LIBEVOLUTIONEVAL: A Benchmark and Study for Version-Specific Code Generation

Sachit Kuhar[1] **Wasi Uddin Ahmad**[2*] **Zijian Wang**[1] **Nihal Jain**[1] **Haifeng Qian**[2*]
Baishakhi Ray[1] **Murali Krishna Ramanathan**[1] **Xiaofei Ma**[1] **Anoop Deoras**[1]
[1]AWS AI Labs [2]NVIDIA

Abstract

Recent advancements in code completion models have primarily focused on local file contexts (Ding et al., 2023b; Jimenez et al., 2024). However, these studies do not fully capture the complexity of real-world software development, which often requires the use of **rapidly-evolving** public libraries. To fill the gap, we introduce LIBEVOLUTIONEVAL, a detailed study requiring an understanding of library evolution to perform in-line code completion accurately. LIBEVOLUTIONEVAL provides a version-specific code-completion task comprised of eight libraries (`torch`, `torchvision`, `scipy`, `pil`, `tqdm`, `pyyaml`, `matplotlib`, and `pandas`) as they evolve over the year along with a detailed analysis of the evolution of two popular and well-maintained public libraries: `PyTorch` and `Matplotlib`. We evaluate popular public models and find that public library evolution significantly influences model performance. We explored mitigation methods by studying how retrieved version-specific library documentation and prompting can improve the model's capability in handling these fast-evolving packages, paving a promising future path in better handling fast-evolving libraries.

1 Introduction

Large Language Models for code (*a.k.a.* code LLMs) (Li et al., 2023; Lozhkov et al., 2024; Roziere et al., 2023) have significantly advanced developer productivity through improved code completion tasks. These models are pivotal not only in code completion, but also in debugging, code summarization, and language translation for software development (Yan et al., 2023; Roziere et al., 2020, 2022; Min et al., 2024). These models are usually evaluated either with code contest dataset (Li et al., 2022) or with a focus on local files for context to enhance the completion of the function (Chen et al., 2021; Ding et al., 2023a; Athiwaratkun et al., 2023; Ding et al., 2023b; Jimenez et al., 2024). However, these studies do not fully encompass the complexities of real-world software development, which requires public libraries. Complexity of code completion with public library APIs increases, as the APIs often evolve—some APIs change their signature, some gets deprecated, while many new APIs surfaced in this evaluation process (McDonnell et al., 2013). While some works perform code completion involving public libraries (Liao et al., 2023; Zan et al., 2022), use documentation of the library for prediction (Qin et al., 2024), and show that zero shot code completions suffer from hallucinations (Patil et al., 2023), these works do not focus on the rapidly evolving nature of public libraries.

Large Language Models are trained on extensive corpora of open-source code, which likely includes public libraries. Consequently, while LLM-generated code may appear reasonable, it might not be accurate for the specific version of the library being used, leading to version-dependent performance issues. Figure 1 shows that Code LLM's generation is correct for `v2.2` but incorrect for `v1.2`. This variability is significant since developers often work with different library versions – newest versions for current projects and older ones for legacy code maintenance. Therefore, the developer's experience with coding assistants depending on LLMs for code completion can vary greatly depending on their specific use case.

Existing benchmarks and studies do not fully capture evolution, revealing a gap in our current evaluation and understanding of code LLMs. This work focuses on the following research questions: (1) Does the performance of code LLMs change as the library evolves? (2) If yes, can retrieving version-specific meta-data like library documentation mitigate the impact of library evolution on code completion? (3) With the evolution of libraries, new

* Work done at AWS AI Labs. Correspondence: {skuhar, rabaisha}@amazon.com

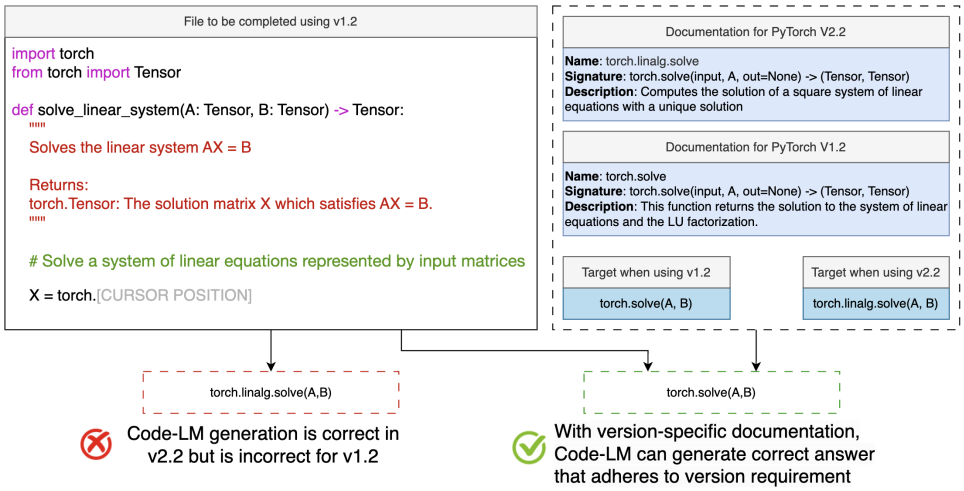


Figure 1: An example of a code completion scenario under LIBEVOLUTIONEVAL. The incomplete code snippet on the left requires the correct API method to solve a linear system specified by two PyTorch tensors. The code LLM performs incorrect code completions due to version mismatch. The version-specific documentation is a potential augmentations that can assist the LLM to perform correct and version-dependent completion.

relationships between APIs are introduced, while existing ones are altered. Can code LLMs effectively adapt to these evolving relationships between APIs? (4) Does the introduction, modification, and deprecation of APIs as libraries evolve to make it more challenging for code LLMs to perform accurate code completions?

To investigate these research questions, we introduce LibEvolutionEval, a benchmark and detailed study specifically designed to understand the impact of public library evolution on code completion. LibEvolutionEval offers version-specific code-completion tasks spanning multiple years for eight libraries – `torch`, `torchvision`, `scipy`, `pil`, `tqdm`, `pyyaml`, `matplotlib`, and `pandas`.

