# Observational Study on usage of Large Language Models in public Github repositories

Davide Zhang[1] and Roberto Magrini[1] *

[1]Scuola di Scienze Matematiche Fisiche e Naturali, UNIVERSITÀ DEGLI STUDI FIRENZE

**Abstract**

*The increasing integration of Large Language Models (LLMs) in software engineering, both as development tools and as components of deployed systems, raises the need to understand how developers adopt these technologies in real-world projects. This study analyzes a large set of public GitHub repositories to identify which LLM technologies they employ and how their usage correlates with other project characteristics. We combine repository metadata with code-search based mining to detect both the SDKs and the models used, enabling an examination of technology popularity, application domains, and common SDK–model combinations. Our dataset also includes monthly snapshots of the most popular repositories from 2022 to 2025. Using statistical analysis, we investigate trends in LLM adoption and explore relationships between the presence of LLM components and repository attributes such as size and activity. The results provide an empirical characterization of the current LLM landscape in open-source software development.*

## Introduction

Our objective is to thoroughly analyze the usage and prominence of LLMs in GitHub repositories. Many different languages can be used to do this analysis, but we decided to only consider the following languages:

- **Python**, the most popular language for LLM inference through programmatic API.
- **Java**, a popular high-level language for the development of web applications.
- **Go**, similar to Java but with a simpler syntax.

We also decided to only consider the official SDK provided by the following makers:

- **OpenAI**
- **Anthropic**
- **Mistral**
- **Google**
- **xAI**
- **Meta**

The list of available models for each maker was obtained by visiting the official site and combining results there with the return of a query to the API of the maker when the call was free.

To reach our goal, we first analyzed for each language the evolution of the percentage of repositories containing LLM-adjacent words in their topic, description, or title. From the data we collected, we performed several analyses, such as which language is the most popular for the development of LLM-integrated applications or which repository metrics are more correlated. Starting from this analysis, we conducted a deeper research on public GitHub repositories by using code search to retrieve all instances of relevant SDK import or model reference. The collected data is combined to obtain SDK-model pairs that can be analyzed and used to derive insights for our study. We then sampled a set of representative repositories from this exhaustive list to apply statistical methods and infer properties of the sampled population.

Our empirical analysis leads to several key findings:

- **Rapid growth of LLM-related projects**, since the introduction of LLMs to public usage in 2022 the proportion of LLM-related repositories among the most-starred projects increased significantly across all languages, with *Python* showing the steepest rise, from a 30% to a stable 80% reached in 2023 (Figure **1**) . The other languages also follow a similar upward trend but not as steep.
- **Python's dominance over LLM-integrated development**, *Python* accounts for over 260,000 repositories, compared to 8,639 for *Java* and 5,511 for *Go* in our combined dataset, making it two orders of magnitude more widely used for LLM applications.
- **OpenAI and Google SDKs are the most adopted among official libraries**, although third-party libraries remain popular across all languages. For *Python* (Figure **6**), official SDKs show much stronger adoption than in *Java* (**Figure 7**) and *Go* (Figure **8**), where third-party libraries remains the most utilized.
- **A small set of LLM models accounts for most real-world usage**, Across languages, the top 20% most-frequent models constitute over 80% of

*Emails: davide.zhang@edu.unifi.it roberto.magrini@edu.unifi.it

model references, with *OpenAI* models dominating and *Google* models appearing as secondary but significant contributors (Figures **9 10 11**).
- **Common usage patterns align with conversational, generative, and agentic workflows**, Based on keyword-based classification, LLMs in public repositories are primarily used for conversational agents, content generation, and multi-step agentic orchestration, with fewer repositories employing retrieval-augmented pipelines or domain-specific applications (Figures **15 16 17**).
- **Repository-level attributes show interesting correlations**, Spearman correlation analyses reveal distinct patterns across languages. In *Python*, open issues correlate strongly with fork counts (Figure **18**), while in *Java* and **Go** stars correlate more with open issues (Figures **20 19**). Statistical hypothesis testing further shows that *Python* LLM repositories have significantly fewer stars on average than *Java* and *Go* repositories ($p < 0.05$ in both one-tailed Welch t-tests), suggesting lower visibility or more experimental project character.

These findings paint a clear picture of the current LLM ecosystem where *Python* is the central hub of LLM-integrated development, *OpenAI* remains the dominant provider, and real-world usage tends to use a relatively narrow set of models and application patterns.

The remainder of this paper is organized as follows:

- **Section 2, Related Work**: Presents the related work, positioning our study within the broader landscape of LLM-integrated software engineering research.
- **Section 3, Experiment Design & Execution**: Details our experimental design, outlining the research questions and the metrics used to answer them, and describes the execution of the experiment, including data collection strategies for keyword-search, library-search, model-search, and repository attribute extraction.
- **Section 4, Data Processing & Cleanup**: Discusses the preprocessing and cleaning steps applied to the dataset.
- **Section 5, Results**: Reports the results for each research question, covering popularity trends, language and SDK distribution, model usage analysis, usage patterns, and repository attribute correlations.
- **Section 6, Threats to Validity**: Describes the various threats that are present in this research and provides some suggestions on how to mitigate them.
- **Section 7, Discussion**: Provides a discussion and interpretation of the results, outlining limitations and implications for future research.
- **Section 8, Conclusion**: Concludes the study with a summary of our contributions and opportunities for follow-up investigations.

