

Parameterizing LLMs in Practice: An Empirical Study of LLMs Integrated into Software Systems

Agustín Olmedo 

LIDTUA (CIC), Facultad de Ingeniería, Universidad Austral, Buenos Aires, Argentina

Jannik Laval 

Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

Christelle Urtado 

SyCoIA, IMT Mines Ales, Ales, France

Sylvain Vauttier 

SyCoIA, IMT Mines Ales, Ales, France

Abstract

Large Language Models (LLMs) are increasingly embedded in software projects, yet little is known about how developers configure LLM parameters in the wild. Characterizing which parameters are defined, how many per project, which groups are co-defined, and what values are preferred can inform academics and tool builders about practices. This article mines open-source Python repositories on GitHub that integrate LLMs and uses AST-based static analysis to extract parameter assignments. Project-level definitions are analyzed to address prevalence (RQ1), configuration complexity and its structure (RQ2), and value distributions (RQ3). The final corpus includes 363 projects and gathers 7892 parameter definitions, with both the dataset and the code available for reproducibility. We observe that *temperature* is most frequently defined (90.36%), followed by *max_tokens* (56.75%), *top_p* (49.86%), and *top_k* (42.70%); penalty parameters are comparatively rare (RQ1). Projects typically define few parameters (mean ≈ 3), and this limited set expands incrementally around a stable core (*temperature* + *max_tokens*) (RQ2). Distributions suggest “defaults-in-practice”: *temperature* $\approx 0.0, 0.7$ and 1.0 , *top_p* ≈ 0.9 , *top_k* $\approx 0-50$, and *max_tokens* at 2^k values (e.g., 256/512/1024) (RQ3). In conclusion, developers favor minimalist, sampling-centric configurations with convergence around specific ranges. These descriptive findings support clearer configuration reporting, offer practical baselines for tools and education, and motivate future work on causes, longitudinal evolution and task/domain stratification.

2012 ACM Subject Classification Software and its engineering $\rightarrow$ Software libraries and repositories

Keywords and phrases Software Engineering for AI, Large Language Model, Hyperparameter, LLM configuration, LLM API, Mining Software Repository

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.60

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Dataset*: <https://doi.org/10.5281/zenodo.21456738> [9]

Funding This research was partially supported by the French National Research Agency (AdaptiveMLOps, ANR-24-IAS2-0004).

1 Introduction

Large Language Models (LLMs) have rapidly transitioned from research prototypes to widely adopted components across software projects, powering code assistants, conversational agents, and task-specific tools. This fast-paced adoption has sparked an equally rapid growth of empirical studies and community discussions around evaluation practices, reproducibility, and the transparent reporting of model configurations in software engineering (SE) contexts [2, 16, 17].



© Agustín Olmedo, Jannik Laval, Christelle Urtado, and Sylvain Vauttier;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 60; pp. 60:1–60:12



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In particular, configuration parameters exposed by LLM APIs (*e.g.*, temperature, top_p, top_k, token limits, etc.) directly influence output diversity, determinism, and verbosity [10]. Therefore, they are central to understanding how developers shape model behavior in practice [17]^{1,2}. Despite the relevance of configuration to the behavior and reliability of LLM-enabled software, we still lack a systematic characterization of *which* parameters developers actually define in real-world software projects, *how many* of them are typically defined, *which groups* of parameters tend to appear within the same project, and *what concrete values* are assigned in practice. Clarifying these aspects matters for at least three reasons: (i) it informs reproducibility and reporting standards in LLM-in-SE studies [2, 16]; (ii) it surfaces *de facto* practices and emergent conventions (*defaults-in-practice*) that can guide future evaluations; and (iii) it provides an empirical baseline for tool builders and academics.

This paper adopts a repository-mining perspective to study LLM configuration practices in the wild. It focuses explicitly on software projects that integrate LLMs as part of their functionality, and it analyzes configuration at the project level, as manifested in source code. Guided by the evidence accessible through static analysis, it addresses the following research questions:

- **RQ1.** Which LLM parameters are defined by developers in real-world software projects?
- **RQ2.** How complex are LLM configurations at the project level, and how is this complexity structured across parameters?
- **RQ3.** What values do developers assign to the LLM parameters they define?

We mine open-source repositories on GitHub and extract parameter assignments using an AST-based analysis pipeline tailored for Python code (*e.g.*, mining keyword arguments, dictionary literals, and class instantiations). We then construct a dataset of occurrence-level parameter definitions. The proposed study design is consistent with current recommendations on transparency, scoping, and limitation reporting for empirical studies involving LLMs [2, 16].

