DOI: 10.15514/ISPRAS-2026-38(3)-9

An Approach to LLM-based Code Completion for Python

[1] *M. V. Volkov, ORCID: 0000-0001-8672-7750 <mvvolkov@mail.ru>*
[1] *A. S. Bozhnyuk, ORCID: 0009-0003-4826-6609 <bozhnyuks@mail.ru>*
[2] *D. V. Vasina, ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[3] *V. A. Vasilyev, ORCID: 0009-0003-2375-0867 <djsuprin@yandex.ru>*
[1] *N. V. Tropin, ORCID: 0009-0006-2910-3961 <niktrop@yandex.ru>*
[2] *M. B. Nikitin, ORCID: 0009-0008-1130-599X <nikitinm117@gmail.com>*
[1] *D.V. Koznov ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>*

[1] *Saint-Petersburg State University,*
7/9, Universitetskaya emb., St. Petersburg, 199034, Russia

[2] *ITMO University,*
bldg. A, 49, Kronverksky Ave., St. Petersburg, 197101, Russia.

[3] *Dubna State University,*
19, Ulitsa Universitetskaya, Dubna, Moscow Oblast, 141982, Russia.

Abstract. Code completion is a critical feature in integrated development environments (IDEs) that boosts developer productivity by suggesting relevant code snippets. Large Language Models (LLMs) are a promising approach to implement this functionality. However, existing LLMs face difficulties dealing with project-wide contexts. This paper introduces a novel approach for Repository-Level Code Completion for Python using the IDE Code Model and RAG. The method constructs an LLM prompt consisting of two components: the In-File Context, which integrates type information and relevant snippets from the current file, and the Repository Context, which retrieves pertinent code from the broader project via specialized retrievers. Experiments conducted on the RepoEval-Updated benchmark using two LLMs (CodeGemma-2b and Qwen-2.5-Coder-14b) showed significant improvements with Exact Match (EM) improving by up to 30% for CodeGemma-2b and 7% for Qwen-2.5-Coder-14b comparing with state-of-the-art approaches No-RAG, Shifted-RAG and GraphCoder. Ablation studies confirmed the In-File Context's critical role and the advantage of combining retrievers, especially for API-Level tasks. These findings demonstrate the value of incorporating Python-specific code semantics into LLM-based completion systems. The method's generalizability across LLM sizes suggests broad applicability.

Keywords: Repository-Level Code Completion; Large Language Model (LLM); Retrieval Augmented Generation (RAG); Integrated Development Environment (IDE); Code Model.

For citation: Volkov M.V., Bozhnyuk A.S., Vasina D.V., Vasilyev V.A, Tropin N.V., Nikitin M.B., Koznov D.V. An Approach to LLM-based Code Completion for Python. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 1, 2026, pp. 153-170. DOI: 10.15514/ISPRAS-2026-38(3)-9.

Автодополнение кода на основе LLM для языка программирования Python

[1] *Волков М.В., ORCID: 0000-0001-8672-7750 <mvvolkov@mail.ru>*
[1] *Божнюк А.С., ORCID: 0009-0003-4826-6609 <bozhnyuks@mail.ru>*
[2] *Васина Д.В., ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[3] *Васильев В.А., ORCID: 0009-0003-2375-0867 <djsuprin@yandex.ru>*
[1] *Тропин Н.В., ORCID: 0009-0006-2910-3961 <niktrop@yandex.ru>*
[2] *Никитин М.Б., ORCID: 0009-0008-1130-599X <nikitinm117@gmail.com>*
[1] *Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>*

[1] *Санкт-Петербургский государственный университет,*
Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7–9.

[2] *Университет ИТМО,*
Россия, 197101, г. Санкт-Петербург, Кронверкский пр., д. 49, лит. А.

[3] *Государственный университет «Дубна»,*
Россия, 141982, г. Дубна, ул. Университетская, д. 19.

Аннотация. Автодополнение кода является ключевой функцией интегрированных сред разработки (IDE), повышающей продуктивность разработчиков за счёт подсказок релевантных фрагментов кода. Большие языковые модели (LLM) представляют перспективный способ реализации этой функциональности, однако существующие LLM испытывают трудности при работе с контекстом уровня всего репозитория. В статье предлагается новый подход к автодополнению кода уровня репозитория для Python на основе модели кода IDE. Метод формирует запрос к LLM из двух компонентов: In-File Context, который объединяет типовую информацию и релевантные фрагменты из открытого файла, и Repository Context, извлекающий значимые участки кода из всего проекта. Эксперименты на бенчмарке RepoEval-Updated с использованием двух LLM – CodeGemma-2b и Qwen-2.5-Coder-14b – показали существенный прирост качества автодополнения: метрика Exact Match (EM) увеличилась на 30% для CodeGemma-2b и на 7% для Qwen-2.5-Coder-14b по сравнению с современными подходами No-RAG, Shifted-RAG и GraphCoder. Исследования подтвердили ключевую роль In-File Context и преимущество комбинирования методов поиска релевантного кода, особенно для задач уровня API. Полученные результаты демонстрируют ценность интеграции специфичной для Python семантики кода в системы автодополнения на базе LLM. Обобщаемость метода на модели разных размеров указывает на его широкую применимость. В дальнейшем возможно расширение экспериментов на дополнительные LLM и задействование большего числа возможностей модели кода. Проведённые исследования демонстрируют существенное повышение точности автодополнения кода в сложных проектах на Python при использовании предложенного метода.

Ключевые слова: автодополнение уровня репозитория; большие языковые модели (LLM); технология RAG; среда разработки IDE; модель кода.

Для цитирования: Волков М.В., Божнюк А.С., Васина Д.В., Васильев В.А, Тропин Н.В., Никитин М.Б., Кознов Д.В.. Подход к автодополнению кода на основе LLM для Python. Труды ИСП РАН, том 38, вып. 3, часть 1, 2026 г., стр. 153–170 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(3)–9.

