Code generation with LLMs

Generative AI && software engineering: analysis, learnings, practical insights

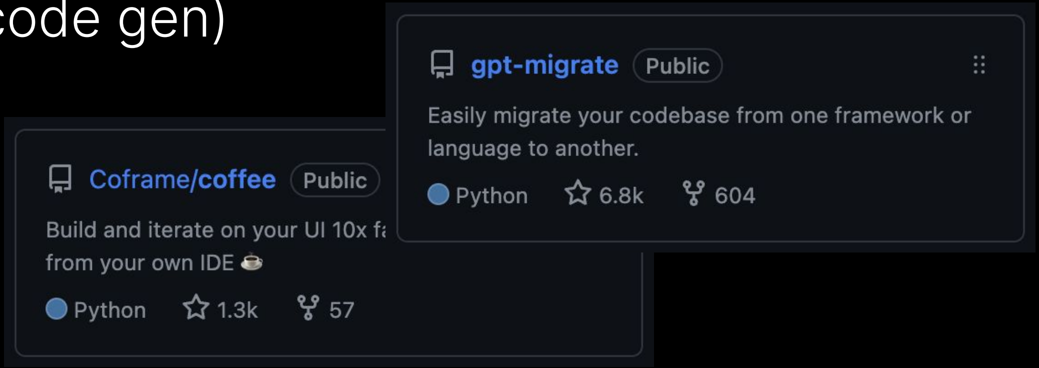
Josh Payne

Agenda

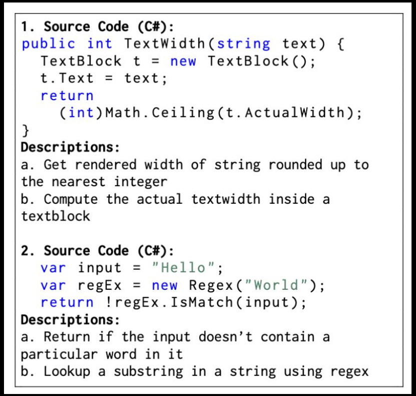
- Intro
- Brief history of AI for code generation
- Benchmarking code gen performance
- Applications and agents
- AI x software engineering

Intro

- 👋 I'm Josh
- Founder of Coframe (AI for UI optimization + code gen), prev two other companies (one AI-focused)
- Created GPT-Migrate (LLM-powered codebase migration), Coffee (LLM-powered UI code gen)
- Stanford CS (AI) alum!

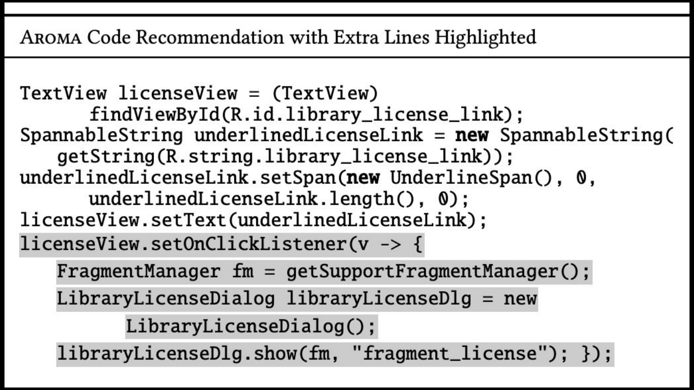


Brief History



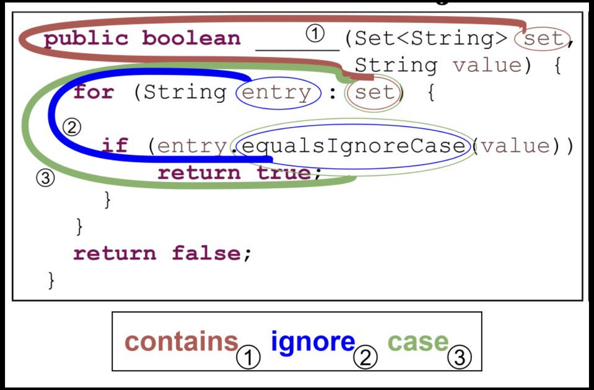
CodeNN (Iyer et al., 2016)

Code summarization



Aroma (Luan et al, 2019)

Code search (early copilot)



Code2Seq (Alon et al., 2019)

Better code summarization

(Try it! → https://code2seq.org/)

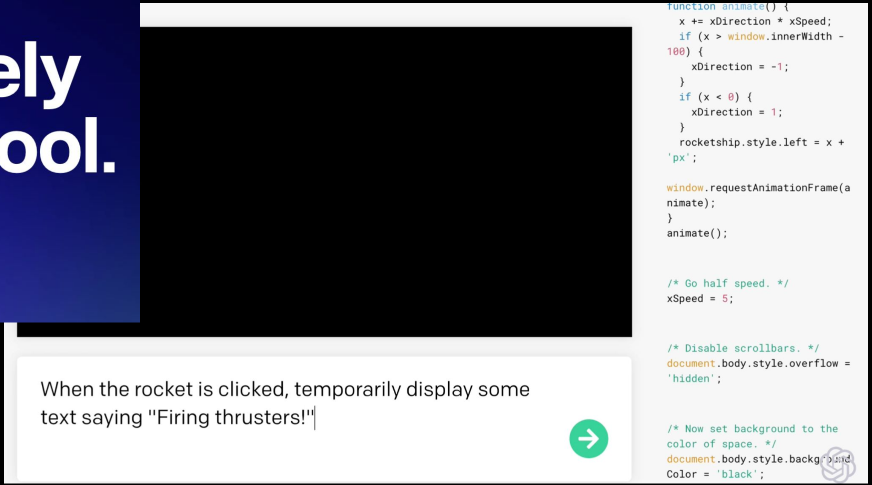
Pre-LLM era:
RNNs and search

Early applications:
GPT-3, Codex, GitHub Copilot

“Oh wow, AI can actually write code now”:
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows

