Towards LLM-Generated Code Tours for Onboarding

Martin Balfroid
martin.balfroid@unamur.be
NADI, University of Namur
Namur, Belgium

Benoît Vanderose
benoit.vanderose@unamur.be
NADI, University of Namur
Namur, Belgium

Xavier Devroey
xavier.devroey@unamur.be
NADI, University of Namur
Namur, Belgium

ABSTRACT

Onboarding new developers is a challenge for any software project. Addressing this challenge relies on human resources (e.g., having a senior developer write documentation or mentor the new developer). One promising solution is using annotated code tours. While this approach partially lifts the need for mentorship, it still requires a senior developer to write this interactive form of documentation. This paper argues that a Large Language Model (LLM) might help with this documentation process. Our approach is to record the stack trace between a failed test and a faulty method. We then extract code snippets from the methods in this stack trace using CodeQL, a static analysis tool and have them explained by `gpt-3.5-turbo-1106`, the LLM behind ChatGPT. Finally, we evaluate the quality of a sample of these generated tours using a checklist. We show that the automatic generation of code tours is feasible but has limitations like redundant and low-level explanations.

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**; • **Software and its engineering** → **Software development process management**.

KEYWORDS

large language models, code tour, developer onboarding

ACM Reference Format:
Martin Balfroid, Benoît Vanderose, and Xavier Devroey. 2024. Towards LLM-Generated Code Tours for Onboarding. In *2024 ACM/IEEE International Workshop on NL-based Software Engineering (NLBSE '24), April 20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3643787.3648033

1 MOTIVATION

For most software projects, successfully onboarding new developers is crucial to guarantee continued productivity. However, each project has unique tools, practices, and codebase peculiarities that can overwhelm newcomers [6, 8, 12]. Moreover, poor documentation and limited project knowledge further hinder the process [12].

Current onboarding strategies, such as assigning mentors, providing training, and documentation[12], are limited as they divert human resources from development activities. It has been suggested that annotated code tours might help developers navigate a new codebase more efficiently. However, writing code tours remains time-consuming and automating this onboarding approach would ease its adoption. We argue that LLMs might help generate code tours since they are already used for generating documentation and explaining code [11, 13]. In this paper, we look at 1) the extent to which we can automatically generate such code tours; and 2) the strengths and limitations of using LLMs for generating code tours. We also provide a replication package [2].

NLBSE '24, April 20, 2024, Lisbon, Portugal

2 BACKGROUND AND RELATED WORK

Onboarding. Onboarding is the process of transitioning new employees into members of the organisation [3]. Lack of documentation, unfamiliarity with the team's work methodology, and unfamiliarity with technologies and tools are common obstacles for onboarding [12]. Assigning mentors, providing training, making documentation available, and monitoring newcomers' progress is recommended [12]. Additionally, comprehensive documentation, regular feedback, and a supportive learning environment should be provided to improve learning, confidence, and socialisation [8]. Early task experimentation, understanding project structure and culture, and regular progress validation are crucial for successful integration [6]. Taylor and Clarke [17] show that, while possibly limiting the broader exploration of a codebase, code tours (i.e., a sequence of interactive steps that describe the code) help understand and navigate codebases. CodeTour [5] is a plugin for Visual Studio Code that allows the creation of such tours

As related work highlights, documenting and guiding newcomers is crucial for onboarding. But documentation is also a time-consuming task that requires some automation in a time-sensitive context. Large Language Models (LLMs) are highly versatile for natural language processing (NLP) tasks. They are initially pre-trained on extensive datasets but may be fine-tuned for specific tasks [4], which makes them good candidates to accomplish said automation.
Automated documentation. Nam et al. [13] explore an in-IDE tool using LLM to aid in understanding and writing code. While the tool enhances task completion by allowing users to complete more sub-tasks compared to traditional web searches, it does not significantly improve task completion speed or deepen code understanding. User feedback indicates that the tool is perceived as more useful and user-friendly than web searches, particularly because of code context in responses. However, the tool's effectiveness varies among users, depending on their background and experience, with some suggesting that combining it with other resources like search engines or API documentation could enhance its utility. MacNeil et al. [11] investigates the use of LLM-generated code explanations for web software development education. It reveals that while students find these explanations helpful, their engagement varies based on explanation type and code complexity. The study notes some issues with relevance and detail in LLM outputs.