1. Introduction

One of the crucial goals of an integrated development environment (IDE) is to make writing code faster. Code Completion assists in this by suggesting the next token or sequence of tokens that the developer can insert into the editor [1-5]. The Code Completion can be categorized into In-File Code Completion [6], where the context is limited to the user's open file, and Repository-Level Code Completion [7], where the context encompasses the entire repository (the entire project, a set of projects, etc.). Repository-Level Code Completion can improve both the quality and relevance of generated code. There are benchmarks that evaluate the quality of Code Completion, such as

HumanEval [6], RepoEval [7], RepoBench [8], CoderEval [9], CrossCodeEval [10], and RepoEvalUpdated [11].

In recent years, machine learning and neural networks have advanced rapidly and are now used in various areas of human activity, including economics and business, and software development as well [12]. In particular, large language models (LLMs), which are neural networks constructed on attention mechanism and transformer elements, show excited efficiency in processing texts (queries, prompts) [13], and look promising for various software engineering tasks. Additionally, there are LLMs specifically designed for context-aware code generation [14], such as CodeGemma-2b [15] from Google and Qwen-2.5Coder-14b [16] from Alibaba.

LLMs are also used for Code Completion [14], demonstrating high efficiency as they are trained on large volumes of program code. However, when addressing Repository-Level Code Completion, LLMs exhibit a significant decrease in the quality of code generation [10]: they struggle to handle repository-specific code and are unable to correctly handle cross-file dependencies. To overcome these difficulties, the Retrieval Augmented Generation (RAG) technique [17] was proposed, which includes retrieving relevant information from an existing knowledge base (Retrieve), augmenting the prompt to the LLM based on this information (Augmentation), and generating code based on the augmented prompt (Generation). RAG has demonstrated its effectiveness for Repository-Level Code Completion [7, 11]. Meanwhile, most existing RAG methods are oriented towards natural language processing and use word similarity metrics to retrieve relevant code fragments [7, 18], or they apply static analysis [11]. These methods are either universal [7, 18] and not sufficiently effective for individual languages, or they utilize semantic information about the code to a limited extent [11].

A large telecommunications company is developing an IDE for the Python language (Python IDE). Code Model is an essential component of Python IDE, following PSI in JetBrains IDEA [19], and collecting information about the syntax and semantics of the Python project for use by the IDE features. In particular, the Python IDE has a powerful type inference mechanism, which enables precise and rapid determination of the types of any entities in the source code.

Addressing Repository-Level Code Completion for Python, it is essential to consider Python-specific characteristics, such as file structure and code semantics, which are often tied to the inferred types within the current scope. This approach can improve the quality of Code Completion for Python based on the RAG methodology.

In this paper, we propose an approach for Repository-Level Code Completion in Python, which utilizes information about the code structure and the types of entities in the current scope during the Retrieval and Augmentation stages.

The main contributions of this article are as follows:

- A new approach is proposed to support Repository-Level Code Completion using IDE Code Model.
- We developed a pilot implementation of the approach for the Python IDE, and experiments were conducted to compare various approaches and strategies for Repository-Level Code Completion. RepoEval-Updated [11] benchmark was used. We evaluated the following LLMs: CodeGemma-2b [15] and Qwen-2.5Coder-14b [16].
- The experiments revealed key aspects of prompt design for Repository-Level Code Completion in Python.

2. Related work

Most LLMs utilize the transformer architecture, which is based on the attention mechanism [20]. The latter allows the model to determine the importance of some words relative to others in the current context, significantly improving the quality of text generation. LLMs have found applications in a wide variety of human activities [21-22]. There are LLMs that are trained on large

volumes of program code, such as GitHub repositories [6]. LLMs like CodeGemma-2b [15] and Qwen-2.5Coder-14b [16] show promising results in tasks related to program code processing, such as Code Completion [14], Test Generation [23], and Program Repair [24].

In the context of Code Completion, such LLMs solve the Fill-In-The-Middle (FIM) [25], which involves generating text at a specific position. The LLM is provided with the text before the position (prefix) and optionally the text after the position (suffix) as a prompt. Modern models for Code Completion, such as CodeGemma-2b, support the FIM format–a prompt format in which special tokens denote the beginning of the prefix and the beginning of the suffix. Fig. 1 shows an example of a prompt in the FIM format.

```
path/file.py↵
<|fim_prefix|>prefix<|fim_suffix|>suffix
<|fim_middle|>
```

Fig. 1. An example prompt for LLM in the FIM format.

Despite all the advantages, LLMs have certain limitations. They require a large number of resources, making it difficult to deploy multiple LLMs simultaneously. Additionally, LLMs can accept only a limited amount of data in the prompt. However, the most significant limitation is that LLMs can only use the data they were trained on for generation. As benchmark results like CrossCodeEval [10] show, the lack of knowledge about the specifics of a particular project or about cross-file dependencies leads to a significant deterioration in the quality of the generated code.

2.2 Repository Level Code Completion

Repository-Level Code Completion [7] involves generating a relevant token or sequence of tokens based on the context of the entire repository. This task differs from In-File Code Completion [6] in that it requires considering not only the context of the current file but also the specifics of the entire repository, including the cross-file dependencies. Since software development typically adheres to the modular design principle [26], program code is distributed across multiple interconnected files. Consequently, Repository-Level Code Completion could provide significant improvement in the quality of the generated code.

There are numerous benchmarks for Repository-Level Code Completion: RepoEval [7], CrossCodeEval [10], CoderEval [9], RepoBench [8], and RepoEvalUpdated [11]. These benchmarks typically mask a portion of the code and then invoke Code Completion. Subsequently, metrics are calculated. Some metrics are string-based (Edit Similarity, Exact Match) [7, 11], while others involve running a set of tests to verify that the semantics of the original code are preserved [9]. LLMs demonstrate quite promising results for Code Completion, however, their performance is more limited in the case of Repository-Level Code Completion due to the lack of project-specific information. To overcome these challenges, methods such as CoCoMic [27] and RepoFusion [28] have been proposed, which are based on the fine-tuning technique, i.e. further training the LLM on a more extensive dataset tailored to a specific domain. However, this method is difficult to apply to closed-source LLMs. Moreover, the dynamic nature of software projects, driven by continuous development, necessitates frequent retraining of the LLM, which can be labor-intensive. Other methods attempt to use pre-trained LLMs while proposing a post-processing system [29] that adjusts the probability of the next token based on information about all tokens in the repository. However, such methods are sensitive to manual parameter tuning, making them cumbersome to operate. Pre-

processing methods are more widely used, enriching relevant information to the LLM prompt based on Retrieval Augmented Generation (RAG).

