Beyond Acceptance Rates: The Impact of JetBrains AI Assistant and FLCC

Analysis of the behavior of users assisted by LLMs in 13 JetBrains IDEs

Remco Schrijver, Pouria Derakhshanfar, Annibale Panichella, Arie van Deursen, and Maliheh Izadi

Delft University of Technology

TU Delft

Beyond Acceptance Rates: The Impact of JetBrains AI Assistant and FLCC

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Remco Schrijver

Delft University of Technology JetBrains

Author: Remco Schrijver
Student id: 4852540

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, TU Delft
University supervisor:	Assistant Prof. Dr. M. Izadi, TU Delft
Company supervisor:	Dr. P. Derakhshanfar, JetBrains
Committee Members:	Assistant Prof. Dr. S. Dumančić, TU Delft
	Associate Prof. Dr. A. Panichella, TU Delft

Abstract

LLM (Large Language Model) powered AI (Artificial Intelligence) assistants are a popular tool used by programmers, but what impact do they have? In this thesis we investigate two such tools designed by JetBrains: AI Assistant and FLCC (Full Line Code Completion). We collected over 40 million actions - including editing, code executions, and typing - in the form of metric data spread out over 26 thousand users. With this data, we look at how user behavior changes when assisted by AI Assistant or FLCC.

Users spent more time in their IDEs (Integrated Development Environment) and typed more when assisted. In most cases, we see a decline in (manual) testing, or at best an equivalent level. And how do multi-programming language benchmarks reflect acceptance rates by users for these respective languages? There seems to be no real correlation between these benchmark results and what users accept in their generations, but the available benchmarks are also limited.

In the end, more research is required to put these results in perspective, for now, it is still up in the air whether the changes in user behavior are positive or negative. But relating this to existing works the impact seems mostly positive on users of AI Assistant and FLCC.

Contents

1 Introduction

LLMs (Large Language Models) in the form of code assistants have seen increasing adoption in the developer community over the last two years. With the introduction of these tools, there seems to be a correlation with a reduction in code quality in public GitHub repositories[1]. Committed code in general stays part of public codebases for a shorter amount of time, this heightened amount of churn suggests the committed code is significantly less useful and has to be thrown out or refactored within 2 weeks. This conforms to the hypothesis defined in the existing literature that wrongful use of code assistants can lead to diminished code quality and/or maintainability [15].

The increased use of code assistants is also seen by researchers associated with GitHub Copilot [37], but the metrics collected are not contextualized, investigate more than just acceptance rates, or identify more in-depth usage patterns. This makes it difficult to gauge whether users of these code assistants follow the recommendation of existing literature, users should test more when integrating code from LLMs into their codebase. It also makes it difficult to determine for which tasks AI Assistants can assist/help users.

We want to observe the impact of the assistance provided by LLMs to users and compare them to user behavior not assisted by LLMs. The behaviors investigated are in the form of microanalysis, state machines, and n-grams of actions, and in the form of macro analysis, session duration, number of typing actions, and number of file switches. On top of this, we look at the number of code executions users make, this is used as a proxy for (manual) testing. This metric is noisy and not adequate for all IDEs, but with the data we have the only option at this moment in time, and allows for sketching a rough picture.

In addition to gaining deeper insights into the usage of AI assistants by users, this research also enables us to investigate the alignment between common benchmarks [9] and completions accepted by users. This can correlate the benchmark performance of a model and its perceived productivity by users [37].

In the end, we found that user behavior barely changes on the micro level, which are chronologically ordered patterns of individual actions. But on the macro level, we do see changes. Users who are assisted by LLMs have longer sessions and type more per hour, but varying results for the number of file switches per hour. Whether or not these changes are positive or negative requires further research. When it comes to (manual) testing we observe a drop in most IDEs where this metric of code executions is relevant, and in the best case, we see no drop. There is no concrete relationship between benchmark performance and acceptance rate for specific languages.

[1]White paper on code churn by GitClear: `https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality`

2

Related Work and Background

In the existing literature on LLMs for assisting programming we see three main methods, where a lot of focus is put on quality and usability. These three main methods of research are: quantitative analysis, qualitative analysis, and user surveys, and some of these are a mixture of analysis methods. In the coming sections, we discuss multiple works for each method of research.

2.1. Quantitative analysis

The first category of analysis, quantitative analysis (e.g. [37, 14, 3]), allows researchers to gain insights into potentially large datasets. In the context of AI programming assistants, this mostly relates to collecting usage metrics or evaluating code generations of the assistant, be that using static analysis or test cases.

The main benefits of quantitative analysis are the following: there are large amounts of collected data that reduce the variance in results and increase reproducibility. As well as this data comes from real-world environments, improving how generalizable the results are. However, the downside of quantitative analysis is that the amount of data also brings downsides with it, first off it is difficult to contextualize data upfront. Secondly, interesting patterns can be investigated more closely, but it requires these patterns to be discovered. This makes it possible that important factors or findings can be left hidden in the data.

2.1.1. Metrics collection

A major quantitative analysis method is collecting usage metrics from IDEs of programmers that use AI programming assistants. Collected metrics can range from acceptance rates, file type worked in, code accepted, code rejected, etc.

Some studies are limited in what conclusions they can make, due to a limited amount of context of the metrics. This might arise for data that is anonymously collected and cannot contain code. This type of study is limited but gives a broader look at general patterns in user behavior and showcases trends in usage. More invasive metric collection, often done for smaller user groups that opt into this type of study provides the necessary context missing in the more mass-collected metrics. These more fine-grained metrics collection studies give the researchers a deeper understanding of the usability and functioning of AI programming assistants, with limited insights into general behavior compared to the mass-collected metrics.

A major study done on one of the largest AI programming assistants, GitHub Copilot, is one of the studies with a broader focus, investigating acceptance rates of programmers when using Copilot [37]. To put the results from the metrics data into context, the researchers opted to include a user survey.

With this context, the researchers determined if there was a correlation between the measured factors and the reported 'perceived productivity' by users. This resulted in the conclusion that using acceptance rates is the best metric that corresponds with the perceived productivity of a user. But acceptance rate

does not correlate to the expected time that the code generated remains unaltered and included in the code base.

On top of this conclusion they found some interesting patterns in acceptance rates, non-working hours have higher acceptance rates. More popular languages have a higher acceptance rate, TypeScript, JavaScript and Python being the major ones with higher acceptance rates.

Other existing literature is a mix of quantitative and qualitative analysis [14], with a heavy accent on the quantitative part. The researchers compared three LLMs in a real-world setting by publishing an open-source IDE extension, garnering 1200 active users. They follow these users for a year, collecting two million completions. Of these two million completions, 1690 were investigated more in-depth — as part of the qualitative study— to identify why the models performed poorly.

The findings from this study showcase decreased performance of models on less mainstream languages, this is in line with the results of benchmarks like MultiPL-E [5] published on huggingface[1]. In a similar vein offline evaluations in the form of benchmarks do not seem to correctly correlate with real-world performance, leaving a gap for increasing the usefulness of benchmarks.

The qualitative part of this research by Maliheh Izadi et al. [14] found an interesting mix of reasons why model generations were rejected. The largest category for rejection with 66.3% was due to the limitation of the models, followed by 24.4% of cases where the model was misused, and the last 9.3% is the user overwriting seemingly correct generation.

With these findings in hand, the researchers suggest a mix of approaches to improve the real-world performance of these models. This includes but is not limited to changing training objectives, mitigating errors due to typographical issues, and improving the usability of the models.

2.1.2. Static analysis of code generations

Static analysis gives a factual but limited insight into the quality of code. This limitation is that at compile time not everything can be known without hampering the expression of the programming language. More conclusive results about the correctness of code are provided by proof assistants, but these require their programming languages. These languages, Coq, Agda, Lean, etc., are not often used, so they are therefore not useful for the evaluation of more used languages like JavaScript, Python, Java, etc.[2]

These popular languages can use unit tests for evaluation, but to use unit tests for analysis, tests either need to be provided or written. In the case of mass-collected generation samples these might not always be provided, requiring researchers to create these. This makes static analysis a good candidate here, with the added benefit that code does not need to be executed. But when the prompt does not provide enough context for the LLM to generate a (semi)-proper code file, static analysis fails.

If static analysis fails, it might be necessary to switch over to a qualitative analysis form of study. This might or might not be feasible based on the amount of data that is available. Instead, the data points that cannot be analyzed by static analysis can be dropped from evaluation, possibly introducing a biased evaluation dataset.

The following research uses static analysis on the MSR 2024 mining challenge dataset, investigating the number of potential code smells generated by ChatGPT in the context of Kubernetes manifest files [35]. Using static analysis to investigate the various generated files found in the source dataset of ChatGPT share URLs. In a later section we will see more research that makes use of this MSR 2024 dataset, and in Section 2.2.1 the details surrounding this dataset will be elaborated on.

The results of [35] suggest that using ChatGPT is helpful, but in 35.8% of the 98 manifests investigated there was at least one code smell. Of these, the objects most troublesome for ChatGPT to generate without any smells were Deployment and Service. The most frequent smells are related to not setting

[1]bigcode scoreboard for multiple languages, selecting columns with less popular languages showcases large discrepancies in performance between popular and less popular languages `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`

[2]Look at the most used programming languages on GitHub commit pushes `https://madnight.github.io/githut/#/pushes/2024/1`

CPU or memory usage requirements, which is a lesser problem and also a logical result because ChatGPT does not know the specifics of the hardware used.

The resulting advice from the researchers is to ensure a form of static analysis is part of the release pipeline when using generated files by ChatGPT. This ensures that they are checked before they are released and used for container orchestration.

2.1.3. Test based evaluation

Most code-based generation benchmarks like HumanEval [6] and MultiPL-E[5] use some evaluation based on testing. A popular one is pass@k, which determines if in the top 'k' code generations one passes the test. This is different from existing similarity evaluations because it verifies functional correctness instead.

A more recent benchmark tries to address the fact that HumanEval fails to capture real-world performance. The main weakness of other benchmarks is identified by Xueying Du et al. [12] that these benchmarks are lacking for complete class generations. To solve this the researchers introduce ClassEval [12]. A benchmark that contains 100 Python class generation tasks, and tested eleven state-of-the-art models on ClassEval.

From this, Xueying Du et al. [12] discovered that all models performed considerably worse on ClassEval compared to HumanEval. While not finding any relation between the performance of a model on method-level coding and class-level coding. The best performers were GPT-4 and GPT-3.5, performing well in a zero-shot context, whilst smaller models like Instruct-Starcoder performed better when generating method-by-method to create a complete class.

Another piece of research looking at an adjacent task to code generations, language translation, uses tests to establish the performance of different models [27]. The findings show that even models fine-tuned for translating code have difficulties with synthetic examples seen in benchmarks. With larger real-world code bases, the complexity of the translation task will increase, creating more challenges for the model to solve.

One such challenge is method overloading, LLMs might not necessarily match all names of overloaded methods. When looking at translation from Java to Python the LLM would rename all overloaded methods to the same name. This is valid in Java code, but it creates invalid Python code because Python does not support method overloading. If the LLM correctly renames all methods, it still needs to replace the correct overloaded method in each call site. This is only one of the potential challenges real-world code bases can present to LLMs that the researchers.

Xueying Du et al. [12] propose a custom way of prompt-crafting that tries to alleviate the symptoms of bugs that occur during translation, improving performance with a 5.5% average. The research itself is one of the first of its kind in terms of scale and adds a foundation to the field of programming language translations by LLMs.

2.2. Qualitative analysis

Qualitative analysis in the context of code generation is reduced to investigating accepted and/or rejected code generations by users. For insightful results, it is best to have a larger overall understanding of the code base, which makes it easier for the researchers to judge why the user made their choice.

An important factor for qualitative analysis is to protect against mislabeling results. In general, these studies that include qualitative analysis guard against mislabeling the reasoning behind accepting/rejecting code generations by using multiple independent experts. Where experts are defined as a person with a decent amount of coding experience, the lowest encountered was seven years. This expertise makes their judgment more sound. However, mistakes can still happen. To mitigate mistakes, experts independently evaluate these code generations, and in the case of a mismatch between them, a discussion takes place.

This discussion can lead to an agreement and a label added, but experts sometimes exclude contested results. These are eliminated from the dataset because they are ambiguous or the experts' understanding of the specific context is lacking.

This means that qualitative studies give insight into evaluations made by users on code generations, allowing for detailed mitigation or improvement strategies. However, this is a rather intensive and time-consuming process, making it difficult to create a study on very large datasets.

A break with this trend is the following study where David OBrien et al. [25] compiled a list of 36,381 with TODO comments out of 102,424 repositories they searched. These comments were selected because the writers of these comments 'self-admit' to technical debt, which means as much as indicating the solution provided might be sub-optimal for the context or should be improved later.

The researchers took 1,140 of these comments and, with the respective context of the comment, generated code bodies using GitHub Copilot. They showcased that Copilot can create technical debt prompted and unprompted, but it is also able to remediate the technical debt indicated by the TODO comments. This depends heavily on the context and prompt given, but it does give insight into the potential abilities of AI programming assistants to perform these types of tasks to remediate tech debt.

2.2.1. ChatGPT Share URLs

The 2024 MSR mining challenge investigated a new feature of ChatGPT, share URLs, these URLs allow others to see the conversation a user had with ChatGPT. MSR created and curated a dataset of these shared URLs. This collection exists out of URLs taken from coding discussion context, Hacker News, GitHub discussions, pull requests, issues, commits, and source code.

The first research of interest investigates these snippets that contain Python or Java code and inspects them using static analysis on issues [32]. From this analysis, they find that ChatGPT uses a lot of undefined or unused variables/methods. The qualitative analysis parts of this research investigate the shared URLs in the context of pull requests. From the context of these pull requests, we determine whether code gets merged, and in which context ChatGPT is referenced and used.

Results from looking at pull requests that contain ChatGPT share URLs show that most generations by ChatGPT are rarely integrated into the code base. If code generations get integrated they are heavily modified. But on the other hand, ChatGPT seems to be seen as a versatile model by programmers, because it is used in every stage of software development. This ranges from using it for code generation to exploring new frameworks and libraries.

2.3. User surveys and studies

This category of research tries to determine how users interact with their AI assistants and what users feel are the strong and weak points when working with AI assistants. However, this type of user-based research might give incomplete or non-deterministic results. This is due to the trade-off of scalability and precision of the studies. Yet, it does support researchers in defining areas for improvement.

One major addition to the body of surveys is by Jenny T. Liang, Chenyang Yang, and Brad A. Myers published at ICSE 2024 [18] investigates how 410 developers interact with their AI programming assistants. Through investigating the answers given by this group of developers, they discovered some interesting insights into why and what developers used their assistants. They found that the assistants are mostly used for reducing keystrokes, improving velocity, and recalling syntax. On the other hand, the assistants were not used to help come up with novel solutions for their problems.

The researchers also investigated programmers who did not use AI programming assistants and their reasoning for not utilizing these tools. The main drivers for not using them were developers having less or no control at times as to what the AI assistant gave for output. Another driver is that some developers found that the code solutions provided by their assistants did not meet their requirements, be they functional or non-functional With these results in hand, the researchers make a case for reducing the friction between developers and their AI programming assistants when controlling or interacting with them. This will reduce the amount of distractions and make them easier to use while programming.

In another user survey Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin [34] utilized a survey among 135 software developers to find out what they expect from their AI assistants, like ChatGPT, and what keeps them from using them. They found that in a lot of cases, developers want to use their assistant for tasks like library selection, but are afraid about the truthfulness of the statements made by the assistant.

To alleviate concerns about the truthfulness of LLMs, Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin introduce a tool that can estimate if a response is truthful or not. The tool generates multiple responses, and if the generations differ greatly from each other, the statements are likely incorrect. With this methodology, they achieved an F1 score of around 0.74 when evaluating against ChatGPT. This could potentially be a tool that could increase trust in generations by ChatGPT and, to an extent other LLMs, at the trade-off of higher costs for using the tool.