## Related Work

The existing literature already includes numerous empirical studies covering a wide range of topics related to Large Language Models (LLMs), such as exploring new domains of application (e.g., assisting in peer review [**zhou-etal-2024-llm**], code understanding [**10.1145/3597503.3639187**], and LLM-as-a-judge scenarios [**gu2025surveyllmasajudge**]), as well as studies focused on improving their performance [**NEURIPS2024-71c3451f**].

However, to our knowledge, there have not been many studies on the popularity of different LLM technologies in LLM-integrated software. An important study on the topic of LLM-integrated software [**weber2024largelanguagemodelssoftware**] attempts to establish new terminology, concepts, and methods for LLM-integrated application engineering. LLM-integrated application engineering refers to an emerging branch of software engineering that studies LLM-integrated applications, which are software systems that leverage LLMs to provide functionalities that would otherwise be infeasible or require substantial development effort to implement. To this end, the study proposes a structured framework for categorizing and analyzing LLM-integrated applications across various domains.

Another paper on the same topic [**10.1145/3715007**] explores LLM-integrated application engineering from the perspective of its challenges. These challenges span different areas, such as prompts, APIs, and plugins, and require developers to navigate unique methodologies and considerations specific to LLM application development. The authors crawled and analyzed approximately 30,000 relevant questions from popular OpenAI developer forums to build a taxonomy of challenges faced by LLM developers and summarized a set of findings and actionable insights.

In summary, we did not find important papers detailing the popularity of various LLM technologies through API usage, but we did find foundational papers on LLM-integrated application development.

# Experiment Design And Execution

## Experiment Design

The goal of this study is to investigate the real-world adoption and usage patterns of Large Language Models (LLMs) in public software projects. To achieve this goal, we define the following research questions:

1. How has the popularity of LLM-integrated software evolved since the first announcement of ChatGPT in 2022?
2. What is the most popular programming language for LLM-integrated software development?
3. What is the most popular SDK for LLM-integrated software development?
4. Which LLM models are most frequently used for inference in public GitHub projects?
5. How are these models being utilized in practice within these repositories?
6. What is the average size, number of stars, forks etc. for a LLM-integrated application and how are they correlated?

To answer these questions, we define the following metrics and data collection strategies:

- **Popularity over time** We will track the presence of LLM-related keywords in repository descriptions, topics, or names among the most-starred public GitHub repositories. This will be done monthly from January 2022 to October 2025. We will calculate the proportion of repositories that match these criteria each month and plot the evolution over time. As result we will build a graph representing the evolution of the proportion of LLM-related repos in the given time interval.
- **Popular programming languages** We will identify repositories that include SDK imports or model inference code and aggregate them by programming language. The language with the highest number of such repositories will be considered the most popular for LLM-integrated software development. This requires a method to query and retrieve all relevant repositories effectively (chunking).
- **Popular SDKs** For each SDK, we will count the total number of repositories that include its import. The SDK with the highest occurrence will be identified as the most widely adopted.
- **Popular LLM models** We will analyze the repositories to detect which models are referenced most frequently, based on model names or API calls for inference.
- **Usage patterns of LLMs** We will perform a qualitative analysis of how models are used in these repositories, categorizing usage into common patterns: Conversational, Generation task based, Retrieval-Augmented, Agentic Orchestration, Multimodal or Domain-Specific.
- **Repository attribute patterns** By randomly sampling five hundred repositories and querying the GitHub endpoint for specific metadata, we hope to estimate through statistical methods the mean of various attributes of LLM-integrated sofwtare like its size or its number of stars. We will lastly produce a table expressing the correlation between these different stats.

By systematically measuring these metrics, we aim to quantify the adoption trends, identify the most popular development practices, and understand how LLMs are integrated into real-world software projects.

## Experiment Execution

The experiment will mainly consist of four phases:

1. **Keyword-Search** –We will use the repository search endpoint to keep track of the popularity metric.
2. **Library-Search** –We will use the code search endpoint to retrieve all occurrences of repositories with corresponding imports.
3. **Model-Search** –We will use the code search endpoint to retrieve all occurrences of repositories with corresponding reference to model names.
4. **Attribute-Search** –Starting from the complete list of all LLM-related repositories obtained previously, we sample and then query the repository endpoint for detailed repo attributes.

We subsequently report the considerations we made for each phase, the limits, the execution processes, and the preliminary results.

**Data Collection for Keyword-Search** To have a rough estimate of the popularity of LLM projects in GitHub for various languages, we first did a monthly analysis on the one hundred most popular repositories created that month and memorized useful metadata like number of stars, forks, topics, description, etc. This analysis went from January 2022, the year in which ChatGPT was launched, to October 2025, when this study began. The code needs a valid GitHub token for authentication, which can be created from the settings page in your GitHub account. When run, the script (repo_search.py) queries GitHub for the most popular repositories created each month, retrieving interesting metadata and writing to a JSON file that serves as storage. The data collection process for this step did not take much time since with one request, we could collect all the repos we needed for a month.

**Setting up Library-Search** The first step is to gather the names and URLs of the GitHub repositories that import libraries of the chosen LLM providers, but we need some boundaries to the domain of the repositories; otherwise, the scale of data to analyze would be too much and would hinder the research. We settled on size boundaries. The domain of this study comprises all the GitHub repositories with files whose size is less than 5MB, plus language constraints. Furthermore, we decided to focus on the most mainstream, accessible, and easily implementable LLMs; this was done because not every LLM has an official or a reputable and endorsed third-party SDK, while other LLMs are not easily accessible, with their documentation and models being put behind a paywall. So for these reasons, we decided that our study will focus on *OpenAI*, *Gemini*, *Anthropic*, *Minstral*, *xAI*, and *Meta*. The former two LLMs are only counted for specific languages; for example, *xAI* only has an official SDK for *Python* and none for the other languages. More importantly, there aren't any reputable third-party libraries, neither for *Java* nor *Go*, and so we have analyzed the usage of *xAI* only on *Python* files; the same goes for *Meta*.

