AI-Facilitated Software Project Generation from Natural Language Using Curated Code Snippets*

Artem Khovrat[1,*,†], Serhii Shchoholiev[2,†], Mariya Shirokopetleva[1,†], Volodymyr Kobziev[1,†], and Volodymyr Strukov[3,†]

[1] *Kharkiv National University of Radio Electronics, 14, Nauky, Ave., Kharkiv, 61166, Ukraine*
[2] *Kay.ai, 240 Kent Avenue, Brooklyn, New York, 11249, United States*
[3] *V.N. Karazin Kharkiv National University, 4, Svobody, Sq., Kharkiv, 61022, Ukraine*

Abstract

This paper addresses the challenge of generating software projects efficiently from natural language descriptions. Instead of relying on unconstrained code generation, which often produces inconsistent or unreliable results, this research explores an approach based on reusing curated, pre-stored code snippets. The study focuses on the critical step of mapping project descriptions to relevant code assets, evaluating how effectively an AI model can predict snippet relevance through systematic experimentation. The methodology combines deterministic filtering with semantic reasoning, utilizing a NoSQL database for snippet storage and large language models for relevance prediction. Multiple models including GPT-4o-mini, O3-mini, and GPT-4o are evaluated on a benchmark of 100 synthetic project descriptions against 100 curated code snippets. The research investigates three prompt engineering strategies: zero-shot, few-shot, and chain-of-thought approaches. Results demonstrate that natural language input can be reliably aligned with reusable code components, with chain-of-thought prompting achieving 43.1% accuracy compared to 30.3% for zero-shot approaches. GPT-4o-mini emerges as the optimal model, balancing performance with cost-effectiveness at approximately 7.33× lower cost than premium alternatives. The findings support the feasibility of snippet-augmented project generation as a pathway toward faster and more consistent software development. This study highlights the potential of combining AI-powered interpretation with structured code reuse, offering an alternative to purely generative approaches that maintains quality while accelerating development cycles. The approach provides a foundation for enterprise-scale deployment and integration into existing coding environments.

Keywords

code generation, artificial intelligence, code snippet, natural language processing[1]

1. Introduction

In enterprise software development, the reuse of existing code is a well-established practice aimed at enhancing productivity and maintaining consistency across projects. However, despite the availability of extensive codebases, developers often resort to "vibe coding" — a rapid, heuristic-driven approach to coding that prioritizes speed over reliability. This method frequently leads to the introduction of defects and technical debt, undermining long-term maintainability. A study by Tornhill and Borg [1] highlights the significant impact of code quality on development efficiency, revealing that low-quality code contains 15 times more defects than high-quality code, and resolving issues in such code takes, on average, 124% more time. This underscores the necessity for a more structured approach to code reuse that balances speed with reliability.

The challenge lies in effectively identifying and integrating relevant, high-quality code snippets from vast repositories. Manual selection is time-consuming and error-prone, while existing

* *ProfIT AI'25: 5th International Workshop of IT-professionals on Artificial Intelligence, October 15–17, 2025, Liverpool, UK*
[1,*] Corresponding author.
[†] These authors contributed equally.
artem.khovrat@nure.ua (A. Khovrat); serhii.shchoholiev@gmail.com (S. Shchoholiev); marija.shirokopetleva@nure.ua (M. Shirokopetleva); volodymyr.kobziev@nure.ua (V. Kobziev); strukov.volodymyr@karazin.ua (V. Strukov)
0000-0002-1753-8929 (A. Khovrat); 0009-0007-2014-4828 (S. Shchoholiev); 0000-0002-7472-6045 (M. Shirokopetleva); 0000-0002-8303-1595 0 (V. Kobziev); 0000-0003-4722-3159 0 (V. Strukov)

automated systems often lack the sophistication to understand the context of a developer's natural language input, leading to irrelevant or suboptimal code suggestions. This paper introduces an AI-powered system designed to enhance software project generation by predicting the relevance of pre-stored code snippets to a given natural language project description [2]. By focusing on the critical task of snippet relevance prediction, this approach aims to streamline the development process, enabling developers to leverage existing code assets efficiently.

