THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Artificial Intelligence-Based System for Boosting Automated Documentation Generation from Code

Andrew Nedilko

Glossary of Terms

Term	Definition
Machine learning (ML)	A branch of artificial intelligence that enables systems to learn and improve from data or experience without being explicitly programmed.
Large language model (LLM)	A type of ML model designed to understand, generate, and process human language by leveraging vast amounts of text data.
Agent	An autonomous system that perceives its environment and takes actions to achieve specific goals
LLM-based agents	An autonomous system that utilizes an LLM to understand and generate human language, enabling it to perform tasks and make decisions based on textual data.
Agentic workflow	A process in which autonomous agents execute tasks and make decisions independently to achieve specific objectives.
Ground truth	Accurate, real-world data or facts (sometimes referred to as correct labels in a labeled dataset) used to train and evaluate the performance of ML models.

Acronyms

Acronym	Definition
LLM	Large language model
NLP	Natural language processing
NL	Natural language
PL	Programming language

Scope of Work (SOW)

Leveraging Artificial Intelligence to Boost the Automated Documentation Generation from Code. Developing a **System Using Large Language Model-Based Agents** to **Improve the Automatic Creation of Documentation from Code in Python Software Engineering Projects.**

Problem Statement

(A) Deliverable	(B) Format	(C) WC
Issue	Manually generated documentation gets costly, time-consuming, and outdated with ongoing code modifications (Khan et al., 2022).	*12*
Reference	Khan, J. Y., & Uddin, G. (2022). Automatic code documentation generation using GPT-3. https://doi.org/10.48550/arXiv.2209.02235	NA
”so what”	Forcing developers to spend 30% more time understanding code (Zelkowitz et al., 1979)	*9*
Reference	Zelkowitz, M. V., Shaw, A. C., & Gannon, J. D. (1979). Principles of software engineering and design.	NA
Problem statement	Manually generated documentation gets costly, time-consuming, and outdated with ongoing code modifications, forcing developers to spend 30% more time understanding code.	*21*
Industry	Software Engineering	NA
PS elaboration 1	The complexity and dynamic nature of modern software systems, characterized by frequent updates, refactoring, and integration of new technologies, makes comprehension and working with existing codebases more challenging for developers.	*30*
PS elaboration 2	Manual documentation generation is particularly challenging in large, distributed teams, where multiple stakeholders contribute to codebases, making it difficult to maintain a unified understanding of the code's functionality and evolution.	*30*

Thesis Statement

(A) Deliverable		WC
Thesis Statement	Large language model-based agents for automated documentation generation will reduce the time developers spend understanding and maintaining code by generating up-to-date, accurate, and informative documentation from source code.	*28*
Research Product	LLM-based agent	*NA*
Format	Python script(s)	*NA*
Deliverable Usage	Python software developers will use this product to automatically generate up-to-date, accurate, and informative documentation from source code, reducing their time spent on understanding and maintaining code.	*27*
Tie back to PS	By automating documentation generation, LLM-based agents address the problem of manual documentation being costly, time-consuming, and out-of-date, thereby reducing the extra time developers spend understanding code.	26
New Contributions	This research introduces a novel approach by leveraging LLM-based agents to automate documentation generation from Python source code, surpassing existing methods in accuracy, speed, and adaptability to ongoing code modifications.	*30*
Scope	Developing an AI system using LLM-based agents to improve automated documentation generation from Python source code in software engineering.	*19*
Main methodology	Machine learning	*NA*
Inputs	Functions: text, files with code: text, code repositories: folder of text files	*NA*
Outputs	Docstrings: text, Readme files: text, documentation files: text	*NA*

Research Questions

(A) Deliverable	(B) Format	(C) WC
Research Question 1	Will fine-tuning Large Language Models used by agents result in higher documentation generation quality as measured by the BLEU, ROUGE, or METEOR score?	*23*
Research Question 2	Will changing Large Language Model parameters, such as temperature and top-p, ensure greater code coverage in auto-generated documentation?	*18*
Research Question 3	Which agentic workflow, reflection or multi-agent collaboration, leads to greater semantic similarity between the auto-generated and ground truth documentation?	*19*

Research Hypotheses

(A) Deliverable	(B) Format	(C) WC
Hypothesis 1	Fine-tuning Large Language Model on domain-specific data will significantly increase the BLEU, ROUGE, or METEOR score compared to using an LLM without fine-tuning.	*23*
Independent Variables	Fine-tuning of LLM	*NA*
Dependent Variable	BLEU, ROUGE, or METEOR score	NA
Testable	Set up experiments with the two LLMs and compare the METEOR scores.	*12*
Hypothesis 2	Adjusting Large Language Model parameters, such as temperature and top-p, will significantly improve code coverage in auto-generated documentation.	*18*
Independent Variables	Temperature, top-p	*NA*
Dependent Variable	Code coverage	NA
Testable	Set up experiments when LLM parameters are adjusted and not adjusted and compare the code coverage.	*16*
Hypothesis 3	Multi-agent collaboration will lead to higher semantic similarity between auto-generated and ground truth documentation compared to the reflection agentic workflow	*20*
Independent Variables	Reflection, multi-agent collaboration agentic workflows	*NA*
Dependent Variable	Semantic similarity	NA
Testable	Set up experiments with two agentic workflows and compare the semantic similarities.	*12*

Annotated Bibliography (1 of 5)

(A) Deliverable	(B) Format	(C) WC
Reference	Qian, C., Cong, X., Liu, W., Yang, C., Chen, W., Su, Y., Dang, Y., Li, J., Xu, J., Li, D., Liu, Z., & Sun, M. (2024). ChatDev: Communicative agents for software development. https://doi.org/10.48550/arXiv.2307.07924	*NA*
Summary	• The proposed system utilizes large language models (LLMs) to automate the entire software development process through a chat-based framework. • The process is divided into designing, coding, testing, and documenting phases, with the agents collaborating via multi-turn dialogues. • The chat chain mechanism breaks down tasks into atomic subtasks, enhancing collaboration and precision.	*19* *18* *14*
Methodology	Machine learning in the form of LLMs for collaborative, multi-agent dialogues.	*NA*
Evaluation	• The proposed system is useful because it effectively reduces software development time and costs, completing projects in under seven minutes for less than one dollar. • Its goal is to streamline software development through natural language communication, minimizing the need for any specialized models. • The proposed LLM-powered software development methods achieved an improved completeness of 0.5600 and executability of 0.8800.	*24* *18* *16*
Relevance	• The article is relevant because it highlights the integration of LLMs into different phases of software development including documentation. • It is very helpful because it provides insights into using LLMs for automated, collaborative processes, relevant for automated documentation generation. • It can shape my research because the proposed mechanism and thought instruction methods offer potential techniques for improving AI-driven documentation systems.	19 20 21

Annotated Bibliography (2 of 5)

(A) Deliverable	(B) Format	(C) WC
Reference	Khan, J. Y., & Uddin, G. (2022). Automatic Code Documentation Generation Using GPT-3. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Article No. 174.* pp. 1–6. https://dl.acm.org/doi/10.1145/3551349.3559548	*NA*
Summary	• The study employs Codex, a GPT-3 based model pre-trained on both natural and programming languages, for automatic code documentation creation. • Codex outperforms existing techniques, especially in one-shot learning, demonstrating significant improvements across six programming languages. • The research highlights Codex's potential and suggests future studies for in-depth evaluations and broader applications.	*20* *15* *15*
Methodology	Machine learning in the form of a GPT-3 based transformer model.	*NA*
Evaluation	• Goal: the study aims to evaluate Codex's effectiveness in generating accurate and informative code documentation. • It is useful as Codex demonstrates state-of-the-art performance in automated documentation generation. • Codex achieved an overall BLEU score of 20.63, indicating high performance compared to previous models.	*15* *12* *15*
Relevance	• The paper validates the use of large language models like GPT-3 for automated documentation. • It will shape my research by demonstrating the importance of leveraging advanced AI models to improve documentation accuracy and efficiency, guiding the development of my proposed system. • The study provides a performance benchmark (BLEU score) for comparing and evaluating the effectiveness of the proposed AI-based documentation system.	14 27 20

Annotated Bibliography (3 of 5)

(A) Deliverable	(B) Format	(C) WC
Reference	Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*. pp. 1536—1547. https://aclanthology.org/2020.findings-emnlp.139.pdf, https://doi.org/10.48550/arXiv.2002.08155	*NA*
Summary	• CodeBERT is a bimodal model pre-trained on both natural languages (NL) and programming languages (PL), supporting tasks like code search and documentation generation. • It uses a hybrid objective function, combining masked language modeling and replaced token detection to identify plausible alternatives sampled from generators. • CodeBERT achieves state-of-the-art performance in NL-PL tasks, evaluated through fine-tuning and zero-shot learning.	*23* *21* *15*
Methodology	Machine learning using a transformer-based model.	*NA*
Evaluation	• Goal: The study aims to create a general-purpose model for NL and PL tasks. • The study is useful as CodeBERT demonstrates very high effectiveness in code-to-text generation tasks. • CodeBERT achieved state-of-the-art performance with an average BLEU-4 score of 17.83 across six programming languages, an improvement over previous models.	*14* *14* *20*
Relevance	• The paper is relevant as it validates the use of large language models for documentation generation. • The study offers performance benchmarks for comparing the effectiveness of the AI-based documentation system being developed. • The study can shape my research because it provides insights into different training objectives and evaluates the type of knowledge learned by the model.	16 16 24