Across hundreds of projects, we observe that: (i) *temperature* is the most frequently defined parameter, with *max_tokens* and *top_p* also appearing in over half of the projects, whereas penalty-style parameters are comparatively rare; (ii) projects typically define a *small* number of parameters (mean ≈ 3), indicating a prevalent minimalist practice; (iii) the most common parameter groups defined at project level consistently include *temperature* plus *max_tokens* and either *top_p* or *top_k*; and (iv) value distributions suggest emerging *defaults-in-practice* – *e.g.*, *temperature* commonly set to 0.0, 0.7 and 1.0, *top_p* near 0.9, moderately small *top_k*, and rounded *max_tokens* (*e.g.*, 256, 512, 1024). These observations complement recent calls for clearer reporting of model versions and configurations and help situate real-world tuning practices within the broader Software Engineering literature on LLMs [2, 16, 17].

Paper structure. The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the study design. Section 4 reports the results, followed by threats to validity and conclusions.

¹ <https://markaicode.com/ollama-temperature-parameter-tuning-guide/>

² <https://platform.minimax.io/docs/api-reference/text-anthropic-api>

2 Related Work

LLMs in Software Engineering. Recent research has increasingly studied Large Language Models (LLMs) as integral components of software systems rather than isolated tools. Beyond prompt design and standalone evaluation, several works analyze how LLMs are embedded into real software pipelines to support tasks such as repository analysis, conversational interfaces, program comprehension, and developer assistance [17].

Empirical guidelines and surveys emphasize the need to treat LLMs as software artifacts whose behavior depends on contextual integration choices, including model selection, prompting strategies, and configuration parameters [2, 16]. Zheng et al. [17] provide a comprehensive survey of LLM usage across software engineering tasks, highlighting that most applications deploy LLMs as part of larger systems (e.g., assistants, agents, and analysis tools), and explicitly call for empirical studies grounded in real-world usage.

Several studies focus on LLM-enabled tools that operate over software repositories. Abedu et al. [1] report on the challenges of integrating LLM-based chatbots for mining software repositories, showing that system design decisions strongly affect effectiveness and reliability. Complementarily, De Martino et al. [4] propose PRIMES, a methodological framework for using LLMs within mining software repositories pipelines, emphasizing reproducibility, transparency, and controlled configuration of LLM components embedded in research workflows.

More broadly, experience reports and system-oriented studies stress that LLM behavior in software systems cannot be understood independently of how models are parameterized and orchestrated within applications [5].

Our study aligns with this body of work by focusing on software projects that integrate LLMs to deliver functionality. Rather than examining prompts or outcomes, we analyze how developers explicitly configure LLM parameters in the source code of real projects, providing complementary evidence on configuration practices that shape LLM behavior in deployed software systems.

LLMs and Mining Software Repositories. Mining Software Repositories (MSR) have been widely adopted as a methodological approach to characterize developer practices at scale. Previous work has applied MSR to study software architecture, development processes, and tool usage in a large corpus of projects [14]. More recently, some studies have explored how LLMs can be incorporated into MSR workflows themselves, for example, in data extraction or analysis pipelines [1, 4]. In contrast, our work takes the complementary perspective of applying MSR techniques to systematically analyze how LLMs are configured in practice, focusing specifically on parameter definitions observed directly in source code. This approach enables empirical characterization of real-world configuration practices grounded in actual software artifacts.

LLM parameter configuration in practice. While prompt-engineering surveys classify strategies for steering model behavior, LLM parameters remain central levers that developers tune in real systems [12]. Beyond general Machine Learning, where hyperparameter reporting and tuning are often under-documented, empirical software engineering with LLMs has specifically called for clearer configuration reporting to support comparability and replication [2, 13]. Our repository-based characterization complements these lines by documenting the prevalence of parameter definitions, common groups in projects, and the observed value bands that emerge as *defaults-in-practice*.

3 Study Design

This section introduces the research questions addressed by this article and describes the study design.

3.1 Research Questions

To understand how developers configure Large Language Models (LLMs) in real-world software projects, we define a set of research questions that align strictly with the evidence obtainable from the static analysis of code from repositories. Following community guidance on transparency and scoping for empirical studies involving LLMs [2, 16] and recent overviews of LLM use in software engineering [17], our questions avoid presuming that parameters observed in the same repository are necessarily used together in the same call to the model's API. Instead, we study which parameters are *defined* in the projects, how many are defined, which groups of parameters tend to be co-defined within projects, and which concrete values developers assign.

