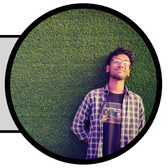
# Prompting LLMs for Software Development Automation


**Dr. Antonio Mastropaolo**

*Instructor*

**Mr. Alvi Haque**



*Teaching Assistant*

WILLIAM & MARY

CHARTERED 1693


**Spring 2026**

# Prompting LLMs For Software Development Automation

Prompting is a technique where a deep learning model is guided to perform a task by embedding input-output examples directly within the input prompt. These examples, also known as “contextual cues,” help the model infer the task’s structure and generate appropriate outputs.

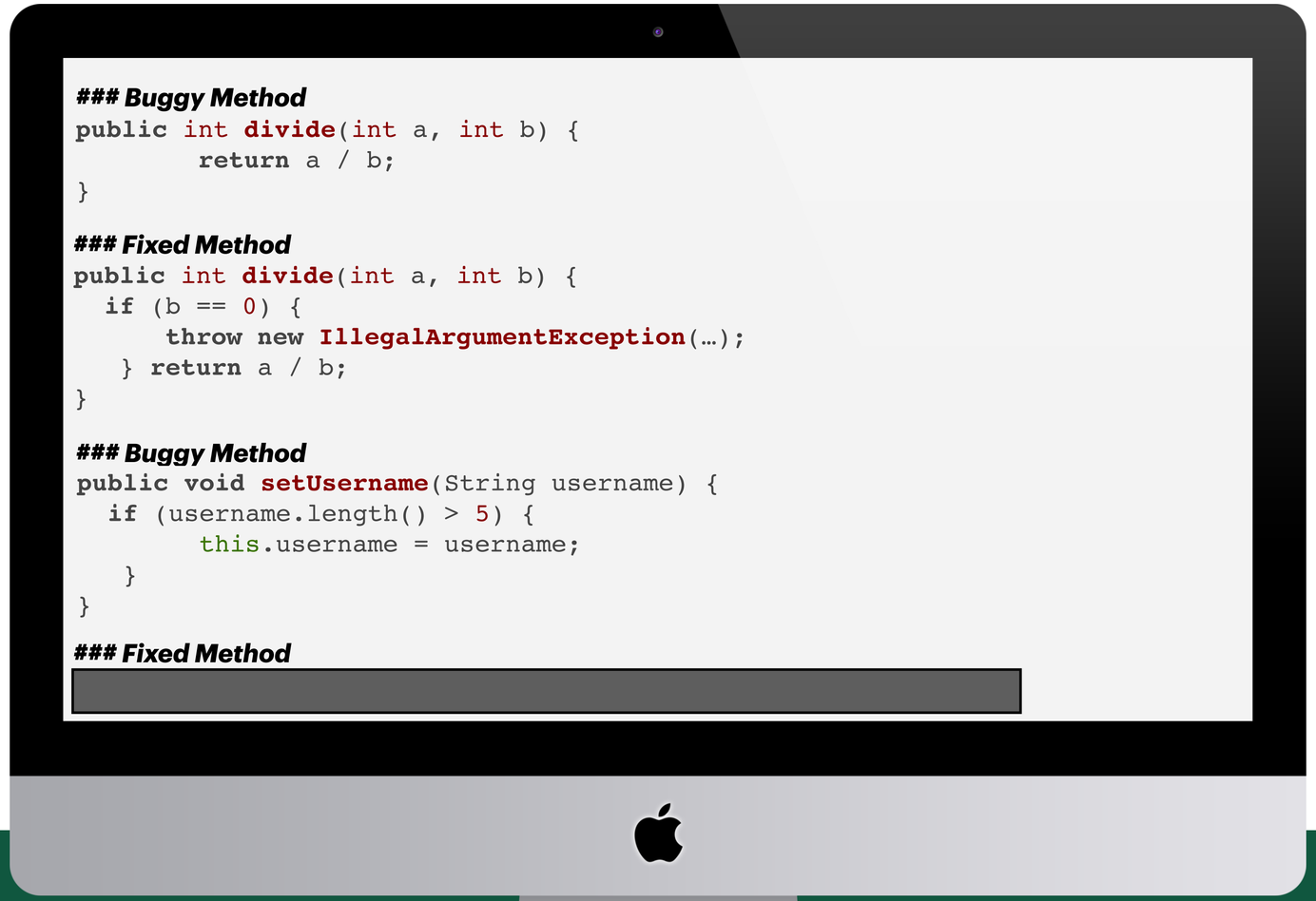
# Prompting LLMs For Software Development Automation

KEY IDEA:

Instead of fine-tuning a pre-trained model for a specific task, we provide a set of demonstrations (examples of inputs and outputs) and possibly a description of the task in the same prompt.
The model if large enough – then generalizes "on-the-fly" to perform the given task.

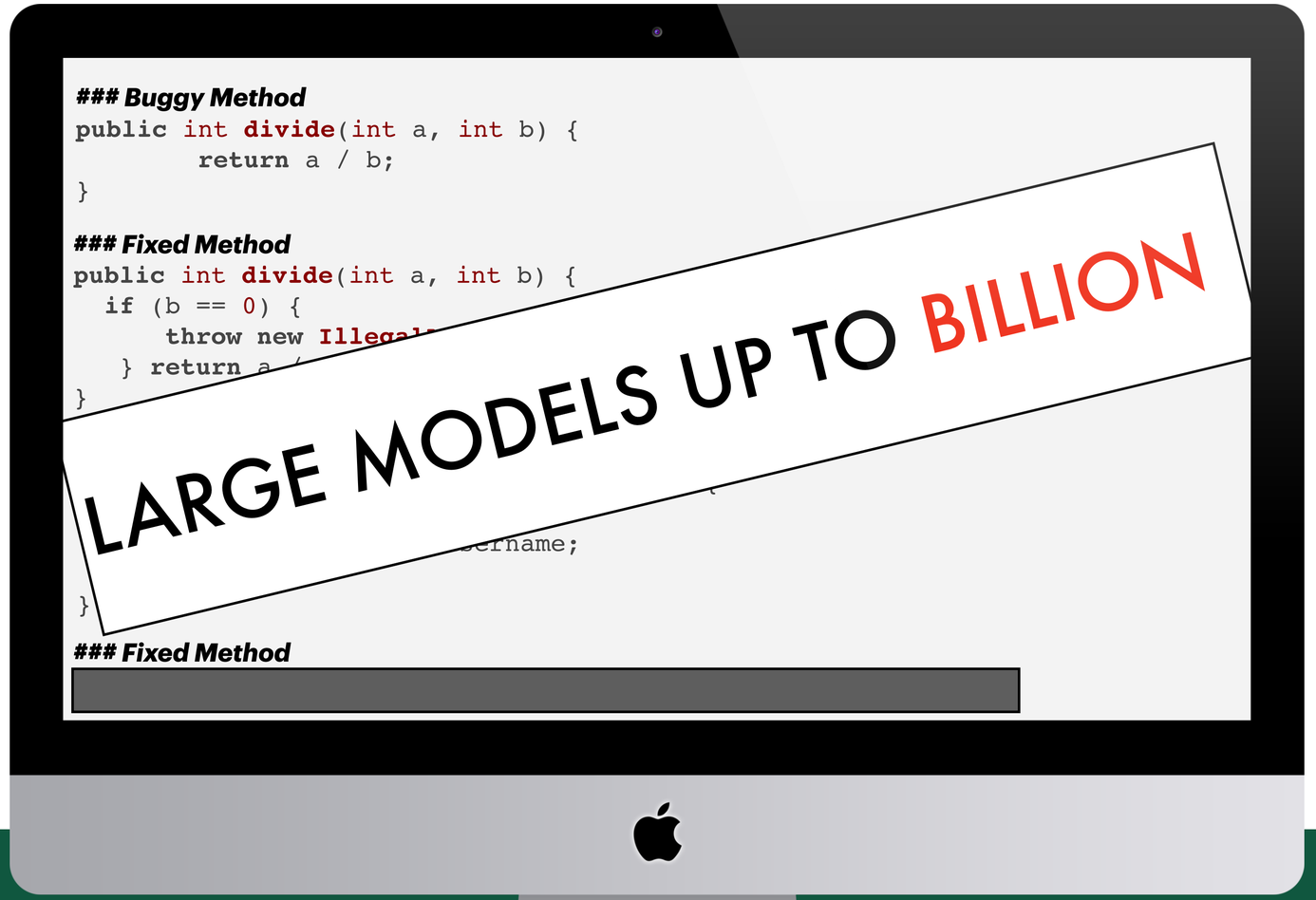
# Prompting LLMs For Software Development Automation

BUG-FIXING



# Prompting LLMs For Software Development Automation

BUG-FIXING



# Prompting LLMs For Software Development Automation

## Prompting vs Pretrain – then – Finetune

- *No model's parameters update —No backprop*
- *Model's parameters adjustment via fine-tuning*

# Prompting LLMs For Software Development Automation

## Prompting vs Pretrain – then – Finetune

- *No model's parameters update —No backprop*
- *Model's parameters adjustment via fine-tuning*