Annotated Bibliography (4 of 5)

(A) Deliverable	(B) Format	(C) WC
Reference	Hu, X., Chen, Q., Wang, H., Xia, X., Lo, D., & Zimmermann, T. (2023). Correlating automated and human evaluation of code documentation generation quality. *ACM Transactions on Software Engineering and Methodology, Volume 31, Issue 4, Article No. 63*, pp 1–28. https://doi.org/10.1145/3502853	*NA*
Summary	• The study investigates the correlation between automated metrics, such as BLEU, METEOR, ROUGE-L, CIDEr, SPICE, and human evaluation in code documentation generation tasks. • The results show a weak correlation between automated metrics and human judgment, indicating the need for improved evaluation methods that align more closely with human assessments.	*23* *26*
Methodology	Empirical Study with Deep Learning-based Techniques.	*NA*
Evaluation	• The article aims to bridge the gap between automated and human evaluations to improve the reliability of code documentation assessments. • It is useful because it captures the limitations of current automated metrics in describing human perspectives on code documentation quality. • The best comment generation model achieved a METEOR score of 16.04%, BLEU score of 21.06%, and ROUGE-L score of 36.2%.	*20* *20* *18*
Relevance	• The article is helpful as it provides insights into the effectiveness of automated metrics, relevant for evaluating AI-based documentation generation systems. • It is relevant since the study's findings can be used to refine the evaluation process in AI-driven documentation generation, ensuring that the generated documentation meets human quality standards.	21 28

Annotated Bibliography (5 of 5)

(A) Deliverable	(B) Format	(C) WC
Reference	Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology. Article No.2*. pp. 1-22. https://doi.org/10.1145/3586183.3606763	*NA*
Summary	• Generative agents simulate human behavior by combining large language models with mechanisms for storing, synthesizing, and retrieving memories. • These agents display believable individual and group behaviors in a sandbox environment, showcasing memory retrieval, reflection, and planning capabilities. • Evaluations demonstrate the agents' ability to spread information, form relationships, and coordinate activities, highlighting the emergent social dynamics.	*18* *19* *18*
Methodology	LLM-based agents.	*NA*
Evaluation	• The article is useful as it provides a framework for creating believable AI agents capable of dynamic interaction. • Its goal is to simulate human behavior accurately for the use in interactive applications. • The agent community formed new relationships during the simulation, with the network density increasing from 0.167 to 0.74.	*18* *14* *18*
Relevance	• The article is relevant because the techniques for memory retrieval and dynamic behavior can inform methods to keep documentation up-to-date and relevant, reducing developer effort in understanding code. • It is helpful because the detailed architecture and behavior analysis of generative agents offer valuable insights for developing AI systems for automated documentation. • It will shape my research by highlighting the importance of memory management, planning, and reflection in AI behavior, which can be applied to enhance code documentation systems.	28 23 27

Data Sources List

ID	Name	Source	Brief Description	Format	Time Span	Num. of records	Access (Y/N)
1	CodeSearchNet Python	HuggingFace Hub: https://huggingface.co/datasets/code-search-net/code_search_net	A dataset of ~0.5 million code-documentation pairs from popular open-source GitHub repositories in the Python programming language.	HuggingFace Dataset format; convertible to pandas dataframe or csv	2019	457,461	Y

Data Source Example

ID	Name	Source	Brief Description	Format	Time Span	Num. of records	Access (Y/N)
1	CodeSearchNet Python	HuggingFace Hub: https://huggingface.co/datasets/code-search-net/code_search_net	A dataset of ~0.5 million code-documentation pairs from popular open-source GitHub repositories in the Python programming language.	HuggingFace Dataset format; convertible to pandas dataframe or csv	2019	457,461	Y

Item	Description	WC
Purpose	• While containing annotated code-documentation pairs, this dataset will be used with the purpose of enabling the training and evaluation of systems for automatic code documentation generation.	26
Data Treatment	• Filter examples in which the number of tokens in the documents is less than 5 or greater than 512. • Filter examples documented in a language other than English. • Remove or clean documentation that contains special tokens (e.g. “<img ...>”, “https:”, and so on)	19 9 15

repository_name	func_path_in_repositor	func_name	whole_func_string	language	func_code_string	func_code_tokens	func_documentation_string	func_documentation_tokens	split_name
proycon/pynlpl	pynlpl/formats/folia.py	AbstractElement.addidsuffix	def addidsuffix(self, idsuffix, recursive = True): """Appends a suffix to this element's ID, and optionally to all child IDs as well. There is sually no need to call this directly, invoked implicitly by :meth:`copy`""" if self.id: self.id += idsuffix	python	def addidsuffix(self, idsuffix, recursive = True): """Appends a suffix to this element's ID, and optionally to all child IDs as well. There is sually no need to call this directly, invoked implicitly by :meth:`copy`""" if self.id: self.id += idsuffix	['def' 'addidsuffix' '(' 'self' ',' 'idsuffix' ',' 'recursive' '=' 'True' ')' ':' 'if' 'self' '.' 'id' ':' 'self' '.' 'id' '+=' 'idsuffix' 'if' 'recursive' ':' 'for' 'e' 'in' 'self' ':' 'try' ':' 'e' '.' 'addidsuffix']	Appends a suffix to this element's ID, and optionally to all child IDs as well. There is sually no need to call this directly, invoked implicitly by :meth:`copy`	['Appends' 'a' 'suffix' 'to' 'this' 'element' 's' 'ID' 'and' 'optionally' 'to' 'all' 'child' 'IDs' 'as' 'well' '.' 'There' 'is' 'sually' 'no' 'need' 'to' 'call' 'this' 'directly' 'invoked'	train

proycon/pynlpl	pynlpl/formats/folia.py	AbstractElement.setparents	def setparents(self):
			"""Correct all parent relations for elements within the scop. There is sually no need to call this directly, invoked implicitly by :meth:`copy`"""
			for c in self:
			if isinstance(c, AbstractElement):
				python	def setparents(self):
					"""Correct all parent relations for elements within the scop. There is sually no need to call this directly, invoked implicitly by :meth:`copy`"""
					for c in self:
						['def' 'setparents' '(' 'self' ')' ':' 'for' 'c' 'in' 'self' ':' 'if' 'isinstance' '(' 'c' ',' 'AbstractElement' ')' ':' 'c' '.' 'parent' '=' 'self' 'c' '.' 'setparents' '(' ')']	Correct all parent relations for elements within the scop. There is sually no need to call this directly, invoked implicitly by :meth:`copy`	['Correct' 'all' 'parent' 'relations' 'for' 'elements' 'within' 'the' 'scop' '.' 'There' 'is' 'sually' 'no' 'need' 'to' 'call' 'this' 'directly' 'invoked' 'implicitly' 'by' ':' 'meth' ':'	train

proycon/pynlpl	pynlpl/formats/folia.py	AbstractElement.setdoc	def setdoc(self,newdoc):
			"""Set a different document. Usually no need to call this directly, invoked implicitly by :meth:`copy`"""
			self.doc = newdoc
			if self.doc and self.id:
				python	def setdoc(self,newdoc):
					"""Set a different document. Usually no need to call this directly, invoked implicitly by :meth:`copy`"""
					self.doc = newdoc
					if self.doc and self.id:
						['def' 'setdoc' '(' 'self' ',' 'newdoc' ')' ':' 'self' '.' 'doc' '=' 'newdoc' 'if' 'self' '.' 'doc' 'and' 'self' '.' 'id' ':' 'self' '.' 'doc' '.' 'index' '[' 'self' '.' 'id' ']' '=' 'self' 'for' 'c' 'in' 'self' ':' 'if' 'isinstance' '(' 'c' ',' 'AbstractElement' ')' ':' 'c' '.'	Set a different document. Usually no need to call this directly, invoked implicitly by :meth:`copy`	['Set' 'a' 'different' 'document' '.' 'Usually' 'no' 'need' 'to' 'call' 'this' 'directly' 'invoked' 'implicitly' 'by' ':' 'meth' ':' 'copy']	train

Appendix

APA Guidelines(*)

- Source citation consists of:
  - A brief **parenthetical citation** in the text
  - A **full reference** at the end of the paper

APA In-text Citations

- An APA in-text citation includes the author's last name and the publication year.
- If you're quoting or paraphrasing a specific passage, you also add a page number.

1 author	(Taylor, 2018, p. 23)
2 authors	(Taylor & Kotler, 2018, p. 23)
3–5 authors	First citation: (Taylor, Kotler, Johnson, & Parker, 2018, p. 23) Subsequent citations: (Taylor et al., 2018, p. 23)
6+ authors	(Taylor et al., 2018, p. 23)

APA Reference List

Smith, T. (2019). *Citing sources and referencing: A quick guide*. (J. M. Taylor, Ed.) (2nd ed.). Amsterdam, The Netherlands: Scribbr.