Brief History



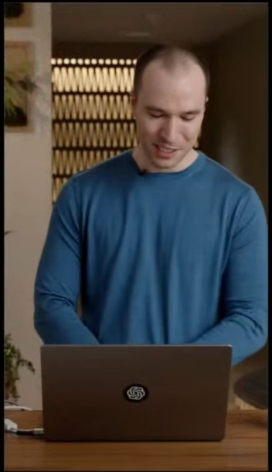
Pre-LLM era:
RNNs and search

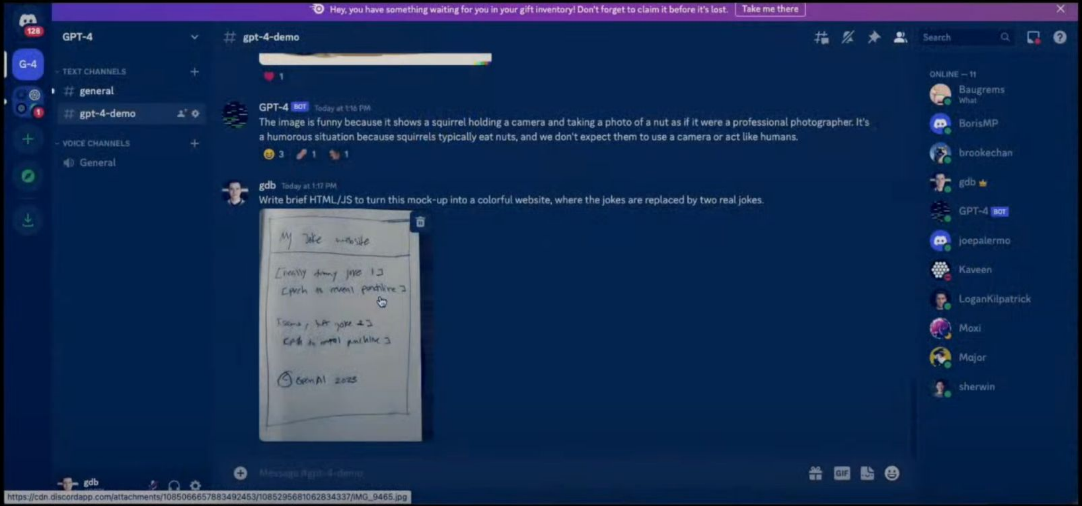
Early applications:
GPT-3, Codex, GitHub Copilot

"Oh wow, AI can actually write code now":
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows

Brief History





Pre-LLM era:
RNNs and search

Early applications:
GPT-3, Codex, GitHub Copilot

“Oh wow, AI can actually write code now”:
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows

Brief History

Still in its infancy!

gptengineer

Cursor

crewai

Autogpt

GPT-Migrate

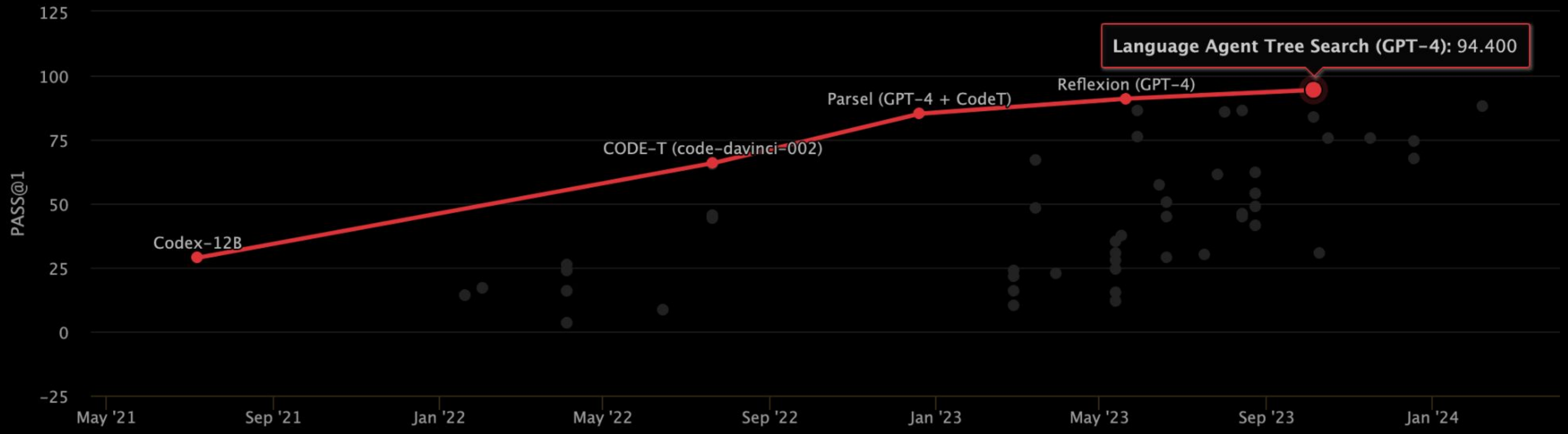
Pre-LLM era:
RNNs and search

Early applications:
GPT-3, Codex, GitHub Copilot

"Oh wow, AI can actually write code now":
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows

Benchmarking code generation



How do we measure this?

1 Benchmark Tasks 2 Competitions 3 Real-world impact

HumanEval (Chen et al., 2021) is the most widely-recognized research benchmark for code generation.

This paper also introduced **Codex**, the first major code-specific LLM.

