the ARMT-based debugging information selection model to extract relevant error messages. The selected information is passed back to the main LLM along with the problematic code for a correction attempt.
  - Test Failure: If tests fail, the Execution Agent runs tests in debug mode to gather more comprehensive debug data (e.g., stack traces, intermediate variable values if feasible). This complete debug data is passed to the ARMT-based model, which selects the most relevant context from this data and the original code. The problematic code, selected debug context, and test error information are then sent to the main LLM for fixing.
- Repetition: The loop (Fixed code -> Retry with new code) repeats from the Compilation and Testing step until successful execution or an iteration limit is reached.

The architecture, inspired by PyCapsule [2], is shown in Fig. 1. The key distinction in our JavaCapsule approach is the explicit use of an ARMT-based model for intelligent selection of both code context during generation and specific debugging information during the error-fixing iterations.

A key aspect is training the ARMT model (or its equivalent) for selection tasks. This requires creating specialized datasets for Java, where generation requests are annotated with relevant parts of the codebase, and error reports are annotated with relevant stack trace lines.

4. Preliminary Considerations and Future Work

This research is currently in the concept development and initial implementation stage. The JavaCapsule system (repository: https://github.com/Vvil1568/JavaCapsule) is under active development, aiming to realize the iterative debugging loop with Docker and Gradle for Java.

Before settling on the ARMT-based selection model, several alternative approaches were considered. One involved using a 7B model where input tokens are replaced by pre-computed BERT embeddings for Java code snippets. Another approach was to fine-tune a 7B model using ParameterEfficient Fine-Tuning (PEFT) techniques like LoRA to generate Java code with explicit type annotations (e.g., transforming ‘someObj.foo()‘ into ‘((SomeClass)someObj).foo()‘). While promising, these approaches primarily focus on altering the generation process itself. The ARMT approach was ultimately chosen for its direct focus on solving the core problem of long-context retrieval and relevance filtering, which we believe is a more fundamental bottleneck for working with large codebases and complex debugging scenarios. A comparative computational cost analysis of these three plans is provided in the next section. Expected Advantages:

- Improved quality of Java code generation for large projects by providing the LLM with relevant context.
- Enhanced automated debugging by focusing the LLM on significant parts of diagnostic information.
- Overcoming LLM context window limitations without needing models with ultra-large context for the entire task.