To the best of our knowledge, no attempt to use an LLM to generate documentation that displays the gradual quality of code tours

has been reported yet. Although we have chosen an LLM-based approach because of the growing interest in LLM, there are alternatives such as term-based, template-based, external description-based, and machine learning-based summaries for generating natural language descriptions from source code [21].

3 PRELIMINARY EVALUATION

Due to the limited context length, it is impossible to provide the entire codebase so that the model can define the tours itself. While this may be possible with small codebases or models where users can input up to 100,000 tokens [1], it is not advisable. Indeed, it has been shown that the longer the prompt, the less able the model is to retrieve information from the middle of the context [10]. Our first research question explores how code tours can be generated effectively for tasks commonly used for onboarding new developers, i.e., bug-fixing [8]: **RQ.1** *To what extent can we automatically generate code tours for debugging?* For that, we rely on a finer-grained definition of code tour: a code tour is a sequence of steps, each explaining a method between a failed test and a faulty one. An excerpt of a code tour is found in the replication package [2], along with a more illustrative pipeline description.

Our second research question investigates the quality of the generated tours: **RQ.2** *How effective are the explanations?* To assess the quality of code explanations, we adapt three of the seven goals proposed by Tintarev and Mashtoff [18] from recommendation system explanations to code explanations: (1) To what extent does the code tour provide the new developer with a clear understanding of how the code works? (**Transparency**) (2) To what extent does the code tour allow the new developer to identify and report inaccuracies in the code explanations and the code? (**Scrutability**) (3) To what extent does the code tour facilitate quicker comprehension and navigation of the codebase? (**Efficiency**)

3.1 Setup and tour generation

For this case study, we use the well-known Defects4j dataset [9], which contains a diverse and extensive range of actual bugs. It has a convenient command-line interface to switch between fixed and faulty versions and run tests. More specifically, we use the data collected by Sobreira et al. [16] to obtain information on the faulty method's location across five projects: Chart, Time, Math, Lang, and Closure, totalling 357 buggy versions of these projects. This saves us wasting time calculating it ourselves, although it is possible to do so. Code tours are based on the stack traces generated by the tests; one faulty method potentially generates multiple stack traces if multiple tests fail. We use the CodeQL static analysis tool to extract code snippets and a Large Language Model (LLM) to explain them:

(1) **Initialisation.** First, each buggy version of the project is checked out, and a CodeQL database is created. We rely on CodeQL to easily extract the faulty methods with their documentation (21 versions failed at this step and were discarded).
(2) **Recording stack traces.** For each buggy version, we execute the tests and record the generated stack traces (6 buggy versions failed at this step). Out of 1736 pairs of failing tests and faulty methods, 752 (43%) were direct calls between the test and the method (so no stack trace to record), and for 64 (4%), the test running failed.

Table 1: Stack traces, each stack trace is used as an input to generate a code tour.

	Generated	Successful	Selected	Sampled
Chart	147	87 (59%)	6 (4%)	4 (3%)
Closure	593	501 (84%)	35 (6%)	4 (1%)
Lang	212	116 (55%)	10 (5%)	4 (2%)
Math	580	146 (25%)	10 (2%)	4 (1%)
Time	204	63 (31%)	8 (4%)	4 (2%)
Tot.	**1,736**	**913 (53%)**	**69 (4%)**	**20** (1%)

