A Study of LLMs' Preferences for Libraries and Programming Languages

Lukas Twist[1] **Mark Harman**[2] **Don Syme**[3]
Joost Noppen[1,4] **Helen Yannakoudakis**[1]
Detlef Nauck[4] **Jie M. Zhang**[1]

[1]King's College London, London, UK
[2]University College London, London, UK
[3]GitHub Next, London, UK
[4]Digital AI Research, BT Group, Ipswich, UK

Abstract

Despite the rapid progress of large language models (LLMs) in code generation, existing evaluations focus on functional correctness or syntactic validity, overlooking how LLMs make critical design choices such as which library or programming language to use. To fill this gap, we perform the first systematic study of LLMs' preferences for libraries and programming languages when generating code, covering eight diverse LLMs. We observe a strong tendency to overuse widely adopted libraries such as `NumPy`; in up to 45% of cases, this usage is not required and deviates from the ground-truth solutions. The LLMs we study also show a significant preference toward `Python` as their default language. For high-performance project initialisation tasks where using `Python` may bring more security and efficiency risks, it remains the dominant choice in 58% of cases, and `Rust` is not used once. These results highlight how LLMs prioritise familiarity and popularity over suitability and task-specific optimality; underscoring the need for targeted fine-tuning, data diversification, and evaluation benchmarks that explicitly measure language and library selection fidelity.

1 Introduction

Large Language Models (LLMs) have recently made rapid progress (Naveed et al., 2024), particularly excelling in code generation (Jiang et al., 2024), with extensive work on evaluating and improving LLM code accuracy, security, and efficiency (Chen et al., 2024a). However, less studied are the *preferences* LLMs exhibit when choosing the libraries and programming languages to use for a coding task. These preferences can have tangible effects on software performance, efficiency, maintainability, and ecosystem diversity — potentially reinforcing dominant technologies and overlooking domain-suitable alternatives. Understanding these tendencies is therefore crucial for both building trustworthy and context-aware code generation systems, and for designing benchmarks that capture not only correctness but also the rationale behind design choices in LLM-generated code.

To address this gap, we present the first systematic study of library and programming language preferences across eight production-grade LLMs. We begin with libraries – vital components of modern software development (Somerville, 2016) – because developers often omit them from prompts (Hao et al., 2024) and may not know which dependencies a task requires (Larios-Vargas et al., 2020), giving LLMs the opportunity to influence library adoption. We then consider the programming language preferences of LLMs. Many end users lack the expertise to judge whether an LLM's choice of language is appropriate (Khurana et al., 2024; Nguyen et al., 2024) and leave the decision entirely to the LLM, especially in end-to-end automatic software generation scenarios (Sarkar and Drosos, 2025).

For both libraries and programming languages, we examine LLM preferences in two key scenarios: writing code for practical tasks from widely studied benchmarks; and generating the initial code for new projects, a crucial phase where LLMs have been found to be particularly helpful (Rasnayaka et al., 2024) and foundational technology choices are made. In addition, we explore whether the technologies that LLMs recommend in natural language (NL) responses match those they actually use when generating code.

Our experiments lead to the following primary observations: **1)** All LLMs we study **heavily favour well established libraries over high-momentum alternatives**. In particular, LLMs favour over-using widely adopted data science libraries even when not required for the task; for example, `NumPy` usage diverges with ground-truth solutions for up to 45% of tasks. **2)** For programming languages, we observe a **significant preference**

Findings of the Association for Computational Linguistics: ACL 2026, pages 331–351
July 2-7, 2026 ©2026 Association for Computational Linguistics

towards using `Python`. To our surprise, even for tasks where high performance and memory safety are critical – and therefore `Python` is considered suboptimal – it remains the most used language in 58% of cases. **3) LLMs do not follow their own recommendations**: the technologies suggested in NL responses are a weak signal of those actually used in generated code. These tendencies likely arise from multiple factors, such as the prevalence of certain libraries in open source repositories, the distribution of programming languages included in training data (Wang et al., 2024), or post-training alignment and fine-tuning (Ziegler et al., 2020).

Such preferences have mixed consequences. On the positive side, defaulting to well-known libraries and languages can speed prototyping and increase interoperability for many users. On the negative side, persistent favouritism can **hurt the open-source community** by marginalizing emerging projects, ultimately triggering **a vicious cycle** with less and less diversity as LLMs evolve with self-generated data (Zhu et al., 2025). Most critically, these biases can present a significant **security threat**, potentially steering users toward suboptimal and insecure implementation choices. Understanding the extent of these preferences is important, not only for assessing the potential risks and broader impacts of LLMs on the software ecosystem, but also for guiding LLM creators in improving model design, training data composition, and alignment strategies to achieve more balanced and context-aware code generation.

Our contributions are as follows.

1. We present the first systematic study of LLM preferences for libraries and programming languages when generating code.
2. We highlight the importance of analysing and quantifying diversity and preference patterns in LLM-generated code, and discuss their potential implications for software development practices and the open-source ecosystem.
3. We release our code, datasets and complete results publicly via our GitHub repository, to encourage further investigation and improvement in this area: `https://github.com/itsluketwist/llm-code-bias`

2 Related Work

Bias and unfairness in LLMs. Bias in the NL outputs of LLMs is well documented: LLMs often inherit and amplify social and representational stereotypes from their training data (Bender et al., 2021; Nadeem et al., 2021; Gallegos et al., 2024), and can preferentially surface newer content in answers (Fang et al., 2025). Social biases also appear in LLM generated code, and several datasets and frameworks now quantify code-based unfairness across demographics (Liu et al., 2023b; Huang et al., 2024; Ling et al., 2025). In contrast, *technological* biases – preferences for the specific technologies used in the code – are less explored. Recent work exposing provider bias (e.g., favouring specific cloud services) motivates our focus on whether analogous biases exist at the library and programming language level (Zhang et al., 2025).

LLM code generation. Code generation in LLMs has been extensively studied (Jiang et al., 2024), including their ability to implement code using external or unseen libraries (Liu et al., 2023a; Zan et al., 2023; Patel et al., 2024). However, evaluation methodology remains an open problem (Paul et al., 2024). Existing approaches typically use datasets of coding tasks, but they are often limited in scope and favour popular programming languages such as `Python` and `Java` (Yadav et al., 2024). These benchmarks are also often compromised via data leakage – where the benchmarks' tasks are included in the LLM training data – giving unfair representations of the LLMs' abilities to complete the tasks (Matton et al., 2024; Zhou et al., 2025). LLM code-generation also raises several ethical issues around code reuse and provenance (German et al., 2009), as they are often trained on large public code corpora (Chen et al., 2021). Together, these issues motivate our investigation into whether LLM coding preferences, and whether they prefer certain technologies due to their exposure to code during training.

LLM recommendations & library selection. Library selection is a core developer skill that requires trade-offs between functionality, performance and maintainability (Larios-Vargas et al., 2020; Tanzil et al., 2024). Increasingly, users turn to LLMs for programming recommendations, with observable declines in public Q&A activity following the release of ChatGPT (del Rio-Chanona et al., 2024; Zhong and Wang, 2024). Furthermore, there is a growing body of work that treats library recommendation as a core capability of LLMs. ToolLLM (Qin et al., 2023) and Gorilla (Patil et al., 2024) show how retrieval-aware or tool-augmented LLMs can recommend and produce API calls re-

liably; while APIGen (Chen et al., 2024b) studies generative approaches for recommending APIs. These prior efforts focus on improving recommendation accuracy; they do not consider how potential biases in LLM recommendations affect developer choices. This gap motivates our study of whether LLM coding recommendations align with the technologies they actually use.

3 Experimental Design

3.1 LLM Selection

We use a diverse set of LLMs in this study, to gain a broad understanding of LLM preferences. LLMs are chosen to vary in number of parameters, availability (open- or closed-source), and intended use case (general or code-specific). Therefore, we chose the following eight LLMs for our study: GPT-4o-mini and GPT-3.5-turbo (OpenAI, 2025), Claude-3.5 Sonnet and Haiku (Anthropic, 2024), Llama-3.2-3B (Meta, 2025), Mistral-7B (Jiang et al., 2023), Qwen-2.5-Coder (Hui et al., 2024) and DeepSeek-LLM (DeepSeek-AI et al., 2024).

To reflect the typical usage of LLMs by developers, which often overlooks the role of parameters (Donato et al., 2025), each LLM is prompted using the default parameter values from its corresponding API. Furthermore, we conduct each LLM interaction in a fresh API session to avoid bias from prompt caching or leakage (Gu et al., 2025); and we do not use a system prompt to ensure that each LLM has its base functionality considered without external influence (Mu et al., 2024).

Full details on LLMs, and how we extract data from their responses, are given in Appendix A.

3.2 Experiment 1: Library Preferences

We focus on a single programming language to enable for a more in-depth analysis. `Python` is chosen due to its vast collection of open-source libraries (PyPI, 2025), its straightforward import syntax, and its popularity in the open-source community (GitHub Staff, 2024). Previous work shows that users often start tasks with NL descriptions without knowing which – or that any – libraries are required (Kuhn and DeLine, 2012); and when prompting LLMs, will initially omit details, iteratively refining their requirements over time (Hao et al., 2024). Therefore, our goal is to measure which external libraries LLMs prefer when prompts do not specify library names.

Benchmark Tasks. We use BigCodeBench (Zhuo et al., 2024) as our primary benchmark. BigCodeBench contains 1,140 `Python` tasks across seven domains (general, computation, visualisation, system, time, network, cryptography) — 813 of which include external libraries in their ground-truth solution. The latest version is released in February 2025, postdating our LLMs' knowledge cut-offs, reducing exposure bias. We use the NL description from each task, and to eliminate bias we filter out any tasks that contain a reference to an external library used in its ground-truth solution. The resulting dataset contains 525 tasks, using 34 distinct libraries in their ground-truth solutions.

We use a BigCodeBench inspired prompt, asking the model to produce self-contained `Python` code along with the directive to use an external library (forcing the use of an external library is a deliberate design choice to focus this experiment on library selection). We generate three independent responses per task to reduce the inherent randomness in LLM outputs (Sallou et al., 2024).

Project initialisation tasks. LLMs have been shown to be particularly helpful in the early stages of software projects (Rasnayaka et al., 2024); this is also a natural decision point for adopting a new library, where the LLM's preferences may influence users' choices. Therefore, to complement benchmark tasks, we also measure library preferences in a more open-ended, project-level setting.

We create five realistic project descriptions inspired by categories on Awesome Python (vinta, 2025), a curated collection of `Python` libraries, to ensure that multiple viable libraries exist for each task. We intentionally use open-ended prompts (allowing the LLM to use one or more libraries), requesting the LLMs "write the initial code" for the following types of projects: *database*, *deep learning*, *distributed computing*, *web scraper*, and *web server*. For each project description, we generate 100 responses per LLM and analyse the distribution of libraries used to mitigate the variance of the evaluation (Madaan et al., 2024).

Complete dataset details and full prompts for Experiment 1 are given in Appendix B.1.

3.3 Experiment 2: Language Preferences

Next, we investigate LLM preferences for programming languages when the prompts do not specify a target language. Although this is less common among professional developers – mostly seen

from novice programmers who struggle to write detailed prompts (Nguyen et al., 2024), and "vibe-coders" (Smith, 2025), that leave more of the technical decisions to the LLM – it will allow us to further study the inherent biases LLMs exhibit when allowed to make coding decisions.