A study in a different area, determining the effect of AI Assistants on the learning capabilities of students. The study closely follows 22 subjects that are using ChatGPT [8]. This user study found that the assistance of ChatGPT in completing SE tasks for students did not create statistical differences in productivity or perceived capabilities. Instead, the use of the AI assistant increased frustration in students when doing these tasks. As a result, the researchers point out a lack of ability of ChatGPT in its current form to support learners with SE tasks. They argue this is something important to keep in mind when embracing LLM technology when teaching.

The last user study we discuss is on the potential advantages or disadvantages of creating safe and secure code when using GitHub Copilot as an AI programming assistant [1]. 25 participants are followed while they solve various programming problems, which can be possibly solved in an insecure way. The participants complete these challenges with or without the help of GitHub Copilot.

The results suggest that Copilot's assistance makes for more secure code when a user is tackling a difficult problem. However, it shows little to no effect when used for easier problems. On top of that, there is no evidence in the results to suggest that Copilot introduces a specific subset of vulnerabilities. But, the researchers note that the results are intriguing, yet the results require more research to make a decisive conclusion on the potential effects of Copilot.

2.4. The rationale for our chosen analysis

With this research, we want to investigate the behavioral impact on users when they are supported by an LLM. When reflecting on the discussed analysis methods we decide to use quantitative analysis. Quantitative analysis is the only adequate option when we want to analyze user behavior, because we want to capture a real-world environment, something more difficult to attain with Qualitative analysis or user surveys and studies.

The cooperation with JetBrains gives us access to large amounts of metrics data, so the quantitative analysis will be through the analysis of large amounts of the metrics data. From that data, we can compare users with and without LLM assistance. This is different from the existing metrics collection study by GitHub [37] where the focus is purely on acceptance rates.

2.5. LLMs

Large Language Models (LLMs) have seen much attention in the last few years. Tools like ChatGPT[3] gained mass media attention and high adoption by consumers. ChatGPT and other tools that speak to the users' imagination like Dall-E and CLIP [28] for image generation, SORA [29] for video generation. But LLM-powered tools also expanded to assist programmers, tools like JetBrains AI Assistant and FLCC deliver various ways of support for programmers.

The capabilities of most support tools vary widely and include but are not limited to:

- Code generation — Users can explain what issue they want to solve in natural language to the LLM in a chat interface and the LLM provides a partial or complete code sample to solve the issue of the user.
- Code completion — Similar to code generation, but instead of the user prompting the model in natural language, the development environment of the user sends context of the code the user is working with and the LLM suggests one or multiple lines to complete the code.
- Test generation — Users can prompt the model with existing code and request in natural language to create unit tests for the code provided to the LLM.

[3]Launch announcement of ChatGPT by OpenAI: `https://openai.com/index/chatgpt/`

- Code summarization — Giving the LLM a piece of code, the LLM then provides a summary of what the code does.
- Commit generation — Provide the LLM with all changes of the code, the LLM then provides a summary and description to use as git [20] commit message.
- Troubleshooting — LLMs can be provided with an error or problem with context like code if needed and help the user solve the issue they are facing.

LLM tools can therefore assist users in various ways, albeit with limitations like truthfulness, correctness, and safety, especially when it comes to generating code that will be integrated into the code base of the user.

2.6. JetBrains AI Assistants

JetBrains AI Assistant[4], also referred to as AI Assistant, is an AI Assistant suite developed by JetBrains. For the period of data collection, the LLM that backed AI Assistant was from the GPT family developed by OpenAI.

This LLM tool suite has more functionalities than the focus of this thesis, namely code completion generation. It is possible to use a chat-based interface, similar to online tools like ChatGPT, but with the added benefit of loading code into the context of the LLM. Another capability is summarizing commits or proposing naming for functions and variables. A side note is that AI Assistant, as of the time of data collection, might function differently compared to how it works currently and can be backed by a different LLM.

In general, due to the commercial nature of this product and the lack of publicized papers, it is more challenging to precisely explain the underlying structure and functionalities of AI Assistant compared to FLCC, the other LLM tools we discuss in the next section.

2.7. FLCC

Full Line Code Completion (FLCC) is a tool provided by JetBrains, shipped with some of their IDEs that use a small LLM that runs locally for single-line suggestions [31]. Running the model locally enables low-latency responses, but limits the model size. This limit is introduced by the limitations of users' hardware, the estimated size limit by the authors is roughly one billion parameters. The current model powering FLCC is a GPT/LLaMa-like model.

The FLCC pipeline first preprocesses and cleans the source code from the current file. Then, the tokenizer uses byte-pair encoding. Special tokens are introduced to aid in optimally using the LLM's context window. The context window contains the file extension, file path, and as much code as fits the remaining window size.

To optimize the memory footprint, the tokenizer encoding is optimized for compression rate. This allows for more code lines to fit in a similar-sized context window. Two of the optimizations are first, 'long tokens' where a single token represents a complete line, i.e. *for i in range(*. Secondly, they introduce whitespace trimming, but if spaces are important for scope, like with Python, then the special tokens called <SCOPE_IN> and <SCOPE_OUT> are used. This allows us to strip whitespace characters, saving memory and increasing the amount of data encoded by the tokenizer. While still being able to reconstruct the code to be valid later.

2.8. IDEs

The data collected spans over 13 IDEs, a small explanation in alphabetical ordering will follow below. This background knowledge aids and gives some insights into why some results differ greatly between particular IDEs or might not apply at all to certain IDEs.

- **Aqua** - Referred to with shorthand QA, an IDE aimed at testing. In particular test automation, works with the following languages: Java, Python, JavaScript, TypeScript, Kotlin, and SQL.

[4]More info here: `https://www.jetbrains.com/ai/`.

- **CLion** - Referred to with shorthand CL, mostly used for C and C++, aimed at providing comprehensive toolchains for these languages and embedded development.
- **DataGrip** - Referred to with shorthand DB, an IDE used for connecting to both relational and NoSQL databases. Mostly used for creating database queries, in the various SQL-dialects and NoSQL query languages.
- **DataSpell** - Referred to with shorthand DS, mostly used for data analysis, focuses heavily on Jupyter Notebooks and Python. Also works with SQL queries and the R language.
- **GoLand** - Referred to with shorthand GO, aimed at the Go but also supports JavaScript, TypeScript, Python, and Bash out of the box.
- **IntelliJ Ultimate/Community Edition** - Referred to with shorthands as IU and IC respectively, aimed at Java and Kotlin development. Community edition has fewer features compared to Ultimate and also at the time of data collection no access to AI Assistant.
- **PhpStorm** - Referred to with the shorthand PS, mostly aimed at PHP development, but also supports JavaScript and TypeScript, also integrates with various SQL databases.
- **PyCharm Professional/Community Edition** - Referred to with shorthands as PY and PC respectively, aimed at Python development with support for Jupyter Notebooks and integrates with various SQL databases.
- **RubyMine** - Referred to with shorthand RM, aimed at Ruby and then mostly Ruby on Rails development. It also integrates the features from WebStorm and DataGrip.
- **RustRover** - Referred to with shorthand RR, and fully aimed at Rust development.
- **WebStorm** - Referred to with shorthand WS, and aimed at web development, works with JavaScript and TypeScript out of the box.

3

Research Proposal

The current research on LLM code generation and its quality and usability is conducted in four main different ways. The first is static analysis of generated code by AI assistants, be that in benchmarks [13] or in the real world [32]. The second aims at surveying developers on how they use these tools [18]. Then there is a dual-pronged approach of user surveying combined with metrics collection of usage patterns [14]. The fourth way is mass metric collection concerning acceptance rates by developers in large tools like GitHub copilot [37].

This research proposes an enhanced version of mass metrics collection to gain more insight into the effects of LLM tools on users. By collecting mass metrics of acceptance rates and user behavior, investigating if there are differences in usage patterns between groups with and without LLM assistance. This gives a more contextualized look at the effects of LLM assistance on programmers than just looking at acceptance rates.

3.1. Research Questions

We define the following three research questions:

- RQ1 — *What does the behavior in the metrics collected between users using AI Assistant, FLCC, or neither look like?*
- RQ2 — *How does support from LLM tools like AI Assistant and FLCC affect users on how much test and code executions per session they do?*
- RQ3 — *How well do multi-programming language benchmarks comparing the performance of models on different programming languages represent acceptance rates for the equivalent programming languages in the real world?*

3.2. Justification

The justification for the first research question is that most large studies with comparable amounts of data have not focused on comparative analysis between users with and without LLM assistance. We think that this might give interesting insights into how LLMs support users and how they do not. On top of creating insights in avenues of future research on the precise impact of LLMs on impacted user behavior.

For the second research question, we pull from existing literature where scholars researching the quality of code generations warn against using LLM-generated code without sufficient testing [15, 32]. Due to LLMs producing code that includes bugs and/or security exploits. Our research aims to find if there are significant differences in the amount of (manual) testing users do when assisted by LLMs.

Finally, for the third research question, most LLMs aimed at programming use a handful of benchmarks to verify performance. However, when comparing the performance of LLMs for different programming languages there are only one or two benchmarks adopted by the community that test for this. With

research question three we investigate if the acceptance rates of various programming languages in the real world correlate to the performance seen in benchmarks comparing performance of LLMs on different programming languages. Allowing us to gauge if benchmarks are in line with real-world acceptance rates.

3.3. Approach

The approach to collecting the metrics is in tandem with the telemetry already collected by the JetBrains AI tool, and possibly augment it with more data. An important factor is the privacy of users in the study and the code they write. Telemetry collected needs to be properly anonymized as well as not expose underlying code written by users.

We aim to collect the following metrics:

- Time spent in various file types, configurations, documentation, code, and tests (RQ1).
- Amount of accepted and rejected generations, average time looking at accepted/rejected, average length of generations accepted, possibly average edit distance of accepted generations on a per-file basis, and split generations based on being a comment or just code (RQ2).
- The extension of a file that is worked in (RQ1).
- Time spent working on each file (RQ1).
- Amount of lines written/edited/deleted for each file (RQ1).
- Time spent unfocused on IDE when the file is open (RQ1, RQ2).

To ensure data is anonymous, the file name, project name, and random user ID will be hashed with a unique salt generated by the user's plugin and also only known by the user. In this way, the file name that can potentially identify a project is sufficiently anonymized so that researchers cannot identify the file.

Deanonymization threat: Amount of lines written/edited/deleted could potentially identify a git commit. Because file lines can identify complete projects the total file lines are omitted from the metrics collection.

3.4. Evaluating results

For research question one we will split the analysis into two parts. First, microanalysis, where we create n-grams of user actions in a chronological order split by session. The second part, macroanalysis, where we use comparative analysis for multiple user groups comparing the session duration, typing behavior, and file switch behavior. Similarly for research question two we use a comparative analysis of the amount of (manual) testing done between user groups. For research question three we will collect the acceptance rate for code completions and compare these with existing benchmarks scoring GPT-3.5 one of the underlying models used by AI Assistant.

4

Methodology

4.1. Data collection

Data collection is performed by an internal JetBrains member, this data collection process is in line with internal JetBrains policies. The JetBrains policy ensures that collected data is sufficiently anonymized. And metrics can only be collected once a review is done by the JetBrains metrics team, verifying that data is anonymous.

The shape of the data collected is discussed in the next section but briefly, there is the raw data set, with all metrics data for the session. Then there is a post-processed dataset where only actions are included and parts of the data are flattened and defined as columns in the dataset. An additional opportunity that arose from the collected data is that we have access to FLCC user data, this was not part of the initial goal of the thesis, but because a small LLM backs FLCC, we decided also to include these user groups where this is applicable.

In Section 3.3, we discussed deanonymization threats, and what information to collect. Most of the information we wanted to collect we also acquire in reality, however have to work within the limits set by the knowledge we can gain from the metrics data provided. That means we no longer have to fear deanonymization, but we do not have information like idle time outside of IDEs.

4.2. Data

The data collected comes from 13 JetBrains IDEs with most aimed at different programming languages, all columns of the metric data are shown in Table 4.1. A more in-depth look at the most important columns is discussed in the upcoming paragraphs.

Table 4.1: Showcasing the different columns of the collected metric data.

Name	Type	Explanation
project_id	hashed identifier	A hashed identifier of the project being worked in.
file_path	hashed identifier	A hashed path of the file being worked in.
file_name_pattern	'regex' pattern	A string denoting the pattern of the file.
file_type	categorical string	The type of file being worked in.
event_id	categorical string	Event that this row describes a category for action.
group_id	categorical string	Grouping of the action, fine-grained compared to event_id.
action	categorical string	Categorical string denoting what action the user did.
time_epoch	epoch number	The epoch when the action took place in milliseconds.
device_id	hashed identifier	A hashed identifier of the device being used.

The most important columns in this data are *action*, *time_epoch*, and *device_id*. These three allow us to determine individual user sessions as well as what they are doing and how long these actions take. This allows us to look at statistics like the number of accepted generations or the difference in time taken to type between users with and without the AI assistant.

The columns *event_id* and *group_id* allow for a broader look at the usage patterns of users, giving general insight into the difference between users with and without the AI assistant. This can potentially vary from difference in editing tasks done, or how much the users move between files.

With the *file_type* column it makes it possible to showcase differences between programming languages and potentially tasks. It allows us to differentiate between different languages or text files for example.

The *file_name_pattern* column serves a similar purpose as the *file_type* column, but more fine-grained. This column allows for looking at the extensions of files, making it easier to potentially identify files that are not known by the IDE. Where this would result in the *file_type* being undefined or unknown.

Then there is the *file_path* column, which allows us to identify different files from each other, giving insight into how much users navigate between files. With this information, it makes it possible for us to identify user behavior. Additionally, it might give insights into whether AI Assistant, enables users to keep working on their files for longer. Because a user needs to look up fewer definitions or investigate previously written code to recall exact code behavior.

4.3. Dataset statistics and User Groups

After the correlation study is completed, another important part of data exploration is identifying the underlying statistics. For that, we first collect basic statistics like the number of data rows, the number of unique actions, user distribution between IDEs, etc. With these basic statistics out of the way, we continue with more interesting statistics.

Those more interesting statistics are mostly the distribution of user groups, we identified the following user groups, where user groups are defined as users that use or do not use the same LLM assistance tools category:

- AI Assistant Users
- FLCC Users
- FLCC and LLM Users
- Other LLM Users
- Non-FLCC and LLM Users (Baseline)

Non-FLCC and LLM users are the largest user group and serve as our baseline. They do not use FLCC nor do they use LLMs, and in the future, this group will also be referred to as either baseline or users without LLM support/assistance. We identify our user groups by either selecting them from code completion suggestion metrics, as well as scanning the entire dataset for particular actions done by the user.

To assign a user group based on the actions they did we first analyze the over 4000 unique actions that were found in the metrics data. We investigate which actions are related to AI Assistant, and later identify other LLM tools made use of in this dataset. This gave us over 10 other tools used by IDE users including Tabby, Cody, and Copilot. We assigned all the actions to their respective tool and then assigned user groups based on what tools they used. With that, we got a complete user group dictionary allowing us to get our results and split them out by the respective user groups.

4.4. N-gram analysis

N-gram analysis [33] gives us more possibilities for studying micro behavior compared to state machines because state machines focus on bi-grams. We create the n-grams by splitting all the datasets on user and session, then sorting the rows at the time of execution. This creates a chronological ordering of events from which we can make arbitrary n-grams.

To give us a more detailed analysis we unpack actions, whenever we encounter a metric data row where it is an action, instead of including the event_id we include the action_id. This results in more fine-grained patterns when evaluating. We store all these n-grams as dictionaries with the respective n-gram as a key combined with the number of times it is found as a value.