(3) **Extracting methods.** For each method appearing in a stack trace, we run a CodeQL query to extract the methods' start and end lines. Seven cases failed at this step, which leaves us with 913 (53%) stack traces successfully generated (details per project are available in the second and third columns of Table 1).
(4) **Selection.** As we used the OpenAI API with model `gpt-3.5-turbo-1106`, the generation is not free (Input: $1 per 1M tokens and Output: $2 per 1M tokens as of now), so for this preliminary evaluation, we decided only to generate code tours based on a subset of stack traces selected using the following criteria: (i) the number of steps of a stack trace falls within the interquartile range of the number of steps observed in all stack traces from the same project; (ii) at least half of the steps are not already in another selected stack trace. Depending on the selection order, this may result in a different sub-set. We have used an alphanumeric order according to the project, version number and tour name; (iii) both the test and faulty method are unique. Which makes 69 stack traces, each used to generate a tour (see fourth column of Table 1).
(5) **Tour generation.** For each method and its corresponding code snippet appearing in a stack trace, we prompt the OpenAI API. We use the chat completion API from OpenAI with the model set to `gpt-3.5-turbo-1106`; this is the latest version of the model, with a maximum output of 4K tokens (but we limited to 512 as longer explanations would likely not be effective for code tour generation) for a context size of 16K tokens and a knowledge cut-off date of September 2021. We set the seed to 2023 for reproducible output. We set the temperature to 0.2, as recommended by Nam et al. [13], to minimise variations in the explanation quality. We provide an example of a code snippet with an explanation within the prompt from Taylor et al. [17], i.e., specification by demonstration [15]. This should ensure the explanations are consistent with the style in the original paper. Without this example, the model tended to describe the methods line-by-line, which is less useful [11]. Since ChatGPT does not provide a system prompt, we use `Llama-2 Chat` [19] one, along with information about the task and the failing test. The order of the methods' appearance in the stack trace gives the order of the explanations in the code tour.

3.2 Data analysis

To assess the quality (**RQ.2**) of the explanation, one evaluator manually analysed 20 (4 randomly sampled from each project) code tours generated by the LLM to identify common problems. From Tintarev and Mashtoff [18], we derived the following checklist:
Transparency. (i) *Unexplained terms*: Are there any unexplained business terms? For example, *hyperplane*, a mathematical term unfamiliar to a junior developer, must be clarified, while the *recursion* is unlikely to require explanation. Moreover, are there any references

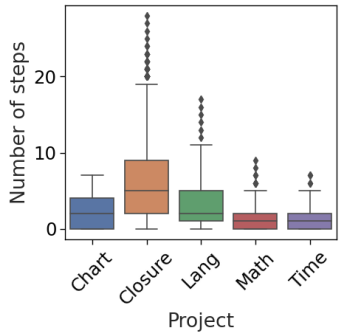

Figure 1: Distribution of the number of steps per code tour.

Table 2: Manual analysis results

Project	Unrelated explanation	Ack't of limitations	Redundant explanation	Low-level explanation	Tot.
Chart	0 (0%)	2 (10%)	17 (81%)	19 (90%)	**21**
Closure	2 (6%)	0 (0%)	24 (69%)	34 (97%)	**35**
Lang	0 (0%)	3 (19%)	12 (75%)	9 (56%)	**16**
Math	2 (6%)	0 (0%)	3 (25%)	11 (92%)	**12**
Time	3 (21%)	0 (0%)	11 (79%)	12 (86%)	**14**
Tot.	**7 (7%)**	**6 (6%)**	**67 (68%)**	**85 (87%)**	**98**

to other code parts (e.g., a class) without prior explanation? (ii) *Lack of context*: Is there a lack of context about the tour? (iii) *Lack of link between steps*: Is there a lack of linking between the steps?
Scrutability. (i)*Unrelated explanation*: Are there explanation segments unrelated to lines of code? (ii) *Inaccurate explanation*: Are there any inaccuracies or false explanations? (iii) *Acknowledgement of limitations*: Are there any passages where the model states that it cannot provide a complete or accurate explanation?
Efficiency. (i) *Redundant explanation*: Are there any parts of the explanation that can be understood by reading the signature or the documentation? For example, restating the signature. (ii) *Low level explanation*: Are there any line-by-line explanation?

In this preliminary evaluation, we focus on four elements related to known common limitations of LLMs: *unrelated explanation*, *acknowledgement of limitations*, *redundant explanation*, and *low-level explanation*. Other elements will be evaluated in future work.