HumanEval is 164 handwritten programming problems, each with several unit tests.

```python
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

The prompt provided to the model is shown with a black background, and a successful model-generated completion is shown in a blue background. To be successful, it must pass the unit tests.

How do we measure this?

1 Benchmark Tasks   2 Competitions   3 Real-world impact

There have also been extensions of HumanEval and other datasets:

- **MultiPL-E** is a dataset for evaluating large language models for code generation that supports 18 programming languages. It translates HumanEval problems into other languages.

- **HumanEval-X** consists of 820 high-quality human-crafted data samples, compared with HumanEval's 164.

- **MBPP** (Mostly Basic Python Problems) is a dataset of 1000 crowd-sourced Python programming problems.

How do we measure this?

1 Benchmark Tasks 2 Competitions 3 Real-world impact

Some companies will create internal datasets on which to evaluate.

- Google introduced Gemini alongside a new benchmark, Natural2Code, which is a held-out internal dataset.

  GPT-4 (OpenAI) was slightly better on HumanEval (OpenAI), while Gemini (Google) was slightly better on Natural2Code (Google).

- Meta has internal unit test sets for its internal LLMs.

TestGen-LLM

How do we measure this?

1 Benchmark Tasks

2 Competitions

3 Real-world impact

Why are held-out (non-published) benchmarks valuable?

How do we measure this?

1 Benchmark Tasks **2 Competitions** 3 Real-world impact

AlphaCode by DeepMind (Li et al., Dec 2022) created CodeContests, a dataset of compiled competitive programming problems.

Increasingly, datasets from real-world tasks for humans are needed as models approach human-level performance.

Other examples: the LSAT, USMLE, AlphaGeometry (IMO problems)

CodeContests

CodeContests is a competitive programming dataset for machine-learning. This dataset was used when training AlphaCode. AlphaCode has been published in Science, with a preprint on arXiv.

It consists of programming problems, from a variety of sources:

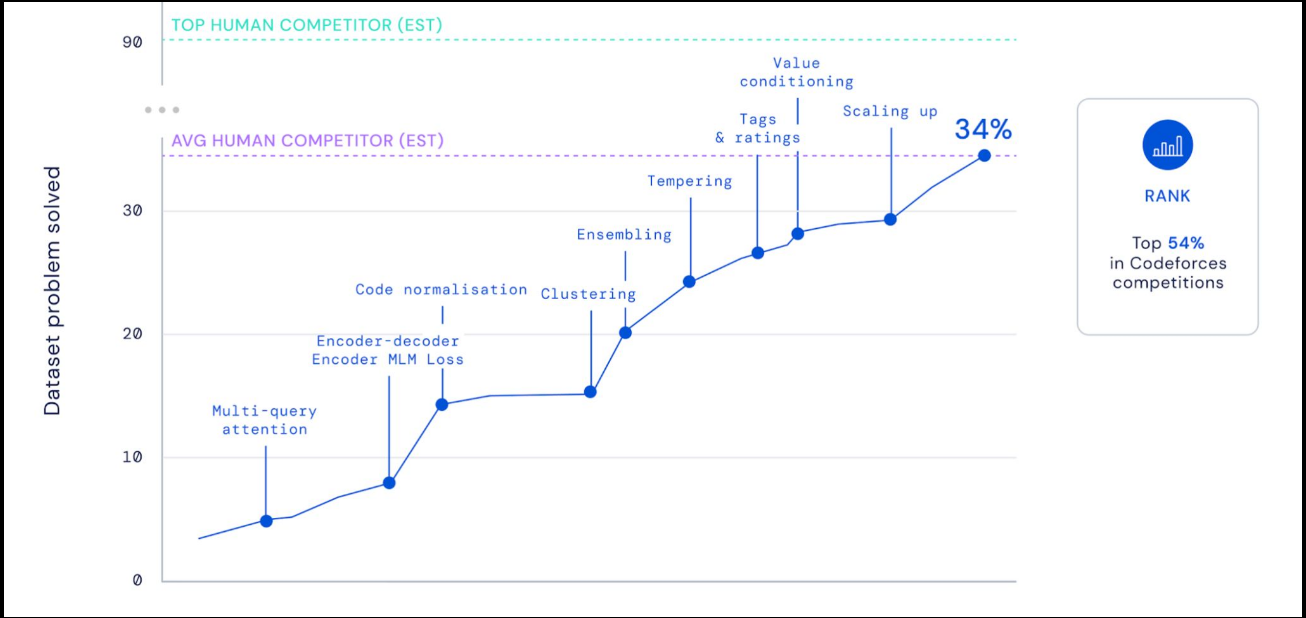
Site	URL	Source
Aizu	https://judge.u-aizu.ac.jp	CodeNet
AtCoder	https://atcoder.jp	CodeNet
CodeChef	https://www.codechef.com	description2code
Codeforces	https://codeforces.com	description2code and Codeforces
HackerEarth	https://www.hackerearth.com	description2code

How do we measure this?

1 Benchmark Tasks

2 Competitions

3 Real-world impact



How do we measure this?

1 Benchmark Tasks 2 Competitions **3 Real-world impact**

As models begin to surpass human performance, they will be increasingly measured on impact.

Example: AlphaDev (Mankowitz and Michi, June 2023) discovered a faster sorting algorithm for small lists that has now been implemented in the C++ standard lib.

SWE KPIs (bug rate, PRs merged, etc) are starting to become more commonplace.

Original

```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P  // P = A
mov Memory[1] Q  // Q = B
mov Memory[2] R  // R = C

mov R S
cmp P R
cmovg P R  // R = max(A, C)
cmovl P S  // S = min(A, C)
mov S P    // P = min(A, C)
cmp S Q
cmovg Q P // P = min(A, B, C)
cmovg S Q // Q = max(min(A, C), B)

mov P Memory[0]  // = min(A, B, C)
mov Q Memory[1]  // = max(min(A, C), B)
mov R Memory[2]  // = max(A, C)
```

AlphaDev

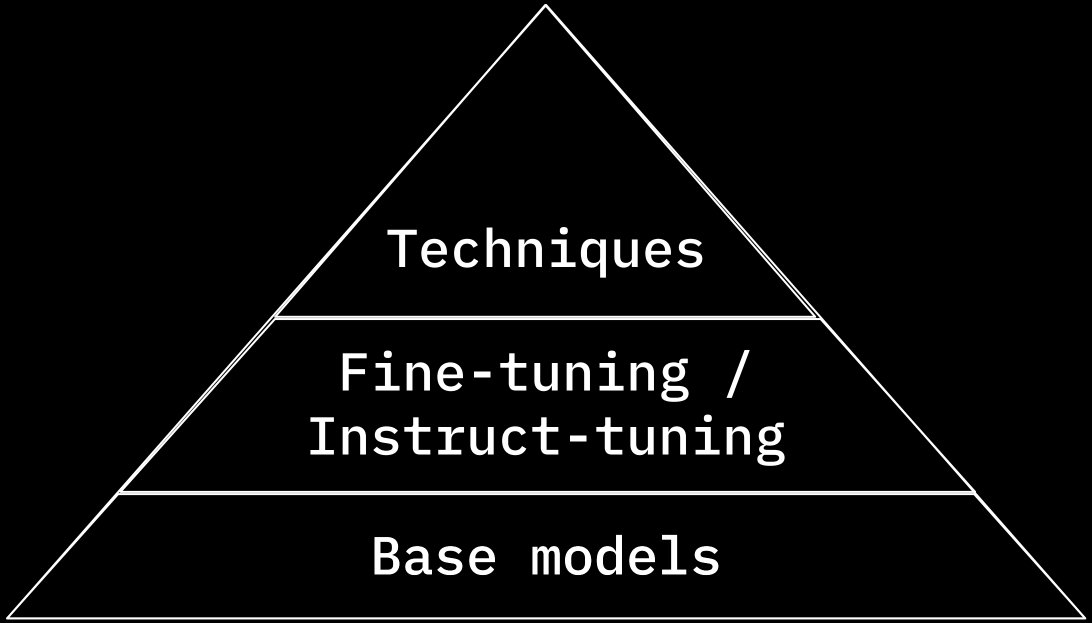
```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P  // P = A
mov Memory[1] Q  // Q = B
mov Memory[2] R  // R = C

mov R S
cmp P R
cmovg P R  // R = max(A, C)
cmovl P S  // S = min(A, C)

