GenAI for Software Engineering 2025

Generative AI

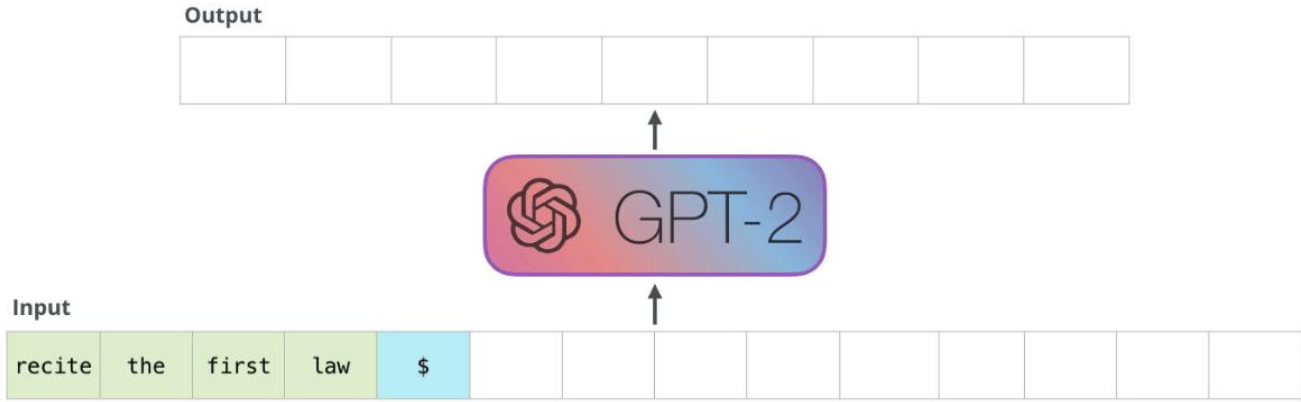
Jorge Melegati

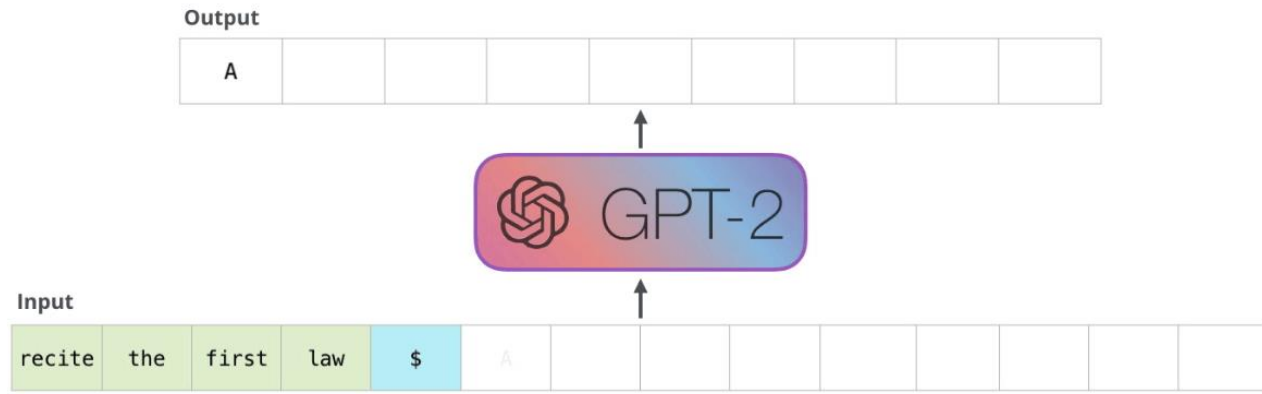
Generative language models

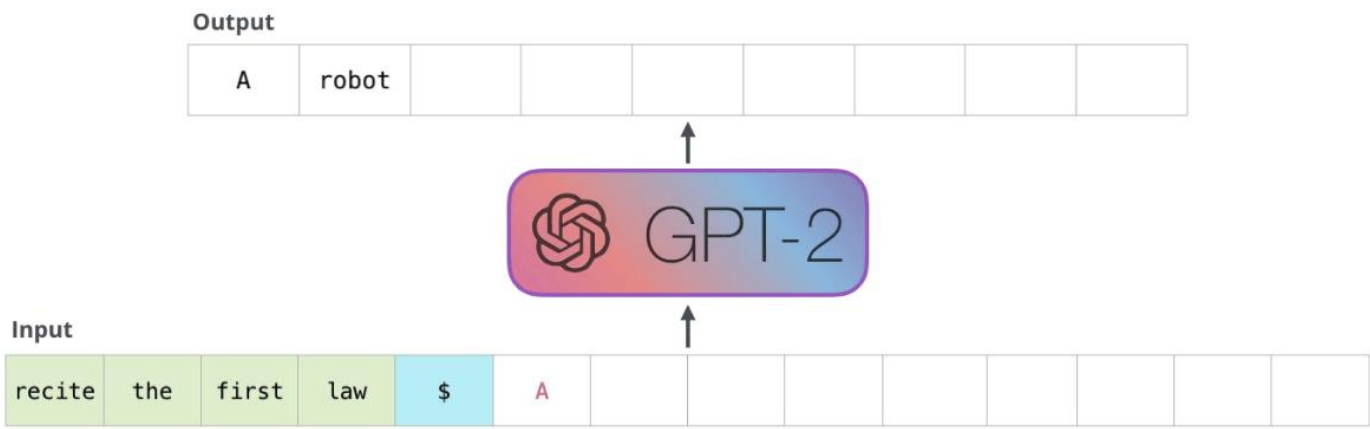
- Generative models are designed to create plausible continuations of the input
- Predict the next token based on previous tokens, one token at a time
- Example: GPT-3 (Generative Pre-trained Transformer)

GPT

- Decoder-only
- Outputs one token at time


Output
GPT-2
Input
recite
the
first
law
$


Output
A
GPT-2
Input
recite
the
first
law
$
A


Output
A
robot
GPT-2
Input
recite
the
first
law
$
A

Language model meta-learning

Large Language Models, such as GPT-2, present the capability of meta-learning!

