Записки научных
семинаров ПОМИ
Том 552, 2026 г.

O. Baryshnikov, A. M. Alekseev, S. I. Nikolenko

QUERY2DIAGRAM: ANSWERING DEVELOPER QUERIES WITH UML DIAGRAMS

Abstract. Software documentation frequently becomes outdated or fails to exist entirely, yet developers need focused views of their codebase to understand complex systems. While automated reverse engineering tools can generate UML diagrams from code, they produce overwhelming detail without considering developer intent. We introduce *query-driven UML diagram generation*, where LLMs create diagrams that directly answer natural language questions about code. Unlike existing methods, our approach produces semantically focused diagrams containing only relevant elements with contextual descriptions. We fine-tune Qwen2.5-Coder-14B on a curated dataset of code files, developer queries, and corresponding diagram representations in a structured JSON format, evaluating with both automatic detection of structural defects and human assessment of semantic relevance. Results demonstrate that fine-tuning on a modest amount of manually corrected data yields dramatic improvements: our best model achieves the highest F1 scores while reducing defect rates below state-of-the-art LLMs, generating diagrams that are both structurally sound and semantically faithful to developer queries. Thus, we establish the feasibility of using LLMs for scalable contextual, on-demand documentation generation. We make our code and dataset publicly available at `https://github.com/i-need-a-pencil/query2diagram`.

§1. Introduction

Software documentation, particularly UML diagrams, plays a crucial role in system comprehension and maintenance. Studies confirm that UML usage improves code quality and modularity, reduces defects, and increases developer productivity [1–4]. However, manually creating and, crucially, *maintaining* up-to-date diagrams requires significant effort, leading to a

Key words and phrases: Large Language Models and Code QA and UML and Documentation Maintenance.

The work of A. Alekseev and S. Nikolenko was supported by the Ministry of Science and Higher Education of the Russian Federation (agreement 075-15-2025-344 dated 29/04/2025 for Saint Petersburg Leonhard Euler International Mathematical Institute at PDMI RAS)..

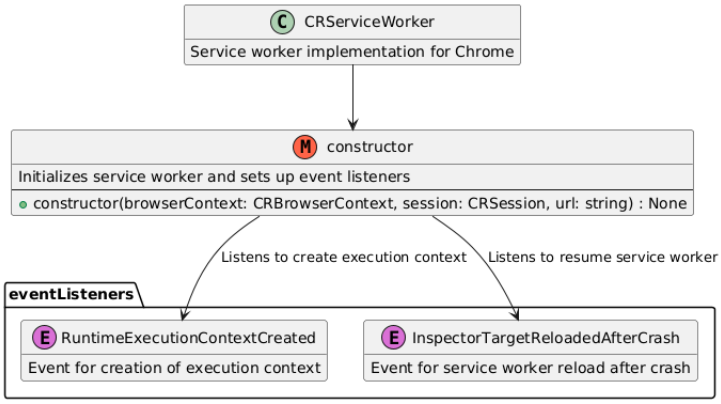

Figure 1. A sample response to the user query "Map out the event listeners set up in the CRServiceWorker constructor and their corresponding actions" based on the TypeScript file `microsoft/playwright/.../crServiceWorker.ts`.

common problem: documentation that either does not exist or diverges from the actual implementation [5].

In the absence of up-to-date UML documentation, diagrams can be generated with *automated reverse-engineering* (RE) tools. However, the resulting UML diagrams often contain overwhelming detail that hinders comprehension; studies and developer surveys show that these diagrams have to be severely simplified for clarity [6, 7]. Some approaches propose interactive filtering mechanisms [8], but typically ignore semantic context and user intent. This creates a gap: developers need focused, contextual views of their codebase, not exhaustive structural dumps.

Recent advances in NLP, specifically large language models (LLMs), offer a promising solution. LLMs excel at understanding both code and natural language, making them uniquely suited to bridge the gap between developer queries and visual representations. Although LLM applications in software engineering are expanding rapidly [9, 10], including tools such as GitHub Copilot [11] and Cursor [12], their potential for query-driven diagram generation remains unexplored.

Motivated by this gap, we present a novel approach: using LLMs to generate RE-UML diagrams that *directly answer developer queries* about

code. Unlike traditional RE tools or recent LLM-based approaches that generate complete diagrams, our method produces *query-focused* diagrams that include only relevant elements with contextual descriptions. Figure 1 illustrates this: given a query about event listeners in a TypeScript file, the system generates a targeted diagram showing only the relevant components and their relationships.

Our approach offers several key advantages: it (1) eliminates the need for pre-existing diagrams, (2) provides direct answers to developer questions through visual representation, (3) includes explanatory descriptions for all elements, and (4) supports multiple programming languages through a single model. We address two research questions:

RQ1 Can LLMs generate relevant, comprehensible graph-like structures (UML diagrams) when provided with code files and high-level queries about design patterns, component interactions, or architectural concerns?

RQ2 Can training on synthetic and curated data effectively control the quality properties of LLM-generated diagrams?

In this work, we fine-tune Qwen2.5-Coder-14B on a carefully curated dataset of code-query-diagram triples, introducing a JSON-based intermediate representation that reduces syntax errors while enabling structured generation. Our dual evaluation strategy combines automatic defect analysis with human relevance assessment. Results demonstrate that fine-tuning on manually corrected data yields diagrams that are both structurally sound and semantically relevant, achieving the best F1 scores while dramatically reducing defect rates.

Our contributions include: (1) a new task formulation for query-driven UML generation from code, (2) a practical method using fine-tuned LLMs with structured output generation, (3) a comprehensive evaluation framework combining structural and semantic metrics, and (4) empirical evidence that even small amounts of high-quality training data can significantly improve diagram quality. We release our code and dataset at `https://github.com/i-need-a-pencil/query2diagram`.

In the following, Section 2 reviews related work on diagram generation, Section 3 provides a description of our approach, Section 5 presents evaluation results and discussions, and Section 6 concludes the article.