When it comes to visualization we encounter the same problem as with the state machine, including all values create diagrams that are almost impossible to understand let alone fit on a single ISO A4 page. Therefore, we decided through trial and error that we include the top 50 most encountered n-grams in the diagram and normalize the value of appearance to the total dataset. This means a subset of the complete Sankey diagram is shown, making for easier interpretation and the possibility to include them in this thesis. We decided on the top 50 most occurring n-grams in particular because more would make diagrams hard to read in some cases and less would present less information.

4.5. Action state machine

State machine analysis [4] serves a similar purpose as the n-gram analysis which was discussed in the previous Section 4.4. It makes it possible to visualize users' behavior and what their action transitions look like. This is done by creating pairs of transitions from action A to B, where A and B will constitute nodes in our state machine, and we count the number of transitions between each pair of nodes. The state machine is represented as a graph and the edges are directional.

However, we have an immediate problem when trying to visualize our state machine. When using actions, we get graphs with over 1000 nodes. If we merge all IDEs, we get over 4000 nodes. This makes it virtually impossible to create a state machine that humans can parse. The solution to this is using, categorized versions of actions in the form of the group_id or event_id columns. This creates a better but less fine-grained representation of the underlying dataset and transitions.

4.6. Comparative analysis

For the remainder of our results, we use comparative analysis. We do this comparison by collecting statistics from the dataset and then comparing the statistics between the earlier defined user groups. When comparing against our baseline, we can investigate whether LLMs introduce major changes in user behavior. However, due to a large amount of compounding factors, it is difficult to draw singular conclusions from the results. Our discussion therefore will also limit itself to speculative hypotheses that can be reinforced by existing literature or used as starting points for future works. Potentially investigating why particular results are found in the form of more controlled studies. The results can additionally be used to compare against already existing studies that were focused on controlled experiments and see if their results generalize to real-world behavior.

4.7. Acceptance rate analysis

We use the acceptance rate in the last research question, and we share some additional results we found with it. It can be interpreted as a proxy for productivity by the reader. However, it is unclear whether acceptance rate is a representative metric for 'real' productivity. Yet a previous study did find that 'perceived' productivity [37], the users' own opinion on their productivity, is related to acceptance rate. Perceived productivity is a biased metric, users score themselves for their productivity, which might not align with 'real' productivity. But to define real productivity would be a semantic and philosophical question, something that is out of the scope of this research.

A side note on acceptance rate however is that: A high acceptance rate does not necessarily mean a model performs well, yet a low acceptance rate most likely indicates a poorly performing one. This is because a low acceptance rate means that the generated code by a model is most likely not useful or applicable to the situation. But when a lot of generations are accepted, this might mean that generations are just 'good enough' for the user, but might require intervention on the user side.

5 Results

5.1. RQ1: Difference in user behavior between AI Assistant, FLCC, and non-LLM users

To investigate the difference in user behavior when assisted by LLMs we separated the analysis into two categories. The first category is on micro level, this micro level entails creating a chronological list of events per user per session. From these lists of actions chronologically ordered we create various sized n-grams, more detail on this is in Section 5.1.1. In the second category we analyze the macro behavior of users, this includes session length, typing behavior, and file switching. Combining these gives us an insight into how a user behaves in their IDE environment, which we describe more in-depth in Section 5.1.2.

5.1.1. Micro Analysis

The microanalysis using n-grams does not give interpretable differences. Patterns between user groups remain almost the same when selecting the top 50 n-grams. This holds for the smaller 3-grams up to the larger 15-grams. The main takeaway of the microanalysis is that the main actions of the user are typing and editing their code. Identifying differences by using state machines for 2-grams there are some shifts in percentages split on user groups. However, to observe these differences, we have to use broad categories, making it no longer a microanalysis of the patterns exhibited by users.

5.1.2. Macro Analysis

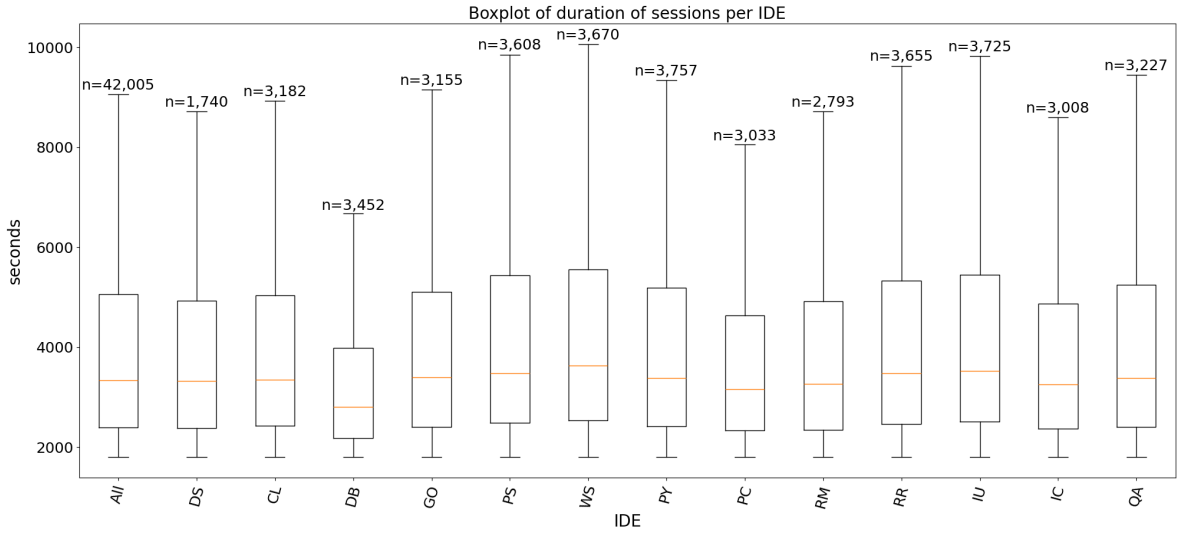
Macro level analysis shows different behavior between user groups, namely session duration, typing behavior, and varying differences for file switches. First, in Section 5.1.3, the results for session duration are shown. In Section 5.1.4 we showcase the typing behavior of users, the amount of typing actions users do, and the time taken for a typing action. Lastly, in Section 5.1.5 we investigate the number of file switches and time spent in files.

5.1.3. Session lengths

To contextualize these results, Figure 5.1 shows boxplots of the session lengths per IDE with the outliers removed. In all further boxplots, outliers are not included due to the noisy nature of the data. The numbers on top of the whiskers are the sample size of the underlying boxplot. This will contextualize anomalies, like large deviations of similar categories, in some of the results. Where these anomalies can be explained as caused by low sample sizes. However, in most of the selected diagrams, the sample sizes give us results that are generalized over a larger user group.

Now comparing the session duration between IDEs in Figure 5.1 we can see the major stand-out is DataGrip. The other thing that stands out is that the community editions of both PyCharm (PC) and IntelliJ (IC), have lower whiskers compared to the Professional and Ultimate counterparts, respectively. That is the extent of the major difference to observe, in most cases, the medians are rather similar.

Figure 5.1: Session Lengths per IDE



Splitting session duration on user groups as in Figure 5.2 shows a general increase in session duration for users that are supported by LLMs. The results of individual IDEs are very similar to the result of the general result of all IDEs combined, only when user groups have a low sample size do we see differences. Even DataGrip, which stood out in Figure 5.1, follows the same pattern of user groups with LLM support having longer sessions.

5.1.4. Typing behavior

The next macro presented is typing behavior. This consists out of two metrics, typing actions per session hour and time spent per typing action. These two measures are split between the user groups allowing comparison and giving us an idea of the impact of LLM assistance.

In Figure 5.3 we observe that the amount of typing actions per session hour increases when an LLM assists users compared to those who are not. This increase lies between 10% to 30%, depending on the tool used. FLCC seems to increase the number of typing actions the most, noting that the sample size is smaller compared to the other user groups, though. In general, this pattern is repeated for individual IDEs, when not taking into account the user groups for individual IDEs that have a very small sample size.

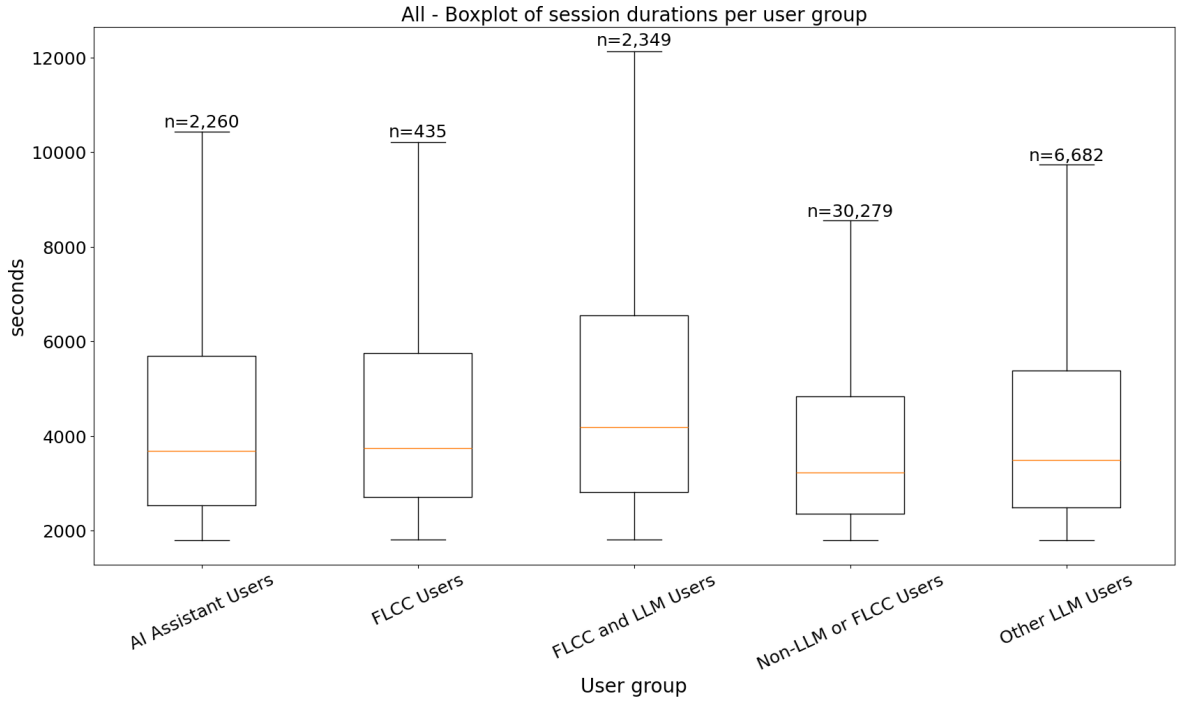
For Figure 5.4, we see that the time taken for a typing action for non-LLM supported users is a lot higher, the median as well as the top of the whiskers. Besides that, what differs from Figure 5.3 is the LLM-supported user groups are much more similar in time spent per typing actions. As with the typing actions per session hour, the pattern we see for the time per typing action is repeated in the individual IDEs when excluding the user groups with small sample sizes.

5.1.5. File switches and Time spent in files

The last macro investigates the number of file switches per session hour and time spent in a file. What we can see in Figure 5.5 is an increase in the number of file switches per session hour for users that an LLM assists. This increase varies from roughly 10% to 20% compared to the baseline user group. Yet this does not translate well to all IDEs, and not all IDEs follow this similar pattern, even when excluding the results for user groups with small sample sizes.

In Figure 5.6, we see a different pattern for the time spent in files compared to Figure 5.5, there is barely any difference between user groups. Medians and whiskers are similar between almost all groups, and in general, the boxplot and median are very heavy at the bottom of the graph, so it is nearly zero seconds in this case. As with the number of file switches, this pattern of fewer file switches for the baseline user group is not present in each IDE. For some IDEs, we see that user groups supported by LLMs spend more time in files, while for other IDEs user groups assisted by LLMs spend less time in files.

Figure 5.2: Session Lengths per user group



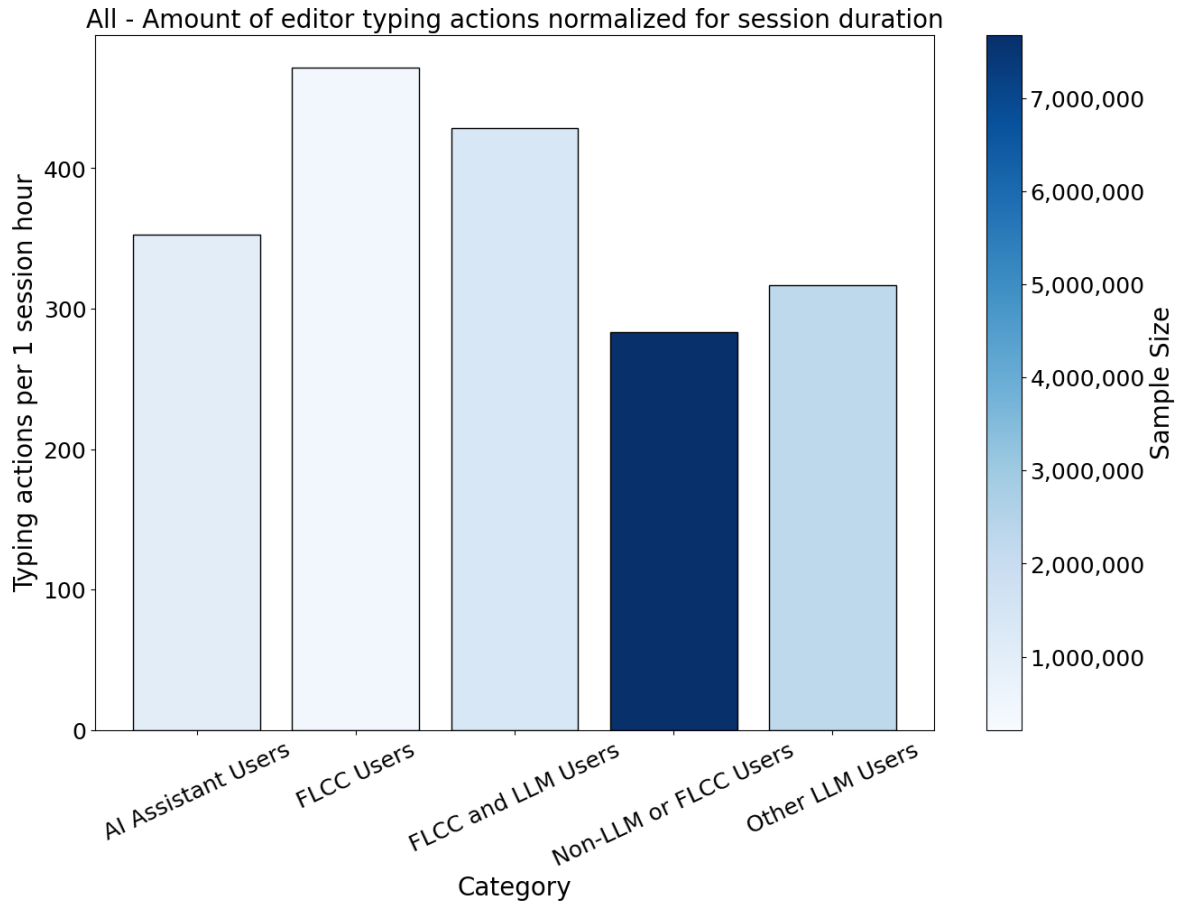


Figure 5.3: Figure showcasing the amount of typing actions per session hour for all IDEs