Since the GitHub Rest API does not support code search based on regular expressions, we decided to do a word list-based research. The word list was built for each language by looking at its import syntax and creating a list of every possible way to import the LLM client. For example, in the case of Python, there are two major import syntaxes for the OpenAI client:

- **import openai.OpenAI as alias**
- **from openai import Openai**

Since GitHub code search does not distinguish between upper and lowercase letters, we can cover both cases by searching for "import openai". Lastly, for the data collection script to work, we need a Python dictionary that associates each word with a label that will be applied to the repository. This dictionary can be easily built manually by associating each library keyword with the providing company.

**Setting up Model-Search** Other than the LLM library used in a project, we are also interested in the specific LLM model that the LLM-integrated application is interfacing with. To do this, we create a starting list of the most important models for each provider by visiting the models and pricing page on their official site. This list is then enriched by combining it with the results of an API call querying the complete list of available models for each provider (when the list operation is available). model-search, just like library-search, uses the same language scope for code mining. This means only for *Python*, *Java* and *Go* files and only for models that can be used by official SDK provided by *OpenAI*, *Gemini*, *Anthropic*, *Minstral*, *xAI*, or *Meta*.

Just like for Library-Search, Model-Search is implemented as code search based on word matching. To avoid being blocked by models with very short names (like o1), we search for \"word\" instead of simply searching for word in a script file. The keyword dictionary, in the case of Model-Search, cannot be simply created manually, given its higher length. We will need to build it in a separate script by simply associating each model keyword with the name of the model.

**Data Collection for Library-Search and Model-Search** The script (mine.py) employs a dynamic chunking strategy based on size to be able to retrieve all occurrences of files satisfying certain queries. This query can be based on model names for Model-Search or on imports for Library-Search. A dedicated part of the script is used to manage rate-limiting from GitHub by pausing the execution each time we hit it until the next reset time. We query the endpoint a page at a time, with each page containing one hundred items, until we finish all results. If we get one thousand or more results, we stop and launch an exception because we could not retrieve all occurrences of the searched pattern. Lastly, based on the number of results, we adjust the size interval so that we can go faster when the queries are mostly empty, while not reaching the thousand cap when the results are dense.

To complete the data collection step, we need to run the same function (collect_repo_by_language) twice for each language: in the first run, we search for library imports, while in the second run for model names. Given the objective of retrieving everything and the long list of models to search for, this data collection step was the most time-consuming of all, needing almost a week to complete.

**Data Collection for Attribute-Search** Starting from the list of collected repos we randomly sample five hundred repositories and query the GitHub endpoint for attributes of interest. The script is quite simple, and it follows more or less the same logic outlined by Keyword-Search. The github_get function allows for managing authentication errors and rate-limiting while repeating the request in case of an unstable internet connection. The execution process did not take much time, and the only error we encountered was the 404 not found error. This error is probably caused by deleted projects or projects that have been made private since the execution of Model and Library-Search. To avoid this error, we just repeat the sampling process multiple times until we reach a run without the error, which should not take too many

runs since this error is rare (you encounter one such repo in five hundred).

## Data Preprocessing and Cleanup

In this section, we discuss how we preprocessed and cleaned up the collected data from the previous steps.

**Data Preprocessing for Keyword-Search** The collected Data for Keyword-search is already in a format that can be worked with. Since we only need to use the topics, description, and name fields, even if some of them are null (description or name), the wordlist matching process would not change. The collected data is memorized in a JSON file as a list of dicts containing various useful repository metadata. This JSON file could then be loaded into a Python list and worked with.

**Data Preprocessing for Library-Search and Model-Search** The data collected by the mining script is stored in a JSON file format, containing a list of dictionaries. The dicts contain the name of the repository, URL, HTML that is a link to the human-readable page of the project, and a label dict. The label dict contains, for each label (the keyword that was found), a list of file paths. Each file path represents the location where the keyword was found. We are interested in knowing the relationship between model data and the library data, so we need to first combine the two data dictionaries before any further actions.

The combine function combines the two JSON files to derive data about Model-Library correlation. First, we consider the repositories appearing in both lists: these repositories contain both the library import and the model reference, and as such, we can label them with both. In the case where the import and the model reference are in the same file (we also remember the filepaths), we can pair the two labels up. If we only found the import in a repository, we label it with the name of the provider, while if we only found a reference to a model, we label it with unknown library. In this way, we obtained a list of repositories with informative labels that could then be used in the analysis phase.

This step is done by running the combine function in the combine.py file. A special dict called model-library map is required for the script to be able to match models to the correct SDK. We built this Python dictionary starting from the previous list of models by creating a key for each model and associating it with an empty list of strings, which is then updated by appending possible parent SDKs. In this way, only correct SDK-Model matchings will be created in the final JSON file.

**Data Preprocessing for Attribute-Search** Like Keyword-Search, the data collected with Attribute-Search can already be worked with without much hindrance. The data is collected as a list of Python dictionaries, containing for each attribute-key a value that can be null. Everything is then stored in a JSON file, like always.

# Results

In this section of the report, we show the results of our study and our answer to each of the initial questions concerning the real-world adoption and usage of LLM technology.