In-text citation

According to new research (Smith, 2019, pp. 11–12) …
As mentioned before (Smith, 2019, pp. 11–12) …
(See Smith, 2019)

* https://www.scribbr.com/citing-sources/apa-vs-mla/

Materials for the Meeting on August 4, 2024

Assignment

1) Develop flow diagram that shows a simple, high level approach of how one would envision going through the research. Please include sub bullets for each step that shows substeps such: as data filtering/preprocessing, model optimization/hypertuning, expected outputs, validation of the outputs, etc.

Note: Start thinking of time intervals for each one of these main activities and how long it will take to get each one completed. (Written estimates or ECDs are not needed for our next meeting)

2) Develop summary for 5 to 10 research articles, no more than 1-2 paragraphs of the article's content and how you think it applies to you or your research topic. (I.e. the takeaway)

3) List any questions/concerns you might have (if any) about your data sets, the methodology, literature or how to connect step A to step B as listed in your flow diagram, etc.

Research Flow Diagram (1)

1. **Data Collection and Preprocessing**
- Identify and collect as many code documentation datasets as possible, similar to CodeXGlue and CodeSearchNet. If needed, collect raw codebases from various projects.
- If raw codebases, extract existing documentation to serve as ground truth.
- Compare the value of each dataset for the research at hand and make final selection.
- Filter and clean the selected data to remove any irrelevant or noisy information. This may include the following actions:
  - ✓ Filter examples where the number of tokens is less than 5 or greater than 512.
  - ✓ Filter examples documented in a language other than English.
  - ✓ Remove or clean documentation that contains special tokens (e.g. “<img ...>”, “https:”, and so on)
- Split data into training, validation, and / or test sets.
- Expected Outputs:
  - ✓ Cleaned and well-organized datasets as ground truth for agent / model evaluation.

Research Flow Diagram (2)

2. **Model Selection and Fine-Tuning**
- Select pre-trained Large Language Models (LLMs) suitable for the task (e.g., GPT-4, Claude Sonnet).
- Establish the required APIs to access the models for inference and fine-tuning.
- If a small language model – procure the resources required to fine-tune the model (cloud GPU instance, Google Colab, etc.).
- Fine-tune models on the training dataset with an emphasis on code-to-text generation.
- Experiment with different architectures, pre-training objectives, and fine-tuning techniques such as SFT, DPO, ORPO etc.
- Expected Outputs:
  - ✓ Fine-tuned LLMs optimized for documentation generation.

3. **Hyperparameter Optimization**
- Adjust hyperparameters (e.g., learning rate, batch size) to improve the fine-tuned model performance.
- Experiment with different LLM parameters, such as temperature and top-p, to achieve better code coverage and other metrics.
- If applicable at all, conduct grid or random search for hyperparameter tuning (not for pre-trained LLMs due to their size).
- Expected Outputs:
  - ✓ Optimized hyperparameters for maximum performance.

Research Flow Diagram (3)

4. **Agentic Workflow Implementation**
- Develop agents based on reflection, multi-agent collaboration, and potentially other agentic workflows.
- Integrate fine-tuned models into these agents.
- Define metrics for evaluating semantic similarity and other relevant aspects.
- Expected Outputs:
  - ✓ Functional agents ready for documentation generation.

5. **Evaluation and Validation**
- Generate documentation using the developed agents.
- Evaluate the quality using relevant metrics, e.g. the BLEU, ROUGE, and METEOR scores.
- Compare the metrics for auto-generated documentation vs. ground truth.
- Expected Outputs:
  - ✓ Comprehensive evaluation results.
  - ✓ Insights into the effectiveness of the developed agents.

Research Flow Diagram (4)

6. **Analysis and Interpretation**
- Analyze evaluation results to draw conclusions.
- Interpret the implications of the findings for the AI and software engineering communities.
- Expected Outputs:
  - ✓ Detailed analysis and interpretation of the results.

7. **Documentation and Reporting**
- Document the entire research process.
- Prepare the Praxis report, including all findings, methodologies, and implications.
- Review and revise the Praxis based on feedback from the advisor(s).
- Expected Outputs:
  - ✓ Finalized Praxis document.

Papers (1)

1. Minh Huynh Nguyen∗, Thang Chau Phan∗, Phong X. Nguyen∗, Nghi D. Q. Bui. **Dynamic Collaborative Agents for Software Development based on Agile Methodology**

The article describes a multi-agent system for software development. It assigns Agile roles (Product Manager, Developer, Tester) to agents working in sprints. A Code Dependency Graph (CDG) is to aid code comprehension and generation. The multi-agent approach and use of Dynamic CDG are directly applicable to developing AI-based documentation systems.

Takeaways: a) Multi-agent collaboration: effective role-specific agents can enhance automated documentation, b) Dynamic code graphs are essential for maintaining accurate and current documentation.

2. Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, Ju¨rgen Schmidhuber. **MetaGPT: meta programming for a multi-agent collaborative framework**.

The article talks about a framework that enhances multi-agent collaboration by integrating Standardized Operating Procedures (SOPs) into LLM-based systems. It assigns specific roles (e.g., Product Manager, Engineer) to agents, promoting efficient task decomposition and reducing errors.

Relevance my research: this is highly relevant for improving automated documentation generation. Implementing SOPs can enhance the quality and consistency of generated documentation. Additionally, the role-based agent collaboration model can be adapted to develop specialized agents for different documentation tasks, ensuring comprehensive and accurate outputs.

Papers (2)

3. Menaka Pushpa Arthur. **Automatic Source Code Documentation using Code Summarization Technique of NLP**.

The paper proposes a novel system for automating source code documentation using NLP techniques. The system employs a Software Word Usage Model (SWUM) built with Context-Free Grammars and NLP preprocessing to generate concise and clear documentation.

The proposed system's approach is highly relevant to the research on AI-based documentation generation. The techniques described can be adapted for Python, enhancing the accuracy and informativeness of the automated documentation. Takeaways: a) NLP Techniques are effective for automatic documentation generation, b) SWUM and CFG are useful for building a robust model to understand code syntax and semantics, c) Comparison with expert documentation using special metrics proves the system's effectiveness.

4. Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, Chi Wang. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation**.

AutoGen is an open-source framework that uses multiple interacting agents to create applications. The framework supports various modes of operation, allowing agents to converse and collaborate on tasks. The framework employs "conversation programming," where agent interactions are controlled through both natural language and programming languages, enhancing the modularity and scalability of multi-agent systems.

Relevance: the principles and capabilities of AutoGen align closely with my research topic. The framework's support for multi-agent collaboration and conversation-based workflows can significantly aid in developing an LLM-based system for automated documentation. The customizable agents and conversation programming features of AutoGen can be utilized to create a robust and adaptable documentation generation system, ensuring high-quality.

Papers (3)

5. Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu. DehengYe. **More Agents Is All You Need**.

The article demonstrates that the performance of LLMs improves with the number of instantiated agents. This method involves feeding a query to multiple LLM agents to generate various outputs and then applying majority voting to determine the final answer. The results suggest that smaller LLMs, when scaled up in ensemble size, can outperform larger counterparts.

Relevance to my research: the sampling-and-voting method described in the article can be adapted to improve the performance of LLM-based agents in generating documentation from code. By employing multiple agents and aggregating their outputs, the proposed research can achieve more accurate and comprehensive documentation.

Potential Questions/Concerns

Data Set Issues

- Are the selected datasets / codebases representative of a wide range of software development projects to ensure generalizability?
- How can we ensure the ground truth documentation is of high quality and up-to-date?

Methodology Concerns

- What criteria should be used to determine the best-performing model, beyond BLEU, ROUGE, and METEOR scores?
- How to balance between fine-tuning and generalization in LLMs to avoid overfitting?
- What iterative processes should be in place to refine the agents based on evaluation feedback?

Literature Connection

- How can the insights from the surveyed articles be integrated into the experimental design and evaluation framework?
- Are there any emerging trends or recent advancements in LLMs that should be considered for the research?