- *The model adapts its behavior on-the-fly by being exposed to "just a few" examples and a potential task description i.e. prompt*
- *The model adapt its behavior after several epochs*

# Prompting LLMs For Software Development Automation

## Prompting vs Pretrain – then – Finetune

- *No model's parameters update —No backprop*
- *Model's parameters adjustment via fine-tuning*

- *The model adapts its behavior on-the-fly by being exposed to "just a few" examples and a potential task description i.e. prompt*
- *The model adapt its behavior after several epochs*

Few-Shot Learning

# Prompting LLMs For Software Development Automation

## Prompting vs Pretrain – then – Finetune

- *No model's parameters update —No backprop*
- *Model's parameters adjustment via fine-tuning*

- *The model adapts its behavior on-the-fly by being exposed to "just a few" examples and a potential task description i.e. prompt*
- *The model adapt its behavior after several epochs*

- *Multi-task by design*
- *The model needs proper adjustment e.g., task-priming*

# Prompting LLMs For Development Automation

**PROS**

## ICL vs Pretrain – then – Finetune

- *No model's parameters update —No backprop*
- *Model's parameters adjustment via fine-tuning*

- *The model adapts its behavior on-the-fly by being exposed to "just a few" examples and a potential task description i.e. prompt*
- *The model adapt its behavior after several epochs*

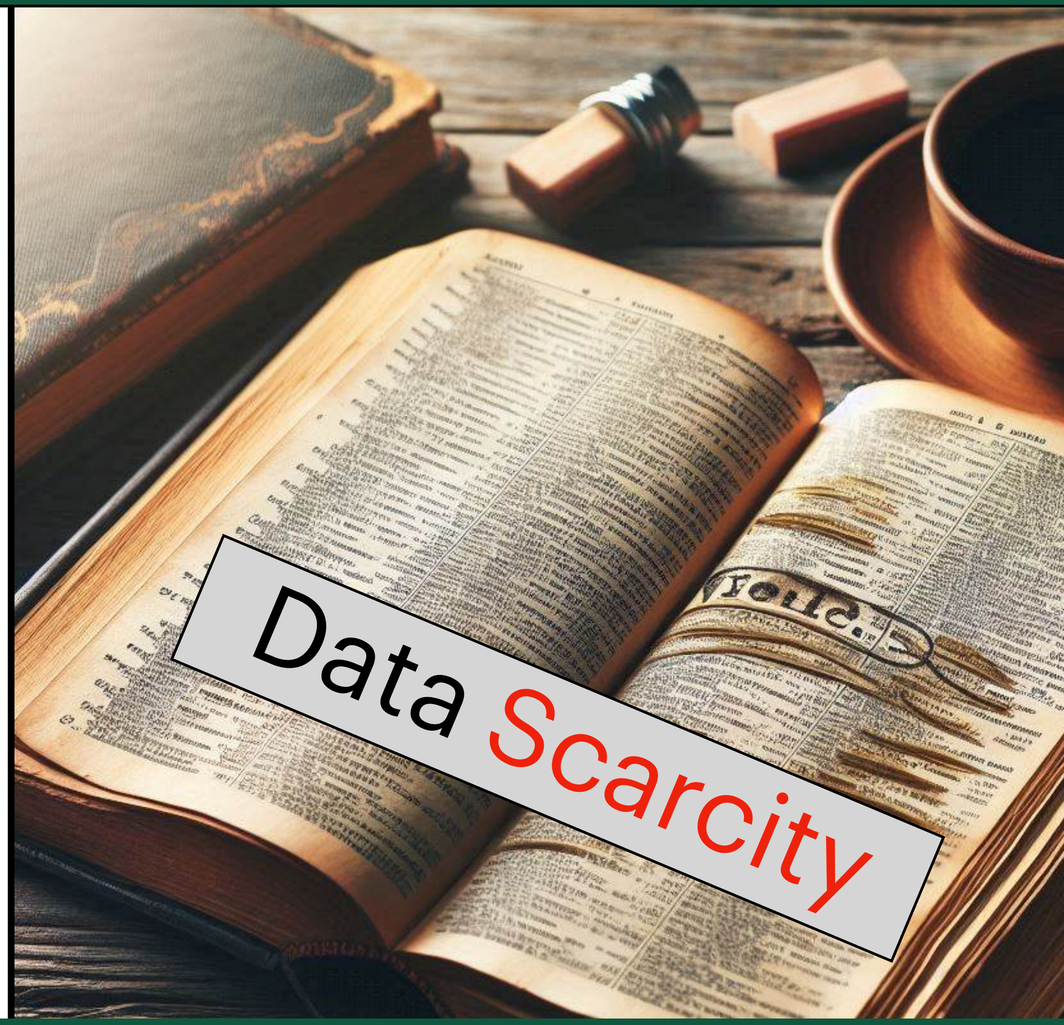
- *Multi-task by design*
- *The model needs proper adjustment e.g., task-pri*

**CONS**

# Prompting LLMs For Software Development Automation

**PROS**

Prompting methods are technically the most effective solution to cope with the lack of data – that is otherwise required in an approach rooted in fine-tuning



# Prompting LLMs For Software Development Automation

**PROS**

Prompting methods are usually paired with Large Language Models, as they are designed to extract information from very large models, up to a **Trillion** parameters, e.g., GPT-4

# Prompting LLMs For Software Development Automation

## PROS

Prompting methods are usually paired with Large Language Models, as they are designed to extract information from very large models, up to a **Trillion** parameters, e.g., GPT-4

AI Democratization

# Prompting LLMs For Software Development Automation

**PROS**

New state-of-the-art performances on various applications, with Software Engineering and Development being no exception.

# Prompting LLMs For Software Development Automation

## CONS

To achieve state-of-the-art results it is required the usage of —usually—**Billion parameters** model. As you can imagine, managing such models is extremely expensive.

# Prompting for Code Summarization

# Prompting LLMs For Software Development Automation

## Few-shot training LLMs for project-specific code-summarization

Toufique Ahmed
University of California, Davis
Davis, California, USA
tfahmed@ucdavis.edu

Premkumar Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu@ucdavis.edu

**ABSTRACT**

Very large language models (LLMs), such as GPT-3 and Codex have achieved state-of-the-art performance on several natural-language tasks, and show great promise also for code. A particularly exciting aspect of LLMs is their knack for few-shot and zero-shot learning: they can learn to perform a task with very few examples. Few-shotting has particular synergies in software engineering, where there are a lot of phenomena (identifier names, APIs, terminology, coding patterns) that are known to be *highly* project-specific. However, project-specific data can be quite limited, especially early in the history of a project; thus the few-shot learning capacity of LLMs might be very relevant. In this paper, we investigate the use few-shot training with the very large GPT (Generative Pre-trained Transformer) Codex model, and find evidence suggesting that one can significantly surpass state-of-the-art models for code-summarization, leveraging project-specific training.

**KEYWORDS**

deep learning, code summarization, large language model

**ACM Reference Format:**
Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559555

### 1 INTRODUCTION

Very large language models (LLMs) are viewed as a revolutionary advance in natural language processing. Models such as GPT-3 [4], which have over 150 billion parameters, are trained using a simple, autoregressive, predict-the-next token regime over enormous corpora. Codex [5], for example is a 12 billion parameters LLM trained on code. While such models certainly perform very well indeed at the task of prediction (*e.g.,* for code completion) , they are also quite good at other tasks, such as generating code from docstrings, and vice versa, after suitable fine-tuning [5].

One of the most exciting aspects of LLMs is *zero, one- or few-shot training*. In this line of work, the LLM is not subject to conventional fine-tuning (as is most typical with BERT, T5, RoBERTA, etc [6, 15, 17]) using a sizeable number of on-task training examples (typically in the range of 100 -100,000 examples); rather it is given a prefix, comprising just a handful of input-input pairs, and then is prompted with a query input (sans output). In this (highly sample-efficient) regime, LLMs are known to perform surprisingly well. Most remarkably, few-shot training *does not require any weight adjustment whatsoever.* Rather, the LLM leverages the information in the first part of the prompt to condition itself to perform the task reflected in the few examples. This works because the massive capacity (billions of parameters!) of the model allows it to condition its generative behaviour on the given prompt in extremely varied, subtle & flexible ways. An example two-shot training prompt, for the task of English-German translation, might be, for example:

```
The sentence "how are you?" in German
is "wie geht es?". The sentence "See
you later!" in German is "Bis Bald!".
The sentence "How much is that apple?"
in German is<submit>
```

If prompted with this, when one hits the submit button, GPT3 responds `"Wie viel kostet diese Apfel?"`, which is a good translation[1]. Likewise, LLMs are known to be capable of few-shot learning on a wide range of tasks, including question-answering, natural language inference, summarization, *etc.* It should be noted that few-shot learning is very challenging indeed, and the aptitude of LLMs to learn to perform different tasks in this regime is quite phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?
(2) Can this few-shot learning capacity be extended to same-project learning on this same task?
(3) How does the performance of LLMs in the above two settings compare with that of state-of-the-art models?

# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

# Prompting LLMs For Software Development Automation

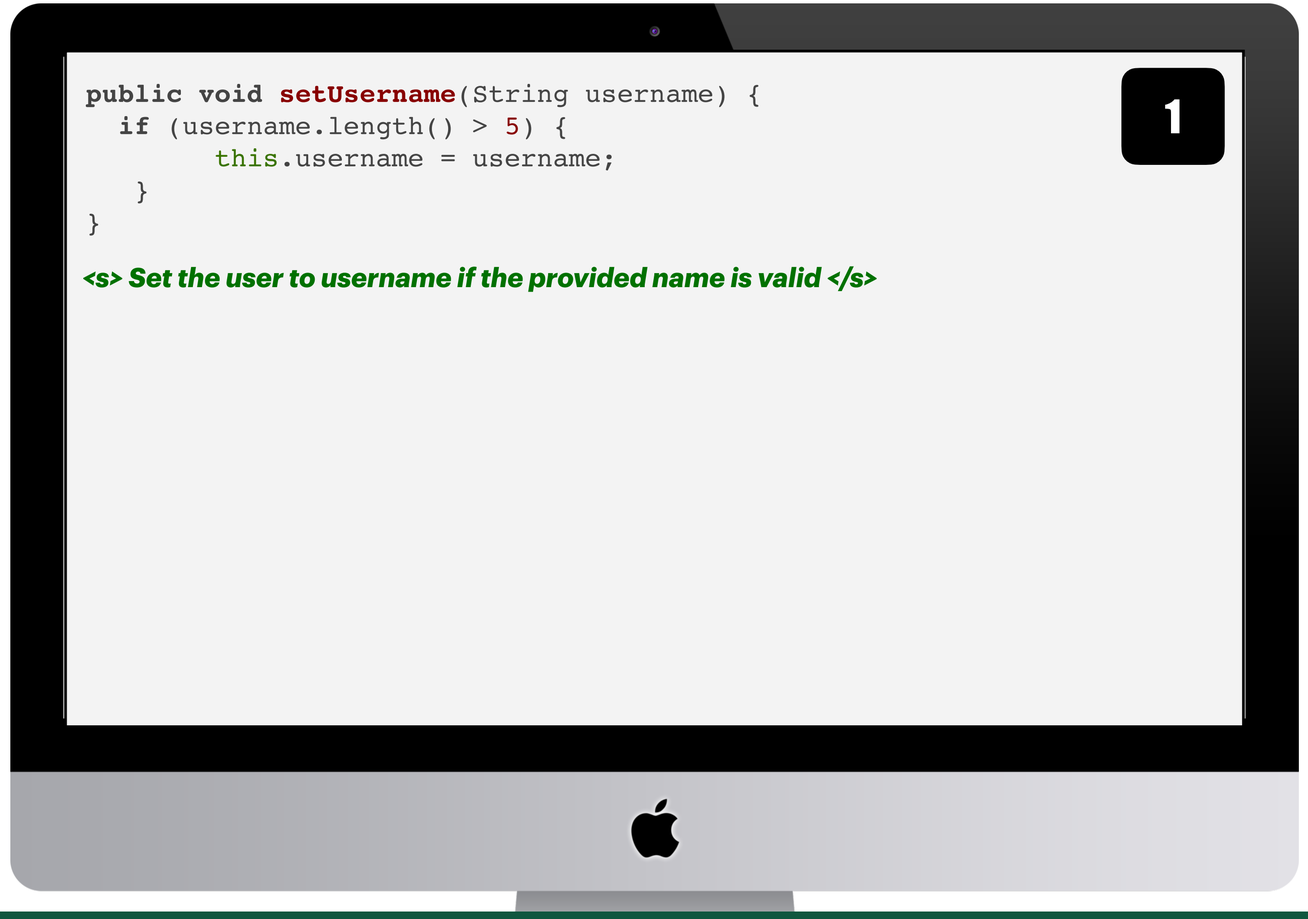
phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?



# Prompting LLMs For Software Development Automation

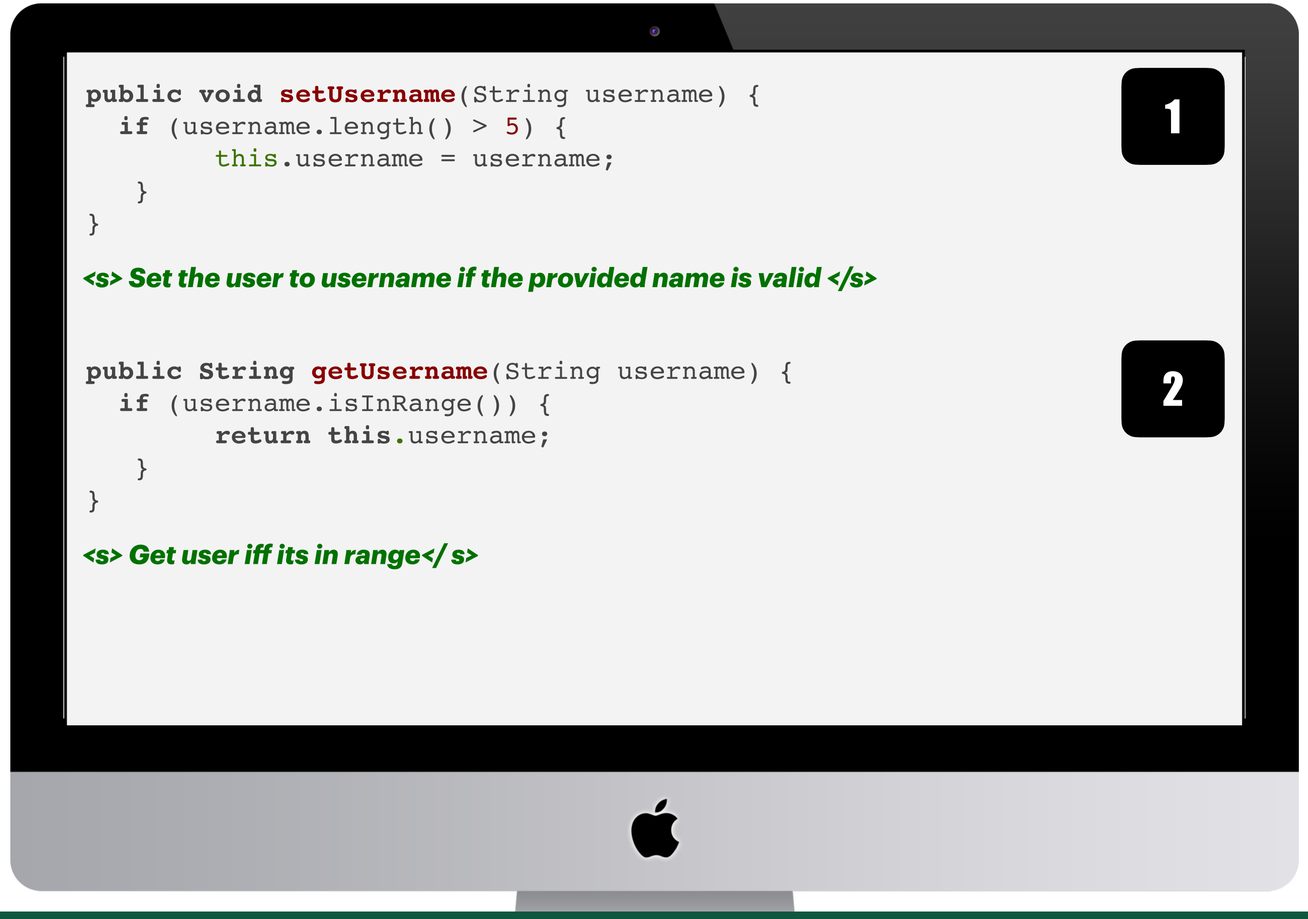
phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?