LibEvolutionEval also performs a detailed analysis of the evolution of two popular libraries (`torch` and `matplotlib`) by providing version-specific meta-data, documentation retrieval tasks, and code-completion tasks. It requires code LLMs to perform version-specific code completions under both realistic (where the evaluation examples are sampled from permissively licensed GitHub repositories that is publicly available) and controlled scenarios (where a template uses API documentation to create evaluation examples that is not publicly available). It provides version-specific API documentation to investigate the impact of library evolution on embedding models during retrieval. It offers tasks based on completion type to compare: (1) completions guided by import statements and clear library prefixes with (2) completions that are object-oriented references and do not have a library-defined prefix. We call them *direct* and *indirect* code completions (see Figure 3), and such tasks evaluate models' ability to adapt to evolving relationships between APIs. Furthermore, LibEvolutionEval also offers tasks based on granularity that compares overall developer experience with performance on specific APIs that have been newly introduced, modified, or deprecated as the library evolves, providing us with insights on the impact of evolution against overall code-completion performance.

We conducted a comprehensive evaluation using widely used code LLMs (Lozhkov et al., 2024; Jiang et al., 2023; OpenAI, 2024) and embedding models (Zhang et al., 2024; OpenAI, 2022) to report the following insights.

- Code LLMs and embedding models exhibit substantial performance variation as public libraries evolve. Providing version-specific API documentation as a context improves code completion performance but does not entirely address inherent version-based bias in version-specific code completions.
- Code LLMs perform indirect API completions better than direct API completions, demonstrating an understanding of evolving relationships between APIs.
- Introduction, modification, and deprecation of APIs make it harder for code LLMs to perform code completion where new models might forget old deprecated APIs while old models cannot predict the latest APIs.

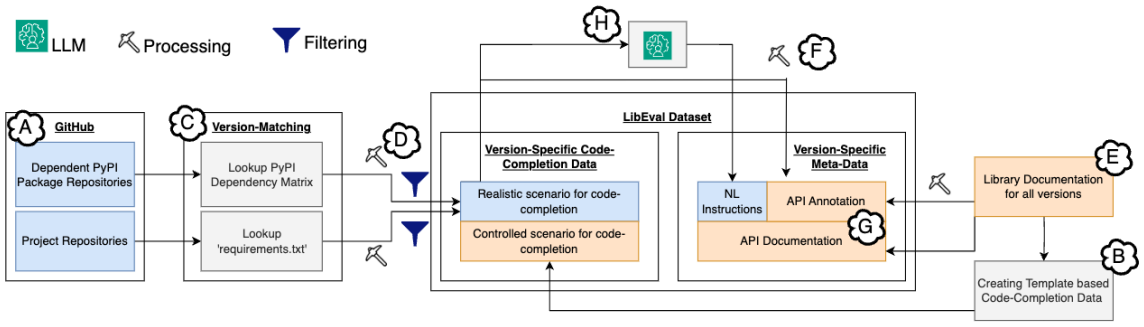


Figure 2: LIBEVOLUTIONEVAL's preprocessing pipeline to obtain version-specific code-completions meta-data including documentation, NL instructions, and API annotation.

2 LIBEVOLUTIONEVAL: Version-Specific Code Completions

Each code completion example in LIBEVOLUTIONEVAL consists of code prompts ending at a position where the LLM is tasked to complete the missing expression, typically involving one or more API calls to the public library under consideration as shown in Figure 1. The uniqueness of this dataset lies in its emphasis on version-specific API usage, reflecting scenarios where developers use LLMs to perform code completions for different versions of the same library. We evaluate code completions under two scenarios: *realistic* (GitHub based) and *controlled* (documentation based).

2.1 Version-Specific Evaluation Creation

API Usage Collection For a realistic scenario, we focus on data written by real-world developers, specifically from permissively licensed GitHub repositories (Figure 2-(A) and Figure 7 in the appendix). This allows us to understand if the impact of API evolution is significant with unknown confounding variables present in real-world code.

For detailed ablations on the other hand, we also simulate a controlled scenario by creating synthetic data for `Matplotlib` and `PyTorch` by taking API documentation and converting it to evaluation examples using a template. The template is designed to make the code LLM predict the API name given its description, service name, and mandatory arguments in the left context (Figure 2-(B) and Figure 8 in the appendix). This allows us to isolate the impact of API evolution without the confounding variables found in real-world code, such as variations in coding styles.

Versioning of API Usage For a realistic setting, the GitHub repositories have a 'requirements.txt' file that mentions the exact version of the library used to develop it. Additionally, if the GitHub repository is a PyPI package (like `torchvision`) that depends on the library under consideration (like `torch`), we use the dependency matrix between different packages to match API usage with the library version (Figure 2-(C)). Next, since the data is created from API documentation for the controlled setting, the version of the API usage example is the same as that of the documentation from which it is derived.

API Evaluation Example Creation To ensure the quality of our proposed benchmark, we employ a series of rule-based and model-based post-processing filters (Figure 2-(D)). We typically restrict the left context provided to the model to the scope of the API being completed, limiting it to the class containing the API call. If no class is present, we include the entire preceding context of the line. Import statements of the target library are also included to provide the LLM with contextual clues (see §B in appendix). Additionally, the initial regex of the API expression (e.g., `torch` from `torch.solve()` as illustrated in Figure 1) is placed just before the cursor position to encourage the model to complete the API expression correctly. Comments are removed from contexts to minimize the risk of API leakage.

API Evaluation Example Selection When encountering multiple API calls on the same line (e.g., `x = torch.ones(x) + torch.zeros(y)`), if an API (e.g., `torch.ones`) has already been included in the evaluation dataset from this line, subsequent APIs on that line (e.g., `torch.zeros`) are excluded. This approach ensures diversity in the contexts represented in the dataset. Additionally, we discard examples if the corresponding API call already exists within the collected evaluation dataset for the

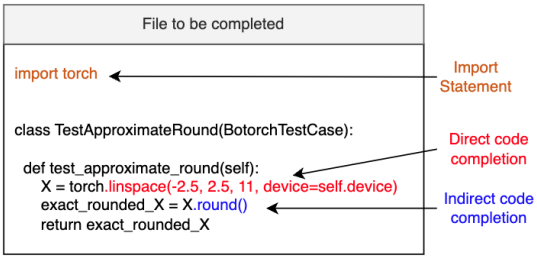


Figure 3: APIs classification based on *completion type*.

same source (e.g. GitHub repository). This means that if we have already included an example of a specific API (e.g., `torch.ones`) from a particular source, we will exclude any additional examples from that same source that use the same API (i.e., `torch.ones`). This strategy aims to ensure the diversity of API calls within the evaluation dataset.

