Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Fine-tuning and prompt engineering for large language models-based code review automation

Chanathip Pornprasit, Chakkrit Tantithamthavorn *

Monash University, Australia

ARTICLE INFO

Keywords:
Modern code review
Code review automation
Large language models
GPT-3.5
Few-shot learning
Persona

ABSTRACT

Context: The rapid evolution of Large Language Models (LLMs) has sparked significant interest in leveraging their capabilities for automating code review processes. Prior studies often focus on developing LLMs for code review automation, yet require expensive resources, which is infeasible for organizations with limited budgets and resources. Thus, fine-tuning and prompt engineering are the two common approaches to leveraging LLMs for code review automation.
Objective: We aim to investigate the performance of LLMs-based code review automation based on two contexts, i.e., when LLMs are leveraged by fine-tuning and prompting. Fine-tuning involves training the model on a specific code review dataset, while prompting involves providing explicit instructions to guide the model's generation process without requiring a specific code review dataset.
Methods: We leverage model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning and persona) on LLMs-based code review automation. In total, we investigate 12 variations of two LLMs-based code review automation (i.e., GPT-3.5 and Magicoder), and compare them with the Guo et al.'s approach and three existing code review automation approaches (i.e., CodeReviewer, TufanoT5 and D-ACT).
Results: The fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 73.17%–74.23% higher EM than the Guo et al.'s approach. In addition, when GPT-3.5 is not fine-tuned, GPT-3.5 with few-shot learning achieves 46.38%–659.09% higher EM than GPT-3.5 with zero-shot learning.
Conclusions: Based on our results, we recommend that (1) LLMs for code review automation should be fine-tuned to achieve the highest performance.; and (2) when data is not sufficient for model fine-tuning (e.g., a cold-start problem), few-shot learning without a persona should be used for LLMs for code review automation. Our findings contribute valuable insights into the practical recommendations and trade-offs associated with deploying LLMs for code review automation.

1. Introduction

Code review is a software quality assurance practice where developers other than an author (aka. reviewers) review a code change that the author creates to ensure the quality of the code change before being integrated into a codebase. While code review can ensure high software quality, code review is still time-consuming and expensive. Thus, neural machine translation (NMT)-based code review automation approaches were proposed [1–3] to facilitate and expedite the code review process. However, prior studies [4,5] found that such approaches are still not perfect due to limited knowledge of the NMT-based code review automation models that are trained on a small code review dataset.

To address the aforementioned challenge of the NMT-based code review automation approaches, recent work proposed large language model (LLM)-based approaches for the code review automation task [5, 6]. A large language model is a large deep learning model that is based on the transformer architecture [7] and pre-trained on massive textual datasets. An example of the LLM-based code review automation approaches includes CodeReviewer [5], a pre-trained LLM that is based on the CodeT5 [8] model. Li et al. [5] showed that their proposed CodeReviewer outperforms prior NMT-based code review automation approaches [1–3]. However, the training process of CodeReviewer requires a lot of computing resources (i.e., two DGX-2 servers equipped with 32 NVIDIA V100 GPUs in total). Such large computing resources are infeasible for organizations with limited budgets.

Since pre-training LLMs for code review automation can be expensive, fine-tuning and prompt engineering are the two common approaches to leveraging LLMs for code review automation. In particular, fine-tuning involves further training LLMs that are already pre-trained on a specific code review dataset. For example, Lu et al. [9] proposed

* Corresponding author.
E-mail addresses: chanathip.pornprasit@monash.edu (C. Pornprasit), chakkrit@monash.edu (C. Tantithamthavorn).

https://doi.org/10.1016/j.infsof.2024.107523
Received 31 January 2024; Received in revised form 13 June 2024; Accepted 6 July 2024
Available online 11 July 2024

LLaMa-Reviewer, which is the LLM-based code review automation approach that is being fine-tuned on a base LLaMa model [10]. On the other hand, prompting [11–13] involves providing explicit instructions to guide the model's generation process without requiring a specific code review dataset. For instance, Guo et al. [14] conducted an empirical study to investigate the potential of GPT-3.5 for code review automation by using zero-shot learning with GPT-3.5.

While Guo et al. [14] demonstrate the potential of using GPT-3.5 for code review automation, their study still has the following limitations. First, the results of Guo et al. [14] are limited to zero-shot GPT-3.5. However, there are other approaches to leverage GPT-3.5 (i.e., fine-tuning and few-shot learning) that are not included in their study. Thus, it is difficult for practitioners to conclude which approach is the best for leveraging LLMs for code review automation. Second, although prior studies [15–17] found that model fine-tuning can improve the performance of pre-trained LLMs, Guo et al. [14] did not evaluate the performance of LLMs when being fine-tuned. Thus, it is difficult for practitioners to conclude whether LLMs for code review automation should be fine-tuned to achieve the most effective results. Third, Guo et al. [14] did not investigate the impact of few-shot learning, which can improve the performance of LLMs over zero-shot learning [18–20]. Hence, it is difficult for practitioners to conclude which prompting strategy (i.e., zero-shot learning, few-shot learning, and a persona) is the most effective for code review automation.

In this work, we aim to investigate the performance of LLMs-based code review automation based on two contexts, i.e., when LLMs are leveraged by fine-tuning and prompting. In particular, we evaluate two LLMs (i.e., GPT-3.5 and Magicoder [21]) and the existing LLM-based code review automation approaches [4–6] with respect to the following evaluation measures: Exact Match (EM) [1,4] and CodeBLEU [22]. Through the experimental study of the three code review automation datasets (i.e., CodeReviewer$_{\text{data}}$ [5], Tufano$_{\text{data}}$ [6], and D-ACT$_{\text{data}}$ [4]), we answer the following three research questions:

(RQ1) What is the most effective approach to leverage LLMs for code review automation?

Result. The fine-tuning of GPT 3.5 with zero-shot learning achieves 73.17%–74.23% higher EM than the Guo et al.'s approach [14] (i.e., GPT 3-5 without fine-tuning). The results imply that GPT-3.5 should be fine-tuned to achieve the highest performance.

(RQ2) What is the benefit of model fine-tuning on GPT-3.5 for code review automation?

Result. The fine-tuning of GPT 3.5 with zero-shot learning achieves 63.91%–1100% higher Exact Match than those that are not fine-tuned. The results indicate that fine-tuned GPT-3.5 can generate more correct revised code than GPT-3.5 without fine-tuning.

(RQ3) What is the most effective prompting strategy on GPT-3.5 for code review automation?

Result. GPT-3.5 with few-shot learning achieves 46.38%–659.09% higher Exact Match than GPT-3.5 with zero-shot learning. On the other hand, when a persona is included in input prompts, GPT-3.5 achieves 1.02%–54.17% lower Exact Match than when the persona is not included in input prompts. The results indicate that the best prompting strategy when using GPT-3.5 without fine-tuning is using few-shot learning without a persona.

Recommendation. Based on our results, we recommend that (1) LLMs for code review automation should be fine-tuned to achieve the highest performance; and (2) when data is not sufficient for model fine-tuning (e.g., a cold-start problem), few-shot learning without a persona should be used for LLMs for code review automation.

Contributions. In summary, the main contributions of our work are as follows:

- We are the first to investigate the performance of LLMs-based code review automation when using model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning and persona).

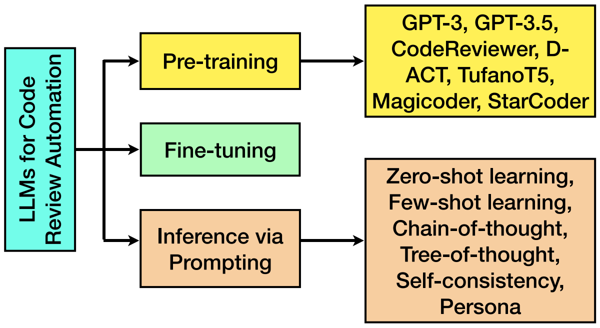

LLMs for Code Review Automation
Pre-training
Fine-tuning
Inference via Prompting
GPT-3, GPT-3.5, CodeReviewer, D-ACT, TufanoT5, Magicoder, StarCoder
Zero-shot learning, Few-shot learning, Chain-of-thought, Tree-of-thought, Self-consistency, Persona

Fig. 1. An overview of the modeling pipeline of LLMs for code review automation.

- We provide recommendations for adopting LLMs for code review automation to practitioners.

Open Science. Our fine-tuned models, script and results are made available online [23].