Benchmark Tasks. We use language-agnostic datasets, chosen to (i) contain NL-only descriptions and (ii) reduce single-language bias or leakage by having published solutions in multiple languages. We use six widely-adopted datasets, spanning three categories: *basic* (Multi-HumanEval, MBXP), *real-world* (AixBench, CoNaLa), and *coding challenge* (APPS, CodeContests). Any task whose description explicitly mentions a programming language from the TIOBE Index (Jansen, 2025) (top 50) is removed. Preliminary experiments showed that a sample of 200 tasks gave a good representation of the results for the larger datasets, therefore if a dataset was over 200 tasks, it would be sampled, allowing results to be fairly aggregated.

We use a lightweight prompt template compatible with each dataset, asking for a code solution and explanation; we request an explanation because we found this to be the most reliable way to encourage the LLM to respond using `Markdown` notation without biasing the results. We generate three independent responses per task to reduce the inherent randomness in LLM outputs (Sallou et al., 2024).

Project Initialisation Tasks. We also want to investigate LLM preferences for programming languages when asked to write initial project code. Our initial results imply that LLMs have a general preference towards using `Python`. Therefore, to challenge this default, we choose project descriptions with non-functional requirements such that `Python` is regarded as a suboptimal choice. Although `Python` offers user-friendly syntax and rapid prototyping, it is less suitable for high-performance workloads due to its interpretive nature and inefficient concurrency implementation (Gavrilova, 2023). For computationally intensive tasks, compiled languages like `C`, `C++`, and `Rust` are generally preferred; they provide greater memory control, lower execution overhead, and true parallel processing (Costanzo et al., 2021).

We prompt the LLM to "write the initial code" for the following project descriptions, each representative of a domain where `Python` is suboptimal: *concurrent web server*, *cross-platform graphical user interface*, *low-latency trading platform*, *parallel task processing library*, and a *system-level application*. The descriptions are intentionally open-ended, allowing the use of multiple languages if required. For each project description, we generate 100 responses per LLM and analyse the distribution of programming languages used to mitigate the variance of the evaluation (Madaan et al., 2024).

Complete dataset details and full prompts for Experiment 2 are given in Appendix B.2.

3.4 Experiment 3: Recommendation Consistency

Finally, we want to examine whether the LLMs' NL recommendations for libraries and programming languages match what they actually use when generating code. This directly tests whether the LLM is self-consistent with its internal knowledge of what technology to use for what task, or if it has bias specific to code generation. To obtain the NL recommendations, we prompt the LLMs to "list, in order" the best `Python` library or programming language to use for each project initialisation task from Experiments 1 and 2.

We generate three responses per task and take the arithmetic mean of the ranks to obtain the final ranking per LLM. We then derive empirical usage rankings from the results of Experiments 1 and 2. For each task, we rank the libraries or programming languages descending by their observed usage frequencies. We measure correspondence between the two rankings with Kendall's τ_b coefficient, a non-parametric (no assumptions about the underlying distribution of the data) rank correlation suitable for ordinal data (natural order but not necessarily equal intervals) that is robust to ties (Kendall and Gibbons, 1990).

Complete prompt selection and calculation details for Experiment 3 are given in Appendix B.3.

4 Results

Here we present the results. *Note* that it is expected that the percentages in the results do not add up to 100% because LLMs were prompted multiple times per task and could respond with different libraries or languages, and sometimes without code.

4.1 Experiment 1: Library Preferences

This experiment aims to assess LLMs' preferences for `Python` libraries during code generation.

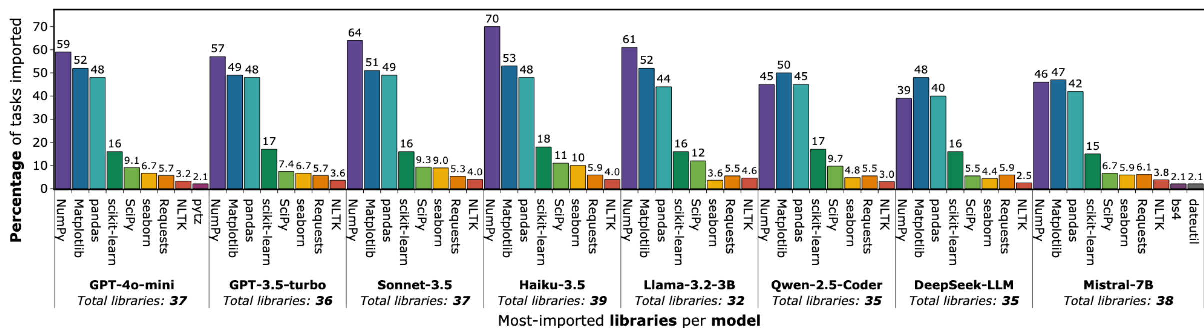

Figure 1: ***Experiment 1: Library Preferences for Benchmark Tasks.*** Libraries used by LLMs when responding to BigCodeBench tasks. For each LLM, we give the most-used libraries with the percentage of tasks that had a response importing them, and the total unique libraries used. Other libraries are imported for *less than 2%* of tasks.

Benchmark tasks. Figure 1 shows the libraries used by LLMs when writing code for BigCodeBench tasks. All LLMs produce very similar distributions: the top three libraries are identical (`NumPy`, `pandas`, `Matplotlib`), with a clear gap between them and the next most-used library. Although a portion of BigCodeBench focuses on computation and visualisation – domains where these libraries are naturally useful – we observe that the LLMs frequently use them when the ground truth does not. For example, `NumPy` is used in responses to 192 of the 305 tasks where it is not part of the ground-truth solution, indicating a tendency to use it even when not required for the given task. *Further ground-truth analysis is provided in Appendix C.1.*

`Python`'s ecosystem is huge: in July 2025 over 7,000 libraries surpassed 100,000 downloads (hugovk, 2025) (still only a fraction of those available). However, each LLM in our study used surprisingly few distinct libraries (32-39) across hundreds of varied tasks. Although not all of the available libraries are suitable, using less than 40 points to a lack of diversity in the libraries that LLMs choose when not given a specific directive.

Prompt sensitivity We use a prompt that explicitly requests an external library to focus our study on library selection. We perform a small ablation study to assess LLM sensitivity to this choice, repeating the experiment with prompt variations that express different levels of strictness around external library usage. The results show that our main findings are robust, with similar library preferences observed across all prompt variants. *Alternative prompts and full results for the ablation study can be seen in Appendix C.2.*

Project initialisation tasks. Table 1 shows the libraries imported when LLMs generate the initial code for five project descriptions. Open-source LLMs show minimal diversity, repeatedly using the same core libraries across runs, whereas closed-source LLMs use a wider variety of libraries for core functionality. We observe a correlation between the library diversity and the LLMs' default temperature settings (our open-source LLMs use a default of 0.6–0.7; our closed-source default to 1.0), suggesting that open-source LLMs give developers a more consistent experience. *We explore the effects of temperature further in Appendix C.3.*

We also observe repeated instances of apparent overuse. For example, Qwen-2.5-Coder imports `TensorFlow` for the *database* project, and multiple LLMs import `NumPy` or `pandas` for *distributed computing*. These patterns mirror the results for the benchmark tasks, indicating similar preferences in both constrained and open-ended tasks. Such tendencies may be related to the recent growth in `Python` that is attributed to its increasing role in AI and data science-based code (GitHub Staff, 2024).

Cases analysis. Our results indicate that LLMs show a preference for established libraries. To understand the extent of this, we perform a case analysis on libraries used in different domains. We use GitHub star growth over time as a lightweight proxy for community momentum and look for alternative libraries that show signs of higher potential but have lower usage rates; Figure 2 shows this for competing libraries in our sample. We acknowledge that GitHub stars are an imperfect measure of ecosystem momentum, and complementary indicators – such as downloads, dependents, or contribution velocity – could provide a more nuanced view of under-used libraries. We use star growth because it is widely available, interpretable, and con-

Table 1: ***Experiment 1: Library Preferences for Project Initialisation Tasks.*** Libraries used by LLMs when writing initial project code. The libraries (*l*) used by each LLM are given, along with the percentage (*p*) of responses that used that library. Libraries considered to provide core functionality are marked with a * and are listed first.

Library Task	GPT-4o-mini		GPT-3.5-turbo		Sonnet-3.5		Haiku-3.5		Llama-3.2-3B		Qwen-2.5-Coder		DeepSeek-LLM		Mistral-7B	
	l	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*
Database	SQLAlchemy*	100%	SQLAlchemy*	100%	SQLAlchemy*	100%	SQLAlchemy*	100%	SQLAlchemy*	100%	SQLAlchemy*	99%	SQLAlchemy*	100%	SQLAlchemy*	100%
	models	11%	-	-	-	-	Pydantic	3%	models	96%	TensorFlow	1%	-	-	-	-
	database	3%	-	-	-	-	dotenv	3%	db	1%	-	-	-	-	-	-
	-	-	-	-	-	-	*Total used*	*5*	-	-	-	-	-	-	-	-
Deep learning	TensorFlow*	97%	TensorFlow*	85%	PyTorch*	100%	scikit-learn*	82%	TensorFlow*	100%	TensorFlow*	100%	TensorFlow*	100%	Keras*	100%
	scikit-learn*	4%	PyTorch*	7%	TorchVision*	93%	TensorFlow*	75%	scikit-learn*	91%	-	-	-	-	scikit-learn*	44%
	PyTorch*	3%	Keras*	6%	scikit-learn*	5%	PyTorch*	25%	NumPy	100%	-	-	-	-	NumPy	100%
	TorchVision*	2%	scikit-learn*	2%	Matplotlib	96%	TorchVision*	14%	Matplotlib	91%	-	-	-	-	-	-
	NumPy	76%	TorchVision*	1%	NumPy	11%	NumPy	86%	-	-	-	-	-	-	-	-
	Matplotlib	64%	NumPy	47%	-	-	Matplotlib	26%	-	-	-	-	-	-	-	-
Distributed computing	Dask*	73%	MPI4py*	33%	Dask*	24%	Dask*	89%	Dask*	100%	Ray*	100%	-	-	Dask*	100%
	MPI4py*	5%	Dask*	31%	Ray*	20%	Ray*	4%	NumPy	55%	-	-	-	-	NumPy	100%
	Ray*	2%	NumPy	6%	Celery*	1%	Celery*	2%	-	-	-	-	-	-	-	-
	NumPy	28%	Joblib	1%	Redis	49%	NumPy	71%	-	-	-	-	-	-	-	-
	Total used	*8*	-	-	*Total used*	*8*	*Total used*	*6*	-	-	-	-	-	-	-	-
Web scraper	BS4*	100%	BS4*	100%	BS4*	100%	BS4*	100%	BS4*	100%	BS4*	100%	BS4*	100%	BS4*	100%
	Requests*	100%	Requests*	100%	Requests*	100%	Requests*	100%	Requests*	100%	Requests*	100%	Requests*	100%	Requests*	100%
	pandas	27%	pandas	13%	pandas	100%	pandas	100%	pandas	100%	pandas	100%	-	-	pandas	100%
	-	-	-	-	-	-	*Total used*	*6*	-	-	-	-	-	-	-	-
Web server	Flask*	98%	Flask*	100%	FastAPI*	48%	Flask*	69%	Flask*	100%	Flask*	100%	Flask*	100%	Flask*	100%
	FastAPI*	2%	-	-	Flask*	39%	FastAPI*	23%	Flask-SQLA	100%	-	-	-	-	-	-
	Flask-REST	19%	-	-	Pydantic	48%	Flask-Cors	52%	-	-	-	-	-	-	-	-
	Flask-Cors	15%	-	-	Uvicorn	48%	Flask-SQLA	31%	-	-	-	-	-	-	-	-
	Total used	*7*	-	-	Flask-Cors	14%	*Total used*	*12*	-	-	-	-	-	-	-	-

Note some library names have been shortened in the table: Flask-REST is Flask-RESTful, Flask-SQLA is Flask-SQLAlchemy, BS4 is BeautifulSoup.

sistently reported across the libraries under study, allowing for uniform comparison. We find high-momentum alternatives in four domains, showing a pattern of LLMs preferring well-established libraries and underusing newer alternatives.