3.3 Results

RQ.1 Generation. As illustrated in Table 1, we generated 913 stack traces (53%) out of 1,736 pairs of failing tests and faulty methods from 357 projects' versions from 5 different projects. Figure 1 reports the distribution of the number of steps in the different tours. The number of steps in tours ranged from 1 to 29 steps. Closure seems to be an outlier with a median of 9 steps, while the medians for the other step lengths are between 2 and 5. 69 tours were generated, totalling 440 steps. In summary, we can see that it is possible to generate code tours for debugging by relying on a stack trace, denoting how an unexpected exception propagates in the code.
RQ.2 Quality. To answer RQ2, one evaluator analysed a sample of 20 tours (4 in each project), totalling 98 steps. The table 2 provides a preliminary quantitative response for the four items on the checklist for which we establish a list of keywords. We are confident that the figures are consistent for these, as we can use the keywords to back up our judgement. The other ones would be meaningless, so we only discuss them in the discussion section. We found 7 (7%) instances where the explanation is about the example given in the prompt. Additionally, we found 5 (5%) instances where the model acknowledges the need for more context or the unavailability of all information in the code snippet. In 67 (68%) instances, the model restates information already given in the documentation, such as the parameters, return value, or exception thrown. Furthermore, in 85 (87%) instances, the explanation provides a low-level, line-by-line code breakdown. In summary, we can see that out of the 98 steps analysed, redundant and low-level explanations are the prevalent issues GPT faces when generating code tours for debugging.

3.4 Discussion and future work

Transparency. After manual analysis of the sampled tours, we saw that the model does not explain business terms or terms related to the code source. This could be expected because we have not specifically asked the model to explain the terms in the prompt (either directly or indirectly through the example). An example where the terms are explained could be included in the prompt to have a more consistent behaviour. Considering the variety of projects, it might be difficult to determine which terms should be explained. Interview developers unfamiliar with the codebase and too see which terms they would seek clarification on, depending on the project and their background is part of our future work to devise general guidelines for designing the prompt.

Regarding the *lack of links between steps*, despite the use of chat history, connections between previous steps in a tour are rare, if nonexistent. This is unfortunate, especially when it comes to generic or utility methods, where explaining why the method is called is more interesting than simply stating what it does without any context. One solution we envision for our future work is to explicitly ask for a connection in the system or an explanation prompt. However, more than this may be required, as making a connection requires having an explanation of the entire tour. Therefore, another generation phase could be added, where the model is asked to create a link between explanations.

Finally, the *lack of context* is a major issue. After investigation, we discovered that, in our case, it is due to the extraction process via CodeQL: the query we used could not extract the methods linked to the test for two projects, Closure and Chart. As a result, all the tours for these projects start somewhere in the stack trace. This is likely due to the folder structure being different from the other three projects. This stresses that the tests are crucial entry points for a debugging tour to understand the context of the tour.
Scrutability. Regarding *unrelated explanations*, LLMs are known to confabulate [7], yet we did not find obvious instances of confabulation in the examined tours. However, we found multiple instances where the model talked about a "*second code snippet*" or some object starting with the name "*Paint*", which was not related to the targeted code snippet. Using "Paint" and "second code snippet" for keyword-based detection, we found a total of 30 instances over 440 steps (7%), which is on par with the prevalence in the manual inspection. We also found six (6%) instances in the sampled steps where the explanation is duplicated from the previous step without being the same method called recursively.

Assessing the presence of *inaccurate explanation* without a good knowledge of the codebase is complicated. Despite inaccurate explanations examples, further validation is required and part of our future work, for instance, using interviews with senior developers.

Efficiency. The model often provides *redundant explanations* from the documentation or the methods' signatures. Usually, it is when it explains things already described in the Javadoc or are obvious, like the signature, the return value, or the exception thrown. We have defined a preliminary list of keywords that can be used to highlight cases where it restates the signature or information that should be available, at the very least, in a good Javadoc. We found 311 (71%) instances with the keywords *param*, *return*, *takes in*, and *thrown*. Note that the *param* keyword sometimes gives false positives because there can be variables or methods named as `PARAM_LIST` or `guessParametersErrors`. Generally, the method has no parameters when a model does not reformulate the parameters.