2.3 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique that aims to integrate domain-specific knowledge into Large Language Models (LLMs) [17]. RAG consists of three steps:

1. **Retrieve:** searching for relevant information from a specific knowledge base.
2. **Augmentation:** presenting the retrieved information in the required format and adding it to the LLM prompt.
3. **Generation**: generating text based on the augmented prompt.

In the context of the Repository-Level Code Completion task, this technique typically involves forming the context of the current file where Code Completion occurs, searching for relevant code snippets, and enriching the prompt with them to improve the LLM's performance. Many existing methods, such as RepoCoder [7] and ReAcc [18], use word similarity metrics to search for relevant code snippets. RLCoder [30] trains a semantic retriever via reinforcement learning from perplexity-based feedback and uses a stop-signal mechanism to include cross-file context only when it is beneficial, avoiding the need for labeled retrieval supervision. These approaches treat program code as ordinary text and ignore the specifics of code structure. Other approaches attempt to address this issue. For example, the aforementioned CoCoMic [27] and RepoHyper [31] construct a graph of methods and use it to improve retrieval. However, they do not account for the statement level and miss important contextual information for Code Completion. GraphCoder tries to solve this problem by building a Code Context Graph using a Control-Flow Graph and a Data-Flow Graph and running a search on it. However, this approach also does not consider the specifics of the Python language and uses code semantics information in a limited way. For instance, GraphCoder does not account for variable types in the current scope.

Before moving on to the proposed approach, it is necessary to consider the Python IDE and its Code Model.

3. Python IDE: Code Model

A large telecommunication company is developing a Python IDE that provides a number of IDE features such as code navigation, static code analysis, code refactoring, and code completion, which greatly simplify software development in Python.

All IDE features are based on the **Code Model** which is responsible for representing the syntax and semantics of the program's source code in the IDE. The Code Model provides the following capabilities:

- **Obtaining a syntactic representation** of each file in the form of abstract syntax trees (ASTs), which allow IDE features to work with the structure of Python code. Each AST node is mapped to a region in the Python file, enabling quick retrieval of the necessary AST node based on the cursor position in the IDE. A similar property is provided by the Program Structure Interface (PSI), which is a program syntax tree used in JetBrains' IDEs [19].
- **Reference resolution**, aiming to retrieve code entities (functions, classes, etc.) by name for IDE services.
- **Type inference**–a powerful mechanism that allows obtaining the type of any entity in the code. For example, the type of a class instance allows retrieving its fields and methods, and the type of a function allows retrieving the types of its arguments and return value.

Many IDE features, such as Code Completion, must operate quickly enough to instantly meet user needs [2]. The Code Model relies on specialized data structures that enable IDE features to retrieve all relevant information about the syntax and semantics of the source code as quickly as possible.

The Code Model is an IDE component with a wide range of capabilities, which allows quickly obtaining a lot of information about Python code. Consequently, the Code Model is crucial for Repository-Level Code Completion.

4. Approach

4.1 Overview

Fig. 2 illustrates the overall workflow of the proposed approach. When a developer types code in the IDE, the system receives data about the open file (including the file path and the text of the file) as well as the cursor position in the editor. Finally, a prompt is constructed from **In-File Context** and the **Repository Context** and passed to the LLM for code generation.

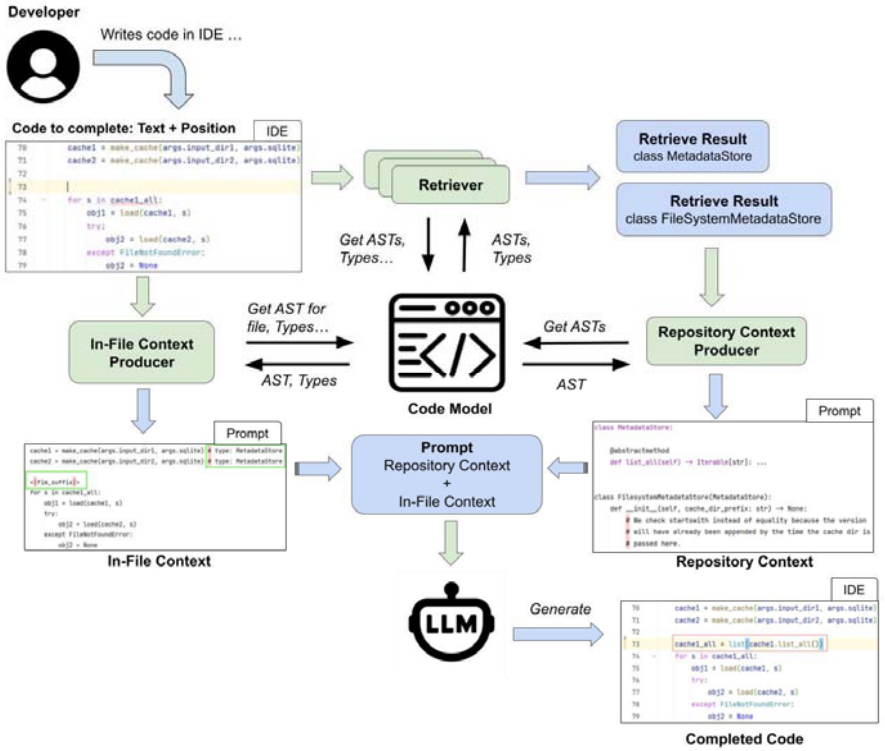

Fig. 2. An overview of the proposed approach.

The In-File Context serves as a textual representation for the LLM of the file currently open in the IDE, detailing the location where Code Completion is invoked and presenting the code surrounding that location. The In-File Context Producer constructs the In-File Context by selecting the most relevant portions of the file, discarding irrelevant sections, and incorporating supplementary information about the Python code adjacent to the Code Completion invocation point to assist the LLM in generating higher-quality code.

The Repository Context provides a textual representation to the LLM of code located outside the currently open file in the IDE, which can nonetheless enhance the quality of Code Completion. The construction of the Repository Context involves retrieving relevant variables, functions, and classes using the Code Model, which are then processed by the Repository Context Producer to form the

Repository Context. Finally, the prompt is formed by concatenating the Repository Context and the In-File Context and is subsequently passed to the LLM.

