2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00019

Smart Prompt Advisor: Multi-objective Prompt Framework for Consistency and Best Practices

Kanchanjot Kaur Phokela*, Samarth Sikand*, Kapil Singi, Kuntal Dey, Vibhu Saujanya Sharma, Vikrant Kaulgud
Accenture Labs, India
{kanchanjot.k.phokela, s.sikand, kapil.singi, kuntal.dey, vibhu.sharma, vikrant.kaulgud}@accenture.com

***Abstract*—Recent breakthroughs in Large Language Models (LLM), comprised of billions of parameters, have achieved the ability to unveil exceptional insight into a wide range of Natural Language Processing (NLP) tasks. The onus of the performance of these models lies in the sophistication and completeness of the input prompt. Minimizing the enhancement cycles of prompt with improvised keywords becomes critically important as it directly affects the time to market and cost of the developing solution. However, this process inevitably has a trade-off between the learning curve/proficiency of the user and completeness of the prompt, as generating such a solutions is an incremental process. In this paper, we have designed a novel solution and implemented it in the form of a plugin for Visual Studio Code IDE, which can optimize this trade-off, by learning the underlying prompt intent to enhance with keywords. This will tend to align with developers' collection of semantics while developing a secure code, ensuring parameter and local variable names, return expressions, simple pre and post-conditions, and basic control and data flow are met.**

***Index Terms*—Prompt Engineering, Artificial Intelligence, Deep Learning, LLM, Ontology**

I. Introduction

In recent times, Large Language Models (LLM) have reached unprecedented levels of popularity due to their unparalleled performance in human-bot conversations, e.g., ChatGPT, code generation and suggestion, e.g., GitHub Copilot as well as other NLP tasks. Majority of these tools, built upon LLMs, have a key component which drives them i.e., natural language prompts. There is an increasing interest in understanding how such models can seamlessly be integrated with the existing software development lifecycle and their impact on developers.

Qualitative and Quantitative aspects of code generation tools such as Copilot and Codex are vividly studied. Studies have highlighted vulnerable code generation [1], use and function correctness [2] of code generation tools by conducting controlled trials with varied sets of programmers and developers.

For generating comprehensive results, the onus is on articulating prompts. While various methodologies to enhance prompt for code generation tools have been conducted, such as incorporating examples in prompt generation for more complete results. It is observed that these experiments were controlled trials with little correlation to enterprise software development. The below issues show the impact on developers during their interaction with code generation tools.

*Equal Contribution

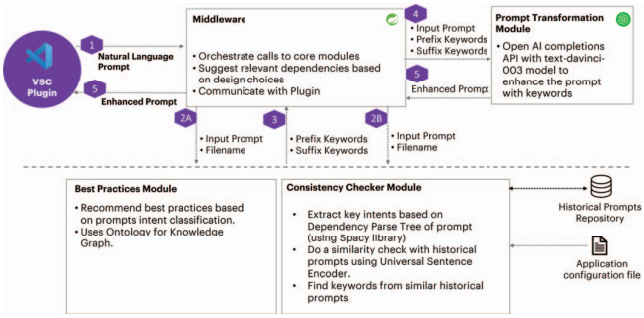


Fig. 1: Architecture of Smart Prompt Advisor.

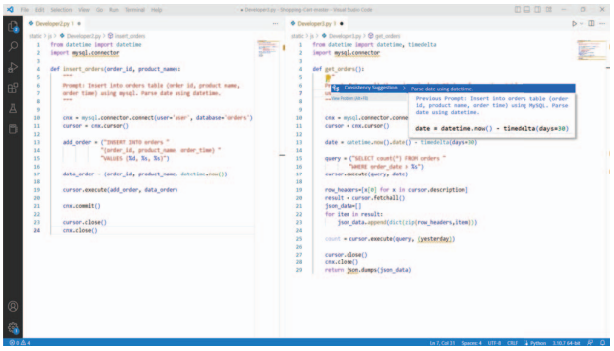
- Each organization has its own set of code principles related to security, coding best practices etc. which need to be adhered by the developers. During the traditional approach of coding, developers tend to have a semantic context of best practices and process flow. As developer mental models shift from code to prompts, it is observed that these semantic contexts are often missed out when articulating prompts.
- Prompt-based development is prone to inconsistencies arising due to differences in prompt articulation. Developers often use diverse ways of articulating their intents in prompts, leading to haphazard and non-standardised ways to create prompts [3]. This could lead to issues like code smells, bugs, vulnerabilities, and data-flow, which might arise at later stages of integration and/or testing.