§2. Related work

Recent studies document both the demand for and the friction in keeping diagrams useful in practice: informal diagrams dominate real projects yet are rarely updated [13]; API–scale structure benefits from interactive, automatically generated visualizations [14]; and large ecosystems require continuously refreshed, machine-generated views to track versions, dependencies, and CVEs [15]. Reverse-engineered models also drive downstream tasks beyond comprehension, such as early defect prediction from UML metrics [16] and guided system migration [17]. These findings jointly motivate *focused*, on-demand diagramming from code, rather than one-shot, complete diagrams that quickly drift from developers' information needs.

Traditional UML generation. Classical reverse engineering (RE) approaches for UML generation fall into three categories. *Static analysis* tools [18, 19] extract structural relationships without code execution; they effectively detect inheritance but struggle with runtime behaviors. *Dynamic analysis* tools such as Caffeine [20] capture runtime behavior and can discover specific object relationships, but require executable code and sufficient coverage. *Hybrid approaches* like Ptidej [21], which enhances a static model with dynamic traces, combine both but demand complex heuristics and manual configuration. Beyond comprehension, traditional RE feeds downstream uses (e.g., migration [17] or defect prediction from UML metrics [16]), yet the shared limitation remains: they generate *complete* structural views without considering user intent or information needs.

LLM-based UML generation. Recent research explores large language models (LLMs) for UML generation from both source code and natural language artifacts. [22] used GPT-4 [23] to generate class diagrams directly from codebases, reporting frequent syntax errors and missing elements. [24] embedded an LLM into a model-driven reverse-engineering (MDRE) pipeline to extract class diagrams from Java programs, while [25] fine-tuned Mistral-7B on AgileUML-derived Java/Python $\leftrightarrow$ UML/OCL pairs, achieving high precision and recall; [26] targeted explicit OCL generation from code, substantially outperforming rule-based baselines; and [27] combined LLMs with MDRE, introducing diagram-granularity levels (CLASS, COARSE, FINE) to control abstraction. [28] evaluated GPT-4o, Gemini 1.5 Pro, Claude 3.5 Sonnet, and Mistral Large on architecture recovery, showing that self-reflection prompts reduce hallucinations and omissions.

Beyond code, several studies use LLMs to translate textual specifications into UML: [29–32] focus on requirements-to-model generation, while [33] applies a hybrid NLP–DSL–MDA pipeline to produce UML and executable code.

Most related to our focus on intent conditioning, [34] showed that guiding diagram construction through natural-language queries can improve productivity and coherence—but their system assumes a pre-existing UML model rather than generating one from code.

Overall, existing LLM approaches either generate complete diagrams from source or derive them from textual specifications; none support *query-driven* diagram generation that filters and abstracts directly *from code* according to developer information needs.

Quality assessment. UML diagram quality is typically evaluated via *syntactic validation* [35] and *metric-based analysis* of structural properties [36], or by element-level precision/recall/F1 against a reference model. Code → UML/OCL work reports such structural metrics—for example, LLM-in-the-loop and fine-tuned pipelines benchmark element extraction against rule-generated ground truth [24–26], while [27] compares LLM outputs with a strong procedural baseline using granularity-aware metrics (CLASS/COARSE/FINE) and WCC. Architecture-oriented recovery additionally analyzes error taxonomies (missing/mistake/hallucination) and the effect of self-reflection prompts [28].

By contrast, NL→UML studies (from specifications rather than code) rely largely on *manual annotation* and rubric-based judgments of syntactic/semantic quality, frequently noting syntax errors or missing elements in raw LLM outputs [22, 31, 32]. Across these strands, prevailing evaluations focus on *structural correctness*, not *task relevance*: they rarely assess whether a generated diagram actually answers a developer's *query*. Practitioner evidence that informal diagrams are ubiquitous yet seldom maintained [13] and ecosystem-scale visualization needs for continuously updated, actionable overviews [15] further suggest prioritizing *focused*, on-demand views. This motivates relevance-oriented metrics that evaluate whether a diagram produced *from code* satisfies a specific information need.

Positioning our work. The literature above separates into (i) traditional analysis-based tools that output exhaustive diagrams and (ii) LLM-driven methods that either generate exhaustive diagrams from code [24,25] or operate from specifications [33], with granularity controls but no intent conditioning [27]. We introduce a different paradigm: *query-driven diagram*

generation from code, producing *focused, contextual* views guided by developer intent. Unlike classical tools, we do not emit "everything," and unlike specification-driven or architecture-only studies [28], our inputs are the source files themselves. Our evaluation complements syntax and element-matching with *relevance-oriented* metrics that ask whether the generated view answers the user's query. To the best of our knowledge, no prior work combines code input with *query-conditioned* selection and abstraction for UML generation.

Concretely, we operationalize queries over code by retrieving intent-aligned fragments and constraints, prompting an LLM to synthesize a minimal UML view, and validating the view against both structural consistency and query relevance.

§3. Method

Below we describe our workflow consisting of three stages: *data collection*, *model adaptation*, and *diagram generation*. Each stage is deliberately lightweight so that new code bases or languages can be added with minimal manual effort.

Data curation: code selection. Using the public GitHub metadata dump by [37], we collected the 150 most starred repositories with permissive licences (MIT, MIT-0, or Apache-2.0), covering twelve mainstream languages (C, C++, Java, Python, JavaScript, TypeScript, Rust, PHP, C#, Scala, Kotlin, Go); multiple languages serve as an additional test for the multi-lingual capabilities of the resulting model. From each repository, we filtered source files in the range of 3K–15K characters, discarded near-duplicates (via Jaccard similarity of simple unigram representations) and non-ASCII files, and stratified the final dataset into a *train/val/test* split with 88/12/24 files respectively, keeping all languages represented.