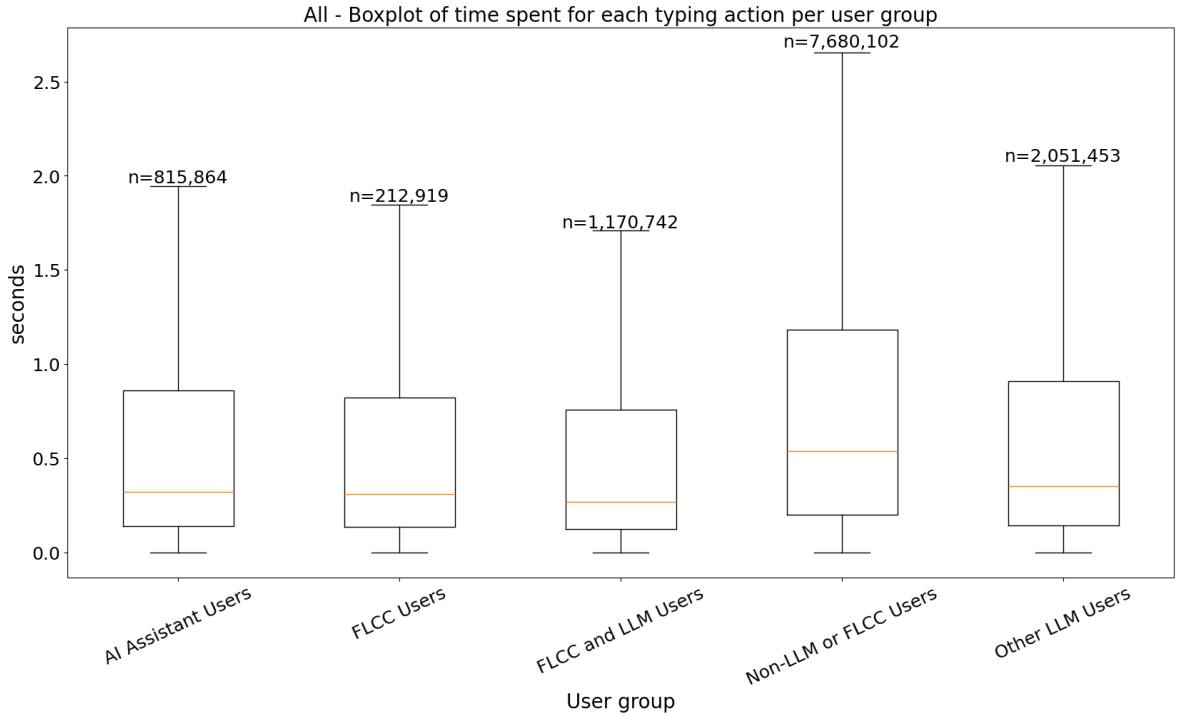


Figure 5.4: Figure showcasing the time taken for typing actions for all IDEs

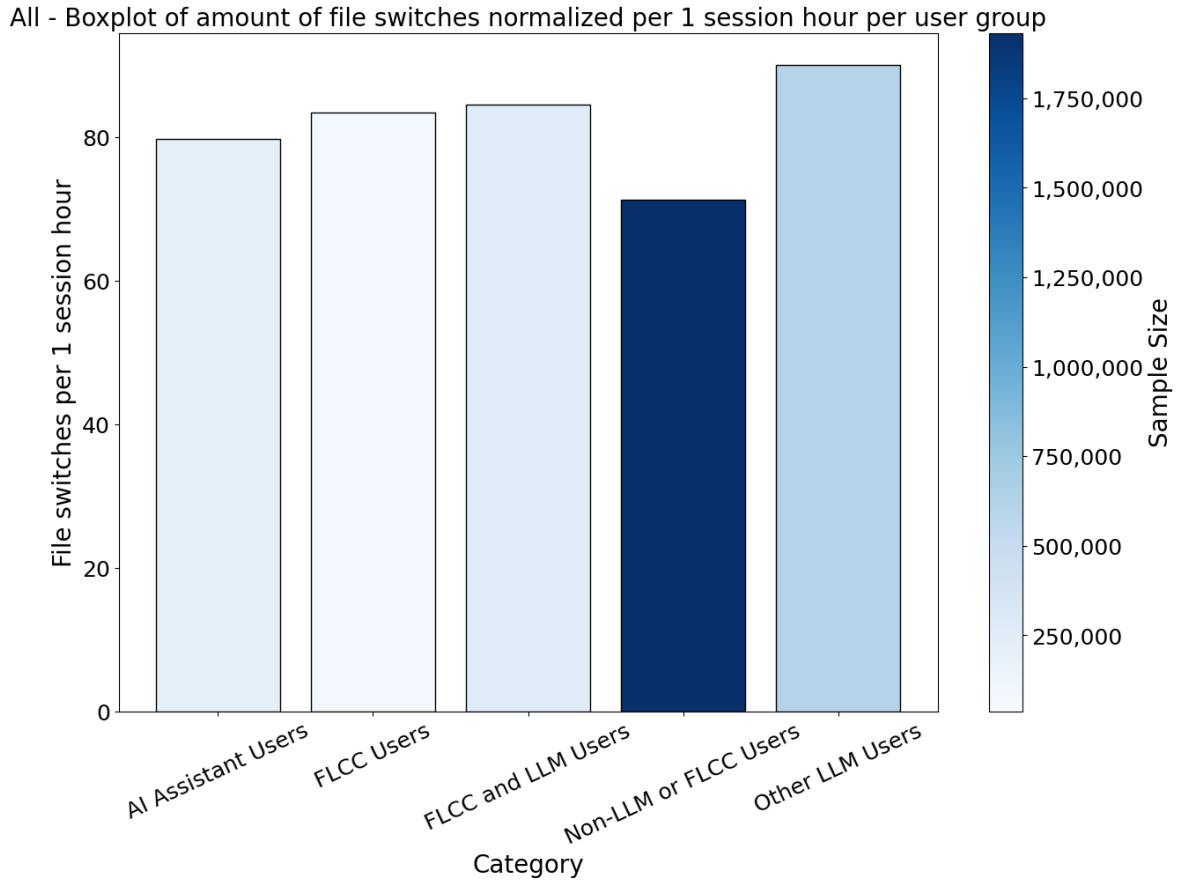


Figure 5.5: Number of file switches per user group for all IDEs

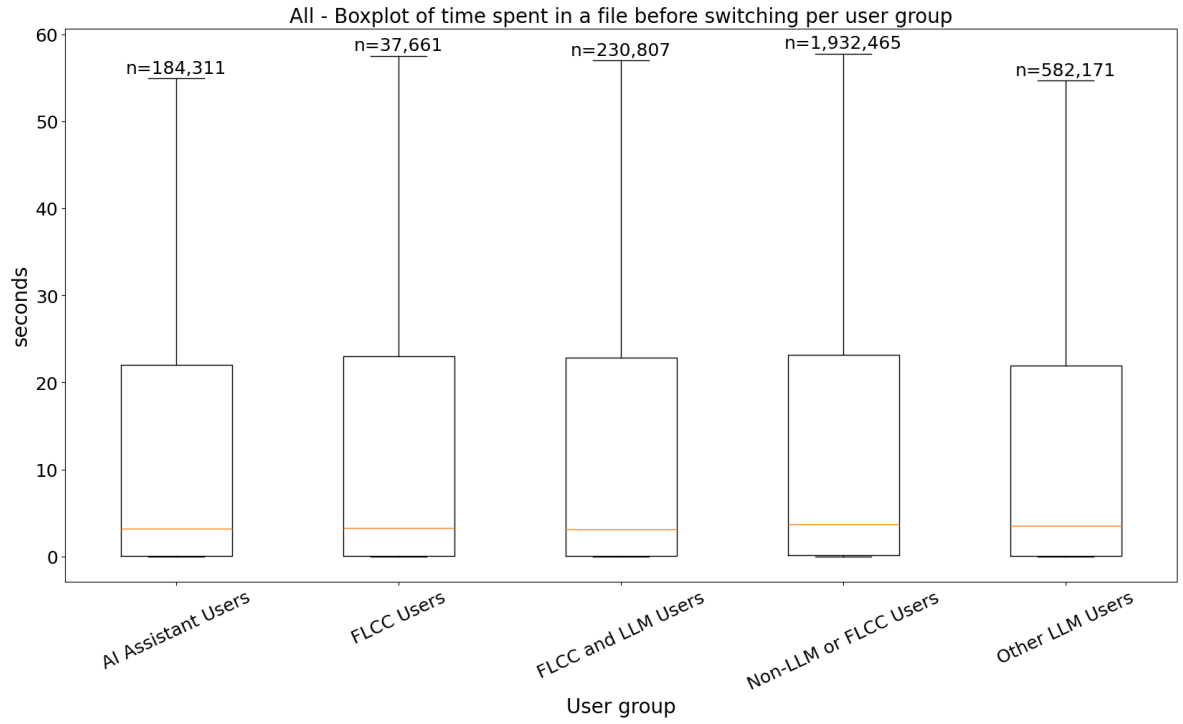


Figure 5.6: Time spent in a file before switching for all IDEs

5.1.6. Summary

To summarize the results, on micro level, there are minor differences between user groups, only those wholly underrepresented in the dataset showcase larger differences. This showcases the behavior of only a couple of individuals instead of a larger, more generalized group. The macro analysis gives a more well-rounded overview where we see that AI/FLCC users have longer session lengths, different typing behavior, and file switches are dependent on the IDE. Thoughts on these results are discussed in Section 6.1.

5.2. RQ2: Relation between accepted generations and amount of 'testing' by users

Due to limitations of the data collection process, it is not possible to identify the precise number of tests users run. However, it is possible to include a more broad category of 'code executions' or 'runs'; this is when a user executes their code or tests. This broader category now considers not only unit testing but also manual testing by users.

Looking at the results from this category, normalizing against session length and translating it to runs per session hour, we see the following. We can see in Figure 5.7 that users with AI Assistant and/or FLCC do fewer code executions per session hour compared to non-LLM users. However, the results may be distorted by the combination of all IDEs.

Due to constraints on the amount of content we can present in this thesis and to prevent this section from expanding dramatically, we will not go over all the individual IDEs. The results are uploaded on GitHub and can be seen there by those who are interested.

We will include the following IDEs: PyCharm Professional, IntelliJ Ultimate, WebStorm, and RustRover. All have sufficiently large sample sizes for the user group we are interested in, and the results between each IDE vary, making for a more convincing analysis later. The other IDEs are excluded due to low sample sizes for our user groups of interest or code executions having a different meaning.

Figure 5.7: Code executions per session hour for all IDEs per user group

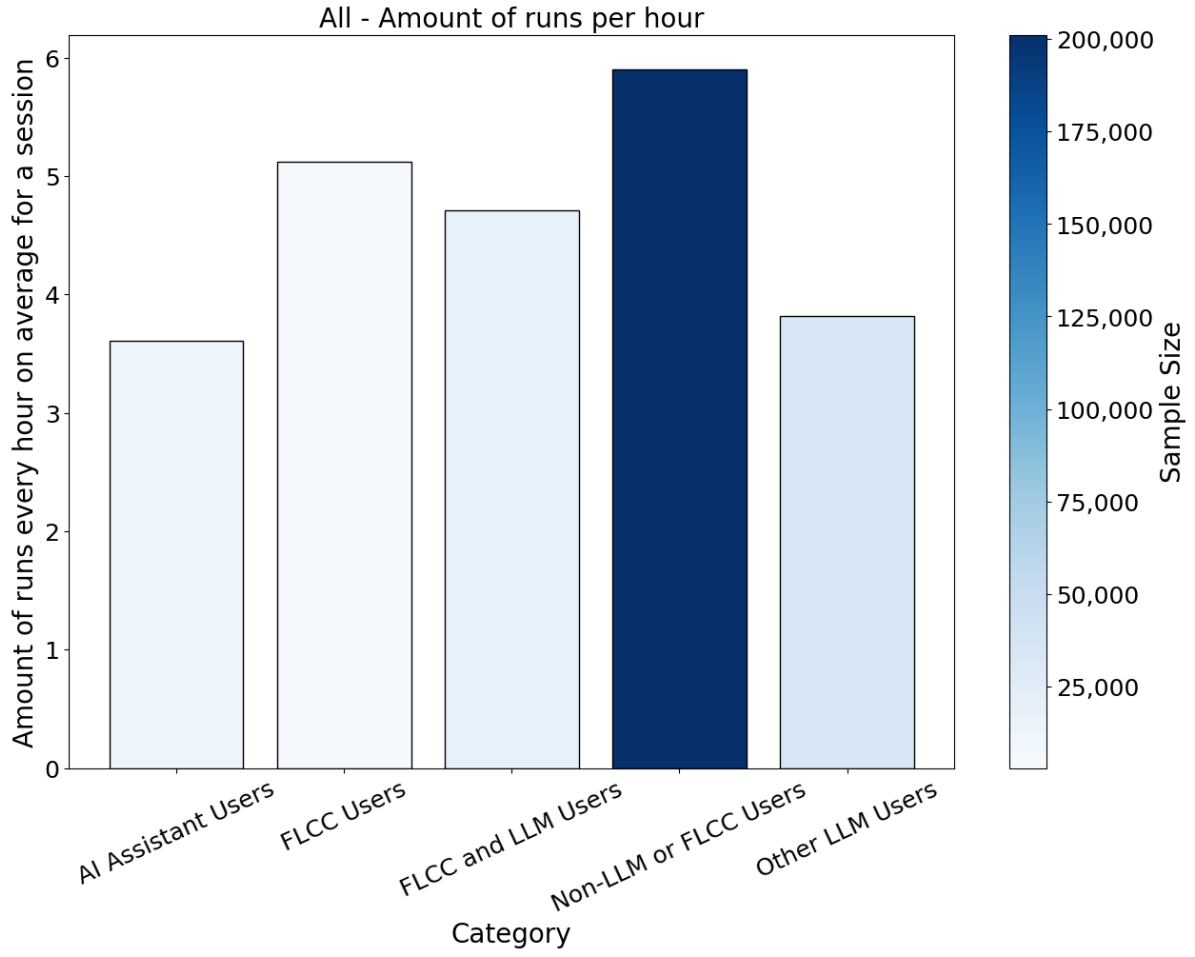
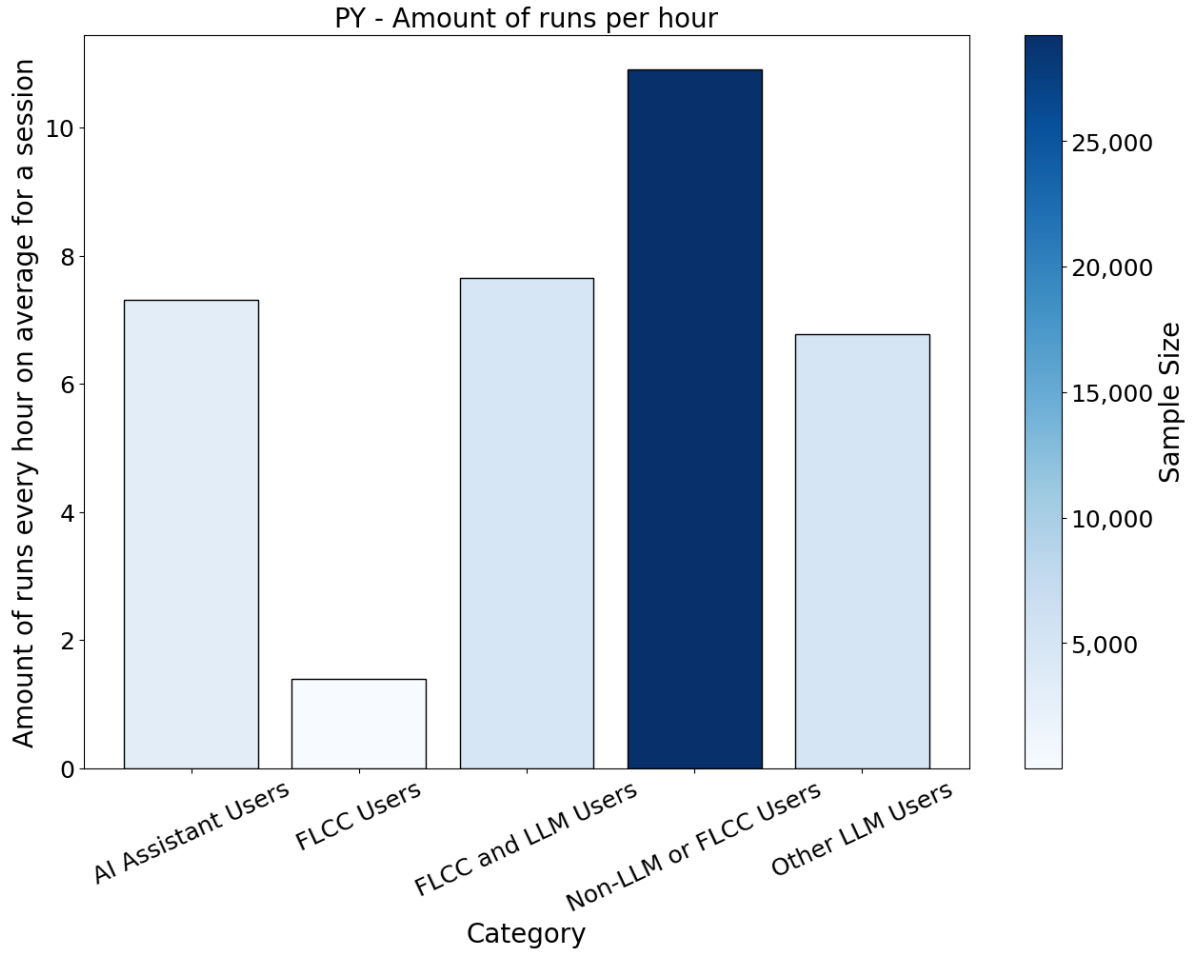