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**1]

Abstract

Natural language processing tasks, such as question answering, machine translation, reading comprehension, and summarization, are typically approached with supervised learning on task-specific datasets. We demonstrate that language models begin to learn these tasks without any explicit supervision when trained on a new dataset of millions of webpages called WebText. When conditioned on a document plus questions, the answers generated by the language model reach 55 F1 on the CoQA dataset - matching or exceeding the performance of 3 out of 4 baseline systems without using the 127,000+ training examples. The capacity of the language model is essential to the success of zero-shot task transfer and increasing it improves performance in a log-linear fashion across tasks. Our largest model, GPT-2, is a 1.5B parameter Transformer that achieves state of the art results on 7 out of 8 tested language modeling datasets in a zero-shot setting

competent generalists. We would like to move towards more general systems which can perform many tasks – eventually without the need to manually create and label a training dataset for each one.

The dominant approach to creating ML systems is to collect a dataset of training examples demonstrating correct behavior for a desired task, train a system to imitate these behaviors, and then test its performance on independent and identically distributed (IID) held-out examples. This has served well to make progress on narrow experts. But the often erratic behavior of captioning models (Lake et al., 2017), reading comprehension systems (Jia & Liang, 2017), and image classifiers (Alcorn et al., 2018) on the diversity and variety of possible inputs highlights some of the shortcomings of this approach.

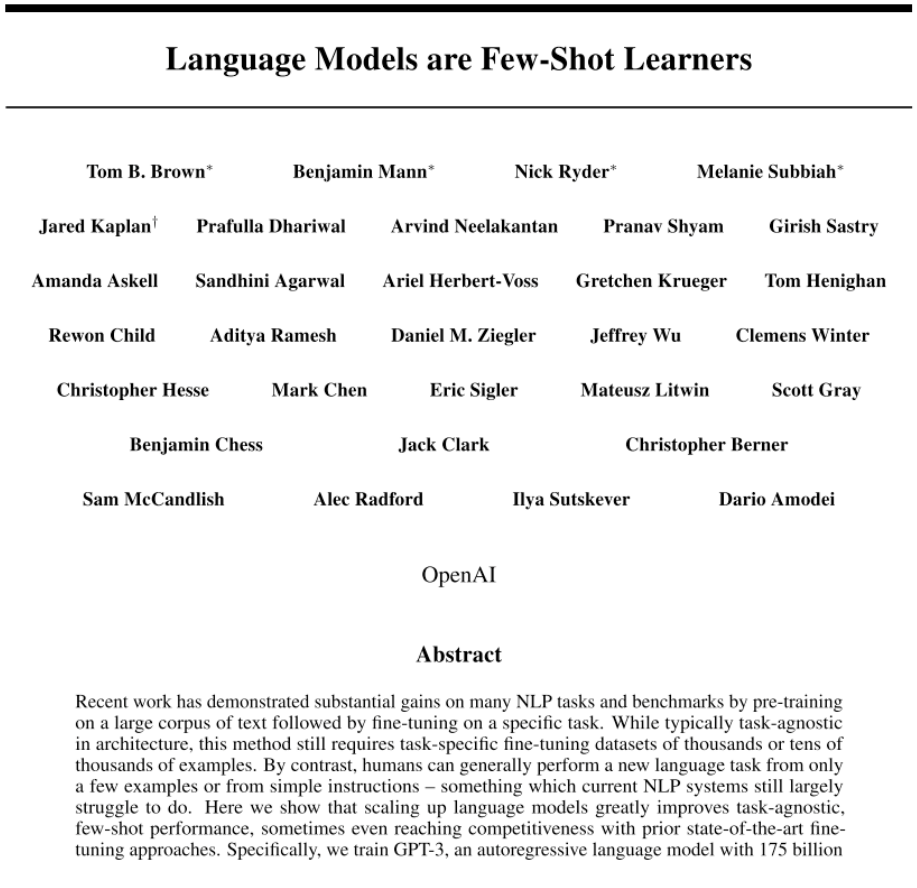
Our suspicion is that the prevalence of single task training on single domain datasets is a major contributor to the lack of generalization observed in current systems. Progress towards robust systems with current architectures is likely to require training and measuring performance on a wide range of domains and tasks. Recently, several benchmarks

Language model meta-learning

- At training time, the model develop a broad set of pattern recognition abilities
- At inference time, use the abilities to rapidly adapt to or recognize the task
  - In-context learning: use the text input as a form of task specification

Examples as in-context learning

- Zero-shot vs few-shot
  - Zero-shot: no examples are provided in the prompt
  - Few-shot: small number of examples provided
- Examples provide more context to the model