To solve the issues described, we propose **Smart Prompt Advisor (SPA)** which will help developers create better prompts by automatically incorporating organization *best practices* and leveraging historical prompts to enforce *consistency*. The enhanced prompts boost the probability of generating optimal results from such tools with minimal iterations.

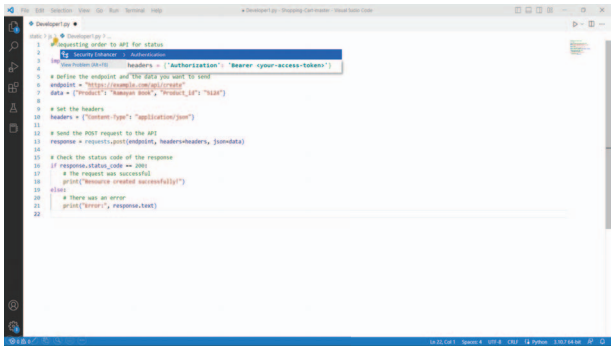
II. Smart Prompt Advisor

SPA is comprised of three key components: *Best Practices Module (BPM)*, *Consistency Checker Module (CCM)* and *Prompt Transformation Module (PTM)*, as shown in Fig. 1. The middleware communicates with BPM and CCM module(both executing in parallel), and redirects necessary data, like input prompt and respective working file, to the appropriate module. The output from BPM and CCM is then passed to PTM module for prompt transformation. The enhanced prompt

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00019



(a) Prompt with project specific consistency enhancements



(b) Prompts with code quality practices like security best practices

Fig. 2: Smart Prompt Advisor

is then sent to the user as suggestions. As depicted in Fig. 2, the IDE plugin provides developer with distinct versions of prompt enhancements encompassing varied improvements to cater to the context of input prompt.

Once the prompts are received, first step is classifying the prompts into following categories (by applying a rule-based algorithm and NLP techniques): (a) vague (unclear intent) (b) complex (multiple unrelated components in one prompt) (c) incomplete (missing actions or intent) (d) abstract prompts (with no or minimal action details) and (e) complete prompts. The prompts are further processed based on their classification, based on three conditions (i) complete prompts are passed to Enhancement sub-method which identifies the constraints to appended, if any (ii) complex prompts are broken down into multiple micro-prompts via a custom prompt simplification sub-method (iii) prompts from remaining categories are reverted back to the user for further refinements and post-refinement are re-evaluated.

The *enhancement* sub-method consists of two steps (i) Identification of the intent and the corresponding technology. ii) Mapping the intent to relevant knowledge graph to extract appropriate best practices and organisational policies. If the intent includes recommended actions as defined in best practices and organisational policies, then BPM module is skipped. To align with project consistencies, the input prompt is then send to Consistency Checker Module.

A. Consistency Checker Module

The purpose of CCM module is to augment developer's prompt with relevant context, extracted from historical prompts and configuration files (containing dependencies between co-related components)

Based on the present working context of the developer, the following steps are processed in order (i) First, the module narrows its search space for historical prompts by extracting appropriate dependencies from configuration files. (ii) Historical prompts are retrieved from the database, based on narrowed search space (iii) Historical prompts are further filtered by leveraging Universal Sentence Encoder [4]. It transforms prompts into embedding vectors, post which pairwise semantic similarity scores are computed and finally, filtered based on a fixed threshold value. (iv) Next, relevant context from historical prompts, as *key-value* pairs, are extracted through a custom algorithm to identify salient keywords (v) Finally, the *key-value* pairs are converted into template-based sentences. These augmentations are then passed to PTM module. The above mentioned steps assume that there exists dependencies(and hence, historical prompts) for the current relevant file. In the event, there are no dependencies(and historical prompts), CCM module will not pass any enhancements to further modules, so an initial set of prompts(in relevant dependencies) used by developers would help CCM yield meaningful responses. Fig 2a highlights the pop-up with consistency improvement to the input prompt based on earlier developed code.

B. Best Practices Module

Shift Lefting for security and best practices is often encouraged while developing software solutions as it minimizes cost and man hours. Our analysis of code completion tools such as Copilot, depicted that, these tools focus on creating executable code, but often perform the bare minimum task. *BPM* ensures the input prompt is holistic and well articulated, by enabling keywords that impel code completion tools to include standards while suggesting code.