Paper Organization. Section 2 describes the related work and formulates research questions. Section 3 describes the study design of our study. Section 4 presents the experiment results. Section 5 discusses our experiment results. Section 6 describes possible threats to the validity. Section 7 draws the conclusions of our work.

2. Related work and research questions

In this section, we provide the background knowledge of code review automation, discuss the existing large-language model-based code review automation approaches, and formulate the research questions.

2.1. Code review automation

Code review is a software quality assurance practice where developers other than an author (aka. reviewers) provide feedback for a code change created by the author to ensure that the code change has sufficient quality to meet quality standards. While code review can ensure high software quality, code review is still time-consuming and expensive. Thus, developers still face challenges in receiving timely feedback from reviewers [24,25]. Therefore, code review automation approaches [1,4–6,26] were proposed to facilitate and expedite the code review process.

Code review automation is generally formulated as a sequence generation task, where a language model is trained to learn the relationship between the submitted code and the revised code. Then, during the inference phase, the model aims to generate a revised version of a code change. Recently, neural machine translation (NMT)-based code review automation approaches were proposed [1–3]. Typically, NMT-based code review automation approaches are trained on a specific code review dataset. However, prior studies [4,5] found that NMT-based code review automation approaches can perform well but is still not perfect. This imperfect performance has to do with the limited knowledge of the NMT-based code review automation approaches that are being trained on a small code review dataset.

2.2. LLMs-based code review automation approaches

Large language models (LLMs) for code review automation refer to large language models specifically designed to support code review automation tasks, aiming to understand and generate source code written by developers and natural languages written by reviewers. Since source code and comments often have their own semantic and syntactical structures, recent work proposed various LLMs-based code review automation approaches [5,6,9]. For example, Li et al. [5] proposed CodeReviewer, a pre-trained LLM that is based on the CodeT5 model [8]. Prior studies found that LLMs-based code review automation approaches often outperform NMT-based ones [4–6]. For example, Li et al. [5] found that their proposed approach outperforms a transformer-based NMT model by 11.76%. Below, we briefly discuss the general modeling pipeline of LLMs for code review automation presented in Fig. 1.

Model Pre-Training refers to the initial phase of training a large language model, where the model is exposed to a large amount of unlabeled data to learn general language representations. This phase aims to initialize the model's parameters and learn generic features that can be further fine-tuned for specific downstream tasks. Recently, there have been many large language models for code (i.e., LLMs that are specifically trained on source code and related natural languages). For example, the open-source community-developed large language models such as Code-LLaMa [27], StarCoder [28], and Magicoder [21]; and the commercial large language models such as GPT-3.5.

However, the development of large language models for code requires expensive GPU resources and budget. For example, GPT-3.5 requires 10,000 NVIDIA V-100 GPUs for model pre-training.[1] LLaMa2 [29] requires Meta's Research Super Cluster (RSC) as well as internal production clusters, which consists of approximately 2000 NVIDIA A-100 GPUs in total. Therefore, many software organizations with limited resources and budgets may not be able to develop their large language models. Thus, fine-tuning and prompt engineering are the two common approaches to leverage the existing LLMs for code review automation when expensive GPU resources are not available for pre-training a large language model from scratch, where these techniques are more desirable for many organizations to quickly adopt new technologies.

Model Fine-Tuning is a common practice, particularly in transfer learning scenarios, where a model pre-trained on a large dataset (source domain, e.g., source code understanding) is adapted to a related but different task or dataset (target domain, e.g., code review automation). Recently, researchers have leveraged model fine-tuning techniques for LLMs to improve the performance of code review automation approaches. For example, Lu et al. [9] proposed LLaMa-Reviewer, which is an LLM-based code review automation approach that is being fine-tuned on a base LLaMa model [10] using three code review automation tasks, i.e., a review necessity prediction task to check if diff hunks need a review, a code review comment generation task to generate pertinent comments for a given code snippet, and a code refinement task to generate minor adjustments to the existing code. Lu et al. [9] found that the fine-tuning step on LLMs can greatly improve the performance of the existing code review automation approaches.

Inference refers to the process of using a pre-trained language model to generate source code based on a given natural language prompt instruction. Therefore, prompt engineering plays an important role in leveraging LLMs for code review automation to guide LLMs to generate the desired output. Different prompting strategies have been proposed.[2] For example, zero-shot learning, few-shot learning [18,30,31], chain-of-thought [32,33], tree-of-thought [32,33], self-consistency [34], and persona [13]. Nevertheless, not all prompting strategies are relevant to code review automation. For example, chain-of-thought, self-consistency and tree-of-thought promptings are not applicable to the code review automation task since they are designed for arithmetic and logical reasoning problems. Thus, we exclude them from our study.

In contrast, zero-shot learning, few-shot learning, and persona prompting are the instruction-based prompting strategies, which are more suitable for software engineering (including code review automation) tasks [35–38]. In particular, zero-shot learning involves prompting LLMs to generate an output from a given instruction and an input. On the other hand, few-shot learning [18,30,31] involves prompting LLMs to generate an output from N demonstration examples $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ and an actual input in a testing set, where x_i and y_i are the inputs and outputs obtained from a training set, respectively. Persona [13] involves prompting LLMs to act as a specific role or persona to ensure that LLMs will generate output that is similar to the output generated by a specified persona.

Table 1
The differences between our work and Guo et al.'s work [14].

	Guo et al. [14]	Our work
LLMs/approaches	GPT-3.5, CodeReviewer [5]	GPT-3.5, Magicoder [21], CodeReviewer [5], TufanoT5 [6], D-ACT [4]
Include fine-tuning LLMs?	No	Yes
Prompting techniques	Zero-shot learning, Persona	Zero-shot learning, Few-shot learning, Persona

2.3. GPT-3.5 for code review automation

Recently, Guo et al. [14] conducted an empirical study to investigate the potential of GPT-3.5 for code review automation. However, their study still has the following limitations (see Table 1).

First, the results of Guo et al. [14] are limited to zero-shot GPT-3.5. In particular, Guo et al. [14] conducted experiments to find the best prompt for leveraging zero-shot learning with GPT-3.5. However, there are other approaches to leverage GPT-3.5 (i.e., fine-tuning and few-shot learning) that are not included in their study. The lack of a systematic evaluation of the use of fine-tuning and few-shot learning on GPT-3.5 makes it difficult for practitioners to conclude which approach is the best for leveraging LLMs for code review automation. To address this challenge, we formulate the following research question.

RQ1: What is the most effective approach to leverage LLMs for code review automation?

Second, the performance of LLMs when being fine-tuned is still unknown. In particular, Guo et al. [14] did not evaluate the performance of LLMs when being fine-tuned. However, prior studies [15–17] found that model fine-tuning can improve the performance of pre-trained LLMs. The lack of experiments with model fine-tuning makes it difficult for practitioners to conclude whether LLMs for code review automation should be fine-tuned to achieve the most effective results. To address this challenge, we formulate the following research question.

RQ2: What is the benefit of model fine-tuning on GPT-3.5 for code review automation?

Third, the performance of LLMs for code review automation when using few-shot learning is still unknown. In particular, Guo et al. [14] did not investigate the impact of few-shot learning on LLMs for code review automation. However, recent work [18–20] found that few-shot learning could improve the performance of LLMs over zero-shot learning. The lack of experiments with few-shot learning on LLMs for code review automation makes it difficult for practitioners to conclude which prompting strategy (i.e., zero-shot learning, few-shot learning, and persona) is the most effective for code review automation. To address this challenge, we formulate the following research question.

RQ3: What is the most effective prompting strategy on GPT-3.5 for code review automation?