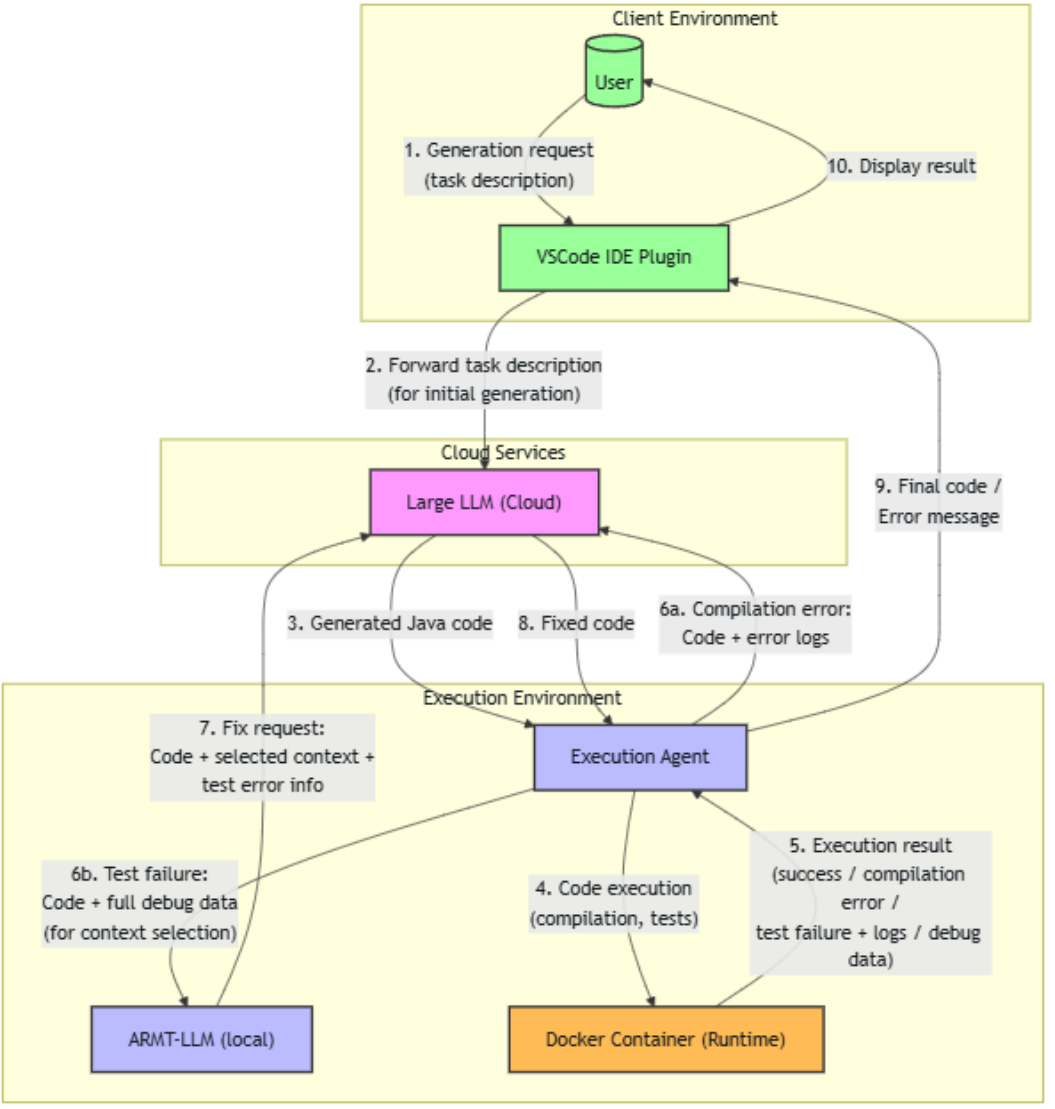

Fig. 1. Adapted architecture for iterative Java code generation and debugging in JavaCapsule. Steps include generation by an LLM (e.g., Gemma), execution/testing in a Docker container via Gradle, and a feedback loop using an ARMT model for context/debugging information selection (adapted from [2]). The "ARMT-LLM (local)" in the diagram represents our proposed ARMT-based selection model.

Main Challenges:

- Training Data Creation: Developing methodologies and tools for creating datasets linking generation/debugging requests with relevant code/stack trace snippets in Java projects. We plan to initiate dataset collection, potentially using automated labeling with existing generative models as a starting point.
- Training the Selection Model: Choosing the architecture (ARMT or similar) and effectively training the model for selective information extraction from structured, yet large, contexts.
- Component Integration: Creating an efficient pipeline combining the selection model, the LLM generator ('gemma3-27b-it'), and the JavaCapsule compilation/testing system.

- Evaluation: The JavaBench benchmark [1] provides a valuable resource. We have started adapting JavaBench for evaluating the Gemma model (adapted version available at: https://github.com/Vvil1568/JavaBench/tree/gemma). This will require further development of custom metrics and test scenarios for our specific tasks.

Future Work Plan:

1. Continue development of the JavaCapsule prototype system for compiling and testing Java code in a container using Gradle.
2. Begin creation of a dataset for training and evaluating the code context selection model using several open-source Java projects.
3. Initiate creation of a dataset for training and evaluating the debugging information selection model based on real or synthetic errors, exploring automated labeling techniques.
4. Experiment with Gemma-3 models as the generator and as the base for the ARMT selector.
5. Compare the proposed approach with baseline LLMs (without context selection) and standard RAG approaches on the adapted JavaBench and custom tasks.
6. Evaluate performance on tasks such as generating new methods, modifying existing code, and fixing errors based on tests within the JavaCapsule framework.

We plan to use code quality metrics (e.g., Pass@k test pass rate from JavaBench, CodeBLEU [13]) and debugging efficiency metrics (number of iterations, percentage of fixed errors).

5. Computational Cost Analysis of Training Approaches

This section provides a high-level estimation of the computational resources required for training the models under the three considered plans. These figures are approximate and intended to provide a sense of scale for each approach. A summary is presented in Table 1.