The key contributions of this research are the development of a relevance prediction algorithm that accurately predicts the relevance of code snippets based on natural language project descriptions, the evaluation of the algorithm's performance through comprehensive experiments demonstrating its capability to enhance the software development process, and providing a foundation for integrating curated code databases and relevance prediction into advanced coding agents, such as Cursor, to further automate and improve the software development lifecycle.

2. Related works

The practice of code reuse has long been recognized as a key strategy to improve software development productivity and maintainability. Studies have shown that systematic reuse of code components, templates, and libraries reduces defects and development time, particularly in enterprise environments with complex, proprietary codebases. Research by Borg et al. [3] investigates the returns of highly maintainable code, revealing that maintaining high code quality can lead to significant reductions in maintenance costs and defect risks. Their study emphasizes the importance of proactive code quality management in sustaining long-term software health.

Recent advances in artificial intelligence have enabled the development of AI-powered coding agents capable of generating code from natural language descriptions. Systems such as GitHub Copilot and Cursor leverage large language models to assist developers by producing code snippets or scaffolds. While these agents can accelerate coding, their outputs often lack precision and consistency, particularly when dealing with enterprise-specific or legacy code. The effectiveness of such systems is therefore closely tied to their ability to retrieve relevant code snippets and adapt them appropriately to the context of a given project [4].

Relevance prediction and code retrieval have emerged as critical components in improving the utility of AI-assisted coding systems. Approaches in this domain typically involve embedding code snippets and project descriptions into a shared semantic space to measure similarity, allowing models to recommend the most contextually relevant components. Prior work has explored techniques such as neural code search, retrieval-augmented generation, and embedding-based similarity metrics to identify applicable code assets efficiently. These methods demonstrate that structured retrieval and prediction mechanisms can significantly enhance code reuse while maintaining high-quality output [5].

This work builds on these foundations by focusing specifically on predicting the relevance of pre-stored code snippets given a natural language project description. Unlike prior research that emphasizes full project generation or LLM-only outputs, this approach isolates the retrieval and relevance prediction step, providing a scalable foundation for future integration into coding agents and enterprise software pipelines.

3. Methodology

The methodology outlines the overall design of the system and the experimental approach taken in this research. The aim is to establish a repeatable framework for connecting natural language project descriptions to relevant code snippets stored in curated repositories. To achieve this, principles from information retrieval, machine learning, and prompt engineering are combined into a unified pipeline.

The section is structured into three main components. First, the process of retrieving candidate snippets from a company-specific code database is described. Second, the rationale for selecting the

large language model used to evaluate snippet relevance is presented, supported by experimental comparisons. Finally, the role of prompt engineering in shaping the interaction between project descriptions and the model is discussed, including the evaluation of different prompting strategies. Together, these elements form the methodological foundation for assessing how effectively AI can assist developers in leveraging existing code snippets for project generation. Implementation is available at https://github.com/Shchoholiev/assets-manager-api.

3.1. Process of getting relevant code snippets

Before the process of identifying relevant code snippets can begin, snippets must be ingested. During ingestion, the complete source code of each snippet is provided to a large language model to generate a rich, task-oriented description that captures purpose, inputs/outputs, dependencies, preconditions, side effects, security/compliance notes, and typical usage. This description is persisted as metadata and later serves as the primary semantic signal during selection. Ingestion also performs schema validation and deduplication to keep the corpus clean.

All code snippets are stored in a NoSQL database (CosmosDB) which serves as a centralized repository of curated components. CosmosDB is chosen because its document model lets us persist the full snippet source code alongside rich relationship data within a single logical record, with automatic indexing for low-latency queries. Each snippet is annotated with metadata such as name, programming language, description, and company identifier [6]. This metadata makes it possible to enforce strict technical and organizational boundaries before introducing semantic reasoning into the selection process.

The process of identifying relevant code snippets starts with a natural language project description provided by the developer. The input includes three components: the programming language, the company identifier, and the textual project description. These parameters act as constraints that guide the retrieval pipeline, ensuring that only contextually appropriate snippets are considered for reuse.

As shown in Fig. 1, the system first applies deterministic filtering. Snippets are restricted by programming language to match the intended technology stack and by company identifier to ensure that only organization-specific, internally approved code is included. This filtering step prevents irrelevant or incompatible code from entering the workflow and reduces the number of candidate snippets that need to be evaluated downstream.