Data curation: query synthesis. We know of no public benchmark currently linking *questions about code* to UML diagrams; [38] notes that in the absence of human-written questions, one can use templates or NLP-based generation methods [39]. We generated user queries with two open LLMs, DeepSeek-R1-Distill-Llama-70B [40] and QwQ-32B-Preview [41] (50% each). Prompt templates detailed in Figure 2 encourage questions about architecture, API usage, or design patterns that can be answered by a single diagram. Each query is also assigned a detail level: *minimal*, *moderate*, or *full*.

````
1. The final output must be a JSON array of strings representing questions,
   enclosed within ‘<candidates>‘ and ‘<final_output>‘ XML tags.
   You must strictly follow this template for the final answer:
   ‘<candidates>["question1", ..., "questionN"]</candidates>
   <final_output>["question1", ..., "questionK"]</final_output>‘.
   During processing, you may represent questions in any format.
2. Questions must focus on aspects of the provided code, such as project structure,
   component interactions, imported modules, code behavior, design patterns,
   code design principles, external API usage, deployment strategies,
   or the potential integration of new components into the code.
3. Each question will be used as a user query for another LLM.
   Therefore, each generated question should either ask the LLM for information
   or instruct it to perform a task.
4. The expected answer for each question must be some kind of diagram
   (e.g., PlantUML or Mermaid.js).
   Ensure each question can be effectively answered with a diagrammatic
   representation rather than just text.
5. Do not mention diagrams explicitly; instead, use similar words such as structure,
   relationships, interactions and so on.
6. You must generate 10 candidate questions within the ‘<candidates>‘ tags.
7. Filter or combine candidate questions to create a final selection
   of 0 to 3 high-quality questions within the ‘<final_output>‘ tags.
   Prioritize question quality over quantity.
   Try to cover different topics in the final selection.
8. The LLM answering the questions will only have access to the content
   of the provided code file. Therefore, ensure all generated questions
   can be answered solely based on the information within that file,
   without requiring external context or knowledge beyond the given code.
9. Each question must be straightforward and as short as possible.
   Since each question is intended to be answered by a single diagram,
   avoid combining multiple questions into one.
10. Each question must be unique.
    Avoid creating duplicate or near-duplicate questions, even with slight
    variations in wording.
11. If you cannot identify any questions that meet all the specified criteria,
    it is perfectly acceptable to return an empty list
    within the ‘<final_output>‘ tags.
    If a question does not fully meet all requirements (or if you’re not
    100% sure), you should remove it.

Generate a list of questions based on the following code file:
‘‘‘
{code}
‘‘‘
````

Figure 2. Prompt template used to generate user queries (sampling, temperature 0.6, top_p 0.9).

Data curation: diagram construction. While public documentation, e.g., in the Lindholmen dataset [5] contains UML diagram images from open source repositories, these diagrams often span multiple files or

```
1. You should generate Python list of nodes, list of edges and list of packages.
   Each element is presented by JSON.
2. Make sure the final diagram is comprehensible: it should be readable,
   understandable by user.
3. For each node write a short description.
4. The diagram should contain all important nodes and edges to code understanding.
5. You can omit some of the standard, boilerplate-code steps.
6. You can add conceptual entities to diagram
   (high-level concepts that are not functions or classes).
7. Try to group related nodes (including those you introduce) using package
   elements, where possible, do not use classes for this purpose.
8. Package elements can be nested.
9. Do not add fake classes to aggregate code entities, use packages for this
   purpose. All fields and methods should be presented as is in code.
10. JSON template for node: {
    "type": Literal["class", "variable", "function", "entity", "method", "field"],
    # a type of node from the predefined types; use the most appropriate
    "name": str, # the actual name of the node to show on the diagram
    "node_id": str, # id or unique name of the node; use the actual name
    of the node if possible
    "description": str, # a short description of node
    "visibility": Literal["private", "protected", "package private", "public"], # an
    access modifier of the node from the predefined types; use the most appropriate
    "return_type": Optional[str], # a return type for functions and methods or
    current node type for fields and variables; can be skipped for other
    types of nodes
    "params": Optional[str], # parameters of functions and methods (as in brackets);
    can be skipped for other types of nodes
    "source_class_id": Optional[str], # node id of the source class for fields and
    methods; can be skipped for other types of nodes
}
11. JSON template for edge: {
    "node_id_from": str, # id of the node where the edge starts
    "node_id_to": str, # id of the node where the edge ends
    "description": Optional[str], # a short description of the edge; can be None
}
12. JSON template for packages: {
    "package_id": str, # id or unique name of the package; it will be shown on the
    diagram
    "children": List[str], # list of ids of nodes to include in the package or
    names of nested packages
    "description": Optional[str], # a short description of the package; can be None
}
```

entire projects, making them GPU-intensive and, more importantly, unaligned with specific user queries; moreover, most diagrams are available only as images, complicating conversion to structured formats.

```
13. JSON template for graph: {
    "nodes": [{node1_JSON}, ..., {nodeN_JSON}],
    "edges": [{edge1_JSON}, ..., {edgeN_JSON}],
    "packages": [{package1_JSON}, ..., {packageN_JSON}]
}
14. Prefer to use camelCase, snake_case, PascalCase for names,
    do not use spaces in names.
15. Package names can not be in edges list, use them only in packages.
16. Do not write any explanations, just write expected output.
17. You should generate three versions of the `Graph template`.
    The first one, called the "minimal version," should include
    only the essential nodes and logic, removing all unnecessary elements.
    The second one, the "medium version," should contain all important nodes
    along with some additional details.
    The third one, the "full version," should incorporate every possible node
    and all possible edges relevant to query.
18. After generating all three graph templates, you should provide the shortest
    possible "text answer" to the user's query using the generated diagrams.
19. Expected output template: {
    "minimal_version": {graph_JSON},
    "medium_version": {graph_JSON},
    "full_version": {graph_JSON},
    text_answer: str,
}

Your task is write nodes, edges and packages to build
a diagram in different formats (PlantUML, mermaid-js, ...)
for the following file in project:
```
{code}
```
You need to generate a diagram for the following user query:
\"{query}\"
```

Figure 3. Prompt template used to generate diagrams with base models (greedy).

Thus, we used a hybrid approach: first, queries were answered by *Claude 3.5 Sonnet*, which yielded the highest JSON validity in preliminary tests[1]; the prompt template is given in the Table 3. The output is a *format-agnostic JSON graph* with lists of *nodes*, *edges*, and (nested) *packages*. Six element types are allowed: *class*, *method*, *field*, *function*, *variable*, and *abstract entity*, with every element and connections between them

[1]We compared with the best models available in early 2025: GPT-4o, o1-preview, o3, QwQ-32B-Preview, DeepSeek-R1, DeepSeek-R1-Distill-Llama-70B, DeepSeek-R1-Distill-Qwen-32B, Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct, and Qwen2.5-Coder-32B-Instruct.