1. *Computation tasks:* `pandas` (2010) appears in the 58% of tasks. `Polars` (2020) is a newer library with similar functionality and twice as fast GitHub star growth, but is not used at all.
2. *Visualisation tasks:* `Matplotlib` (2011) and `seaborn` (2012) are heavily used, for 57% and 17% of tasks. `Plotly` (2013) has additional functionality and higher momentum, but is used for only *one* problem and by *three* LLMs.
3. *Web server project:* `Flask` (2010) is used in *88%* of responses. `FastAPI` (2018) has shown substantially faster recent adoption, yet was used only in 9% of responses, by only *three* LLMs.
4. *Distributed computing project:* `Dask` (2015) is the most common choice, used in 52% of the responses. `Ray` (2016) and `Celery` (2009) both have similar functionality along with more stars and faster growth, but they were used minimally (in 16% and 0.4% of responses, respectively).

Experiment 1 Summary: LLMs show a strong tendency to use older, well-established libraries, often overlooking newer alternatives with high community momentum; as shown in our four case studies, where usage rate differences range from **36%** to **79%**. We also observe the overuse of data science libraries: for up to **45%** of the benchmark tasks where `NumPy` is imported, its usage differs from the ground-truth solutions.

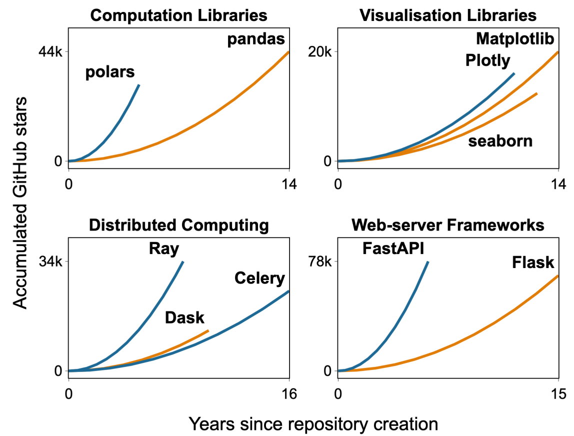

Figure 2: ***Case Analysis.*** GitHub growth statistics for libraries studied in case analysis. For each library, the accumulated GitHub stars are plotted over its lifetime (data from GitHub, December 2024). The top/bottom two graphs show case analysis for benchmark/project initialisation tasks. **Orange** curves indicate libraries that are heavily used in this study, **Blue** curves indicate libraries with low usage rates.

4.2 Experiment 2: Language Preferences

This experiment investigates LLMs' programming language preferences during code generation.

Benchmark tasks. Table 2 shows the programming languages used by LLMs when generating code for benchmark tasks. Across nearly all datasets we observe a strong preference for `Python`. Except for AixBench, LLMs used `Python` for at least 93.5% of tasks, the highest percentage of responses in a non-`Python` language was considerably lower at 19.5%. In the *coding challenge* benchmarks (CodeContests and APPS) chosen for their

Table 2: *Experiment 2: Language Preferences for Benchmark Tasks.* Programming languages used by LLMs when writing code for benchmark tasks. *Dataset* shows the benchmark name, as well as the programming languages for which the ground truth solutions are provided. For each LLM and dataset, the preferred languages (*l*) are given, along with the percentage (*p*) of dataset problems that have a response using that language, and the total count of different languages used (when necessary). The most-used language in each case is in bold.

Dataset	GPT-4o-mini		GPT-3.5-turbo		Sonnet-3.5		Haiku-3.5		Llama-3.2-3B		Qwen-2.5-Coder		DeepSeek-LLM		Mistral-7B	
	l	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*	*l*	*p*
Multi-HumanEval	**Python**	**100.0%**	**Python**	**100.0%**	**Python**	**98.8%**	**Python**	**99.4%**	**Python**	**100.0%**	**Python**	**100.0%**	**Python**	**100.0%**	**Python**	**100.0%**
(10 languages)*	-	-	-	-	JavaScript	1.2%	*Total used*	*4*	-	-	-	-	-	-	*Total used*	*3*
MBXP	**Python**	**100.0%**	**Python**	**100.0%**	**Python**	**97.0%**	**Python**	**99.5%**	**Python**	**100.0%**	**Python**	**100.0%**	**Python**	**98.5%**	**Python**	**99.0%**
(10 languages)*	-	-	JavaScript	1.0%	JavaScript	7.5%	Java	2.5%	-	-	-	-	-	-	JavaScript	0.5%
	-	-	-	-	Java	1.0%	C++	2.0%	-	-	-	-	-	-	-	-
	-	-	-	-	C	0.5%	JavaScript	2.0%	-	-	-	-	-	-	-	-
AixBench	**Python**	**75.2%**	**Python**	**64.3%**	**Java**	**53.5%**	**Java**	**59.7%**	**Python**	**75.2%**	**Python**	**68.2%**	**Python**	**78.3%**	**Python**	**62.0%**
(Java)	Java	27.1%	Java	38.8%	Python	38.8%	Python	55.0%	Java	18.6%	Java	26.4%	Java	18.6%	Java	24.8%
	JavaScript	10.9%	JavaScript	10.1%	JavaScript	18.6%	JavaScript	20.2%	C#	3.1%	JavaScript	3.1%	JavaScript	2.3%	JavaScript	14.0%
	Total used	*7*	*Total used*	*7*	*Total used*	*9*	*Total used*	*12*	*Total used*	*8*	*Total used*	*5*	C#	0.8%	*Total used*	*9*
CoNaLa	**Python**	**99.0%**	**Python**	**98.5%**	**Python**	**98.0%**	**Python**	**99.0%**	**Python**	**99.0%**	**Python**	**98.0%**	**Python**	**99.0%**	**Python**	**97.5%**
(Python)	JavaScript	6.0%	JavaScript	2.5%	JavaScript	12.5%	JavaScript	15.5%	JavaScript	1.0%	JavaScript	3.0%	JavaScript	1.5%	JavaScript	4.0%
	Java	4.5%	Java	1.5%	Java	3.5%	Java	13.0%	-	-	Java	2.0%	Java	1.0%	Java	2.5%
	Total used	*7*	C++	0.5%	*Total used*	*9*	*Total used*	*11*	-	-	*Total used*	*5*	C++	0.5%	*Total used*	*5*
APPS	**Python**	**99.5%**	**Python**	**99.5%**	**Python**	**93.5%**	**Python**	**93.5%**	**Python**	**98.0%**	**Python**	**98.0%**	**Python**	**98.5%**	**Python**	**98.5%**
(Python)	JavaScript	1.0%	JavaScript	1.5%	C++	10.0%	C++	7.5%	Ruby	0.5%	C++	1.5%	Ruby	0.5%	Ruby	0.5%
	Ruby	0.5%	Ruby	0.5%	JavaScript	5.5%	JavaScript	4.5%	Java	0.5%	Ruby	0.5%	-	-	Java	0.5%
	R	0.5%	Java	0.5%	*Total used*	*5*	*Total used*	*6*	C	0.5%	R	0.5%	-	-	R	0.5%
CodeContests	**Python**	**100.0%**	**Python**	**98.5%**	**Python**	**96.5%**	**Python**	**94.0%**	**Python**	**97.5%**	**Python**	**97.0%**	**Python**	**96.5%**	**Python**	**97.5%**
(C++, Java, Python)	-	-	C++	2.0%	C++	19.5%	C++	15.0%	C++	2.0%	C++	6.0%	-	-	C++	2.0%
	-	-	-	-	Java	0.5%	-	-	JavaScript	0.5%	-	-	-	-	JavaScript	0.5%
All datasets	**Python**	**96.8%**	**Python**	**95.1%**	**Python**	**89.8%**	**Python**	**92.0%**	**Python**	**96.1%**	**Python**	**94.9%**	**Python**	**96.1%**	**Python**	**94.1%**
	Java	4.0%	Java	5.0%	Java	7.3%	Java	10.3%	Java	2.3%	Java	3.5%	Java	2.4%	Java	3.5%
	JavaScript	2.6%	JavaScript	2.1%	JavaScript	7.1%	C++	7.8%	C++	0.6%	C++	1.7%	JavaScript	0.6%	JavaScript	2.6%
	C#	0.5%	C++	0.6%	C++	6.2%	JavaScript	6.6%	JavaScript	0.5%	JavaScript	0.9%	C#	0.1%	C#	0.6%
	Total used	*11*	*Total used*	*8*	*Total used*	*13*	*Total used*	*14*	*Total used*	*9*	*Total used*	*8*	*Total used*	*6*	*Total used*	*13*

* Multi-HumanEval and MBXP benchmarks contain solutions in the following languages: *C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, TypeScript*.

multilingual solution ecosystems, the LLMs default disproportionately to `Python`, suggesting that the preference is not merely an artifact of single-language contamination. The *real world* datasets (AixBench and CoNaLa) show greater language diversity than the synthetic benchmarks: up to twelve different languages appear in responses versus at most five for other datasets. Still, the overall set of languages used remains consistently narrow across LLMs. *Further analysis of the similarities is given in Appendix C.4.*

Project initialisation tasks. Table 3 shows the languages used when LLMs wrote the initial code for various project descriptions where highly-performant languages – such as `C`, `C++` and `Rust` (Costanzo et al., 2021) – are considered optimal. Even in these settings, the LLMs in our study show a clear preference for using `Python`: it is the most used language in 23/40 instances, despite being explicitly suboptimal. Some LLMs are particularly prone to this behaviour (Haiku-3.5, Llama-3.2-3B and DeepSeek-LLM favoured `Python` for 4/5 projects). `JavaScript` is the next most used, but still only the preferred choice in 8/40 instances.

Python analysis. Although the preference to `Python` may be unsurprising, it is still valuable to quantify and understand, for both LLM users and creators. Bias in an LLM is usually attributed to the training / pre-training stages, where the knowledge of the LLM begins to mirror the underlying data corpora (Guo et al., 2024). Therefore, we might expect the `Python` bias to arise from an abundance of `Python` in the training data. The precise code corpora used to train LLMs typically remain undisclosed (Bommasani et al., 2023), but there is strong evidence that they are heavily based on large-scale GitHub archives (Majdinasab et al., 2025).

Although `Python` has seen a sudden surge in popularity, it has not historically been the most used programming language; only in 2024 did it record the highest number of GitHub contributors (GitHub Staff, 2024), and lead the TIOBE Index (Jansen, 2025). This would suggest that evenly sampling GitHub would yield a more balanced language distribution than the `Python` preferences we observe. Therefore, we posit that it is likely that LLM creators are favouring `Python` when constructing their training corpora; or encouraging `Python` in the post-training / fine-tuning stages.

EXPERIMENT 2 SUMMARY: All LLMs demonstrate a strong preference for `Python` as their default programming language. For benchmark tasks, `Python` accounts for **90–97%** of generated solutions. Even for project initialisation tasks where `Python` is arguably suboptimal, it remains the most-used language in **58%** of cases.

Table 3: ***Experiment 2: Language Preferences for Project Initialisation Tasks.*** Programming languages used by LLMs when writing project code in scenarios where `Python` is suboptimal. The languages (l) used by each LLM are given, along with the percentage (p) of responses that used the language. The most-used language is in bold.