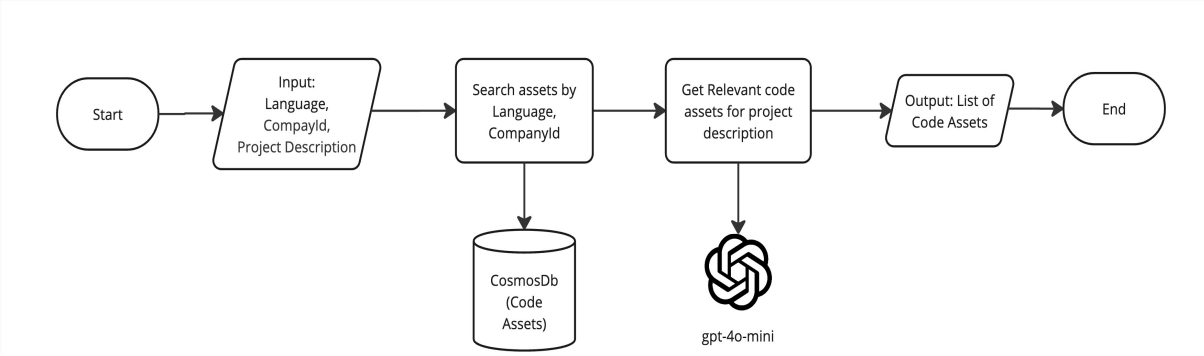

Figure 1: Flowchart for retrieving relevant code snippets [created by the authors].

Once the candidate snippets are identified, they are passed to the large language model (LLM). At this stage, the LLM evaluates the semantic relationship between the project description and the available snippets; however, to keep inference practical, only each snippet's project-level metadata (its identifier, name, and short description) is sent to the model. During ingestion, the full source is distilled into a robust, task-oriented description that captures purpose, inputs/outputs, dependencies, and security/compliance notes; this serves as a compact semantic proxy for the code that the model can reliably consume. Full source files are intentionally excluded: including raw code for a large candidate set would quickly exceed typical context windows and make prompts

unwieldy, while per-snippet ranking via separate LLM calls would drive latency and cost to impractical levels. Although reading full code might improve relevance during research experiments, it is not viable for a production system operating at enterprise scale. Instead, the model selects the snippets it deems most relevant based on the metadata and returns a structured list with brief textual justifications, providing transparency about why particular assets were chosen.

For example, given the project description “a secure, modern login service for customer accounts,” the system first narrows candidates by language and company_id. The LLM then favors an authentication snippet whose metadata shows support for modern login flows (e.g., OIDC), MFA, token-based sessions, and the company’s encryption and logging standards. In its rationale, it notes that these features directly address authentication and data security for the stated use case, while excluding look-alike snippets that lack required compliance or use a different tech stack.

The final output consists of the most relevant snippets accompanied by concise model rationales. This hybrid workflow—combining deterministic filtering with semantic reasoning—yields selections that are both technically sound and contextually appropriate, increasing transparency and developer trust. After retrieval, the proposed snippets are confirmed with the developer, then a starter project is generated and compiled to validate correctness.

3.2. Model selection

Several LLMs were evaluated for the task of selecting relevant code snippets given a natural-language project description. The goal of this study was not end-to-end project generation, but to determine which model most reliably identifies the correct subset of pre-stored, curated snippets. To ensure fair comparison, all models received the same project description, the same filtered pool of candidate snippets (after language/company constraints), and the same prompt format. Each model returned the snippets it deemed relevant along with a short justification. A benchmark of 100 synthetic project descriptions was constructed to mirror concise enterprise requirements (e.g., authentication, logging/auditing, API scaffolding, messaging, batch processing). Descriptions vary in wording and specificity to test whether models map intent — not just keywords — to appropriate building blocks. The candidate pool contains 100 curated, company-scoped code snippets from the CosmosDB repository, each tagged with language and companyId and accompanied by a short functional description (e.g., "JWT auth middleware," "transactional outbox publisher," "service template with health checks," "centralized logging adapter," "base CI pipeline").