Component	Field Name	Type	Description
Output	nodes	List[dict]	List of nodes
	edges	List[dict]	List of edges
	packages	List[dict]	List of packages
Node	type	str	Node type
	name	str	Display name
	node_id	str	Unique ID
	description	str	Short description
	visibility	str	Access modifier
	return_type	Optional[str]	Return or data type
	params	Optional[str]	Method/function parameters
	source_class_id	Optional[str]	ID of the parent class
Edge	node_id_from	str	Source node ID
	node_id_to	str	Target node ID
	description	Optional[str]	Edge label
Package	package_id	str	Package name
	children	List[str]	Included node or sub-package IDs
	description	Optional[str]	Package description

Table 1. JSON graph format and component descriptions.

containing a brief description (see full schema in Table 1). We initially produced 264 training and 36 validation graphs.

Then, due to frequent minor and severe defects (see a list in the Table 6), all graphs were manually reviewed and corrected until all defects were fixed. Graphs with irreparable defects, e.g. with only a single node, were discarded. The resulting JSON graphs can be converted to PlantUML or Mermaid for visualization; we have experimented with generating diagrams directly in markup languages but encountered frequent syntax errors, consistent with the observations by [22]. Figure 4 shows a sample graph and a rendered diagram.

Model adaptation and generation: fine-tuning. As the backbone, we chose *Qwen-2.5-Coder-14B-Instruct* due to its strong zero-shot coding performance within a single-GPU budget (NVIDIA Tesla V100, 32GB VRAM). To fine-tune the 14B model, we used QLoRA [42] which quantizes W_0 to a 4-bit NF4 representation and full precision adapters, training with supervised cross-entropy loss on the (query, diagram) pairs. We used

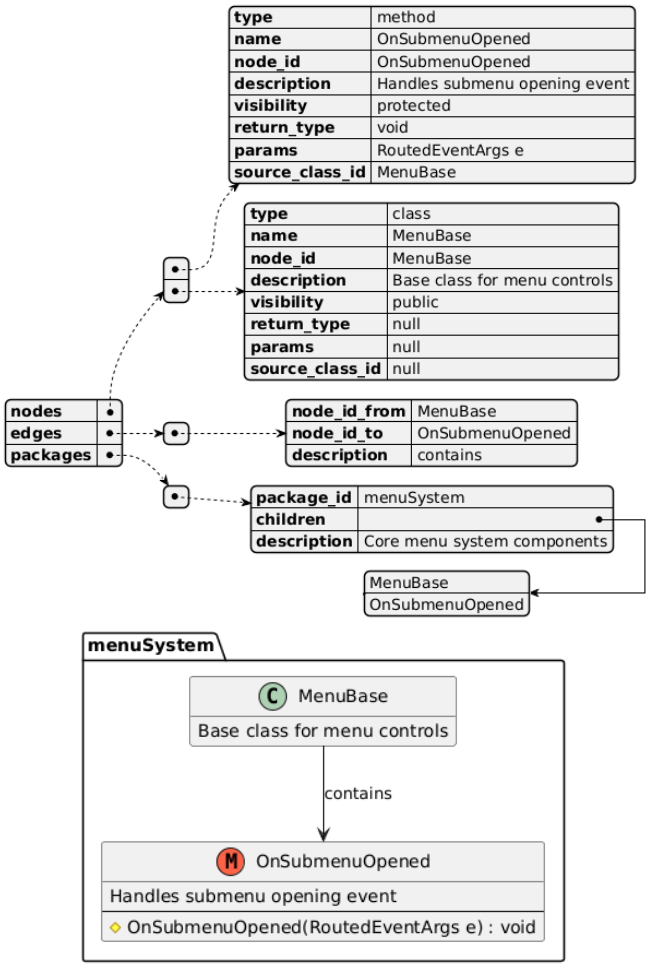

Figure 4. A sample JSON graph and visualization rendered with PlantUML's toolkit.

LLaMA-Factory [43] and incorporated Unsloth [44] optimizations including L2 regularization, dropout, cosine learning rate scheduling, gradient accumulation, checkpointing, and FP16 precision. Hyperparameters are listed in Table 2.

Hyperparameter	Value	Hyperparameter	Value
Learning rate	3e−4	Dropout	0.05
L2 regularization	1e−7	Batch size	2
LoRA rank	64	Gradient accumulation	32
LoRA α	64	Training steps	24

Table 2. Training hyperparameters.

```
Your task is to generate json with "nodes", "edges" and "packages"
for a diagram.
The diagram must answer to the user query using the code.

<code>
{code} </code>
<query>
{query} [{version} version]
```

Figure 5. Prompt template used to generate diagrams with fine-tuned models (greedy).

Metric	Description
Sufficiency	# of nodes that *must* be in the response to answer the query
Completeness	# of nodes that are not required to be present in the response, but helpful for understanding of the answer
Hallucinations	# of nodes that cannot be derived from the code, e.g., "invented" variables and classes
Verbosity	# of nodes irrelevant to the query in a diagram

Table 3. Metrics designed to evaluate whether each node is relevant to the user's information need.

Model adaptation and generation: inference. At inference, we employ *vLLM* [45] for fast token streaming and *Outlines* [46] to impose the JSON schema as a constrained grammar, guaranteeing syntactically valid graphs even for small models. The same prompt template (see Figure 5) was used for both fine-tuning and decoding.

Model	Su ↑	Co ↑	Ha ↓	Ve ↓
GPT-4o	181	51	**0**	**13**
Claude-3.5-Sonnet	234	76	1	14
Qwen2.5-Coder-14B	252	**159**	19	115
+SFT: Claude Synth	**258**	150	10	61
+SFT: Fixed Claude Synth	**258**	145	4	62

Table 4. Number of nodes by relevance classes. **Su** — Sufficiency, **Co** — Completeness, **Ha** — Hallucinations, **Ve** — Verbosity.