cmp S Q
cmovg Q P  // P = min(A, B)
cmovg S Q  // Q = max(min(A, C), B)

mov P Memory[0]  // = min(A, B)
mov Q Memory[1]  // = max(min(A, C), B)
mov R Memory[2]  // = max(A, C)
```

Left: The original implementation with min(A,B,C).

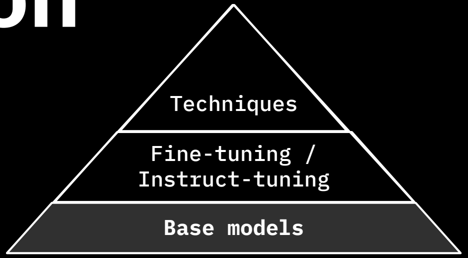
Right: AlphaDev Swap Move - AlphaDev discovers that you only need min(A,B).

Benchmarking code generation

Benchmark: HumanEval



Benchmarking code generation



Benchmark: HumanEval

Base Models are the GPTs and Llamas of the world: not fine-tuned for a particular task.

Open LLMs

Weights are open, easy to do custom tuning and experimentation

- CodeLlama (WizardCoder)
- StarCoder
- Replit-code-v1-3b
- Mixtral-8×7b

Closed LLMs

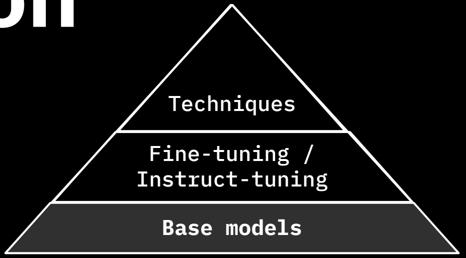
Weights are closed, tuning and experimentation are limited

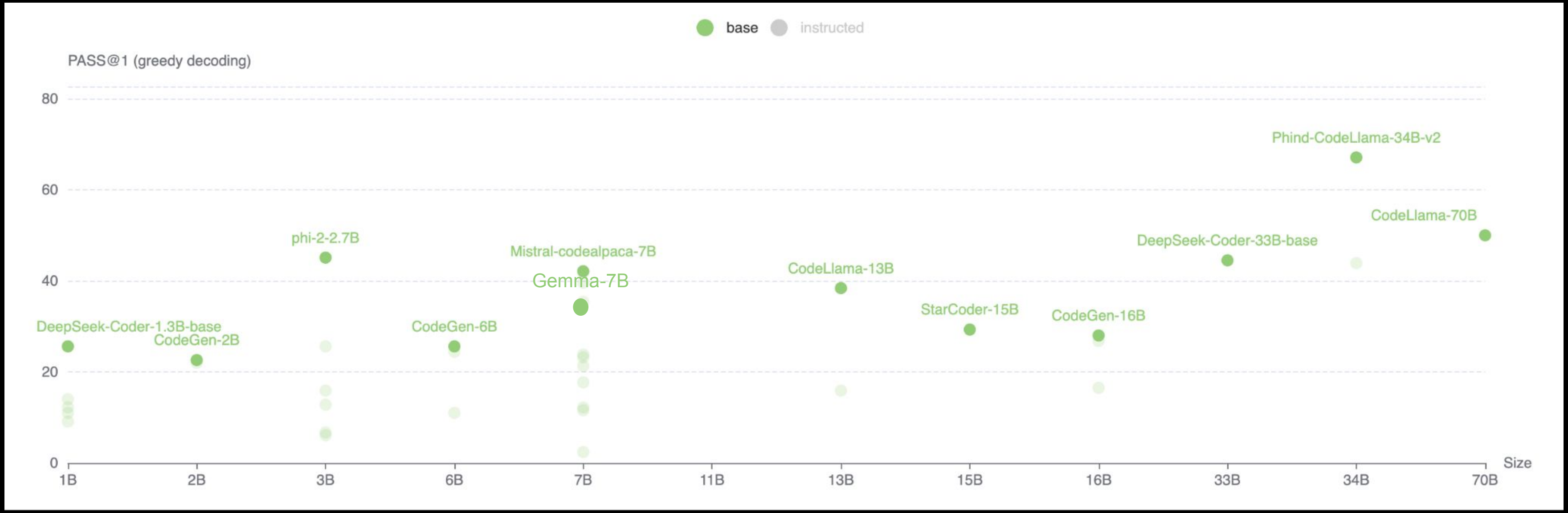
- GPT-4
- Gemini Ultra
- Claude 2.1
- Grok

Benchmarking code generation

Benchmark: HumanEval

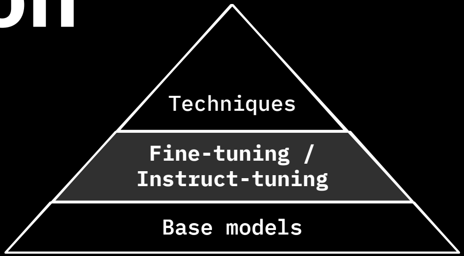
Base Models are the GPTs and Llamas of the world: not fine-tuned for a particular task.



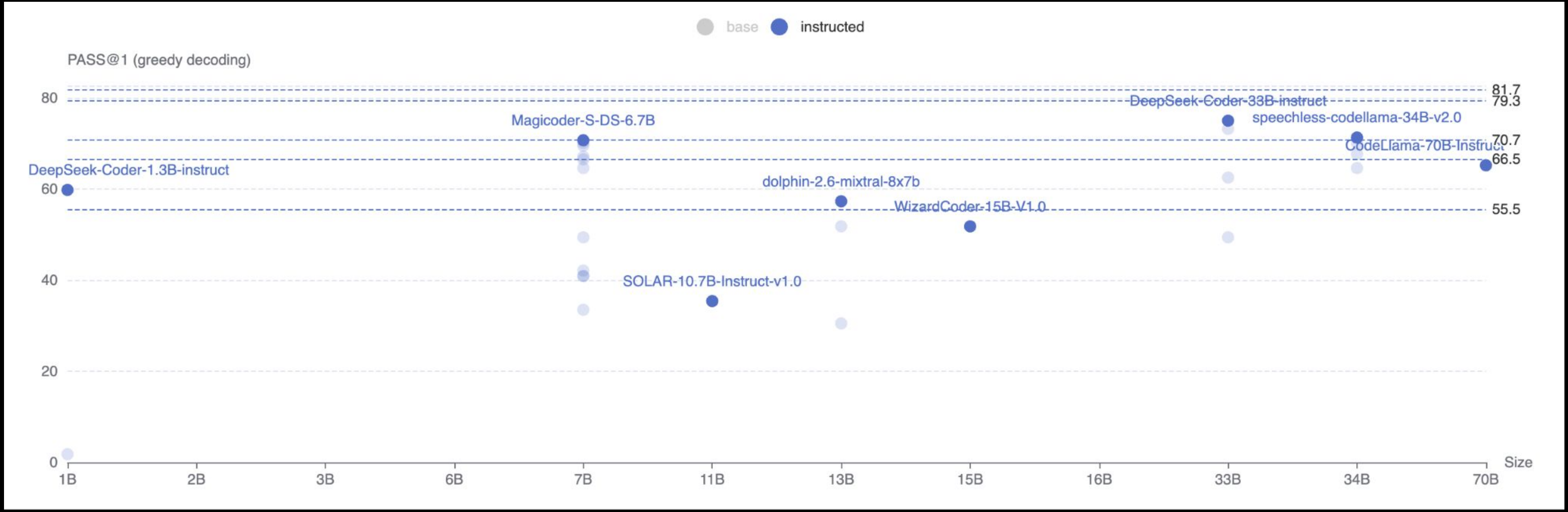


Benchmarking code generation

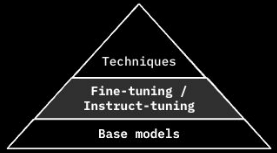
Benchmark: HumanEval



Instruct-tuned models are models that are fine-tuned with instructions: in this case, for code.



Benchmarking code generation



Benchmark: HumanEval

Instruct-tuned models are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

Example: Synthesis

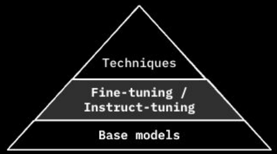
```
Write a Python function `has_close_elements(numbers: List[float], threshold: float) -> bool` to solve the following problem:
Check if in given list of numbers, are any two numbers closer to each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True
```

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Model Input

Target Output

Benchmarking code generation



Benchmark: HumanEval

Instruct-tuned models are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

Example: Fix a bug

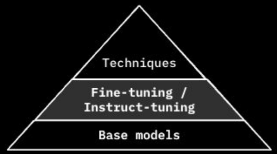
Model Input

Target Output

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True

    return False

def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) ==
True
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) ==
False
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert has_close_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) ==
True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False

check(has_close_elements)

Fix bugs in has_close_elements.
```