Ground-truth sets were defined manually by selecting the minimal snippet set that would plausibly satisfy each description in a starter-project context. During scoring, only snippet IDs present in the provided candidate list were considered valid; references to out-of-scope or nonexistent snippets were treated as errors.

Commonly available models spanning a wide cost/quality range were tested: gpt-4.1-nano, gpt-4o-mini, o3-mini, gpt-4o, o1, and gpt-4.5. To account for deployment constraints in enterprise settings, both selection quality (distribution across the five outcome categories) and cost efficiency (published price per 1M in + 1M out tokens) were considered, as well as qualitative factors such as stability across prompts.

The stacked bar chart in Fig. 2 summarizes outcome distributions per model. Higher bars in “Exact match” and lower bars in “Mismatch” indicate better performance. Mid-tier models demonstrated strong accuracy without incurring the steep costs of frontier models, while the smallest model showed more frequent “Partial—Mixed” and “Mismatch” outcomes. In practice, “Partial—Extra” is often acceptable, whereas “Partial—Missing” and “Mismatch” impose higher developer overhead.

Balancing selection quality with cost and latency, gpt-4o-mini was adopted as the default model. On the 100-case benchmark it delivered competitive Exact rates with acceptable Partial—Extra at a fraction of the cost of larger models, satisfying enterprise constraints. It also responds well to prompt design, yielding further gains under guided reasoning prompts. By contrast, auto-

reasoning models such as o3-mini perform better with broad, high-level prompts but lose accuracy when given more detailed, constrained instructions. Accordingly, gpt-4o-mini is used as the backbone for the remainder of the study with its prompt engineering explored in the next section, with o3-mini as a comparative baseline.

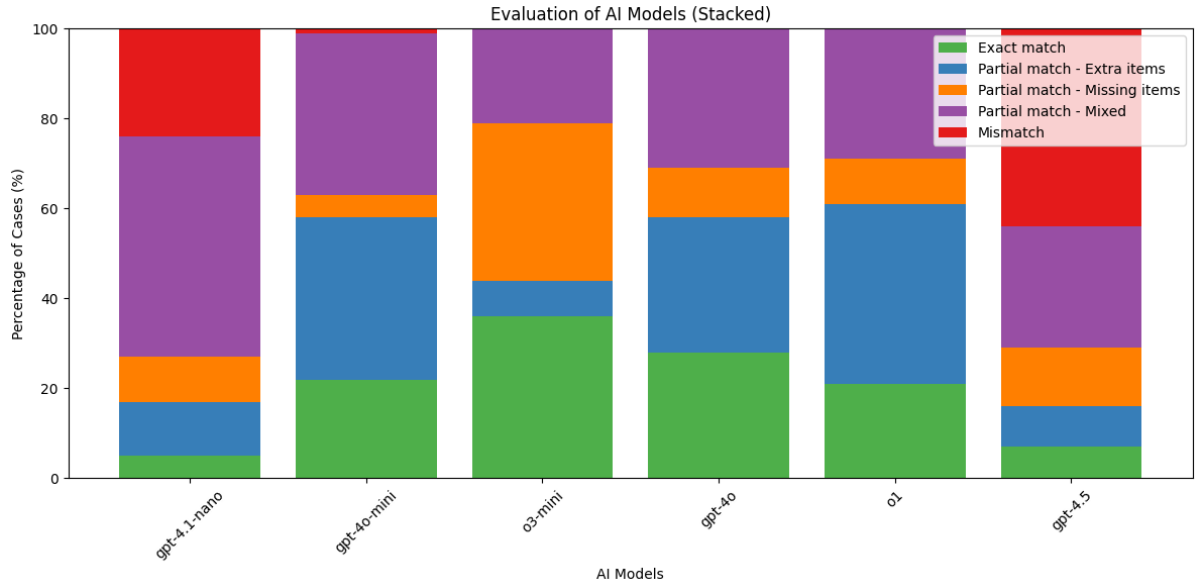

Figure 2: Model evaluation results [created by the authors].

3.3. Prompt engineering

Three prompting styles are compared for the snippet-selection task under identical conditions (same description, same candidate list filtered by language and company ID). Each prompt enforces an output schema with valid snippet IDs only and requests a brief rationale. Datasets, prompts, ground truth, and evaluation scripts are available at https://github.com/Shchoholiev/assets-manager-start-projects-evaluation.