Language Task	GPT-4o-mini		GPT-3.5-turbo		Sonnet-3.5		Haiku-3.5		Llama-3.2-3B		Qwen-2.5-Coder		DeepSeek-LLM		Mistral-7B	
	l	p	l	p	l	p	l	p	l	p	l	p	l	p	l	p
Concurrenct web server	**JavaScript**	**73%**	**JavaScript**	**99%**	**JavaScript**	**100%**	**Python**	**100%**	**Go**	**100%**	**Python**	**100%**	**JavaScript**	**94%**	**JavaScript**	**100%**
	Python	28%	Python	1%	-	-	-	-	-	-	-	-	Python	6%	-	-
	Go	2%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cross-platform graphical user interface	**JavaScript**	**72%**	**JavaScript**	**64%**	**Python**	**99%**	**Python**	**81%**	**Python**	**100%**	**JavaScript**	**100%**	**Python**	**100%**	**Dart**	**81%**
	Dart	38%	C++	29%	Dart	2%	JavaScript	19%	-	-	Dart	100%	-	-	JavaScript	48%
	Python	14%	Python	11%	-	-	TypeScript	2%	-	-	-	-	-	-	-	-
	C++	3%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Low-latency trading platform	**Python**	**100%**	**Python**	**96%**	**Java**	**63%**	**Python**	**100%**	**Python**	**100%**	**Python**	**100%**	**Python**	**100%**	**Python**	**53%**
	-	-	Java	2%	C++	30%	-	-	-	-	-	-	-	-	none	47%
	-	-	JavaScript	2%	Python	7%	-	-	-	-	-	-	-	-	-	-
	-	-	C++	1%	-	-	-	-	-	-	-	-	-	-	-	-
Parallel task processing library	**Python**	**99%**	**Python**	**80%**	**Python**	**96%**	**Python**	**100%**	**Python**	**100%**	**C++**	**100%**	**Python**	**100%**	**Python**	**100%**
	C++	1%	C++	11%	C++	4%	-	-	-	-	-	-	-	-	-	-
	-	-	Java	9%	-	-	-	-	-	-	-	-	-	-	-	-
System-level application	**Python**	**81%**	**C**	**63%**	**C**	**50%**	**Python**	**100%**	**Python**	**100%**	**C**	**100%**	**C++**	**100%**	**C++**	**51%**
	C	23%	Python	37%	Python	50%	-	-	-	-	-	-	-	-	C	49%

Table 4: ***Experiment 3: Recommendation Consistency for Project Initialisation Tasks.*** Kendall's τ correlation between LLMs' NL recommendations and their actual coding choices. Only statistically significant correlations are given (p-values < 0.05) - there were none for language tasks. Values near 1.0 / -1.0 indicate strong agreement / disagreement.

Library Task	GPT-4o-mini	GPT-3.5-turbo	Sonnet-3.5	Haiku-3.5	Llama-3.2-3B	Qwen-2.5-Coder	DeepSeek-LLM	Mistral-7B
Database	-	-	-	-	-	-	-	-
Deep learning	-	0.57	-	-	-	-	-	-
Distributed computing	0.41	-	0.60	0.60	-	-	-	-
Web scraper	0.49	-	0.42	-	-	0.55	0.60	0.73
Web server	-	-	-	-	-	-	-	-

4.3 Experiment 3: Recommendation Consistency

This experiment assesses whether the libraries or programming languages LLMs recommend for a task are consistent with what they actually use. Table 4 shows the rank correlation results; recommendations are provided in Appendix C.5. Overall, we observe very low consistency between the recommended and used technologies. For library tasks, there is inconsistent moderate agreement ($\tau \approx$ 0.5-0.7): only *distributed computing* and *web scraper* have significant correlations for multiple LLMs, likely because those domains have clearer, community-standard choices. For language tasks, there is no significant correlation; direct comparison shows rare agreement, with the top recommended language matching the most used language in only 7/40 instances. As expected from the task selection, the LLMs did not highly recommend `Python` (sometimes not suggesting it at all), but still defaulted to using it in most instances.

This inconsistency suggests that LLMs lack a universal knowledge representation of "which technology best fits these requirements", which they can apply to both code and NL answers. This does not imply a specific internal failure mode, but it does indicate that an LLMs' NL answers are a weak signal for what to expect from its generated code. *In Appendix C.6, we explore prompt engineering strategies as mitigation for this inconsistency.*

EXPERIMENT 3 SUMMARY: There is low consistency between the libraries and programming languages LLMs recommend and those that they actually adopt, with the majority of correlations not statistically significant. For project initialisation tasks, LLMs contradict their own language recommendations **83%** of the time.

5 Discussion

Our experiments indicate systematic tendencies in LLMs' choices of libraries and programming languages. Here, we balance the practical upsides and downsides of these preferences.

Benefits. LLMs' strong tendency to generate code in `Python` and to rely on widely adopted libraries offers several advantages. First, `Python` 's concise syntax and extensive ecosystem improves readability and lowering cognitive load for both novice and expert users. Second, the reliance on well-established, actively maintained libraries often leads to more reliable and executable outputs. This behaviour also promotes code standardisation and compatibility across development environments,

which can accelerate prototyping, improve reproducibility, and facilitate educational use.

Risks. However, this homogenising tendency also introduces several risks. It may lead to language and library bias, where LLMs systematically overlook alternative languages or emerging libraries that offer better performance, safety, or domain suitability. Users, especially non-experts, may over-trust the model's default choices, reproducing outdated or suboptimal dependencies without critical evaluation. Such bias may distort technology adoption, creating a feedback loop that amplifies the dominance of a few ecosystems represented in training data.

Moreover, excessive reliance on popular libraries can obscure opportunities for innovation and diversity in software design, challenging fairness and inclusivity in software tooling, and reducing discoverability and slow adoption of newer, potentially better tools. Developers working in underrepresented languages or regions may find LLMs less useful or accurate for their contexts.

Practical Implications. As an increasing share of code is generated with the help of LLMs, the broader software ecosystem is likely to reflect the coding patterns that LLMs produce — including their preferred libraries and languages. At scale, even small systematic preferences can compound, shaping dependency graphs, influencing educational materials, and steering future tooling and benchmarks.

Understanding and measuring these preferences is therefore important to ensure that LLMs makes good technological choices in practice, rather than systematically reinforcing conservative or suboptimal defaults as LLM-generated code becomes more widespread.

The recent rise of coding agents will also shape these dynamics. As they generate code in real-world repositories, their choices influence ecosystem trends; early evidence suggests more diverse library usage and disciplined dependency addition, potentially mitigating these biases for established repositories (Twist and Zhang, 2026).

Research challenges. We call for future work on understanding, measuring, and mitigating language and library biases in LLM-based code generation, particularly their potential safety implications. This includes studies that examine how training data distributions shape model preferences, alongside the development of benchmarks and metrics that capture language diversity, dependency selection, and adaptability beyond correctness alone.

Beyond measurement, there is a need to design diversity-aware and evolution-resilient LLMs that maintain proficiency in established languages like `Python` while supporting innovation and inclusivity across the broader software landscape. An important complementary direction is the study of other coding preferences – such as programming paradigms, architectures, data structures, and typing styles – to determine whether similar biases arise and whether mitigation strategies generalise.

6 Conclusion

This paper provides the first empirical study on LLM preferences for libraries and programming languages when generating code. We observe that LLMs consistently favour well-established libraries over high-momentum alternatives; and exhibit a strong preference towards `Python`, even for tasks where it can be seen as suboptimal. Understanding whether these preferences reflect systematic biases – and when they are helpful versus harmful – is essential to ensure that future LLMs enrich, rather than narrow, the software ecosystem.

Limitations & Threats to Validity

Here we discuss potential limitations to our study due to the generalisability to other LLMs, and potential threats to internal and external validity.

Generalisability. We deliberately use a wide range of production-grade LLMs (open / closed, varying amounts of parameters, and different providers) so that our results reflect common behaviour rather than the quirks of a single LLM. We observe similar preferences for all LLMs in our study (discussed in further detail in Appendix C.4), the likely root cause of which is that many LLMs are trained or fine-tuned on large GitHub code corpora and `Python`-centric benchmarks. Although future (more advanced) LLMs may not exhibit similar tendencies, we believe this is unlikely unless training and evaluation practices change.

Internal validity. The threats to internal validity lie in our automatic data extraction and the robustness of our study design. To alleviate the first threat, we have both unit tested the code responsible and manually verified the extraction process on 100

random samples from language and library experiments. We took several steps to reduce the second threat and increase confidence in the stability of our results. Our prompt templates draw on existing studies that generate code with LLMs (Zhuo et al., 2024) and examine how users interact with LLMs (Nguyen et al., 2024); full prompts are reported in Appendix B. We apply these prompts across multiple task types and domains rather than relying on a single benchmark, which helps ensure that the observed preferences are not tied to a specific task formulation, and we generate multiple responses per prompt to reduce randomness. We also perform an ablation study (Appendix C.2) to test sensitivity to prompt style, observing similar patterns regardless of the exact language used.

External validity. The threats to external validity lie in datasets and LLMs used, and our analysis of underused libraries. We mitigate the threat of dataset contamination by carefully selecting benchmark datasets to minimise it, opting for datasets released after the LLMs' knowledge cut-off, or datasets with a variety of programming languages to reduce the possibility that the observed preference is merely caused by contamination. Another threat is the nondeterministic nature of LLMs and their opaque updates (Sallou et al., 2024), introducing variability; we reduce this threat by repeating the experiments multiple times and specifying the exact LLM version to use. Finally, our case analysis of underused libraries uses GitHub stars as a simple, comparable signal; this inevitably leaves out richer ecosystem cues (downloads, dependents, activity) that future work could use to present a more nuanced view of underused libraries.

Acknowledgements

This work is supported by the UKRI Centre for Doctoral Training in Safe and Trusted AI, the EPSRC iCASE Award (ref EP/Y528572/1), the ITEA grants GreenCode (project number 23016), and GENIUS (project number 23026).

References

Mehmet Akhoroz and Caglar Yildirim. 2025. Conversational AI as a Coding Assistant: Understanding Programmers' Interactions with and Expectations from Large Language Models for Coding. *Preprint*, arXiv:2503.16508.

Andrew Peng, Michael Wu, John Allard, Logan Kilpatrick, and Steven Heidel. 2023. GPT-3.5 Turbo fine-tuning and API updates. https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/.

Anthropic. 2024. Claude 3 Model Card.

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, and 6 others. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations, {ICLR} 2023, Kigali, Rwanda, May 1-5, 2023*. arXiv.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *Preprint*, arXiv:2108.07732.

Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, Virtual Event Canada. ACM.

Rishi Bommasani, Kevin Klyman, Shayne Longpre, Sayash Kapoor, Nestor Maslej, Betty Xiong, Daniel Zhang, and Percy Liang. 2023. The Foundation Model Transparency Index. *Preprint*, arXiv:2310.12941.

William Bugden and Ayman Alahmar. 2022. The Safety and Performance of Prominent Programming Languages. *International Journal of Software Engineering and Knowledge Engineering*, 32(05):713–744.

Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. 2024a. A Survey on Evaluating Large Language Models in Code Generation Tasks. *Journal of computer science and technology*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating Large Language Models Trained on Code. *Preprint*, arXiv:2107.03374.

Yujia Chen, Cuiyun Gao, Muyijie Zhu, Qing Liao, Yong Wang, and Guoai Xu. 2024b. APIGen: Generative API Method Recommendation. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 171–182. IEEE Computer Society.

Jishnu Ray Chowdhury and Cornelia Caragea. 2025. Zero-Shot Verification-guided Chain of Thoughts. *Preprint*, arXiv:2501.13122.

Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti. 2021. Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10.

DeepMind. 2024. Competitive programming with AlphaCode. https://deepmind.google/discover/blog/competitive-programming-with-alphacode/.

DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, and 68 others. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. *Preprint*, arXiv:2401.02954.

R. Maria del Rio-Chanona, Nadzeya Laurentsyeva, and Johannes Wachs. 2024. Large language models reduce public knowledge sharing on online Q&A platforms. *PNAS Nexus*, 3(9).

Benedetta Donato, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2025. Studying How Configurations Impact Code Generation in LLMs: The Case of ChatGPT. In *The Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension*. arXiv.

Hanpei Fang, Sijie Tao, Nuo Chen, Kai-Xin Chang, and Tetsuya Sakai. 2025. Do Large Language Models Favor Recent Content? A Study on Recency Bias in LLM-Based Reranking. *Preprint*, arXiv:2509.11353.

Carlo A. Furia, Richard Torkar, and Robert Feldt. 2024. Towards Causal Analysis of Empirical Software Engineering Data: The Impact of Programming Languages on Coding Competitions. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–35.

Isabel O. Gallegos, Ryan A. Rossi, Joe Barrow, Md Mehrab Tanjim, Sungchul Kim, Franck Dernoncourt, Tong Yu, Ruiyi Zhang, and Nesreen K. Ahmed. 2024. Bias and Fairness in Large Language Models: A Survey. *Computational Linguistics*, 50(3):1097–1179.

Yulia Gavrilova. 2023. Pros and Cons of Python. https://serokell.io/blog/python-pros-and-cons.

Daniel M. German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2009. Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90.

GitHub Staff. 2024. Octoverse: AI leads Python to top language as the number of global developers surges.

Sam Gross. 2023. PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org. https://peps.python.org/pep-0703/.

Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. 2025. Auditing prompt caching in language model APIs. In *Forty-second International Conference on Machine Learning*.

Yufei Guo, Muzhe Guo, Juntao Su, Zhou Yang, Mengqiu Zhu, Hongfei Li, Mengyang Qiu, and Shuo Shuo Liu. 2024. Bias in Large Language Models: Origin, Evaluation, and Mitigation. *Preprint*, arXiv:2411.10915.

Huizi Hao, Kazi Amit Hasan, Hong Qin, Marcos Macedo, Yuan Tian, Steven H. H. Ding, and Ahmed E. Hassan. 2024. An Empirical Study on Developers Shared Conversations with ChatGPT in GitHub Pull Requests and Issues. *Empirical Software Engineering*, 29(6):150.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. *Preprint*, arXiv:2206.13179.

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 *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, Virutal*. arXiv.

Dong Huang, Jie M. Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. 2024. Bias Testing and Mitigation in LLM-based Code Generation. *ACM Transactions on Software Engineering and Methodology*.

hugovk. 2025. Top PyPI Packages. https://hugovk.github.io/top-pypi-packages/.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-Coder Technical Report. *Preprint*, arXiv:2409.12186.

Paul Jansen. 2025. TIOBE Index.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *Preprint*, arXiv:2310.06825.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology*.

Dawid Karczewski. 2021. Python vs C++: Technology Comparison. https://www.ideamotive.co/blog/python-vs-cpp-technology-comparison.

Maurice G. Kendall and Jean Dickinson Gibbons. 1990. *Rank Correlation Methods*, 5th ed edition. Oxford University Press, New York, NY.

Anjali Khurana, Hari Subramonyam, and Parmit K. Chilana. 2024. Why and When LLM-Based Assistants Can Go Wrong: Investigating the Effectiveness of Prompt-Based Interactions for Software Help-Seeking. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*, pages 288–303.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pages 22199–22213, Red Hook, NY, USA. Curran Associates Inc.

Adrian Kuhn and Robert DeLine. 2012. On designing better tools for learning APIs . In *2012 4th International Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE 2012)*, pages 27–30, Los Alamitos, CA, USA. IEEE Computer Society.

Enrique Larios-Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 245–256.

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, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Lin Ling, Fazle Rabbi, Song Wang, and Jinqiu Yang. 2025. Bias Unveiled: Investigating Social Bias in LLM-Generated Code. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(26):27491–27499.

Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023a. CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 434–445, Luxembourg, Luxembourg. IEEE.

Yan Liu, Xiaokang Chen, Yan Gao, Zhe Su, Fengji Zhang, Daoguang Zan, Jian-Guang Lou, Pin-Yu Chen, and Tsung-Yi Ho. 2023b. Uncovering and Quantifying Social Biases in Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. arXiv.

Lovish Madaan, Aaditya K. Singh, Rylan Schaeffer, Andrew Poulton, Sanmi Koyejo, Pontus Stenetorp, Sharan Narang, and Dieuwke Hupkes. 2024. Quantifying Variance in Evaluation Benchmarks. *Preprint*, arXiv:2406.10229.

Vahid Majdinasab, Amin Nikanjam, and Foutse Khomh. 2025. Trained Without My Consent: Detecting Code Inclusion In Language Models Trained on Code. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–46.

Markdown Guide. 2025. Extended Syntax | Markdown Guide. https://www.markdownguide.org/extended-syntax/.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13215–13223, Miami, Florida, USA. Association for Computational Linguistics.

Meta. 2025. Llama 3.2 | Model Cards and Prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_2/.

Norman Mu, Jonathan Lu, Michael Lavery, and David Wagner. 2024. A closer look at system message robustness. In *Neurips Safe Generative AI Workshop 2024*.

Moin Nadeem, Anna Bethke, and Siva Reddy. 2021. StereoSet: Measuring stereotypical bias in pretrained language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5356–5371, Online. Association for Computational Linguistics.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2024. A Comprehensive Overview of Large Language Models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72.

Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q. Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–26.

Muhammed Nihal. 2024. The Race to Zero Latency: How to Optimize Code for High-Frequency Trading Quant Firms.

Mbithe Nzomo. 2025. Absolute vs Relative Imports in Python – Real Python. https://realpython.com/absolute-vs-relative-python-imports/.

OpenAI. 2025. Models - OpenAI API. https://platform.openai.com.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. GPT-4 Technical Report. *Preprint*, arXiv:2303.08774.

Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. 2024. Evaluating In-Context Learning of Libraries for Code Generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), {NAACL} 2024, Mexico City, Mexico, June 16-21, 2024*. arXiv.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 87–94. IEEE Computer Society.

Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. 2024. Is Temperature the Creativity Parameter of Large Language Models? In *International Conference on Computational Creativity*. arXiv.

PyPI. 2025. PyPI · The Python Package Index. https://pypi.org/.

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. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. 2024. An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, pages 111–118, New York, NY, USA. Association for Computing Machinery.

Matthew Renze and Erhan Guven. 2024. The Effect of Sampling Temperature on Problem Solving in Large Language Models. In *Findings of the Association for Computational Linguistics: {EMNLP} 2024, Miami, Florida, USA, November 12-16, 2024*. arXiv.

June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: The Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, pages 102–106, New York, NY, USA. Association for Computing Machinery.

Advait Sarkar and Ian Drosos. 2025. Vibe coding: Programming through conversation with artificial intelligence. *Preprint*, arXiv:2506.23253.

SciPy. 2025. Kendalltau — SciPy v1.16.2 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kendalltau.html.

Matthew Smith. 2025. AI Vibe Coding: Engineers' Secret to Fast Development - IEEE Spectrum. https://spectrum.ieee.org/vibe-coding.

Ian Somerville. 2016. *Software Engineering, Global Edition*. Pearson Education.

Minaoar Hossain Tanzil, Gias Uddin, and Ann Barcomb. 2024. "How do people decide?": A Model for Software Library Selection. In *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, pages 1–12.

Lukas Twist and Jie M. Zhang. 2026. A Study of Library Usage in Agent-Authored Pull Requests. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, MSR '26, Rio de Janeiro, Brazil. ACM.

vinta. 2025. Awesome Python. https://awesome-python.com/.

Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R. Lyu. 2024. Exploring Multi-Lingual Bias of Large Code Models in Code Generation. *Preprint*, arXiv:2404.19368.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pages 24824–24837, Red Hook, NY, USA. Curran Associates Inc.

Ankit Yadav, Himanshu Beniwal, and Mayank Singh. 2024. PythonSaga: Redefining the Benchmark to Evaluate Code Generating LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 17113–17126, Miami, Florida, USA. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, {MSR} 2018, Gothenburg, Sweden, May 28-29, 2018*. arXiv.

Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-Library-Oriented Code Generation with Large Language Models. *Preprint*, arXiv:2307.15370.

Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Qian Wang, Chao Shen, and Yang Liu. 2025. The Invisible Hand: Unveiling Provider Bias in Large Language Models for Code Generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21376–21403, Vienna, Austria. Association for Computational Linguistics.

Li Zhong and Zilong Wang. 2024. Can LLM replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, volume 38 of *AAAI'24/IAAI'24/EAAI'24*, pages 21841–21849. AAAI Press.

Xin Zhou, Martin Weyssow, Ratnadira Widyasari, Ting Zhang, Junda He, Yunbo Lyu, Jianming Chang, Beiqi Zhang, Dan Huang, and David Lo. 2025. LessLeak-Bench: A First Investigation of Data Leakage in LLMs Across 83 Software Engineering Benchmarks. *Preprint*, arXiv:2502.06215.

Xuekai Zhu, Daixuan Cheng, Hengli Li, Kaiyan Zhang, Ermo Hua, Xingtai Lv, Ning Ding, Zhouhan Lin, Zilong Zheng, and Bowen Zhou. 2025. How to Synthesize Text Data without Model Collapse? *Preprint*, arXiv:2412.14689.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *13th International Conference on Learning Representations (ICLR25)*. arXiv.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2020. Fine-Tuning Language Models from Human Preferences. *Preprint*, arXiv:1909.08593.

Appendices

A LLM Details

In this study, we evaluate a range of LLMs, with our selection described in Section 3.1. To support transparency and reproducibility, Table 5 lists the exact version of each LLM and the configuration options used.

A.1 API Usage

Although we rely on default API settings for *temperature* and *top_p*, we also explicitly set these values during the API calls to ensure reproducibility if default values change in the future. Closed-source LLMs are prompted using the default values of their corresponding APIs; open-source LLMs are prompted using the default values published on each LLM's *Hugging Face*[1] repository. Each LLM used in this study is accessed via an API, under the providers' terms of service, and is used as expected.

A.2 Response Data Extraction

The LLMs used in the study typically format their responses as `Markdown` allowing for automatic data extraction, any responses where `Markdown` cannot be detected are saved separately and subjected to manual analysis for inclusion in our results. In `Markdown`, code blocks are denoted by a triple backtick followed by the programming language name (Markdown Guide, 2025), therefore, we can use `regex` matching to extract code blocks and programming languages. For library experiments, if a `Python` code block has been extracted, further `regex` matching is used to extract the imported libraries. There are two correct syntaxes for external imports in `Python` (Nzomo, 2025): "`import <library>`"; and "`from <library> import <member>`".

B Experimental Details

B.1 Experiment 1: Library Preferences

Benchmark Tasks. We use BigCodeBench (Zhuo et al., 2024) as our primary benchmark for investigating library preferences, as discussed in Section 3.2. BigCodeBench is released under the Apache license 2.0, allowing usage for this study; and we use the BigCodeBench

[1] https://huggingface.co/models

Table 5: ***LLM Configuration.*** Detailed information of the LLMs used in the study. Entries marked with "–" indicate that the information was not available at the time of the study (September 2025).