5.1 General Assumptions

The following assumptions are used for the calculations:

- Precision: BF16 (2 bytes per parameter).
- Optimizer: AdamW, which requires 2 additional values per parameter, resulting in 4 bytes per parameter for optimizer states. Gradients require 2 bytes/param.
- Total Memory per Parameter (Full Training): 2 (params) + 4 (optimizer) + 2 (gradients) = 8 bytes. For PEFT, this applies only to trainable parameters.
- Total FLOPs (heuristic): Approximately $6\times N \times D$ for full training and $\approx 3\times N_{full} \times D$ for PEFT, where N is the number of parameters and D is the number of tokens.
- Model FLOPs Utilization (MFU): We assume a realistic MFU of 40% of the GPU's peak theoretical performance.

5.2 Plan 1: 1.1B ARMT-based Context Selector (Full Tuning)

- Model and Task: A 1.1B parameter ARMT-like model for context selection, trained via full fine-tuning.
- Training Data: Estimated at 40 billion tokens.
- VRAM (Video Memory) Estimation:
  - Model States: 1.1×109 params ×8 bytes/param = 8.8 GB.
  - Activations: Can consume 15-30 GB or more, depending on batch size and sequence length.
  - Total Estimated VRAM: 24-44+ GB.

- Training Time Estimation:
  - Total TFLOPS Required: 6×(1.1×109)×(40×109) = 2.64×1020 FLOPS, or 264,000 PetaFLOPS.
  - Estimated GPU-Hours (H100):

$$\frac{2.64 \times 10^{20}}{1000 \times 10^{12} \times 0.4 \times 3600} \approx 183.300\ GPU - hours$$

5.3 Plan 2: 7B Model with BERT Embeddings (Full Tuning)

- Model and Task: A 7B parameter model for code generation, where input tokens are precomputed BERT embeddings. This involves full fine-tuning.
- Training Data: Estimated at 15 billion BERT-vector ”tokens”.
- VRAM (Video Memory) Estimation:
  - Model States: 7×109 params ×8 bytes/param = 56 GB.
  - Activations: For a 7B model, activations can easily require 20-40+ GB.
  - Total Estimated VRAM: 56 GB + (20-40+ GB) = 76-96+ GB. This requires A100 (80GB) or H100 GPUs.
- Training Time Estimation:
  - Total TFLOPS Required: 6×(7×109)×(15×109) = 6.3×1020 FLOPS, or 630,000 PetaFLOPS.
  - Estimated GPU-Hours (H100):

$$\frac{6.3 \times 10^{20}}{1000 \times 10^{12} \times 0.4 \times 3600} \approx 437.500\ GPU - hours$$

5.4 Plan 3: 7B Model with Explicit Types (PEFT)

- Model and Task: Fine-tuning a 7B parameter model using PEFT (LoRA) to generate Java code with explicit type annotations.
- Training Data: Requires a large dataset of Java code pre-processed with explicit types, estimated at 100 billion tokens.
- VRAM (Video Memory) Estimation:
  - Frozen Model: 7×109 params ×2 bytes/param = 14 GB.
  - LoRA Adapters (70M params): Optimizer states and gradients for adapters require 70×106×6 bytes ≈ 0.42 GB.
  - Activations: Calculated for the full 7B model, requiring 20-40+ GB.
  - Total Estimated VRAM: 14 GB + 0.5 GB + (20-40+ GB) = 35-55+ GB. Suitable for 48GB-class GPUs and above.
- Training Time Estimation:
  - Total TFLOPS Required: $3 \times (7 \times 109) \times (100 \times 109) = 2.1 \times 1021$ FLOPS, or 2,100,000 PetaFLOPS.
  - Estimated GPU-Hours (H100):

$$\frac{2.1 \times 10^{21}}{1000 \times 10^{12} \times 0.4 \times 3600} \approx 1{,}458.000\ GPU - hours$$

These estimates underscore that all considered plans require access to significant high-performance computing infrastructure.

Table 1. Summary table of computational cost estimates for different training approaches.