4.2 Building In File Context

The In-File Context Producer constructs the first part of the prompt for the LLM based on the text of the file open in the IDE and the cursor position where Code Completion is invoked. The purpose of this component is to create the most relevant context for the LLM using the file's text and the cursor position. This is accomplished through the following steps:

1. **Determining the Scope:** the scope refers to the current code block in which Code Completion is occurring. The In-File Context Producer queries the Code Model to obtain the Abstract Syntax Tree (AST), which is then used to identify the scope.
2. **Inferring Types of Variables and Arguments within the Scope:** using the AST, the In-File Context Producer collects variables, arguments, and functions within the scope to request their types from the Code Model. The inferred types are appended to the variables as comments, as demonstrated in Figure 2.
3. **Slicing the Text of the Current File:** with the assistance of the AST, the In-File Context Producer traverses the current file and segments it into logical units (such as functions, classes, methods, imports, etc.). Each segment (excluding the scope itself) is assigned a cost. The cost is calculated based on the nature of the entity within the logical segment (e.g., imports are assigned a higher cost than comments) and calculated using the BM-25 function [32]. The contents of the logical segments serve as the texts, while the code within the scope acts as the query. During prompt construction, segments with low costs are discarded.

Thus, by leveraging the Code Model, the construction of the In-File Context enables the utilization of Python-specific information, such as file structure and inferred types.

4.3 Building Repository Context

Constructing process of the second part of the prompt (the Repository Context) comprises two following stages:

1. **Retrieving code entities that are relevant to the current scope**, i.e. functions, classes, and variables outside the file currently open in the IDE that are most relevant to the current scope. This task is executed by the **Retriever** components.
2. The **Repository Context Producer** transforms the retrieval results into a textual representation optimally suited for the LLM.

The following Retrievers are used at the first stage:

- **Local Data Retriever** uses the Abstract Syntax Tree (AST) of the current file to identify the scope. It then queries the Code Model to determine the types of function arguments (if the code is being written within a function) and the types of variables within the scope. If the inferred type is a function, it collects the types of its arguments and return values; if it is a class, it gathers the types of its fields, methods, and parent classes. These types are subsequently returned as retrieval results.
- **Import Resolve Retriever** collects all entities imported into the current file. By leveraging the AST, it gathers the imports and resolves their names using the Code Model.
- **Structured BM-25 Retriever** operates based on a search mechanism employing the BM-25 function. As previously noted, BM-25 is a ranking function that evaluates the relevance of a document to a query. The Code Model is used to construct a corpus of documents, where each document is a code snippet. This document corpus is built by analyzing the AST of all project files during the IDE startup phase to generate structured code snippets.

Each Retriever ultimately produces a ranked list of retrieval results. The top N results are selected and passed to the Repository Context Producer. Experimental analysis has determined that the optimal value is N = 5. The Repository Context Producer constructs the Repository Context based on the retrieval results. The textual representation of each retrieval result (i.e., the code of a class, function, or variable) is transformed using the Code Model. These textual representations are segmented and evaluated using the AST and BM-25 similar to the construction of the In-File Context. Additionally, inferred types are appended where applicable (e.g., for functions, type hints for arguments and return values are included). Thus, the search and formation of the Repository Context leverage information specific to the Python language, which can be readily extracted from the Code Model. Finally, the Repository Context is enriched with the In-File Context to form the complete prompt.

5. Evaluation

To evaluate the proposed approach, an experimental study was conducted. The following research questions were specified:

- **RQ-1 (Effectiveness):** How does the quality of the proposed approach compare with other LLM-oriented approaches when evaluated on different LLMs?
- **RQ-2 (Ablation):** How do individual steps contribute to the overall approach performance?

5.1 Dataset

We used RepoEval-Updated benchmark [11] – an improved version of RepoEval [7]. This is one of the the most recent open-source benchmarks for Repository-Level Code Completion.

The RepoEval-Updated dataset is based on ten real-world Python projects. However, these projects were not suitable for our study because they were published in 2022–2023. The LLMs used here were released in 2024, which introduces a risk of data leakage if projects from the dataset appear in the models' training corpora (e.g., Qwen-2.5-Coder-14b was released on November 11, 2024). This could distort the results. Therefore, we reused the structure of RepoEval-Updated but collected a new set of ten Python projects published after November 11, 2024, to eliminate potential training-data interference. A Python script using the GitHub API was developed to gather repositories and their metadata. Repositories were selected according to the following criteria:

1. At least 60% of the code in the repository is written in Python;
2. The project contains at least 10 Python source files;
3. The repository is a full-fledged Python project (not just a set of isolated scripts), with cross-file relationships and dependencies on external libraries;
4. The date of the first commit is no earlier than November 11, 2024.

We checked criteria 1-3 by developing a special script; the last one was verified manually by inspecting code and dependency of the projects' files. In total, 10 Python repositories meeting these requirements were collected; their characteristics are shown in Table 1. Table 1 also reports the commit SHAs used for each repository to ensure reproducibility, so that the experiments are run against the exact same project versions.

We formed a dataset consisting of tasks based on the collected repositories. Each task describes the location where Code Completion is invoked: a file path and a cursor position.

These tasks are divided into the following categories:

1. **Line-Level tasks** are generated by deleting random lines of code from Python files. The removed code must contain at least five tokens and must not be a comment.

2. **API-Level tasks** are also generated by deleting lines of code, but the removed code must include at least one function, method, variable, or class name that is not defined in the current file.

For each task, the system must generate a line of code to replace the removed (target) line. Quality is computed by comparing the generated line with the target line using the metrics described in Section 5.2.

200 tasks of each category were created for each Python project, and the final dataset includes 4,000 tasks.

Table 1. Collected Repositories.