Model	Version	Platform	Release	Knowledge cut-off	Open-source?	Code model?	Size	Parameters	
								temp	*top_p*
GPT-4o-mini (OpenAI et al., 2024)	`gpt-4o-mini-2024-07-18`	`https://openai.com/api/`	July '24	Oct. '23	✗	✗	-	1.0	1.0
GPT-3.5-turbo (Andrew Peng et al., 2023)	`gpt-3.5-turbo-0125`	`https://openai.com/api/`	Nov. '22	Sep. '21	✗	✗	-	1.0	1.0
Sonnet-3.5 (Anthropic, 2024)	`claude-3-5-sonnet-20241022`	`https://claude.com/platform/api`	Oct. '24	July '24	✗	✗	-	1.0	1.0
Haiku-3.5 (Anthropic, 2024)	`claude-3-5-haiku-20241022`	`https://claude.com/platform/api`	Oct. '24	July '24	✗	✗	-	1.0	1.0
Llama-3.2-3B (Meta, 2025)	`llama-3.2-3b-instruct-turbo`	`https://api.together.xyz/`	Sep. '24	Dec. '23	✓	✗	3B	0.6	0.9
Mistral-7B (Jiang et al., 2023)	`mistral-7b-instruct-v0.3`	`https://api.together.xyz/`	May '24	May '24	✓	✗	7B	0.7	1.0
Qwen-2.5-Coder (Hui et al., 2024)	`qwen-2.5-coder-32b-instruct`	`https://api.together.xyz/`	Nov. '24	Mar. '24	✓	✓	32B	0.7	0.8
DeepSeek-LLM (DeepSeek-AI et al., 2024)	`deepseek-llm-67b-chat`	`https://api.together.xyz/`	Nov. '23	May '23	✓	✗	67B	0.7	0.95

dataset as intended, to prompt LLMs to generate code.

We use a consistent prompt template for all tasks, inspired by the original BigCodeBench prompt with the added directive to use an external library to ensure that we can best measure the LLM preferences. The prompt template we use is given in Prompt 1, and a full example prompt (using a BigCodeBench task from the computation domain) is given in Prompt 2.

Prompt 1 *"**<task description>** You should write self-contained python code. Choose, import and use at least one external library."*

Prompt 2 *"Create a dictionary where keys are specified letters and values are lists of random integers. Then calculate the mean of these integers for each key and return a dictionary of these means. You should write self-contained python code. Choose, import and use at least one external library."*

Project Initialisation Tasks. To investigate library preferences in a more realistic setting, we use a selection of project initialisation tasks inspired by the groups of libraries on Awesome Python (vinta, 2025), as described in Section 3.2. The exact task descriptions used are listed below, along with a selection of the possible libraries for each, as per the libraries given on Awesome Python.

1. **Database**: *"Write the initial python code for a database project with an orm layer."*
   Possible libraries: Django, peewee, Pony, pyDAL, SQLAlchemy.
2. **Deep learning**: *"Write the initial python code for a deep learning project implementing a neural network."*
   Possible libraries: Caffe, Keras, PyTorch, scikit-learn, TensorFlow.
3. **Distributed computing**: *"Write the initial python code for a distributed computing project."*
   Possible libraries: Celery, Dask, Luigi, PySpark, Ray.
4. **Web scraper**: *"Write the initial python code for a web scraping and analysis library."*
   Possible libraries: BeautifulSoup, lxml, MechanicalSoup, Scrapy.
5. **Web server**: *"Write the initial python code for a backend API web server."*
   Possible libraries: Django Rest, FastAPI, Flask, Pyramid, Starlette.

B.2 Experiment 2: Language Preferences

Benchmark Tasks. We use a selection of benchmark datasets to investigate LLM preferences for programming languages, chosen for their language-agnostic NL-only task descriptions and the likelihood of having solutions published in multiple languages, as described in Section 3.3. Here, we provide full details of each of the six datasets chosen, including: the number of tasks; number of valid tasks (those that do not contain references to programming languages); programming languages of the provided ground-truth solutions; and the dataset licence, showing that the dataset is available for use by this study. Additionally, all dataset usage is consistent with their intended use, to prompt LLMs to generate code.

1. **Basic**: Short coding problems that an entry-level programmer could solve.
   - **Multi-HumanEval** (Athiwaratkun et al., 2023): A multi-language version of the popular HumanEval (Chen et al., 2021) dataset originally published by OpenAI in 2021, to test code generation from docstrings.
     Tasks: 160.
     Solution languages: C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, Typescript.
     Licence: Apache license 2.0.

- **MBXP** (Athiwaratkun et al., 2023): A multi-language version of the popular MBPP (Austin et al., 2021) dataset originally published by Google in 2021, containing simple problems that require short solutions.
  Tasks: 968.
  Solution languages: C#, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, Typescript.
  Licence: Apache license 2.0.

2. **Real-world**: Code generation tasks constructed from realistic coding situations that developers have faced. Because the problems are realistic, there are likely to be solutions available online in many languages.
   - **AixBench** (Hao et al., 2022): A method-level code generation dataset built from comments found in public GitHub `Java` methods.
     Tasks: 161 total, 129 valid.
     Solution language: Java.
     Licence: MIT.
   - **CoNaLa** (Yin et al., 2018): A code generation from NL dataset built from coding problems found on Stack Overflow.
     Tasks: 2,879 total, 2,659 valid.
     Solution language: Python.
     Licence: MIT.
3. **Coding challenge**: In-depth problems that typically require more thinking in order to solve. Sourced from online coding competition websites, they are language-agnostic by nature. Users can solve the problems in any language they choose, so there are solutions published online in many languages (Furia et al., 2024).
   - **APPS** (Hendrycks et al., 2021): A large collection of problems from various open-access coding websites, covering various difficulty levels.
     Tasks: 10,000 total, 8,623 valid.
     Solution language: Python.
     Licence: MIT.
   - **CodeContests** (Li et al., 2022): A competitive coding dataset from 2024, initially used to train AlphaCode (DeepMind, 2024).
     Tasks: 13,610 total, 12,830 valid.
     Solution languages: C++, Java, Python.
     Licence: Creative Commons Attribution 4.0.

We use a consistent prompt template compatible with each chosen dataset, the exact template is given in Prompt 3.

Prompt 3 *"Generate a code-based solution, with an explanation, for the following task or described function: **<task description>**"*

Project Initialisation Tasks. To investigate library preferences in a more realistic setting, we use a selection of project initialisation tasks designed to challenge the LLMs' default programming language choice of `Python`, as described in Section 3.3. The exact task descriptions used are listed below, along with a description of why `Python` is a suboptimal choice compared to other options.

1. **Concurrent web server**: *"Write the initial code for a high-performance web server to handle a large number of concurrent requests."*
   High performance web servers require efficient thread management and concurrency control, areas where `Rust` (with full async capabilities) or `C++` (with multithreading) outperform `Python` (Bugden and Alahmar, 2022).
2. **Cross-platform graphical user interface**: *"Write the initial code for a modern cross-platform application with a graphical user interface."*
   Graphical user interfaces benefit from native performance optimisations and low-latency rendering, `Python` is inefficient with native libraries, making low-level languages like `C` and `C++` more suitable (Karczewski, 2021).
3. **Low-latency trading platform**: *"Write the initial code for a low-latency trading platform that will allow scaling in the future."*
   Financial trading systems demand minimal execution delays, and benefit from precise control over hardware and low-level memory management, areas where `C++` and `Go` are better than Python (Nihal, 2024).
4. **Parallel task processing library**: *"Write the initial code for a high performance parallel task processing library."*
   `Python` is suboptimal here because it does not have true parallelism due to the global interpreter lock (Gross, 2023), compiled languages with true built-in parallelism like `C`, `C++`, and `Rust` are more suitable.
5. **System-level application**: *"Write the initial code for a command line application to perform system-level programming."*
   System-level programming inherently relies on direct hardware interaction and efficient memory usage, making `C`, `C++`, and `Rust` better choices than Python (Costanzo et al., 2021).

Table 6: ***Ground Truth Analysis.*** The *eight* most-used libraries when solving BigCodeBench tasks, with the number of tasks that contain those libraries in the ground-truth solution. Per model, $\in GT$ shows the number of tasks with a response using that library when it is in the ground-truth, $\notin GT$ is the number of tasks with a response using that library when it is **not** in the ground-truth

Library	Ground Truth Task Count ↓	GPT-4o-mini		GPT-3.5-turbo		Sonnet-3.5		Haiku-3.5		Llama-3.2-3B		Qwen-2.5-Coder		DeepSeek-LLM		Mistral-7B	
		$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$	$\in GT$	$\notin GT$
pandas	232	197	57	195	57	198	61	197	58	182	52	188	49	167	43	179	43
NumPy	220	181	133	181	122	188	152	206	165	182	143	155	83	130	76	149	94
Matplotlib	216	217	56	215	46	208	63	216	63	214	64	214	50	208	48	207	44
scikit-learn	98	84	4	84	6	80	6	90	5	81	7	88	4	81	6	75	6
SciPy	62	40	8	33	6	42	7	47	12	42	22	41	10	22	7	26	9
Seaborn	43	25	10	24	11	28	19	29	28	16	3	22	3	20	3	21	10
Requests	25	27	3	27	3	26	2	27	4	26	3	27	2	27	4	27	5
NLTK	18	10	7	13	6	11	10	12	9	12	12	11	5	7	6	10	10

B.3 Experiment 3: Recommendation Consistency

For this experiment, we examine whether LLMs are consistent between their library recommendations for a task and what they actually use, as described in Section 3.4. For this, we use Kendall's τ to compare a *recommendation ranking*, provided by the LLMs NL recommendations, against the *usage ranking* provided by the usage results from Experiments 1 and 2. Here, we give full details of how we derive both rankings and complete the calculation.

Prompt design. We use Prompt 4 to generate LLM recommendations for a task, asking for the best "python libraries" or "languages" depending on which technological preference we are investigating. The prompt is intentionally conversational to mirror typical developer-LLM interactions (Akhoroz and Yildirim, 2025); and we do not limit the number of recommendations to allow the LLM to provide all suitable options, as it would be interesting to see if the LLM does not recommend a technology at all but then uses it in a response. The ranking from each response is manually extracted, with the arithmetic mean rank of each item calculated to give a final ranking.

To check sensitivity to prompt wording, we conducted a small ablation study, varying the exact language used to ask for recommendations, across a handful of representative project initialisation tasks and LLMs. We used language such as: "list, in order, the best..." (final prompt), "list the top...", "recommend the best...". The prompt variations produced similar ranked outputs, but we found that asking for the recommendations "in order" gave a response that could more reliably be converted into a ranking.

Prompt 4 *"List, in order, the best **(python libraries | languages)** for the following task: **<task description>**"*

Worked example. Here we present an example for how we calculate the Kendall τ statistic for our data, for the LLM GPT-4o-mini and the *concurrent web server* project to investigate library preferences. We first need to determine the *recommendation ranking* using Prompt 4. In our example, the LLM responded with the following three rankings:

1. `rust, go, javascript, cpp, java, csharp, elixir, python, php, ruby`
2. `go, rust, javascript, cpp, java, csharp, elixir, python, kotlin`
3. `go, rust, javascript, cpp, csharp, java, python, php, elixir, scala`

Taking averages, we get a final *recommendation ranking* of: `go, rust, javascript, cpp, java, csharp, elixir, python, php, kotlin, ruby, scala`

When generating responses for the task in Experiment 2, `JavaScript` was used 73% of the time, `Python` 28% and `Go` 2%. This gives a *usage ranking* of: `JavaScript, Python, Go`.

A tuple is then created for each item (e.g. `Python`, `Rust`, `Java`, etc...), containing both of its ranks – in this example `JavaScript` has a tuple of (1, 3), and `Python` has (2, 8). If an item is not included within a ranking, then it is assumed to be ranked joint last with other items that are also not included. These pairs of item ranks are then processed using the implementation of Kendall's τ provided by SciPy (SciPy, 2025), giving both the calculated statistic and the p-value as outputs.