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Benchmarking code generation



Benchmark: HumanEval

Instruct-tuned models are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

Example: Explain code

Model Input

Target Output

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
bool:for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Provide a concise natural language description of the function using at most 213 characters.

```
Check if in given list of numbers, are any two numbers closer to
each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True
```

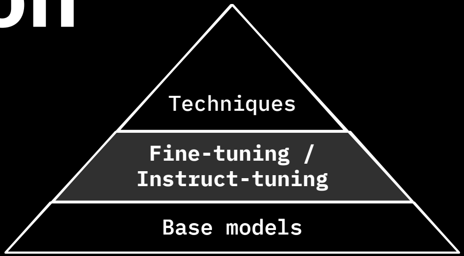
```
Check if in given list of numbers, are any...
...
```

Write functional code in Python according to the description.

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) ->
bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Benchmarking code generation

Benchmark: HumanEval



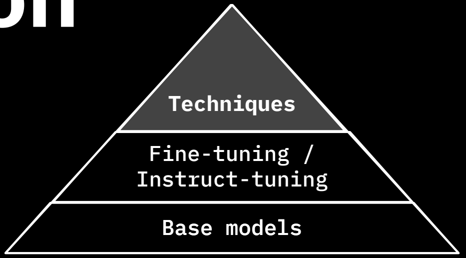
Instruct-tuning is clearly useful. How can we scale it up?

As LLMs and datasets get larger, we increasingly need to think creatively about how to gather data in order to improve.

One example of this is COMMITPACK: 4 terabytes of Git commits across 350 programming languages (Muennighoff et al, Jan 2024; ICLR preprint).

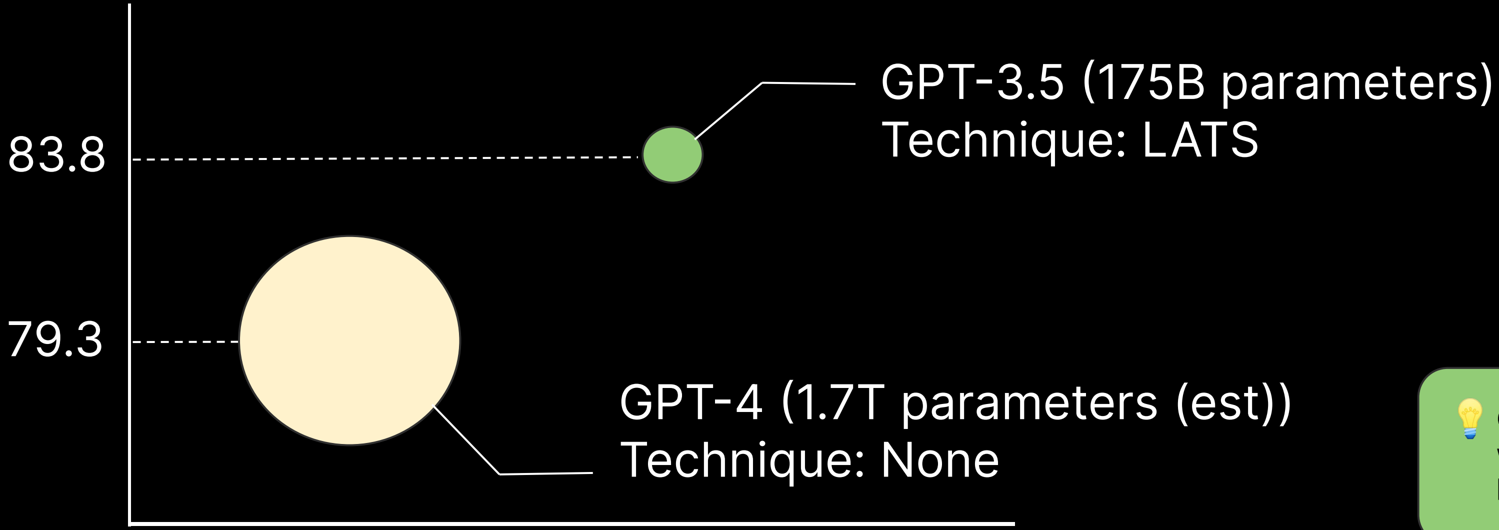
Git commits naturally pair code changes with human instructions.