Materials for the Meeting on August 18, 2024

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Documentation Generation Datasets

#	Name	Purpose	Language	Description	Link	Comment
1	CodeXGlue	Code summarization	Python and others	14 datasets for 10 diversified programming language tasks covering **code-code** (clone detection, defect detection, cloze test, code completion, code refinement, and code-to-code translation), **text-code** (natural language code search, text-to-code generation), **code-text** (code summarization) and **text-text** (documentation translation) scenarios	https://github.com/microsoft/CodeXGLUE	
2	CodeSearchNet	Code summarization	Python and others	Large dataset of functions with associated documentation written in Go, Java, JavaScript, PHP, Python, and Ruby from open source projects on GitHub	https://github.com/github/CodeSearchNet	https://paperswithcode.com/dataset/codesearchnet
3	CoNaLa	Evaluation of code generation tasks	Python	A benchmark of code and natural language pairs, for the **evaluation of code generation tasks**. The dataset was crawled from Stack Overflow, automatically filtered, then curated by annotators, split into 2,379 training and 500 test examples. The automatically mined dataset is also available with almost 600k examples.	https://huggingface.co/datasets/neulab/conala, https://conala-corpus.github.io/mining.html	
4	PyTorrent		Python	**Uses CodeSearchNet schema!**	https://github.com/fla-sil/PyTorrent?tab=readme-ov-file	https://paperswithcode.com/dataset/pytorrent
5	Notebookcdg	Code documentation generation	Jupyter notebooks	28,625 code–documentation pairs from top 10% voted notebooks on Kaggle	https://paperswithcode.com/dataset/notebookcdg	
6	Docstring Corpus	Automated code documentation	Python	Python functions and docstrings for automated code documentation (code2doc) and automated code generation (doc2code) tasks	https://huggingface.co/datasets/teven/code_docstring_corpus, https://www.pure.ed.ac.uk/ws/portalfiles/portal/44423958/code_docstring_corpus_IJCNLP17_1.pdf	Check docstrings: some are really short - is it OK? https://github.com/EdinburghNLP/code-docstring-corpus
7	Docs-python-v1	Both	Python	Python code - documentation pairs	https://huggingface.co/datasets/ASHu2/docs-python-v1	Is it only docstring?
8	Funcom	Source code summarization	Java	Collection of ~2.1 million Java methods and their associated Javadoc comments	https://huggingface.co/datasets/apcl/funcom-python, https://github.com/mcmillco/funcom?tab=readme-ov-file	https://paperswithcode.com/dataset/funcom
9	Callcon-public	Source code summarization	Java	Paper: Function Call Graph Context Encoding for Neural Source Code Summarization	https://github.com/aakashba/callcon-public	
10	Attn-to-FC	Code Summarization	Java	We use the dataset of 2.1m Java methods and method comments, already cleaned and separated into training/val/test sets	https://github.com/Attn-to-FC/Attn-to-FC	

Documentation Generation Datasets

#	Name	Purpose	Language	Description	Link	Comment
11	ICPC2020_GNN	Code Summarization	Java	2.1 million Java method-comment pairs	https://github.com/acleclair/ICPC2020_GNN	Is it the same as Attn-to-FC?
12	Deepcomm	Ground truth for documentation generation models	Java	Java methods with associated comments	https://github.com/xing-hu/EMSE-DeepCom	https://paperswithcode.com/dataset/deepcom-java
13	Java Scripts	Code summarization	Java	The Java dataset introduced in Hybrid-DeepCom (Deep code comment generation with hybrid lexical and syntactical information), commonly used to evaluate automated code summarization. It is basically a further version of DeepCom-Java.		https://paperswithcode.com/dataset/hybrid-deepcom-java
14	HumanEval	Both documentation and code generation	Python	164 programming problems with a function sig- nature, docstring, body, and several unit tests. They were handwritten to ensure not to be included in the training set of code generation models	https://huggingface.co/datasets/openai_humaneval	
15	The Stack	Multiple	Multiple	Huge - 6TB	https://huggingface.co/datasets/bigcode/the-stack	
16	MBPP	Both documentation and code generation	Python	Mostly Basic Python Problems. 974 rows	https://huggingface.co/datasets/google-research-datasets/mbpp	

Documentation Generation Results

#	Name	Approach / Method	Evaluation Dataset	Metric Type	Metric Value	Limitation / constraints	Recommended future work
1	Khan et al. (2022). Automatic Code Documentation Generation Using GPT-3.	Codex, a GPT-3-based model, was used for automatic code documentation generation in a zero-shot and one-shot learning setup. Codex was pre-trained on large-scale data, including natural and programming languages, enabling it to generate documentation with minimal additional training.	**CodeSearchNet** - a large collection of code and documentation pairs across six programming languages: Python, Java, PHP, GO, JavaScript, and Ruby. Significant improvements over existing models - Codex outperformed the nearest competitor, CoTexT, by 11.21%.	1-shot results: **smoothed BLEU score** for six programming languages, Python BLEU	 20.63 22.28	* Using only 1,000 samples per language for evaluation - might not cover all variations in code and documentation. * Using only zero-shot and one-shot learning. * Codex was not fine-tuned (no such feature). * Relying on default parameter settings from Codex's official documentation.	* Evaluation on a larger sample sizes and additional programming languages. *Try few-shot learning and fine-tuning to improve documentation generation. * Evaluate different parameter settings in Codex.
2	Feng et al. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages.	CodeBERT is a bimodal pre-trained model that captures the semantic connections between natural language (NL) and programming language (PL) across six programming languages. The model is trained using a hybrid objective function that includes masked language modeling (MLM) and replaced token detection (RTD). CodeBERT leverages both bimodal data (NL-PL pairs) and unimodal data (either NL or PL alone) to learn general-purpose representations that can be fine-tuned for downstream tasks such as code documentation generation.	CodeBERT was trained and evaluated using a dataset derived from GitHub repositories, consisting of 2.1 million bimodal datapoints (function-level NL-PL pairs) and 6.4 million unimodal code datapoints across six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. The evaluation for code documentation generation was conducted using the **CodeSearchNet** corpus.	**Smoothed BLEU-4 score** using the combined MLM and RTD pre-training objectives: Overall Python	 17.83 19.06	* The pre-training objectives of CodeBERT were primarily designed for NL-PL understanding tasks rather than generation tasks. * The model did not include an abstract syntax tree (AST) in its input, which might have limited its performance in code-related tasks. Efforts to incorporate AST did not yield significant improvements, suggesting a potential area for further research. * The evaluation on a programming language not seen during training (C#) showed that CodeBERT generalized well, but slightly underperformed compared to models that utilized AST.	* Investigate the incorporation of AST into the pre-training process. * Develop better generators with bimodal evidence or more sophisticated neural architectures to enhance the RTD objective. * Extend CodeBERT to support more programming languages and explore domain/language adaptation methods to improve generalization. * Explore generation-specific learning objectives to further enhance CodeBERT's capabilities in code documentation generation tasks.
3	Gu et al. (2002). Assemble Foundation Models for Automatic Code Summarization	The paper proposes a transfer learning approach named AdaMo, which assembles existing foundation models like CodeBERT (for code representation) and GPT-2 (for text generation). The approach also introduces Gaussian noise to simulate contextual information and enhance latent representations. The paper includes adaptive schemes such as continuous pretraining and intermediate finetuning to further improve the model's performance.	The evaluation was conducted on two datasets: **BASTS and SIT**. **CodeSearchNet** (CSN) was used for some other aspects. All three datasets include pairs of code snippets and code comments in Java and Python. Each dataset was used to validate different aspects of the proposed model. The system outperformed SOTA models.	**Python BASTS:** Corpus BLEU Sentence BLEU METEOR ROUGE **Python SIT:** Corpus BLEU Sentence BLEU METEOR ROUGE	 5.19% 16.46% 12.51% 27.31% 26.52% 33.9% 21.68% 41.25%	The incoordination of parallel representations when reusing pretrained models affected zero-shot performance. Reduced effectiveness of Gaussian noise in code summarization compared to natural language processing, likely due to the artificial nature of code data. While continuous pretraining and intermediate finetuning improved results, the benefits varied depending on the dataset and task, suggesting that more work is needed to optimize these techniques.	Future work could focus on improving the alignment of latent representations between pretrained encoder and decoder models. Exploring other types of token-level objectives besides Masked Language Modeling (MLM) and further refining the use of Gaussian noise or other noise models. Implementing more advanced refiners or neural models to manipulate latent representations could offer significant performance gains.
4	Elnaggar et al. (2020). CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing	The paper describes the use of an encoder-decoder transformer model, specifically the CodeTrans model, which applies the transformer architecture to various software engineering tasks, including automatic documentation generation. The model is trained using different strategies such as single-task learning, transfer learning, multi-task learning, and multi-task learning with fine-tuning. Larger models generally perform better, especially when combined with transfer learning or multi-task learning with fine-tuning. Multi-task learning is particularly beneficial for small datasets, as it helps avoid overfitting.	The dataset used for evaluating the automatic documentation generation task is the **CodeSearchNet** Corpus Collection. This dataset includes functions / methods from six programming languages: Python, Java, Go, Php, Ruby, and Javascript, with corresponding documentation extracted from GitHub repositories. The model outperformed the existing state-of-the-art model, CodeBERT, in this task.	**Smoothed BLEU-4 score:** Python Java Go PHP Ruby JavaScript	 20.39% 21.87% 19.54% 26.32% 15.26% 18.99%	The performance of multi-task learning is heavily influenced by the size and characteristics of the dataset. For large datasets like the one used for Code Comment Generation, multi-task learning models did not perform as well as single-task learning models. Additionally, larger models tend to overfit on small datasets. The model's effectiveness is also dependent on the preprocessing steps, such as parsing and tokenizing the code, which can be complex and may require the user to be proficient in programming.	Investigate the impact of function and parameter names on model performance, especially when these are disguised or obfuscated. Explore methods to better represent the code structure in the model's input, as this could potentially improve performance. Consider the effect of preprocessing on task performance and explore models that do not require extensive preprocessing, making them more accessible to non-experts. Test the model on unseen programming languages and human language tasks to evaluate its generalizability across different domains.