Language Models are Few-Shot Learners

Tom B. Brown* Benjamin Mann* Nick Ryder* Melanie Subbiah*

Jared Kaplan† Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry

Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan

Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter

Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

Benjamin Chess Jack Clark Christopher Berner

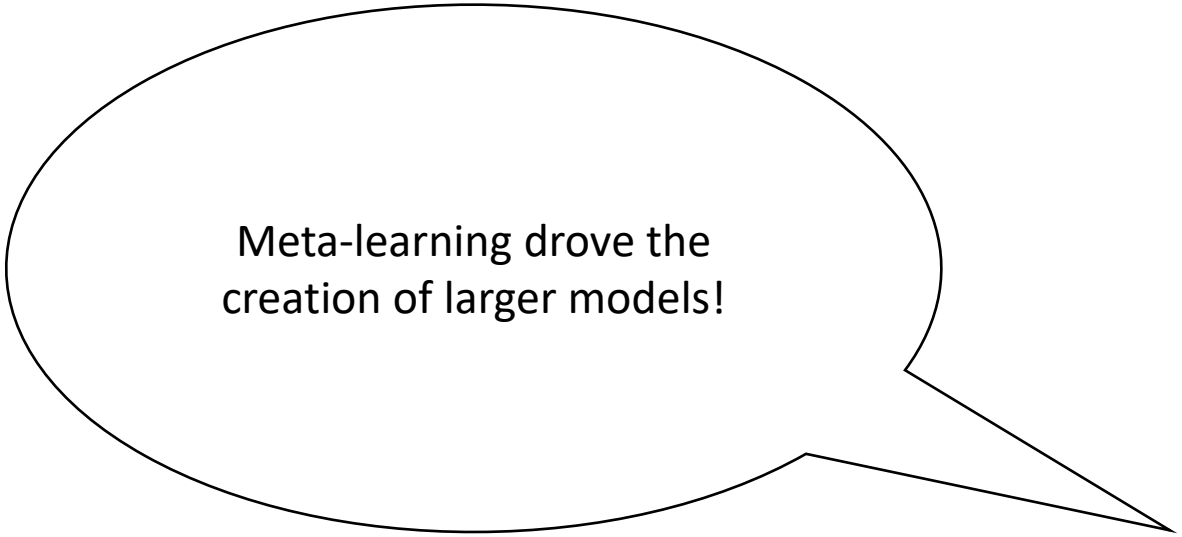
Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI

Abstract

Recent work has demonstrated substantial gains on many NLP tasks and benchmarks by pre-training on a large corpus of text followed by fine-tuning on a specific task. While typically task-agnostic in architecture, this method still requires task-specific fine-tuning datasets of thousands or tens of thousands of examples. By contrast, humans can generally perform a new language task from only a few examples or from simple instructions – something which current NLP systems still largely struggle to do. Here we show that scaling up language models greatly improves task-agnostic, few-shot performance, sometimes even reaching competitiveness with prior state-of-the-art fine-tuning approaches. Specifically, we train GPT-3, an autoregressive language model with 175 billion

Models are becoming larger…



Model	Size (parameters)
BERT (base)	110 million
BERT (large)	340 million
GPT-1	117 million
GPT-2	1.5 billion
GPT-3 and GPT-3.5	175 billion
GPT-4	Undisclosed (sources say 1.7 trillion)

Early LLMs limitation

- Misalignment between:
  - traditional training objective (predict next token)
  - the objective of following user intentions

Instruct tuning LLMs

- Fine-tuning: Reinforcement learning from human feedback
- Human preferences as reward signal to fine-tune the model

Training language models to follow instructions with human feedback

Long Ouyang* Jeff Wu* Xu Jiang* Diogo Almeida* Carroll L. Wainwright*
Pamela Mishkin* Chong Zhang Sandhini Agarwal Katarina Slama Alex Ray
John Schulman Jacob Hilton Fraser Kelton Luke Miller Maddie Simens
Amanda Askell† Peter Welinder Paul Christiano*†
Jan Leike* Ryan Lowe*
OpenAI

Abstract

Making language models bigger does not inherently make them better at following a user's intent. For example, large language models can generate outputs that are untruthful, toxic, or simply not helpful to the user. In other words, these models are not *aligned* with their users. In this paper, we show an avenue for aligning language models with user intent on a wide range of tasks by fine-tuning with human feedback. Starting with a set of labeler-written prompts and prompts submitted through a language model API, we collect a dataset of labeler demonstrations of the desired model behavior, which we use to fine-tune GPT-3 using supervised learning. We then collect a dataset of rankings of model outputs, which we use to further fine-tune this supervised model using reinforcement learning from human feedback. We call the resulting models *InstructGPT*. In human evaluations on our prompt distribution, outputs from the 1.3B parameter InstructGPT model are preferred to outputs from the 175B GPT-3, despite having 100x fewer parameters. Moreover, InstructGPT models show improvements in truthfulness and reductions in toxic output generation while having minimal performance regressions on public NLP datasets. Even though InstructGPT still makes simple mistakes, our results show that fine-tuning with human feedback is a promising direction for aligning language models with human intent.

Instruct tuning LLMs

- Goals: the bot should be:
  - Helpful (help to solve the task)
  - Honest (not fabricate information or mislead)
  - Harmless (not cause harm to people or the environment)

In-context learning strategies

- Standard Prompt Engineering
- Chain-of-thought
- Multiple agents

Prompt Engineering

- Prompts consist of context and instructions fed to a generative language model to achieve a defined task

- Prompt engineering consists of practices and techniques to optimize prompts to efficiently and effectively use generative language models

Prompting principles

- Principle 1: Write clear and specific instructions
- Principle 2: Give the model time to “think”

Writing clear and specific instructions

- Tactic 1: Use delimiters to clearly indicate distinct parts of the input
  - Delimiters can be ```, " " ”, <>,<tag></tag>

```
text = f"""
You should express what you want a model to do by \
providing instructions that are as clear and \
specific as you can possibly make them. \
This will guide the model towards the desired output, \
and reduce the chances of receiving irrelevant \
or incorrect responses. Don't confuse writing a \
clear prompt with writing a short prompt. \
In many cases, longer prompts provide more clarity \
and context for the model, which can lead to \
more detailed and relevant outputs.
"""
prompt = f"""
Summarize the text delimited by triple backticks \
into a single sentence.
```{text}```
"""
response = get_completion(prompt)
print(response)
```
```