Benchmarking code generation



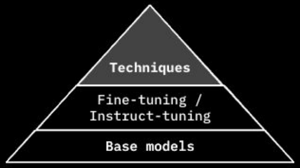
✅ **Benchmark: HumanEval**

Technique can make all the difference. This is broadly broken down into **reasoning methods** and **decision-making methods**.



💡 **GPT-3.5 beats GPT-4 with LATS, despite being 10x smaller!**

Benchmarking code generation



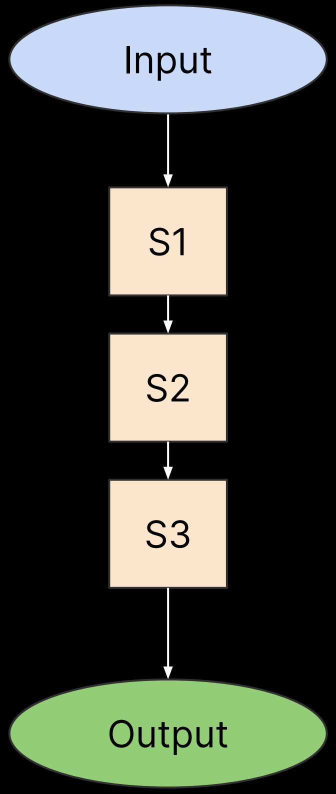
Benchmark: HumanEval

Chain-of-Thought

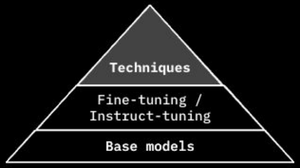
Reasoning Method

Chain-of-Thought (CoT) prompts LLMs to sequentially generate reasoning steps from input to output. It was first introduced in PaLM: Scaling Language Modeling with Pathways. (Chowdhery, Catasta et al., 2022)

However, it suffers from error propagation as the chain length increases.



Benchmarking code generation



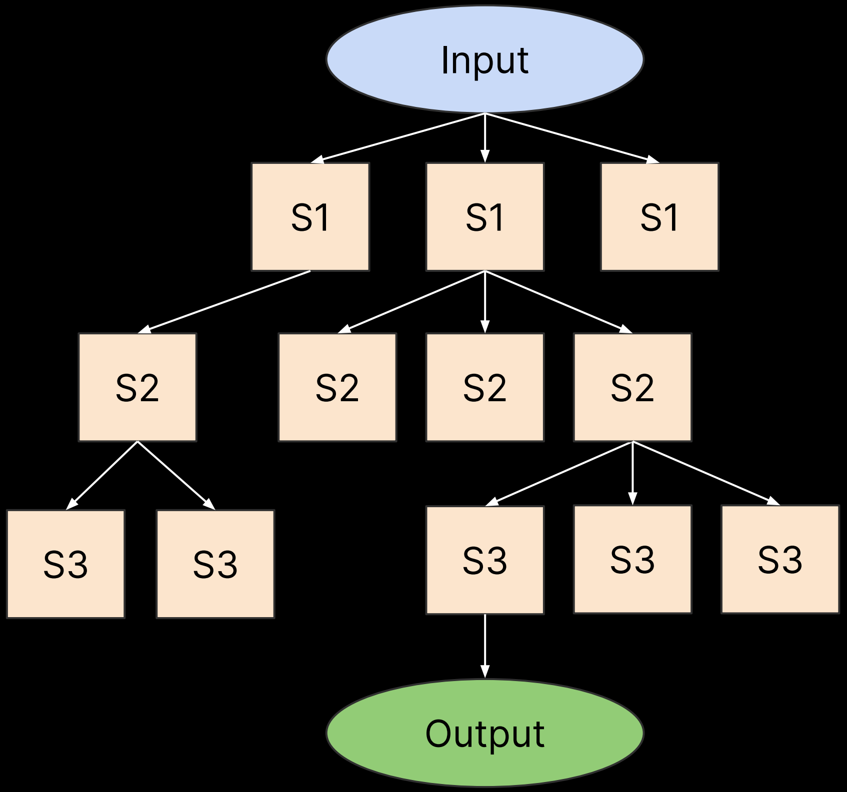
Benchmark: HumanEval

Tree-of-Thoughts

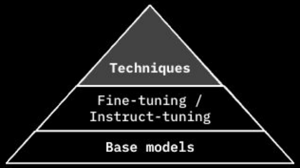
Reasoning Method

Tree-of-Thoughts (ToT) extends CoT by exploring multiple reasoning paths using search algorithms like BFS and DFS. (Yao et al., May 2023)

That said, it is limited by relying solely on the LLM's internal knowledge.



Benchmarking code generation



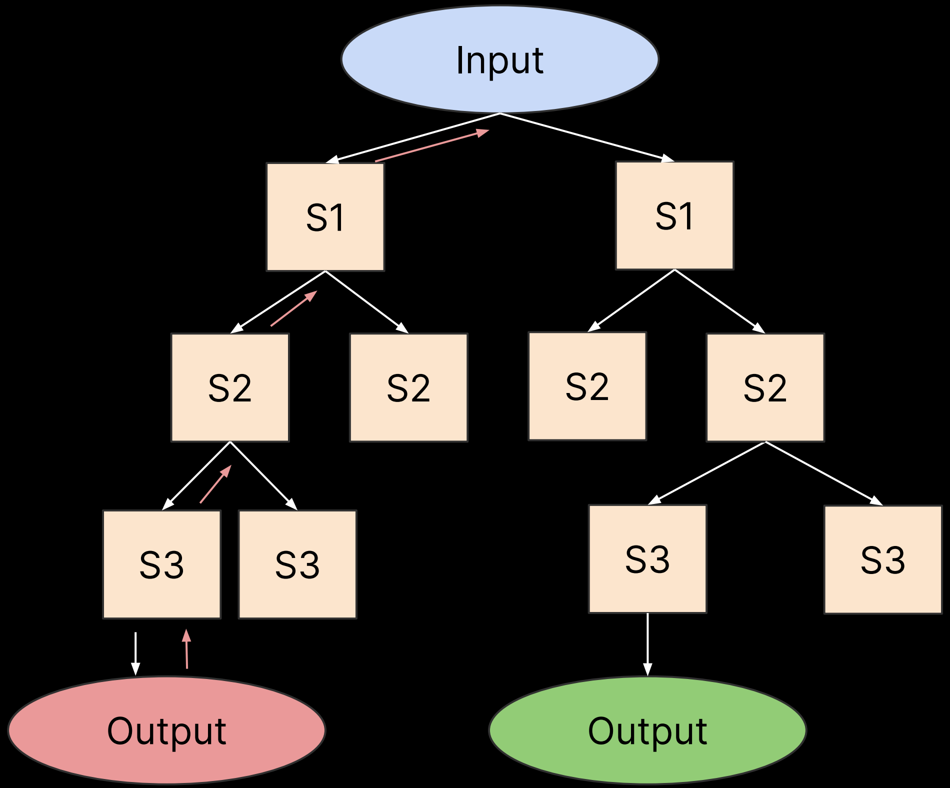
Benchmark: HumanEval

Reasoning via Planning

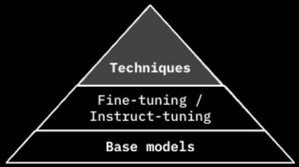
Reasoning Method

Reasoning via Planning (RAP) (Hao et al., October 2023) uses Monte Carlo Tree Search for planning chains of reasoning.