Documentation Collection We systematically collect documentation for detailed analysis of library evolution from their publicly available websites (Figure 2-(E)). This includes comprehensive details such as API signatures, names, types, input parameters (noting it's optional/mandatory nature), and code usage examples. This documentation serves as a foundation for understanding these APIs' expected usage and evolution over time.

2.2 API Data Classification

Completion Type This classification assesses an LLM's ability to track the evolving relationships between APIs as a library changes. It does so by comparing completions based on import-driven prefixes with those using open-vocabulary prefixes (Figure 3 and Figure 2-(F)). These are:

- **Direct Code Completions**: These completions are driven by import statements, with prefixes derived directly from the public library's import statements. As an example, `nn.ReLU()` is a direct API completion from `import torch.nn as nn`.
- **Indirect Code Completions**: These completions lack a well-defined prefix which originates from referenced objects instantiated through direct API calls. Figure 3 shows that variable `X` is defined by `X = nn.linspace` and is later used in `X.round`. These completions test a model's deeper contextual understanding, requiring it to identify the corresponding direct API call and comprehend the library's version-specific relationships between APIs to achieve accurate code completion.

Type	Old Version	Current Version	Next Version
Introduced API	Not Supported	API.Foo(x)	API.Foo(x)
Deprecated API	API.Foo(x)	API.Foo(x)	Not Supported
Modified API	API.Foo(x, y)	API.Foo(x)	
Modified API		API.Foo(x)	API.Foo(x, y)

Table 1: Classification of APIs based on *granularity*.

Granularity This evaluation examines LLMs' ability to adapt to rapid API changes, including introductions, deprecations, and modifications as the library evolves. We annotate APIs using their documentation as (Table 1 and Figure 2-(G)):

- **Introduced**: API added in the current version and not present in the previous version.
- **Deprecated**: API present in the current version but removed in the next version.
- **Modified**: The name of the API does not change but its arguments are updated.
- **Unchanged**: API does not change when compared to the previous/next version.

We then cross-reference these annotated APIs with the version-specific evaluation examples. This allows us to label both API documentation (see Figure 4) evaluation examples based on granularity to create subsets for analysis.

2.3 LLM Context Classification

In-File Context We assess the models' code completion capabilities using only the context available within the current file, replicating a typical development environment scenario. The import statements, the right context, and the left context extracted are given to the model (see Figures 1 and 13a). This methodology ensures that our evaluation accurately reflects the practical conditions faced by developers that use version-unaware code completions and is aimed to serve as a baseline.

Library Version-Aware Context While the in-file context mimics a realistic code-completion setting , there still exists ambiguity for the LLM to perform code completion. For example, there might be two valid responses based on the in-file context corresponding to two different versions of the library, as shown in Figure 1. To mitigate this issue, we add a comment before left context that tells the LLM the version of the library under consideration.

Version-Specific Retrieved API-Context Development on the success of retrieve-and-generate frameworks for repository-level code completions (Zhang et al., 2023; Ding et al., 2023b), we adapt this retrieve-and-generate approach for the

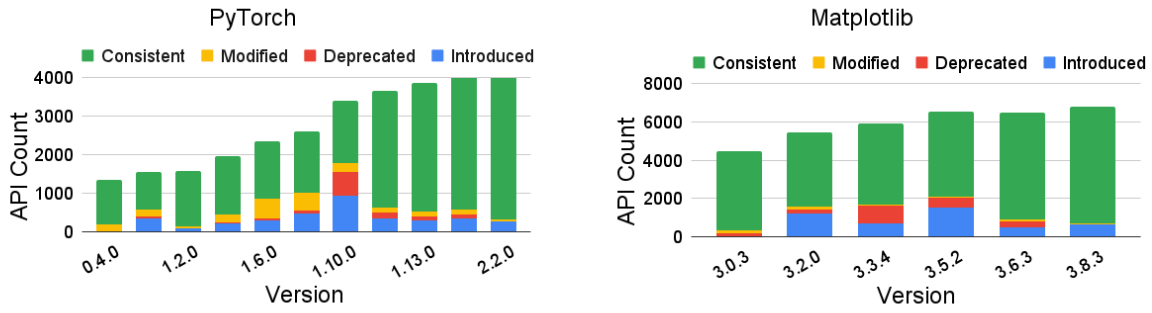


Figure 4: Illustration of the evolution of PyTorch and Matplotlib public libraries over time. This highlights the rapid evolution of modern public libraries.

Feature	Assorted	PyTorch	Matplotlib
# API Documentations	-	29.4K	35.6K
# Eval Examples	4.5K	20.1K	10.1K
Avg. # lines in prompt	66.25	104.91	84.36
Avg. # tokens in prompt	732.06	1149.34	995.91
Avg. # lines in reference	1.27	1.21	1.25
Avg. # tokens in reference	18.74	13.73	17.47

Table 2: LIBEVOLUTIONEVAL statistics.

retrieval of public library documentation. Our documentation retrieval database is organized by versions of public libraries. It contains comprehensive metadata, including API signatures, input parameters, usage examples, and detailed natural language descriptions of APIs and their parameters. For each code completion task, we generate a query using the natural language instruction describing the developer's intent. This is done by giving target code completion to Anthropic's Claude v2 (Anthropic, 2023) and asking it to not reveal the regular expressions corresponding to the name and input arguments of target code completion (refer to § C for the Appendix and Figure 2-(H)). We utilize an embedding model (CodeSage (Zhang et al., 2024) by default) to determine the similarity between the query and the available API entries, selecting the top 3 matching APIs. These are then formatted as commented code, incorporating API signatures and parameters, placed before the left context to serve as the version-specific documentation as shown in Figure 13b.