[1] https://gaming.lenovo.com/emea/threads/17314-The-hardware-behind-ChatGPT
[2] https://www.promptingguide.ai/

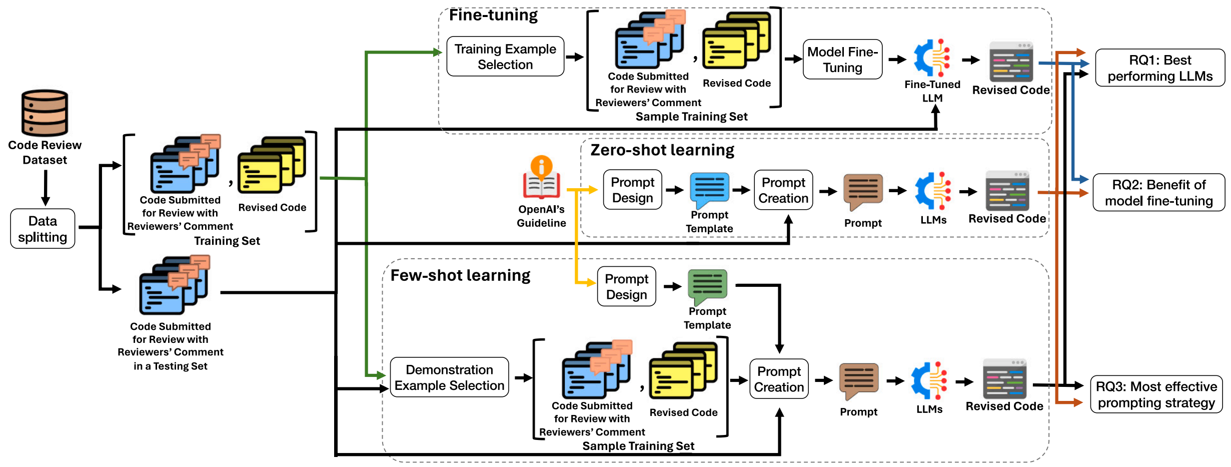

Fine-tuning
Training Example Selection
Code Submitted for Review with Reviewers' Comment
Revised Code
Sample Training Set
Model Fine-Tuning
Fine-Tuned LLM
Revised Code
RQ1: Best performing LLMs
Code Review Dataset
Data splitting
Code Submitted for Review with Reviewers' Comment
Revised Code
Training Set
Zero-shot learning
OpenAI's Guideline
Prompt Design
Prompt Template
Prompt Creation
Prompt
LLMs
Revised Code
RQ2: Benefit of model fine-tuning
Code Submitted for Review with Reviewers' Comment in a Testing Set
Few-shot learning
Prompt Design
Prompt Template
Demonstration Example Selection
Code Submitted for Review with Reviewers' Comment
Revised Code
Sample Training Set
Prompt Creation
Prompt
LLMs
Revised Code
RQ3: Most effective prompting strategy

Fig. 2. An overview of our experimental design (A persona is a part of zero-shot and few-shot learning).

Table 2
Experimental settings in our study. We do not include experimental settings #3 and #4 since LLMs already learn the relationship between input (i.e., code submitted for review) and output (i.e., revised code).

Experimental setting	Fine-Tuning	Inference technique	
		Prompting	Use Persona
#1	✓	Zero-shot	✗
#2			✓
#3		Few-shot	✗
#4			✓
#5	✗	Zero-shot	✗
#6			✓
#7		Few-shot	✗
#8			✓

3. Experimental design

In this section, we provide an overview and details of our experimental design.

3.1. Overview

The goal of this work is to investigate which LLMs perform best when using model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning [18,30,31], and persona [13]). To achieve this goal, we conduct experiments with two LLMs (i.e., GPT-3.5 and Magicoder [21]) on the following datasets that are widely studied in the code review automation literature [4,9,14,39]: CodeReviewer$_{data}$ [5], Tufano$_{data}$ [6] and D-ACT$_{data}$ [4]. We use Magicoder [21] in our experiment since it is further trained on high-quality synthetic instructions and solutions.

In this study, we conduct experiments under six settings as presented in Table 2. According to the table, when the LLMs are fine-tuned, we use zero-shot learning with and without a persona. We do not use few-shot learning with the fine-tuned LLMs since the LLMs already learn the relationship between an input (i.e., code submitted for review) and an output (i.e., improved code). On the other hand, when the LLMs are not fine-tuned, we use zero-shot learning and few-shot learning, where each inference technique is used with and without a persona. Finally, we conduct 36 experiments in total (2 LLMs $\times$ 6 settings $\times$ 3 datasets).

Fig. 2 provides an overview of our experimental design. To begin, the studied code review datasets are split into training and testing sets. The training set consists of the code submitted for review and reviewers' comments as input; and revised code as output. On the other hand, the testing set consists of only code submitted for review and reviewers' comments. Next, to fine-tune the studied LLMs, we first randomly obtain a set of training examples from the training set since using the whole training set is prohibitively expensive. Then, we use the selected training examples to fine-tune the studied LLMs. On the other hand, to use the inference techniques (i.e., zero-shot learning, few-shot learning and a persona), we first design prompt templates for each inference technique based on the guideline from OpenAI.[3,4] However, since few-shot learning requires demonstration examples, we select a set of demonstration examples for each testing sample from the training set. Then, we create prompts that look similar to the prompt templates. Finally, we use the studied LLMs to generate revised code from given prompts. We explain the details of the studied datasets, model fine-tuning, inference via prompting, evaluation measures, and hyper-parameter settings below.

3.2. The studied datasets

Recently, Tufano et al. [1,2] collected datasets with the constraint that revised code must not contain the code tokens (e.g., identifiers) that do not appear in code submitted for review. Thus, such datasets do not align with the real code review practice since developers may add new code tokens when they revise their submitted code. Therefore, in this study, we use the CodeReviewer [5], TufanoT5 [6], and D-ACT [4] datasets, which do not have the above constraint in data collection instead. The details of the studied datasets are as follows (the statistic of the studied datasets is presented in Table 3).

- **CodeReviewer$_{data}$**: Li et al. [5] collected this dataset from the GitHub projects across nine programming languages (i.e., C, C++, C#, Java, Python, Ruby, php, Go, and Javascript). The dataset contains triplets of the code submitted for review (diff hunk granularity), a reviewer's comment, and the revised version of the code submitted for review (diff hunk granularity).
- **Tufano$_{data}$**: Tufano et al. [6] collected this dataset from Java projects in GitHub, and 6388 Java projects hosted in Gerrit. Each record in the dataset contains a triplet of code submitted for review (function granularity), a reviewer's comment, and code after being revised (function granularity). Tufano et al. [6] created two types of this dataset (i.e., Tufano$_{data}$ (with comment) and Tufano$_{data}$ (without comment)).

[3] https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

[4] https://platform.openai.com/docs/guides/prompt-engineering/strategy-write-clear-instructions

Table 3
A statistic of the studied datasets (the dataset of Android, Google and Ovirt are from the D-ACT$_{data}$ dataset [4]).

Dataset	# Train	# Validation	# Test	# Language	Granularity	Has Comment
CodeReviewer$_{data}$ [5]	150,405	13,102	13,104	9	Diff Hunk	✓
Tufano$_{data}$ [6]	134,238	16,779	16,779	1	Function	✓/✗
Android [4]	14,690	1,836	1,835	1	Function	✗
Google [4]	9,899	1,237	1,235	1	Function	✗
Ovirt [4]	21,509	2,686	2,688	1	Function	✗

- **D-ACT$_{data}$**: Pornprasit et al. [4] collected this dataset from the three Java projects hosted on Gerrit (i.e., Android, Google and Ovirt). Each record in the dataset contains a triplet of codebase (function granularity), code of the first version of a patch (function granularity), and code of the approved version of a patch (function granularity).

3.3. Model fine-tuning

To fine-tune the studied LLMs, as suggested by OpenAI, we first select a few training examples to fine-tune an LLM to see if the performance improves. Thus, we randomly select a set of examples from the whole training set by using the `random` function in Python to reduce bias in the data selection. However, there is no existing rule or principle to determine the number of examples that should be selected from a training set. Thus, we use the trial-and-error approach to determine the suitable number of training examples. To do so, we start by using approximately 6% training examples from the whole training set to fine-tune GPT-3.5. We find that GPT-3.5 that is fine-tuned with such training examples outperforms the existing code review automation approaches [4–6]. Therefore, based on the above finding, we use 6% training examples for the whole experiment.

After that, the selected training examples is used to fine-tune the studied LLMs. In particular, we fine-tune GPT-3.5 by using the API provided by OpenAI.[5] On the other hand, to fine-tune Magicoder [21], we leverage the state-of-the-art parameter-efficient fine-tuning technique called DoRA [40].

3.4. Inference via prompting

In this work, we conduct experiments with the following prompting techniques: zero-shot learning, few-shot learning and a persona. We explain each prompting technique below.

For *zero-shot learning*, we first design the prompt template as presented in Fig. 3(a) by following the guidelines from OpenAI[3,4] to ensure that the structure of the prompt is suitable for GPT-3.5. The prompt template consists of the following components: an instruction and an input (i.e., code submitted for review and a reviewer's comment).

Then, we create prompts by using the prompt template in Fig. 3(a) and the code submitted for review with a reviewer's comment in a testing set. Finally, we use the LLMs to generate revised code from the created prompts.