Figure 5.8: Code executions per session hour for PyCharm Professional per user group



PyCharm Professional amount of code execution
When looking at Figure 5.8, we ignore the FLCC user group due to the low number of samples. However, when we look at the other user groups, most importantly AI Assistant and FLCC and LLM users we see a decrease in the amount of code executions compared to the baseline user group. Lowering from around 11 code executions per session hour to only 7-7.5 code executions per hour, a decrease of roughly 30%. We will discuss this major difference more in-depth later in Section 6.2.1.

IntelliJ Ultimate amount of code execution
Now, when looking at the results for IntelliJ Ultimate in Figure 5.9 we see a very different result when compared to Figure 5.8 for PyCharm Professional. First, the FLCC user dataset is larger, and the results are more in line with the other user groups, both increasing our confidence in the result. But most importantly, we do not see a major drop in code executions for LLM users, except for AI Assistant users, where the drop is roughly 20%. On the other hand, FLCC and LLM users have an increase of around 10%-20% when compared to our baseline. In general, code executions are lower than those of PyCharm Professional.

WebStorm amount of code execution
For WebStorm, we have differences from other IDEs, in general in Figure 5.10, users with LLMs either increase or have an equal amount of code executions compared to the baseline. What should be mentioned is that WebStorm is a tool mostly used for web development; users will make use of the hot reload features available, limiting the number of manual executions we catch. This explains why there is a low amount of nominal executions compared to other IDEs. We still capture when users run tests or

Figure 5.9: Code executions per session hour for IntelliJ Ultimate per user group

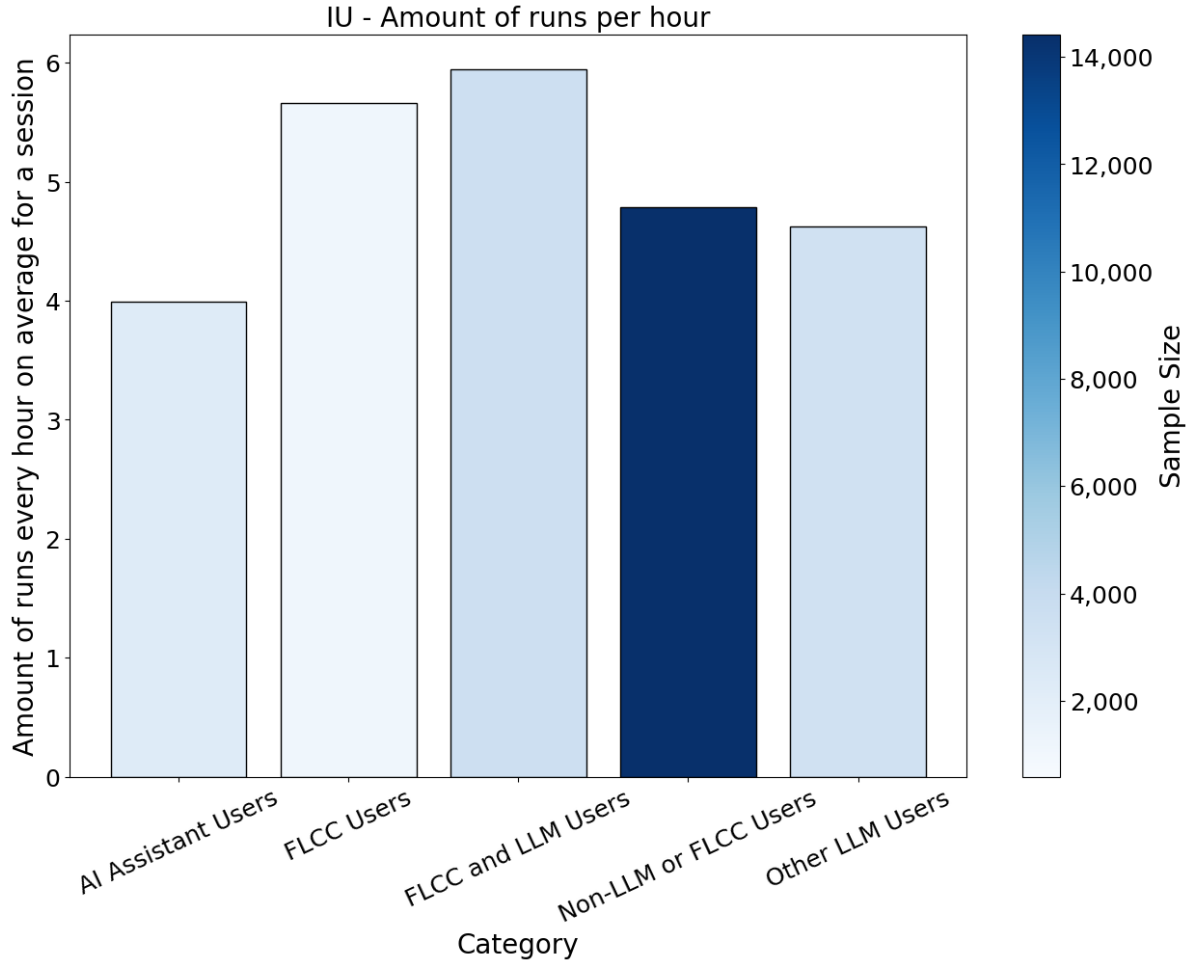
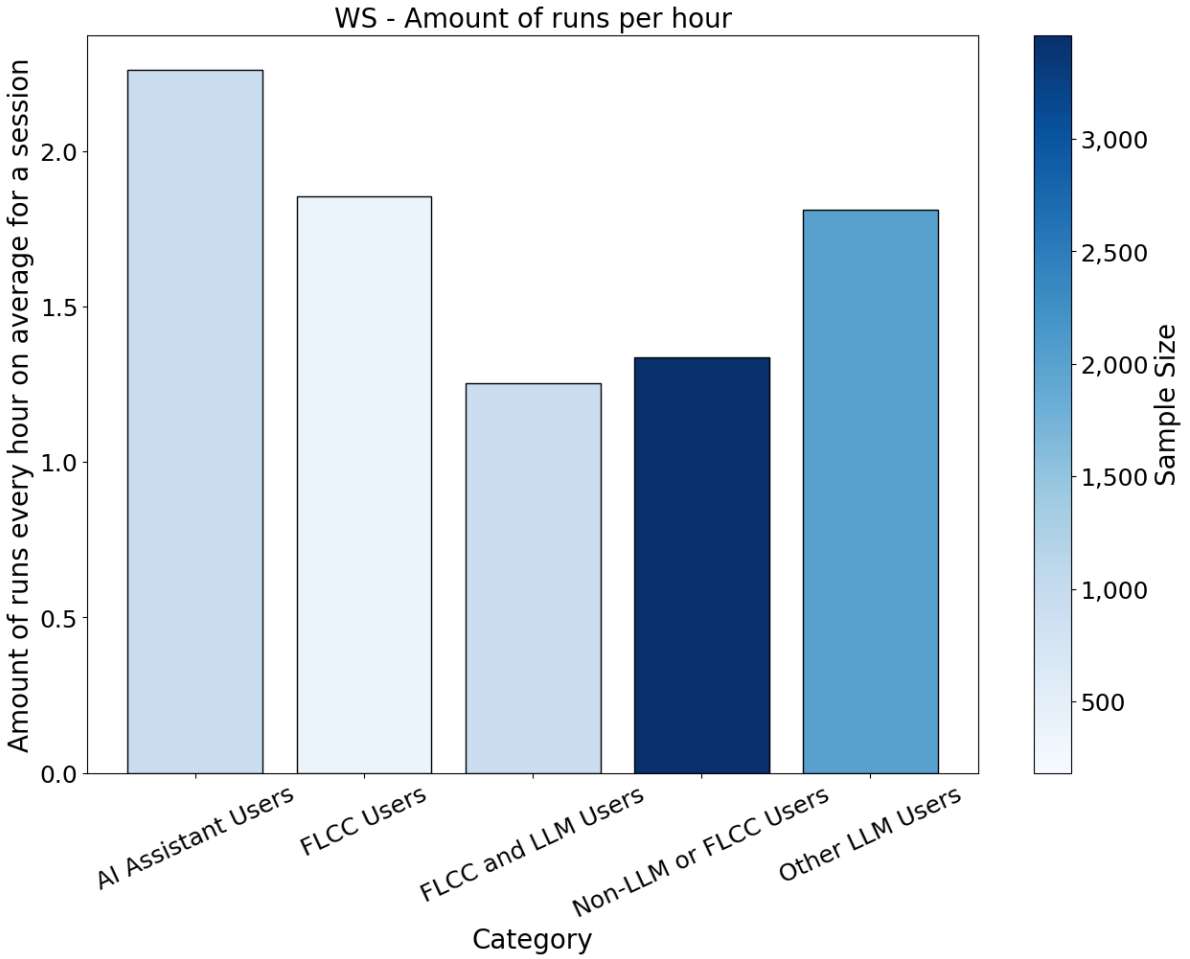


Figure 5.10: Code executions per session hour for WebStorm per user group



fully restart their builds.

RustRover amount of code execution
In Figure 5.11, the diagram is flat with our baseline user group as zero point. There is a small spike in code executions for FLCC users which is roughly a 10%-15% jump but this user group has a small sample size. The larger related user group of FLCC and LLM users has a smaller increase. However, the other user groups have marginal differences, topping out at 5%. The effect of LLMs on code executions in RustRover seems minimal compared to other IDEs, but the implications of this are discussed later in Section 5.2.

5.2.1. Summary

In short, the difference we see in code execution per hour depends on the IDE, some IDEs showcase a steep decline, others have a similar number of code executions, and others show a small increase for LLM-assisted users when compared to the baseline user group. However, the pattern taken from all the IDEs together is that code executions per hour decrease for users assisted by LLMs.

Figure 5.11: Code executions per session hour for RustRover per user group

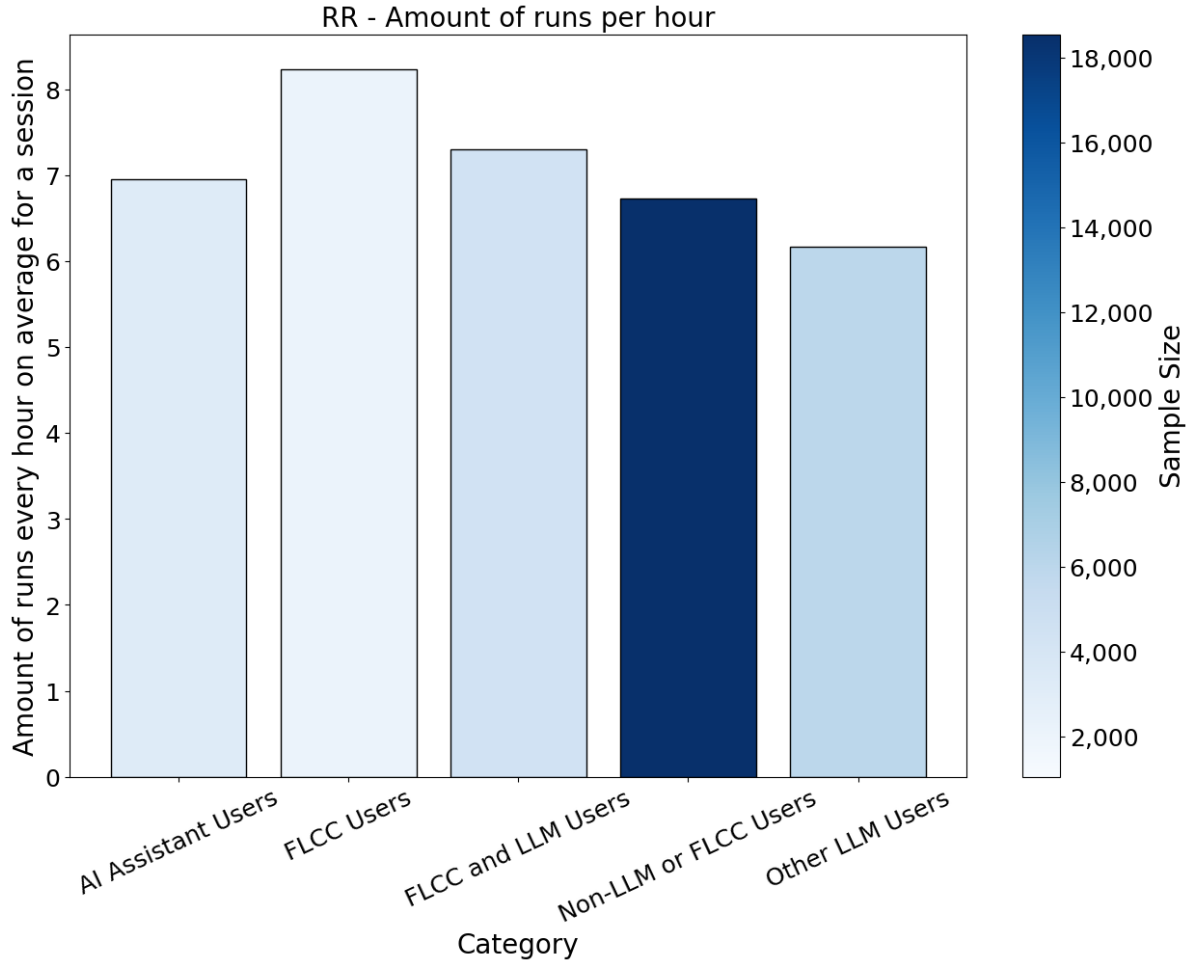
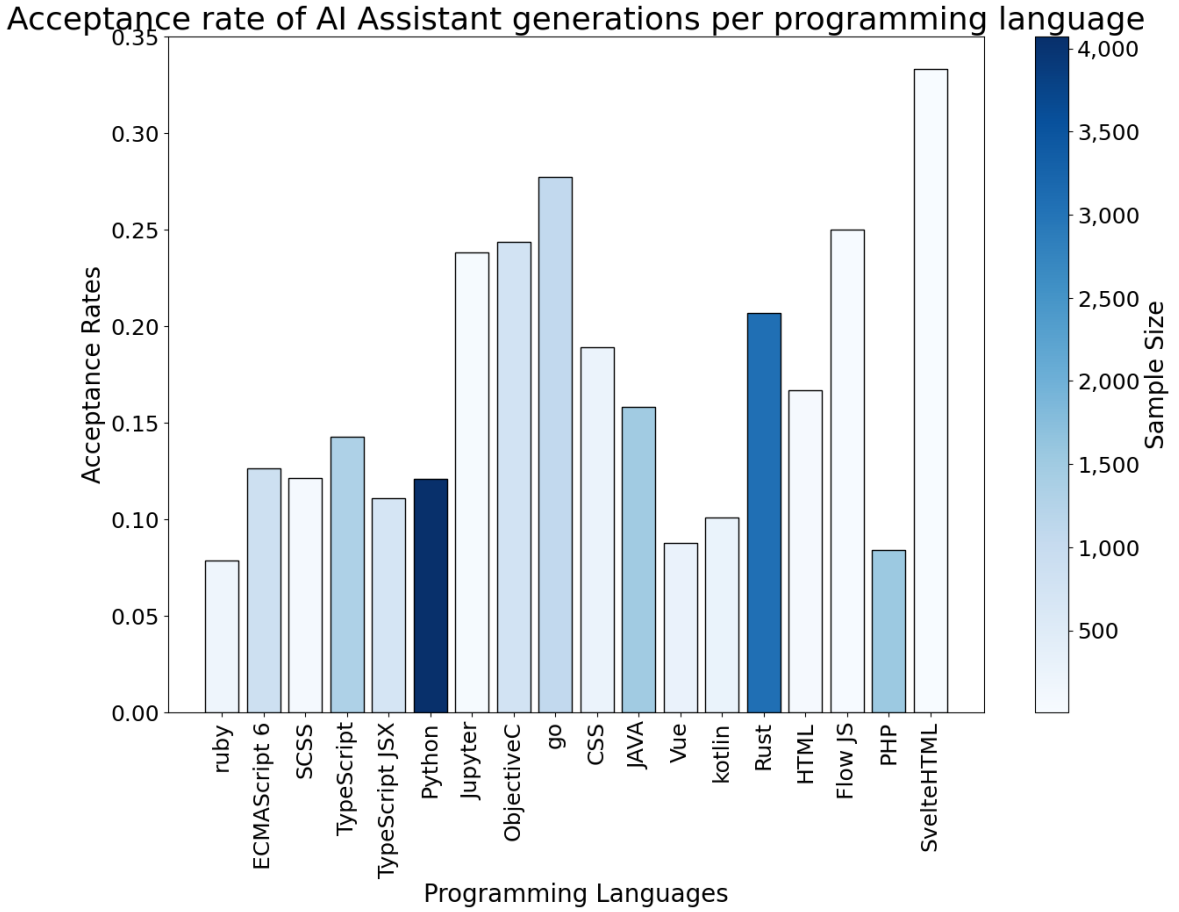