2.4 Dataset Statistics and Scope

Statistics We present the statistics of LIBEVOLUTIONEVAL in Table 2. We use the StarCoder tokenizer (Li et al., 2023) to compute the number of tokens. For version-specific characteristics, see §F in the appendix.

Scope In addition to left contexts and target code completions, we include the subsequent code lines from the source code files in LIBEVOLUTIONEVAL examples. By providing the source code lines both to the left (prompt or prefix) and to the right (suffix) of the references, LIBEVOLUTIONEVAL enables the evaluation of code LLMs for their fill-in-the-middle (FIM) capabilities (Bavarian et al., 2022). Furthermore, the meta-data from documentation allows us to conduct evaluations using RAG.

3 Experimental Setup

Models We benchmark public code LLMs: Mistral (Jiang et al., 2023), StarCoder2 (Lozhkov et al., 2024), GPT-4o-mini (OpenAI, 2024) and CodeGen 1.0 (Nijkamp et al., 2023). We benchmark version-specific retrieval tasks using CodeSage (Zhang et al., 2024) and OpenAI-ada-002 (OpenAI, 2022). Lastly, we conducted scaling experiments with StarCoder (Li et al., 2023) (1B, 3B, 7B), StarCoder2 (3B, 7B, 15B), and CodeSage (Small, Large).

Evaluation Metrics For code completion, we concentrate on the correctness of APIs called by calculating the F1 score (Ding et al., 2023b). For documentation retrieval, to evaluate the performance of embedding models by using Mean Reciprocal Rank (MRR), assessing how well they retrieve version-specific documentation and whether their performance varies with library evolution.

Inference We maintain uniform hyperparameters across all models. The maximum sequence length is 8K tokens, with each context trimmed to include the nearest 4K tokens from the API expression. A maximum generation length is 128 tokens. We report the results of the greedy search. During the post-processing phase we check if the source code following target code completion is being generated; if so, the generation is truncated accord-

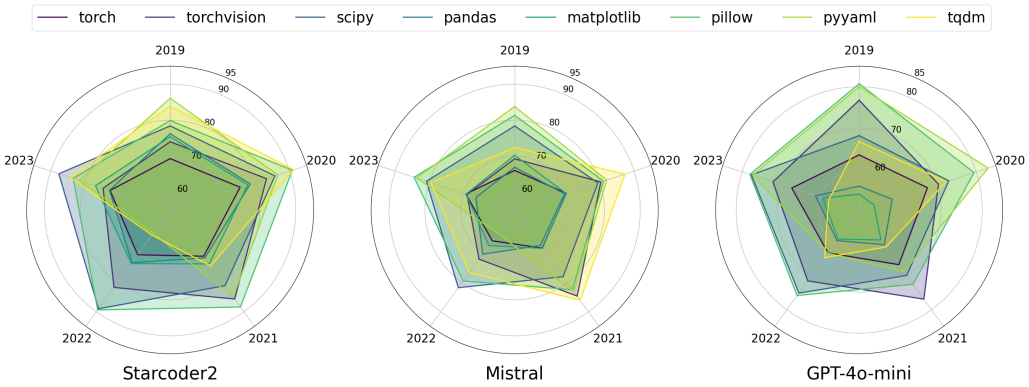


Figure 5: Illustration of the code completion performance of the Starcoder2, Mistral, and GPT-4o-mini models by measuring the F1 score. The performance of code LLMs varies significantly as libraries evolve.

Model	Completion Strategy	Context Setting	PyTorch	Matplotlib
Starcoder2-7B	Fill-in-the-Middle	In-File (Not Version-Aware)	68.8	69.7
		+ Version-Aware	69.3	70.1
		+ Version-Aware RAG	**73.3**	**75.4**
Mistral-7B	Left-Context Only	In-File (Not Version-Aware)	65.8	60.18
		+ Version-Aware	66.04	61.2
		+ Version-Aware RAG	**67.6**	**69.05**
GPT-4o-Mini	Instruction-based (w/ Example)	In-File (Not Version-Aware)	64.3	52.5
		+ Version-Aware	64.78	53.1
		+ Version-Aware RAG	**70.14**	**66.7**

Table 3: Code completion performances under different input types and context prompting strategies. Each model is evaluated in three context settings: in-file (not version-aware), version-aware, and version-aware RAG.

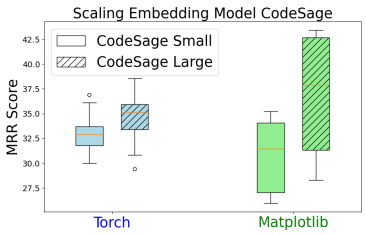
ingly. We transform the newly generated text, into an AST to extract API expressions (Ding et al., 2023b). If no API expressions are identifiable, the generation is left unchanged. We apply the same post-processing on the target completions before calculating the evaluation metrics.

4 Results

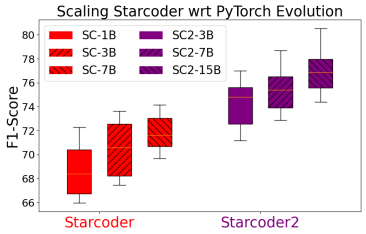
Library evolution impacts code LLM performance We evaluate how the performance of code LLMs changes as libraries evolve by performing code completions for eight libraries: `torch`, `torchvision`, `scipy`, `pandas`, `pillow`, `pyyaml`, and `tqdm`. This evaluation uses StarCoder2, Mistral, and GPT4o-mini (more in §I). As shown in the first two columns of Table 3, these models employ different completion strategies: StarCoder2 uses fill-in-the-middle, Mistral utilizes left-context only, while GPT4o-mini follows an instruction-based approach with a one-shot example. All eight libraries are benchmarked in realistic scenarios. Figure 5 shows that the developer experience can vary significantly across all models and libraries as public libraries evolve, highlighting the need for better model adaptation to API changes.