For *few-shot learning* [18,30,31], we first design the prompt template as presented in Fig. 3(b). Similar to zero-shot learning, we follow the guidelines from OpenAI when designing the prompt template. The prompt template consists of the following components: demonstration examples, an instruction and an input (i.e., code submitted for review and a reviewer's comment).

In few-shot learning, demonstration examples are required to create a prompt. Thus, we select three demonstration examples, where each example consists of two inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code), by using BM25 [41]. We use BM25 [41] since prior work [12,42] shows that

(Persona) You are an expert software developer in *<lang>*. You always want to improve your code to have higher quality.

(Instruction) Your task is to improve the given submitted code based on the given reviewer comment. Please only generate the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Persona) You are an expert software developer in *<lang>*. You always want to improve your code to have higher quality. You have to generate an output that follows the given examples.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*. Your task is to improve your submitted code based on the comment that another developer gave you.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Fig. 3. Prompt templates for zero-shot learning and few-shot learning that contain simple instructions (<*lang*> refers to a programming language). The text in blue is omitted when reviewers' comments are not used in the experiments.

BM25 [41] outperforms other sample selection approaches for software engineering tasks. In this work, we use BM25 [41] provided by the `gensim`[6] package. We select three demonstration examples for each testing sample since Gao et al. [11] showed that GPT-3.5 using three demonstration examples achieves comparable performance (i.e, 90% of the highest Exact Match) when compared to GPT-3.5 that achieves the highest performance by using 16 or more demonstration examples.

Then, we create prompts from the prompt template in Fig. 3(b); the code submitted for review and a reviewer's comment in the testing set; and the demonstration examples of the code submitted for review. Finally, we use LLMs to generate revised code from the prompts.

For *persona* [13], we include a persona in the prompt templates in Fig. 3 to instruct GPT-3.5 to act as a software developer. We do so to ensure that the revised code generated by GPT-3.5 looks like the source code written by a software developer.

3.5. The evaluation measures

We use the following measures to evaluate the performance of the studied LLMs (i.e., GPT-3.5 and Magicoder [21]) and code review automation approaches (i.e., CodeReviewer [5], TufanoT5 [6], and D-ACT [4]).

1. **Exact Match (EM)** [4–6] is the number of the generated revised code that is the same as the actual revised code in the testing dataset. We use this measure since it is widely used for evaluating code review automation approaches [1,4,6]. To compare

[5] https://platform.openai.com/docs/guides/fine-tuning/create-a-fine-tuned-model

[6] https://github.com/piskvorky/gensim

Table 4
The evaluation results of GPT-3.5, Magicoder and the existing code review automation approaches.

Approach	Fine-Tuning	Inference technique		CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
		Prompting	Use Persona	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
GPT-3.5	✓	Zero-shot	✗	37.93%	49.00%	22.16%	82.99%	6.02%	79.81%	2.34%	74.15%	6.71%	81.08%	3.05%	74.67%
			✓	37.70%	49.20%	21.98%	83.04%	6.04%	79.76%	2.29%	74.74%	6.14%	81.02%	2.64%	74.95%
	✗		✗	17.72%	44.17%	13.52%	78.36%	2.62%	74.92%	0.49%	61.85%	0.16%	61.04%	0.48%	56.55%
			✓	17.07%	43.11%	12.49%	77.32%	2.29%	73.21%	0.57%	55.88%	0.00%	50.65%	0.22%	45.73%
		Few-shot	✗	26.55%	47.50%	19.79%	81.47%	8.96%	79.21%	2.34%	75.33%	2.89%	81.40%	1.64%	73.83%
			✓	26.28%	47.43%	20.03%	81.61%	9.18%	78.98%	1.62%	74.65%	2.45%	81.07%	1.67%	73.29%
Magicoder	✓	Zero-shot	✗	27.43%	44.86%	11.14%	69.77%	1.97%	69.25%	0.27%	65.39%	0.57%	69.30%	0.30%	64.19%
			✓	27.98%	45.36%	11.06%	69.60%	2.12%	68.84%	0.65%	65.41%	1.13%	69.30%	1.00%	64.16%
	✗		✗	9.75%	39.45%	8.65%	73.90%	0.81%	59.49%	0.16%	47.37%	0.08%	48.82%	0.04%	44.38%
			✓	9.93%	39.48%	8.71%	73.57%	1.51%	67.65%	0.27%	47.46%	0.08%	48.58%	0.11%	43.65%
		Few-shot	✗	15.89%	36.24%	2.93%	4.36%	1.99%	7.49%	0.22%	37.61%	0.49%	42.50%	0.74%	40.33%
			✓	17.80%	38.93%	2.89%	3.70%	1.84%	6.96%	0.27%	16.59%	0.65%	18.83%	0.82%	19.55%
Guo et al. [14]	✗	Zero-shot	✓	21.77%	59.85%	–	–	–	–	–	–	–	–	–	–
CodeReviewer [5]	–	–	–	33.23%	55.43%	15.17%	80.83%	4.14%	78.76%	0.54%	75.24%	0.81%	80.10%	1.23%	75.32%
TufanoT5 [6]				11.90%	43.39%	14.26%	79.48%	5.40%	77.26%	0.27%	75.88%	1.37%	82.25%	0.19%	73.53%
D-ACT [4]				–	–	–	–	–	–	0.65%	75.99%	5.98%	81.85%	1.79%	79.77%

Table 5
The statistical details of GPT-3.5, Magicoder and the existing code review automation approaches.

Model	# parameters
GPT-3.5	175 B
Magicoder [21]	6.7 B
TufanoT5 [6]	60.5 M
CodeReviewer [5]	222.8 M
D-ACT [4]	222.8 M

the generated revised code with the actual revised code, we first tokenize both revised code to sequences of tokens. Then, we compared the sequence of tokens of the generated revised code with the sequence of tokens of the actual revised code. A high value of EM indicates that a model can generate revised code that is the same as the actual revised code in the testing dataset.

2. **CodeBLEU** [22] is the extended version of BLEU (i.e., an n-gram overlap between the translation generated by a deep learning model and the translation in ground truth) [43] for automatic evaluation of the generated code. We do not measure BLEU like in prior work [5,6] since Ren et al. [22] found that this measure ignores syntactic and semantic correctness of the generated code. In addition to BLEU, CodeBLEU considers the weighted n-gram match, matched syntactic information (i.e., abstract syntax tree: AST) and matched semantic information (i.e., data flow: DF) when computing the similarity between the generated revised code and the actual revised code. A high value of CodeBLEU indicates that a model can generate revised code that is syntactically and semantically similar to the actual revised code in the testing dataset.

3.6. The hyper-parameter settings

In this study, we use the following hyper-parameter settings when using GPT-3.5 to generate revised code: temperature of 0.0 (as suggested by Guo et al. [14]), top_p of 1.0 (default value), and max length of 512. To fine-tune GPT-3.5, we use hyper-parameters (e.g., the number of epochs and learning rate) that are automatically provided by OpenAI API.

For Magicoder [21], we use the same hyper-parameter as GPT-3.5 to generate revised code. To fine-tune Magicoder, we use the following hyper-parameters for DoRA [40]: attention dimension (r) of 16, alpha (α) of 8, and dropout of 0.1 .

4. Result

In this section, we present the results of the following three research questions.

(RQ1) What is the most effective approach to leverage LLMs for code review automation?

Approach. To address this RQ, we leverage fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning, and persona) on GPT-3.5 and Magicoder (The details of GPT-3.5 and Magicoder are presented in Table 5) . Then, we measure EM of the results obtained from GPT-3.5, Magicoder and Guo et al.'s approach [14].

Result. The fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 73.17%–74.23% higher EM than the Guo et al. [14]'s approach. Table 4 shows the results of EM achieved by GPT-3.5, Magicoder and Guo et al.'s approach [14]. The table shows that when GPT-3.5 and Magicoder are fine-tuned, such models achieve 73.17%–74.23% and 26.00%–28.53% higher EM than the Guo et al.'s approach [14], respectively.

The results indicate that model fine-tuning could help GPT-3.5 and Magicoder to achieve higher EM when compared to the Guo et al.'s approach [14]. The higher EM has to do with model fine-tuning. When GPT-3.5 or Magicoder is fine-tuned, such models learn the relationship between inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code) from a number of examples in a training set. On the contrary, Guo et al.'s approach [14] only relies on the instruction and given input to generate revised code, which GPT-3.5 never learned during model pre-training.