# Prompting LLMs For Software Development Automation

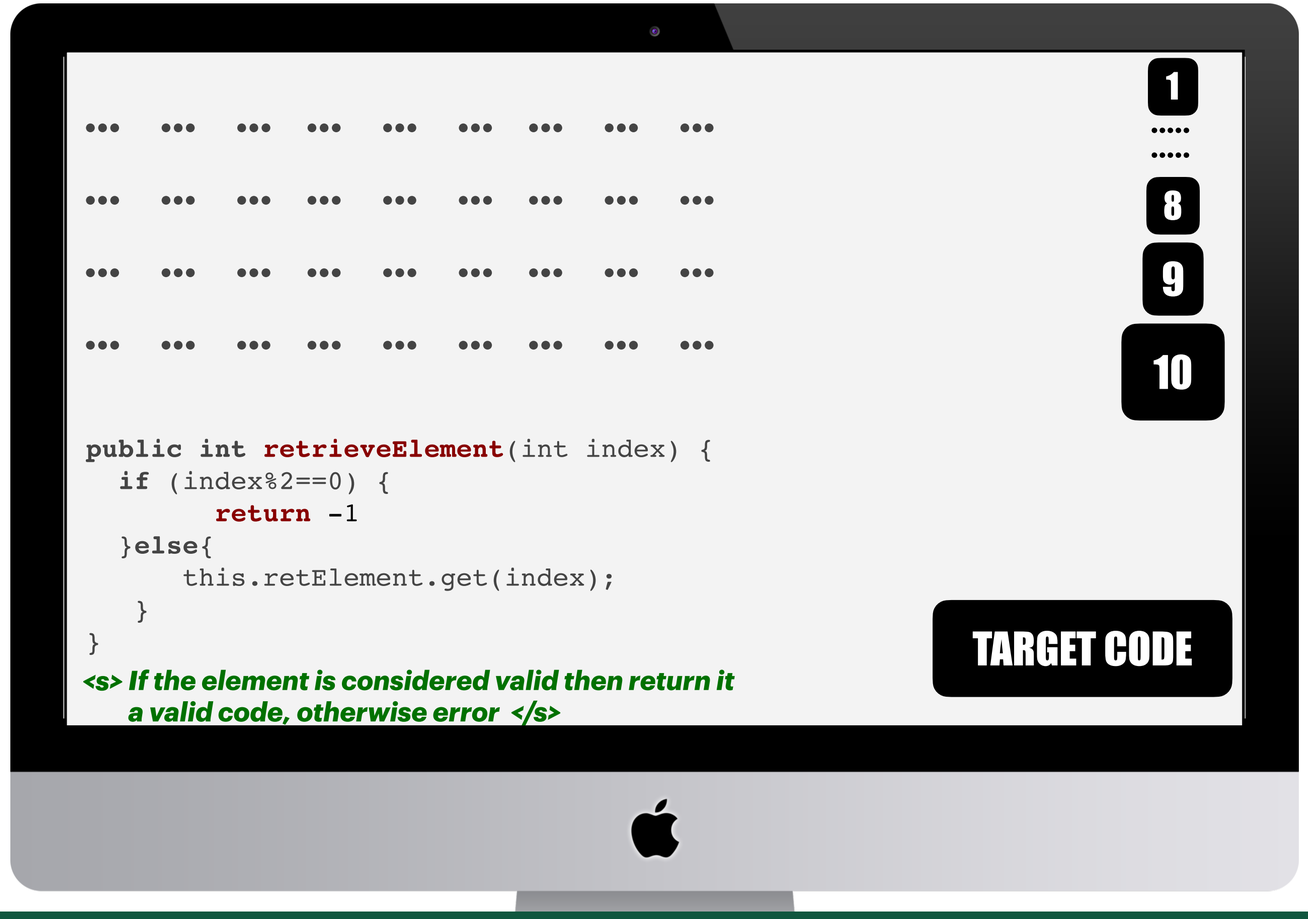
phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?



# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

| Language | Models | | | | | |
|---|---|---|---|---|---|---|
| | CodeBERT | PolyGlot CodeBERT | GraphCodeBERT | PolyGlot GraphCodeBERT | CodeT5 | **10 Shot Learning** Codex |
| Java | 18.8 | 20.22 | 18.52 | 19.94 | 19.78 | **21.88** |
| Python | 17.73 | 18.19 | 17.35 | 18.33 | 19.98 | **20.76** |
| Ruby | 12.61 | 14.64 | 12.6 | 14.9 | 15.33 | **16.95** |
| JS | 14.30 | 16.34 | 15.21 | 15.92 | 15.98 | **18.42** |
| Go | 18.5 | 19.18 | 18.71 | 19.3 | 19.91 | **22.65** |
| PHP | 25.88 | 26.46 | 25.97 | 26.54 | 26.32 | **26.63** |
| Average | 17.97 | 19.17 | 18.06 | 19.16 | 19.55 | **21.22** |

# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

**Observation 1**. With 10 samples, Codex outperforms all fine-tuned foundation models CodeT5, CodeBERT, GraphCodeBERT, Polyglot CodeBERT, and PolyGlotGraphCodeBERT in all six programming languages, even though the fine-tuned models are trained with thousands of data.

*Cross-project Shot Selection*
*i.e., RANDOM*

# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?

(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

**Observation 4**. Zero-shot and one-shot training in Codex do not work for code summarization task.

# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?
(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

| | Codex Cross-project | Codex (same-project) |
|---|---|---|
| P1 | 19.28 | **19.65** |
| P2 | 20.11 | **22.34** |
| ... | 26.97 | **39.46** |
| | 18.91 | **19.29** |
| | 22.23 | **23.03** |
| | 20.52 | **22.74** |
| | 18.98 | **19.65** |
| P8 | 26.95 | **28.82** |
| | 21.74 | **24.37** |

# Prompting LLMs For Software Development Automation

phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?
(2) Can this few-shot learning capacity be extended to same-project learning on this same task?

| | Codex Cross-project | Codex (same-project) |
|---|---|---|
| P1 | 19.28 | **19.65** |
| P2 | 20.11 | **22.34** |
| ... | 26.97 | **39.46** |
| | 18.91 | **19.29** |
| | 22.23 | **23.03** |
| | 20.52 | **22.74** |
| | 18.98 | **19.65** |
| P8 | 26.95 | **28.82** |
| | 21.74 | **24.37** |

**Observation 2.** Same-project few-shot training improves the Codex model's performance for all 8 projects.

# Prompting LLMs For Software Development Automation

## Few-shot training LLMs for project-specific code-summarization

Toufique Ahmed
University of California, Davis
Davis, California, USA
tfahmed@ucdavis.edu

Premkumar Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu@ucdavis.edu

**ABSTRACT**

Very large language models (LLMs), such as GPT-3 and Codex have achieved state-of-the-art performance on several natural-language tasks, and show great promise also for code. A particularly exciting aspect of LLMs is their knack for few-shot and zero-shot learning: they can learn to perform a task with very few examples. Few-shotting has particular synergies in software engineering, where there are a lot of phenomena (identifier names, APIs, terminology, coding patterns) that are known to be *highly* project-specific. However, project-specific data can be quite limited, especially early in the history of a project; thus the few-shot learning capacity of LLMs might be very relevant. In this paper, we investigate the use few-shot training with the very large GPT (Generative Pre-trained Transformer) Codex model, and find evidence suggesting that one can significantly surpass state-of-the-art models for code-summarization, leveraging project-specific training.

**KEYWORDS**

deep learning, code summarization, large language model

**ACM Reference Format:**
Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559555

### 1 INTRODUCTION

Very large language models (LLMs) are viewed as a revolutionary advance in natural language processing. Models such as GPT-3 [4], which have over 150 billion parameters, are trained using a simple, autoregressive, predict-the-next token regime over enormous corpora. Codex [5], for example is a 12 billion parameters LLM trained on code. While such models certainly perform very well indeed at the task of prediction (*e.g.*, for code completion) , they are also quite good at other tasks, such as generating code from docstrings, and vice versa, after suitable fine-tuning [5].

One of the most exciting aspects of LLMs is *zero, one- or few-shot training*. In this line of work, the LLM is not subject to conventional fine-tuning (as is most typical with BERT, T5, RoBERTA, etc [6, 15, 17]) using a sizeable number of on-task training examples (typically in the range of 100 -100,000 examples); rather it is given a prefix, comprising just a handful of input-input pairs, and then is prompted with a query input (sans output). In this (highly sample-efficient) regime, LLMs are known to perform surprisingly well. Most remarkably, few-shot training *does not require any weight adjustment whatsoever.* Rather, the LLM leverages the information in the first part of the prompt to condition itself to perform the task reflected in the few examples. This works because the massive capacity (billions of parameters!) of the model allows it to condition its generative behaviour on the given prompt in extremely varied, subtle & flexible ways. An example two-shot training prompt, for the task of English-German translation, might be, for example:

```
The sentence "how are you?" in German
is "wie geht es?". The sentence "See
you later!" in German is "Bis Bald!".
The sentence "How much is that apple?"
in German is<submit>
```