## Popularity over time

We respond to this question by analyzing the results collected from Keyword-Search. We analyzed in total five languages: *Python*, *Java*, *C#*, *JavaScript* and *Go*. We analyzed the name, the description, and the topics for each repository and classified a repository as LLM-related if it contained at least a word from our list of keywords. We then just show the proportion of LLM-related projects grouped by the month of creation of the project. In Figure **1** we can observe

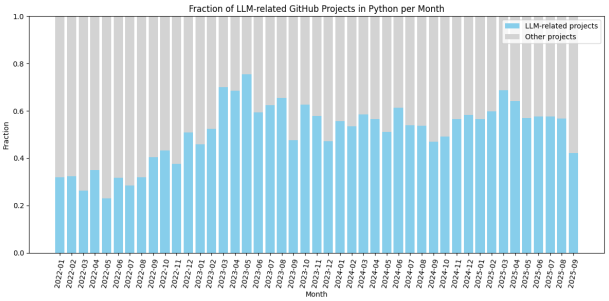


**Figure 1:** *monthly proportion of Public Python projects with LLM-related terms in their description, name or topics*

that the proportion of LLM-related terms in the most starred public GitHub Python projects, started from a level of 30% in 2022 and increased until a stable 80% in the first half of 2023. We can see that a majority of the most popular new Python projects were about AI and LLMs in the last three years.

For Java, in Figure **2**, we observe a lower incidence of LLM-related terms appearing in the most starred recent projects, however we can still see an evident increase in popularity in the last two years.

For Golang and Javascript GitHub projects, Figures **4** and **5**, we see higher proportions of LLM-related projects and a similar peak in the may of 2025.

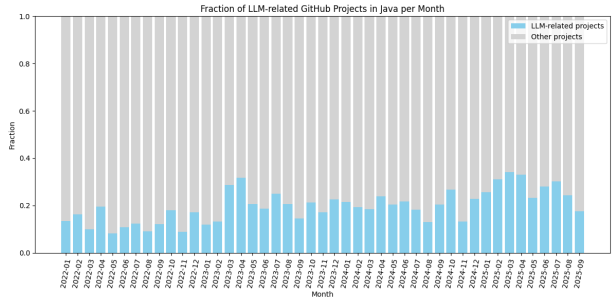


**Figure 2:** *monthly proportion of Public Java projects with LLM-related terms in their description, name or topics*

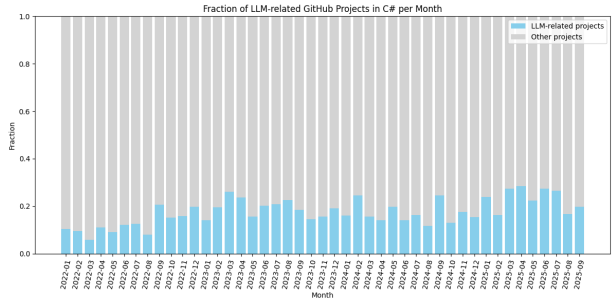


**Figure 3:** *monthly proportion of Public C# projects with LLM-related terms in their description, name or topics*

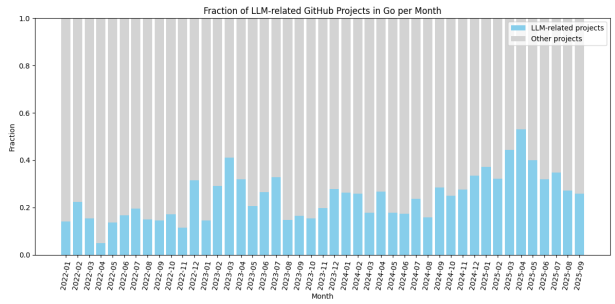


**Figure 4:** *monthly proportion of Public Go projects with LLM-related terms in their description, name or topics*

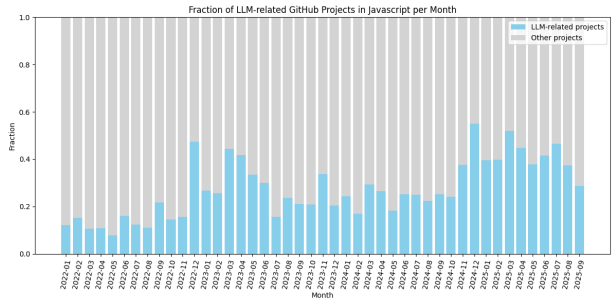


**Figure 5:** *monthly proportion of Public Javascript projects with LLM-related terms in their description, name or topics*

## Popular programming languages

To respond to this question we just compare the number of repositories we collected for each language in Model-Search and Library-Search.

1. Number of repositories in the combined JSON file for Python: 262581
2. Number of repositories in the combined JSON file for Java: 8639
3. Number of repositories in the combined JSON file for Golang: 5511

From the collected data we can see that Python is two order of magnitude more used for LLM-integrated application development than the other two analyzed languages.

## Popular SDKs

Each of the selected provider of LLM technology offers its own SDK to interface smoothly with its services. Like what we have already said in the experiment design section, official SDKs exist only for the most popular languages. We have built three graphs, one for each of the three languages we collected data for (Figures **6**, **7**, **8**). Each graph will show the number of repositories that were found with the specific import statement that links it to the corresponding official SDK. Lastly, if a model reference was fopund in a repository with no known import pattern then the repository will be linked to 'other', in other words to a third party SDK.