(RQ2) What is the benefit of model fine-tuning on GPT-3.5 for code review automation?

Approach. To address this RQ, we fine-tune GPT-3.5 as explained in Section 3. Then, we measure EM and CodeBLEU of the results obtained from the fine-tuned GPT-3.5 and the non fine-tuned GPT-3.5 with zero-shot learning.

Result. The fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 63.91%–1100% higher EM than those that are not fine-tuned. Table 4 shows that in terms of EM, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 63.91%–1100% higher than those that are not fine-tuned. In terms of CodeBLEU, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 5.91%–63.9% higher than those that are not fine-tuned.

The results indicate that fine-tuned GPT-3.5 achieve higher EM and CodeBLEU than those that are not fine-tuned. During the model fine-tuning process, GPT-3.5 adapt to the code review automation task by directly learning the relationship between inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code) from a number of examples in a training set. In contrast, non fine-tuned GPT-3.5 is given only an instruction and inputs which are not

```
public static void writeSegmentedCopyRatioPlot(final String sample_name, final
String tnFile, final String preTnFile, final String segFile, final String
outputDir, final Boolean log) {
-   String logArg = “FALSE";
-   if (log) {
-       logArg = “TRUE";
-  }
+  String logArg = log ? "TRUE" : "FALSE";
    final RScriptExecutor executor = new RScriptExecutor();
    executor.addScript(new Resource(R_SCRIPT,
CopyRatioSegmentedPlotter.class));
    executor.addArgs("--args", "--sample_name=" + sample_name, "--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
    executor.exec();
}
```

(a) The difference between code submitted for review and revised code that GPT-3.5 with zero-shot learning (no persona) correctly generates.

```
- public static void writeSegmentedCopyRatioPlot(final String sample_name,
final String tnFile, final String preTnFile, final String segFile, final
String outputDir, final Boolean log) {
+ public static void writeSegmentedCopyRatioPlot(final String sampleName,
final String tnFile, final String preTnFile, final String segFile, final
String outputDir, final boolean log) {
-   String logArg = “FALSE";
-   if (log) {
-       logArg = “TRUE";
-   }
+ String logArg = log ? "TRUE" : "FALSE";
    final RScriptExecutor executor = new RScriptExecutor();
    executor.addScript(new Resource(R_SCRIPT, CopyRatioSegmentedPlotter.
class));
-   executor.addArgs("--args", "--sample_name=" + sampleName, “--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
+   executor.addArgs("--args", "--sample_name=" + sample_name, “--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
```

(b) The difference between code submitted for review and revised code that GPT-3.5 with zero-shot learning (use persona) incorrectly generates.

```
func (e *MessagingEngine) SubmitLocal(event interface{}) {
        e.unit.Launch(func() {
                err := e.process(e.me.NodeID(), event)
-                 if engine.IsInvalidInputError(err) {
+                 if err != nil {
                        e.log.Fatal().Err(err).Str("origin", e.me.NodeID().
String()).Msg("failed to submit local message")
                }
        })
```

(c) The difference between code submitted for review and revised code that GPT-3.5 with few-shot learning (no persona) correctly generates.

```
func(e * MessagingEngine) SubmitLocal(event interface {}) {
    e.unit.Launch(func() {
        err: = e.process(e.me.NodeID(), event)
+       if err != nil {
            if engine.IsInvalidInputError(err) {
                e.log.Fatal().Err(err).Str(""
                    origin "", e.me.NodeID().String()).Msg(""
                    failed to submit local message "")
+           } else {
+               e.log.Fatal().Err(err).Str(“origin", e.me.NodeID().
String()).Msg("unexpected error while processing local message")
+           }
+       }
    })
+ }
```

(d) The difference between code submitted for review and revised code that GPT-3.5 with few-shot learning (use persona) incorrectly generates.

Fig. 4. (RQ3) Examples of the difference between code submitted for review and revised code generated by GPT-3.5 with zero-shot learning and few-shot learning.

```
  } // Show the loading indicator until data has been
fetched.
- if (!totalPagesFromData && totalPagesFromData !== 0) {
+ if (totalPagesFromData === null) {
    return fullPageLoadingIndicator; }
…
```

(a) Example of the code change for *bug fixing* (modify if condition)

```
- protected synchronized void closeLedgerManagerFactory() {
+ protected void closeLedgerManagerFactory() {
    LedgerManagerFactory lmToClose;
    synchronized(this) {
...
```

(b) Example of the code change for *bug fixing* (remove `synchronized` keyword)

```
"userscripts", cmd)
     log.misc.debug("Userscript to run  {}".format(cmd_path))
-    runner.run(cmd, *args, env=env, verbose=verbose).
+    runner.run(cmd_path, *args, env=env, verbose=verbose)
     runner.finished.connect(commandrunner.deleteLater)
     runner.finished.connect(runner.deleteLater)
```

(c) Example of the code change for *refactoring* (change variable name)

```
public EventDefinition(IEventDeclaration declaration, Stream
InputReader streamInputReader) {
-   this.fDeclaration = declaration;
-   this.fStreamInputReader = streamInputReader;
+   fDeclaration = declaration;
+   fStreamInputReader = streamInputReader;
```

(d) Example of the code change for *refactoring* (remove `this` qualifier)

```
...
        return false;
            if (! ServerDB::serverExists(srvnum))
                return false;
-           if (! ServerDB::getConf(srvnum, “autostart",Meta
::mp.bAutoStart).toBool())
-               return false;
            Server *s = new Server(srvnum, this);
            if (! s->bValid) {
                delete s;
...
```

(e) Example of the code change for *other* (remove if condition)

```
public void move() {
...
    if (!newX.equals(spriteBase.getX()) || !
newY.equals(spriteBase.getY())) {
        Logger.log(String.format("Monster moved from (%f,
%f) to (%f, %f)", spriteBase.getX(), spriteBase.getY(),
newX, newY));
    }
-   this.setChanged();
-   this.notifyObservers();
}
```

(f) Example of the code change for *other* (remove method call)

Fig. 5. Example of the code changes of each type.

presented during model pre-training. Therefore, fine-tuned GPT-3.5 can better adapt to the code review automation task than those that are not fine-tuned.

(RQ3) What is the most effective prompting strategy on GPT-3.5 for code review automation?

Approach. To address this RQ, we use zero-shot learning and few-shot learning with *non fine-tuned* GPT-3.5, where each inference technique is used with and without a persona, to generate revised code as explained in Section 3. Then, similar to RQ2, we measure EM and CodeBLEU of the results obtained from GPT-3.5.

Result. GPT-3.5 with few-shot learning achieves 46.38%–659.09% higher EM than GPT-3.5 with zero-shot learning. Table 4 shows that in terms of EM, the use of few-shot learning on GPT-3.5 helps GPT-3.5 to achieve 46.38%–241.98% and 53.95%–659.09% higher than the use of zero-shot learning on GPT-3.5 without a persona and with a persona, respectively. In terms of CodeBLEU, the use of few-shot learning on

GPT-3.5 helps GPT-3.5 to achieve 3.97%–33.36% and 5.55%–60.27% higher than the use of zero-shot learning on GPT-3.5 without a persona and with a persona, respectively.

The results indicate that GPT-3.5 with few-shot learning can achieve higher EM and CodeBLEU than GPT-3.5 with zero-shot learning. When few-shot learning is used to generate revised code from GPT-3.5, GPT-3.5 has more information from given demonstration examples in a prompt to guide the generation of revised code from given code submitted for review and a reviewer's comment. In other words, such demonstration examples could help GPT-3.5 to correctly generate revised code from a given code submitted for review and a reviewer's comment.

When a persona is included in input prompts, GPT-3.5 achieves 1.02%–54.17% lower EM than when the persona is not included in input prompts. Table 4 also shows that when a persona is included in prompts, GPT-3.5 with zero-shot and few-shot learning achieves 3.67%–54.17% and 1.02%–30.77% lower EM compared to when a persona is not included in prompts, respectively. Similarly, when a persona is included in prompts, GPT-3.5 with zero-shot and few-shot learning achieves 1.33%–19.13% and 0.15%–0.90% lower CodeBLEU compared to when a persona is not included in prompts, respectively.

The results indicate that when a persona is included in prompts, GPT-3.5 with zero-shot and few-shot learning achieves lower EM and CodeBLEU. To illustrate the impact of the persona, we present an example of the revised code that GPT-3.5 generates when zero-shot learning with and without a persona is used, and an example of the revised code that GPT-3.5 generates when few-shot learning with and without a persona is used in Fig. 4.