3.3.1. Zero-Shot

A single instruction without exemplars that specifies the task and output schema. It is the lowest-cost, lowest-latency configuration and serves as the baseline. Zero-shot performs well when snippet names/descriptions are clear and the schema is explicit, but it is sensitive to phrasing and more prone to over- or under-selection if constraints are not enforced strictly (Fig. 3 for example)

```
Project Description:
I am developing a mortgage calculation microservice.
It should accurately calculate mortgage details and generate statements.

Available Code Snippets:
[
    {
        "id": "c5212788-0454-4269-a1da-23440509389c",
        "name": "ATM Integration Service",
        "description": "Connects ATM networks with banking system."
    },
    ...
]

Return a list of relevant code snippets as a JSON array.
```

[7].

Figure 3: Zero-shot prompt example [created by the authors].

3.3.2. Few-Shot

The instruction is preceded by one compact worked example that demonstrates the mapping from a description to a set of snippet IDs. The exemplar improves schema adherence and reduces spurious selections by giving the model a concrete pattern to imitate while keeping token overhead modest. Care is taken to keep the exemplar short, stylistically consistent with the evaluation items, and different from the current query to avoid leakage or superficial cue matching (Fig. 4 for example) [8].

```
user = """
Project Description:
I am developing a mortgage calculation microservice.
It should accurately calculate mortgage details and generate statements.

Available Code Snippets:
{assets}

Return a list of relevant code snippets as a JSON array.
"""

assistant = """
{
    "ids": [
        "baa059d9-6c33-48e7-9a29-a101c210d165",
        "b73c177f-c73d-4d90-bcae-95e9a6f2c0d2"
    ]
}
"""

user = """
Project Description:
I'm enhancing our ATM network by integrating a microservice
that synchronizes ATM transactions in real time.

Available Code Assets:
{assets}

Return a list of relevant code assets as a JSON array.
"""
```

Figure 4: Few-shot prompt example [created by the authors].

3.3.3. Chain-of-thought (CoT)

The instruction asks the model to articulate a brief reasoning step before emitting the final JSON answer. This encourages the model to align functions mentioned in the description with capabilities in the candidate list (e.g., security, auditing, messaging) and helps disambiguate near-miss snippets. CoT typically increases exact selections and reduces mixed/mismatch outcomes at the cost of additional tokens; reasoning length is capped and the final answer is still required in a fenced, machine-parseable schema to preserve determinism (Fig. 5 for instance) [9].

```
user = """
Project Description:
I am developing a mortgage calculation microservice. It should accurately
calculate mortgage details and generate statements.

Available Code Assets:
{assets}

Return a list of all relevant code assets as a JSON object with key "ids"
and array of string inside <json></json> tag based on the project
description.
Explain your reasoning step by step before returning the final answer.
"""

assistant = """ Let's analyze the project requirements step by step:
1. **Understanding the Goal**: The project is focused on **mortgage
calculations** and generating **statements**.
2. **Identifying Key Functions**:
- The microservice needs to **calculate mortgage details** accurately.
- It should also be able to **generate mortgage statements**.
3. **Matching Available Code Assets**:
- **"Mortgage Calculation Service"** (ID:
`"baa059d9-6c33-48e7-9a29-a101c210d165"`)
    - This service is specifically designed to **calculate mortgage details
    based on user inputs**, making it essential for the project.
- **"Electronic Statement Generator"** (ID:
`"b73c177f-c73d-4d90-bcae-95e9a6f2c0d2"`)
    - Since the microservice also needs to **generate mortgage statements**,
    this service is directly relevant.
**Final Decision**: Based on this reasoning, the two most relevant assets
are:
<json>
    {ids}
</json>
"""
```

Figure 5: Chain-of-thought (CoT) prompt example [created by the authors].

3.4. Prompt engineering — evaluation and results

The three prompting styles were assessed on the same 100-case benchmark described in Section III-B, utilizing the same model (gpt-4o-mini), identical decoding settings, and the same candidate list per case (filtered by language and companyId). Outputs were scored using the categories from Section III-C: Exact, Partial—Extra, Partial—Missing, Partial—Mixed, and Mismatch. The distribution of outcomes per style is reported as percentages over the 100 cases.