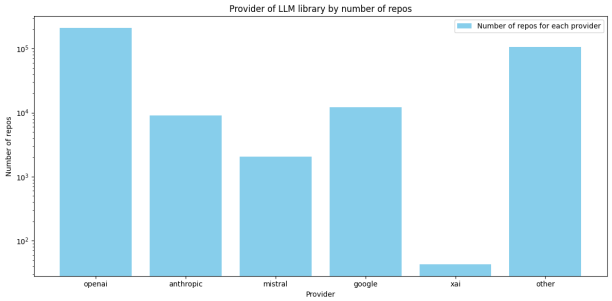


**Figure 6:** *Number of repositories found with code search with the imports of the relative SDK in Python*


**Figure 7:** *Number of repositories found with code search with the imports of the relative SDK in Java*

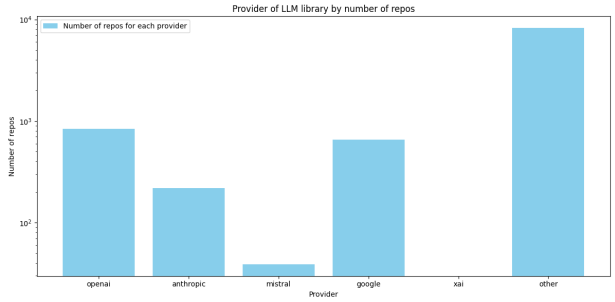


**Figure 8:** *Number of repositories found with code search with the imports of the relative SDK in Golang*

These three graphs allow us to conclude that third-party library remain very popular in all the languages, while among official SDKs, *OpenAI* and *Google* remain dominant, with *xAI* being the less used given that it does not have free models.

## Popular LLM models

Using the list of all models we derived previously with the list-model functionality for each provider, we count the number of references for each model in our dataset. In all three languages we see the trend of the 20% top most used models making up more than 80% of total model usage, suggesting that most developer use the most accessible and/or popular model (Figures **9**, **10**, **11**).

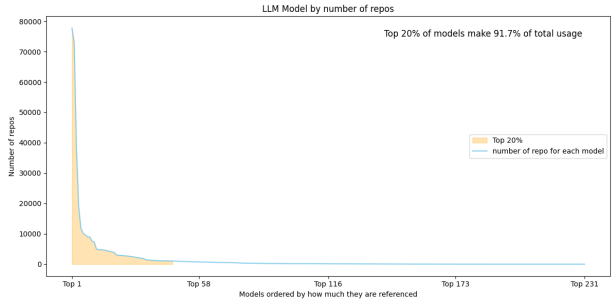


**Figure 9:** *Number of references for each model in decreasing order for Python repositories*

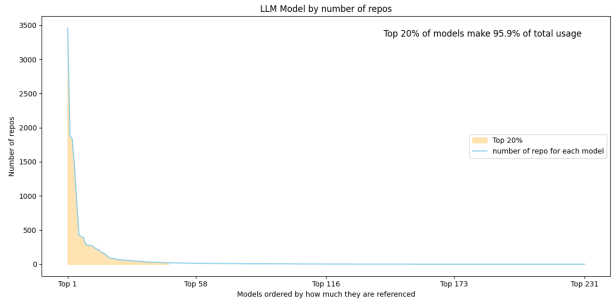


**Figure 10:** *Number of references for each model in decreasing order for Java repositories*

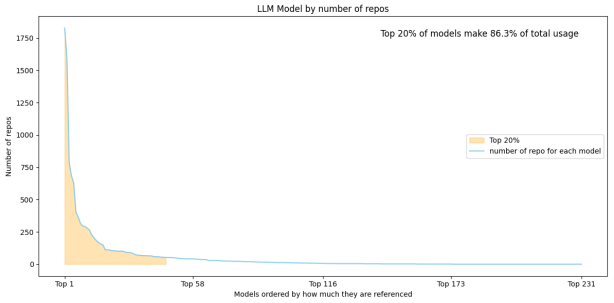


**Figure 11:** *Number of references for each model in decreasing order for Golang repositories*

In particular, for the twenty most referenced models, we conducted a deeper analysis to examine which libraries they were used in association with. Our method does not, in fact, guarantee that if an import statement for a library and a reference to a model appear in the same repository, or even in the same file, the model is actually used through that library. It only underlines association. Now we can show the graphs we made to show the association of various models with the five official SDKs we are considering (Figures **12**, **13**, **14**).