Fig. 4(a) presents the revised code that GPT-3.5 with zero-shot learning (no persona) correctly generates. In this figure, GPT-3.5 suggests an alternative way to initialize variable `logArg`. In contrast, Fig. 4(b) presents the revised code that GPT-3.5 with zero-shot learning (use persona) incorrectly generates. In this figure, GPT-3.5 suggests changing the variable type from `Boolean` to `boolean` and changing the variable name from `sample_name` to `sampleName` in addition to suggesting an alternative way to initialize variable `logArg`.

Fig. 4(c) presents another example of the revised code that GPT-3.5 with few-shot learning (no persona) correctly generates. In this figure, GPT-3.5 suggests a new `if` condition. On the contrary, Fig. 4(d) presents the revised code that GPT-3.5 with few-shot learning (use persona) incorrectly generates. In this figure, GPT-3.5 suggests an additional `if` statement and an additional `else` block.

The above examples imply that when a persona is included in prompts, GPT-3.5 tends to suggest additional incorrect changes to the submitted code compared to when a persona is not included in prompts.

The above results indicate that the best prompting strategy when using GPT-3.5 without fine-tuning is few-shot learning without a persona.

5. Discussion

In this section, we discuss the implications of our findings, the additional results of GPT-3.5, and the cost and benefits of using GPT-3.5.

5.1. Implications of our findings

GPT-3.5 does not require a lot of training data for model fine-tuning to adapt to the code review automation task since Table 4 shows that GPT-3.5 that is fine-tuned on a subset of a training set outperforms the studied code review automation approaches [4–6]. The results imply that GPT-3.5 can adapt to the code review automation task by learning from a small set of training examples (approximately 20k training examples in this study), unlike the studied code review automation approaches that require the whole training set to adapt to the code review automation task.

Recommendations to practitioners. LLMs for code review automation should be fine-tuned to achieve the highest performance. The reason for this recommendation is the results of RQ2 show that fine-tuned GPT-3.5 outperforms those that are not fine-tuned. In contrast, when data is not sufficient for model fine-tuning (e.g., a cold-start problem), few-shot learning without a persona should be used for LLMs for code review automation. The reason for this recommendation is the results of RQ3 show that GPT-3.5 with few-shot learning outperforms GPT-3.5 with zero-shot learning, and GPT-3.5 without a persona outperforms GPT-3.5 with persona.

5.2. The characteristics of the revised code that are correctly generated by GPT-3.5

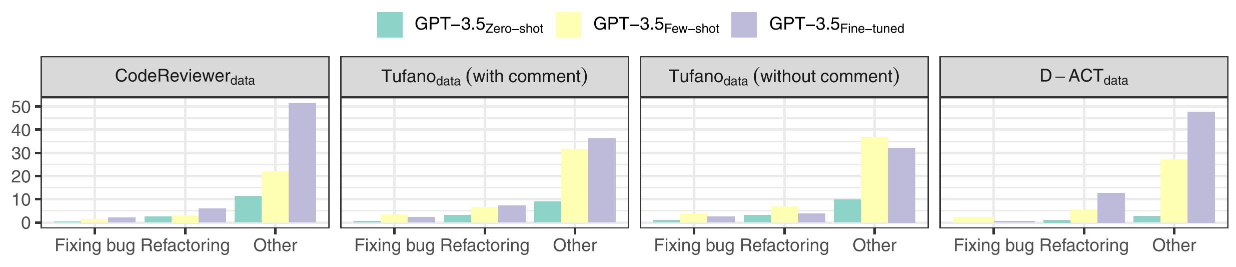
The results of RQ2 and RQ3 demonstrate the benefits of model fine-tuning and few-shot learning on GPT-3.5 for the code review automation task, respectively. However, practitioners still do not clearly understand the characteristics of the code changes of the revised code that GPT-3.5 correctly generates. To address this challenge, we aim to qualitatively investigate the revised code that GPT-3.5 can correctly generate. To do so, we randomly select the revised code that is only correctly generated by a particular model (e.g., we randomly obtain the revised code that only fine-tuned GPT-3.5 correctly generates while the others do not.) by using a confidence level of 95% and a confidence interval of 5%. Then, we classify the code changes of the selected revised code into the following categories based on the taxonomy of code change created by Tufano et al. [1] (the examples of the code changes in each category are depicted in Fig. 5):

- *fixing bug*: The code changes in this category involve fixing bugs in the past. The following sub-categories are related to this category: exception handling, conditional statement, lock mechanism, method return value, and method invocation.
- *refactoring*: The code changes in this category involve making changes to code structure without changing the behavior of the changed code. The following sub-categories are related to this category: inheritance; encapsulation; methods interaction; readability; and renaming parameter, method and variable.
- *other*: The code changes that cannot be classified as neither *fixing bug* nor *refactoring* will fall into this category.

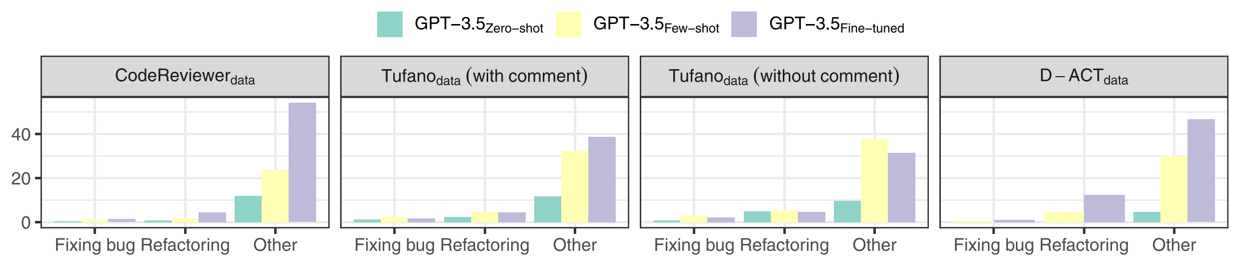
Fig. 6 presents the characteristics of the code changes (i.e., *Fixing Bug*, *Refactoring* and *Other*) of the revised code that GPT-3.5 correctly generates. According to the figure, for the Tufano$_{\text{data}}$ (with comment), CodeReviewer$_{\text{data}}$ and D-ACT$_{\text{data}}$ dataset, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 achieves the highest EM for the code changes of *Refactoring* and *Other*. In contrast, for the Tufano$_{\text{data}}$ (without comment) dataset, non fine-tuned GPT-3.5 with few-shot learning achieves the highest EM for the code changes of all categories. According to Figs. 6(a) and 6(b), we also find that fine-tuned GPT-3.5 and non fine-tuned GPT-3.5 that zero-shot and few-shot learning are used achieve the highest EM for the code changes of type *other* across all studied datasets. The reason for this result is that we do not specify the characteristics of the revised code (i.e., *fixing bug* and *refactoring*) in prompts. Thus, such models possibly generate revised code that is not specific to *fixing bug* or *refactoring*. Therefore, the majority of the code changes of the generated revised code are categorized as *other*.

5.3. The impact of the size of training dataset on fine-tuned GPT-3.5

The results of RQ2 show that model fine-tuning can help increase the performance of GPT-3.5. However, little is known whether fine-tuned GPT-3.5 can achieve higher performance when being fine-tuned with larger training sets. Therefore, we conduct experiments by using



(a) The results of GPT-3.5 (persona is included in prompts)



(b) The results of GPT-3.5 (persona is not included in prompts)

Fig. 6. The EM achieved by GPT-3.5$_{\text{Zero-shot}}$, GPT-3.5$_{\text{Few-shot}}$ and GPT-3.5$_{\text{Fine-tuned}}$ categorized by the types of code change. Here, GPT-3.5$_{\text{Zero-shot}}$ and GPT-3.5$_{\text{Few-shot}}$ refer to non fine-tuned GPT-3.5 with zero-shot learning and few-shot learning, respectively. On the other hand, GPT-3.5$_{\text{Fine-tuned}}$ refers to fine-tuned GPT-3.5 with zero-shot learning.

Table 6
The evaluation results of GPT-3.5 when being fine-tuned with different sizes of training sets.

