ACTA SCIENTIAE

ISSN: 2178-7727

Low-Code API Generation with LLMs: How GPT-Based Models Are Changing API Development

Anjali Kumari[1], Vegda Jayesh[2], Harvani RadhaKumari[3], Sama Anudeep[4], prof. Vivek Dave[5]

[12345] Parul Institute of Engineering and Technology-MCA,
Faculty of IT and Computer Science,Parul University
[5]Head of PG Department, Parul Institute of Engineering and Technology-MCA,
Faculty of IT and Computer Science,Parul University

ABSTRACT

Relying on prebuilt elements and visual interfaces, LCDPs or low-code development platforms are making it far easier and faster to create software. API development is undergoing transformation with the incorporation of large language models (LLMs) such as GPT which makes development is faster, simpler, and more cost effective. Development of APIs requires substantial skill but with LLMs, developers and non-developers can easily create, modify, and integrate APIs with ease.

The study seeks to find out how LLMs assist in the automation processes of testing, documentation, and code generation to create low-code APIs. By incorporating AI, organizations can enhance system integration, reduce human effort and make the workflows more efficient when combined with low-code platforms. This exploration is done with the use GPTEngineer and AutoGen to determine optimal coding techniques through prototyping and testing the feasibility of LLM driven API development.

To automatically generate Java REST APIs, a unique framework is devised known as AutoAPIEval. Instead of the single LLM prompt used for the models chatGPT, stepwise prompting and structured planning are more effective at achieving the desired objective. Performance metrics such as lines of code (LOC) written, along with the ability to create files are used to assess the effectiveness of various LLMs.

INTRODUCTION

"Let's be real, software development can be a major headache. But Low-Code Systems Development (LCSD) is a breath of fresh air. It's all about ditching the coding drudgery and building apps faster and easier. Think visual tools, drag-and-drop interfaces, and model-based designs – basically, making development way less of a slog, thanks to these cool low-code platforms.

These platforms are like souped-up software development kits. They give you a graphical user interface and all the pre-built goodies – user interfaces, business objects, data tables, you name it. You just mix and match these components, tweak the data fields, UI, and business logic, and bam – you've got a working app.

And APIs? Ugh, they used to be a nightmare, demanding serious coding skills and a ton of time. But now, with these user-friendly platforms and smart programs (like GPT-powered large language models), it's a whole new ballgame. Suddenly, coders *and* non-coders can create, modify, and use APIs without getting bogged down in code.

This combination of simplified development and AI smarts is a total game-changer for businesses. They can build APIs more smoothly, automate practically everything, and connect with other systems seamlessly. These intelligent programs can even help write API code, generate documentation, run tests, and even suggest improvements. Less manual work, faster development

We'll look at the upsides, the challenges, and what the future might hold for these tools, showing how they're changing the way developers and companies build APIs.

ISSN: 2178-7727

Conceptual Framework

This section breaks down the basics of Low-Code Development Platforms , Generative AI , and AI-Generated Content , and why they matter in today's world of software development and automation.

Understand Low-code development platforms

Al-

Imagine building software without the coding headaches. That's the promise of low-code development. These platforms empower businesses to create applications quickly and efficiently, without requiring a huge investment in coding expertise And that saves a ton on setup, training, and getting things up and running. LCDPs have come a long way, evolving into cloud-based powerhouses with visual, model-driven environments. You don't need to be a tech whiz to build an app anymore.

LCDPs are basically software development kits that let you design applications, workflows, and automation using graphical interfaces, not lines and lines of code. Since many live in the cloud, they play nice with popular SaaS apps like Zapier, Amazon Web Services, and Trello, which makes connecting and automating things a breeze.

The whole point of low-code platforms is to make software creation, deployment, and maintenance easier and cheaper than traditional coding. They even empower "citizen developers" – people without formal coding training – to build applications. This means more innovation across the company, and it helps organizations keep up with the fast pace of digital change.

A big plus of LCDPs is their "What You See Is What You Get" (WYSIWYG) approach. What you design visually is pretty much what you get in the final app. With easy drag-and-drop and pre-built templates, LCDPs make app development accessible to pretty much anyone, so they can build real-world software solutions without pulling their hair out.

Understanding Generative AI and AI-Generated Content

Generative AI (GenAI) is a real game-changer. It's transforming how we create and use digital content. Using advanced machine learning, GenAI can create text, images, music, and videos that look and sound incredibly human. It's a powerful tool for all sorts of industries, from automating content creation to making customer interactions way better. GenAI is changing how we work and making digital asset production faster and more efficient.

The brains behind GenAI are Large Language Models (LLMs). These are specialized neural networks that process and generate sequences of data, especially text. Trained on mountains of written material, they can predict and create language that's coherent and makes sense in context. By spotting patterns in all that data, LLMs let AI systems create responses that are meaningful and sound natural.

This field has made huge leaps thanks to innovations like self-attention mechanisms and positional encoding. These advancements have made a real difference in how well these models understand and generate language. And the development of Generative Pre-trained Transformer (GPT) models, like ChatGPT, is a direct result of all this progress.

METHODOLOGY

This section describes how the research is conducted to answer the defined research questions.

We use literature and the knowledge gained from examining existing solutions to prototype a custom solution and then use it to examine the application of it for autonomic code generation within the specific Ericsson system. We then evaluate the outputs of the tool and use the output to analyze the feasibility of the usage of

ISSN: 2178-7727

the tool by conducting interviews with Ericsson developers that are familiar with the target application. A map of the phases the study undergoes can be found in the appendix A.1.