Writing clear and specific instructions

- Tactic 2: Ask for a structured output
  - JSON, HTML

```
prompt = f"""
Generate a list of three made-up book titles along \
with their authors and genres.
Provide them in JSON format with the following keys:
book_id, title, author, genre.
"""
response = get_completion(prompt)
print(response)
```

Writing clear and specific instructions

- Tactic 3: Ask the model to check whether the conditions are satisfied

```
text_1 = f"""
Making a cup of tea is easy! First, you need to get some \
water boiling. While that's happening, \
grab a cup and put a tea bag in it. Once the water is \
hot enough, just pour it over the tea bag. \
Let it sit for a bit so the tea can steep. After a \
few minutes, take out the tea bag. If you \
like, you can add some sugar or milk to taste. \
And that's it! You've got yourself a delicious \
cup of tea to enjoy.
"""
prompt = f"""
You will be provided with text delimited by triple quotes.
If it contains a sequence of instructions, \
re-write those instructions in the following format:

Step 1 - ...
Step 2 - …
…
Step N - …

If the text does not contain a sequence of instructions, \
then simply write \"No steps provided.\"

\"\"\"{text_1}\"\"\"
"""
response = get_completion(prompt)
print("Completion for Text 1:")
print(response)
```

Writing clear and specific instructions

- Tactic 4: Few-shot programming

```
prompt = f"""
Your task is to answer in a consistent style.

<child>: Teach me about patience.

<grandparent>: The river that carves the deepest \
valley flows from a modest spring; the \
grandest symphony originates from a single note; \
the most intricate tapestry begins with a solitary thread.

<child>: Teach me about resilience.
"""
response = get_completion(prompt)
print(response)
```

Give the model time to “think”

- Tactic 1: Specify the steps required to complete a task
- Tactic 2: Instruct the model to work out its own solution before rushing to a conclusion

Give the model time to “think”

- Tactic 1: Specify the steps required to complete a task

```
text = f"""
In a charming village, siblings Jack and Jill set out on \
a quest to fetch water from a hilltop \
well. As they climbed, singing joyfully, misfortune \
struck—Jack tripped on a stone and tumbled \
down the hill, with Jill following suit. \
Though slightly battered, the pair returned home to \
comforting embraces. Despite the mishap, \
their adventurous spirits remained undimmed, and they \
continued exploring with delight.
"""
# example 1
prompt_1 = f"""
Perform the following actions:
1 - Summarize the following text delimited by triple \
backticks with 1 sentence.
2 - Translate the summary into French.
3 - List each name in the French summary.
4 - Output a json object that contains the following \
keys: french_summary, num_names.

Separate your answers with line breaks.

Text:
```{text}```
"""
response = get_completion(prompt_1)
print("Completion for prompt 1:")
print(response)
```
```

Give the model time to “think”

- Tactic 2: Instruct the model to work out its own solution before rushing to a conclusion

```
prompt_2 = f"""
Your task is to perform the following actions:
1 - Summarize the following text delimited by
  <> with 1 sentence.
2 - Translate the summary into French.
3 - List each name in the French summary.
4 - Output a json object that contains the
  following keys: french_summary, num_names.

Use the following format:
Text: <text to summarize>
Summary: <summary>
Translation: <summary translation>
Names: <list of names in summary>
Output JSON: <json with summary and num_names>

Text: <{text}>
"""
response = get_completion(prompt_2)
print("\nCompletion for prompt 2:")
print(response)
```

Prompt patterns

- Prompt patterns document successful approaches for systematically engineering different output and interaction goals when working with conversational LLMs. (White et al., 2023)

- Fundamental contextual statements: written descriptions of the important ideas to communicate in a prompt to an LLM.

General prompt patterns

Pattern Category	Prompt Pattern
Input Semantics	*Meta Language Creation*
Output Customization	*Output Automater* *Persona* *Visualization Generator* *Recipe* *Template*
Error Identification	*Fact Check List* *Reflection*
Prompt Improvement	*Question Refinement* *Alternative Approaches* *Cognitive Verifier* *Refusal Breaker*
Interaction	*Flipped Interaction* *Game Play* *Infinite Generation*
Context Control	*Context Manager*

Example: Persona pattern

- Intent: To make the LLM output to take a certain point of view

- Fundamental contextual statements:
  - “Act as persona X”
  - “Provide outputs that persona X would create”

Example: Recipe pattern

- Intent: To provide a sequence of steps to create “ingredients” to achieve a stated goal.

- Fundamental contextual statements:
  - “I would like to achieve X”
  - “I know that I need to perform steps A,B,C”
  - “Provide a complete sequence of steps for me”
  - “Fill in any missing steps”
  - “Identify any unnecessary steps”

Example

“From now on, act as a security reviewer. Pay close attention to the security details of any code that we look at. Provide outputs that a security reviewer would regarding the code.”

Example

“I am trying to deploy an application to the cloud. I know that I need to install the necessary dependencies on a virtual machine for my application. I know that I need to sign up for an AWS account. Please provide a complete sequence of steps. Please fill in any missing steps. Please identify any unnecessary steps.”

Another example?

- The paper:
  - https://arxiv.org/abs/2302.11382

Pattern Category	Prompt Pattern
Input Semantics	*Meta Language Creation*
Output Customization	*Output Automater* *Persona* *Visualization Generator* *Recipe* *Template*
Error Identification	*Fact Check List* *Reflection*
Prompt Improvement	*Question Refinement* *Alternative Approaches* *Cognitive Verifier* *Refusal Breaker*
Interaction	*Flipped Interaction* *Game Play* *Infinite Generation*
Context Control	*Context Manager*

Prompt patterns for SE

Requirements Elicitation	Requirements Simulator Specification Disambiguation Change Request Simulation
System Design and Simulation	API Generator API Simulator Few-shot Example Generator Domain-Specific Language (DSL) Creation Architectural Possibilities
Code Quality	Code Clustering Intermediate Abstraction Principled Code Hidden Assumptions
Refactoring	Pseudo-code Refactoring Data-guided Refactoring

What are potential issues when applying GenAI for implementation?