Figure 5.12: Acceptance rates for AI Assistant per programming language



5.3. RQ3: Differences in acceptance rates and benchmark performance of code generation concerning different programming languages

The benchmark we decided to compare to the acceptance rates of different programming languages is CrossCodeEval [9]. Note that the metrics for evaluating the models, edit similarity, and exact match, are metrics that are less advanced compared to newer metrics like CodeBLEU [30] or pass@k [7].

There does not seem to be an overall relationship between acceptance rates as seen in Figure 5.12 of users for specific programming languages compared to benchmarks in Table 5.1. We see Python with a very low acceptance rate, while Python is the largest dataset, with TypeScript there is an increase of around 0.03, which is a 30% increase. Such a large difference is not seen in the results from CrossCodeEval. The only thing that is inline with the benchmark is an improvement for Java compared to TypeScript and Python. For TypeScript, there is an improvement of 0.02-0.03 and almost 0.05 points compared with Python. This is a similar increase of 30% we see in the benchmark when comparing Java to Python. However, TypeScript performs relatively better than other languages when evaluating results from CrossCodeEval.

5.3.1. Summary

For AI Assistant there is no clear relation between the performance on CrossCodeEval [9], and the acceptance rate for the respective programming language. The relative differences between acceptance rates do not follow the same pattern as in the benchmark. Where in the benchmark Python and TypeScript perform similarly, AI Assistant has a greatly differing acceptance rate for TypeScript and Python.

Table 5.1: Table showcasing benchmark results of GPT-3.5-Turbo as taken from CrossCodeEval [9]. The results included are the languages that we also have acceptance rate data on, so Python, Java, and TypeScript.

Model for Code Match	Python		Java		TypeScript	
	EM	ES	EM	ES	EM	ES
GPT-3.5-Turbo + Retrieval w/ Ref.	15.72	58.88	22.72	68.50	14.15	58.40
Model for Identifier Match	Python		Java		TypeScript	
	EM	F1	EM	F1	EM	F1
GPT-3.5-Turbo + Retrieval w/ Ref.	23.49	50.14	31.79	60.52	20.65	51.54

5.4. Other findings from acceptance rates

In the coming sections, we showcase three interesting additional findings. First, we showcase the acceptance rates of FLCC and AI Assistant per programming language, comparing both. Second, we showcase the impact of country codes of the users on acceptance rates. Third, we showcase the acceptance rates of AI Assistants depend on the length of the suggestion.

5.4.1. FLCC and AI Assistant acceptance rates

In the results observed in Figure 5.13, we observe some major differences in acceptance rates, most in favor of FLCC. This is not due to some having lower sample sizes, the only large discrepancy is between Python, where FLCC has over 20,000 samples, but AI Assistant still has over 3,000 samples. The difference between FLCC and AI Assistant is also one of the largest ones we see.

5.4.2. Effects of Country Codes on acceptance rates

The existing consensus is that LLMs seem to have a bias for the more dominant spoken languages in the underlying training datasets, most notably English. With this in mind, we plot acceptance rates related to the user's country code, which gives us information on the locale and potential language they might use.

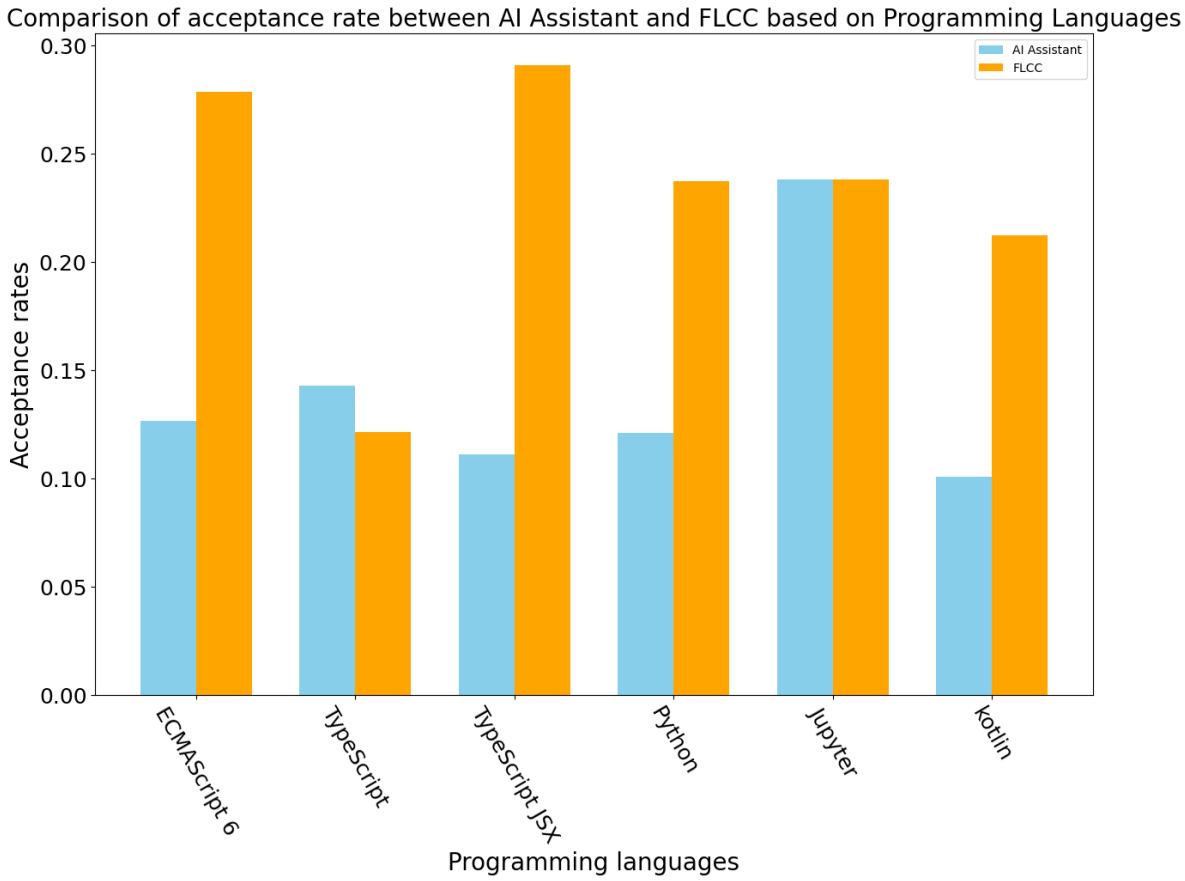
The result of this can be gathered from Figure 5.14, where the diagrams only include country codes with at least 500 samples. The first thing that stands out is that more English-aligned countries have a better acceptance rate, except for the British when it comes to AI Assistant. In general, FLCC seems to assist more users from different countries than AI Assistant. However in both cases, we see a clear drop in acceptance rate for Japanese, and in the case of FLCC an even larger drop in performance for Ukraine and Egypt.

5.4.3. Relation between suggestion length and AI Assistant acceptance rates

There were large differences between the acceptance rates of FLCC and AI assistant, as seen in Section 5.4.1, most favoring FLCC. This is interesting because FLCC uses a smaller LLM model compared to AI Assistant, where implicitly the expectation is that a larger model should have a higher acceptance rate. So what is the major difference between FLCC and AI Assistant? FLCC only suggests single-line completions, whereas AI Assistant can also suggest multi-line completions. To determine this we investigate if the acceptance rate changes based on the length of the suggested completions, this can be seen in Figure 5.15. Important to note is that we eliminated all acceptance rates that have less than 200 samples, this makes the diagram easier to interpret and removes the more noisy results.

What we see are two major things: acceptance rates do not follow a consistent trend line, and longer suggestions correlate with higher acceptance rates. But this is difficult to be certain about due to the large difference in sample sizes, and the longer completions have sample sizes under 1000. The other interesting result is that a major part of the suggestions is a single line, just like FLCC.

Figure 5.13: Acceptance rate comparison between FLCC and AI Assistant for the various IDEs they are supported in



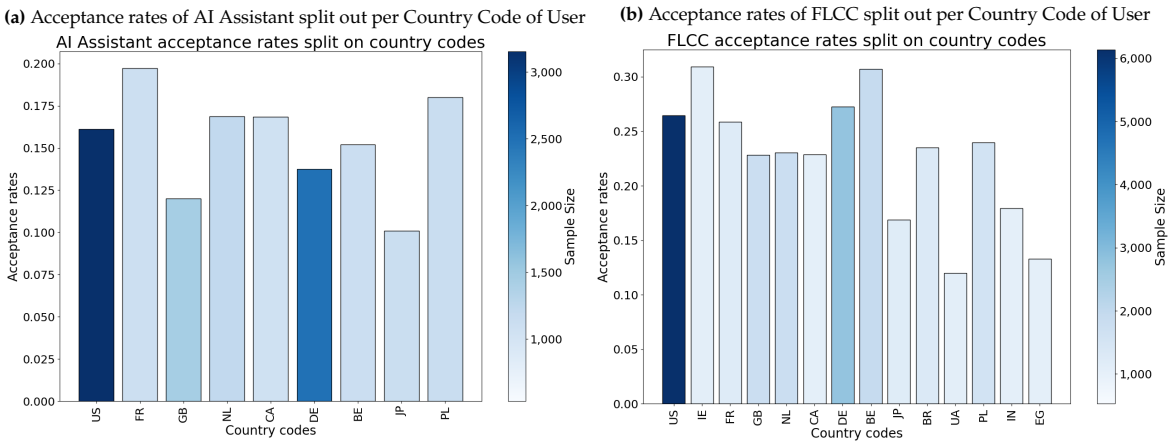
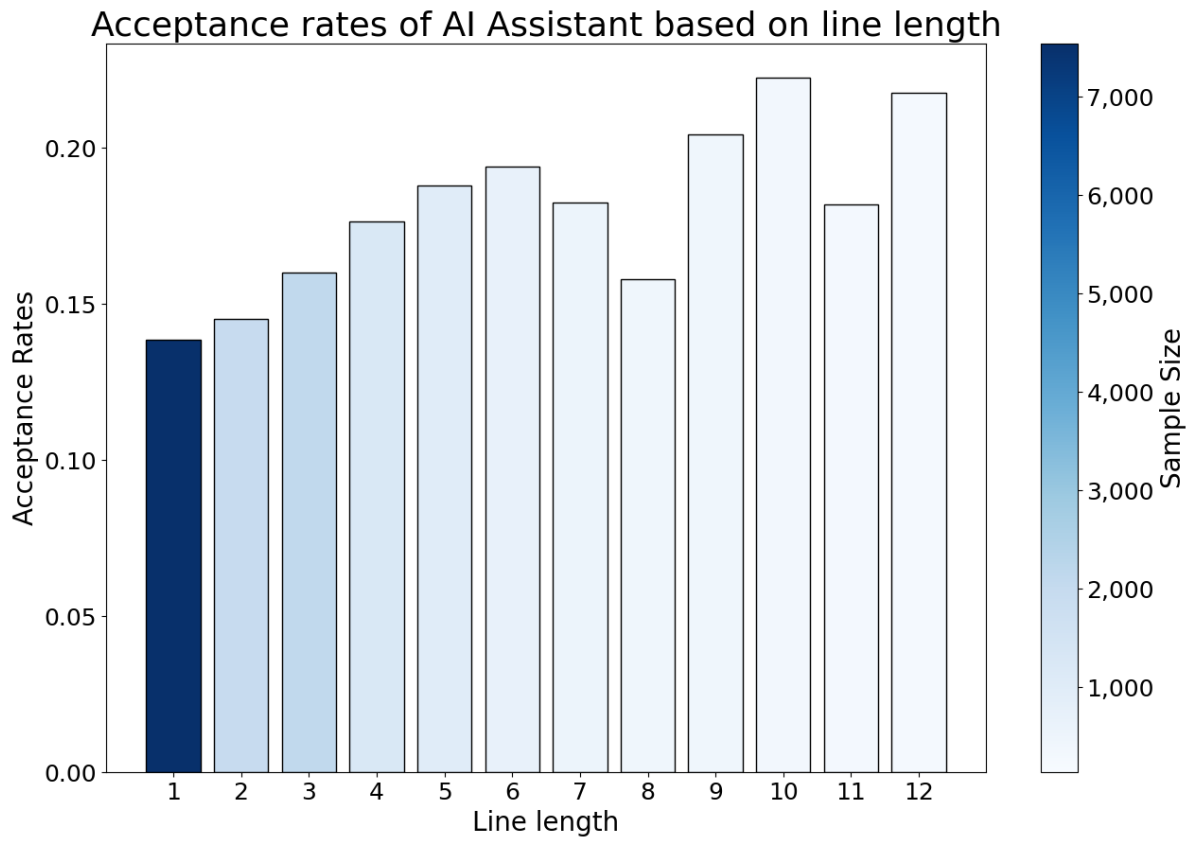


Figure 5.14: Acceptance rates per country code for both AI Assistant and FLCC

Figure 5.15: Acceptance rate of AI Assistant related to the length of the suggestion



6 Discussion

In this chapter, we interpret the results we showcased from our data analysis in Chapter 5. We follow the same ordering for discussing the results as we presented them, so we will first discuss research question one on user behavior in Section 6.1. This is followed by research question two, discussing the relation between LLM usage and code executions in Section 6.2. Then we interpret the results from research question three in Section 6.3, followed by a discussion in Section 6.4 on the additional findings we got from investigating the acceptance rates.