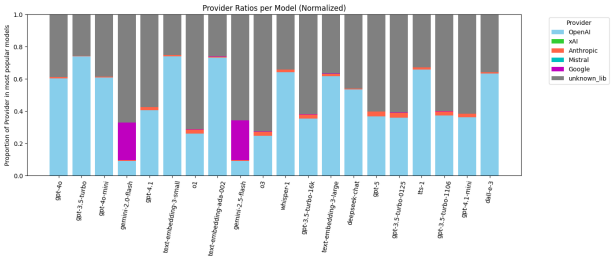


**Figure 12:** *Proportions of models found within the same file or repository of an import for Python projects*

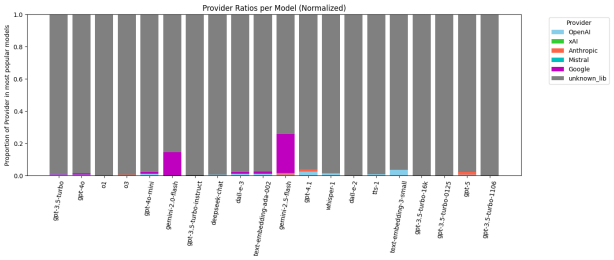


**Figure 13:** *Proportions of models found within the same file or repository of an import for Java projects*

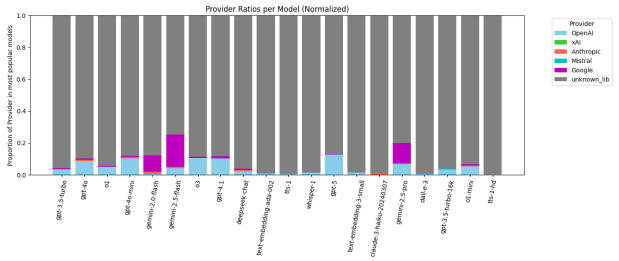


**Figure 14:** *Proportions of models found within the same file or repository of an import for Golang projects*

We see that our script was not able to associate most model references to a known official SDK in *Java* and *Go*, but was able to do so for the *Python* dataset. This may be caused by the higher popularity of official SDKs in Python and the higher popularity of *Python* itself for LLM-integrated software development. The most popular models do not change much between languages and are mostly dominated by *OpenAI* models with the occasional *Google* model. We can also observe that in correspondence of *Google* models we can see a significant proportion of association with the *Gemini* SDK, confirming the correctness of our method.

## Usage patterns of LLMs

Let us first discuss the classification method we decided to settle on:

1. **Conversational Agents** — Used for direct human-LLM interaction in dialog form (e.g., chatbots, virtual assistants, helpdesk systems).
2. **Generation Agents** — Used to produce text, code, or other content using LLMs (e.g., article writing, code completion, story generation).
3. **Retrieval-Augmented** — Uses external knowledge sources to enhance LLM outputs and improve correctness (e.g., document question answering, knowledge-base assisted generation).
4. **Agentic Orchestration** — Used when the LLM drives multi-step actions, calls APIs, or interacts with other software components (e.g., workflow automation, multi-step task agents, Auto-GPT style agents).
5. **Multimodal** — Handles multiple data types, such as text, images, audio, and video (e.g., image captioning, audio transcription, video summarization).
6. **Domain-Specific** — Tailored LLM applications designed for a particular field or domain (e.g., legal AI assistants, medical Q&A systems, financial analysis tools).

We based our classification method by matching the description, topics and name of the repository against a list of probable keywords for each category. This keyword matching method is not very efficient and leaves a lot of the repositories unclassified. Nevertheless we think that this method is a reliable way to classify the repositories based on purpose.

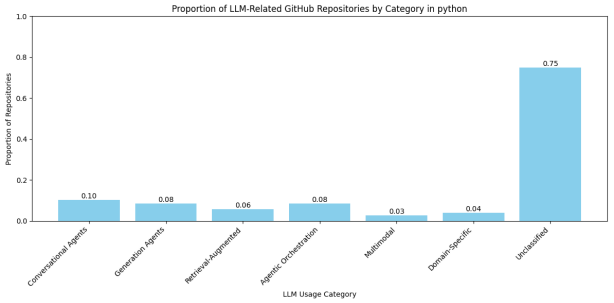


**Figure 15:** *Proportions of various LLM-integrated software category for Python*

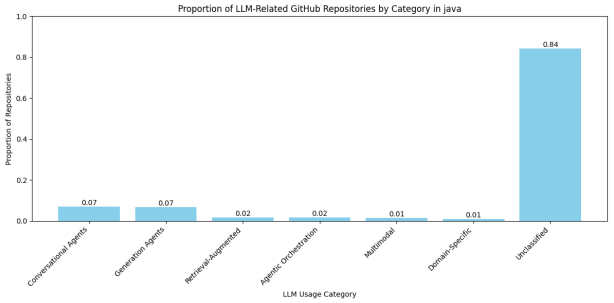


**Figure 16:** *Proportions of various LLM-integrated software category for Java*

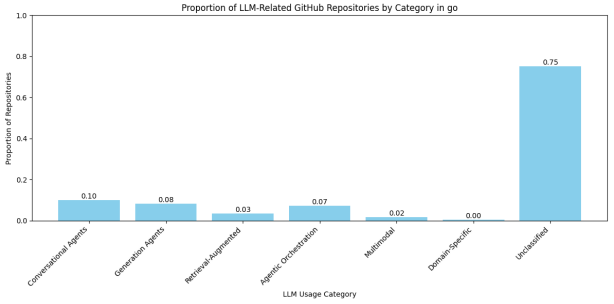


**Figure 17:** *Proportions of various LLM-integrated software category for Golang*

We can observe from the bar charts (Figures **15**, **16**, **17**) we built that between the classified repositories most llm are used as conversational agents, for generation tasks or for agentic orchestration.

## Repository attribute patterns

Lastly, we try to statistically analyze the attributes of an average LLM-integrated software on GitHub. Let us start by checking how correlated are the numeric attributes of the LLM-using GitHub repositories by computing the Spearman's rho for each pair.