Regarding *low-level explanation*, our initial attempts with the model often result in line-by-line enumerations. However, according to the related work, students prefer summarized explanations [11]. Thus, we used an example from Taylor et al.'s paper [17] to guide the model. Unfortunately, this has not eliminated the issue of low-level explanation. Interestingly, the model tends to add a summary at the end of the explanation, indicating that the model is doing a chain of thought. Ideally, we only need this summarised bit. For that, we propose to prompt the model twice. First, ask it to explain the method using the chain of thought. Then, ask it to provide a summarized explanation based on its chain of thought. Experiments with different prompting strategies are part of our future work.

Threats to validity. The study's validity presents some limitations: the use of well-known projects possibly included in the Large Language Models' training data, the subjective evaluation by a single person unfamiliar with the codebase, the focus on small Java projects with English explanation generated by GPT 3.5 limiting broader applicability, and the lack of real developer involvement in the study. Future work should address these issues by incorporating various code bases the model has never seen, multiple evaluators, a wider range of languages and project sizes, real developer participation for a complete evaluation and other LLMs.

4 CONCLUSION

Overall, our preliminary evaluation shows that while generating code tours using LLMs and stack traces for debugging and onboarding new developers is feasible, several limitations must be addressed. The LLM sometimes acknowledges information gaps, but explanations can lack focus, repeat information from signatures and documentation and offer low-level code explanations.

Future efforts will explore additional strategies to generate and correct the code tour steps and evaluate subjective aspects like transparency with junior developers and scrutability with senior developers. To mitigate low-level explanations, we will refine our prompting strategy and use a double-prompt chain-of-thought and a final prompt to link the steps coherently.

Although `GPT-4` [14] is better than `gpt-3.5` models, some companies may prefer open-weight options like `Mixtral-8x7B` for confidentiality reasons. It has demonstrated superior performance compared to the `gpt-3.5` models and even comes close to matching `GPT-4` in some benchmarks [20], all with lower resource demands. We plan to further study its potential in our future work.

ACKNOWLEDGMENTS

This research was partially supported by the ARIAC project (No. 2010235), funded by the Service Public de Wallonie (SPW Recherche).

REFERENCES

[1] Anthropic. 2023. Introducing 100K Context Windows — anthropic.com. https://www.anthropic.com/index/100k-context-windows.
[2] Martin Balfroid, Benoît Vanderose, and Xavier Devroey. 2024. Replication package. https://doi.org/10.5281/zenodo.10527351
[3] Talya N Bauer and Berrin Erdogan. 2011. Organizational socialization: The effective onboarding of new employees. (2011).
[4] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the Opportunities and Risks of Foundation Models. *arXiv e-prints* (2021), arXiv–2108.
[5] Jonathan Carter. 2020. Codetour - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=vsls-contrib.codetour
[6] Barthélémy Dagenais, Harold Ossher, Rachel KE Bellamy, Martin P Robillard, and Jacqueline P De Vries. 2010. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 275–284.
[7] Benj Edwards. 2023. *Why CHATGPT and Bing Chat are so good at making things up.* https://arstechnica.com/information-technology/2023/04/why-ai-chatbots-are-the-ultimate-bs-machines-and-how-people-hope-to-fix-them/
[8] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. 2021. A case study of onboarding in software teams: Tasks and strategies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 613–623.
[9] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.
[10] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. *arXiv e-prints* (2023), arXiv–2307.
[11] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 931–937.
[12] Gerardo Matturro, Karina Médici Barrella, and Patricia Benitez. 2017. Difficulties of Newcomers Joining Software Projects Already in Execution. *2017 International Conference on Computational Science and Computational Intelligence (CSCI)* (2017), 993–998.
[13] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2023. In-IDE Generation-based Information Support with a Large Language Model. *arXiv e-prints* (2023), arXiv–2307.
[14] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[15] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.
[16] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*.
[17] Grace Taylor and Steven Clarke. 2022. A Tour Through Code: Helping Developers Become Familiar with Unfamiliar Code.. In *PPIG*. 114–126.
[18] Nava Tintarev and Judith Masthoff. 2015. Explaining recommendations: Design and evaluation. In *Recommender systems handbook*. Springer, 353–382.
[19] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[20] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, et al. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv e-prints* (2023), arXiv–2306.
[21] Yuxiang Zhu and Minxue Pan. 2019. Automatic Code Summarization: A Systematic Literature Review. *ArXiv* abs/1909.04352 (2019).