RQ1. Which LLM parameters are defined by developers in real-world software projects?

Establishing the prevalence of parameter definition provides a baseline characterization of which aspects of LLM behavior developers consider worth controlling in practice. It also supports reproducibility-oriented reporting by clarifying which parameters are most commonly set explicitly, as recommended by recent guidelines [2, 16].

RQ2. How complex are LLM configurations at the project level, and how is this complexity structured across parameters?

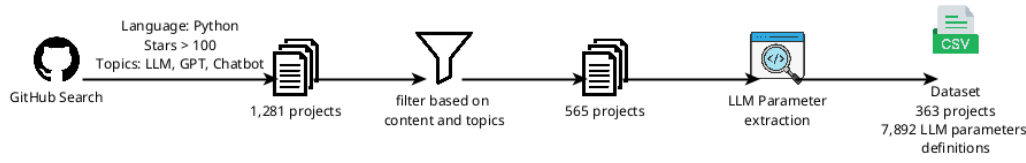
The number of parameters defined within a project is a proxy for configuration complexity. Understanding this number informs discussions about typical effort and the granularity at which practitioners customize LLM behavior across diverse codebases [17]. Furthermore, regarding how this complexity is structured, instead of assuming co-use in a single model's API call, we analyze project-level co-definition to reveal emergent defined parameter groups (*e.g.*, *temperature* with *max_tokens*, or *temperature* with *top_p*). This perspective respects the observational limits of repository mining while still surfacing meaningful patterns that may inform future, finer-grained studies [2, 16].

RQ3. What values do developers assign to the LLM parameters they define?

Distributions of parameter values capture *defaults-in-practice* and tuning conventions (*e.g.*, concentration of *temperature* around certain values). Such evidence can guide future controlled evaluations and help contextualize reported configurations in empirical research [17].

3.2 Data collection

In this section, we describe the data collection process. This involves selecting GitHub repositories of software projects that use LLMs and then extracting the LLM parameter definitions from the project's source code, resulting in a dataset of defined LLM parameters. The data collection process is shown in Figure 1. The source code used to create the dataset and the dataset is available online [9].



■ **Figure 1** Data collection process.

3.2.1 Project Selection

For this study, we focus on software projects that use LLMs in their code. Given the need for transparent access to source code and metadata [6], we restrict our analysis to open-source projects hosted on GitHub, the most widely used platform for open-source software development. To identify relevant projects, we queried the GitHub REST API and applied the following inclusion criteria:

1. **Project Topics:** Projects that include at least one of the following topics: *GPT*, *LLM*, or *Chatbot*.
2. **Programming Language:** Python. We restrict our analysis to Python projects as this language is predominant for integrating LLMs into software systems, serving as a primary ecosystem for LLM SDKs and frameworks [7, 15].
3. **Popularity Threshold:** Projects that have more than 100 stars. Prior work has shown that the number of stars is a proxy for project maturity, visibility, and community interest [3]. This filter helps exclude prototypes, proofs of concept, and early-stage repositories with little adoption.

Applying these filters yielded 1281 candidate projects.

Exclusion criteria. However, our study focuses specifically on software projects, excluding repositories that serve educational, demonstrative, or curatorial purposes. Therefore, we removed projects whose name or description contained any of the following terms: *scratch*, *papers*, *paper*, *colab*, *education*, *course*, *tutorial*, *guide*, *example*, *lessons*, *list*, *sample*, *notebook*, *deprecated*, *abandoned*, *articles*, *notes*, *doc*, or *demo*. This list was constructed inductively by manually reviewing the names and descriptions of the initial set of 1281 projects and identifying terms commonly associated with non-software or auxiliary content.

Additionally, some repositories matched our initial criteria but were unrelated to LLMs due to ambiguous terminology. For example, the term “gpt” may refer to the GUID Partition Table, rather than to Generative Pretrained Transformer. To mitigate such false positives, we refined the selection by requiring that a project include at least three of the following LLM-related topics: *ai*, *chatgpt*, *dall-e*, *generative-ai*, *generativeai*, *gpt*, *llm*, *chatbot*, *transformers*, *language-model*, *Chatbots*, *llms*, *agent*, *gemini*, *llama*, *app*, *ui*, *claude*, *artificial-intelligence*, *Large-language-models*, *lora*, *mistral*, *Langchain*, *Assistant*, *Ollama*, *agents*, *bert*. This list was compiled by examining the complete set of topics present across all 1281 initially collected repositories. After applying these additional topic-based and content-based filters, the final dataset consisted of 565 software projects that are suitable for our empirical analysis.