Repository	#Files	First commit date	Commit SHA
wassim249/YT-Navigator	60	2025-08-03	61e3ddfbb8c52ed23c01a03cc73498fb48bc7a5a
microsoft/markitdown	52	2024-11-13	041be5447148e7455d38af986b1fc64bc420b62a
browser-use/web-ui	52	2025-01-02	383b04ab5fc3a67938b6964c9ba4c4c45a7f67b8
AghastyGD/lazy-ninja	21	2025-02-09	bf25b62dc800f35e46c303023663d657c48d7ba1
camel-ai/owl	74	2025-03-04	58102ac75baf3f8fc1e24e000a17a6fbedb00ac3
FoundationAgents/OpenManus	72	2025-03-06	8d23b06e46bf56556f7323ef837ad0c834868368
huggingface/smolagents	65	2024-12-05	4f877034dc7c24377c380edc1d344d3086a5c60c
corbado/passkeys-python-django	24	2025-01-08	7497ac255308550df2c456ce70c1267aafd94eb9
huggingface/open-r1	33	2025-01-24	8cf42663fdc09b82d2593ac86e4748fa7d1b38fb
openai/gpt-oss	62	2025-06-23	9e5b84198755a6e3d89d5f63f96c9bdef6ea3d84

5.2 Evaluation Metrics

The experiments were evaluated using the following groups of metrics:

- **Code Match** is a group of metrics based on code string similarity and includes:
  - *Exact Match (EM):* a binary metric that equals 1 if the generated code equals the removed code and 0 otherwise.
  - *Edit Similarity (ES):* a more precise metric that measures the similarity between two strings, calculated using the formula $ES = 1 - Lev(s, s') / max(|s|, |s'|)$, where *Lev* is the Levenshtein distance [33].
- **Identifier Match** is a group of metrics based on collecting identifiers from the code strings and comparing them using:
  - *Exact Match (EM):* a complete match of all identifiers.
  - *F1 Score (F1):* a harmonic mean of precision and recall for sequences of strings.

5.3 State of the art approaches

Since the proposed approach emphasizes code syntax and semantics rather than string similarity and limited parsing, we compare it with the following three methods:

- **No-RAG.** A baseline for Code Completion that sends the text of the current file to the LLM. If the file exceeds the LLM's context window, a region around the code-completion trigger position is extracted.
- **Shifted-RAG.** A baseline that retrieves similar code snippets from a knowledge base using a fixed-size sliding window and BM25. In this study, the window size is 20 with a stride of 1. This method is referenced in ReACC [18].
- **GraphCoder.** A more advanced method that builds a code-context graph and uses it to retrieve relevant code fragments [11].

In selecting these methods, we followed the procedure described in the GraphCoder paper [11], excluding Vanilla-RAG, which is a simplified version of Shifted-RAG that consistently underperforms on benchmarks. We also excluded RepoCoder and used GraphCoder instead, since GraphCoder reports better results than RepoCoder; moreover, RepoCoder relies on sliding-window retrieval combined with regeneration, which places it in a different class of algorithms.

We also considered more recent approaches such as RepoHyper and RLCoder. Because our work targets a lightweight, IDE-integrated pipeline with commercially available LLMs and reproducible, prompt-only baselines, reproducing these training-intensive setups (and ensuring a fair comparison) was outside the scope of our experimental setting. Finally, we excluded IDECoder because its authors did not release source code.

5.4 Generation

The following LLMs were used for code generation:

- **CodeGemma-2b** (Google), released on April 8, 2024 [15].
- **Qwen-2.5-Coder-14b** (Alibaba), released on November 11, 2024 [16].

These models were selected as up-to-date code-oriented LLMs released in 2024. Both are trained for Code Completion and support the Fill-in-the-Middle (FIM) prompt format. They are also comparatively accessible in terms of deployment cost, which is relevant given the GPU memory requirements.

Each LLM was deployed on a dedicated server equipped with two NVIDIA GeForce RTX 4080 GPUs. In the experiments, the maximum length of the generated completion was set to 100 tokens, and the temperature was set to 0 to ensure reproducibility.

5.5 Results and Discussion

5.5.1 RQ 1 (Effectiveness)

The comparative results with the baselines are presented in Table 2.

The results indicate not only overall superiority of the proposed method over the baselines but also several patterns. First, the large gains on CodeGemma-2b (especially for API-Level tasks) suggest that integrating RAG with a Python IDE Code Model helps the LLM capture project specifics. Outperformance over Shifted-RAG and GraphCoder shows that the method leverages available project information more effectively by selecting context via AST, type inference, and other code-model capabilities.

The difference in improvement magnitude between CodeGemma-2b and Qwen-2.5-Coder-14b can be attributed to differences in base capability. As a larger model, Qwen-2.5-Coder-14b starts from a stronger baseline, leaving less headroom; nevertheless, the method still yields measurable gains, which supports its robustness.

Table 2. Comparison of the proposed approach with baselines (all metrics in %).

Task	Approach	CodeGemma-2b				Qwen-2.5-Coder-14b			
		Code Match		Identifier Match		Code Match		Identifier Match	
		EM	ES	EM	F1	EM	ES	EM	F1
Line-Level	No-RAG	19.95	30.20	27.60	29.77	51.85	69.93	59.35	68.26
	Shifted-RAG	30.05	51.36	38.60	49.71	52.90	70.27	60.25	68.41
	GraphCoder	36.90	56.64	43.60	53.49	53.55	71.27	60.85	69.48
	Proposed	**51.75**	**72.94**	**60.25**	**71.07**	**58.25**	**74.10**	**65.00**	**71.86**
API-Level	No-RAG	16.89	27.01	19.77	28.38	45.80	64.83	48.99	66.85
	Shifted-RAG	23.66	46.27	27.35	47.67	46.97	65.80	50.40	67.55
	GraphCoder	28.70	50.38	31.55	49.86	48.28	66.36	51.47	68.30
	Proposed	**47.07**	**68.99**	**50.86**	**70.41**	**53.03**	**69.71**	**56.27**	**71.86**

Answer to RQ-1. The proposed approach outperforms, across all metrics, the universal Shifted-RAG method based on string-similarity retrieval and the GraphCoder method that employs code information in a limited way.