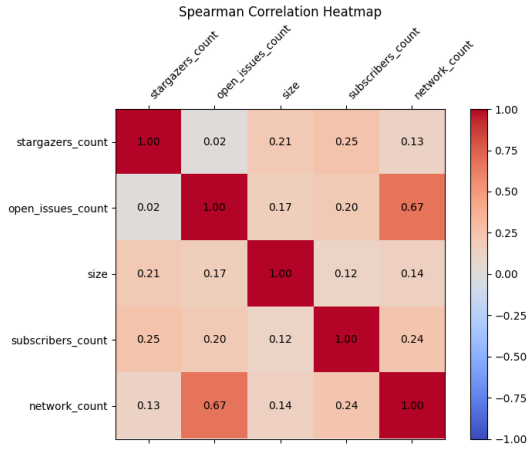


**Figure 18:** *Spearman's Correlation Heatmap for Python*

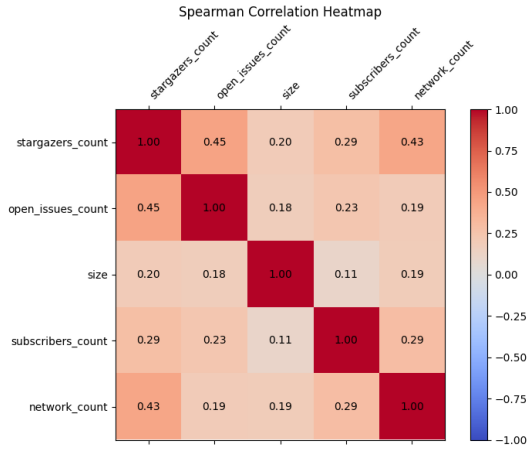


**Figure 19:** *Spearman's Correlation Heatmap for Go*

The Spearman's correlation matrices for the *Java* and *Go* datasets are very similar with each-other (Figures **20**, **19**), while the one built from the *Python* dataset is quite different (Figure **18**). The biggest observable difference is that for *Python* projects the number of open issues correlates the most with the network count (number of forks) while in the other two languages it correlates the most with the number of stars of a project. This suggests that *Python* LLM projects are less visible, with activity driven by

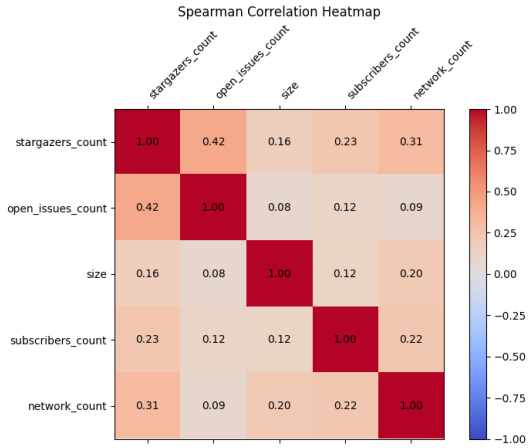


**Figure 20:** *Spearman's Correlation Heatmap for Java*

contributor engagement rather than popularity. This may be due to *Python*'s low barrier to entry, leading to many experimental projects with few stars. Let us Hypothesis test this.
First test (Python vs Java):

- $H_0$ — **Null Hypothesis**
  The mean number of stars in *Python* LLM Repositories is equal to the mean number of stars in *Java* LLM Repositories.
- $H_1$ — **Alternative Hypothesis**
  The mean number of stars in *Python* LLM Repositories is less than the mean number of stars in *Java* LLM Repositories.

Second test (*Python* vs *Go*):

- $H_0$ — **Null Hypothesis**
  The mean number of stars in *Python* LLM Repositories is equal to the mean number of stars in *Go* LLM Repositories.
- $H_1$ — **Alternative Hypothesis**
  The mean number of stars in *Python* LLM Repositories is less than the mean number of stars in *Go* LLM Repositories.

We randomly sampled 500 repositories for all three languages and took record of the star number for each repository. We then applied the Box-Cox transformation after shifting the number of stars by one to avoid zeros. The Box-Cox transformation is needed to stabilize variance and make the distribution more Gaussian and suitable for t-tests, transforming power law like distributions (e.g., the number of stars in GitHub repositories **[lima2014codingscalegithubcollaborative]**) into more normal shapes. We then apply Welch's t-test (one-tailed) to compute the t-statistics and the p-value for both Hypothesis tests assuming a significance level of $\alpha = 0.05$. One sided tests are reasonable here because we suppose that the number

of stars is lower in *Python* LLM projects.

Sample statistics before Box-Cox:

- $\overline{X}_{Python} = 18.99$
- $S^2_{Python} = 270.93^2$
- $\overline{X}_{Java} = 25.34$
- $S^2_{Java} = 179.75^2$
- $\overline{X}_{Go} = 188.41$
- $S^2_{Go} = 1986.66^2$

Sample statistics after Box-Cox:

- $\overline{X'}_{Python} = 0.173$
- $S'^2_{Python} = 0.268^2$
- $\overline{X'}_{Java} = 0.245$
- $S'^2_{Java} = 0.352^2$
- $\overline{X'}_{Go} = 0.393$
- $S'^2_{Go} = 0.506^2$

Lambda parameter for Box-Cox transformation:

- $\lambda_{Python} = -0.7517$
- $\lambda_{Java} = -0.6540$
- $\lambda_{Go} = -0.4958$

For different values of lambda the Box-Cox transformation applies different operations on the dataset. When $\lambda = 1$ then we have the original variable, when $\lambda = 0$ , we have the logarithm transformation, when $\lambda = -1$ we have the reciprocal transformation, and when $\lambda = 0.5$ we have the square root. Our $\lambda \approx$ ˇ0.7 indicates a transformation close to reciprocal-root behavior T-statistic and p-value for the two Welch t-tests:

- Python vs Java —
  $t = -3.6139$
  $p = 0.000159$
- Python vs Go —
  $t = -8.5982$
  $p = 0.000000$

Both values are lower than the significance level $\alpha = 0.05$ leading us to reject the null hypotheses $H_0$ for both comparisons. In other words, there is enough evidence to support that the mean number of stars for *Python* LLM projects are lower than those of *Java* and *Go* LLM projects. Thus, *Python* LLM repositories have significantly fewer stars, supporting the observation of lower visibility compared to *Java* and *Go* projects. A natural thing we can derive is that since *Python* repositories are generally less visible and more experimental, their size may be smaller than their corresponding *Java* and *Go* projects. The paper **[zhang2009distributionprogramsizesimplications]** supports a log-normal distribution of program sizes so we can use the same procedure we applied for the number of stars. First size test (Python vs Java):

- $H_0$ — Null Hypothesis
  The mean size of Python LLM Repositories is equal to the mean size of Java LLM Repositories.
- $H_1$ — Alternative Hypothesis
  The mean size of Python LLM Repositories is less than the mean size of Java LLM Repositories.