Version aware contexts enhance code LLM performance Table 3 demonstrates a clear improvement in model performance as additional contextual information is provided during the code completion task. In the baseline *In-File* setting, where the models rely solely on the code context within a file, the performance is the lowest across all models. Introducing version awareness significantly enhances accuracy, as models can better disambiguate API usage across different library versions. The most notable improvements occur in the *Version-Aware RAG* setting, where documentation relevant to the specific library version is retrieved and used to further refine API completions. This enriched context enables models to generate more precise completions by taking into account the evolving API landscape. These results emphasize the critical role that version-specific and dependency-aware contexts play in improving the accuracy and reliability of code completions.

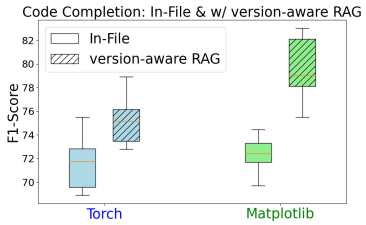
Furthermore, Figure 6c visualizes the impact of using version-aware RAG compared to in-file context settings for the StarCoder2 model, focusing on the evolution of `PyTorch` and `Matplotlib`. Although version-aware RAG consistently improves the per-



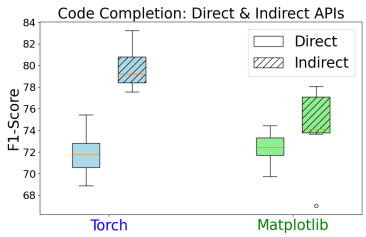
(a) MRR Scores for CodeSage Small and Large Models



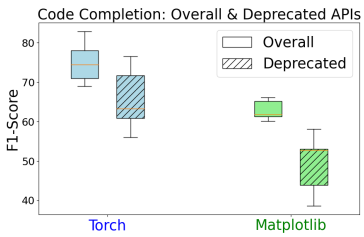
(b) F1 Scores for Starcoder2 and Starcoder Models with respect to PyTorch



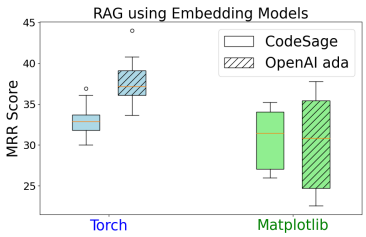
(c) In-File vs Version-Aware RAG Performance for Starcoder2



(d) Direct vs Indirect Code Completion for Starcoder2



(e) Overall vs Deprecated Set Code Completion for Starcoder2



(f) MRR Scores for open CodeSage and Open-Ada-002 models.

Figure 6: Detailed analysis of the impact on code completion as `PyTorch` and `Matplotlib` evolve using API-completion and documentation retrieval tasks.

formance of version-specific code completions, it does not fully address the model's internal bias toward certain versions of the libraries. The box plot highlights variance in the model's predictions, suggesting that despite the enhanced contextual information, underlying biases in the LLM remain, likely stemming from the uneven distribution of training data across library versions.

Library evolution impacts documentation retrieval using embedding models We measure the performance of version-specific documentation retrieval as libraries evolve, using embedding models. As shown in Figure 6f, we benchmark public CodeSage-Small and the closed-source OpenAI Ada models on `Torch` and `Matplotlib`. We observe that the performance of these embedding models fluctuates with the evolution of libraries. This insight sheds light on why version-aware RAG enhances performance for version-specific code completion tasks, but cannot fully resolve the variance in performance across different library versions. The embedding models themselves exhibit bias toward certain library versions, explaining the persistent performance gaps in version-specific code completions.

Impact of scaling and model updates on handling evolving APIs We evaluate the impact of scaling both embedding models (Figure 6a) and code-completion models (Figure 6b) across different sizes to observe improved retrieval and code completion performance respectively. However, performance still fluctuates as libraries evolve, suggesting that while scaling improves results, it does not fully address the challenges posed by evolving libraries. Furthermore, updating to newer versions, such as from StarCoder-7B to StarCoder2-7B shows that while performance improvements overall are observed (due to better training methods), these do not address the biases introduced by the evolution of public libraries. This points to a need for more specialized training techniques, such as fine-tuning using versioned datasets or incorporating explicit temporal data about API evolution.

Direct vs indirect API completion We evaluate the performance of direct and indirect code completions as libraries evolve in Figure 6d. The model exhibits better performance for indirect code completions than direct code completions. From subsection 2.2, we know that every indirect code completion example contains a corresponding parent direct API call in the left context. We posit that the code LLM can understand that it needs to perform code completion so that the generated code serves as an attribute related to this version-specific parent direct API call present in the left context.

Impact of different APIs on code completion In Figure 6e, we compare overall performance with the subset of deprecated APIs for code completions in `PyTorch` and `Matplotlib`, focusing on a controlled setting. The results show that code LLMs

(a) StarCoder2 (Model Release: 2024) on Matplotlib Deprecated APIs.

Version Year	Deprecated API Score	Overall Score
2019	38.58	**61.83**
2020	43.93	**60.12**
2021	53.01	**61.31**
2022	52.74	**65.15**
2023	57.14	**66.08**

(b) CodeGen 1.0 (Knowledge Cutoff: 2022) on Matplotlib Introduced APIs.

Version Year	Introduced API Score	Overall Score
2020	53.14	**62.89**
2021	54.23	**62.85**
2022	56.29	**60.04**
2023	44.08	**59.44**
2024	41.37	**58.57**

(c) StarCoder2 (Knowledge Cutoff: 2024) on PyTorch Introduced/Deprecated APIs.

Version Year	Deprecated/Intro--duced API Score	Overall Score
2018	68.78	**71.06**
2020	59.10	**75.45**
2021	68.13	**72.84**
2022	60.93	**71.02**
2023	67.13	**72.82**

Table 4: Performance comparison of models on different API sets across library versions. Underlined scores indicate a significant performance drop while bold scores are maximum across the two settings considered.

struggle with deprecated APIs, consistently performing worse compared to the overall set. This observation aligns with our qualitative results that models prefer API calls for newer versions of the library (see Figure 1 and 9). To verify this in a realistic scenario, Table 4c provides a version-by-version analysis for PyTorch, comparing newly introduced and deprecated APIs. We observe consistently lower performance on deprecated APIs, except for 2018, a year dominated by newly introduced APIs, which are now widely adopted. These findings demonstrate that rapid changes to API impact models' ability to complete code accurately, with deprecated APIs posing particular challenges.