Documentation Generation Results

#	Name	Approach / Method	Evaluation Dataset	Metric Type	Metric Value	Limitation / constraints	Recommended future work
5	Liu et al. (2021). HAConvGNN: Hierarchical Attention Based Convolutional Graph Neural Network for Code Documentation Generation in Jupyter Notebooks	The paper proposes the HAConvGNN model, which uses a Hierarchical Attention-based Convolutional Graph Neural Network to generate **code documentation for Jupyter notebooks**. The model encodes multiple code cells and their Abstract Syntax Tree (AST) structures, employing hierarchical attention to consider both low-level (token-level) and high-level (cell-level) relationships in the code.	The authors created a new dataset called notebookCDG for the evaluation, which includes approximately 28,625 code-documentation pairs extracted from 2,476 highly-ranked Kaggle notebooks.	ROUGE-1: Precision, Recall, F1 ROUGE-2 Precision , Recall, F1 ROUGE-L Precision, Recall, F1	22.87, 16.92, 16.58 6.72, 4.86, 4.97 24.03, 18.60, 18.54	The hierarchical attention mechanism is critical for improving performance. Treating multiple code cells as a single sequence significantly degraded performance. Although HAConvGNN performed well, human evaluations showed that it still lagged behind ground-truth documentation, especially in terms of informativeness. **The model is specialized for Jupyter notebooks, which may limit its generalizability to other coding environments**.	Integrate the hierarchical attention mechanism with transformer models like T5 to potentially improve performance further. Conduct more extensive human evaluations in real-world applications to better understand the effectiveness of their model in practical settings. Adapt the model for broader contexts, such as applying it to entire code repositories with hierarchical structures beyond Jupyter notebooks.
6	Hu et al. (2023). Correlating automated and human evaluation of code documentation generation quality.	The paper evaluates state-of-the-art deep learning-based techniques for automatic code documentation generation. These techniques leverage large-scale source code corpora to generate documentation such as code comments and commit messages. The study replicates three state-of-the-art approaches for each task (comment generation and commit message generation) and makes a **performance comparison using both automated metrics and human evaluation**.	* **Java code comment dataset** was provided by Hu et al. with 455k <Java method, comment> pairs for training and 5k for testing. * **Commit message dataset** was provided by Liu et al. with 26k commits in the training set, and 3k commits in both the validation and test sets. * **Automatic metrics**: BLEU, METEOR, ROUGE-L, CIDEr, and SPICE. * **Human evaluation**: 24 participants evaluated the generated documentation on a 5-point Likert scale based on six metrics: Naturalness, Expressiveness, Content Adequacy, Conciseness, Usefulness, and Code Understandability. * Results: METEOR showed the strongest correlation with human evaluation (Pearson correlation of 0.7 and Kendall correlation of 0.5) - still lower than correlation between human annotators (Pearson0.8 and Kendall 0.7).	Results for best approach for comment generation: METEOR(%) BLEU(%) ROUGE-L(%) CIDEr SPICE(%)	16.04% 21.06% 36.2% (0.3) 1.85(3.4) 23.1(0.28)	* The study found that automated metrics (such as BLEU, ROUGE-L, CIDEr, and SPICE) do not correlate strongly with human judgment, especially in terms of language-related metrics (Naturalness and Expressiveness). * The moderate correlation of METEOR with human evaluation suggests that these automated metrics are not fully reliable as proxies for human judgment in code documentation generation tasks. * There is a significant variance in human judgment, highlighting the subjective nature of evaluating documentation quality by humans. This subjectivity complicates the task of developing automated metrics that align closely with human evaluations.	* Improving automated metrics: by incorporating content-related aspects, such as the presence of keywords relevant to the source code, to better reflect the adequacy and usefulness of generated documentation. * The study focused on Java datasets, and future work should explore other programming languages. * Further research is needed to create automated evaluation metrics that better capture the quality of documentation as perceived by human evaluators.
7	Qian et al. (2024). ChatDev: Communicative agents for software development.	**Multiple large language model (LLM) agents that communicate and collaborate with a purpose of software development**. The agents operate in a chat-powered framework where they engage in multi-turn dialogues, with specific roles assigned to each agent (e.g., CEO, CTO, programmer, reviewer, and tester). The process is structured using a "chat chain," breaking down the software development lifecycle into phases such as design, coding, and testing.	Newly developed **Software Requirement Description Dataset (SRDD)**: 1,200 software task prompts across five main areas: Education, Work, Life, Game, and Creation. These prompts were categorized into 40 subcategories, with each subcategory containing 30 unique tasks. ChatDev outperformed baseline methods (GPT-Engineer and MetaGPT) in all these metrics, indicating a significant improvement in the overall quality of the generated software.	Completeness Executability Consistency Quality	0.5600 0.8800 0.8021 0.3953	* Potential for coding hallucinations (requires manual intervention). * The need for clear, detailed requirements is emphasized, as vague guidelines can lead to lower-quality outputs. * Current metrics may not fully capture all aspects of software quality, such as robustness and user-friendliness. * Multi-agent approaches require more tokens and time, increasing computational costs.	* Enhancing agent capabilities with fewer interactions to reduce computational demands. * Expanding the evaluation metrics to include functionalities, robustness, safety, and user-friendliness. * Further development of datasets that better reflect real-world software development tasks. Improving the clarity and detail of software requirements to optimize agent performance.

Documentation Generation Results

#	Name	Approach / Method	Evaluation Dataset	Metric Type	Metric Value	Limitation / constraints	Recommended future work
8	Luo et al. (2024). RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation	RepoAgent utilizes an LLM powered framework to **generate, maintain, and update repository-level documentation automatically**. The approach involves three stages: * **Global Structure Analysis**: Parsing and analyzing the code's global structure and relationships using Abstract Syntax Tree (AST) analysis and the Jedi library. * **Documentation Generation**: Utilizing the parsed information to generate comprehensive documentation with detailed functionality, parameters, code descriptions, notes, and examples. * **Documentation Update**: Automatically updating documentation based on code changes using Git integration, ensuring synchronization between code and documentation.	RepoAgent was evaluated on **nine Python repositories** of varying sizes and complexity, **chosen for their classic status or high popularity on GitHub**. The specific repositories are described in Appendix A.1. RepoAgent demonstrated strong performance in both qualitative and quantitative evaluations: Documentation generated by RepoAgent was preferred over human-authored documentation, with win rates of 70% and 91.33% for the Transformers and LlamaIndex repositories, respectively..	Win rate: Transformers LlamaIndex Accuracy of identifying function parameters	70% 91.33% varies by LLM	* Programming Language Limitation: currently only Python support. * AI-generated documentation may still require human review for accuracy. * The performance of RepoAgent is highly dependent on the capabilities of the underlying LLMs. * There is a lack of standardized evaluation methods for assessing the quality and professionalism of generated documentation.	* Support multiple programming languages beyond Python. * Reduce dependency on specific LLMs to ensure long-term stability and wider applicability. * Establishing benchmarks and datasets for assessing the quality of generated documentation. * Apply RepoAgent to a broader range of downstream tasks and potentially integrating chat-based interactions for enhanced communication between code and humans.
9	Shubhang et al. (2024). A Comparative Analysis of Large Language Models for Code Documentation Generation	The paper **compares the effectiveness of several Large Language Models (LLMs) in generating code documentation across different levels, including inline, function, and file-level documentation**. The models evaluated include both closed-source (GPT-3.5, GPT-4, Bard) and open-source (Llama2, StarChat) LLMs. The analysis focuses on multiple evaluation metrics to assess the quality and efficiency of the generated documentation.	A set of **14 Python code snippets** was selected from well-documented public repositories, covering various documentation levels (inline, function, and file-level). These snippets were used to generate documentation using the five different LLMs under study. The performance of the LLMs was evaluated based on the following metrics: Accuracy, Completeness, Relevance, Understandability, Readability, and Time Taken. **Accuracy**: Most models performed well, with GPT-3.5, GPT-4, and Bard outperforming the original documentation, while StarChat lagged behind. **Completeness**: GPT-3.5 and GPT-4 excelled, surpassing the original documentation, while Llama2 and StarChat showed slightly lower performance. **Relevance, Understandability, and Readability**: Closed-source models (GPT-3.5, GPT-4, Bard) performed better or on par with the original documentation, while Llama2 and StarChat were less effective. **Time Taken**: GPT-4 took the longest time to generate documentation, followed by Llama2 and Bard, with GPT-3.5 and StarChat being the fastest.	Mean across all LLMs: Accuracy Completeness Relevance Understandability Readability Time Taken	 2.86 4.192 3.833 3.551 4.525 16.977	* The file-level documentation consistently performed worse across all metrics except time taken, highlighting the challenge of generating high-quality documentation at this level. * The study showed significant performance differences between closed-source and open-source LLMs, with closed-source models generally performing better. * There was a clear trade-off between the quality of documentation and the time taken to generate it, with more sophisticated models like GPT-4 being slower, but producing higher-quality outputs.	* Reduce the time taken to generate documentation without compromising quality, especially for models like GPT-4. * Explore new techniques or specialized models to enhance the quality of file-level documentation. * Conduct more extensive comparisons across a broader range of codebases and programming languages to validate the findings and improve LLM-based documentation tools.