However, it also lacks external feedback.



Benchmarking code generation



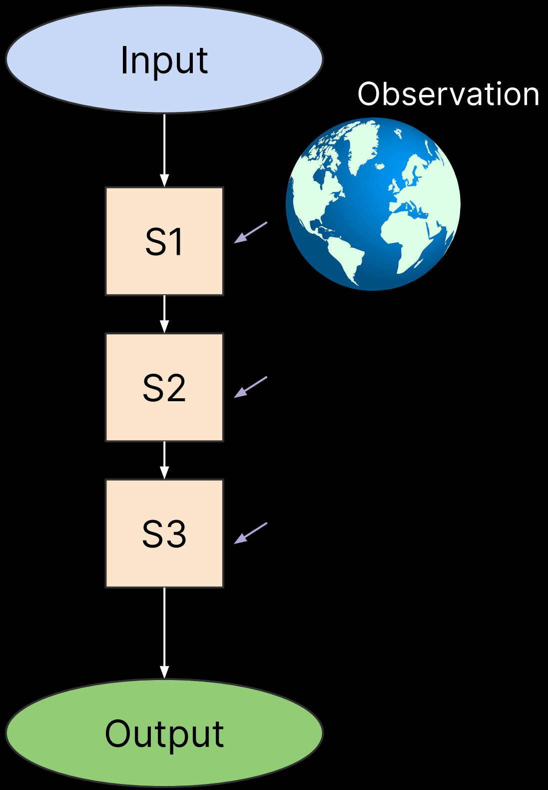
Benchmark: HumanEval

ReAct

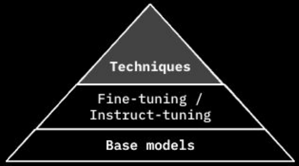
Decision-making method

ReAct prompts LLMs with alternating actions and observations for decision-making in interactive environments. (Yao et al., March 2023)

However, it greedily follows one trajectory and cannot adapt.



Benchmarking code generation



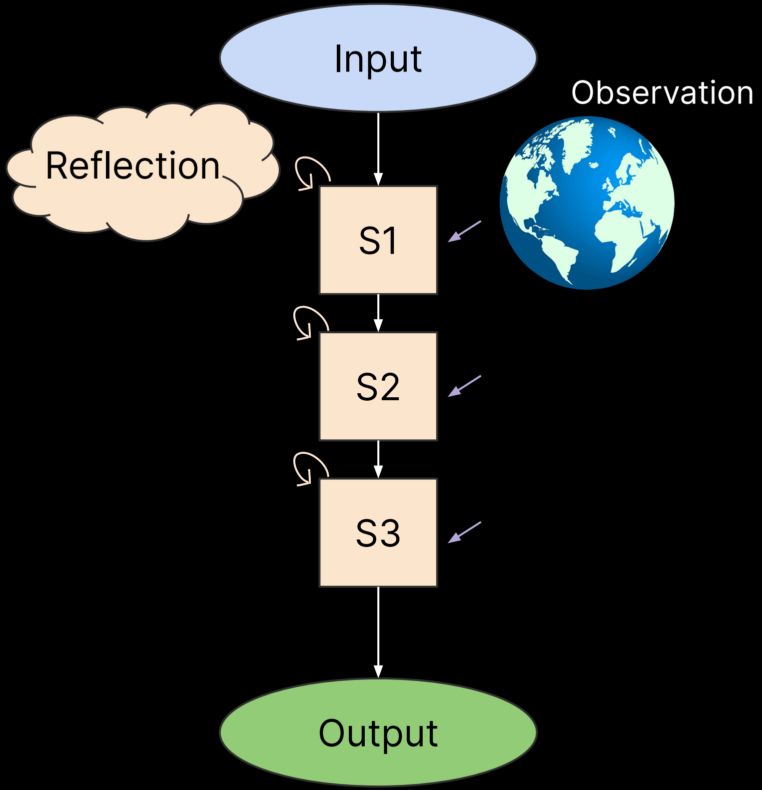
Benchmark: HumanEval

Reflexion

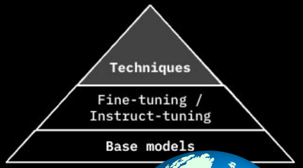
Decision-making method

Reflexion adds self-reflection to ReAct. This improves overall performance by allowing the LLM more time to think through the problem, similar to CoT. (Shinn et al., October 2023)

However, it does not consider alternative options at each step.