Exact matches increased with prompt guidance: 16% (zero-shot) → 22% (few-shot) → 34% (CoT). Zero-shot produced the highest rate of Partial—Extra (tending to include superfluous snippets), while few-shot reduced this by anchoring the format and selection behavior to the exemplar. CoT further improved precision and lowered Mixed/Mismatch cases by encouraging brief reasoning against the candidate list; it showed a modest rise in Partial—Missing (the model occasionally chose a minimal, defensible set) that is considered acceptable in practice. Fig. 6 summarizes these distributions.

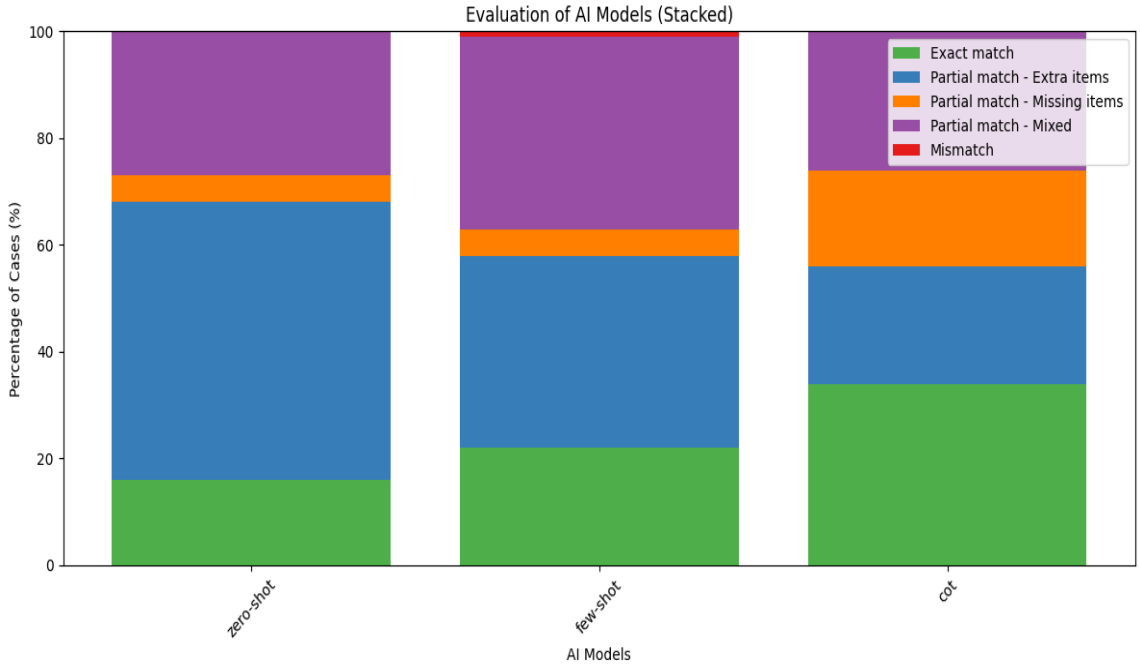

Figure 6: Prompting technique comparison [created by the authors].

4. Results

A single weighted accuracy is reported that emphasizes exact matches and normalizes to a 0–100 scale:

$$Accuracy(\%) = \frac{15E + 3X + 1M + 2D}{15 * 100} \quad (1)$$

where E is the share (%) of Exact matches, X is Partial–Extra, M is Partial–Missing, and D is Partial–Mixed; W (Mismatch) has weight 0 and is omitted.

The coefficients encode "developer effort": Exact gets the dominant weight (15) because it requires no rework; Extra earns partial credit (3) since the solution is functionally complete with minor cleanup; Mixed (2) is valued above Missing (1) because it typically contains more of the needed functionality; and Mismatch contributes nothing. The normalization by 15 makes A=100 when E=100%, keeping the metric interpretable and comparable across experiments.