§4. Evaluation criteria

We assess our system from two complementary angles: *structural soundness* of the graphs and their *semantic adequacy* with respect to the query.

Automatic defect analysis. First, a Python checker scans every JSON graph for 19 defect patterns grouped into *minor* (stylistic issues, suspicious structure, missing expected connections etc.), *severe* (when components have to be removed or modified), and *unacceptable* (unrenderable) categories (for a list of defects see Appendix, Table 6). We report (i) *micro* defects per node, (ii) *macro* defects per node averaged per diagram, and (iii) the mean defects per diagram.

Human relevance annotation. Second, to assess the actual relevance of the diagram, two experts independently classify each node into Sufficiency, Completeness, Hallucination, or Verbosity categories (see Table 3). Based on confusion counts, we compute standard classification metrics: precision, recall, and F_1, plus their *hard* versions that treat only Sufficiency as true positives (see a full description in Table 5).

Metrics. It is difficult to provide a "gold set" of nodes that must be present in a diagram for a given query and code, as diagrams can be built with different approaches; e.g., some models group configuration parameters, with each group responsible for a specific routine, thereby omitting dozens of hashmap fields.

Still, measuring recall is clearly important, so we used a compromise: the number of false negatives (FN) is estimated as the difference between the maximum number of relevant nodes found across all models for a given query and the number of true positives for a specific model. Note that these

Metric	Formula
TP	$\lvert\mathrm{Su}\rvert + \lvert\mathrm{Co}\rvert$
FP	$\lvert\mathrm{Ha}\rvert + \lvert\mathrm{Ve}\rvert$
FN$\star$	$\mathbf{max}(\lvert\mathrm{Su}\rvert) + \mathbf{max}(\lvert\mathrm{Co}\rvert) - \mathrm{TP}$
$\mathrm{TP}_{\mathrm{hard}}$	$\lvert\mathrm{Su}\rvert$
$\mathrm{FN}_{\mathrm{hard}}\star$	$\mathbf{max}(\lvert\mathrm{Su}\rvert) - \mathrm{TP}_{\mathrm{hard}}$
Precision	$\mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$
Recall$\star$	$\mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$
F1$\star$	$2 * \mathrm{TP}/(2 * \mathrm{TP} + \mathrm{FP} + \mathrm{FN})$
$\mathrm{Precision}_{\mathrm{hard}}$	$\mathrm{TP}_{\mathrm{hard}}/(\mathrm{TP}_{\mathrm{hard}} + \mathrm{FP})$
$\mathrm{Recall}_{\mathrm{hard}}\star$	$\mathrm{TP}_{\mathrm{hard}}/(\mathrm{TP}_{\mathrm{hard}} + \mathrm{FN}_{\mathrm{hard}})$
$\mathrm{F1}_{\mathrm{hard}}\star$	$2 * \mathrm{TP}_{\mathrm{hard}}/(2 * \mathrm{TP}_{\mathrm{hard}} + \mathrm{FP} + \mathrm{FN}_{\mathrm{hard}})$

Table 5. Metrics. |Su|—"Sufficiency" nodes count, |Co|—"Completeness", |Ha|—"Hallucinations", |Ve|—"Verbosity".

metrics (marked with an asterisk $\star$ in Table 5) differ from standard definitions. Since FN and $\mathrm{FN}_{\mathrm{hard}}$ represent lower bounds, the corresponding Recall, F1, $\mathrm{Recall}_{\mathrm{hard}}$, and $\mathrm{F1}_{\mathrm{hard}}$ metrics are upper-bound estimates.

§5. Evaluation results

Models. We benchmark five models: *GPT-4o*, *Claude 3.5 Sonnet*, *Qwen2.5-Coder-14B*, and two fine-tuned versions of *Qwen2.5-Coder-14B* fine-tuned on diagram data (Section 3): trained on synthetic diagrams generated by *Claude 3.5 Sonnet* "as-is" (Claude Synth) and after manual corrections (Fixed Claude Synth).

Defect analysis. Table 7 reveals that fine-tuning on high-quality, manually corrected data reduces diagram defects: the *Qwen2.5-Coder-14B SFT: Fixed Claude Synth* model achieved the lowest defect counts, outperforming *GPT-4o* and *Claude 3.5 Sonnet*. Fine-tuning on raw synthetic data yielded ambiguous results. No unacceptable defects were observed in any experiment.

Relevance evaluation. Two experts annotated *48 diagrams per model* using the protocol from Section 4. The inter-annotator agreement was high (Cohen's $\kappa = 0.82 \pm 0.02$), and all residual conflicts were resolved through discussion to produce a consensus gold set. Tables 4 and 8 show a pronounced precision–recall trade-off among all base models. *GPT-4o*

Minor Defects	**Severe Defects**
Spaces in node names	Non-unique packages ids
Spaces in package ids	Non-unique nodes ids
Single node	Edges from/to non-valid node ids
No edges	More packages than nodes
Edge to itself	Packages without nodes
Repeated edges	Child in multiple packages
More than one edge between two nodes	Methods/fields have empty source class id
Edge to source class	Packages recursion
No edge from source class	Multiple connected components
Invalid node name	**Unacceptable Defects**
Invalid node id	Broken JSON
Invalid package id	Non-drawable diagram
Name of node not found in code	
Single-node package	
Class is in the package, but its method/field is not in the same package	

Table 6. Defects list.