3.2.2 LLM Parameter Extraction

Large Language Model parameters are configuration settings that govern how a model produces text during prediction. These parameters allow developers to finetune model behavior, influencing characteristics such as creativity, determinism, verbosity, and lexical diversity. Understanding how these parameters are configured in realworld software projects provides insight into usage patterns and developer preferences.

In this work, we analyze how developers modify LLM parameters within their software repositories. We focus on the parameters that modern large language models commonly support, namely: temperature, top-k, top-p, min-p, frequency penalty, presence penalty, repetition penalty, and maximum tokens [8, 11].

To extract these configurations, we clone each selected Python project and statically analyze its source code using Python’s built-in `ast` module. AST-based parsing detects parameter values accurately across variable assignments, function calls, dictionary structures, and class instantiation patterns. This approach ensures robust identification while avoiding false positives caused by comments, examples, or documentation. Each extracted parameter instance is represented as a row in our dataset and includes: **project name, file path, line number, parameter name, and assigned value**.

Path exclusions. To ensure that we capture only parameters relevant to the actual system behavior, we exclude files whose paths contain any of the following terms: *doc*, *test*, *readme*, *fixture*, *benchmark*, *example*, *deprecated*, *not_working*, *demo*, *beta*, *migrations*, *sample*, *lock*, *evaluation*, or *result*. These directories typically contain non-production or auxiliary artifacts. After applying these path exclusions, our final dataset contains **7892 LLM parameter definitions** extracted from **363 projects**. It should be noted that when extracting the parameters from the 565 selected projects, solely 363 projects remained because others: (i) defined the parameters in some path that we excluded or (ii) only used the model with default parameters.

4 Results

RQ1: Which LLM parameters are defined by developers in real-world software projects?

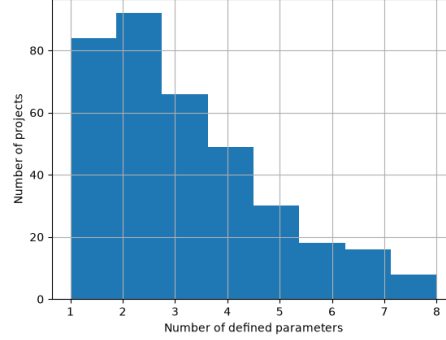
To answer this question, we obtained the number and percentage of projects that define each parameter from our dataset. Table 1 shows this information. We can highlight that the *temperature* parameter is defined in a majority of projects (90.36%), followed by the *max_token* and *top_p* parameters, which are defined in slightly more than half of the projects, and the *top_k* parameter, which is defined in 42.70% of the projects. We can observe that the penalty parameters (*repetition_penalty*, *presence_penalty*, and *frequency_penalty*) are used much less frequently (20.11%, 19.28% and 18.46% respectively). Finally, the *min_p* parameter is defined in only 16 projects (4.41%). These figures suggest that, in practice, developers prioritize sampling diversity/length controls over penalty-style regularization.

Findings (RQ1).

Most projects explicitly define *temperature*; over half define *max_tokens* and *top_p*, while penalty parameters are defined much less frequently. Overall, developers tend to control sampling and output length more often than repetition-related penalties.

■ **Table 1 Definition of parameters in projects.** Number and percentage of projects that define each parameter.

Parameter	# Projects	Projects (%)
temperature	328	90.36%
max_tokens	206	56.75%
top_p	181	49.86%
top_k	155	42.70%
repetition_penalty	73	20.11%
presence_penalty	70	19.28%
frequency_penalty	67	18.46%
min_p	16	4.41%



■ **Figure 2** Histogram of projects based on the number of parameters they define.

RQ2: How complex are LLM configurations at the project level, and how is this complexity structured across parameters? To understand the complexity of LLM configurations at the project level, we first examine the number of parameters defined in each project. Figure 2 shows the histogram of projects based on the number of parameters they define. We can observe that almost half of the projects define one or two parameters (176 projects). Next, there are 66 projects that define three parameters, followed by 49 projects that define four parameters, and 30 that define five parameters. Finally, there are 18 projects that define six parameters, 16 projects that define seven parameters and 8 that define eight parameters. Furthermore, the average number of parameters defined in a project is 3. These results suggest that most repositories adopt *lightweight* configurations, with relatively few projects engaging in broader, multi-parameter tuning.