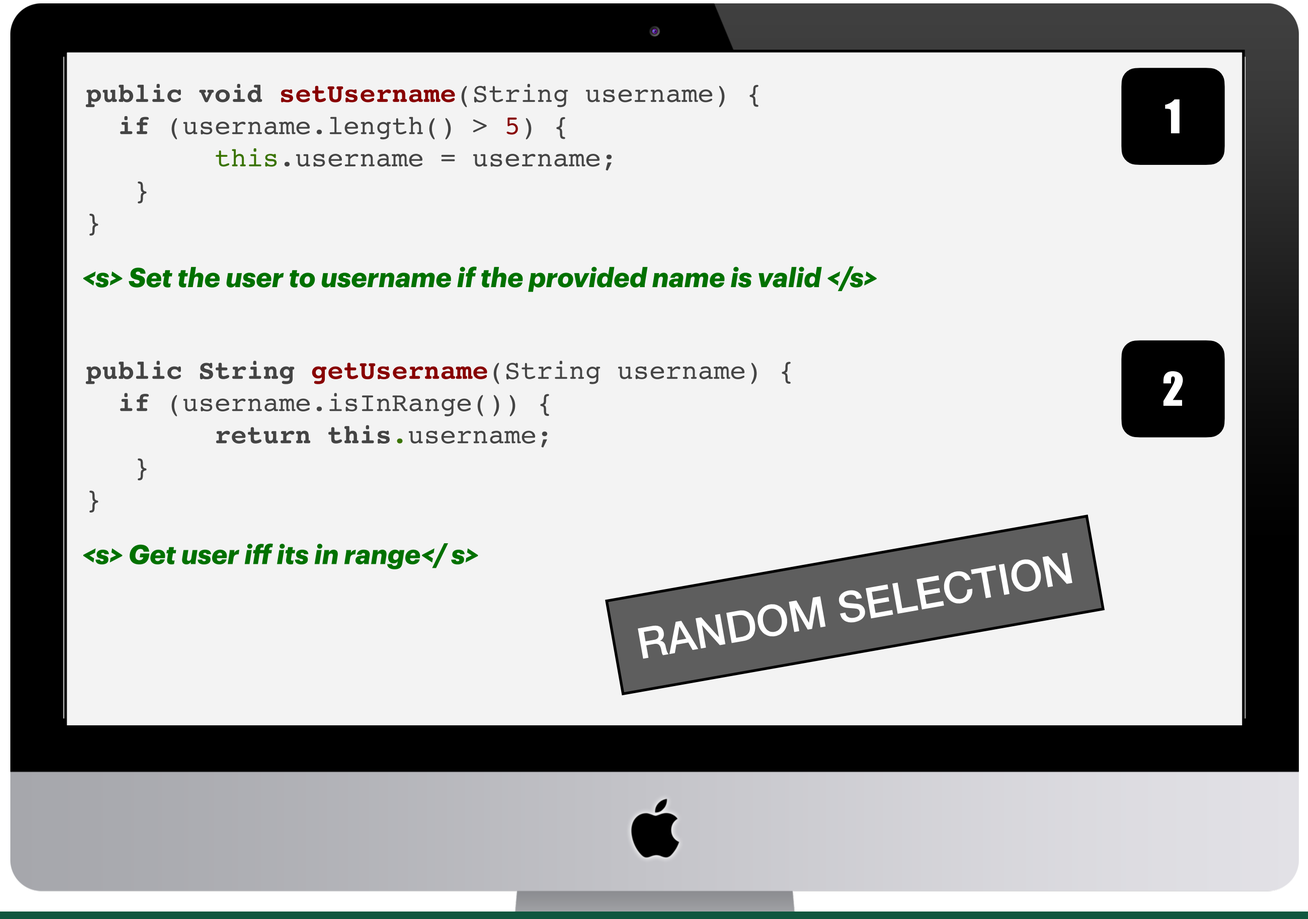
If prompted with this, when one hits the submit button, GPT3 responds `"Wie viel kostet diese Apfel?"`, which is a good translation[1]. Likewise, LLMs are known to be capable of few-shot learning on a wide range of tasks, including question-answering, natural language inference, summarization, *etc.* It should be noted that few-shot learning is very challenging indeed, and the aptitude of LLMs to learn to perform different tasks in this regime is quite phenomenal[2].

Few-shot learning has a peculiar and interesting salience for software engineering: for dealing with project-specific linguistic phenomena. Most seasoned engineers are very well aware of this: different projects leverage different domain-specific concepts, and these are reflected in identifier naming, API calls, and coding patterns. It's been well-known right from the outset that language modeling for code has to deal with project-specific phenomena [10, 11, 22]. The sticking point here, however, is that project-specific data, especially early-on in a project's history, may be quite limited in volume. This suggests an exciting synergy with the few-shot learning aptitude of LLMs, which can make do with just a few samples.

In this short paper, we address three research questions:

(1) Does the few-shot learning capacity of large language models extend to the task of code summarization?
(2) Can this few-shot learning capacity be extended to same-project learning on this same task?
(3) How does the performance of LLMs in the above two settings compare with that of state-of-the-art models?






antoniomastropaolo.com

aura-se-lab.github.io

# Prompting LLMs For Software Development Automation

HOW CAN WE SELECT THE MOST RELEVANT EXAMPLES/SHOTS FOR THE TASK WE AIM TO SUPPORT?

# Prompting LLMs For Software Development Automation

HOW CAN WE SELECT THE MOST RELEVANT EXAMPLES/SHOTS FOR THE TASK WE AIM TO SUPPORT?

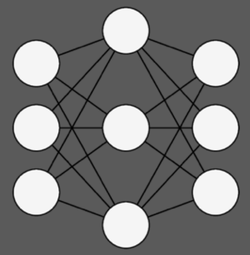
LOG MESSAGE GENERATION

*INPUT*

```
public ConnectionConsumer createConnectionConsumer
    (final Destination destination)
       throws JMSException{

   if(LOGGER.isTraceEnabled())
   {
       ActiveMQRALogger.LOGGER.
         trace("Create connectionConsumer");
   }
   else {
       throw new IllegalStateException(ISE);
   }
}
```

*LANCE*



*OUTPUT*

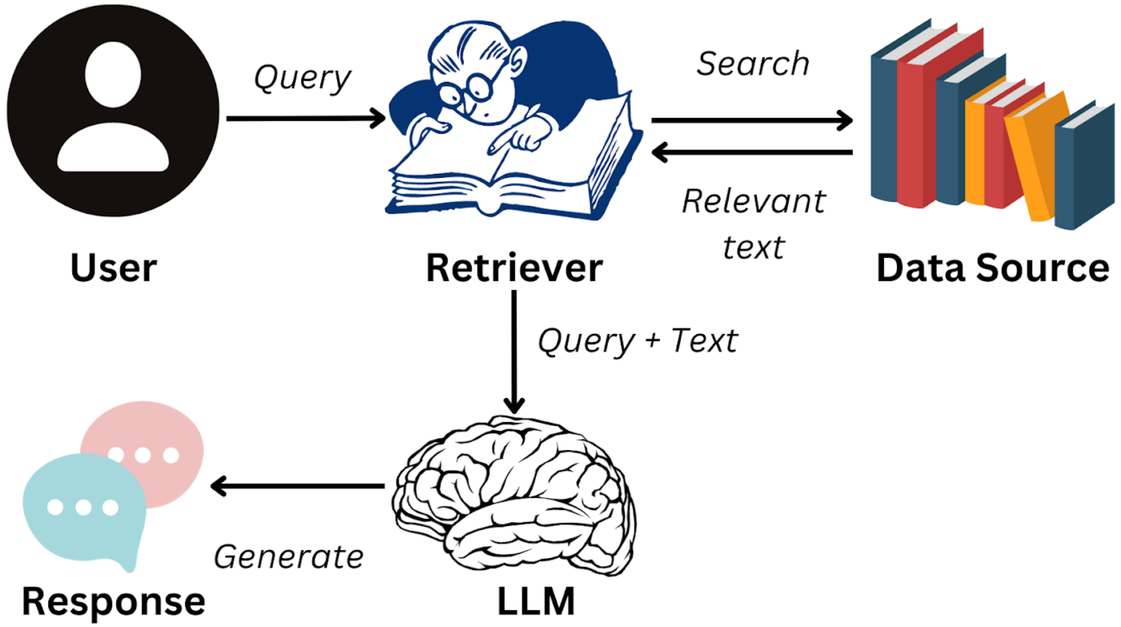
```
public ConnectionConsumer createConnectionConsumer
    (final Destination destination)
       throws JMSException{

   if(LOGGER.isTraceEnabled())
   {
       ActiveMQRALogger.LOGGER.
         trace("Create connectionConsumer");
   }
   else {
       throw new IllegalStateException(ISE);
   }
}
```

# Retrieval Augmented Generation

# Prompting LLMs For Software Development Automation

## Retrieval Augmented Generation



# Prompting LLMs For Software Development Automation


```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

# Prompting LLMs For Software Development Automation


```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

# Prompting LLMs For Software Development Automation


```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE

# Prompting LLMs For Software Development Automation


```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE

# Prompting LLMs For Software Development Automation


```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE

# Prompting LLMs For Software Development Automation

RAG

```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
```

0.5

CodeBLEU

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE


antoniomastropaolo.com

aura-se-lab.github.io

# Prompting LLMs For Software Development Automation

RAG

```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

0.5

0.9

CodeBLEU

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE


antoniomastropaolo.com

aura-se-lab.github.io

# Prompting LLMs For Software Development Automation

RAG

```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}

public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}

public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
```

0.5

0.9

0.3

CodeBLEU

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE

# Prompting LLMs For Software Development Automation

RAG

```java
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}
```

```java
public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}
```

0.9

```java
public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    if (filePath == null || filePath.isEmpty()) {
        logger.severe("File path is empty or null");
        return false;
    } else {
        logger.info("Processing file at: " + filePath);
        // Assume file processing logic here
        logger.info("File processed successfully");
        return true;
    }
}
```

CodeBLEU

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE

# Prompting LLMs For Software Development Automation


```java
public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}
```