Size of training set	CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
6%	37.93%	49.00%	22.16%	82.99%	**6.02%**	79.81%	2.34%	74.15%	6.71%	81.08%	**3.05%**	74.67%
10%	37.72%	48.83%	22.31%	83.43%	5.37%	**80.68%**	**2.51%**	75.10%	5.98%	80.65%	2.71%	75.13%
20%	**38.80%**	**49.33%**	**22.84%**	**83.44%**	5.65%	80.42%	2.34%	**76.04%**	**7.52%**	**81.40%**	2.83%	**75.46%**

Table 7
The evaluation results of GPT-3.5 for different prompt templates. P1 refers to the prompt templates with simple instructions (Fig. 3). P2 refers to the prompt templates with instructions being broken down into smaller steps.(Fig. 7). P3 refers to the prompt templates with detailed instructions (Fig. 8).

Prompt design	Prompting	CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
		EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
P1	Zero-shot	17.72%	44.17%	13.52%	78.36%	2.62%	74.92%	0.49%	61.85%	0.16%	61.04%	0.48%	56.55%
P2		14.47%	43.52%	11.24%	79.05%	2.25%	76.54%	0.54%	66.10%	0.16%	67.07%	0.33%	60.76%
P3		11.94%	41.18%	9.86%	76.18%	1.26%	72.31%	0.05%	53.03%	0.08%	46.22%	0.26%	41.20%
P1	Few-shot	26.55%	47.50%	19.79%	81.47%	8.96%	79.21%	2.34%	75.33%	2.89%	81.40%	1.64%	73.83%
P2		25.25%	48.45%	15.82%	80.16%	6.84%	76.50%	0.60%	75.94%	3.56%	81.40%	1.67%	74.18%
P3		25.14%	48.60%	15.07%	79.83%	5.81%	75.76%	0.38%	74.95%	2.91%	81.18%	1.49%	73.31%

10% and 20% of training sets to fine-tune GPT-3.5 (we do not use persona in these experiments).

Table 6 shows the results of EM and CodeBLEU that fine-tuned GPT-3.5 achieves across different sizes of training sets. The table shows that GPT-3.5 that is fine-tuned with 20% of a training set achieves 2.29%–12.07% higher EM and 0.54%–2.55% higher CodeBLEU than GPT-3.5 that is fine-tuned with 6% of a training set. In addition, GPT-3.5 that is fine-tuned with 20% of a training set achieves 2.38%–25.75% higher EM and 0.01%–1.25% higher CodeBLEU than GPT-3.5 that is fine-tuned with 10% of a training set, respectively. The results indicate that fine-tuned GPT-3.5 achieves higher performance when being fine-tuned with a larger training set.

5.4. The impact of prompt design on GPT-3.5

In RQ3, we use the prompt templates in Fig. 3 that contain simple instructions to conduct experiments. However, prior work [44,45] found that the design of prompts has an impact on the performance of LLMs. Thus, we further investigate the impact of prompt design on GPT-3.5 for code review automation. To do so, we conduct experiments by using the following two new prompt designs (we do not include a persona in these prompt designs since the results of RQ3 show that GPT-3.5 without a persona outperforms GPT-3.5 with a persona). First, we use the prompt design that a single instruction is broken into smaller steps,[7] as depicted in Fig. 7. Second, we use the prompt design that contains more detailed instructions,[8] as depicted in Fig. 8.

Table 7 shows the results of EM and CodeBLEU that GPT-3.5 achieves across different prompt designs. The table shows that for zero-shot learning, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 16.44%–45.45% higher EM than GPT-3.5 that is prompted by the prompt with an instruction being broken down into smaller steps. In addition, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 37.12%–880.00% higher EM than GPT-3.5 that is prompted by the prompt with a detailed instruction.

The table also shows that for few-shot learning, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 5.15%–290.00% higher EM than GPT-3.5 that is prompted by the prompt

7 https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions#break-the-task-down

8 https://www.promptingguide.ai/introduction/tips

(Instruction) Follow the steps below to improve the given submitted code
step 1 - read the given submitted code and a reviewer comment
step 2 - identify lines that need to be modified, added or deleted
step 3 - generate the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*.

Follow the steps below to improve the given submitted code
step 1 - read the given submitted code and a reviewer comment in the above examples
step 2 - identify lines that need to be modified, added or deleted in the examples
step 3 - read the given submitted code and a reviewer comment
step 4 - identify lines that need to be modified, added or deleted
step 5 - generate the improved code without your explanation.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Fig. 7. Prompt templates for zero-shot learning and few-shot learning that the instructions are broken into smaller steps (<*lang*> refers to a programming language). The text in blue is omitted when reviewers' comments are not used in the experiments.

(Instruction) A developer asks you to help him improve his submitted code based on the given reviewer comment. He emphasizes that the improved code must have higher quality, conforms to coding convention or standard, and works correctly. He tells you to refrain from putting the submitted code in a class or method, and providing global variables or an implementation of methods that appear in the submitted code. He asks you to recommend the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*.

A developer asks you to help him improve his submitted code based on the given reviewer comment. He emphasizes that the improved code must have higher quality, conforms to coding convention or standard, and works correctly. He tells you to refrain from putting the submitted code in a class or method, and providing global variables or an implementation of methods that appear in the submitted code. He asks you to recommend the improved code without your explanation.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Fig. 8. Prompt templates for zero-shot learning and few-shot learning that more details are added to the instructions (<*lang*> refers to a programming language). The text in blue is omitted when reviewers' comments are not used in the experiments.

with an instruction being broken down into smaller steps. Furthermore, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 5.61%–515.79% higher EM than GPT-3.5 that is prompted by the prompt with detailed instruction.

The results indicate that GPT-3.5 that is prompted by the prompts with a simple instruction achieves the highest EM when compared to other prompt designs. Thus, the results imply that the prompt with a simple instruction is the most suitable for GPT-3.5 for code review automation.

5.5. *Cost and benefits of using GPT-3.5 for code review automation*

Cost is one of the factors that determine the choices of AI services for practitioners. In the case of GPT-3.5 provided by OpenAI, the cost of using GPT-3.5 varies depending on usage.[9] In particular, the cost of using zero-shot learning and few-shot learning with GPT-3.5 is approximately 0.002 USD per query (for 1k input tokens and 1k output tokens) and 0.0035 USD per query (for 4k input tokens and 1k output tokens), respectively. On the other hand, the cost for fine-tuning one GPT-3.5 is approximately 40 USD (we use approximately 8k examples from a training set), and the cost for using fine-tuned GPT-3.5 is approximately 0.009 USD per query (for 1k input tokens and 1k output tokens).

Assume that a software developer uses GPT-3.5 to help him/her review code submitted for review 1000 times per day on average and he/she works for 25 days per month, the total GPT-3.5 usage is $1000 \times 25 \times 12 = 300{,}000$ times per year. Such GPT-3.5 usage accounts for \$600 per year (300,000 × 0.002) when using zero-shot learning with GPT-3.5, \$1,050 per year (300,000 × 0.0035) when using few-shot learning with GPT-3.5, and \$2740 per year (40 + (300,000 × 0.009)) when using fine-tuned GPT-3.5. However, when compared to the average yearly salary of software engineers around the world,[10] the usage cost of GPT-3.5 is approximately 62.23%–91.73% and 97.51%–99.46% less than the lowest average salary (i.e., \$7255 in Nigeria) and the highest average salary (i.e., \$110,140 in the United States), respectively. Nevertheless, the results of RQ1 show that fine-tuned GPT-3.5 achieves the highest EM. In addition, the results of RQ3 show that the use of few-shot learning without persona on GPT-3.5 helps GPT-3.5 achieve the highest EM and CodeBLEU. Thus, we recommend that GPT-3.5 for code review automation should be fine-tuned. Otherwise, leveraging GPT-3.5 by using few-shot learning without persona can be considered as an alternative.

6. Threats to validity

We describe the threats to the validity of our study below.

6.1. *Threats to construct validity*

Threats to construct validity relate to the example selection techniques that we use to select examples for few-shot learning, and the design choices of the persona. We explain each threat below.

In this study, we only use the BM25 [41] technique to select three demonstration examples for few-shot learning. However, using more demonstration examples or different techniques to select demonstration examples may lead to results that differ from the reported results. Thus, more demonstration examples and other techniques for selecting demonstration examples can be explored in future work.

Since the code review automation task is related to revising the patches written by software developers, we use the persona (i.e., *an expert software developer*) to ensure that the revised code generated by GPT-3.5 looks like the source code written by a software developer. However, there are other similar personas (e.g., a senior software engineer, or a front-end software developer) that we do not explore. Thus, future work can explore other design choices of prompt and persona to find the optimal prompt and persona for code review automation tasks.