Furthermore, to understand how the complexity of parameter configurations is structured, we are interested in knowing which groups of parameters are defined in the projects. For example, for projects that define two parameters, we want to understand which ones they typically configure. We are also interested in knowing the number of projects and the percentage in which each group of parameters is defined, to determine if there are parameter groups that developers tend to be more interested in. It should be clarified that we are not talking about configurations here, but about groups of parameters that are defined in the project.

Table 2 shows, for projects that define exactly k parameters, the groups of size k that appear within those projects, and reports the number and proportion of projects in which each group is defined. We emphasize that this is *project-level co-definition*, not guaranteed co-use in a single invocation. For projects that define exactly two parameters, the most common groups are $\{\text{max_tokens}, \text{temperature}\}$ (59.78%) and $\{\text{temperature}, \text{top_p}\}$ (15.22%), followed by $\{\text{temperature}, \text{top_k}\}$ (13.04%). For projects with three parameters, the groups $\{\text{temperature}, \text{top_k}, \text{top_p}\}$ (31.82%), $\{\text{max_tokens}, \text{temperature}, \text{top_k}\}$ (27.27%), and $\{\text{max_tokens}, \text{temperature}, \text{top_p}\}$ (19.70%) are the most frequent. For four parameters, $\{\text{max_tokens}, \text{temperature}, \text{top_k}, \text{top_p}\}$ dominates (38.78%), and for five parameters the configuration $\{\text{frequency_penalty}, \text{max_tokens}, \text{presence_penalty}, \text{temperature}, \text{top_p}\}$ is most common (50.00%). Taken together, these observations suggest that *temperature* appears in the most frequent groups across k , with *max_tokens* and one of *top_p/top_k* often accompanying it as the group size grows.

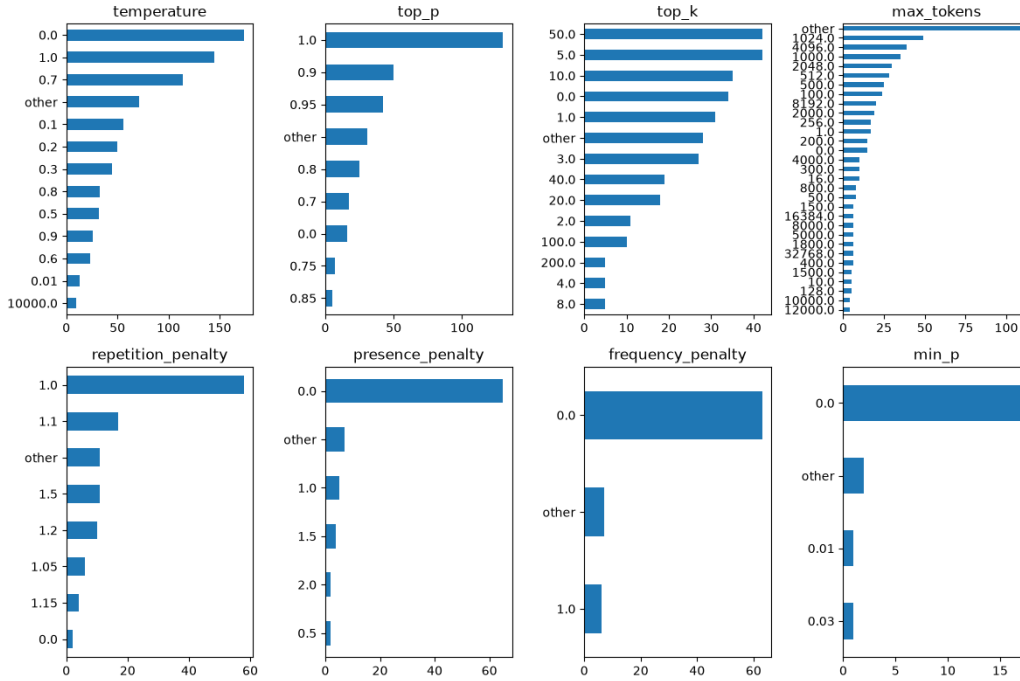
■ **Table 2** Parameter groups defined in projects. Number and percentage of projects that define each group of parameters.