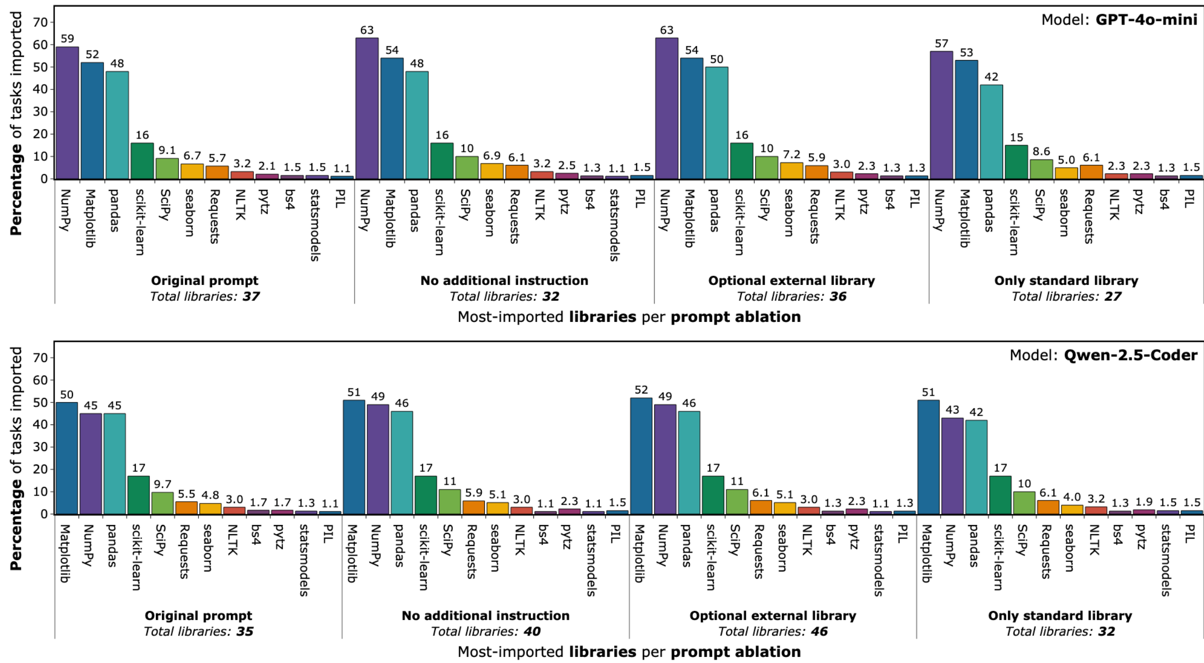

Figure 3: ***Ablation study results.*** Libraries used by LLMs when responding to BigCodeBench tasks across various prompt templates. For GPT-4o-mini (top) and Qwen-2.5-Coder (bottom), we give the most-used libraries with the percentage of tasks that had a response importing them, and the total unique libraries used. Other libraries are imported for *less than 1%* of tasks.

C Extended Results

Here we present some additional results that we could not fit into the main paper, along with further statistical analysis of the results from the experiments.

C.1 Ground Truth Analysis

In Experiment 1, when investigating LLMs' preferences for libraries when solving benchmark tasks, we observe an apparent overuse of data science libraries; often using a heavyweight library for a simple task (for example, importing `Numpy` to perform simple calculations). Here, we investigate this further, presenting some detailed results of how often the most-used libraries are imported in solutions where they match the ground truth.

Table 6 shows the results of this analysis. Although this pattern of overuse is visible in the top three libraries (`pandas`, `NumPy`, `Matplotlib`), we observe that `NumPy` in particular is used for many tasks where it is not a part of the ground-truth solution. Across LLMs, for 35-45% of the tasks for which `NumPy` is imported, it is not a part of the ground-truth solution.

C.2 Ablation Study

Motivation. When analysing library preferences on the benchmark tasks, our main prompt explicitly asks the model to import an external library. We deliberately made this choice to focus the experiment on library selection, but it raises a fair concern: the instruction itself might inflate external-library usage. Therefore, to check how sensitive our results are to this wording, we perform a small ablation study.

Experimental Design. We repeat the *benchmark task* experiment for library preferences (from Section 3.2) for both GPT-4o-mini and Qwen-2.5-Coder, using alternative prompt templates. We keep the BigCodeBench-style request to "write self-contained python code", and vary only the additional phrasing so that the requirement to use external libraries is expressed with different levels of strictness. The prompt templates we use are as follows, from most to least strict regarding the use of extrernal libraries:

1. **Original prompt:** *"**<task description>** You should write self-contained python code. Choose, import and use at least one external library."*
2. **Optional external library:** *"**<task description>** You should write self-contained python code. Use an external library if you want to."*
3. **No additional instruction:** *"**<task description>** You should write self-contained python code. "*

Table 7: ***Results for Varying Temperature.*** Languages used for project initialisation tasks when varying temperature. For each temperature (t) and project, the preferred languages (l) are given, along with the percentage (p) of responses that used that language, and a count of the total used (if necessary). The most-used language in each case is in bold.

Language Task	GPT-4o-mini								Qwen-2.5-Coder							
	temp=0.0		temp=0.5		temp=1.0		temp=1.5		temp=0.0		temp=0.5		temp=1.0		temp=1.5	
	l	p	l	p	l	p	l	p	l	p	l	p	l	p	l	p
Concurrent web server	**JavaScript**	**73%**	**JavaScript**	**63%**	**JavaScript**	**68%**	**JavaScript**	**58%**	**Python**	**83%**	**Python**	**85%**	**Python**	**82%**	**Python**	**86%**
	Python	27%	Python	36%	Python	32%	Python	39%	Go	16%	Go	15%	Go	17%	Go	11%
	Go	1%	Go	1%	Go	3%	*Total used*	*6*	-	-	-	-	JavaScript	2%	JavaScript	5%
Cross-platform graphical user interface	**JavaScript**	**84%**	**JavaScript**	**94%**	**JavaScript**	**76%**	**JavaScript**	**62%**	**Dart**	**96%**	**Dart**	**100%**	**Dart**	**92%**	**Dart**	**81%**
	Dart	41%	Dart	50%	Dart	48%	Dart	30%	JavaScript	11%	JavaScript	7%	JavaScript	19%	JavaScript	35%
	Total used	*4*	*Total used*	*4*	*Total used*	*5*	*Total used*	*6*	Python	3%	-	-	Python	4%	Python	2%
Low-latency trading platform	**Python**	**100%**	**Python**	**100%**	**Python**	**100%**	**Python**	**65%**	**Python**	**100%**	**Python**	**100%**	**Python**	**100%**	**Python**	**100%**
	C++	1%	-	-	-	-	JavaScript	1%	-	-	-	-	-	-	SQL	1%
Parallel task processing library	**Python**	**99%**	**Python**	**100%**	**Python**	**99%**	**Python**	**97%**	**Python**	**58%**	**Python**	**69%**	**Python**	**56%**	**Python**	**62%**
	C++	1%	-	-	C++	1%	C++	4%	C++	42%	C++	31%	C++	44%	C++	38%
System-level application	**Python**	**87%**	**Python**	**93%**	**Python**	**89%**	**Python**	**83%**	**C**	**100%**	**C**	**100%**	**C**	**100%**	**C**	**100%**
	C	16%	C	9%	C	17%	C	17%	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	Go	1%	-	-	-	-	-	-	-	-

4. **Only standard library:** *"<task description> You should write self-contained python code. Use only the standard library unless strictly necessary."*

Results. The results are shown in Figure 3. Across all prompt variants, we observe that the relative library preferences remain remarkably stable. For both GPT-4o-mini and Qwen-2.5-Coder, the same small set of well-established libraries consistently dominate usage, regardless of whether the prompt explicitly requires an external dependency, merely allows one, or discourages non-standard libraries. Softening or removing the instruction to use external libraries does not substantially reduce the usage rates of the most common libraries; in some cases, these libraries are used even more frequently.

A clear shift only emerges when the prompt explicitly instructs the LLM to "use only the standard library." Under this condition, both models show a notable reduction in external library usage rates, alongside a decrease in the total number of distinct libraries imported. But even then, when external libraries are used, they are drawn from the same well-established set observed under other prompt variants.

Overall, these results suggest that prompt phrasing primarily affects how often models rely on external libraries, but has limited influence on which libraries are selected. The consistency of these patterns across prompt variants and across two different LLMs supports our central finding that LLMs exhibit stable preferences for established libraries, rather than these preferences being an artefact of a particular prompt formulation.

Table 8: ***LLM Similarities for Benchmark Tasks.*** Kendall's τ correlation between different LLMs', comparing the languages and libraries they use for benchmark tasks. Only statistically significant correlations are given (p-values < 0.05). Values near 1.0 / -1.0 indicate strong agreement / disagreement.

	GPT-4o-mini	GPT-3.5-turbo	Sonnet-3.5	Haiku-3.5	Llama-3.2-3B	Qwen-2.5-Coder	DeepSeek-LLM	Mistral-7B
GPT-4o-mini	-	0.66	0.58	0.49	0.44	-	0.45	0.52
GPT-3.5-turbo	0.58	-	0.58	0.55	0.57	-	0.62	0.67
Sonnet-3.5	0.49	0.62	-	0.50	0.60	-	0.47	0.51
Haiku-3.5	0.50	0.65	0.60	-	0.56	0.40	0.57	0.43
Llama-3.2-3B	0.65	0.57	0.50	0.55	-	0.51	0.53	0.65
Qwen-2.5-Coder	0.57	0.62	0.54	0.61	0.51	-	-	0.44
DeepSeek-LLM	0.51	0.62	0.44	0.52	0.55	0.47	-	0.65
Mistral-7B	0.44	0.55	0.40	0.44	0.49	0.43	0.50	-

Upper triangle: Language Tasks; lower triangle: Library Tasks

C.3 Varying Temperature

Motivation. Temperature is considered the creativity parameter (Peeperkorn et al., 2024) of LLMs, altering the variability and randomness in responses; adjusting its value is an obvious method for potentially improving the diversity in LLMs' coding choices. We also make the deliberate decision in our experiments to use the default temperature for each LLM, to match the expected usage by developers, but this causes the open-source LLMs in our study (that have lower default temperatures) to appear less diverse. Therefore, we conduct an initial investigation into how adjusting the temperature may allow LLMs to diversify their choice of programming language during project initialisation tasks.

Experimental Design. We repeat the *project initialisation* experiment for library preferences (from Section 3.3) for both an open-source and a closed-source LLM (GPT-4o-mini and Qwen-2.5-Coder),

Table 9: *LLM Recommendations.* Top *five* library or programming language recommendations given by each LLM for each project initialisation task.