6.1. RQ1 - Differences in behavior of LLM/FLCC and normal users

Due to the copious amounts of results for this research question and for the sake of keeping things interesting, we will not individually discuss every IDE. Instead, we opt for a more generalized conclusion for the micro and macroanalysis in Section 5.1.1 and Section 5.1.2 respectively.

6.1.1. Lacking differences in major usage patterns as seen in the 3-gram analysis

The pattern we observe with the 3-grams, and in general with the complete n-gram analysis, is similar behavior between user groups. This is caused by the fact that we only look at the top 50 most frequent n-grams to enable us to interpret our Sankey diagrams. Because of this limited selection, small patterns, including actions that do not happen frequently, get lost. The results give us insight into the fact that the major workflow of users is not affected by LLM assistance, but sadly, they do not give us detailed information on user behavior for individual IDEs or user groups.

In the end, the comparative analysis of this micro behavior is not interesting, and it is difficult to compare the results or make solid conclusions. We explored comparison through the form state machines with 2-grams, as discussed in Section 4.5. The state machines make comparisons straightforward in the form of edit similarity, but this has its share of problems. It ignores the semantic similarity of nodes, and while not impossible to overcome, calculating edit similarity is an NP-Hard, or more formally APX-Hard [19] problem for graphs. Either forcing us to use approximations or limiting our graph size, where the latter returns us to the problem of potentially missing small differences between user groups.

Another measure to compare the state machines is a form of semantic similarity, overlaying the equivalent action nodes onto each other, and then comparing transition values. But seeing that this comparison was only a minor part of the complete analysis we opted for no longer pursuing this angle.

6.1.2. Longer session for users with LLM support, both positively and negatively interpretable

Looking at our generalized results in Figure 5.2, we determine that session lengths for user groups with LLM support are longer than those of the baseline user group. As alluded to in the title of this section, this can be interpreted positively or negatively. For now, we will look at these results in isolation but in Section 6.2 we combine multiple factors for a more holistic interpretation.

As for the session lengths of users assisted by LLMs, a positive reason for this could be that users are working longer because their frustration drops and blockers get removed more easily, this is supported by existing controlled studies [23]. Less frustration and blockers can allow users to get in a flow state [16] more easily, and for longer periods, increasing their session duration. However, this does raise an immediate question. If we assume our user groups professionally use their IDE, except users of the community editions. Why does session length increase, when professional use entails a fixed time window when users work on something? So, there are weaknesses to this hypothesis which would require further analysis.

The negative interpretation of these results is as follows: users with LLM support are slower to complete their tasks when compared to their baseline peers. Whether this is due to the LLM hampering the users or some sort of self-selecting bias where less strong programmers reach out to LLMs more early is unclear. Also, in general, this is a bit difficult to nail down exactly because when looking at metrics like file switches and typing actions per session hour we see an increase for most LLM-supported users. Therefore, if they are indeed hampered by their LLM, this would mean they are, however, doing more actions concerning typing and navigating compared to their baseline peers.

In general, credence can be given to both hypotheses, but these are not strong conclusions. To perfectly nail down the reasons behind the results we observe we will need further research, which will not be without its challenges, but this will be discussed in Section 9.1.

6.1.3. More typing but no conclusive difference in file switches for users supported by LLMs

In this section we discuss two of the macros together, typing behavior and file switch behavior. Because both are related in the form of navigation and editing, and both could be interpreted as a rough measure of the activity of the users.

First off, we observe that LLM-supported users make more typing actions per session hour. In tandem with that, typing speeds also increase. A hypothesis we have is that users are likely to spend more time typing and less time is spent on other activities, be that in or outside of the IDE. This result can be interpreted positively, as the users have an increase in the main activity in an IDE, typing. Yet, when playing devil's advocate, one could argue that this might be a symptom of users forced to fix mistakes in the generation of LLM-based suggestions.

A metric that was expected to decrease, especially for AI Assistant users, is the number of file switches. Because AI Assistant is capable of loading multiple files into its context window. Potentially eliminating the need for users to look up exact interfaces, method names, or constructors from other files. Something that existing completion providers are also partially capable of. Yet this hypothesis is exactly the opposite of the actual results: there are more file switches and faster file switches. First off when looking at it positively, this could indicate increased productivity, LLM-supported users are done more quickly in their files and then moving on to the next one to work on those. Yet this explanation might be a bit too naive. Another negative interpretation could be that the current LLM-based assistants give the users less benefit than expected. However, it is difficult to relate this to an increase in the number of file switches.

A side note to the results for file switches is the suspicion that this measurement is noisy. Some individual IDE boxplots show the median time spent in a file close to 1 second or sometimes even lower. This is a very low median of time spent in files, the time is so short that most humans would not even have the time to read anything. The hypothesis for this is that tools like debuggers or users that use a lot of 'go to definition' in a very short time, create this skew to very low time spent in a file. But this noisiness and unnatural result make it difficult to make a strong conclusion because it is likely we are not only catching the manual file-switching behavior we wanted to capture.

6.2. RQ2 - Code Executions and the use of LLM/FLCC

For our results, we will pick out two of the four individual IDEs we showcased the results for, PyCharm Professional and RustRover, which both give very opposite results. But before that, we discuss our thoughts on the general results for all the IDEs.

In general, when combining all the IDEs, we see that baseline users have more code executions than the other groups with LLM support tools. This can be negative or positive, so let us first start with two negative scenarios as to why this occurs and end it with a more positive spin with a recommendation for future work in this area.

As for the negative scenarios, the simple conclusion could be that users do not follow the recommendations of existing literature [15] where it is advised to do more testing when using LLM code snippets. Users do less testing, because they put more trust in the generated code, assuming that the code is most likely correct. This is a likely scenario, both from personal experience on how others use LLMs and how the use is portrayed and discussed in the general discourse on programming. But also by existing literature, where giving students access to LLM assistants in a learning environment on programming dropped the amount of testing with 8.6% as well as the tests that are created being useless in 78% of the cases [21].

For the next negative scenario, and here we will pull in other results, in particular Figure 5.2 one thing stands out. The baseline user group has a smaller session duration, partially affected by Community Editions only providing baseline and other LLM user groups. But this trend is repeated with all the other individual IDEs we see.

What is the importance of this? Well combining these facts with the result that LLM users also have fewer code executions per session hour, users might be negatively affected in their performance by the LLM support. The users might do fewer code executions because they spend more time to complete their tasks. However, this hypothesis is weak because existing work in the form of controlled studies suggests an improvement in task completion speeds [23]. On top of that, we do not see a huge drop in typing speeds, or amount of typing actions, which is another indicator of the speed people are working at. However, this would not take into account that users are spending a lot of time fixing mistakes created by the LLM code snippets. While the first negative scenario is more likely, nonetheless in general this whole topic is worth full investigation. But now onto a more lenient interpretation of the results.

A positive interpretation of the results is that users use code completions more like a scaffold generator where they edit parts of the code, revamping most of it. Or using only short completions, giving the users confidence in the correctness of what is generated. This is supported by the results of the completion lengths sample size we saw in Figure 5.15. But this does not fully explain the drop in code executions, because one would then expect users that users are less affected by the LLM tools, remaining on a similar level of code executions. Thus if this scenario is correct, this might hint at an actual reduction of productivity when compared to normal users, but this is only speculation without more in-depth data on what code is exactly written. So once again this is a great avenue for further research, potentially giving us interesting results on how LLMs affect productivity.

6.2.1. Lowered code executions per hour for LLM users in PyCharm Professional

In general, the results for PyCharm Professional in Figure 5.8 follow the general picture we framed above, except for the large outlier of the FLCC user group. This is most likely due to a very low sample size. Now, there is however, one interesting fact: Python acceptance rates for AI Assistants are very low, as seen in Figure 5.12; this might be a reason why we see a lower code execution rate. The AI Assistant users might be very 'picky' in what they accept, potentially hinting that the users feel they select only high-quality generations, therefore gaining more trust in the code that is written and foregoing testing.

This is a potentially sensible hypothesis, but this sensibility is quickly shattered by the fact that FLCC has an almost twice as high acceptance rate for Python see Figure 5.13. Yet when comparing the AI Assistant and FLCC and LLM user groups, which will stand in as a proxy for FLCC users, are equivalent to each other concerning the amount of code executions.

6.2.2. RustRover users see the minimal effect of LLMs on amount of code executions

RustRover is a different case from the other IDEs for which we saw the results. LLM support seems to have no impact on the number of code executions. In general, this is a bit of a puzzling result. Other results for this IDE follow similar trends of the other IDEs, sometimes showcasing some more of these egalitarian results, but nothing major can explain the similarity. FLCC users have a somewhat higher

amount of code executions, but the sample size is a bit smaller compared to the others. Combining them with the FLCC and LLM user groups would net us only a minor difference, nothing like the differences in other IDEs.

What could have happened is that in this case, we failed to capture some essence of manual/automated testing, however, the runs per hour are still remarkably high at around 7-8. This is in line with what we saw for PyCharm Professional. So this is possible but does not seem to be extremely likely. What might be the most likely conclusion we can make is that we have captured an increase in productivity or task completion speeds, but not an increase in code executions, so a net drop in testing. However this is defined on a shaky basis, these results could also just mean that RustRover users are spending more time cleaning up code generations from LLMs therefore being forced to type more. A more in-depth look as to why the users seem to be unaffected might be interesting.

6.3. Relation between acceptance rates and multi-programming language benchmarks

The results for this research question are more limited than initially predicted, this is due to the limited amount of multi-programming language benchmarks that allow comparative analysis of the same model for different programming languages. What can be said is that the performance on CrossCodeEval [9] for GPT-3.5-Turbo does not correlate to the acceptance rate for AI Assistant on the respective programming language. We could see these results for one of the following reasons: Either users are fine with accepting non-correct generations and editing them after the fact, thus using them to reduce the amount of typing required. Another case could be that benchmarks and their metrics fail to reflect real-world scenarios and how they perform with user input. Something that CrossCodeEval is specifically created to address because the authors felt that existing benchmarks did not reflect real-world usage. Lastly, a reason for this large difference is that the limitations and strengths of the OpenAI models are offset and hampered by the pre-and post-processing respectively in combination with prompt engineering that is done in AI Assistant. This could change the model's performance greatly, and more than initially expected when picking GPT-3.5-Turbo as a proxy for AI Assistant in the benchmark.

In the end, the hypothesis that benchmarks do not reflect real-world usage well is supported by existing literature. When comparing the performance of models on different languages in a real-world setting in this study [14] with results from [5].

6.4. Other findings

6.4.1. Acceptance rate disparity between FLCC and AI Assistant

Looking at the acceptance rate disparity between FLCC and AI Assistant it is easy to see that FLCC performs better in the languages that it is capable to generate completions for compared to AI Assistant. One reason why this is the case could be that FLCC generates only single-line completions, whereas AI Assistant can also generate longer completions of multiple lines.

Therefore we also had a look at the effect of suggestion lengths on acceptance rates. This can be seen in Figure 5.15, and showcased two things: there is a slight upward trend for longer suggestions, and the more major thing: AI Assistant predominantly suggests single line completions.

Why this is the case is unknown; this might be due to post-processing done before showing a suggestion. But this might also be because the model is optimized for shorter suggestions, either through the prompts provided. Or as is suggested in other literature, evaluating against acceptance rate optimizes for shorter suggestions [24]. This is because statistically, shorter suggestions are easier to correct.

If shorter suggestions are problematic is difficult to tell, single line completions are easy for the user to grasp and check for correctness. But if the acceptance rates for longer suggestions are representative, longer suggestions could net higher acceptance rates or at least equal acceptance rates. While delivering a large productivity boost while the programmer will have to type less.

As for the question proposed at the beginning of this subsection, why does FLCC perform better compared to AI Assistant on single line completions? One of the reasons could be the smarter invocation of the model, reducing the amount of unwanted suggestions, therefore increasing the acceptance rate. A

qualitative analysis of when suggestions are given comparing FLCC and AI Assistant would be necessary to resolve this. But the model potentially might just suggest more relevant single line completions compared to AI Assistant due to its limited context window. Whereas FLCC only suggests to users what is relevant in the current file, being more of a quick scaffolding tool compared to full logic generation.

6.4.2. Less English aligned country codes, except the British, having lowered acceptance rates

A more novel result, which is not included in most literature, is the acceptance rate split out on country code. Note country code and not locale, introducing the possibility that this data is noisy due to that some groups include multiple different locales and/or not the ones expected from the country code. Now sample sizes are not enormous for most user groups except the United States and Germany, as seen in Figure 5.14 both for AI Assistant and FLCC, another factor to keep in mind for these results.

One thing that stands out is that potentially more English-aligned countries, and users of the Latin script, have a significantly higher acceptance rate. Excluding some outliers like the British when using AI Assistant, they have a similar acceptance rate to Japanese, however, this is not reflected in FLCC. German also has a lower acceptance rate in AI Assistant than the US-based users, which again is not seen with the results of FLCC. The fact that FLCC seems to suffer less when faced with different country codes is interesting, with it being a smaller model it is normally expected to generalize poorly compared to larger models. One reason why it could be less affected is that due to its small context window, fewer non-English words in the form of comments, and variable/function names make it into that context window. Something that would affect AI Assistant more, or it could just simply be that the FLCC users in general use more English.

One thing that is more reliable to claim is that non-Latin script users have lower acceptance rates, These include Japanese, Ukrainian, Egyptian, and to a certain extent Indian, although including the latter is difficult because this can hint at 25 to 50 different locales[1] as defined by the Indian government. But we see a significant drop in acceptance rates, something which is not unexpected and backed up by earlier works on general LLM performance for different languages [17, 36, 26]. To conclude, the results here could indicate a concerning underlying problem where non-English-aligned users get a poorer performing assistant, harming the potential benefits they receive compared to their English or at least Latin script-using peers.

[1] https://www.education.gov.in/sites/upload_files/mhrd/files/upload_document/languagebr.pdf

7

Threats

Although sufficient care and precautions were taken to mitigate threats to the research, the following threats to this research were identified.

7.1. IDE differences

Using the data from the various JetBrains IDEs means that some difference in results might be caused by the difference between the individual IDEs instead of purely the performance of the AI assistant or FLCC on each aspect. Where obvious we separated results of individual IDEs and compared them in the user groups of the individual IDE.

However, the broader analysis can still be skewed when looking at the results of the complete dataset. Whenever we think this is the case we mention this in the presented results. This threat is of particular importance when looking at user behavior. Because someone using DataSpell or DataGrip has different aims and different workflows when compared to more standard code editing IDEs like PyCharm or IntelliJ.

This is also seen in the different languages that are standard to be used with particular IDEs, different actions occur for different languages. Rust actions show up in RustRover but not in most others, just like Python actions show up in PyCharm and DataSpell but are almost absent in others. Yet it is important to note that these are marginal amounts of actions when compared to the bulk, which is typing and editing.

The evaluation of the AI assistants is not the pure evaluation of the underlying LLMs, but also the prompt-engineering done to augment the AI assistant of JetBrains. So the results shown are based on the complete AI assistant and not just the LLM.