5.5.2 RQ 2 (Ablation)

We evaluated the following configurations (see Table 3):

- **Retrievers (Retrieve/No-Retrieve).** *Retrieve* enables the repository-context builder that fetches relevant entities from other project files to enrich the completion context. *No-Retrieve* disables this mechanism, restricting the prompt to the open file only.
- **Retrieve Count (N).** The number of retrieved results used to build the repository context. We tested **N = 5, 10, 15, 100**.
- **In-File Context Producer (IF/No-IF).** *No-IF* forms the open-file context by extracting a fixed code window around the completion point. *IF* constructs the context using syntax and semantics of the code as described in Section 4.2.
- A crucial finding is that careful construction of the In-File Context (IF) has the largest impact on code-completion quality, and this effect holds for both CodeGemma-2b and Qwen-2.5-Coder-14b. Because Python files are often sizable, selecting only the most relevant code fragments is critical to reduce redundancy and fit the LLM's context window. The proposed method addresses this by focusing on the parts most closely related to the completion site and augmenting them with semantic information (e.g., inferred types and structure beyond the open file), which substantially improves the accuracy and relevance of generated code.
- Adding repository context via retrievers provides additional gaëins, but only when a strong In-File Context is present, and this trend also holds for both LLMs. These improvements indicate that information from other files (e.g., imported modules or base classes) can materially help, especially on API-Level tasks where cross-file dependencies are central.

Table 3. Results of different approach configurations for Line-Level and API-Level tasks (in %).

Task	Configuration	CodeGemma-2b				Qwen-2.5-Coder-14b			
		Code Match		Identifier Match		Code Match		Identifier Match	
		EM	ES	EM	F1	EM	ES	EM	F1
Line-Level	No-Retrieve / No-IF	19.95	30.20	27.60	29.77	51.85	69.93	59.35	68.26
	No-Retrieve / IF	46.05	69.24	55.35	67.79	54.75	72.72	62.25	71.12
	Retrieve (N = 5) / No-IF	20.80	30.63	28.40	30.21	52.55	70.26	60.05	68.65
	Retrieve (N = 10) / No-IF	20.80	30.63	28.40	30.21	52.60	70.23	60.05	68.54
	Retrieve (N = 15) / No-IF	20.80	30.63	28.40	30.21	52.55	70.30	60.00	68.59
	Retrieve (N = 100) / No-IF	20.80	30.65	28.40	30.24	52.50	70.25	59.95	68.63
	Retrieve (N = 5) / IF	**51.75**	**72.94**	**60.25**	**71.07**	**58.25**	**74.10**	**65.00**	**71.86**
	Retrieve (N = 10) / IF	51.70	72.95	60.30	71.07	58.15	74.10	65.10	71.92
	Retrieve (N = 15) / IF	51.75	72.99	60.30	71.09	58.25	74.18	65.20	71.95
	Retrieve (N = 100) / IF	51.65	72.84	60.15	71.02	58.35	74.09	65.10	71.90
API-Level	No-Retrieve / No-IF	16.89	27.01	19.77	28.38	45.80	64.83	48.99	66.85
	No-Retrieve / IF	41.71	65.47	45.80	67.21	48.69	67.30	51.57	69.33
	Retrieve (N = 5) / No-IF	18.05	27.75	20.73	29.02	46.51	65.36	49.75	67.30
	Retrieve (N = 10) / No-IF	18.05	27.75	20.73	29.02	46.71	65.36	49.85	67.29
	Retrieve (N = 15) / No-IF	18.05	27.75	20.73	29.00	46.41	65.20	49.49	67.11
	Retrieve (N = 100) / No-IF	18.05	27.75	20.73	29.02	46.61	65.29	49.75	67.16
	Retrieve (N = 5) / IF	**47.07**	**68.99**	**50.86**	**70.41**	**53.03**	**69.71**	**56.27**	**71.86**
	Retrieve (N = 10) / IF	47.02	68.95	50.81	70.37	53.08	69.76	56.37	71.91
	Retrieve (N = 15) / IF	47.07	68.99	50.86	70.41	53.08	69.76	56.27	71.89
	Retrieve (N = 100) / IF	47.02	68.96	50.81	70.37	53.03	69.75	56.27	71.88

Repository context ceases to matter without IF. EM gains from Retrievers without IF are only +0.85% on Line-Level and +1.16% on API-Level for CodeGemma-2b. Qwen-2.5-Coder-14b shows similarly minimal effects. The open-file snippet alone tends to fill the prompt budget for large Python files, leaving no room for external context. GraphCoder attempts to mitigate this by reserving

only half of the prompt for the current file, but it does not perform fine-grained selection of in-file fragments, which degrades completion quality. These observations underscore the need for precise open-file context construction.

Another consistent observation across both LLMs is the effect of Retrieve Count (N): increasing N to 100 does not improve results and can slightly worsen them. This emphasizes the balance between context volume and relevance: overly large contexts introduce noise and make it harder for the LLM to focus on key code fragments.

Answer to RQ-2. Building the open-file context is the key driver of quality improvements. Repository context yields additional gains (particularly on API-Level tasks), but loses impact without a strong in-file context. The retrieve-count parameter N requires careful tuning; excessive values can harm performance. These findings are consistent across both evaluated LLMs.

5.5.3 Error Analysis

In addition to metric-based evaluation, we provide a qualitative discussion of typical failure modes observed in Repository-Level Code Completion. The following categories are the most common and informative for interpreting the results.

- **Wrong method/attribute selection.** The completion refers to an existing variable or object but selects an incorrect method or attribute (often a "plausible" API such as to_dict() vs. as_dict()). This failure mode is common when the inferred type is too coarse (e.g., a base class or Any), multiple candidate types remain possible within the scope, or the relevant method exists only in a subclass defined elsewhere in the repository. The Local Data Retriever reduces these cases by exposing fields and methods of the inferred types in the prompt, yet the remaining errors reflect limitations of static type inference in Python and ambiguity of runtime behavior.
- **Hallucinated symbols.** The model sometimes invents helper functions, constants, or exception names that are consistent with naming conventions but do not exist in the repository. This effect is most noticeable in API-Level tasks, where the removed line requires referencing identifiers defined outside the current file. The primary cause is insufficient grounding: if the defining snippet is not present in the constructed prompt (either because it was not retrieved or was dropped during slicing), the LLM may fall back to a "reasonable" guess. Increasing the retrieved context volume is not always beneficial (Section 5.5.2), since excessive retrieval can introduce noise and reduce the effective salience of truly relevant definitions.