Project Task	GPT-4o-mini	GPT-3.5-turbo	Sonnet-3.5	Haiku-3.5	Llama-3.2-3B	Qwen-2.5-Coder	DeepSeek-LLM	Mistral-7B
Database	1. sqlalchemy 2. djangoorm 3. peewee 4. tortoise-orm 5. ponyorm	1. sqlalchemy 2. peewee 3. djangoorm 4. ponyorm 5. sqlobject	1. sqlalchemy 2. djangoorm 3. peewee 4. tortoiseorm 5. ponyorm	1. sqlalchemy 2. djangoorm 3. peewee 4. sqlobject 5. ponyorm	1. sqlalchemy 2. djangoorm 3. peewee 4. ponyorm 5. djangomodel	1. sqlalchemy 2. djangoorm 3. peewee 4. tortoiseorm 5. sqlobject	1. sqlalchemy 2. djangoorm 3. peewee 4. sqlobject 5. storm	1. sqlalchemy 2. alembic 3. flask-sqlalchemy 4. pyramid-sqlalchemy 5. djangoorm
Deep learning	1. tensorflow 2. torch 3. keras 4. fastai 5. mxnet	1. tensorflow 2. torch 3. keras 4. sklearn 5. numpy	1. tensorflow 2. torch 3. keras 4. numpy 5. sklearn	1. tensorflow/keras 2. torch 3. jax 4. mxnet 5. keras	1. numpy 2. scipy 3. tensorflow 4. keras 5. torch	1. tensorflow 2. torch 3. keras 4. fastai 5. mxnet	1. tensorflow 2. keras 3. torch 4. mxnet 5. chainer	1. tensorflow 2. torch 3. keras 4. sklearn 5. mxnet
Distributed computing	1. dask 2. ray 3. celery 4. apachespark 5. multiprocessing	1. dask 2. celery 3. pyspark 4. ray 5. mpi4py	1. ray 2. dask 3. pyspark 4. celery 5. multiprocessing	1. dask 2. ray 3. pyspark 4. celery 5. zeromq	1. dask 2. joblib 3. ray 4. apachespark 5. mpi4py	1. dask 2. ray 3. apachespark 4. celery 5. joblib	1. dask 2. ray 3. pyspark 4. mpi4py 5. pycompss	1. dask 2. pyspark 3. celery 4. ipyparallel 5. futures
Web scraper	1. requests 2. bs4 3. lxml 4. scrapy 5. pandas	1. bs4 2. scrapy 3. pandas 4. requests 5. numpy	1. requests 2. bs4 3. selenium 4. scrapy 5. pandas	1. requests 2. bs4 3. scrapy 4. lxml 5. selenium	1. bs4 2. scrapy 3. requests 4. lxml 5. selenium	1. requests 2. bs4 3. lxml 4. scrapy 5. pandas	1. bs4 2. requests 3. scrapy 4. selenium 5. pandas	1. requests 2. bs4 3. pandas 4. numpy 5. matplotlib
Web server	1. flask 2. fastapi 3. django 4. tornado 5. falcon	1. flask 2. django 3. fastapi 4. tornado 5. sanic	1. fastapi 2. flask 3. aiohttp 4. starlette 5. djangorestframework	1. fastapi 2. flask 3. djangorestframework 4. tornado 5. starlette	1. flask 2. django 3. fastapi 4. pyramid 5. sanic	1. flask 2. djangorestframework 3. fastapi 4. tornado 5. bottle	1. flask 2. django 3. fastapi 4. pyramid 5. bottle	1. flask 2. fastapi 3. djangorestframework 4. pyramid 5. tornado
Concurrent web server	1. go 2. rust 3. javascript 4. cpp 5. java	1. go 2. rust 3. cpp 4. java 5. javascript	1. rust 2. go 3. cpp 4. java 5. javascript	1. rust 2. go 3. cpp 4. java 5. erlang/elixir	1. rust 2. go 3. cpp 4. javascript 5. python	1. go 2. rust 3. c 4. cpp 5. java	1. cpp 2. go 3. rust 4. python 5. javascript	1. c/cpp 2. go 3. rust 4. erlang/elixir 5. java
Cross-platform graphical user interface	1. javascript 2. csharp 3. python 4. java 5. dart	1. javascript 2. python 3. java 4. dart 5. csharp	1. dart 2. javascript/typescript 3. python 4. csharp 5. java	1. dart 2. javascript 3. python 4. csharp 5. kotlin	1. java 2. csharp 3. kotlin 4. javascript 5. python	1. csharp 2. kotlin 3. flutter 4. react 5. java	1. javascript 2. python 3. csharp 4. java 5. swift	1. javascript 2. dart 3. kotlin 4. swift 5. csharp
Low-latency trading platform	1. cpp 2. java 3. rust 4. go 5. python	1. cpp 2. java 3. python 4. go 5. rust	1. cpp 2. java 3. rust 4. c 5. go	1. cpp 2. rust 3. go 4. java 5. python	1. rust 2. cpp 3. go 4. java 5. python	1. cpp 2. rust 3. java 4. go 5. python	1. cpp 2. java 3. python 4. csharp 5. javascript	1. cpp 2. java 3. python 4. go 5. rust
Parallel task processing library	1. cpp 2. rust 3. go 4. java 5. python	1. cpp 2. rust 3. go 4. java 5. python	1. rust 2. cpp 3. go 4. java 5. c	1. rust 2. cpp 3. go 4. d 5. julia	1. cpp 2. rust 3. csharp 4. java 5. go	1. cpp 2. rust 3. go 4. java 5. python	1. cpp 2. java 3. python 4. csharp 5. go	1. cpp 2. rust 3. go 4. java 5. scala
System-level application	1. c 2. cpp 3. rust 4. go 5. python	1. c 2. rust 3. go 4. cpp 5. python	1. c 2. rust 3. cpp 4. go 5. assembly	1. c 2. rust 3. cpp 4. go 5. d	1. c 2. cpp 3. rust 4. go 5. assembly	1. c 2. cpp 3. rust 4. go 5. assembly	1. c 2. cpp 3. rust 4. go 5. python	1. c/cpp 2. rust 3. go 4. assembly 5. swift

with varying temperatures. The APIs we use accept temperatures of 0.0-2.0, but larger temperatures can lead to unreliable response parsing (Renze and Guven, 2024), an effect we observed in our preliminary testing. Therefore, to ensure reliable response parsing, we use the following temperatures: 0.0, 0.5, 1.0 and 1.5.

Results. The results are shown in Table 7. For GPT-4o-mini, altering temperature has a clear but minimal impact: for each task, a higher temperature led to the most used language being used in fewer responses, along with a wider variety of languages. For Qwen-2.5-Coder, altering the temperature seems to have had no effect on the diversity of languages used at all; higher temperatures did not cause a wider variety of languages to be used. Interestingly, for both LLMs, a temperature of 0.0 does not guarantee that the most used language will remain consistent.

C.4 Similarities Across LLMs

From our results LLMs appear to have very similar preferences, here, we do some additional statistical analysis to investigate the extent of this. We compare the empirical usage rankings of libraries or programming languages for each previous experiment and each pair of LLMs. We again calculate Kendall's τ coefficient to understand the rank correlation. Table 8 shows the results for the benchmark tasks.

From the table, we can observe that there is a median coefficient of 0.54 (range 0.40-0.67) across all LLM pairs for languages used for benchmark tasks, with only three results not statistically significant. For libraries, the coefficients for all LLM pairs are statistically significant, with a median coefficient of 0.53 (range 0.40-0.65). This indicates that all LLMs have similar preferences when solving benchmark tasks. For project initialisation tasks, the correlation between preferences is much less clear. Only 16% of the coefficients between LLMs have statistical significance, and 13% is un-

Table 10: *Reasoning via Prompt Engineering.* Languages used for project initialisation when inducing reasoning via prompt engineering. The languages used are given, with the rank assigned to the language by the LLM, and the percentage of responses that used the language for each prompt style. The most-used language in each case has its percentage is in bold. *Base prompt* shows the results from the original experiment, for comparison.

GPT-4o

Language Task	Language	Rank ↓	*Base Prompt*	Step-by-step	Double-check	First list
Concurrent web server	Go	#1	2%	**77%**	47%	**98%**
	Rust	#2	0%	0%	0%	2%
	JavaScript	#3	**73%**	23%	**53%**	0%
	Python	#8	28%	0%	0%	0%
Cross-platform graphical user interface	JavaScript	#1	**72%**	**84%**	**50%**	**96%**
	Python	#3	14%	9%	33%	1%
	Dart	#5	38%	8%	21%	3%
	C++	#12	3%	0%	0%	0%
Low-latency trading platform	C++	#1	0%	**88%**	**66%**	**100%**
	Rust	#3	0%	2%	12%	0%
	Go	#4	0%	5%	15%	0%
	Python	#5	**100%**	9%	7%	0%
Parallel task processing library	C++	#1	1%	20%	8%	37%
	Rust	#2	0%	**54%**	**61%**	**61%**
	Go	#3	0%	20%	17%	1%
	Python	#5	**99%**	6%	15%	1%
System-level application	C	#1	23%	**92%**	**96%**	**100%**
	Rust	#3	0%	8%	5%	0%
	Python	#5	**81%**	4%	1%	0%

Qwen-2.5-Coder

Language Task	Language	Rank ↓	*Base Prompt*	Step-by-step	Double-check	First list
Concurrent web server	Go	#1	0%	**94%**	**95%**	**92%**
	Rust	#2	0%	6%	3%	8%
	Python	#12	**100%**	0%	2%	0%
Cross-platform graphical user interface	C#	#1	0%	6%	4%	7%
	Kotlin	#2	0%	0%	0%	2%
	Dart	#6	**100%**	**90%**	**81%**	**89%**
	Python	#7	0%	2%	15%	0%
Low-latency trading platform	C++	#1	0%	**89%**	**35%**	**96%**
	Rust	#2	0%	0%	2%	0%
	Java	#3	0%	1%	0%	0%
	Go	#4	0%	6%	15%	14%
	Python	#5	**100%**	27%	6%	10%
Parallel task processing library	C++	#1	**100%**	**42%**	**56%**	**66%**
	Rust	#2	0%	4%	30%	34%
	Python	#5	0%	0%	4%	0%
System-level application	C	#1	**100%**	**98%**	**94%**	**100%**
	C++	#2	0%	2%	4%	0%
	Rust	#3	0%	0%	2%	0%

defined due to an exact match of a single choice of technology.

We assume that the similarity across LLMs reflects shared training data sources – such as public GitHub repositories (Majdinasab et al., 2025) – while differences arise from their specific data and training variations. These differences appear to be more pronounced for the open-ended project initialisation tasks, hence the lack of significance in those coefficients.

C.5 LLM Recommendations

In Experiment 3 we investigate whether an LLMs NL recommendations for which library or programming language to use for a task aligns with what they actually use when writing code. In Section 4 we could only fit the rank correlation results (Table 4), without details of the *recommendation ranking* itself. Here, in Table 9, we present the top five recommendations from all LLMs, for each project initialisation task. Full rankings are available in our GitHub repository.

C.6 Reasoning via Prompt Engineering

Motivation. Chain-of-thought (CoT) prompting has been shown to elicit reasoning in LLMs (Wei et al., 2022); it may enable more optimal programming language choices when coding. Therefore, we conduct an initial investigation into whether these strategies can improve the consistency between an LLM's programming language recommendations and what it chooses to use when generating code.

Experimental Design. We repeat the *project initialisation* experiment for language preferences (from Section 3.3) for both GPT-4o-mini and Qwen-2.5-Coder. We append the original prompts with one of the following, either text that has been shown to elicit zero-shot reasoning behaviour, or with the directive to first rank the languages:

1. **Step-by-step:** *"Think step by step about which coding language you should use and why."* (Kojima et al., 2022)
2. **Double-check:** *"Double check the reasoning for your coding language choice before writing code."* (Chowdhury and Caragea, 2025)
3. **First-list:** *"First, list in order, the best coding languages for the task, then use this list to inform your language choice."*

Results. The results are shown in Table 10. The consistency between NL recommendations and code responses has improved across the board, with the top-ranked programming language now being the most used in 23/30 instances. Asking the LLM to first rank suitable languages in context shows the best alignment between its original ranking and the languages used, although it was still not perfect, showing potential for variability in the recommendations. Recommendations for the "Parallel processing" task are still inconsistent across all reasoning prompts; both LLMs now use `Rust` much more, but this only aligned with the recommendations for GPT-4o-mini. The reasoning prompts seemed to greatly increase the diversity of languages used by Qwen-2.5-Coder, indicating that they create more uncertainty in the responses.