Plan	Parameters (Training)	VRAM (Estimate)	Tokens (Training)	TFLOPS (Total)	GPU-Hours (H100, 40% MFU)	Suitable GPUs (VRAM)
1. 1B+ARMT (context selection)	1.1B (full)	24-44+ GB	40 billion	264,000 PFLOPs	183.300	4090(48GB), A100, H100
2. 7B+BERT emb (generation)	7B (full)	76-96+ GB	15 billion (BERT)	630,000 PFLOPs	437.500	A100(80GB), H100
3. 7B+Types (generation, PEFT)	7B (70M PEFT)	35-55+ GB	100 billion	2,100,000 PFLOPs	1,458.000	4090(48GB), A100, H100

6. Conclusion

Generating and debugging code for complex, large Java projects using LLMs presents a current and unresolved challenge. Context window limitations and the difficulty of interpreting the entire codebase or full stack traces are significant constraints. In this research, we propose a novel approach based on using an Associative Recurrent Memory Transformer (ARMT) type model for intelligent selection of relevant code context and diagnostic information. This selected information is then passed to a main LLM (Gemma-3-27b-it) for code generation or correction within an iterative loop involving compilation and testing, embodied in our developing JavaCapsule system. The adaptation of benchmarks like JavaBench will be crucial for evaluation. We expect this approach to enhance the accuracy, relevance, and efficiency of automated Java code generation and debugging, opening new possibilities for applying LLMs in enterprise-level software development. Future work will focus on dataset creation, training the selection model, and experimentally validating the proposed system.

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. arXiv preprint arXiv:2406.12902, 2024.
[2]. Adnan M., Xu Z., Kuhn C. C. N. Large Language Model Guided Self-Debugging Code Generation. arXiv preprint arXiv:2502.02928, 2025.
[3]. Zhong L., Wang Z., Shang J. LDB: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. arXiv preprint arXiv:2402.16906, 2024.
[4]. Bulatov A., Kuratov Y., Burtsev M. S. Recurrent memory transformer. Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 11079-11091.
[5]. Rodkin I., Kuratov Y., Bulatov A., Burtsev M. Associative Recurrent Memory Transformer. In Proc. of the ICML 2024 Next Generation of Sequence Modeling Architectures Workshop, 2024.
[6]. 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. arXiv preprint arXiv:2406.10149, 2024.
[7]. Chen M., Tworek J., Jun H., Yuan Q., Pinto H. P. D. O., Kaplan J., ... Brockman G. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.
[8]. Li R., Allal L. B., Zi Y., Muennighoff N., Kocetkov D., Mou C., ... Li J. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161, 2023.
[9]. Hui B., Yang J., Cui Z., Yang J., Liu D., Zhang L., ... Lin J. Qwen2. 5-Coder Technical Report. arXiv preprint arXiv:2409.12186, 2024.

[10]. Gu A., Dao T. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. arXiv preprint arXiv:2312.00752, 2023.
[11]. Peng B., Alcaide E., Anthony Q., Albalak A., Arcadinho S., Cao H., ... Zhu R. J. RWKV: Reinventing RNNs for the Transformer Era. arXiv preprint arXiv:2305.13048, 2023.
[12]. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., ... Kiela D. Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 9459-9474.
[13]. Ren S., Zhou D., Zhang S., Liu S., Chen Y., Sun H., ... Liu Y. CodeBLEU: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297, 2020.

Информация об авторах / Information about authors

Владимир Игоревич ВАСИЛЕВСКИЙ – стажер-исследователь Лаборатории Облачных и Мобильных Технологий Факультета Компьютерных Наук НИУ ВШЭ. Сфера научных интересов: большие языковые модели, генерация и отладка кода, обработка длинных последовательностей, компиляторы.

Vladimir Igorevich VASILEVSKIY is a research assistant at the Cloud and Mobile Technologies Laboratory of the Faculty of Computer Science, HSE University. His research interests include large language models, code generation and debugging, long sequence processing, and compilers.

Дмитрий Владимирович АЛЕКСАНДРОВ – профессор департамента программной инженерии факультета компьютерных наук НИУ “Высшая школа экономики”, заведующий научно-учебной лаборатории облачных и мобильных технологий. Сфера научных интересов: методы и технологии искусственного интеллекта, машинное обучение и анализ данных, iOS разработка, разработка мобильных приложений, разработка программного обеспечения, indoor навигация, базы данных, разработка игр.

Dmitry Vladimirovich ALEXANDROV is a Professor in the Department of Software Engineering, Faculty of Computer Science, National Research University “Higher School of Economics”. He is also the Head of the Research and Educational Laboratory of Cloud and Mobile Technologies. His research interests include methods and technologies of artificial intelligence, machine learning and data analysis, iOS development, mobile application development, software development, indoor navigation, databases, game development.