Model	**Macro avg**		**Micro avg**		**Mean**	
	Low	**Med**	**Low**	**Med**	**Low**	**Med**
GPT-4o	0.299	0.074	0.290	0.069	1.479	**0.354**
Claude 3.5 Sonnet	0.376	0.117	0.369	0.102	2.500	0.688
Qwen2.5-Coder-14B						
No SFT	0.283	0.283	0.249	0.081	2.833	0.917
Claude Synth	0.218	0.181	0.277	0.112	2.770	1.125
Fixed Claude Synth	**0.144**	**0.043**	**0.162**	**0.038**	**1.583**	0.375

Table 7. Aggregated number of defects (lower is better). Macro-Averaged: per-node then per-diagram. Micro-Averaged: overall per-node. Mean: per-diagram

and *Claude 3.5 Sonnet* deliver the highest precision, introducing almost no "Hallucination" or "Verbosity" nodes, but miss some relevant content,

Metrics		GPT-4o	Claude-3.5-Sonnet	Qwen2.5-Coder-14B	+SFT: Synth	+SFT: Fixed Synth
Precision	micro	0.947	**0.954**	0.754	0.852	0.859
	macro	0.943	**0.944**	0.815	0.863	0.895
Precision$_h$	micro	0.933	**0.940**	0.653	0.784	0.796
	macro	**0.940**	0.938	0.786	0.828	0.867
Recall	micro	0.439	0.587	**0.778**	0.773	0.763
	macro	0.540	0.710	0.797	**0.805**	0.799
Recall$_h$	micro	0.580	0.750	0.808	**0.827**	**0.827**
	macro	0.630	0.836	0.828	**0.845**	0.842
F1	micro	0.600	0.727	0.766	**0.810**	0.808
	macro	0.648	0.779	0.765	0.815	**0.823**
F1$_h$	micro	0.539	0.668	0.668	**0.730**	**0.730**
	macro	0.602	0.742	0.719	0.763	**0.767**

Table 8. Micro- and macro-averaged metrics (**h** stands for "hard"). "Synth" stands for *Claude*-generated synthetic data.

resulting in low recall. Conversely, the untuned *Qwen2.5-Coder-14B* maximizes recall by producing many "Sufficiency" and "Completeness" nodes at the cost of the lowest precision.

The proposed fine-tuning scheme effectively closes this gap. The *Qwen2.5-Coder-14B SFT: Fixed Claude Synth* (*Qwen* tuned on manually corrected diagram data) achieves the best F_1 scores, retaining the recall of the base *Qwen* while markedly improving precision by suppressing "Hallucinations" and "Verbosity" nodes. Macro-averaged metrics are consistently higher than micro-averaged ones, indicating that all models fare better on smaller, less complex diagrams. The "hard" metrics reinforce the same picture: Precision$_h$ is lower than standard precision for all models, while

Recall_h is higher, implying that many true positives belong to the supplementary "Completeness" class and that models are most proficient at recovering indispensable "Sufficiency" nodes rather than non-essential ones.

Overall, our results demonstrate that fine-tuning a code-specialized LLM even on a very small set of high-quality, manually corrected diagrams produces diagrams that are both structurally sound and semantically faithful, making *Qwen2.5-Coder-14B SFT: Fixed Claude Synth* the strongest candidate for practical applications in query-driven code visualization.

Additional observations. We noted an intriguing strategy employed by models such as *Claude 3.5 Sonnet*: generated nodes of type "*entity*" (a UML node type) occasionally serve as abstractions of otherwise overly detailed structures (e.g., representing a database connection's parameters as several grouped entities rather than enumerating every individual parameter). This abstraction can facilitate comprehension of the response and reduce the cognitive load on the user by avoiding excessive detail. However, it complicates scoring because distinguishing legitimate abstractions from potential hallucinations remains challenging.

For qualitative evaluation, Figure 6 shows two sample generated diagrams with all element types.

§6. Conclusion

In this work, we introduce query-driven UML diagram generation, a novel approach that bridges the gap between developer information needs and automated documentation. Our experiments have shown that LLMs can successfully generate focused, comprehensible diagrams that directly answer high-level queries about code architecture, design patterns, and component interactions. By fine-tuning Qwen2.5-Coder-14B on carefully curated data, we have achieved significant improvements in both structural correctness (lowest defect rates) and semantic relevance (best F1 scores), proving that even modest amounts of high-quality training data can effectively shape LLM behavior for specialized tasks.

Our dual evaluation framework—combining automatic defect analysis with human relevance assessment—provides a robust methodology for assessing diagram quality beyond simple syntactic correctness. The results validate both research questions: LLMs can generate relevant graph structures from code (**RQ1**), and targeted training successfully controls diagram quality properties (**RQ2**). The fine-tuned models show significant

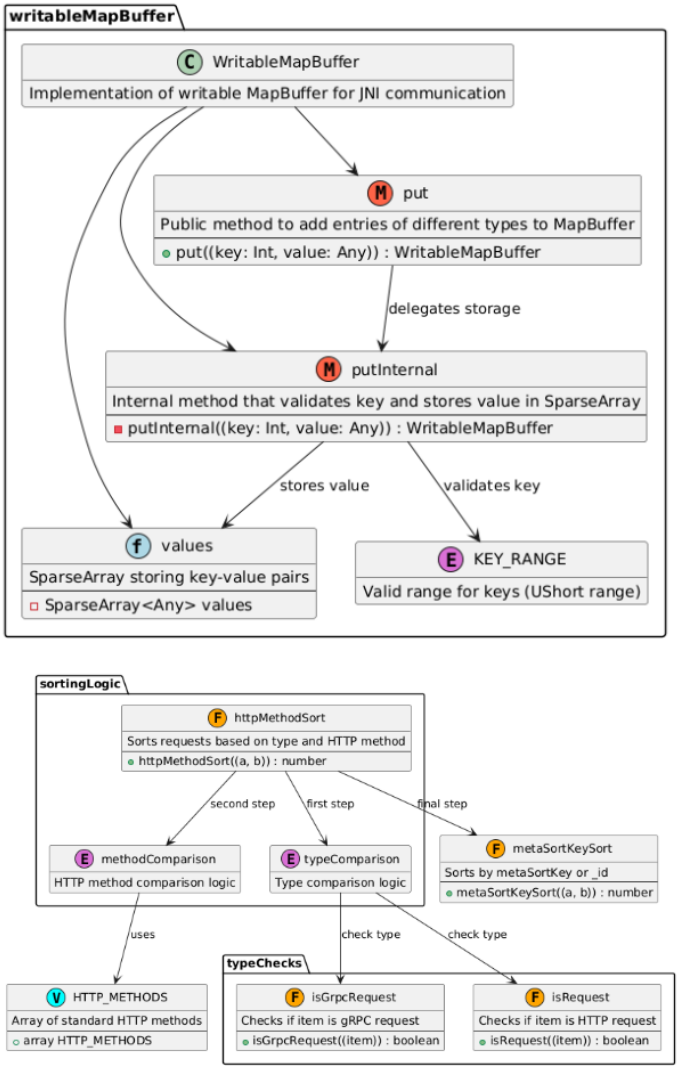

Figure 6. Sample generated diagrams.

improvements in structural correctness with dramatically reduced defect rates, while maintaining or improving semantic relevance.