7.2. External LLM tools

A major threat to this research is that we cannot check or control what users do outside of their IDE environments. This means we cannot be sure that users do not interact with LLMs outside of their IDE. The one major tool that comes to mind is of course ChatGPT[1], this is mostly accessed in the web browser and can be used for coding. However, it is of course a chat interface and not necessarily a code completion approach. So in general this should only be a threat for the first two research questions that are looking at user behaviour. The latter one focused on completion rates.

In general, we cannot eliminate this issue, and in the end, we then should assume that our non-FLCC and LLM users are the baseline users and they can still use LLM tools but outside of the IDE environment. The observations we make and discussion points we raise then still hold when looking at efficiency or performance.

[1] `https://openai.com/index/chatgpt/`

7.3. Missing or misattributed actions to user groups

Because there are over 4000 unique actions it is very possible that we missed some particular actions that should qualify a user to be part of a different one. To mitigate the threat we checked the same list on multiple occasions and made use of a self-developed search script to look at the keywords associated with individual tools and user groups, to ensure that as many keywords as possible were included. With that, we also were in contact with JetBrains members to verify if there were any more pointers or actions that could help determine a user group.

7.4. Acceptance rates that do not take deletions into account

This is a large limiting factor on the collected acceptance rate measures. It is unclear whether or not users immediately delete their undo their accepted generations, this was not included in the analysis. Therefore the showcased acceptance rates may paint too much of a positive picture. This is something that should be kept in mind when viewing these results. However both AI Assistant and FLCC have then more positive results.

7.5. Selected benchmark results do not translate to AI Assistant

Due to the confidential nature of AI Assistant, it is not transparent which OpenAI model is used. Most likely it is of the GPT-3 family, which is why this result from CrossCodeEval [9] was selected. On top of this uncertainty, AI Assistant does preprocessing and prompt engineering before sending a request to the LLM, as well as post-processing. However in CrossCodeEval [9] does add context of multiple different files, replicating the processes done by AI Assistant. Nevertheless, this difference between the two still makes for a potential threat.

8

Conclusion

To conclude, we have answered the three research questions we defined. The first research question is on the difference in user behavior between user groups with and without LLM support. With that, we see longer sessions, more typing per hour, and slightly faster typing. Depending on the IDE we observe a varying amount of file switches, where sometimes the LLM-supported users do more file switches, and sometimes the users without LLM support. Showcasing interesting differences which can be interpreted positively or negatively, but based on other literature hint towards an improvement for the users with LLM support.

The second research question was on the number of code executions per session hour, where the code executions stand as a proxy for (manual) testing. In general, the number of code executions decreases for users supported by LLMs. For some IDEs, we see an equal number of code executions, but those are only a few. This result could be interpreted positively, but most likely users put too much trust in the LLM code suggestions, thus foregoing testing.

The third and last research question was on the relationship between acceptance rates of AI Assistant for different programming languages compared to benchmarks that compare different programming languages. The conclusion to this question is that there is a limited amount of benchmarks that compare programming language performance for LLMs. And that there is no clear relationship between the acceptance rates of AI Assistant and the performance of GPT-3.5-Turbo the model that it uses. The reason for this might be that the benchmark does not replicate the real-world scenarios correctly, or AI Assistant's pre- and post-processing and prompt engineering have a significant effect on the performance of the underlying model.

We also had some additional findings, showcasing that JetBrains AI Assistant prefers shorter suggestions, and mostly single line completions. On top of that, FLCC outperforms AI Assistant on these single line completions in almost all scenarios. The country code of the user impacts acceptance rates heavily when the language of those country codes does not use the Latin script.

All these findings also gave us insights into potential future works, because we have a large number of them we discuss these in the upcoming chapter.

9

Future Works

9.1. Investigating why session length increases but code executions drop

In Section 6.1 we discuss what caused the change in session lengths for users supported by LLMs and in Section 6.2 the impact of LLMs on amount of code executions. Although the discussion about potential causes is interesting, it was not possible to indicate if this increase in session length is beneficial for the users' workflow and productivity. It is difficult to nail this down exactly with the current data.

Because of this difficulty in gauging the effect of this increase in session length, we make the case for an in-depth look at the potential causes of this change, either through user surveys or potentially collecting more data on a smaller group of developers. The importance lies in tracking the user in more real-world scenarios, eliminating most controlled experiments. Because in user studies it becomes apparent that users identify themselves as being faster, thus shorter 'sessions' as a result [23].

What we want to investigate is if sessions are longer because of users staying more in a so-called "state of flow" [16] where programming is perceived as more effortless.

9.2. Does testing decrease when assisted by LLMs

As for the reduced amount of code executions, this is more surprising. Literature [15] warns against thoughtlessly integrating generated code, instead advising to add more tests. One would assume more added tests would require more executions, and if one does not add more automated tests, at least the user would do more manual testing.

To confirm if people are writing fewer or more tests it would be nice to have a smaller group of people who share more information about what files they are working on. Potentially what code they are writing, as well as what the LLMs are suggesting. This makes it possible to see if people are spending more time writing tests or not. Then in the end this research can serve as an example for LLM tool developers of how users interact with their tools. This allows them to either add more safeguards or focus more on the correctness of code generation, reducing the amount of testing necessary, or creating the tools in such a way that encourages more testing.

9.3. A more multi-programming language benchmark landscape

A future work not necessarily directly related to this research but stemming from a need for evaluations is a broader look at benchmarks comparing model performance between programming languages. Right now the field is small with only three major benchmarks: MultiPL-E [5]a, MultiEval[2], CrossCodeEval[9], which are barely used.

The problem here is not even necessarily the lack of benchmarks, but mostly the use of them, for some reason evaluation against these benchmarks does not happen often. Retroactively benchmarking models

do not result in published papers, so the interest of the scientific community is low. So making it easier to run these multilingual benchmarks would be a great first step. Combining that with a more thorough benchmark in the style of CrossCodeEval would at least be beneficial for the greater community.

9.4. Deeper analysis of micro user behavior

In our microanalysis, we mostly focused on comparative analysis, limiting us in what we investigated per user group. There might be a lot of interesting results when investigating what happens before and after users do particular LLM actions, either accepting generations, using the chat interface, asking for commit summaries, etc. This research should be relatively low-hanging fruit, most of the data and processing are already in place, and actions related to AI Assistant and FLCC have already been identified.

9.5. Investigate if AI Assistant optimizes for single line code completions

What we saw in the results of acceptance rates split out per line length in Figure 5.15 is that AI Assistant mostly generates single-line completions. It might be interesting to see if this is caused by the underlying LLM, something in the pre-and post-processing, or the prompt engineering.

This could potentially be studied by using an ablation study by letting the LLMs generate code for a large sample of coding exercises and removing parts of AI Assistant looking at the impact of what the suggestion length is. Or another way would be to do some A/B testing using AI Assistant and disabling parts of it, instead of using code exercises. The benefit of the latter is that this research then approaches the real world more closely.

9.6. Investigate completion suggestion invocation between FLCC and AI Assistant

In our discussion on the performance difference between FLCC and AI Assistant, we raised the possibility that the acceptance rate could be different because AI Assistant suggestions are provided at a time when the user does not need or want them. It could be interesting to analyze the two, this could potentially exist out of a qualitative study on when invocations take place. This qualitative study could then be combined with a comparative analysis, where invocation strategies are switched between FLCC and AI Assistant.

9.7. Controlled study of Locale impact on code generations

Programming communities with languages different from English seem to gain less from LLMs in their development workflow. One reason for this is that prompting in one language and possibly providing comments in a different one might confuse the model. Or in another case, it might be beneficial to use models that are fine-tuned on the language used. This creates an avenue for an ablation study showcasing the influence of these factors.

9.8. Looking at factors in manual invocations for users

This topic has been researched more often and a proper body of work already exists [24, 22]. However, it might be interesting to also look at real-world examples of manual invocations. Which have a higher acceptance rate compared to automatic code suggestions. Align with what is found in existing works.

There might be potential for more heuristics as to when to suggest a code completion by an LLM. Optimizing these invocations would be beneficial for the companies offering their products if they use their hardware to run the LLMs. This is also beneficial for the users, as they will potentially have a decreased amount of unwanted suggestions. And when the LLMs are running locally on the device it will use fewer system resources.

10

Reproducibility

Due to the confidential nature of some of the source data, it is not possible to fully replicate the study. However all the resulting diagrams are stored in a public GitHub repository[1].

For those with access to confidential data, the code written to sanitize, process, and analyze is available on the internal JetBrains repository. To ensure others can easily execute the code there is a *requirements.txt* that can be used together with pip. For nearly complete reproducibility there also is a *nix* [10] *flake* included. One only needs to use *nix-direnv*[2] and with that it will automatically load a Python virtual environment, with the correct Python and pip versions. This then allows users to install the requirements defined in the *requirements.txt* needed to run the code. Of course, full information can be found in the repositories README.

If users wish to go further it is possible to replicate the entire underlying OS can be replicated by using the personal NixOS [11] configuration[3].

All these factors combined should make it reliable for others to replicate the study, but realizing that Nix(OS) is not a popular package manager let alone OS. Even without this, it is still possible to almost completely reproduce the environment used by inspecting the *flake.nix* file. The specifications in the flake, allow replicating underlying major dependencies like Python versions. This is something that is not often seen with replication packages provided within the field. We hope that this can serve as an example to others on how to handle their replication packages. Or if nothing else to make them aware of the importance of explicitly defining dependencies for their projects, and using tightly bounded version specifications in package managers like pip, Gradle, etc. Something that has been a gripe with other studies and their reproducibility packages.

[1] `https://github.com/RemcoSchrijver/Artifacts-Beyond-Acceptance-Rates-The-Impact-of-JetBrains-AI-Assistant-a`
and if unavailable this fork
`https://github.com/AISE-TUDelft/Artifacts-Beyond-Acceptance-Rates-The-Impact-of-JetBrains-AI-Assistant-and-FL`

[2] An improved version of direnv tailored for nix usage, more can be found out here `https://github.com/nix-community/nix-direnv`

[3] Personal NixOS configuration tagged to replicate the thesis environment: `https://github.com/RemcoSchrijver/nix-config/tree/Thesis`

References

[1] Owura Asare, Mei Nagappan, and N. Asokan. *A User-centered Security Evaluation of Copilot*. 2024.

[2] Ben Athiwaratkun et al. *Multi-lingual Evaluation of Code Generation Models*. 2023. arXiv: 2210.14868 [cs.LG].

[3] Moritz Beller et al. "When, how, and why developers (do not) test in their IDEs". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 179–190. DOI: 10.1145/2786805.2786843. URL: https://doi.org/10.1145/2786805.2786843.

[4] Egon Börger. "The Abstract State Machines Method for High-Level System Design and Analysis". In: *Formal Methods: State of the Art and New Directions*. Ed. by Paul Boca, Jonathan P. Bowen, and Jawed Siddiqi. London: Springer London, 2010, pp. 79–116. DOI: 10.1007/978-1-84882-736-3_3. URL: https://doi.org/10.1007/978-1-84882-736-3_3.

[5] Cassano et al. "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation". In: *IEEE Transactions on Software Engineering* 49.7 (2023), pp. 3675–3691. DOI: 10.1109/TSE.2023.3267446.

[6] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374 [cs.LG].

[7] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[8] Rudrajit Choudhuri et al. *How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering*. 2024.

[9] Yangruibo Ding et al. *CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion*. 2023. arXiv: 2310.11248 [cs.LG].

[10] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and Policy-Free System for Software Deployment". In: *Proceedings of the 18th USENIX Conference on System Administration*. LISA '04. Atlanta, GA: USENIX Association, 2004, pp. 79–92.

[11] Eelco Dolstra and Andres Löh. "NixOS: a purely functional Linux distribution". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 367–378. DOI: 10.1145/1411204.1411255. URL: https://doi.org/10.1145/1411204.1411255.

[12] Xueying Du et al. *ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation*. 2023. arXiv: 2308.01861 [cs.CL].

[13] Xueying Du et al. "Evaluating Large Language Models in Class-Level Code Generation". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. DOI: 10.1145/3597503.3639219. URL: https://doi.org/10.1145/3597503.3639219.

[14] Maliheh Izadi et al. *Language Models for Code Completion: A Practical Evaluation*. 2024. arXiv: 2402.16197.

[15] Kevin Jesse et al. *Large Language Models and Simple, Stupid Bugs*. 2023. arXiv: 2303.11455 [cs.SE].

[16] Anne Landhäußer and Johannes Keller. "Flow and Its Affective, Cognitive, and Performance-Related Consequences". In: *Advances in Flow Research*. Ed. by Stefan Engeser. New York, NY: Springer New York, 2012, pp. 65–85. ISBN: 978-1-4614-2359-1. DOI: 10.1007/978-1-4614-2359-1_4. URL: https://doi.org/10.1007/978-1-4614-2359-1_4.

[17] Bryan Li et al. *Eliciting Better Multilingual Structured Reasoning from LLMs through Code*. 2024. arXiv: 2403.02567 [cs.CL]. URL: https://arxiv.org/abs/2403.02567.

[18] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. *A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges*. 2023. arXiv: 2303.17125 [cs.SE].

[19] Chih-Long Lin. "Hardness of approximating graph transformation problem". In: *Algorithms and Computation*. Ed. by Ding-Zhu Du and Xiang-Sun Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 74–82. ISBN: 978-3-540-48653-4.

[20] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[21] Simone Mezzaro, Alessio Gambi, and Gordon Fraser. "An Empirical Study on How Large Language Models Impact Software Testing Learning". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 555–564. DOI: 10.1145/3661167.3661273. URL: https://doi.org/10.1145/3661167.3661273.

[22] Aral Moor, Arie Deursen, and Maliheh Izadi. *A Transformer-Based Approach for Smart Invocation of Automatic Code Completion*. May 2024.

[23] Hussein Mozannar et al. *Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming*. 2024. arXiv: 2210.14306 [cs.SE].

[24] Hussein Mozannar et al. *When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming*. 2024. arXiv: 2306.04930 [cs.HC].

[25] David OBrien et al. *Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot*. 2024.

[26] Millicent Ochieng et al. *Beyond Metrics: Evaluating LLMs' Effectiveness in Culturally Nuanced, Low-Resource Real-World Scenarios*. 2024. arXiv: 2406.00343 [cs.CL]. URL: https://arxiv.org/abs/2406.00343.

[27] Rangeet Pan et al. *Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code*. 2023. arXiv: 2308.03109 [cs.SE].

[28] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. arXiv: 2204.06125 [cs.CV]. URL: https://arxiv.org/abs/2204.06125.

[29] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. arXiv: 2204.06125 [cs.CV]. URL: https://arxiv.org/abs/2204.06125.

[30] Shuo Ren et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: 2009.10297 [cs.SE].

[31] Semenkin et al. "Context Composing for Full Line Code Completion". In: *arXiv preprint arXiv:2402.09230* (2024).

[32] Mohammed Latif Siddiq et al. *Quality Assessment of ChatGPT Generated Code and their Use by Developers*. 2024. URL: https://s2e-lab.github.io/preprints/msr_mining_challenge24-preprint.pdf.

[33] Ching Y. Suen. "n-Gram Statistics for Natural Language Understanding and Text Processing". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1.2 (1979), pp. 164–172. DOI: 10.1109/TPAMI.1979.4766902.

[34] Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin. *ChatGPT Incorrectness Detection in Software Reviews*. 2024.

[35] Yue Zhang et al. *Does Generative AI Generate Smells Related to Container Orchestration?: An Exploratory Study with Kubernetes Manifests*. 2024.

[36] Jun Zhao et al. *LLaMA Beyond English: An Empirical Study on Language Capability Transfer*. 2024. arXiv: 2401.01055 [cs.CL]. URL: https://arxiv.org/abs/2401.01055.

[37] Albert Ziegler et al. *Productivity Assessment of Neural Code Completion*. 2022. arXiv: 2205.06537 [cs.SE].