Temporal analysis of model performance on introduced and deprecated APIs The tables compare the performance of StarCoder2 and CodeGen-1.0 on `Matplotlib`'s deprecated and introduced APIs, respectively, in a controlled setting. Table 4a shows that StarCoder2 struggles with older, deprecated APIs from 2019 and 2020, indicating that API forgetting contributes to version-specific performance drops. In contrast, Table 4b highlights a sharp decline in CodeGen-1.0's performance on introduced APIs from 2023 and 2024, revealing its 2022 knowledge cutoff. This suggests that such tasks could be used to estimate a model's knowledge cutoff. In general, the results underscore the need to develop model training techniques to better handle library evolution.

5 Related Works

Large language models for code excel in various software development tasks (Yan et al., 2023; Roziere et al., 2020, 2022; Min et al., 2024), facilitating the developments of coding assistants. Similarly, developments in code embedding models used for retrieval (Robertson et al., 2009; Guo et al., 2022; Zhang et al., 2024; OpenAI, 2022) have further enhanced LLMs' capabilities. In this journey, evaluation benchmarks have played a pivotal role with numerous works developing benchmarks to evaluate code LLMs (Zheng et al., 2023; Cassano et al., 2023; Hendrycks et al., 2021; Lu et al., 2021; Puri et al., 2021; Clement et al., 2021; Ding et al., 2023a; Wang et al., 2023; Lu et al., 2022). These studies typically assess code completion abilities given local file contexts, both in-file (Chen et al., 2021; Athiwaratkun et al., 2023; Lu et al., 2021) and repository-level (Ding et al., 2023b; Zhang et al., 2023; Liu et al., 2024; Ding et al., 2024). However, they do not fully encompass the complexities of real-world software development, which requires extensive use of public libraries. Some works have explored code completion involving public libraries (Liao et al., 2023; Zan et al., 2022; Qin et al., 2024; Patil et al., 2023), but they do not address the rapidly evolving nature of these libraries. To fill this gap, we introduce LIBEVOLUTIONEVAL that evaluates the performance of LLMs on code completion across multiple versions of public libraries, capturing their evolution and reflecting real-world scenarios where developers interact with different versions of the same library.

6 Conclusion

In this paper, we introduced LIBEVOLUTIONEVAL, a comprehensive benchmark specifically designed to assess the performance of Code Large Language Models (code LLMs) in code completion tasks as public libraries evolve. Our results demonstrate significant variability in LLM performance based on the API version, highlighting the challenges of handling library evolution. The findings underscore the necessity for future advancements in code completion technologies to consider the dynamic nature of public libraries, aiming to improve developer productivity and accuracy in real-world settings.

Acknowledgements: The authors also thank Ming Tan and Hantian Ding for their constructive feedback during the paper writing process. Additionally, we would like to thank team members from Amazon Q Developer for their insightful discussions, which have contributed to the refinement of our work.

References

Anthropic. 2023. *Claude*.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Colin Clement, Shuai Lu, Xiaoyu Liu, Michele Tufano, Dawn Drain, Nan Duan, Neel Sundaresan, and Alexey Svyatkovskiy. 2021. Long-range modeling of source code files with eWASH: Extended window access by syntax hierarchy. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4713–4722, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023a. A static evaluation of code completion by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 347–360, Toronto, Canada. Association for Computational Linguistics.

Yangruibo Ding, Zijian Wang, Wasi U. Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, Torino, Italia. ELRA and ICCL.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023b. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li.

2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE.

Marcus J. Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2024. Beyond accuracy: Evaluating self-consistency of code large language models with identitychain. In *The Twelfth International Conference on Learning Representations*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

OpenAI. 2022. Embedding ada-002. `https://platform.openai.com/docs/guides/embeddings/using-embeddings`.

OpenAI. 2024. Hello gpt-4o. `https://openai.com/index/hello-gpt-4o/`. [Accessed: October 10, 2024].

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. ReCode: Robustness evaluation of code

generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5067–5089, Singapore. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2369–2375. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Codesage: Code representation learning at scale. In *The Twelfth International Conference on Learning Representations*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Supplementary Material: Appendices

A Limitations

This study involves a zero-shot approach to evaluate the impact of the evolution of public libraries on Code LLMs. Pre-training a model exclusively with version-specific data from public libraries might help to reduce the version-dependent discrepancies observed in zero-shot settings. Additionally, it is important to acknowledge that CodeLMs are trained on vast repositories of unlabeled code, raising the possibility that the model might have previously encountered some of the evaluation data. This potential overlap should be carefully considered when interpreting the results of this study.

File to be completed

```
import torch


class Test_JIT(unittest.TestCase):
    def _get_script_module(self, f, *args):
        class MyModule(torch.jit.ScriptModule):
            def __init__(self):
                super(MyModule, self).__init__()
                self.module = f(*args)
                self.module.eval()

            @torch.jit.script_method
            def forward(self, tensor):
                return self.module(tensor)

        return MyModule()

    def _test_script_module(self, tensor, f, *args):
        jit_out = self._get_script_module(f, *args).cuda()(tensor)
        py_out = f(*args).cuda()(tensor)

        self.assertTrue(torch.allclose(jit_out, py_out))

    def test_torchscript_spectrogram(self):
        @torch.jit.script
        def jit_method(sig, pad, window, n_fft, hop, ws, power, normalize):
            return F.spectrogram(sig, pad, window, n_fft, hop, ws, power, normalize)

        tensor = torch.rand((1, 1000))
        n_fft = 400
        ws = 400
        hop = 200
        pad = 0
        window = [CURSOR POSITION]
```

Target Completion

```
torch.hann_window(ws)
```

Figure 7: Examples of API evaluation.

```
import logging
import matplotlib
import mpl_toolkits

def test_zlim3d_api(bottom, top, emit, auto, *, zmin, zmax):
    """
    Testing for Set 3D z limits.  See matplotlib.axes.Axes.set_ylim() for full documentation
    """

    # Creating a logger object
    logger = logging.getLogger(__name__)

    # Validates mandatory parameters for null and empty values
    if bottom is None or top is None or emit is None or auto is None or * is None or zmin is None or zmax is None:
        logger.error("Test parameters should not be null or empty.")
        return None

    try:
        response = mpl_toolkits.[CURSOR_POSITION]
        logger.info("test_zlim3d_api executed successfully.")
    except Exception as exception:
        logger.exception("Error occurred while executing test_zlim3d_api : ", exception)
        raise

    return response
```

Figure 8: Matplotlib API evaluation example created synthetically from documentation.