**Concat**

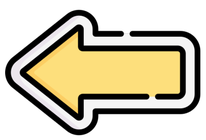
```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

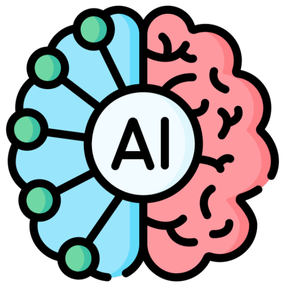
**TARGET CODE**

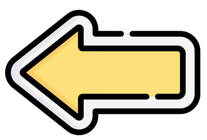
# Prompting LLMs For Software Development Automation


```java
public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created successfully");
    }
}
```

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info("User " + username + " del. successfully");
    }
}
```





**Concat**

```java
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

**TARGET CODE**

# Prompting LLMs For Software Development Automation


## APPROACHES FOR IMPLEMENTING THE RETRIEVER

Embedding-based methods

Textual/Overlapping-based methods

# Prompting LLMs For Software Development Automation

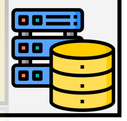
## APPROACHES FOR IMPLEMENTING THE RETRIEVER

Embedding-based methods

```
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}
```

```
public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created
                                    successfully");
    }
}
```

```
public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    ..........
```



# Prompting LLMs For Software Development Automation

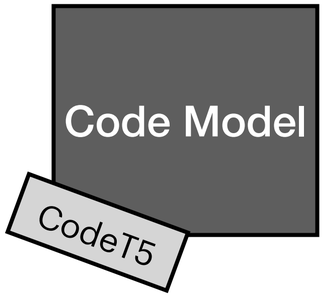
## APPROACHES FOR IMPLEMENTING THE RETRIEVER

Embedding-based methods

```
public int multiply(int a, int b) {
    Logger logger = Logger.getLogger("Calculator");
    logger.info("Starting multiply method");
    int result = a * b;
    logger.info("Multiplication result: " + result);
    return result;
}
```

```
public void createUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting createUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        logger.info("User " + username + " created
                                    successfully");
    }
}
```

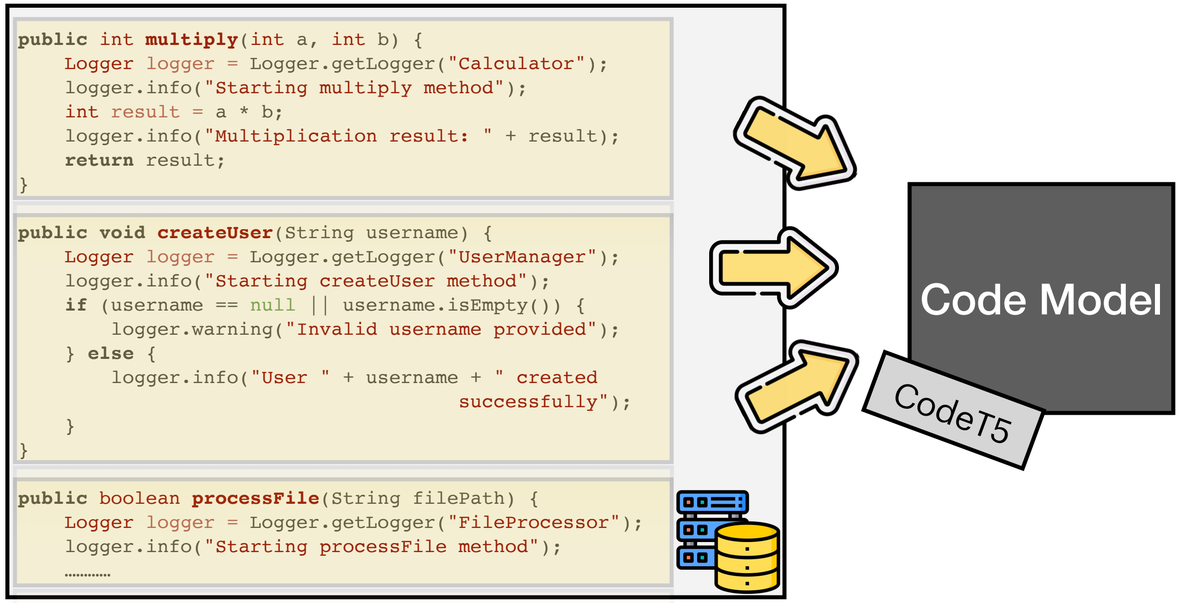
```
public boolean processFile(String filePath) {
    Logger logger = Logger.getLogger("FileProcessor");
    logger.info("Starting processFile method");
    ..........
```



# Prompting LLMs For Software Development Automation

## APPROACHES FOR IMPLEMENTING THE RETRIEVER

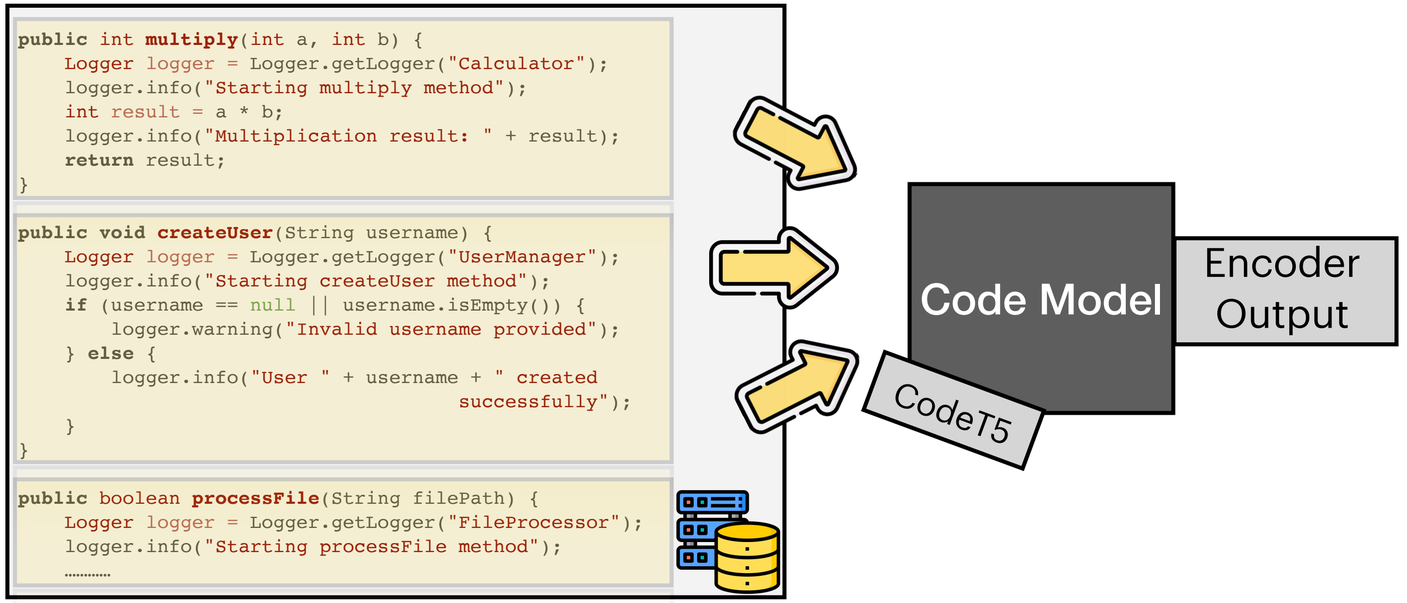
Embedding-based methods



# Prompting LLMs For Software Development Automation

## APPROACHES FOR IMPLEMENTING THE RETRIEVER

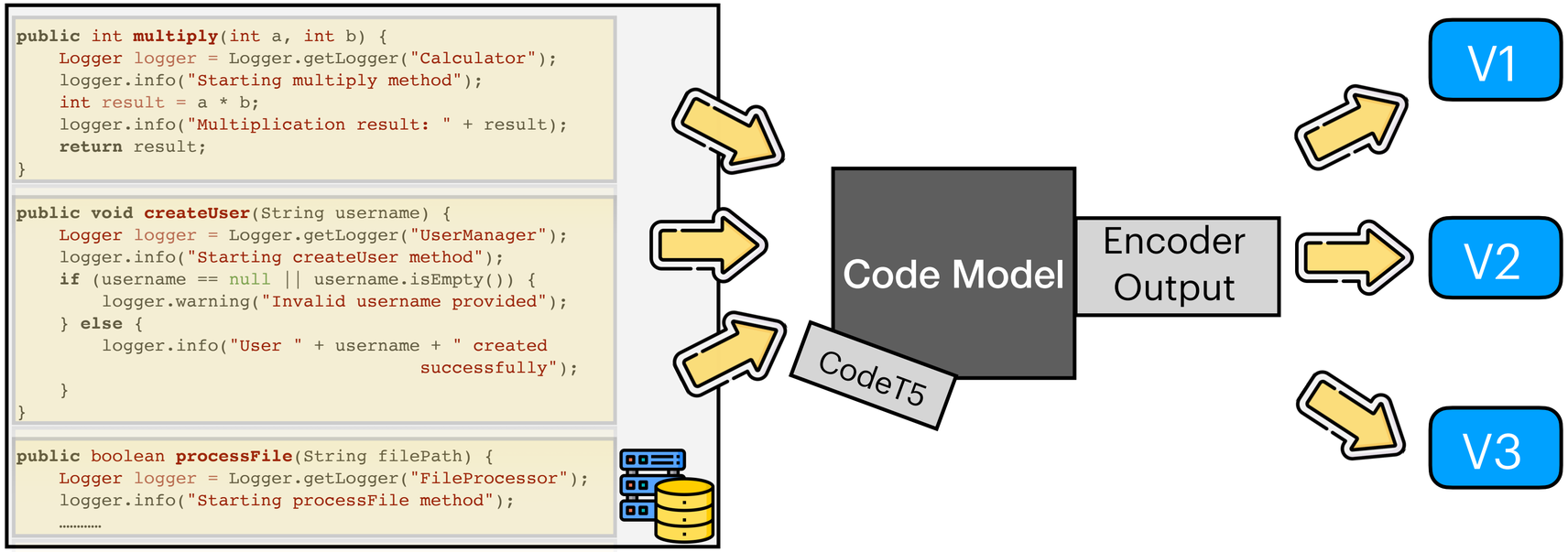
Embedding-based methods



# Prompting LLMs For Software Development Automation

## APPROACHES FOR IMPLEMENTING THE RETRIEVER

Embedding-based methods

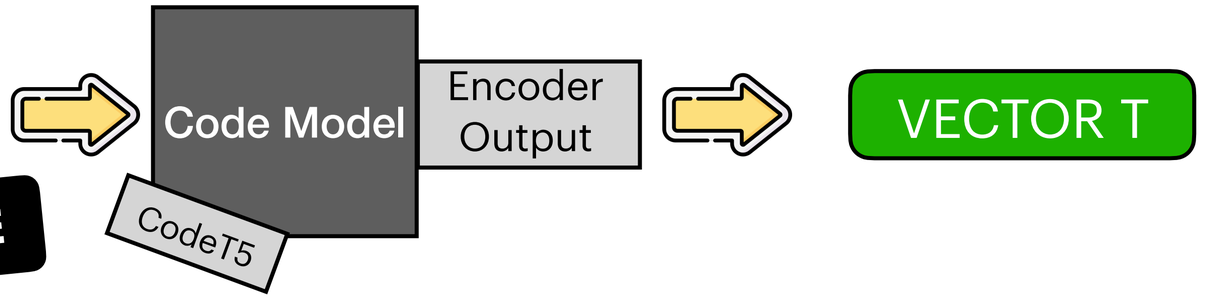


# Prompting LLMs For Software Development Automation


## APPROACHES FOR IMPLEMENTING THE RETRIEVER