Researching prompting methods

We research ways to improve code generation by studying existing methods, tools, and research. We find relevant literature from sources like IEEE Xplore, Scopus, and arXiv using search terms related to AI, LLMs, and prompt engineering. Additionally, we explore GitHub to identify software projects that could aid in code generation.

We test different software by running them with LLMs and comparing their results to a single, well-structured prompt. If a software-LLM combination performs well, we analyze its key contributing factors through its documentation and source code. Our evaluation process involves systematically modifying prompts while keeping the input conditions uniform, such as using the same OpenAPI specification for REST APIs.

To understand the impact of prompt orchestration, we only tweak prompts and orchestration methods while keeping other software and LLM parameters unchanged. The number of tests per software depends on its potential usefulness. Finally, we apply our findings to develop our own solution and conduct experiments to measure whether it improves code generation compared to a single, comprehensive prompt.

Prompting methods in existing works

We tested different tools to improve code generation with LLMs:

- GPTEngineer breaks tasks into smaller steps and prompts the LLM for each step. This helps fit complex tasks within the LLM's context window.
- MemGPT stores important details from prompts for later use. However, it struggled to retain full API specifications, limiting its effectiveness.
- AutoGen creates multiple LLM agents that collaborate. However, when using it, the LLMs struggled to correctly decide which agent should handle the next step, leading to errors. A manually controlled flow worked better.

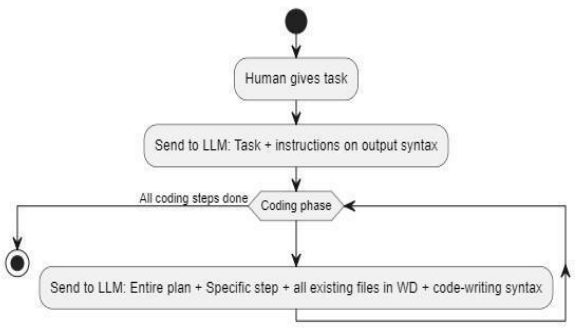
Solution development

This helps us compare its performance against existing frameworks. Python is chosen for its simplicity and rapid development benefits, despite being slower than compiled languages. Our goal is to combine the strengths of previous tools while addressing their limitations for more accurate and relevant code generation.

Design of implementation

We design our custom tool to generate Java REST APIs autonomously, balancing detailed context in prompts without overwhelming the LLM. To improve accuracy, we include key details like the task, existing files, completed features, and file-writing instructions. A planning phase is introduced, where the LLM first outlines the necessary files and their contents before coding. This structured approach improves code generation while keeping the LLM focused and organized.

Experiment design



ISSN: 2178-7727

We test different LLMs to evaluate their ability to generate a Java REST API using two methods: phased prompting and a single comprehensive prompt. The LLMs are randomly selected from open-source models compatible with the Ollama platform to avoid bias. Each model runs the phased prompting process three times and the single comprehensive prompt three times to get an average performance. We then collect and analyze all generated code and related documents.

Experiment design

- Average number of relevant LOC: The number of lines relevant to the task, indicating the precision of the generated code.
- Percent relevant LOC: The percentage of LOC that are relevant out of the generated LOC.
- Average number of files successfully written to: The number of files that the LLM successfully writes to. A write is only deemed successful if only the code is written to the file. No characters that are products of the code-writing syntax should be written to the file for the file-write to be deemed successful, i.e. the ‘-characters encapsulating the code in the Markdown syntax for code blocks [11].
- Average number of times LLM failed to write to a file: Number of times the LLM failed to write to a file by not following the code-writing syntax successfully. Includes the times the LLM produced code relevant to the task, but did not follow the file-writing syntax at all.
- Percent successful files-writes: The percentage of the times the code successfully was written to a file.

We run the phased prompting solution on a laptop, while LLM inferences are processed on a more powerful server with an Intel Xeon CPU, 131.6GB RAM, and an NVIDIA RTX A6000 GPU. The setup uses Python 3.10.12 on Ubuntu 22.04 (WSL) and runs LLMs via the Ollama API in Docker. We test several high-performing LLMs compatible with our hardware, downloaded from Ollama on April 30, 2024.

Interview

Interviews are conducted with Ericsson developers that are familiar with the target REST API in order for us to answer research question 3 3. The interviews are conducted in a structured format asking each question consecutively, focusing on the value of the AI-generated output [37]. The sessions

CONCLUSION

We present AutoAPIEval, a simple framework for evaluating LLMs in API-based code generation. It works with any library offering API documentation and assesses tasks like API recommendation and code example generation using key evaluation metrics. Testing ChatGPT, MagiCoder, and DeepSeek Coder on JRE 8 revealed that ChatGPT followed instructions well but often generated unexecutable code. Factors like API popularity and model confidence influenced quality, and retrieval-augmented generation showed mixed results across models. Our study provides insights for improving LLM-based code generation.

REFERENCES

1. J. Gruber, “Markdown,” 2004. [Online]. Available: https://daringfireball.net/ projects/markdown/
2. Q. Wang, L. Ding, Y. Cao, Z. Tian, S. Wang, D. Tao, and L. Guo, “Recursively summarizing enables long-term dialogue memory in large language models,” arXiv:2308.15022 [cs.CL], 2023.
3. Ollama, “Ollama - get up and running with large language models locally,” accessed on: 06/05/2024. [Online]. Available: https://github.com/ollama/ ollama
4. J. C. Process, The Java® Language Specification, Java SE 22 Edition. Oracle, 2024. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se22/html/
5. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering, 1st ed. Berlin: Springer, 2012, vol. 9783642290442.