File to be completed using v1.6

```
import torch
from torch import Tensor
from torch.autograd import gradcheck
from torch.jit.annotations import Tuple
from torch.nn.modules.utils import _pair


class RoIAlignTester(RoIOpTester, unittest.TestCase):

    # Truncated lines of code

    @unittest.skipIf(not torch.cuda.is_available(), "CUDA unavailable")
    def test_roi_align_autocast(self):
        for x_dtype in (torch.float, torch.half):
            for rois_dtype in (torch.float, torch.half):
                with
```

Target Completion	CodeLM's Answer:
torch.cuda.amp.autocast()	torch.autocast(x_dtype)

Task requires version-specific code completion

Code-LM's predicted API exists in v1.10 and not v1.6

Figure 9: Qualitative example of an error in code completion on LIBEVOLUTIONEVAL focused set.

B LIBEVOLUTIONEVAL Generation

Realistic Scenario Python files using the library are processed to extract library usage patterns by parsing the files into Abstract Syntax Trees (ASTs) using the `tree-sitter` library. This comprehensive approach allows us to identify syntactic elements such as function calls and import statements specific to the library. The AST is systematically traversed to detect both direct and indirect API calls to the library and its submodules. Direct API calls are identified by their explicit invocation in the source code, typically involving function calls directly on the library modules (e.g., `torch.nn.Linear`). Indirect API calls are recognized through variables or objects that are assigned to library functions or classes and used later in the code, which requires tracking variable scopes and aliases across the codebase. The broader structural context for a direct API call is determined by locating the closest enclosing syntactic structure, such as a function or a class method, in the AST. This enclosing structure is regarded as the scope of the API call. The entire block of code constituting this scope is extracted as a context. This context includes parameter lists, internal variable declarations, and other code elements within the same block, providing a comprehensive view of how the API is integrated into the function. The context for indirect API calls includes not only the block where the variable is used but also poten-

tially broader code segments that influence or are influenced by the variable use. This methodical extraction of context ensures that each API call, whether direct or indirect, is analyzed within its operational environment concerning left context.

Controlled Scenario The library usage data for controlled ablations was created synthetically using the documentation for each version. Each documentation is converted into a code completion example to be used for evaluation using a template (see Figure 8). The template highlights the service name, API description, and mandatory arguments and does not leak the name of the API.

C Generating Natural Language Instructions Using Claude

```
f"""Given the following API:
{api}
Generate a single-sentence instruction of natural language that describes the API functionality
without explicitly mentioning the name or any specific parameters/values of the API. This instruction
would be
used as a code comment to assist with code completion for the same API.

<INSTRUCTION>
[Generated natural language instruction goes here]
</INSTRUCTION>


The generated instruction should:
- Be a single simple sentence
- Convey clearly the primary functionality of the API
- Use simple, concise language suitable for a code comment
- Avoid repetition or redundancy with the API itself
- Not explicitly mention the API name or any specific parameters/values

The goal is to provide a helpful prompt for a LLM to predict the API based on the surrounding code
context  and instruction.

Example:
For the API "torch.optim.SGD(global_model.parameters(), lr=0.1)", a suitable instruction could be:
<INSTRUCTION>
Initialize a stochastic gradient descent optimizer for model training.
</INSTRUCTION>
"""
```

Figure 10: The template used to prompt Claude to create natural language instructions from a target code completion expression. The template explicitly guides the LLM not to provide the name or the arguments required for the code completion in the natural language instruction.

D Retrieval Performance vs Model Size on LIBEVOLUTIONEVAL

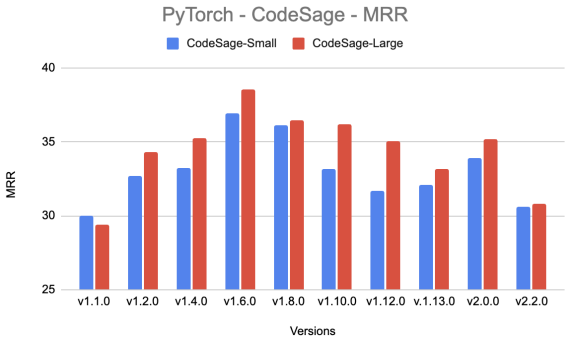
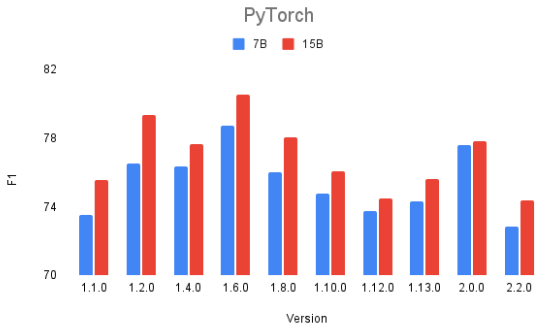
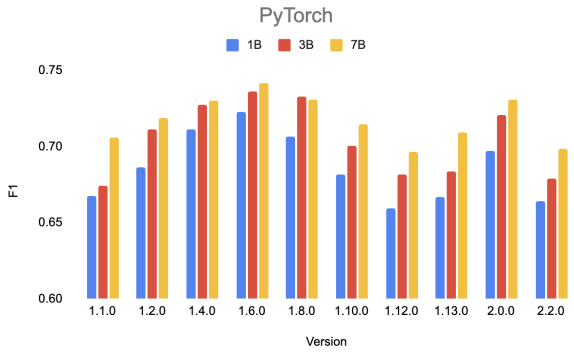


Figure 11: Larger CodeSage models perform better at documentation retrieval

E Code Completions Performance vs Model Size on LIBEVOLUTIONEVAL



(a) Starcoder2 15B and 7B on PyTorch.