We will talk about two problems:

- Coupling
- Reliability

Coupling

- Problem: code generated by LLMs might be coupled to specific libraries or frameworks
  - Consequence: difficult to maintain

- Solution: Intermediate Abstraction Pattern
  - Instruct the LLM to introduce an intermediate abstraction

Intermediate Abstraction – Contextual statements

1. If you write or refactor code with property X
2. that uses other code with property Y
3. (Optionally) Define property X
4. (Optionally) Define property Y
5. Insert an intermediate abstraction Z between X and Y
6. (Optionally) Abstraction Z should have these properties

Intermediate Abstraction - example

Whenever I ask you to write code, I want you to separate the business logic as much as possible from any underlying third-party libraries. Whenever business logic uses a third- party library, please write an intermediate abstraction that the business logic uses instead so the third-party library could be replaced with an alternate library if needed.

Let’s try

- In ChatGPT, run the following prompt:

“Create the code of a simple REST API implementing CRUD operations for manipulating a customer database”

- Then, re-run using the pattern

The Reliability issue.
Idea: Assured LLM-Based SE

- Paper from developers of Meta
- Generate-and-test approach

2024 IEEE/ACM 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering (InteNSE)

Assured LLM-Based Software Engineering

Nadia Alshahwan*
Mark Harman
Inna Harper
Alexandru Marginean
Shubho Sengupta
Eddy Wang
Meta Platforms Inc.,
Menlo Park, California, USA

ABSTRACT

In this paper we address the following question: How can we use Large Language Models (LLMs) to improve code independently of a human, while ensuring that the improved code

(1) does not regress the properties of the original code ?
(2) improves the original in a verifiable and measurable way ?

To address this question, we advocate Assured LLM-Based Software Engineering; a generate-and-test approach, inspired by Genetic Improvement. Assured LLMSE applies a series of semantic filters that discard code that fails to meet these twin guarantees. This overcomes the potential problem of LLM's propensity to hallucinate. It allows us to generate code using LLMs, independently of any human. The human plays the role only of final code reviewer, as they would do with code generated by other human engineers.

This paper is an outline of the content of the keynote by Mark Harman at the International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering, Monday 15th April 2024, Lisbon, Portugal.

1 INTRODUCTION

There has been a great deal of recent work on LLM–based code generation [7, 20] and on sub-areas such as Software Testing [35]. We use the term "LLM–based Software Engineering" [7] to name this rapidly-developing area of Software Engineering. LLM-based Software Engineering (LLMSE) is any application in which the software products and/or processes are based on the use of Large Language Models, in the same way that Search Based Software Engineering is any application in which the products and processes are based on computational search [16], and the way in which Component-based Software Engineering is based on components [32].

We distinguish between online and offline LLMSE. In online LLMSE, the results of LLM inferences are required in real time. For example, for code completion applications, such as CoPilot [41] and CodeCompose [28], where the results are provided directly in the Integrated Development Environment (IDE) editor as the software engineer interacts with it. The LLM provides suggested completions of the current code construct, method or fragment being edited in the IDE.

Tests are like rails…

- Tests provide constraints
- But also guide the execution

Hands-on: implementation & testing

Based on our paper on GenAI for TDD

Published at the AI for Agile Software Engineering Workshop @XP2024
URL: https://link.springer.com/chapter/10.1007/978-3-031-72781-8_3

Generative AI for Test Driven Development: Preliminary Results

Moritz Mock(✉), Jorge Melegati, and Barbara Russo
Free University of Bozen-Bolzano, 39100 Bolzano, Italy
{momock,jorge.melegati,brusso}@unibz.it

Abstract. Test Driven Development (TDD) is one of the major practices of Extreme Programming for which incremental testing and refactoring trigger the code development. TDD has limited adoption in the industry, as it requires more code to be developed and experienced developers. Generative AI (GenAI) may reduce the extra effort imposed by TDD. In this work, we introduce an approach to automatize TDD by embracing GenAI either in a collaborative interaction pattern in which developers create tests and supervise the AI generation during each iteration or a fully-automated pattern in which developers only supervise the AI generation at the end of the iterations. We run an exploratory experiment with ChatGPT in which the interaction patterns are compared with the non-AI TDD regarding test and code quality and development speed. Overall, we found that, for our experiment and settings, GenAI can be efficiently used in TDD, but it requires supervision of the quality of the produced code. In some cases, it can even mislead non-expert developers and propose solutions just for the sake of the query.