Table 1
Model Evaluation Results

Model	Price per 1M input + 1M output tokens (USD)	Accuracy (%)
gpt-4.1-nano	0.50	14.6
gpt-4o-mini	0.75	34.3
o3-mini	5.50	42.7
gpt-4o	12.50	38.9
o1	75.00	33.5
gpt-4.5	225.00	13.3

On a 100-case benchmark against 100 curated, company-scoped snippets, mid-tier models offered the best practicality–accuracy balance. With the metric above, gpt-4o-mini reached 34.3%, which is ≈80% of o3-mini (42.7%) and below gpt-4o (38.9%); the smallest model lagged (gpt-4.1-nano: 14.6%), and the most expensive frontier model underperformed (gpt-4.5: 13.3%). Despite scoring below o3-mini, gpt-4o-mini's cost and latency profile makes it the preferable default backbone for frequent, large-scale runs for enterprise use case.

Prompting materially shifts outcomes for gpt-4o-mini: 30.3% (zero-shot) → 34.3% (few-shot) → 43.1% (CoT) on the same cases. Few-shot raises Exact from 16%⟶22% and cuts Extra from 52%⟶36%, which means fewer superfluous snippets to clean up, though Partial–Mixed rises (27%⟶36%) as the model hews more tightly to the exemplar. CoT then delivers the biggest jump by pushing Exact to 34% and lowering Partial–Mixed to 26%, while Partial–Extra drops to 22%; Partial– Missing increases (5%⟶18%), but the metric's heavy weight on Exact dominates, yielding the best overall score. In practice, CoT's short, constrained rationales help the model rule out look-alike snippets (e.g., logging vs auditing adapters) and align selections to compliance cues in the metadata, reducing triage despite the uptick in minimal sets. Mismatch stays ~0–1% across prompts, indicating stable schema adherence.

On cost, $0.75 (gpt-4o-mini) vs $5.50 (o3-mini) is ~7.33× cheaper per token. Normalized by the new accuracy metric, gpt-4o-mini delivers ~5.9× more accuracy per dollar than o3-mini (≈45.7 vs 7.8 percentage-points per $1 per 1M tokens). This comfortably funds CoT prompting by default while staying within enterprise constraints. Moreover, o3-mini (an auto-reasoning model) shows low upside from prompt engineering in this setting because it favors broad, high-level prompts; tighter, schema-constrained instructions do not yield proportional gains. Hence gpt-4o-mini remains the best balance of accuracy, predictability, and cost for snippet selection at scale.

5. Limitations and future directions

The benchmark relies on synthetic, enterprise-style descriptions and a single-organization snippet corpus. While this design controls variability and protects proprietary code, it limits external validity. Real specifications are longer, noisier, and interleave functional and non-functional requirements; future studies should replicate these experiments on multi-org, real-world backlogs to assess generalization.

To better approximate developer effort and risk, weighted metric should be replaced — or at least calibrated — using an LLM-as-judge protocol rather than fixed coefficients. Concretely, a judge model would receive the project description, the selected snippet set, and concise metadata (and, when feasible, quick static checks or minimal tests), then score the outcome on a rubric that

distinguishes critical vs. benign deviations (e.g., missing an authentication dependency vs. including a harmless utility). The rubric would be anchored with labeled exemplars, using pairwise comparisons for robustness, and scores would be calibrated via scale-anchoring and isotonic regression. To ensure reliability, agreement against human ratings would be measured. This judge-based metric is task-aware, explainable, and better aligned with practitioner costs than plain, static weights.

To strengthen external validity, future evaluations should include models from other vendors, including closed source and open source models, acknowledging that they are trained on different corpora, supervision mixes, architectural choices, alignment procedures, tokenizers, and context limits — all of which can materially affect retrieval and selection behavior. Comparisons should use a standardized protocol (same prompts, decoding settings, candidate pools, and scoring), report both aggregate accuracy and error profiles, and stratify by domain and prompt style.

Beyond the methodological constraints discussed above, an important practical limitation lies in maintaining and scaling snippet databases in industrial settings. As repositories grow, ensuring snippet freshness, dependency compatibility, and security compliance becomes nontrivial. In production environments, versioning, deduplication, and quality auditing must be automated through integration with existing CI/CD and Git workflows. Enterprise deployment further requires strict access controls, metadata refresh pipelines, and continuous retraining of embeddings to reflect code evolution. From a scalability standpoint, large-scale snippet retrieval may require distributed vector databases or hybrid search architectures to sustain low-latency, high-throughput selection under enterprise workloads. Addressing these challenges will be key to operationalizing the proposed system in real-world development ecosystems.