```
public void deeleteUser(String username) {
    Logger logger = Logger.getLogger("UserManager");
    logger.info("Starting deleteUser method");
    if (username == null || username.isEmpty()) {
        logger.warning("Invalid username provided");
    } else {
        // Assume logic here
        logger.info(<LOG_MESSAGE>);
    }
}
```

TARGET CODE



# Prompting LLMs For Software Development Automation


## APPROACHES FOR IMPLEMENTING THE RETRIEVER

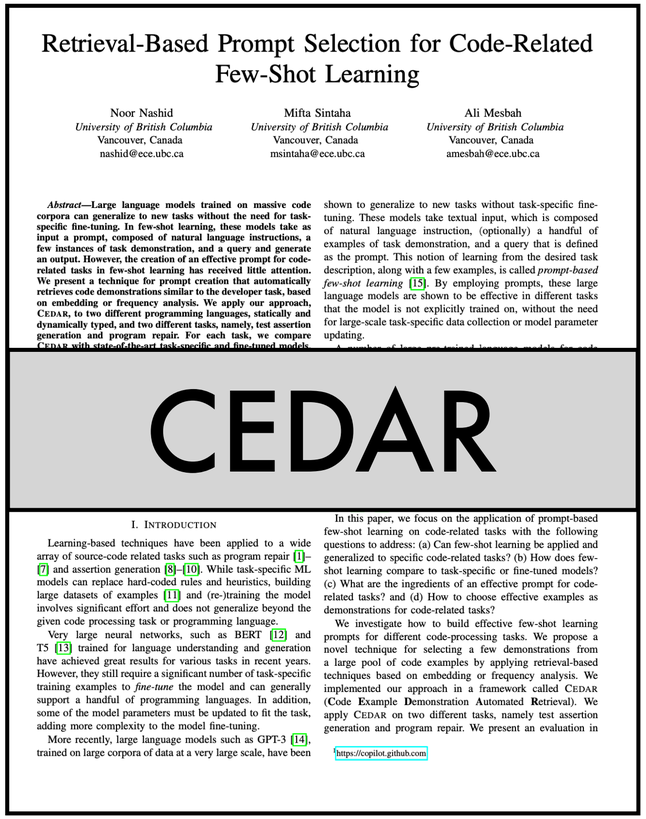
VECTOR T  VECTOR #1

VECTOR T  VECTOR #2

VECTOR T  VECTOR #3

Cosine Sim./Euclidean Distance

# Prompting LLMs For Software Development Automation



Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning

Noor Nashid
*University of British Columbia*
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
*University of British Columbia*
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
*University of British Columbia*
Vancouver, Canada
amesbah@ece.ubc.ca

***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models**

CEDAR

## I. Introduction

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

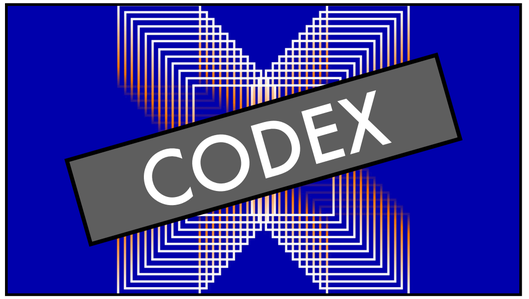
In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (Code Example Demonstration Automated Retrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1]https://copilot.github.com

Assert Statement Generation **1**

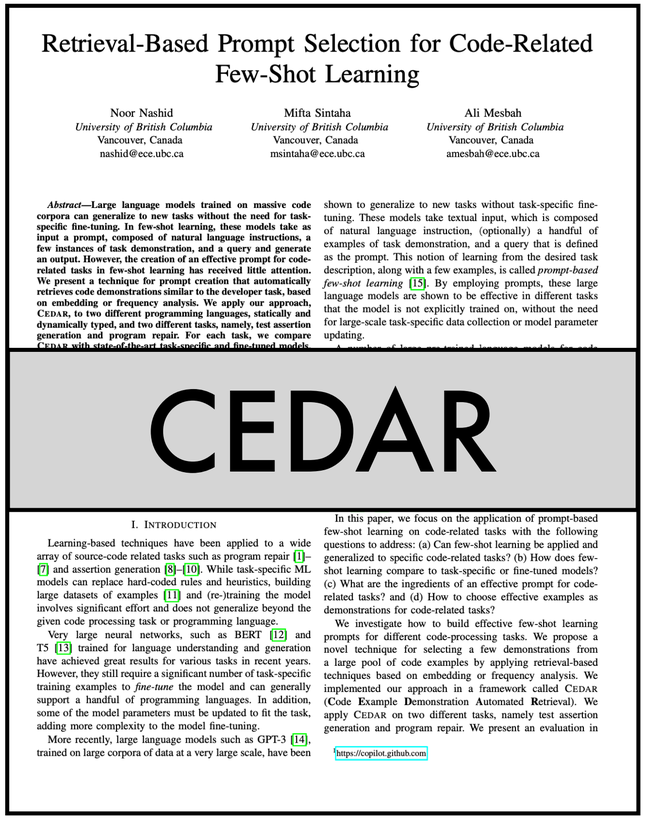
Automated Program Repair **2**




***org.junit.Assert.assert...***

***Code fixing the issue***

# Prompting LLMs For Software Development Automation



Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning

Noor Nashid
*University of British Columbia*
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
*University of British Columbia*
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
*University of British Columbia*
Vancouver, Canada
amesbah@ece.ubc.ca

***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models**

CEDAR

## I. Introduction

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

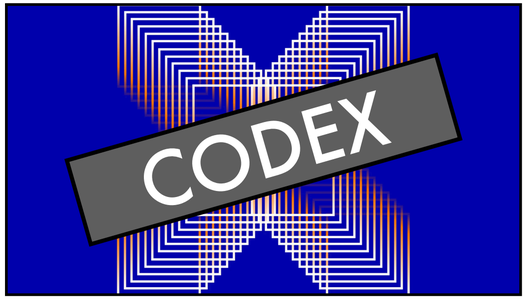
In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (Code Example Demonstration Automated Retrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1]https://copilot.github.com

Assert Statement Generation 1

Automated Program Repair 2