[9] https://openai.com/pricing
[10] https://codesubmit.io/blog/software-engineer-salary-by-country/

6.2. Threats to internal validity

Threats to internal validity relate to the randomness of GPT-3.5 and Magicoder, and the hyper-parameter settings that we use to fine-tune GPT-3.5 and Magicoder. The results that we obtain from GPT-3.5 and Magicoder may vary due to the randomness of GPT-3.5 and Magicoder. However, doing the same experiments multiple rounds can be expensive due to large testing datasets. Finally, we do not explore all possible combinations of hyper-parameter settings (e.g., the number of epoch or learning rate) when fine-tuning GPT-3.5 and Magicoder. We do not do so since the search space of hyper-parameter settings is large, which can be expensive. Nonetheless, the main goal of this study is not to find the best hyper-parameter settings for code review automation, but to investigate the performance of GPT-3.5 and Magicoder on code review automation tasks when using model fine-tuning.

6.3. Threats to external validity

Threats to external validity relate to the generalizability of our findings in other software projects. In this study, we conduct the experiment with the dataset obtained from recent work [4–6]. However, the results of our experiment may not be generalized to other software projects. Thus, other software projects can be explored in future work.

Another threat relates to the updates to GPT-3.5 made by OpenAI in future. Due to the updates, reproduced experiment results may differ from those reported in this paper.

7. Conclusion

In this work, we investigate the performance of LLMs (i.e., GPT-3.5 and Magicoder) for code review automation when using model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning, and persona). We also compare the performance of the LLMs with the existing code review automation approaches [4–6]. Our results show that (1) fine-tuned GPT-3.5 performs best for code review automation and (2) the best prompting strategy when using GPT-3.5 without fine-tuning is few-shot learning without a persona. Based on the results, we recommend that (1) LLMs for code review automation should be fine-tuned to achieve the highest performance; and (2) when data is not sufficient for model fine-tuning, few-shot learning without a persona should be used for LLMs for code review automation.

CRediT authorship contribution statement

Chanathip Pornprasit: Writing – review & editing, Writing – original draft, Methodology, Data curation, Conceptualization. **Chakkrit Tantithamthavorn:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

Chakkrit Tantithamthavorn was supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941).

References

[1] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, Denys Poshyvanyk, On learning meaningful code changes via neural machine translation, in: Proceedings of ICSE, 2019, pp. 25–36.
[2] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, Gabriele Bavota, Towards automating code review activities, in: Proceedings of ICSE, 2021, pp. 163–174.
[3] Patanamon Thongtanunam, Chanathip Pornprasit, Chakkrit Tantithamthavorn, AutoTransform: Automated code transformation to support modern code review process, in: Proceedings of ICSE, 2022, pp. 237–248.
[4] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, Chunyang Chen, D-ACT: Towards diff-aware code transformation for code review under a time-wise evaluation, in: Proceedings of SANER, 2023, pp. 296–307.
[5] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al., Automating code review activities by large-scale pre-training, in: Proceedings of ESEC/FSE, 2022, pp. 1035–1047.
[6] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, Gabriele Bavota, Using pre-trained models to boost code review automation, in: Proceedings of ICSE, 2022, pp. 2291–2302.
[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, Attention is all you need, in: Proceedings of NIPS, 2017, pp. 5999–6009.
[8] Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of EMNLP, 2021, pp. 8696–8708.
[9] J. Lu, L. Yu, X. Li, L. Yang, C. Zuo, Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning, in: Proceedings of ISSRE, 2023, pp. 647–658.
[10] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al., Llama: Open and efficient foundation language models, 2023, arXiv preprint arXiv:2302.13971.
[11] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, Michael R. Lyu, What makes good in-context demonstrations for code intelligence tasks with llms? in: Proceedings of ASE, 2023, pp. 761–773.
[12] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Michael R. Lyu, Constructing effective in-context demonstration for code intelligence tasks: An empirical study, in: Proceedings of ASE, 2023.
[13] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, Douglas C. Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023, arXiv preprint arXiv:2302.11382.
[14] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, Xin Peng, Exploring the potential of ChatGPT in automated code refinement: An empirical study, 2023, arXiv preprint arXiv:2309.08221.
[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al., Palm: Scaling language modeling with pathways, J. Mach. Learn. Res. (2023) 1–113.
[16] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, Quoc V. Le, Finetuned language models are zero-shot learners, 2021, arXiv preprint arXiv:2109.01652.
[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.
[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al., Language models are few-shot learners, Proc. NeurIPS (2020) 1877–1901.
[19] Sungmin Kang, Juyeon Yoon, Shin Yoo, Large language models are few-shot testers: Exploring llm-based general bug reproduction, in: Proceedings of ICSE, 2023, pp. 2312–2323.
[20] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, Xiangke Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning.
[21] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, Lingming Zhang, Magicoder: Source code is all you need, 2023, arXiv preprint arXiv:2312.02120.
[22] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, Shuai Ma, Codebleu: a method for automatic evaluation of code synthesis, 2020, arXiv preprint arXiv:2009.10297.
[23] Supplementary material, https://github.com/awsm-research/LLM-for-code-review-automatiton.
[24] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, Jacek Czerwonka, Code reviewing in the trenches: Challenges and best practices, IEEE Softw. (2017) 34–42.
[25] Peter C. Rigby, Christian Bird, Convergent Contemporary Software Peer Review Practices, in: Proceedings of ESEC/FSE, 2013, pp. 202–212.

[26] Rosalia Tufan, Luca Pascarella, Michele Tufanoy, Denys Poshyvanykz, Gabriele Bavota, Towards automating code review activities, in: Proceedings of ICSE, 2021, pp. 1479–1482.
[27] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al., Code llama: Open foundation models for code, 2023, arXiv preprint arXiv:2308.12950.
[28] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al., Starcoder: may the source be with you!, 2023, arXiv preprint arXiv:2305.06161.
[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al., Llama 2: Open foundation and fine-tuned chat models, 2023, arXiv preprint arXiv:2307.09288.
[30] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, Weizhu Chen, What makes good in-context examples for GPT-3? 2021, arXiv preprint arXiv:2101.06804.
[31] Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, et al., Larger language models do in-context learning differently, 2023, arXiv preprint arXiv:2303.03846.
[32] Seungone Kim, Se June Joo, Doyoung Kim, Joel Jang, Seonghyeon Ye, Jamin Shin, Minjoon Seo, The cot collection: Improving zero-shot and few-shot learning of language models via chain-of-thought fine-tuning, 2023, arXiv preprint arXiv:2305.14045.
[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Proc. NeurIPS (2022) 24824–24837.
[34] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, Denny Zhou, Self-consistency improves chain of thought reasoning in language models, 2022, arXiv preprint arXiv:2203.11171.
[35] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, Pinjia He, Prompting for automatic log template extraction, 2023, arXiv preprint arXiv:2307.09950.
[36] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, Danny Dig, Unprecedented code change automation: The fusion of LLMs and transformation by example, 2024, arXiv preprint arXiv:2402.07138.
[37] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, James Noble, Towards AI-assisted synthesis of verified dafny methods, 2024, arXiv preprint arXiv:2402.00247.
[38] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, Michael R. Lyu, Llmparser: A llm-based log parsing framework, 2023, arXiv preprint arXiv:2310.01796.
[39] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, David Lo, Generation-based code review automation: How far are we? 2023, arXiv preprint arXiv:2303.07221.
[40] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Min-Hung Chen, DoRA: Weight-decomposed low-rank adaptation, 2024, arXiv preprint arXiv:2402.09353.
[41] Stephen Robertson, Hugo Zaragoza, et al., The probabilistic relevance framework: BM25 and beyond, Found. Trends Inf. Retriev. (2009) 333–389.
[42] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, Yiling Lou, Evaluating instruction-tuned large language models on code comprehension and generation, 2023, arXiv preprint arXiv:2308.01240.
[43] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu, BLEU: A method for automatic evaluation of machine translation, in: Proceedings of ACL, 2002, pp. 311–318.
[44] David OBrien, Sumon Biswas, Sayem Mohammad Imtiaz, Rabe Abdalkareem, Emad Shihab, Hridesh Rajan, Are prompt engineering and TODO comments friends or foes? An evaluation on GitHub copilot, in: Proceedings of ICSE, 2024, pp. 1–13.
[45] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, Omer Tripp, A deep dive into large language models for automated bug localization and repair, 2024, arXiv preprint arXiv:2404.11595.