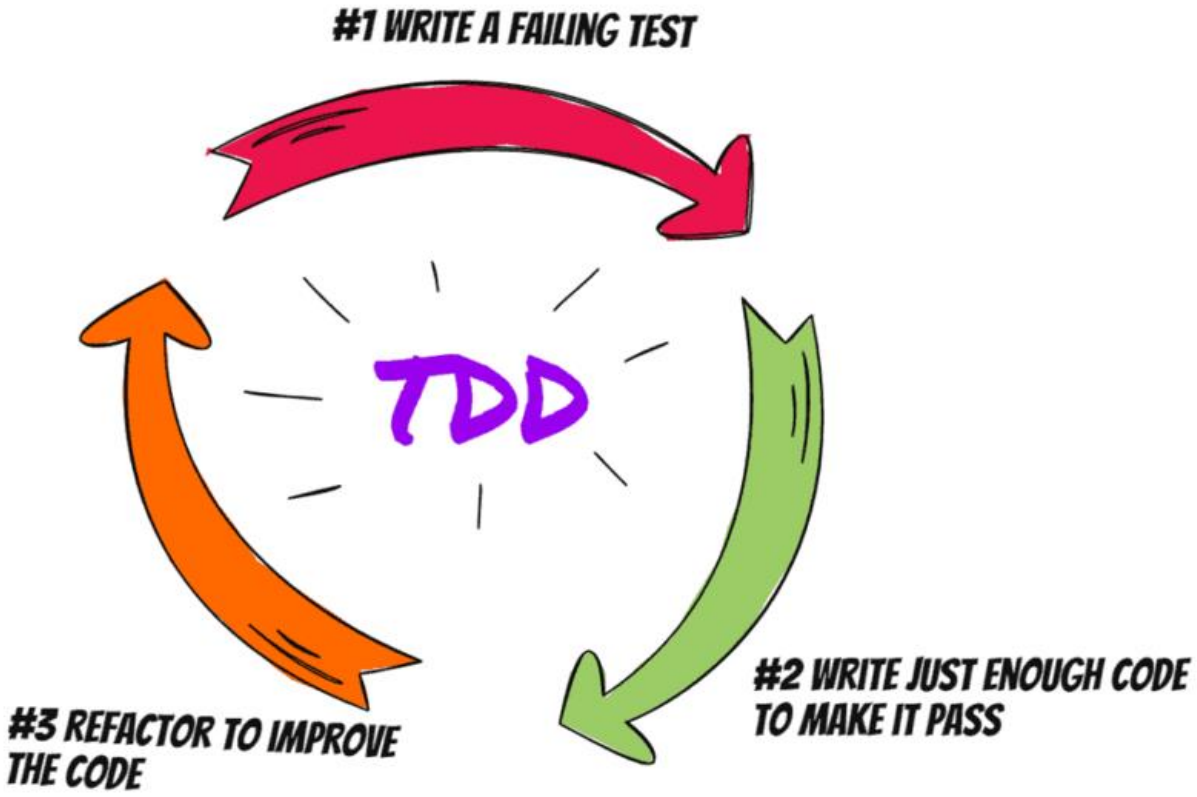
Keywords: AI4SE · Test Driven Development · Generative AI

1 Introduction

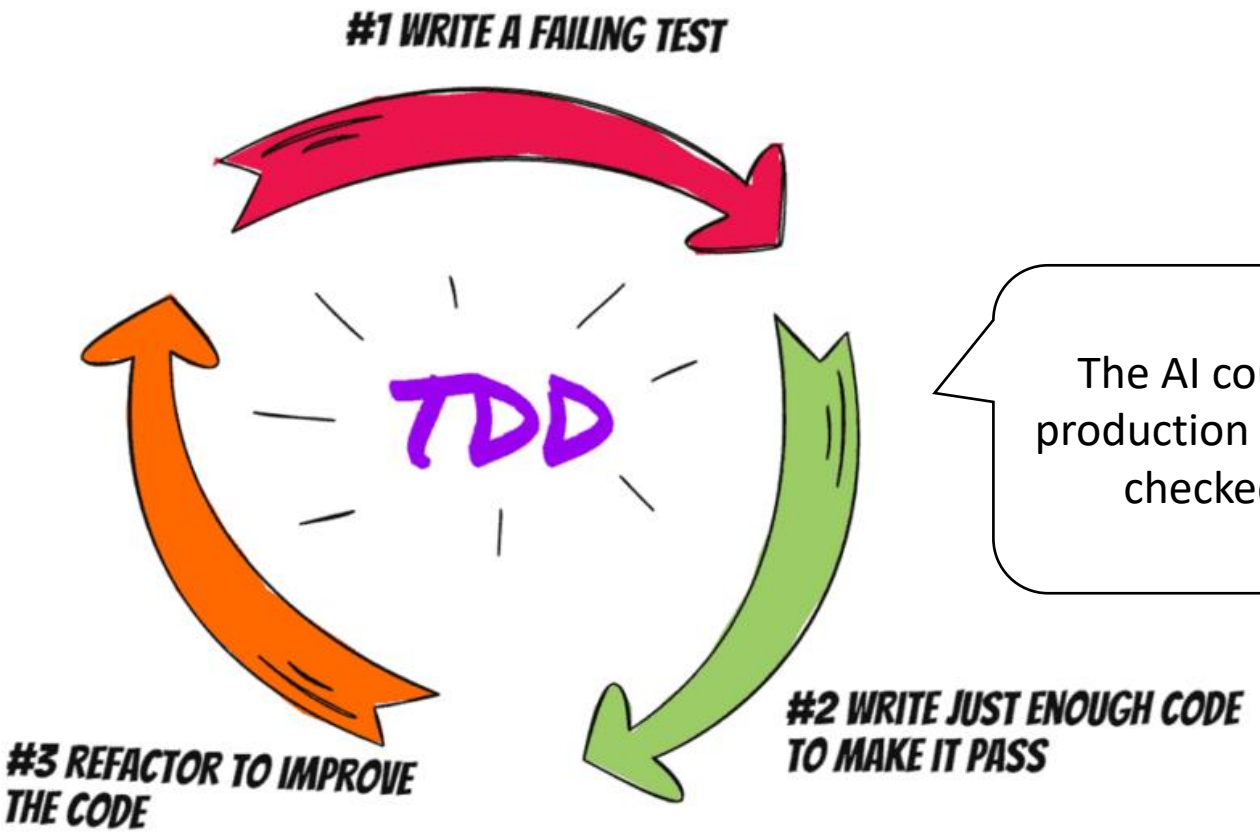
Test-driven development (TDD) is one of the major practices in Extreme Programming (XP) [1]. Nevertheless, its effectiveness is still controversial [7,9]. The major strength of TDD lies in its ability to deliver high-quality code through the granularity and uniformity of development [6]. To be effective, TDD developers must have a strong command of the practice and experience in development [3]. To facilitate its adoption, automation can be an option, and generative artificial intelligence (GenAI) tools, such as GitHub's Copilot[1] and OpenAI's ChatGPT, can be helpful [5]. In particular, recent literature has shown promising results in software testing. For instance, Piya and Sullivan [12] ... which a test suite is fed to ChatGPT, and ... upon test failures. Liang et al. [11] ...