Second size test (Python vs Golang):

- $H_0$ — Null Hypothesis
  The mean size of Python LLM Repositories is equal to the mean size of Golang LLM Repositories.
- $H_1$ — Alternative Hypothesis
  The mean size of Python LLM Repositories is less than the mean size of Golang LLM Repositories.

Sample statistics before Box-Cox:

- $\overline{X}_{Python} = 24.19$
- $S^2_{Python} = 75.03^2$
- $\overline{X}_{Java} = 49.25$
- $S^2_{Java} = 356.95^2$
- $\overline{X}_{Go} = 31.98$
- $S^2_{Go} = 169.90^2$

Sample statistics after Box-Cox:

- $\overline{X'}_{Python} = 0.643$
- $S'^2_{Python} = 0.648^2$
- $\overline{X'}_{Java} = 0.900$
- $S'^2_{Java} = 0.817^2$
- $\overline{X'}_{Go} = 0.728$
- $S'^2_{Go} = 0.678^2$

Lambda parameter for Box-Cox transformation:

- $\lambda_{Python} = -0.5846$
- $\lambda_{Java} = -0.3503$
- $\lambda_{Go} = -0.4997$

T-statistic and p-value for the two Welch t-tests:

- Python vs Java —
  $t = -5.5173$
  $p = 0.000000$
- Python vs Go —
  $t = -2.0314$
  $p = 0.021239$

The statistical tests once again supports our reasoning. These findings support the hypothesis that Python LLM projects tend to be less prominent and smaller in scope. However, identifying the underlying causes, such as ecosystem norms, project maturity, or contributor profiles, requires further analysis.

# Threats to validity

When interpreting the results of this study, several potential threats to validity must be considered:

- External Validity:
  - Our analysis focuses exclusively on public GitHub repositories and three programming languages: *Python*, *Java*, and *Go*.
  - Projects in private repositories or written in other languages may exhibit different patterns of LLM usage, limiting generalizability.
  - Additionally, we only consider LLMs with official SDKs from six providers; thus, findings may not generalize to niche or proprietary LLMs.
- Internal Validity:
  - The reliance on keyword and import-based code searches may miss repositories that use unconventional naming, dynamic imports, or indirect API wrappers.
  - Rate-limiting and the GitHub search API cap (1000 results per query) might have caused incomplete data retrieval for highly popular SDKs or models.
- Construct Validity:
  - The classification of repositories into usage patterns (Conversational, Generation, etc.) is based on keyword matching in topics, description, and repository names, this method may misclassify or leave many repositories unclassified, potentially affecting the accuracy of usage pattern analysis.
  - Similarly, model-library association is inferred based on co-occurrence within a file or repository, which may not reflect actual usage.
- Conclusion Validity:
  - Statistical analyses rely on random samples of 500 repositories for hypothesis tests, which are only sampled one time.
  - While Box-Cox transformations help normalize skewed data, extreme values in popularity or size could influence results.
  - Correlations observed between repository attributes are descriptive and do not imply causation.

# Discussion

Our results provide several insights into the real-world adoption of LLMs in GitHub repositories. *Python* dominates LLM-integrated development, with two orders of magnitude more repositories than *Java* or *Go*, while *OpenAI* SDK and models are the most widely adopted across all languages. Other SDKs like *xAI* have minimal presence, likely due to limited accessibility. The usage patterns are primarily Conversational, Generation, and Agentic Orchestration, reflecting common application scenarios like chatbots, content generation, and workflow automation. We lastly concluded, through statistical analysis, that Python repositories tend to have fewer stars and smaller size, suggesting many experimental or prototype projects, whereas *Go* and *Java* projects are larger and more visible.

Our research is useful for LLM-integrated software developers: it helps in selecting LLM models and SDKs with the largest community support. For example, using *Python* with *OpenAI* SDK maximizes compatibility and access to widely tested models. As for lessons learned wefound that SDK accessibility influences greatly the language choice, while keyword-based classification is effective for high-level categorization but could be improved with NLP-based semantic analysis for more accurate usage pattern identification.

# Conclusions

This study offers a comprehensive observational analysis of LLM adoption in public GitHub repositories.

## Main Contributions

1. Quantified the evolution of LLM-integrated software popularity over 2022–2025, showing rapid adoption, especially in *Python*.
2. Identified the most popular SDKs (*OpenAI*, *Google*) and models, highlighting the 80/20 pattern where a few models dominate usage.
3. Classified usage patterns across repositories, providing actionable insights for developers and researchers.
4. Conducted statistical analysis of repository attributes, revealing differences in visibility and project scale across languages.

## Future Work

- Extend the study to **private repositories** or **other programming languages** to improve external validity.

- Enhance usage pattern classification using **NLP-based semantic analysis** or **machine learning** to reduce misclassification.
- Investigate **causal relationships** between SDK choice, project visibility, and adoption of LLMs.
- Explore **multi-model projects** and **cross-SDK integrations** to better understand complex LLM application architectures.