Overall, these failure modes indicate that Repository-Level Code Completion quality is limited not only by the generative capability of the LLM, but also by the precision of symbol resolution, type inference, and the relevance–noise tradeoff in context construction. A systematic, quantitative study of how type injection affects hallucination rates is left for future work; nevertheless, type information is expected to reduce guesswork when inferred types are precise, while inaccurate or overly generic inferred types may provide limited guidance.

5.6 Threats to Validity

The main threat to the validity of the experimental study lies in the implementation of the baselines used for comparison. We relied on the publicly available code of GraphCoder and RepoCoder. For Shifted-RAG, no reference implementation was available, so it was implemented by the author based on the description in the literature. In the original GraphCoder work, different LLMs were used to evaluate Code Completion quality; therefore, we modified the GraphCoder code to integrate CodeGemma-2b and Qwen-2.5-Coder-14b.

We identified a data-leakage issue during GraphCoder experiments: the prompt sent to the LLM could inadvertently contain the ground-truth line that had been removed from the current file. The

cause was that, when forming the repository context, GraphCoder's retrieval pipeline could return code fragments originating from the same file as the completion site. We prevented this leakage by filtering out any same-file fragments from the repository context.

6. Conclusions

The experiments confirm the effectiveness of the proposed Repository-Level Code Completion approach for Python. Integrating the Python IDE Code Model with a retrieval-augmented pipeline substantially improves code-generation quality by exposing the LLM to syntax- and semantics-aware context. The In-File Context is the main driver of these gains, while the Repository Context complements it by capturing cross-file dependencies, which is especially beneficial for API-Level tasks.

The magnitude of improvements differs across models. The smaller CodeGemma-2b benefits most from the in-file context and retrievals, indicating a stronger dependence on external context, whereas Qwen-2.5-Coder-14b shows smaller but consistent gains. These consistent trends across LLMs support the method's robustness. We did not evaluate Function-Body Completion (masking function bodies and checking unit tests) due to effort constraints, but the Line-Level and API-Level results already demonstrate the advantage of leveraging Python-specific semantics via the Code Model.

Future research should aim to expand experiments to a wider range of LLMs to validate the generalizability of our findings. Additionally, exploring alternative ways to utilize the Code Model (such as incorporating more sophisticated type inference or advanced code analysis) could refine the system's capabilities. Another promising direction is to integrate code-completion tools into specialized IDEs (for the development of embedded real-time systems [34], broadcasting systems [35], etc.), and evaluate the effectiveness of the proposed approach in those settings, adapting it where necessary.

References

[1]. Amann S., Proksch S., Nadi S., Mezini M. A study of visual studio usage in practice. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, 2016, pp. 124-134.
[2]. Semenkin A., Bibaev V., Sokolov Y., Krylov K., Kalina A., Khannanova A., Savenkov D., Rovdo D., Davidenko I., Karnaukhov K., Vakhrushev M., Kostyukov M., Podvitskii M., Surkov P., Golubev Y., Povarov N., Bryksin T. Full line code completion: Bringing AI to desktop. arXiv preprint arXiv:2405.08704, 2024.
[3]. Korada L. GitHub Copilot: The disrupting AI companion transforming the developer role and application lifecycle management. Journal of Artificial Intelligence & Cloud Computing, vol. 3, pp. 1-4, 2024.
[4]. Biswas S. Tabnine AI tool and comparison with similar tools, 2023.
[5]. Ramamoorthy L. AI for software engineering – enhancing developer experience with Codeium and Copilot. International Journal of Science and Research (IJSR), vol. 14, 2025.
[6]. Chen M. et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.
[7]. Zhang F., Chen B., Zhang Y., Liu J., Zan D., Mao Y., Lou J., Chen W. RepoCoder: Repository-level code completion through iterative retrieval and generation, arXiv preprint arXiv:2303.12570, 2023.
[8]. Liu T., Xu C. RepoBench: Benchmarking repository-level code auto-completion systems, 2023.
[9]. Yu H., Shen B., Ran D., Zhang J., Zhang Q., Ma Y., Liang G., Li Y., Wang Q., Xie T. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24), 2024, Art. 37.
[10]. Ding Y. et al. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion, arXiv preprint arXiv:2310.11248, 2023.
[11]. Liu W., Yu A., Zan D., Shen B., Zhang W., Zhao H., Jin Z., Wang Q. GraphCoder: Enhancing repository-level code completion via code context graph-based retrieval and language model, 2024.
[12]. Vasiliev R., Koznov D., Chernishev G., Khvorov A., Luciv D., Povarov N. TraceSim: a method for calculating stack trace similarity, pp. 25-30, 2020.
[13]. Achiam J. et al. GPT-4 technical report, 2023.