[1] https://copilot...
[2] ...

- Automatic tests are useful for a generate-and-test approach
- Test-Driven Development (TDD) can be a good approach here
- TDD can guarantee that code will not be produced without tests



Developers can write the test code – which is simpler than production code.



The AI could generate the production code which will be checked by the tests.

Docker image

- For the next exercises, we will use a Docker image

- Run the following in a terminal

export OPENAI_API_KEY= <key>

docker run –e OPENAI_API_KEY –d melegati/levelup

- Once the container is running, connect your IDE to it

Exercise: implementing and testing

- The script levelup.tdd in Docker implements a code generator based on provided tests
- To run it, use:

  python –m levelup.tdd -c=<code-file> -t=<test-file>

- Task: implement using TDD a function that converts values in different metric units
  - Ex: convert(“2in”, “cm”) returns 5.08
  - Do not forget to import unittest and run it inside the main

Exercise: improving code (refactoring)

- The script levelup.refactor in Docker
  - refactors code
  - checks if the provided tests pass with the refactored code
  - checks if the refactored code is “better” than original code

- To run it, use:

  python –m levelup.refactor -c=<code-file> -t=<test-file>

- Task: use the script to refactor the code you created

Any ideas?

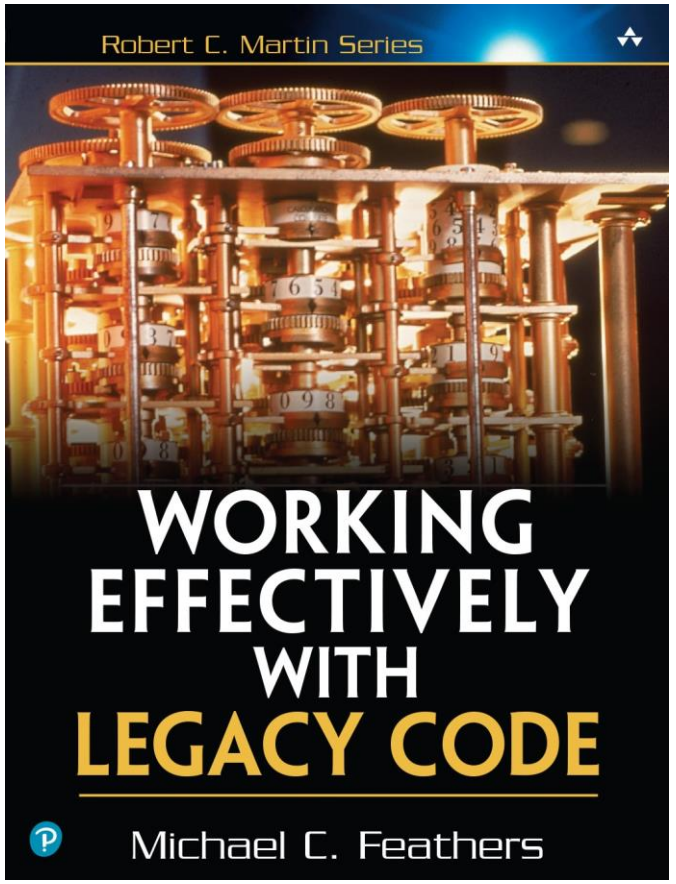
- So far, we used tests to guarantee that code is “correct”

- What else can we do in principle?

Legacy code

“To me, legacy code is simply code without tests.”

In principle, we can generate tests and check them with the code!



Other patterns

- Paper:
  - https://arxiv.org/pdf/2303.07839

Requirements Elicitation	Requirements Simulator Specification Disambiguation Change Request Simulation
System Design and Simulation	API Generator API Simulator Few-shot Example Generator Domain-Specific Language (DSL) Creation Architectural Possibilities
Code Quality	Code Clustering Intermediate Abstraction Principled Code Hidden Assumptions
Refactoring	Pseudo-code Refactoring Data-guided Refactoring

Examples of Prompt Engineering for software maintenance

- Fault localization
- Automatic program repair
- Code editing

Fault localization and program repair

Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs

Shengbei Jiang*
Beijing Jiaotong University
Beijing, China
21291232@bjtu.edu.cn

Jiabao Zhang*
Beijing Jiaotong University
Beijing, China
21281296@bjtu.edu.cn

Wei Chen*
Beijing Jiaotong University
Beijing, China
21281275@bjtu.edu.cn

Bo Wang†
Beijing Jiaotong University
Beijing, China
wangbo_cs@bjtu.edu.cn

Jianyi Zhou
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
zhoujianyi2@huawei.com

Jie M. Zhang
King's College London
London, United Kingdom
jie.zhang@kcl.ac.uk

ABSTRACT

Automated debugging is an emerging research field that aims to automatically find and repair bugs. In this field, Fault Localization (FL) and Automated Program Repair (APR) gain the most research efforts. Most recently, researchers have adopted pre-trained Large Language Models (LLMs) to facilitate FL and APR and their results are promising. However, the LLMs they used either vanished (such as Codex) or outdated (such as early versions of GPT). In this paper, we evaluate the performance of recent commercial closed-source general-purpose LLMs on FL and APR, i.e., ChatGPT 3.5, ERNIE Bot 3.5, and IFlytek Spark 2.0. We select three popular LLMs and evaluate them on 120 real-world Java bugs from the benchmark Defects4J. For FL and APR, we designed three kinds of prompts for each, considering different kinds of information. The results show that these LLMs could successfully locate 53.3% and correctly fix 12.5% of these bugs.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

Code (LLM4Code Workshop '24). ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Bugs are ubiquitous in modern software systems, threatening our daily lives. However, program debugging is time-consuming and challenging, consuming more than half of the developers' programming time [5]. Hence, a large body of research efforts have been dedicated to automated debugging techniques.