```
org.junit.Assert.assert...
```

***Code fixing the issue***

# Prompting LLMs For Software Development Automation

## Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning


Noor Nashid
*University of British Columbia*
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
*University of British Columbia*
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
*University of British Columbia*
Vancouver, Canada
amesbah@ece.ubc.ca



***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models. The empirical results show that, with only a few relevant code demonstrations, our prompt creation technique is effective in both tasks with an accuracy of 76% and 52% for exact matches in test assertion generation and program repair tasks, respectively. For assertion generation, CEDAR outperforms existing task-specific and fine-tuned models by 333% and 11%, respectively. For program repair, CEDAR yields 189% better accuracy than task-specific models and is competitive with recent fine-tuned models. These findings have practical implications for practitioners, as CEDAR could potentially be applied to multilingual and multitask settings without task or language-specific training with minimal examples and effort.**

***Index Terms*—Large Language Models, Transformers, Few-shot learning, Program repair, Test assertion generation**


### I. INTRODUCTION

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

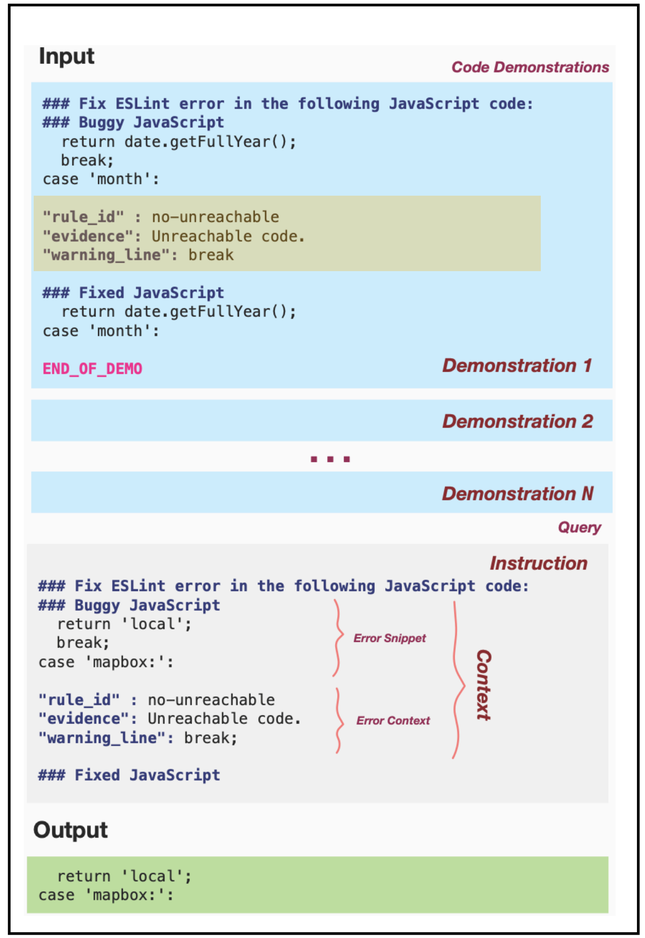
More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

A number of large pre-trained language models for code generation [16]–[18] have been proposed, which primarily focus on prompts to generate code from natural language descriptions. Large language models such as CODEX are employed by GITHUB COPILOT [1] for code completion tasks. Recent studies assess how the adoption of code completion with large language models could be helpful to developer productivity [19]–[21], or solving coding interview problems and competitive programming [22]–[24], or how to employ feedback from external sources such as generated tests to improve the quality of generated code [25], [26]. There are also efforts to employ large language models to patch simple bugs [27], [28].

In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (**C**ode **E**xample **D**emonstration **A**utomated **R**etrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1] https://copilot.github.com



# Prompting LLMs For Software Development Automation

Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning


Noor Nashid
*University of British Columbia*
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
*University of British Columbia*
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
*University of British Columbia*
Vancouver, Canada
amesbah@ece.ubc.ca



***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models. The empirical results show that, with only a few relevant code demonstrations, our prompt creation technique is effective in both tasks with an accuracy of 76% and 52% for exact matches in test assertion generation and program repair tasks, respectively. For assertion generation, CEDAR outperforms existing task-specific and fine-tuned models by 333% and 11%, respectively. For program repair, CEDAR yields 189% better accuracy than task-specific models and is competitive with recent fine-tuned models. These findings have practical implications for practitioners, as CEDAR could potentially be applied to multilingual and multitask settings without task or language-specific training with minimal examples and effort.**

***Index Terms*—Large Language Models, Transformers, Few-shot learning, Program repair, Test assertion generation**


## I. INTRODUCTION

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

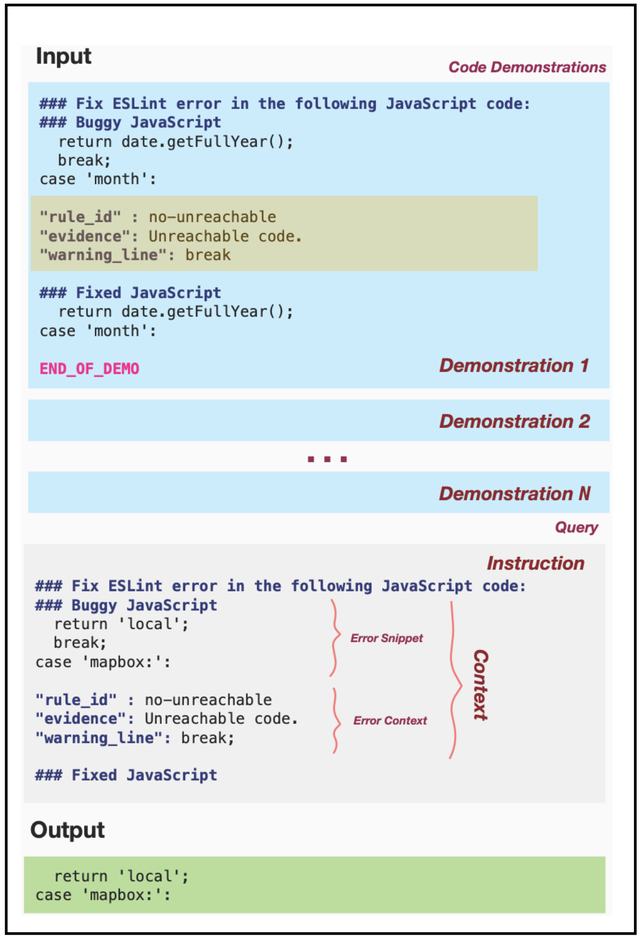
More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

A number of large pre-trained language models for code generation [16]–[18] have been proposed, which primarily focus on prompts to generate code from natural language descriptions. Large language models such as CODEX are employed by GITHUB COPILOT [1] for code completion tasks. Recent studies assess how the adoption of code completion with large language models could be helpful to developer productivity [19]–[21], or solving coding interview problems and competitive programming [22]–[24], or how to employ feedback from external sources such as generated tests to improve the quality of generated code [25], [26]. There are also efforts to employ large language models to patch simple bugs [27], [28].

In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (**C**ode **E**xample **D**emonstration **A**utomated **R**etrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1] https://copilot.github.com




*Static Analyzer – ESLint*

# Prompting LLMs For Software Development Automation

## Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning


Noor Nashid
*University of British Columbia*
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
*University of British Columbia*
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
*University of British Columbia*
Vancouver, Canada
amesbah@ece.ubc.ca



***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models. The empirical results show that, with only a few relevant code demonstrations, our prompt creation technique is effective in both tasks with an accuracy of 76% and 52% for exact matches in test assertion generation and program repair tasks, respectively. For assertion generation, CEDAR outperforms existing task-specific and fine-tuned models by 333% and 11%, respectively. For program repair, CEDAR yields 189% better accuracy than task-specific models and is competitive with recent fine-tuned models. These findings have practical implications for practitioners, as CEDAR could potentially be applied to multilingual and multitask settings without task or language-specific training with minimal examples and effort.**

***Index Terms*—Large Language Models, Transformers, Few-shot learning, Program repair, Test assertion generation**


### I. INTRODUCTION

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

A number of large pre-trained language models for code generation [16]–[18] have been proposed, which primarily focus on prompts to generate code from natural language descriptions. Large language models such as CODEX are employed by GITHUB COPILOT [1] for code completion tasks. Recent studies assess how the adoption of code completion with large language models could be helpful to developer productivity [19]–[21], or solving coding interview problems and competitive programming [22]–[24], or how to employ feedback from external sources such as generated tests to improve the quality of generated code [25], [26]. There are also efforts to employ large language models to patch simple bugs [27], [28].

In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (**C**ode **E**xample **D**emonstration **A**utomated **R**etrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1] https://copilot.github.com

| Task | Approach | Model type | Exact Match Acc. (%) |
|---|---|---|---|
| Assertion Generation | ATLAS | Task-specific | 17.66 |
| | Mastropaolo et al. [9] | Fine-tuned | 57.60 |
| | Mastropaolo et al. [10] | Fine-tuned | 68.93 |
| | **CEDAR** | **Few-shot learner** | **76.55** |
| Program Repair | HOPPITY | Task-specific | 7.90 |
| | CODONUT | Task-specific | 11.70 |
| | SEQUENCER | Task-specific | 17.90 |
| | TFIX | Fine-tuned | 49.30 |
| | **CEDAR** | **Few-shot learner** | **51.72** |