[14]. Husein R., Aburajouh H. Large language models for code completion: A systematic literature review. Computer Standards & Interfaces, 2024.
[15]. CodeGemma Team. CodeGemma: Open code models based on Gemma. arXiv preprint arXiv:2406.11409, 2024.
[16]. Hui B. et al. Qwen2.5-Coder technical report. arXiv preprint arXiv:2409.12186, 2024.
[17]. Lewis P. et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20), 2020, pp. 9459-9474.
[18]. Lu S. et al. ReACC: A retrieval-augmented code completion framework, 2022, pp. 6227-6240.
[19]. Kurbatova Z., Golubev Y., Kovalenko V., Bryksin T. The IntelliJ Platform: A framework for building plugins and mining software data, 2021, pp. 14-17.
[20]. Vaswani A. et al. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), 2017, pp. 6000-6010.
[21]. Scherbakov D., Hubig N., Jansari V., Bakumenko A., Lenert L. The emergence of large language models as a tool in literature reviews: an LLM-automated systematic review. arXiv preprint arXiv:2409.04600, 2024.
[22]. Qiu J. et al. LLM-based agentic systems in medicine and healthcare. Nature Machine Intelligence, vol. 6, 2024.
[23]. Lemieux C., Inala J.P., Lahiri S.K., Sen S. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In Proceedings of the 45th International Conference on Software Engineering (ICSE'23), 2023, pp. 919-931.
[24]. Chow Y., Grazia L., Pradel M. PyTy: Repairing static type errors in Python, 2024, pp. 1-13.
[25]. Bavarian M. et al. Efficient training of language models to fill in the middle. arXiv preprint arXiv:2207.14255, 2022.
[26]. Su Z., Devanbu P. On the localness of software. In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2014, pp. 269-280.
[27]. Ding Y. et al. CoCoMic: Code completion by jointly modeling in-file and cross-file context. arXiv preprint arXiv:2212.10007, 2022.
[28]. Shrivastava D. et al. RepoFusion: Training code models to understand your repository. arXiv preprint arXiv:2306.10998, 2023.
[29]. Khandelwal U. et al. Generalization through memorization: Nearest neighbor language models, 2019.
[30]. Wang Y., Wang Y., Guo D., Chen J., Zhang R., Ma Y., Zheng Z. RLCoder: Reinforcement Learning for Repository-Level Code Completion. In 47th IEEE/ACM International Conference on Software Engineering (ICSE'25), 2025, pp. 1140–1152.
[31]. Phan H.N., Phan H.N., Nguyen T.N., Bui N.D.Q. RepoHyper: Better context retrieval is all you need for repository-level code completion. arXiv preprint arXiv:2403.06095, 2024.
[32]. Robertson S., Walker S., Hancock-Beaulieu M., Gatford M., Payne A., Okapi at TREC-4, 1995.
[33]. Levenshtein V.I. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady, vol. 10, pp. 707-710, 1966.
[34]. Terekhov A.N., Romanovskii K.Y., Koznov D.V., Dolgov P.S., Ivanov A.N. RTST++: Methodology and a CASE tool for the development of information systems and software for real-time systems. Programming and Computer Software, vol. 25, no. 5, pp. 276-281, 1999.
[35]. Koznov D.V., Peregudov A.F., Bugajchenko D.Y., Chernyatchik R.I., Kazakova A.S., Pavlinov A.A. Vizual'naya sreda proektirovaniya sistem televizionnogo veshchaniya. Sistemnoe programmirovanie, vol. 2, no. 1, pp. 142-168, 2006.

Информация об авторах / Information about authors

Михаил Валериевич ВОЛКОВ – кандидат физико-математических наук, закончил физический факультет СПбГУ и аспирантуру Стокгольмского Университета. Занимался исследованиями в области квантовой механики. В данный момент профессиональные и научные интересы лежат в области разработки IDE, кодовых моделей и вывода типов для динамически типизированных языков.

Mikhail Valeryevich VOLKOV – PhD in Physics, graduated from the Physics Department of St. Petersburg State University and completed his postgraduate studies at Stockholm University. He

was engaged in research in the field of quantum mechanics. At the moment his professional and scientific interests are in the field of IDE development, code model and type inference for dynamically typed languages.

Александр Сергеевич БОЖНЮК – программист в компании 2ГИС. Закончил математико-механический факультет СПбГУ по направлению «Программная инженерия», а также магистратуру на факультете Математики и Компьютерных Наук по направлению «Разработка ПО и науки о данных». В сферу научных и профессиональных интересов входит разработка инструментов статического анализа и рефакторингов кода, теория компиляции, теория виртуальных машин, внедрение LLM в бизнес-процессы.

Alexander Sergeevich BOZHNYUK is a software engineer at 2GIS company. He graduated from the Faculty of Mathematics and Mechanics of St. Petersburg State University with a major in Software Engineering, and also completed a master's program at the Faculty of Mathematics and Computer Science in Software Development and Data Science. His research and professional interests include the development of static analysis and code-refactoring tools, compilation theory, virtual machine theory, and the integration of LLMs into business processes.

Дарья Владимировна ВАСИНА – ведущий инженер Санкт-Петербургской лаборатории средств разработки облачного ПО. Окончила факультет компьютерных технологий и управления Университета ИТМО по направлению «Информатика и вычислительная техника». Специализируется на создании инструментов для разработки программного обеспечения с интеграцией технологий искусственного интеллекта.

Darya Vladimirovna VASINA is a Lead Engineer at the St. Petersburg Cloud Software Development Tools Laboratory. She graduated from the Faculty of Computer Technologies and Control at ITMO University, majoring in Computer Science and Engineering. She specializes in creating software development tools with the integration of artificial intelligence technologies.

Владимир Анатольевич ВАСИЛЬЕВ – магистр техники и технологий, закончил факультет системного анализа и управления государственного университета «Дубна» и аспирантуру УНЦ ОИЯИ. Более 15 лет занимается профессиональной разработкой в российских и международных коммерческих и научных организациях. Специализируется на проектировании и разработке комплексных программных продуктов и высоконагруженных систем, в том числе инструментов IDE.

Vladimir Anatolyevich VASILYEV holds a Master's degree in Engineering and Technology, graduated from the Faculty of System Analysis and Management at Dubna State University and completed postgraduate studies at the JINR University Center. He has been engaged in professional development in Russian and international commercial and scientific organizations for more than 15 years. Specializes in the design and development of complex software products and high-load systems, including IDE tools.

Николай Владимирович ТРОПИН – ведущий инженер команды Python IDE, закончил математико-механический факультет СПбГУ. Работает в области создания инструментов разработки с 2013 года.

Nikolay Vladimirovich TROPIN is the leading engineer of the Python IDE team, graduated from the Mathematics and Mechanics Faculty of St. Petersburg State University. He has been working in the field of development tools since 2013.

Максим Борисович НИКИТИН – инженер в области машинного обучения, компания Яндекс. Окончил Санкт-Петербургский государственный университет по направлению «Математика, алгоритмы и анализ данных». В настоящее время обучается в Университете ИТМО по программе «Искусственный интеллект». В сферу его научных и профессиональных

интересов входят искусственный интеллект, машинное обучение, обработка естественного языка и обучение с подкреплением.

Maxim Borisovich NIKITIN is a Machine Learning Engineer at Yandex. He graduated from St. Petersburg State University with a degree in Mathematics, Algorithms, and Data Analysis. He is currently pursuing a degree in Artificial Intelligence at ITMO University. His research and professional interests include artificial intelligence, machine learning, natural language processing, and reinforcement learning.

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования Санкт-Петербургского государственного университета. Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV – Dr. Sci. (Tech.), Assoc. Prof., professor at St. Petersburg State University. Research interests: software engineering, model-driven software development, program data, machine learning.