# Defined parameters	Defined parameters	# Projects	Projects (%)
1	temperature	59	70.24%
	max_tokens	17	20.24%
	top_k	5	5.95%
	repetition_penalty	2	2.38%
	top_p	1	1.19%
2	max_tokens, temperature	55	59.78%
	temperature, top_p	14	15.22%
	temperature, top_k	12	13.04%
	max_tokens, top_k	6	6.52%
	top_k, top_p	4	4.35%
	presence_penalty, temperature	1	1.09%
3	temperature, top_k, top_p	21	31.82%
	max_tokens, temperature, top_k	18	27.27%
	max_tokens, temperature, top_p	13	19.70%
	repetition_penalty, temperature, top_p	10	15.15%
	max_tokens, repetition_penalty, temperature	3	4.55%
	repetition_penalty, temperature, top_k	1	1.52%
4	max_tokens, temperature, top_k, top_p	19	38.78%
	repetition_penalty, temperature, top_k, top_p	13	26.53%
	frequency_penalty, presence_penalty, temperature, top_p	8	16.33%
	max_tokens, repetition_penalty, temperature, top_p	4	8.16%
	max_tokens, min_p, temperature, top_p	1	2.04%
	max_tokens, repetition_penalty, temperature, top_k	1	2.04%
	max_tokens, presence_penalty, temperature, top_p	1	2.04%
	max_tokens, frequency_penalty, temperature, top_p	1	2.04%
	max_tokens, frequency_penalty, temperature, top_k	1	2.04%
5	frequency_penalty, max_tokens, presence_penalty, temperature, top_p	15	50.00%
	max_tokens, repetition_penalty, temperature, top_k, top_p	8	26.67%
	frequency_penalty, max_tokens, temperature, top_k, top_p	2	6.67%
	max_tokens, presence_penalty, temperature, top_k, top_p	1	3.33%
	frequency_penalty, presence_penalty, temperature, top_k, top_p	1	3.33%
	max_tokens, presence_penalty, repetition_penalty, temperature, top_p	1	3.33%
	max_tokens, min_p, temperature, top_k, top_p	1	3.33%
	frequency_penalty, max_tokens, presence_penalty, repetition_penalty, temperature, top_k	1	3.33%
6	frequency_penalty, max_tokens, presence_penalty, temperature, top_k, top_p	11	61.11%
	frequency_penalty, presence_penalty, repetition_penalty, temperature, top_k, top_p	3	16.67%
	max_tokens, presence_penalty, repetition_penalty, temperature, top_k, top_p	2	11.11%
	max_tokens, min_p, repetition_penalty, temperature, top_k, top_p	1	5.56%
	frequency_penalty, max_tokens, presence_penalty, repetition_penalty, temperature, top_p	1	5.56%
7	frequency_penalty, max_tokens, presence_penalty, repetition_penalty, temperature, top_k, top_p	11	68.75%
	frequency_penalty, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p	2	12.50%
	max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p	1	6.25%
	frequency_penalty, max_tokens, min_p, presence_penalty, temperature, top_k, top_p	1	6.25%
	frequency_penalty, max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_p	1	6.25%
8	frequency_penalty, max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p	1	100.00%

Findings (RQ2).

Projects typically define a small set of parameters (mean ≈ 3), and it is uncommon to find repositories that configure more than five parameters. This points to a prevalent *minimalist* configuration practice. Furthermore, across projects, the most common groups nearly always include *temperature*, frequently paired with *max_tokens* and either *top_p* or *top_k*. As the number of defined parameters increases, groups tend to expand around this core set.

RQ3. What values do developers assign to the LLM parameters they define? In this research question, we are interested in understanding the specific values that developers define for each parameter. To do this, we obtain the different values that are defined for each parameter in each project. Figure 3 shows the defined values for each parameter. To focus on dominant choices, we display the most frequent values that jointly account for 90% of the observations for that parameter and aggregate the remainder into an “other” bin.



■ **Figure 3** Parameters values. Each sub-figure corresponds to one of the parameters where each bar represents the number of projects that define the indicated value for the parameter.

The distributions reveal that developers tend to adopt a relatively narrow band of values for the most frequently defined parameters. Regarding temperature, we observe that the values 0.0 and 1.0 were frequently defined, followed by 0.7. Temperatures in the ranges 0.1-0.6 and 0.8-0.9 being less frequent, suggest a general preference for deterministic results in some projects and more creative results in others. For *top_p*, values higher than 0.9 dominate, indicating that most developers rely on a standard nucleus-sampling threshold rather than fine-grained tuning. Similarly, *top_k* values tend to fall within a small range (0–50), which suggests an emphasis on moderate vocabulary restriction. For *max_tokens*, the most common settings appear in a set of rounded values, likely reflecting conventional generation lengths or example defaults from popular libraries. *Presence_penalty* and *frequency_penalty* are most often defined at 0.0 while *repetition_penalty* is frequently defined at 1.0. The *min_p* parameter is defined in very few projects and is frequently set to 0.0.