Additional Materials and Potential Topic Change Proposal for the Meeting on August 18, 2024

Code Generation Datasets

Automatic code generation from text is a task reverse to documentation generation from source code. I could even use the same datasets but reverse the direction. But the true benefit from this task is the ability to use test cases for evaluation, and for this, I would need these specific datasets:

1. HumanEval

- Popular benchmark for evaluating code generation models - contains programming problems with corresponding unit tests that can be used to verify the correctness of generated solutions.

2. APPS (Automated Programming Progress Standard)

- A large set of programming problems, ranging from simple to complex. Includes test cases.

3. MBPP (Mostly Basic Python Problems)

- Designed to evaluate code generation models on Python programming tasks. It consists of a large number of Python problems with a set of unit tests that assess the correctness of the generated Python code.

4. SPoC (Student Programming Contest)

- Derived from student submissions in programming contests, including some with unit tests.

5. CodeContest

- Competitive programming problems with comprehensive test cases.

6. MultiPL-E

- HumanEval + MBPP translated into other programming languages.

Code generation leaderboard:
https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

Code Generation Metrics

1. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review
https://arxiv.org/pdf/2406.12655v1

2. Evaluating Large Language Models Trained on Code

- This paper presents the evaluation of Codex, a large language model trained on code. The authors use the HumanEval dataset, which includes unit tests, to assess the functional correctness of the code generated by Codex. https://arxiv.org/abs/2107.03374

3. L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models

- https://ui.adsabs.harvard.edu/abs/2023arXiv230917446N/abstract

4. "Program Synthesis with Large Language Models"

- https://arxiv.org/abs/2108.07732

5. Fully Autonomous Programming with Large Language Models
https://dl.acm.org/doi/10.1145/3583131.3590481

EvalPlus code generation leaderboard:
https://evalplus.github.io/leaderboard.html

Metrics Objectivity

The **documentation generation metrics** described in the above past results include the BLEU, ROUGE, METEOR and other similar scores:

- BLEU measures the precision of n-grams in the generated text relative to the reference text. It considers the proportion of n-grams in the generated text that also appear in the reference, with a penalty for shorter outputs.
- ROUGE primarily measures recall, focusing on the proportion of n-grams in the reference text that appear in the generated text. ROUGE variants can also measure precision and F1 score.
- METEOR is a more complex metric that incorporates both precision and recall of n-grams, with additional consideration for synonymy, stemming, and word order.

Limitations for semantic comparison:

- Reliance on n-grams: These metrics focus on exact or near-exact n-gram matches, which means they might not capture the true meaning of a text. Two texts could have different n-grams but convey the same idea, or they could have similar n-grams but different meanings.
- Lack of Context Understanding: These metrics do not account for context, logical structure, or deeper semantic connections, making them less effective for evaluating the overall meaning or intent behind the text.

On the other hand, **the code generation evaluation datasets** use test cases, e.g. in HumanEval and MBPP, which a more objective measure which makes more sense because the generated code either passes the tests or not.

Potential for a pivot

Should I change my Praxis topic from "**Using LLM-Based Agents to Automatically Generate Documentation from Source Code**" to "**Using LLM-Based Agents for Automatic Code Generation**"?

Pros

- This is a broader and, thus, *more important task*.
- It has a *much bigger impact* on the software engineering industry and is, thus, *more interesting*.
- It has *more objective metrics* of whether the generated content is successful (test cases).
- *More work was done on code generation in the past* including agents while I was able to find only 9 papers for documentation generation, and only 3 of them use the CodeSearchNet dataset and BLEU scores for reproducibility, plus some papers use outdated models (BERT) or methods.
- There are *code generation leaderboards with past results* while there are none for documentation generation.

Cons

- *The competition is very tight* in the code generation domain with all the big players working hard on improving their models e.g. OpenAI's GPT-4o or Meta's Llama 3 models – it will be challenging to improve the state-of-the-art results already shown on several leaderboards.
- Most probably, by using agents I could still show *improvements over the majority of models*.

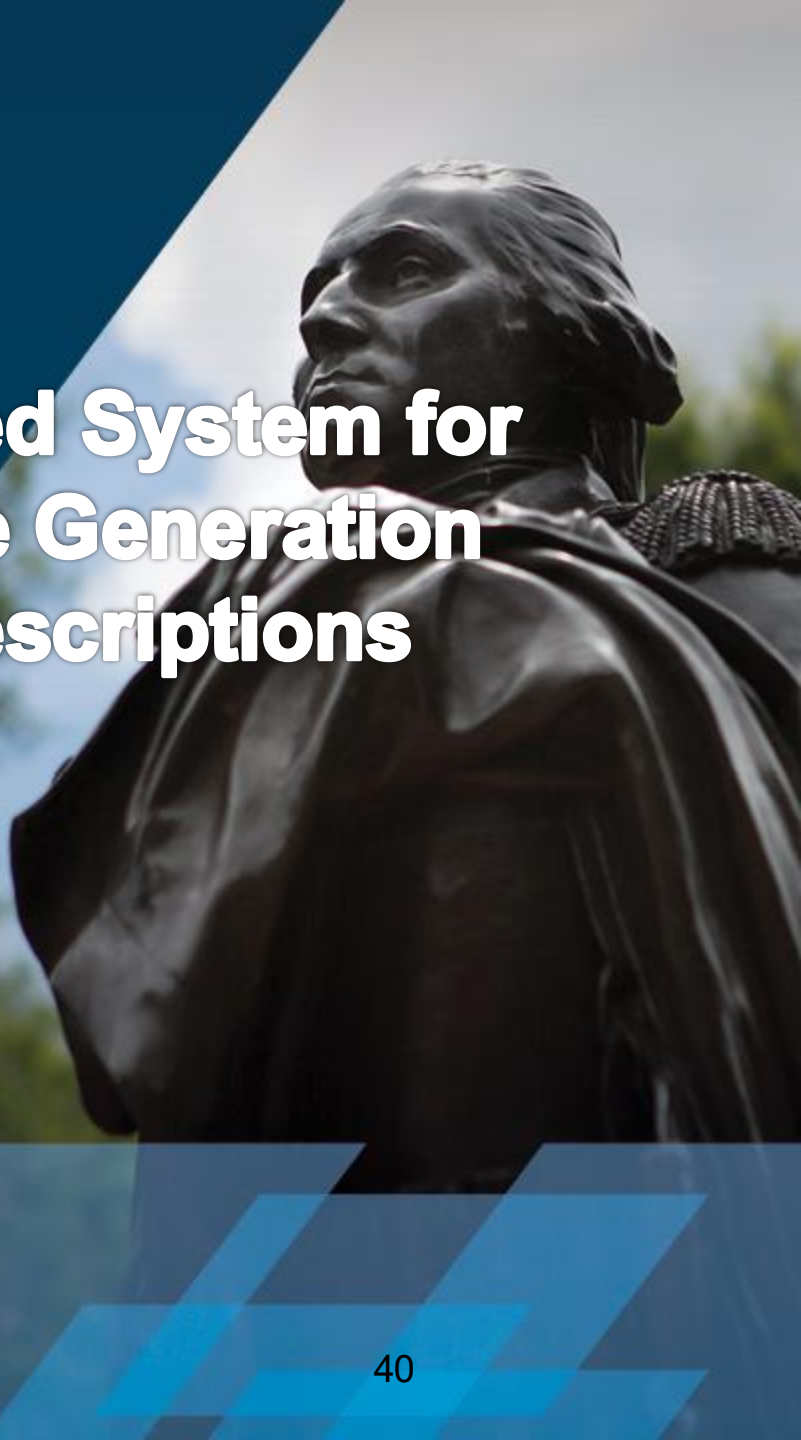
Materials for the Meeting on September 1 and September 21, 2024

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Artificial Intelligence-Based System for Boosting Automated Code Generation from Natural Language Descriptions

Andrew Nedilko



Scope of Work (SOW)

Developing a System Using Small Language Model-Based Agents to Improve the Automatic Generation of Code from Natural Language Descriptions in Python Software Engineering Projects and Comparing the Results with Automatic Code Generation Using Proprietary Large Language Models (LLMs) in Terms of the Generated Code Quality.