Additionally, while the current benchmark uses synthetic, well-controlled project descriptions to isolate model behavior, future research should incorporate diverse, real-world project briefs from open-source and industrial backlogs. Such inputs would introduce realistic noise, ambiguity, and interleaved functional/non-functional requirements, offering a more rigorous test of snippet relevance prediction under production conditions.

6. Conclusion

This study demonstrates the potential of integrating AI-powered generative models to automate software project generation from natural-language inputs. By grounding assembly in curated code snippets, the approach cuts setup time, standardizes scaffolds, and lets developers focus on higher-value design work. Importantly, this should not replace unconstrained code synthesis but extend it: use free-form generation for genuinely novel logic and "glue" while anchoring core functionality in vetted components. This hybrid reduces the variability of "vibe coding," improving reliability, security, and compliance without sacrificing speed—turning AI from a code copier into a quality-aware accelerator of real-world development.

From a deployment standpoint, the selection algorithm is a strong candidate for packaging as a Model Context Protocol (MCP) service, exposing endpoints for deterministic filtering, snippet metadata retrieval, relevance selection, and starter assembly. Such an MCP tool can be plugged into Cursor, Windsurf, or other code-generation environments so developers can invoke snippet-augmented generation directly from the editor, receive concise rationales, and enforce organizational policies—making the hybrid retrieval plus generation workflow immediately actionable in day-to-day practice.

Acknowledgments

The authors would like to thank the Armed Forces of Ukraine for the opportunity to write a valid work during the full-scale invasion of the Russian Federation on the territory of Ukraine. Also, the authors wish to extend their gratitude to Kharkiv National University of Radio Electronics for providing licences for additional software to prepare algorithms and the paper.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly Edu in order to check grammar and spelling. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content

References

[1] A. Tornhill, and M. Borg, "Code Red: The Business Impact of Code Quality - A Quantitative Study of 39 Proprietary Production Codebases" 2022, arXiv:2203.04374. URL: https://arxiv.org/abs/2203.04374

[2] A. Khovrat, V. Kobziev, V. Volokhovskyi and O. Nazarov, "Using Classifiers Based on Large Language Models and Naïve Bayes for Domain Specific Text," 2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT), Lviv, Ukraine, 2024, pp. 1-4, doi: 10.1109/CSIT65290.2024.10982586.

[3] M. Borg, I. Pruvost, E. Mones and A. Tornhill, "Increasing, not Diminishing: Investigating the Returns of Highly Maintainable Code" 2024, arXiv:2401.13407. URL: https://arxiv.org/abs/2401.13407

[4] Yo. Ishibashi, Yo. Nishimura, "Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization" 2024, arXiv:2404.02183. URL: https://arxiv.org/abs/2404.02183

[5] Z. Yang, S. Chen, C. Gao, Zh. Li, X. Hu, K. Liu and X. Xia, "An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities" 2025, arXiv:2501.13742 URL: https://arxiv.org/abs/2501.13742v1

[6] Y. Liu, G. Deng, Yue Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, "Prompt Injection attack against LLM-integrated Applications", 2024, arXiv:2306.05499 URL: https://arxiv.org/abs/2306.05499

[7] Y. Li, "A Practical Survey on Zero-shot Prompt Design for In-context Learning", 2023, arXiv:2309.13205 URL: https://arxiv.org/abs/2309.13205

[8] H. Dang, L. Mecke, F. Lehmann, S. Goller, D. Buschek, "How to Prompt? Opportunities and Challenges of Zero- and Few-Shot Learning for Human-AI Interaction in Creative Applications of Generative Models", 2022, arXiv:2209.01390 URL: https://arxiv.org/abs/2209.01390

[9] Y. Yao, Z. Liu, H. Zhao, "Beyond Chain-of-Thought, Effective Graph-of-Thought Reasoning in Language Models", 2023, arXiv:2305.16582 URL: https://arxiv.org/abs/2305.16582