Findings (RQ3).

Developers converge on a narrow set of values for key parameters. *Temperature* is most often set to 0.0, 0.7 and 1.0, while *top_p* clusters around 0.9 and *top_k* around moderately small values. Common choices for *max_tokens* correspond to values of the form 2^k , suggesting conventional or template-driven usage. In general, the penalty parameters are configured to penalize repetition, presence, and frequency only slightly. Overall, the distributions reveal emerging *defaults-in-practice* for high-frequency parameters and scattered exploration for less commonly used ones.

5 Threats To Validity

Internal validity. Our results depend on the correct extraction of LLM parameter definitions from Python source code using AST-based static analysis. Project-specific wrappers, dynamic configuration patterns, aliasing of parameter names, or indirection through configuration files may lead to false negatives or, less frequently, false positives. In addition, parameters defined conditionally or overridden at runtime may not be captured by our static approach. To mitigate these risks, we adopted conservative extraction rules, excluded non-production paths, retained full provenance (project, file, and line) for each extracted instance, and performed randomized spot checks on the dataset.

Construct validity. We operationalize a parameter as being *defined* when an explicit value assignment appears anywhere in a repository, and we interpret parameter groups as sets of parameters co-defined at the project level. This construct does not imply that parameters are used together in a single LLM invocation, nor does it capture dynamic overrides, environment-driven defaults, or provider-side implicit defaults. We address this threat by aligning our research questions, analysis, and interpretations strictly with project-level definitions, and by explicitly avoiding claims about runtime behavior or causal effects.

External validity. Our corpus consists of open-source Python projects hosted on GitHub, selected using topic filters and a popularity threshold. This sampling strategy may bias the dataset towards active, visible repositories and may under-represent private, industrial, or less popular projects, as well as ecosystems using other programming languages. We mitigate this threat by making all selection criteria explicit and grounding them in prior MSR literature that uses stars as a proxy for maturity and visibility.

Conclusion validity. Our analyses are descriptive in nature and should not be interpreted as causal evidence about configuration effectiveness, output quality, or developer intent. Given the number of parameters and value choices inspected, some patterns may appear salient by coincidence, particularly for infrequently used parameters. To reduce this risk, we emphasize high-support patterns, report complete tables and distributions, avoid causal language, and interpret results conservatively.

6 Conclusion

This paper provides a repository-based view of how developers *define* LLM parameters in real-world software projects. Using AST-based extraction over open-source Python repositories, we addressed three questions on prevalence (RQ1), configuration complexity and its structure (RQ2), and value distributions (RQ3). We find that *temperature* is the parameter most frequently defined, with *max_tokens* and *top_p* also common, while penalty-style parameters are comparatively rare. Projects typically define few parameters (mean ≈ 3), the most frequent project-level defined parameter groups consistently include *temperature* with *max_tokens* and either *top_p* or *top_k*, and value distributions suggest emerging *defaults-in-practice* (e.g., *temperature* $\sim 0.0, 0.7$ and 1.0 , *top_p* ~ 0.9 , moderately small *top_k*, and rounded *max_tokens* such as 256/512/1024). These descriptive results make configuration practices visible and measurable, supporting clearer reporting and reproducibility in LLM-in-SE studies, and offering practical baselines for tools and education.

6.1 Future Directions

This section describes the future lines of research that stem from the findings presented in this paper.

Understanding *why* developers make these choices. Our evidence characterizes *what* is defined, but not the underlying rationale. Future work should investigate the *causes* behind observed practices, including (a) developer intent and mental models (*e.g.*, interviews/surveys with practitioners), and (b) the role of examples and library templates (*e.g.*, LangChain/Ollama/vLLM code templates).

Longitudinal evolution of configurations. Another promising direction is to study *how* configurations evolve over time within each project. Mining commit histories and release tags can reveal adoption curves (*e.g.*, when *top_p* first appears), churn (frequency of parameter changes), stability of value ranges, and responses to upstream API changes.

Task- and domain-specific patterns. Future work can consider task inference (*e.g.*, RAG³, agents, evaluation, TTS⁴/ASR⁵, vision) and quantify whether certain tasks are associated with distinctive configuration profiles (*e.g.*, lower *temperature* for evaluation pipelines, higher *max_tokens* for drafting/long-form generation).