(b) Starcoder 1B, 3B, and 7B on PyTorch.

Figure 12: Larger Starcoder and Starcoder2 models perform better at code completions.

F Detailed Statistics of Data

Main Result

Library	2019	2020	2021	2022	2023
torch	286	513	708	835	738
torchvision	16	16	14	16	16
scipy	70	70	70	70	68
pandas	60	60	60	59	60
matplotlib	90	90	90	89	90
pillow	40	40	40	40	40
pyyaml	8	8	8	Na	8
tqdm	12	12	12	12	12

Table 5: API Examples per Year for Libraries

Matplotlib Library Ablation Data

Version	Deprecated	Overall	Direct	Indirect	Introduced
3_0_3	40	1000	297	359	N/A
3_2_0	40	1000	154	175	200
3_3_4	40	1000	339	395	200
3_5_2	40	1000	291	298	200
3_6_3	40	1000	84	31	200
3_8_3	N/A	1000	37	8	200

Table 6: Comparison between Matplotlib Focussed vs. Comprehensive dataset.

PyTorch Library Data

Table 7: PyTorch API Usage Data

Version	Direct	Indirect	Intr/Depr	Deprecated	Overall
v_1_1_0	286	174	50	N/A	1000
v_1_2_0	341	189	35	9	1000
v_1_4_0	452	262	45	14	1000
v_1_6_0	513	300	50	N/A	1000
v_1_8_0	564	353	50	40	1000
v_1_10_0	708	466	50	40	1000
v_1_12_0	876	543	0	40	1000
v_1_13_0	835	503	0	40	1000
v_2_0_0	738	441	36	40	1000
v_2_2_0	680	375	0	N/A	1000

F.1 PyTorch Documentation Data

Table 8: Torch version changes over time.

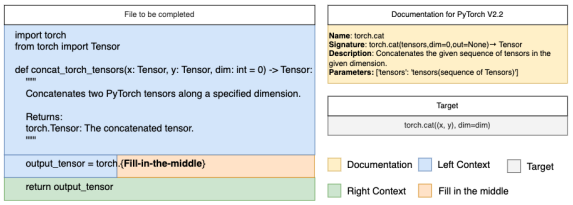
Torch Version	Total	New	Deleted	Modified	Consistent
0.4.0	1187	0	8	172	1179
1.1.0	1518	339	44	198	963
1.2.0	1548	74	10	43	1438
1.4.0	1752	214	14	225	1507
1.6.0	2031	293	48	523	1482
1.8.0	2455	472	77	466	1591
1.10.0	3324	946	596	237	1631
1.12.0	3625	353	135	126	3051
1.13.0	3784	294	113	120	3337
2.0.0	4018	347	87	148	3504
2.2.0	4187	256	0	68	3863

F.2 Matplotlib Documentation Data

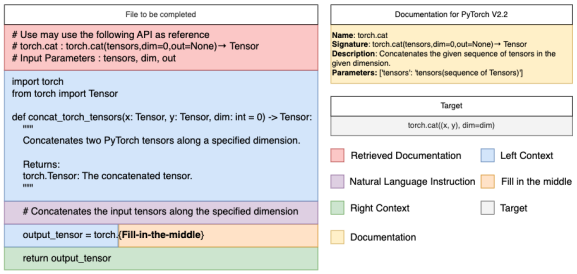
Table 9: API changes across versions.

Version	New	Deleted	Modified	Consistent
3.0.3	0	180	135	4141
3.2.0	1191	249	152	3875
3.3.4	705	909	69	4240
3.5.2	1505	523	84	4407
3.6.3	478	292	133	5571
3.8.3	631	0	59	6123

G Visualizing prompts given to CodeLMs in LIBEVOLUTIONEVAL



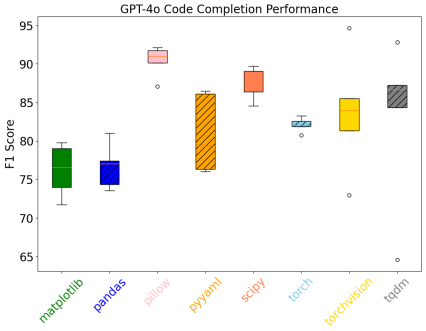
(a) Prompt in a zero-shot setting



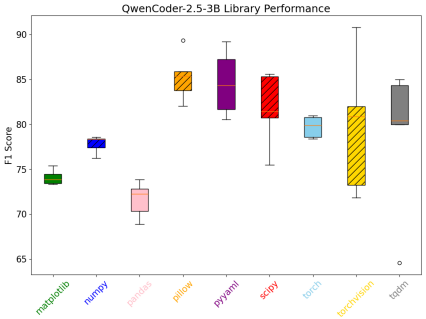
(b) Prompt when augmentation is provided

Figure 13: Visualizing prompts given to code LLMs in LIBEVOLUTIONEVAL.

H Additional Model Benchmarking



(a) GPT-4o benchmarked on LIBEVOLUTIONEVAL.



(b) QwenCoder-2.5 benchmarked on LIBEVOLUTIONEVAL.

Figure 14: Comparison of GPT-4o and QwenCoder-2.5 benchmarked on LIBEVOLUTIONEVAL.

I BM25 RAG Experiment

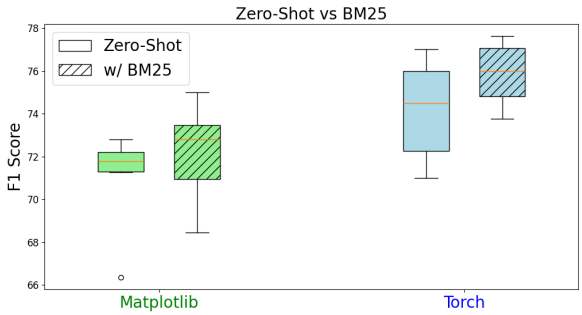


Figure 15: Zero shot and BM25 on code completion.