Problem Statement

(A) Deliverable	(B) Format	(C) WC
Issue	Using proprietary large language models (LLMs) to automatically generate code is costly and not safe from the sensitive data protection and intellectual property standpoints (Yan et al., 2024).	*23*
Reference	Yan, B., Li, K., Xu, M., Dong, Y., Zhang, Y., Ren, Z, Cheng X. (2024). On Protecting the Data Privacy of Large Language Models (LLMs): A Survey. https://arxiv.org/pdf/2403.05156	NA
"so what"	Forcing developers to spend twice as much time writing code manually (McKinsey Digital, 2023)	*11*
Reference	McKinsey Digital (2023). Unleashing developer productivity with generative AI. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/unleashing-developer-productivity-with-generative-ai	NA
Problem statement	Using proprietary large language models (LLMs) to automatically generate code is costly and not safe from the sensitive data protection and intellectual property standpoints forcing developers to spend twice as much time writing code manually.	*34*
Industry	Software Engineering	NA
PS elaboration 1	Proprietary LLMs are expensive in deployment and/or inference and expose sensitive data, pushing teams to code manually, slowing development and increasing costs.	*22*
PS elaboration 2	Data privacy and intellectual property risks with proprietary LLMs discourage their use, compelling developers to spend more time coding manually.	*20*

Thesis Statement

(A) Deliverable		WC
Thesis Statement	Agents based on open-source small language models (SLM) deployed in resource-constrained environments for automated code generation will ensure lower costs and sensitive data protection, reducing the manual coding time and speeding up development cycle.	*32*
Research Product	SLM-based agents	*NA*
Format	Python script(s)	*NA*
Deliverable Usage	Python software developers will use this product to automatically generate code while ensuring sensitive data protection and reducing time for manual coding.	*22*
Tie back to PS	By paving the road for automated code generation, SLM-based agents reduce the overall time developers spend writing code while still preserving data privacy.	23
New Contributions	This research introduces a novel approach by leveraging SLM-based agents to automate code generation from natural language descriptions, surpassing SLMs and approaching the proprietary LLMs in code quality.	*30*
Scope	Developing an AI system using SLM-based agents to improve automated code generation from natural language descriptions in software engineering.	*19*
Main methodology	Machine learning	*NA*
Inputs	Docstrings: text, code descriptions: text	*NA*
Outputs	Functions: text, code snippets: text, code files: text, code repositories: folder of text files	*NA*

Research Questions

(A) Deliverable	(B) Format	(C) WC
Research Question 1	Will fine-tuning Small Language Models used by agents result in higher code generation quality as measured by the maintainability index?	*23*
Research Question 2	Will changing Small Language Model parameters, such as temperature and top-p, ensure greater code quality based on lower cyclomatic complexity?	*18*
Research Question 3	Which agentic workflow, reflection or multi-agent collaboration, leads to a greater number of tests passed?	*19*

Research Hypotheses

(A) Deliverable	(B) Format	(C) WC
Hypothesis 1	Fine-tuning Small Language Model on domain-specific data will noticeably increase the maintainability index compared to using an LLM without fine-tuning.	*20*
Independent Variables	Fine-tuning of LLM	*NA*
Dependent Variable	Maintainability index	NA
Testable	Set up experiments with the two LLMs and compare the maintainability indices.	*12*
Hypothesis 2	Adjusting Small Language Model parameters, such as temperature and top-p, will noticeably improve the cyclomatic complexity of auto-generated code.	*18*
Independent Variables	Temperature, top-p	*NA*
Dependent Variable	Cyclomatic complexity	NA
Testable	Set up experiments when LLM parameters are adjusted and not adjusted and compare the cyclomatic complexity.	*16*
Hypothesis 3	Multi-agent collaboration will lead to a noticeably greater number of tests passed compared to the reflection agentic workflow	*18*
Independent Variables	Reflection, multi-agent collaboration agentic workflows	*NA*
Dependent Variable	Number of tests passed	NA
Testable	Set up experiments with two agentic workflows and compare the number of tests passed.	*14*

Code Generation Metrics

When measuring the quality of automatically generated code, a **combination of several metrics** designed to measure different aspects of code may provide a more comprehensive approach to evaluating the quality of automatically generated code.

1. **Functional Correctness**: comprehensive test cases to verify the code produces the correct outputs. Good code should pass all relevant tests.
2. **Code Coverage**: measures the percentage of code executed during testing. Achieving high coverage ensures most parts of the code are tested.
3. **Cyclomatic Complexity** (code reliability and maintainability using radon): evaluates the complexity of functions or methods by counting the number of linearly independent paths. Lower complexity indicates simpler, more maintainable code.
4. **Code Adherence to Best Practices**, for example PEP 8 Compliance for Python (e.g., using pylint, pycodestyle, flake8): checks if the generated code follows Python's style guide (PEP 8), which emphasizes readability and consistency.
5. **Code Comments and Docstrings** (e.g., pydocstyle): analyzes the presence and quality of docstrings and inline comments, ensuring the code is well-documented and easy to understand.
6. **Execution Time and Memory Usage** (e.g., timeit, memory_profiler): benchmarks Python code to evaluate its runtime performance and memory consumption. Optimized code should have minimal time and memory overhead.

Code Generation Metrics (2)

7. **Semantic Match**: scores like BLEU, ROUGE, METEOR measure how closely the generated Python code matches human-written reference implementations for the same functionality.
8. **Static Analysis Tools** (e.g., bandit): tools like bandit can check for security vulnerabilities, potential bugs, and code smells in Python code.
9. **Maintainability Index** (e.g., using radon): calculated based on factors like cyclomatic complexity, lines of code, and comment density, providing a single score that reflects the maintainability of the Python code.
10. **Modularity and Cohesion**: measures how well the code is organized into functions, classes, or modules, with each having a single, well-defined purpose. Ideally, the code should have high cohesion and low coupling.
11. **Duplication Metrics** (e.g., using flake8 plugins): detects duplicated code blocks to encourage reusable functions or classes instead of repeated code.
12. **Exception Handling** : reviews how well the code manages errors and exceptions using try-except blocks and custom exceptions. Robust error handling practices indicate higher quality.

Code Generation Metrics - References

1. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review

- https://arxiv.org/pdf/2406.12655v1

2. Evaluating Large Language Models Trained on Code

- This paper presents the evaluation of Codex, a large language model trained on code. The authors use the HumanEval dataset, which includes unit tests, to assess the functional correctness of the code generated by Codex. https://arxiv.org/abs/2107.03374

3. L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models

- https://ui.adsabs.harvard.edu/abs/2023arXiv230917446N/abstract

4. "Program Synthesis with Large Language Models"

- https://arxiv.org/abs/2108.07732

5. Fully Autonomous Programming with Large Language Models

https://dl.acm.org/doi/10.1145/3583131.3590481

EvalPlus code generation leaderboard:

https://evalplus.github.io/leaderboard.html

Code Generation Datasets (Func. Tests)

1. HumanEval

- Popular benchmark for evaluating code generation models - contains programming problems with corresponding unit tests that can be used to verify the correctness of generated solutions.

2. MBPP (Mostly Basic Python Problems)

- Designed to evaluate code generation models on Python programming tasks. It consists of a large number of Python problems with a set of unit tests that assess the correctness of the generated Python code.

3. APPS (Automated Programming Progress Standard)

- A large set of programming problems, ranging from simple to complex. Includes test cases.

4. SPoC (Student Programming Contest)

- Derived from student submissions in programming contests, including some with unit tests.

5. CodeContest

- Competitive programming problems with comprehensive test cases.

6. MultiPL-E

- HumanEval + MBPP translated into other programming languages.

Code generation leaderboard:
https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

Semantic Match

Some common natural language generation metrics include the BLEU, ROUGE, METEOR scores and their variations:

- BLEU measures the precision of n-grams in the generated text relative to the reference text. It considers the proportion of n-grams in the generated text that also appear in the reference, with a penalty for shorter outputs.
- ROUGE primarily measures recall, focusing on the proportion of n-grams in the reference text that appear in the generated text. ROUGE variants can also measure precision and F1 score.
- METEOR is a more complex metric that incorporates both precision and recall of n-grams, with additional consideration for synonymy, stemming, and word order.

Limitations of such metrics:

- Reliance on n-grams: These metrics focus on exact or near-exact n-gram matches, which means they might not capture the true meaning of a text. Two texts could have different n-grams but convey the same idea, or they could have similar n-grams but different meanings.
- Lack of Context Understanding: These metrics do not account for context, logical structure, or deeper semantic connections, making them less effective for evaluating the overall meaning or intent behind the text.

On the other hand, **the code generation evaluation datasets** use test cases, e.g. in HumanEval and MBPP, which seem to be more objective measures of how a model accomplished a code generation task. Another more interesting metric, as opposed to comparing n-grams, would be embedding both the ground truth and the generation code and comparing the two embeddings (e.g. using the cosine distance)

Papers (1)

1. Ahmed Soliman, Samir Shaheen, Mayada Hadhoud. **Leveraging pre-trained language models for code generation**.

This paper explores new hybrid models for code generation using pre-trained language models like RoBERTa, ELECTRA, etc. The models are evaluated on the CoNaLa and DJANGO datasets, demonstrating significant improvements in code generation accuracy.

This article is highly relevant to code generation research as it investigates how pre-trained transformer-based language models can be effectively adapted to generate code. The hybrid model approach and performance improvements offer valuable insights into optimizing language models for generating accurate, contextually appropriate code.

2. Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, Zhi Jin. 2024. **Enhancing Code Generation Performance of Smaller Models by Distilling the Reasoning Ability of LLMs**.