From “defaults-in-practice” to outcomes. Distributions suggest conventional bands (*e.g.*, *temperature* $\sim$ 0.0, 0.7 and 1). A key extension is to relate these bands to *outcomes*: output quality, diversity, stability, cost/latency. Controlled experiments on representative tasks and models – using values mined from repositories as treatments – can test whether defaults-in-practice are well-calibrated, suboptimal, or context-dependent.

Data Availability Statement

The dataset supporting this study, including the mining scripts, raw repository data, and analysis notebooks, is openly available on Zenodo at <https://doi.org/10.5281/zenodo.21456738> under a CC-BY 4.0 license. The replication package contains all materials necessary to reproduce our findings.

References

- 1 Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. LLM-Based chatbots for mining software repositories: Challenges and opportunities. In *Proceedings of EASE 2024*, 2024. URL: <https://das.encs.concordia.ca/pdf/abedu2024llm.pdf>.
- 2 Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. Guidelines for Empirical Studies in Software Engineering involving Large Language Models, 2026. [arXiv:2508.15503](https://arxiv.org/abs/2508.15503).

³ Retrieval-Augmented Generation

⁴ Text-to-Speech

⁵ Automatic Speech Recognition

- 3 Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016.
- 4 Vincenzo De Martino, Joel Castaño, Fabio Palomba, Xavier Franch, and Silverio Martínez-Fernández. A Framework for Using LLMs for Repository Mining Studies in Empirical Software Engineering. In *2025 IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering (WSESE)*, pages 6–11, 2025. doi:10.1109/WSESE66602.2025.00008.
- 5 Nikita Gladkikh. The impact of configuration management on the reliability of LLM pipelines. *IJSES*, 2025.
- 6 Thomas Klebel, Vincent Traag, Ioanna Grypari, Lennart Stoy, and Tony Ross-Hellauer. The academic impact of open science: a scoping review. *Royal Society Open Science*, 12(3):241248, March 2025. doi:10.1098/rsos.241248.
- 7 Pranjal Kumar. Large Language Models (LLMs): survey, technical frameworks, and future challenges. *Artificial Intelligence Review*, 57(10):260, 2024. doi:10.1007/S10462-024-10888-Y.
- 8 Let’s Data Science Team. LLM Sampling: Temperature, Top-K, Top-P, and Min-P Explained, 2026. URL: <https://letsdatascience.com/blog/llm-sampling-temperature-top-k-top-p-and-min-p-explained>.
- 9 Agustin Olmedo, Jannik Laval, Christelle Urtado, and Sylvain Vauttier. Parameterizing LLMs in Practice: An Empirical Study of LLMs Integrated into Software Systems, July 2026. doi:10.5281/zenodo.21456738.
- 10 OpenAI. OpenAI API reference: Chat completions. <https://developers.openai.com/api/reference/resources/chat/subresources/completions/methods/create>, 2026. Accessed March 2026.
- 11 Selva Prabhakaran. LLM Temperature, Top-P, and Top-K Explained with Simulations. <https://machinelearningplus.com/gen-ai/llm-temperature-top-p-top-k-explained/>, 2026. Accessed March 2026.
- 12 Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, Hyojung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncareenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2025. arXiv:2406.06608.
- 13 Sebastian Simon, Nikolay Kolyada, Christopher Akiki, Martin Potthast, Benno Stein, and Norbert Siegmund. Exploring hyperparameter usage and tuning in machine learning research. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 68–79, 2023. doi:10.1109/CAIN58948.2023.00016.
- 14 Mohamed Soliman, Michel Albonico, Ivano Malavolta, and Andreas Wortmann. Mining Software Repositories for Software Architecture: A Systematic Mapping Study. *Information and Software Technology*, 181:107677, 2025. doi:10.1016/J.INFSOF.2025.107677.
- 15 Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. LLMs love python: A study of LLMs’ bias for programming languages and libraries. *arXiv e-prints*, pages arXiv–2503, 2025.
- 16 Stefan Wagner, Marvin Muñoz Barón, Davide Falessi, and Sebastian Baltes. Towards Evaluation Guidelines for Empirical Studies involving LLMs. In *Workshop on Empirical Software Engineering (WSESE 2025)*, 2025. URL: <https://assets.empirical-software.engineering/pdf/wsese25-llm-guidelines.pdf>.
- 17 Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of Large Language Models in Software Engineering tasks. *Empirical Software Engineering*, 30(50), 2025. doi:10.1007/S10664-024-10602-0.