Best Practices Module is further composed of three methods (i) Object and Intent Detector (ii) Connected Concept Detector and (iii) Constraint-based Concept Retainer. *Object and Intent Detector* method identifies objects and intent using Named Entity Recognition(NER) and Part-of-Speech(POS) tagging methods. It also identifies relations between two identified objects which are established via one or more intents. Intent refers to the action to be performed on the objects identified by POS tags. These identified intent are then sent to Connected Concept Detector.

Connected Concept Detector method leverages a domain-customized knowledge base to detect ontologically connected concepts for each intent, which in turn maps the intent onto the domain. Document similarity metric is applied to calculate

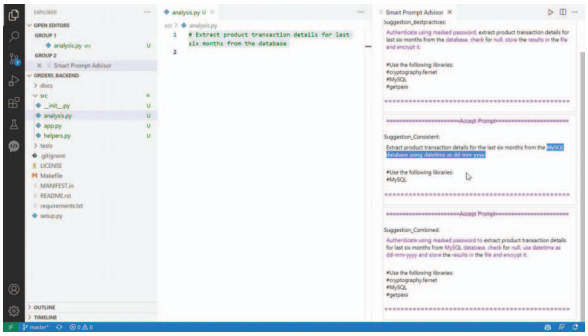


Fig. 3: Smart Prompt Advisor plugin depicting rephrased prompt suggestion

scores between the intents and the domain concepts. The domain-customized knowledge base is populated based on industry standard, organisation policies and technology best practices. These concepts and their corresponding similarity score are then sent to constraint based concept retainer method.

Constraint based concept retainer method identifies positioning of the keywords within the input prompt while retaining the original intent of the prompt. The positioning of the keyword is identified by scoring the keywords with proximity to their related concept, the ones above the threshold are considered. These keywords and their corresponding positioning are sent to Prompt Transformation Module for sentence formation. Fig 2b depicts the popup suggestion for an enhanced input prompt in terms of best practices basis the context of the prompt

C. Prompt Transformation Module

The purpose of *PTM* module is to create a coherent and concise prompt, taking into consideration outputs of *CCM* and *BPM* modules. We leverage OpenAI's $text-davinci-003$ model via API to combine and rephrase the input prompt with augmentations/keywords from CCM and BPM modules.

Prompt Transformation Module consists of two methods i) Concept-to-Action Mapper and ii) Constraint-Enhanced Prompt Generator. *Concept-to-Action Module* method classifies the responses from CCM and BPM modules as prefix and suffix, to be later attached to original prompt. *Constraint-Enhanced Prompt Generator* method leverages a prompt template that combines the original prompt and responses categorized as prefix and suffix into a coherent and enhanced rephrased prompt. We leverage $text-davinci-003$ model to generate the final enhanced prompt.

The reason for leveraging $text-davinci-003$ model is due to its proven ability to transform any given text consistently, as compared to other NLP models. We provide the developers with three versions of the enhanced prompts, based on outputs of intermediary modules (i) CCM only (ii) BPM only (iii) CCM and BPM combined (as shown in Fig. 3).

The enhanced prompt chosen by the developer to generate code via AI-powered code generation tools, is stored in the database(mentioned in Section II-A) for future processing. This repetitive cycle of processing historical prompts to generate new prompts, will enhance the developer's experience of using Copilot like tools.

III. Related Work

Prompt Engineering to become one of the most active areas of research and encompasses wide spectrum of practices and concepts, according to recent literature survey [5]. For Large Code Models(LCM), like OpenAI's Codex, some studies have explored how incorporation of test examples or cases in the prompt influence the overall accuracy of solution [6] [7]. Researchers are working on algorithms to optimize prompts based on whether they generate desired outputs [8]. While there is a growing corpus of studies on optimizing prompts, to the best of our knowledge, none of the studies attempts to tackle the context of our problem statement.

IV. Conclusions and Future Work

In this paper, we have curated a *Smart Prompt Advisor* framework, wherein to enhance input prompt to LLM-based code completion tools, by augmenting keywords based on the prior state of the art of the solution and standards & best practices. While LLMs ensure to make the solution complete, we have found that adding these facts can organize the *chain of thoughts* and thereby guide developers to achieve optimal minimal prompt iterations. In future, we aim to include sustainability best practices as well while performing prompt enhancement to best support developers in code designing.

References

[1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[2] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[3] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.

[4] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.

[5] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[6] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," in *The Eleventh International Conference on Learning Representations*, 2022.

[7] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *arXiv preprint arXiv:2208.05950*, 2022.

[8] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.