This study opens several promising research avenues for the future. First, extending beyond single-file analysis to generate diagrams that capture inter-module dependencies and project-wide architectural patterns (made practical with recent advances in long-context LLMs) would significantly increase practical utility. Second, conversational interfaces where developers can iteratively refine diagrams through follow-up queries ("zoom into the error handling") would create better documentation experiences. Third, while we focused on class diagrams, the approach could extend to sequence diagrams (for execution flows), activity diagrams (for business logic), or state machines (for component lifecycles), each requiring specialized training data and evaluation metrics. Fourth, it would be great to explore hybrid approaches that combine LLM-generated diagrams with static analysis tools, looking for the best of both worlds. Finally, together with domain-specific adaptation, this could lead to the overall goal of *real-time documentation*: by integrating query-driven generation into IDEs, developers could ultimately have on-demand, always-current documentation with zero user effort. This is the vision that we would like to fulfill in the future, and we hope that in this work we have taken important steps towards it.

Limitations and future directions

While our results demonstrate the promise of query-driven diagram generation, several limitations point toward future improvements.

Training methodology: our experiments focus on supervised fine-tuning (SFT). We explored alignment techniques, specifically ORPO [47], using manually corrected diagrams as positive examples and their uncorrected counterparts as negatives, but obtained mixed results requiring further investigation. A natural next step involves leveraging our automatic defect metrics as training signal in online reinforcement learning approaches.

Evaluation scope: although the natural language descriptions attached to diagram elements appear contextually appropriate and useful for understanding, we did not conduct human evaluation of their actual utility for developers. Similarly, descriptions of the relationships (edges) between nodes often seem plausible and meaningful, but high annotation cost prevented their systematic assessment.

Metric design: our recall metrics estimate false negatives by comparing against the maximum relevant nodes found across all models for each query, providing upper bounds rather than absolute values. However, defining a single "gold standard" for diagram content remains theoretically challenging, as different developers may legitimately prefer different abstractions or levels of detail for the same query. To address this, our evaluation employs multiple safeguards (minimal/full versions, hard metrics, independent defect analysis) to ensure consistent model differentiation.

Context window: our analysis is constrained to single code files, which limits practical applicability, as real-world developer queries often span multiple files. However, our evaluation design—using large files aggregated from multiple repositories—naturally simulates challenges faced by retrieval-augmented generation (RAG) systems: irrelevant code (false positive retrievals) and incomplete external dependencies (false negative retrievals) mirror typical retrieval imperfections. Annotators were instructed to penalize both error types, allowing our results to demonstrate model robustness to imperfect inputs. Systematic evaluation with controlled FP/FN rates and integration with actual RAG pipelines remains future work.

Generalization: our experiments used a specific model family (Qwen2.5-Coder) and a relatively small training dataset, and broader validation would strengthen our claims.

We believe that these limitations, rather than diminishing our contributions, highlight the richness of this research direction. Each constraint above represents an opportunity for future work, moving towards intelligent documentation systems that adapt to developer needs.

References

1. E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, *The impact of UML documentation on software maintenance: An experimental evaluation.* — IEEE Transactions on Software Engineering **32** (2006), 365–381.
2. A. Nugroho and M. R. V. Chaudron, *Evaluating the Impact of UML Modeling on Software Quality: An Industrial Case Study*, in: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2009.
3. A. Nugroho and M. R. V. Chaudron, *The impact of UML modeling on defect density and defect resolution time in a proprietary system.* — Empirical Software Engineering **19** (2014), 926–954.

4. A. Nugroho and M. R. V. Chaudron, *A survey into the rigor of UML use and its perceived impact on quality and productivity*, in: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2008, pp. 90–99.
5. R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, *The quest for open source projects that use UML: mining GitHub*, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016, pp. 173–183.
6. H. Osman, A. van Zadelhoff, D. R. Stikkolorum, and M. R. Chaudron, *UML class diagram simplification: What is in the developer's mind?*, in: Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling, 2012, pp. 1–6.
7. H. Osman, A. van Zadelhoff, and M. R. V. Chaudron, *UML class diagram simplification — a survey for improving reverse engineered class diagram comprehension*, in: International Conference on Model-Driven Engineering and Software Development **2**, 2013, pp. 291–296.
8. A. Egyed, *Automated abstraction of class diagrams.* — ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002), 449–491.
9. Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, *A Survey on Large Language Models for Software Engineering*, ArXiv preprint arXiv:2312.15223 (2023).
10. V. Lomshakov and S. Nikolenko, *Large language models for source code generation and editing.* — Zapiski Nauchnykh Seminarov POMI **540** (2024), 276–350.
11. Copilot Team, *Microsoft Copilot: Your AI companion*, `https://copilot.microsoft.com/` (2023).
12. Cursor Team, *Cursor — The AI Code Editor*, `https://cursor.com/` (2023).
13. R. Jongeling, A. Cicchetti, and F. Ciccozzi, *How are Informal Diagrams Used in Software Engineering? An Exploratory Study of Open-Source and Industrial Practices.* — Software and Systems Modeling **24** (2025), 601–613.
14. A. Štěpánek, D. Kuťák, B. Kozlíková, and J. Byška, *Helveg: Diagrams for Software Documentation.* — IEEE Transactions on Visualization and Computer Graphics **31** (2025), 9079–9090.
15. V. Kan, M. P. Lnu, S. Berhe, C. El Kari, M. Maynard, and F. Khomh, *Automated UML Visualization of Software Ecosystems: Tracking Versions, Dependencies, and Security Updates*, in: Procedia Computer Science, 8th International Conference on Emerging Data and Industry (EDI40) **257**, 2025, pp. 834–841.
16. B. Battulga, L. Tsoodol, E. Dovdon, N. Bold, and O.-E. Namsrai, *Metric-Based Defect Prediction from Class Diagram.* — Array **27** (2025), 100438.
17. N. Khiati, D. Bouchiha, Y. Atig, and S. Boukli Hacene, *WA2MA: A Model-Driven Approach for Reengineering Web Applications into Mobile Applications.* — Edelweiss Applied Science and Technology **9** (2025), 1530–1544.
18. P. Tonella and A. Potrich, *Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers*, in: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, 2001, pp. 376–385.