Among these debugging techniques, Fault Localization (FL) techniques and Automated Program Repair (APR) approaches are emerging fields and gaining traction. FL techniques aim to automatically localize buggy code elements (e.g., lines or methods) [9–11, 17], while APR techniques aim to automatically generate patches to fix buggy programs without human intervention [18–20, 23].

FL and APR techniques have adopted deep-learning models to facilitate the capture and comprehension of the context of the buggy code [10, 11, 17, 22], and deep-learning-based approaches have been recognized as the state of the art. However, their performance is still limited because their training data only contains buggy code

Fault localization experiments

- FL Prompt 1: src + "There is a bug in the above code, please help me locate it.”
- FL Prompt 2: src + stack + "There is a bug in the above code, please help me locate it by considering the stack trace.”
- FL Prompt 3: src + stack + assert + "There is a bug in the above code, please help me locate it by considering the stack trace information and failure assertion code."

Fault localization experiments - Results

- 120 bugs in Java from the Defects4J dataset
  - 6 open-source projects
- ChatGPT 3.5 detected 47 (39%) of them

Automated-Program Repair Experiment

- APR Prompt 1: src + "There is a bug in line X of the code, please help me fix it.”
- APR Prompt 2: src + "There is a bug in S, please help me fix it."
- APR Prompt 3: src + "There is a bug in the last statement, please help me fix it."

Automated-Program Repair - Results

- Same sample
- ChatGPT 3.5 solved 9 bugs (7,5%)

Code editing

Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions

Federico Cassano
Northeastern University
Boston, MA, USA

Luisa Li
Northeastern University
Boston, MA, USA

Akul Sethi
Northeastern University
Boston, MA, USA

Noah Shinn
Northeastern University
Boston, MA, USA

Abby Brennan-Jones
Wellesley College
Wellesley, MA, USA

Anton Lozhkov
Hugging Face
New York, NY, USA

Carolyn Jane Anderson
Wellesley College
Wellesley, MA, USA

Arjun Guha
Northeastern University and Roblox
Boston, MA, USA

Abstract

A significant amount of research is focused on developing and evaluating large language models for a variety of code synthesis tasks. These include synthesizing code from natural language instructions, synthesizing tests from code, and synthesizing explanations of code. In contrast, the behavior of instructional code editing with LLMs is understudied. These are tasks in which the model is instructed to update a block of code provided in a prompt. The editing instruction may ask for a feature to added or removed, describe a bug and ask for a fix, ask for a different kind of solution, or many other common code editing tasks.

We introduce a carefully crafted benchmark of code editing tasks and use it to evaluate several cutting edge LLMs. Our evaluation exposes a significant gap between the capabilities of state-of-the-art open and closed models. For example, even GPT-3.5-Turbo is 8.8% better than the best open model at editing code.

We also introduce a new, carefully curated, permissively licensed training set of code edits coupled with natural language instructions. Using this training set, we show that we can fine-tune open Code LLMs to significantly improve their code editing capabilities.

Instruction Provided to the Model

Edit the C4 class and its methods to represent the C8 group.

Code Diff Between Before and After Segments

```
-class C4(nn.Module):
+class C8(nn.Module):
-       """Represents the C4 class of group theory,
+       """Represents the C8 class of group theory,
        where each element represents a discrete rotation."""

    def __init__(self):
        super().__init__()

    def size(self):
        """Outputs the size of this group."""
-           return 4
+           return 8

    def elements(self):
        """Returns all the elements of this group"""
-           return torch.tensor([0., np.pi/2, np.pi, 3*np.pi/2])
+           d = np.pi / 4
+           return torch.tensor([0., d, d*2, d*3, d*4, d*5, d*6, d*7])
```

Figure 1: An abbreviated example of a code editing task from

Example of code editing prompt

```
## Code Before:
{before}
## Instruction:
{instruction}
## Code After:
```

(d) Prompt utilized for our fine-tuned EditCoder models as well as the baseline Deepseek-Coder-Base models.

````
<system>
You are PythonEditGPT. You will be
provided the original code snippet and
an instruction that specifies the changes
you need to make. You will produce the changed
code, based on the original code and the
instruction given. Only produce the code,
do not include any additional prose.
<user>
## Code Before
```py
def add(a, b):
    return a + b
```

## Instruction
Add a "sub" function that subtracts two numbers.
Also write docstrings for both functions and
change a,b to x,y.
<assistant>
## Code After
```py
def add(x, y):
    """Adds two numbers."""
    return x + y

def sub(x, y):
    """Subtracts two numbers."""
    return x - y
```
<user>
## Code Before
```py
{before}
```
## Instruction
{instruction}
````

(b) Conversation template utilized for all chat models with a 'system' prompt. The prompt is then adapted to the specific model chat format. This is the prompt utilized for: GPT-4, GPT-3.5-Turbo, CodeLlama-Instruct, and Deepseek-Coder-Instruct models.