LLMs have significantly improved code generation using the Chain-of-Thought prompting solving complex programming tasks. However, smaller models struggle to match this reasoning ability. This paper introduces the CodePLAN framework, which transfers LLMs' reasoning capabilities to smaller models through distillation, using multi-task learning for code and solution plan generation. This article is relevant to my research as it addresses a key challenge: improving the performance of smaller models to match the reasoning abilities of LLMs.

3. Gurusha Juneja, Subhabrata Dutta, Soumen Chakrabarti, Sunny Manchhanda, Tanmoy Chakraborty. **Small Language Models Fine-tuned to Coordinate Larger Language Models Improve Complex Reasoning**.

This paper introduces a modular approach that separates problem decomposition from solution generation, using a smaller language model to decompose complex problems into simpler subproblems and a solver LM to generate solutions. This article is relevant to my research as it highlights the benefits of modular approaches for handling complex reasoning and code generation tasks.

Papers (2)

4. Minh Huynh Nguyen∗, Thang Chau Phan∗, Phong X. Nguyen∗, Nghi D. Q. Bui. **Dynamic Collaborative Agents for Software Development based on Agile Methodology**

The article describes a multi-agent system for software development. It assigns Agile roles (Product Manager, Developer, Tester) to agents working in sprints. A Code Dependency Graph (CDG) is to aid code comprehension and generation. The multi-agent approach and use of Dynamic CDG are directly applicable to developing AI-based documentation systems.

Takeaways: a) Multi-agent collaboration: effective role-specific agents can enhance automated documentation, b) Dynamic code graphs are essential for maintaining accurate and current documentation.

5. Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, Ju¨rgen Schmidhuber. **MetaGPT: meta programming for a multi-agent collaborative framework**.

The article talks about a framework that enhances multi-agent collaboration by integrating Standardized Operating Procedures (SOPs) into LLM-based systems. It assigns specific roles (e.g., Product Manager, Engineer) to agents, promoting efficient task decomposition and reducing errors.

Relevance my research: this is highly relevant for improving automated documentation generation. Implementing SOPs can enhance the quality and consistency of generated documentation. Additionally, the role-based agent collaboration model can be adapted to develop specialized agents for different documentation tasks, ensuring comprehensive and accurate outputs.

Papers (3)

6. Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, Chi Wang. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation**.

AutoGen is an open-source framework that uses multiple interacting agents to create applications. The framework supports various modes of operation, allowing agents to converse and collaborate on tasks. The framework employs "conversation programming," where agent interactions are controlled through both natural language and programming languages, enhancing the modularity and scalability of multi-agent systems.

Relevance: the principles and capabilities of AutoGen align closely with my research topic. The framework's support for multi-agent collaboration and conversation-based workflows can significantly aid in developing an LLM-based system for automated documentation. The customizable agents and conversation programming features of AutoGen can be utilized to create a robust and adaptable documentation generation system, ensuring high-quality.

Benefits of Automatic Code Generation

- According to McKinsey's research "Unleashing developer productivity with generative AI", GenAI can significantly **decrease the time developers spend on coding (up to 45%)** which can enable companies to reduce labor costs. This can be especially impactful for large-scale projects where efficiency gains translate into significant cost reductions.
- According this study by the same company "The state of AI in early 2024: Gen AI adoption spikes and starts to generate value", the AI adoption accelerates and starts creating real value.
- In its study "Quantifying GitHub Copilot's impact on developer productivity and happiness", GitHub concluded that **developers using GitHub Copilot finished their task much faster – 55% faster** than the developers who preferred not to use GitHub Copilot.
- But developer productivity goes beyond speed - between 60–75% of developers reported they **feel more fulfilled with their job**, feel less frustrated when coding, and are **able to focus on more satisfying work** when using GitHub Copilot. Also, developers reported that GitHub Copilot helped them stay in the flow (73%) and preserve mental effort during repetitive tasks (87%).

Benefits of Automatic Code Generation (2)

- **Improved Code Quality** through consistency and adherence to best practices: automatically generated code adheres to coding standards and best practices consistently, reducing the likelihood of human errors and bugs as detailed in these studies: ”AICodeReview: Advancing code quality with AI-enhanced reviews” and “Impact of AI Tools on Software Development Code Quality”.
- **Faster Time-to-Market as a Competitive Advantage**: Accelerated development cycles allow companies to bring products to market more quickly, providing a significant competitive edge according to “Gartner Top Strategic Technology Trends for 2023”.
- **Reduction in code maintenance effort**: consistent and well-structured code generated by automation tools is easier to understand and maintain, reducing long-term maintenance costs.

Materials for the Meeting on October 1, 2024

Praxis Timeline

Date	Deliverable
October 5	
October 19	**Benchmarking: Dataset and Leaderboard Selection**
October 31 / November 2	Chapter 1 Introduction section (7 pages)
November 16	
November 30	**Model Selection and Fine-Tuning**
December 14	Chapter 2 Literature Review section (18 pages)
December 28	**Agentic Workflow Implementation**
January 11	
January 25	Chapter 3 Methodology (16 pages)
February 8	**Evaluation and Interpretation**
February 22	Chapter 4 Results (28 pages)
March 8	**Documentation and Reporting**
March 22	Chapter 5 Discussion and Conclusions (3 pages)
April 5	References (7 pages), Appendices (21 pages, preferred < 10)
April 19, May 3, May 17	

Notes

- Chapters are Praxis chapters
- Other elements **in bold** are Praxis components described on the subsequent slides
- Total counts for reference:
  - Main body total 72 (expected to be 70-90)
  - Total 116 (prefer < 110)

Praxis Components

1. Benchmarking: Dataset and Leaderboard Selection

- Download and prepare for use several code generation evaluation datasets.
- Get a list of code generation leaderboards using LLMs and, if possible, SLMs. These benchmarks will be used to compare with my research results.
- Analyze the value of each dataset / leaderboard for research purposes and generate a shortlist of code generation evaluation datasets.
- For each shortlisted dataset, test run the main code generation evaluation method(s) that was used to generate the previous benchmarks.
- Decide if the data in the datasets needs filtering or cleaning (depending on how the previous benchmark tests were done). Potential cleaning may include:
  - ✓ Filter examples with too few or too many tokens.
  - ✓ Filter examples in another language (not English).
  - ✓ Remove or clean documentation that contains special tokens (e.g. “<img …>”, “https:”, and so on)
- If possible and feasible, download and prepare for use several code generation training datasets to fine-tune SLMs with the purpose to generate better quality code.

Praxis Components (2)

2. Model Selection and Fine-Tuning

- Select pre-trained Small Language Models (SLMs) suitable for the task (e.g., Llama 8, Mistral 8, etc.). Analyze any preliminary results of their application for code generation.
- Establish the required APIs to use the models for inference and fine-tuning and / or
- Procure resources required to fine-tune the model (cloud GPU instance, Google Colab, etc.).
- If feasible, fine-tune the models on the training dataset(s) with an emphasis on text-to-code generation. Experiment with different architectures, pre-training objectives, and fine-tuning techniques such as SFT, DPO, ORPO etc.
- If feasible, adjust hyperparameters (e.g., learning rate, batch size) to improve the fine-tuned model performance and experiment with different SLM parameters, such as temperature and top-p, to achieve better results.

3. Agentic Workflow Implementation

- Develop agents based on reflection, multi-agent collaboration, and potentially other agentic workflows.
- Integrate original and, if feasible, fine-tuned SLMs into the agents.
- Define metrics, other than the number of tests completed, to evaluate the generated code, e.g. semantic similarity, etc.

Praxis Components (3)

4. Evaluation and Interpretation

- Generate code using the developed agents and measure the number of tests passed.
- Compare the results with current best tests shown on the leaderboard.
- Evaluate other code quality metrics.
- Analyze code generation results to draw conclusions.
- Interpret the implications of findings for the AI and software engineering communities.

5. Documentation and Reporting

- Document the entire research process.
- Prepare the Praxis report, including all findings, methodologies, and implications.
- Review and revise the Praxis based on feedback from the advisor(s).

Materials for the Meeting on October 15, 2024

Leaderboards, datasets, code

See a separate submission in “Leaderboards-datasets-code.docx”

Materials for the Meeting on November 2, 2024

Initial SLM Candidates

1. **Llama 3** – an advanced language model from Meta considered one of the best open-source models in its category. Description: https://ai.meta.com/blog/meta-llama-3/ . Usage: https://huggingface.co/meta-llama/Meta-Llama-3-8B.

2. **Mixtral** – advanced mix of experts for better reasoning. One of the best small language models out there. It's able to leverage a wide spectrum of knowledge through a blend of various domains. Mixtral creates new models capable of running on local machines while still achieving comparable power to full-scale LLMs. Description: https://mistral.ai/news/mixtral-of-experts/. Usage: https://huggingface.co/docs/transformers/en/model_doc/mixtral.

3. **DeepSeek-Coder-V2** – among the best small language models for code generation. Description and usage examples: https://github.com/deepseek-ai/DeepSeek-Coder-V2.

Note: most of the effort was dedicated to submitting the *Introduction* Chapter.

Title

Text