19. A. Sutton and J. I. Maletic, *Recovering UML class models from C++: A detailed explanation.* — Information and Software Technology **49** (2007), 212–229.
20. Y.-G. Guéhéneuc, R. Douence, and N. Jussien, *No Java without Caffeine: A tool for dynamic analysis of Java programs*, in: Proceedings 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 117–126.
21. Y.-G. Guéhéneuc, *A reverse engineering tool for precise class diagrams*, in: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research, 2004, pp. 28–41.
22. M. Shehata, B. Lepore, H. Cummings, and E. Parra, *Creating UML Class Diagrams with General-Purpose LLMs*, in: 2024 IEEE Working Conference on Software Visualization (VISSOFT), 2024, pp. 157–158.
23. J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, *GPT-4 Technical Report*, ArXiv preprint arXiv:2303.08774 (2023).
24. H. A. Siala and K. Lano, *Using Large Language Models to Extract UML Class Diagrams from Java Programs*, in: 8th International Conference on Software and System Engineering (ICoSSE 2025), 2025, pp. 70–74.
25. H. A. Siala and K. Lano, *Leveraging LLMs for Abstracting UML and OCL Representations from Java and Python Programs*, 2025.
26. H. A. Siala and K. Lano, *Towards Using LLMs in the Reverse Engineering of Software Systems to Object Constraint Language*, in: Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025, pp. 885–890.
27. A. Boronat and J. Mustafa, *MDRE-LLM: A Tool for Analyzing and Applying LLMs in Software Reverse Engineering*, in: Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025, pp. 850–854.
28. D. Amalfitano, M. De Luca, T. Santilli, and P. Pelliccione, *Automated Software Architecture Design Recovery from Source Code Using LLMs*, in: Software Architecture **15929**, Springer, Cham, 2025, pp. 73–89.
29. J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, *On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML.* — Software and Systems Modeling **22** (2023), 781–793.
30. D. De Bari, G. Garaccione, R. Coppola, M. Torchiano, and L. Ardito, *Evaluating Large Language Models in Exercises of UML Class Diagram Modeling*, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2024, pp. 393–399.
31. A. Ferrari, S. Abualhaijal, and C. Arora, *Model generation with LLMs: From requirements to UML sequence diagrams*, in: 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW), 2024, pp. 291–300.

32. M. Jahan, M. M. Hassan, R. Golpayegani, G. Ranjbaran, C. Roy, B. Roy, and K. Schneider, *Automated Derivation of UML Sequence Diagrams from User Stories: Unleashing the Power of Generative AI vs. a Rule-Based Approach*, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 138–148.
33. Z. Babaalla, A. Jakimi, and M. Oualla, *LLM-Driven MDA Pipeline for Generating UML Class Diagrams and Code.* — IEEE Access **13** (2025), 171266–171283.
34. M. Ben Chaaben, *Software Modeling Assistance with Large Language Models*, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 188–191.
35. B. Unhelkar, *Verification and Validation for Quality of UML 2.0 Models*, John Wiley & Sons, 2005.
36. M. Genero, M. Piattini, and C. Calero, *A survey of metrics for UML class diagrams.* — Journal of Object Technology **4** (2005), 59–92.
37. P. Elmers, *Code and Comment Consistency Classification with Large Language Models*, Master's thesis, Eindhoven University of Technology, 2023.
38. E. Hoque, P. Kavehzadeh, and A. Masry, *Chart question answering: State of the art and future directions.* — Computer Graphics Forum **41** (2022), 555–572.
39. T. A. Ghaleb, M. A. Alturki, and K. Aljasser, *Program comprehension through reverse-engineered sequence diagrams: A systematic review.* — Journal of Software: Evolution and Process **30** (2018), e1965.
40. DeepSeek-AI Team, *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, ArXiv preprint arXiv:2501.12948 (2025).
41. Qwen Team, *QwQ: Reflect Deeply on the Boundaries of the Unknown*, `https://qwenlm.github.io/blog/qwq-32b-preview/` (2024).
42. T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, *QLoRA: Efficient Finetuning of Quantized LLMs.* — Advances in Neural Information Processing Systems **36** (2023), 10088–10115.
43. Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma, *LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models*, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), 2024.
44. D. Han, M. Han, and Unsloth Team, *Unsloth*, `http://github.com/unslothai/unsloth` (2023).
45. W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, *Efficient memory management for large language model serving with PagedAttention*, in: Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 611–626.

46. B. T. Willard and R. Louf, *Efficient Guided Generation for Large Language Models*, ArXiv preprint arXiv:2307.09702 (2023).
47. J. Hong, N. Lee, and J. Thorne, *ORPO: Monolithic Preference Optimization without Reference Model*, in: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, 2024, pp. 11170–11189.

HSE University
194100, 3k1 Kantemirovskaya Str.,
St. Petersburg, Russia
E-mail: baryshnikovod@gmail.com

Поступило 12 января 2026 г.

St. Petersburg Department
of Steklov Mathematics Institute,
Fontanka 27 , St.Petersburg 191023, Russia;
Markov Laboratory
for Artificial Intelligence,

St. Petersburg University,
Universitetskaya Emb. 7-9, St Petersburg 199034, Russia
E-mail: anton.m.alexeyev@gmail.com

St. Petersburg Department
of Steklov Mathematics Institute,
Fontanka 27 , St.Petersburg 191023, Russia;
ISP RAS Research Center
for Trusted Artificial Intelligence,
Moscow, Russia
E-mail: sergey@logic.pdmi.ras.ru