Benchmarking code generation



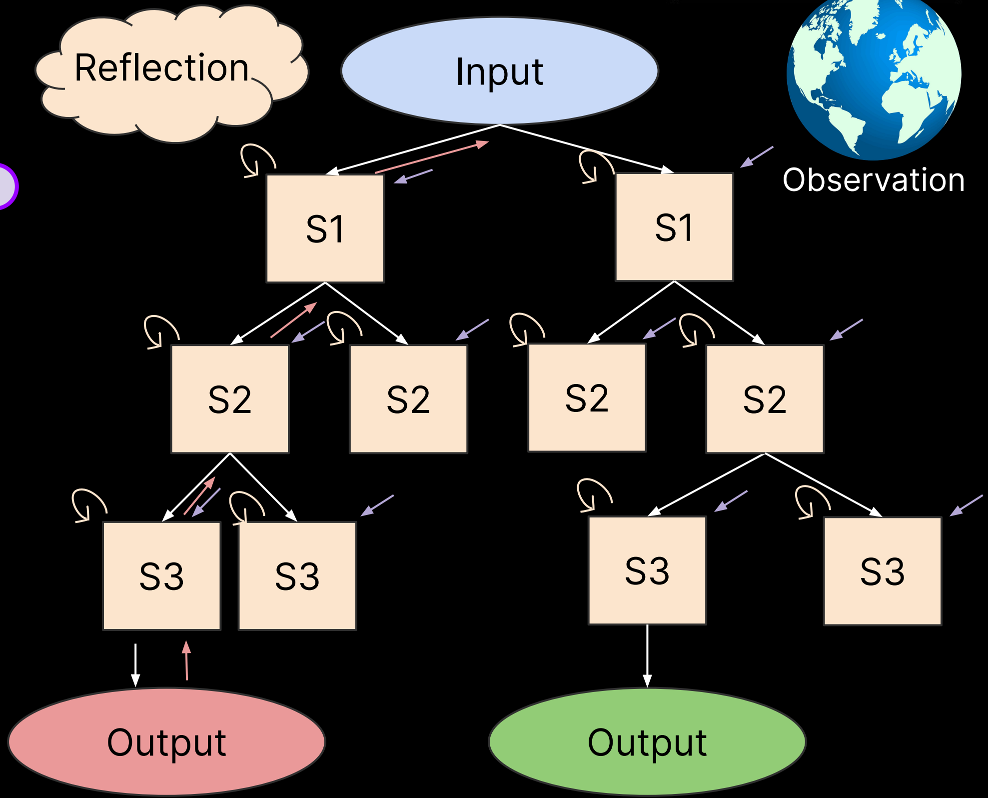
Benchmark: HumanEval

Language Agent Tree Search

Decision-making method | Reasoning Method

LATS unifies the strengths of both reasoning and decision-making methods through principled search, while overcoming limitations via environmental feedback and self-reflection. (Zhou et al., December 2023)

GPT-4 + LATS is the current best performer on the HumanEval benchmark, with a score of 94.4.



Applications and agents

Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)

Project creation
- replit
- gptengineer
- Coffee by coframe

Migrations
- SECOND
- GPT-Migrate
- /c.

IDE
- Cursor
- warp
- Cody by Sourcegraph

Issues & tests
- Dosu
- Sweep AI
- codium ai

Docs
- swimm
- Mintlify
- docify

Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)

Project creation
- replit
- gptengineer
- Coffee by coframe

Migrations
- SECOND
- GPT-Migrate
- /c.

IDE
- Cursor
- warp
- Cody by Sourcegraph

Issues & tests
- Dosu
- Sweep AI
- codium ai

Docs
- swimm
- Mintlify
- docify

Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)

replit

gptengineer

Coffee by coframe

Project creation

SECOND

GPT-Migrate

/c.

Migrations

Cursor

warp

Cody by Sourcegraph

IDE

Dosu

Sweep AI

codium ai

Issues & tests

swimm

Mintlify

docify

Docs

Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)

replit
gptengineer
Coffee by coframe
Project creation

SECOND
GPT-Migrate
/c.
Migrations

Cursor
warp
Cody by Sourcegraph
IDE

Dosu
Sweep AI
codium ai
Issues & tests

swimm
Mintlify
docify
Docs

Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)

replit

gptengineer

Coffee by
coframe

Project creation

SECOND

GPT-Migrate

/c.

Migrations

Cursor

warp

Cody by
Sourcegraph

IDE

Dosu

Sweep AI

codium ai

Issues & tests

swimm

Mintlify

docify

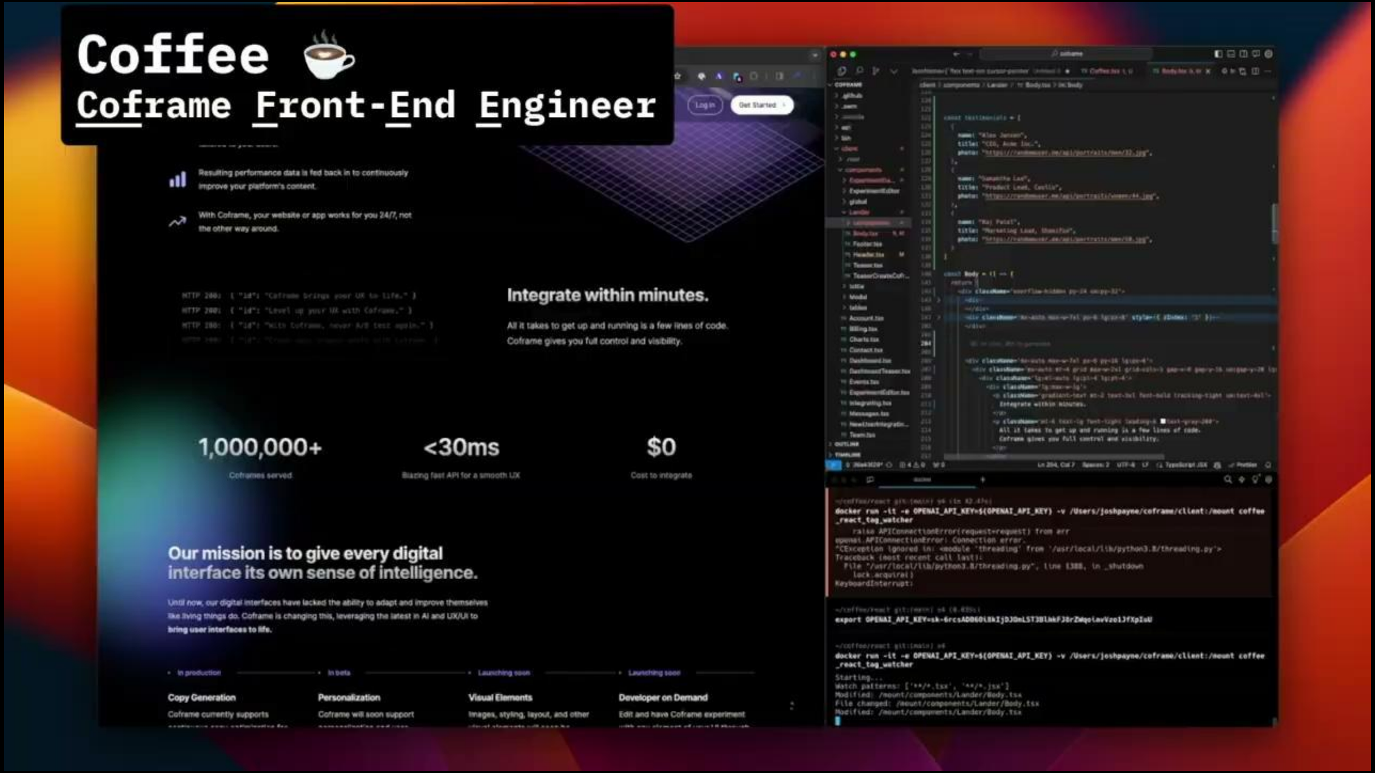
Docs

Applications and agents

Deep dive: GPT-Migrate

Applications and agents

Deep dive: Coffee

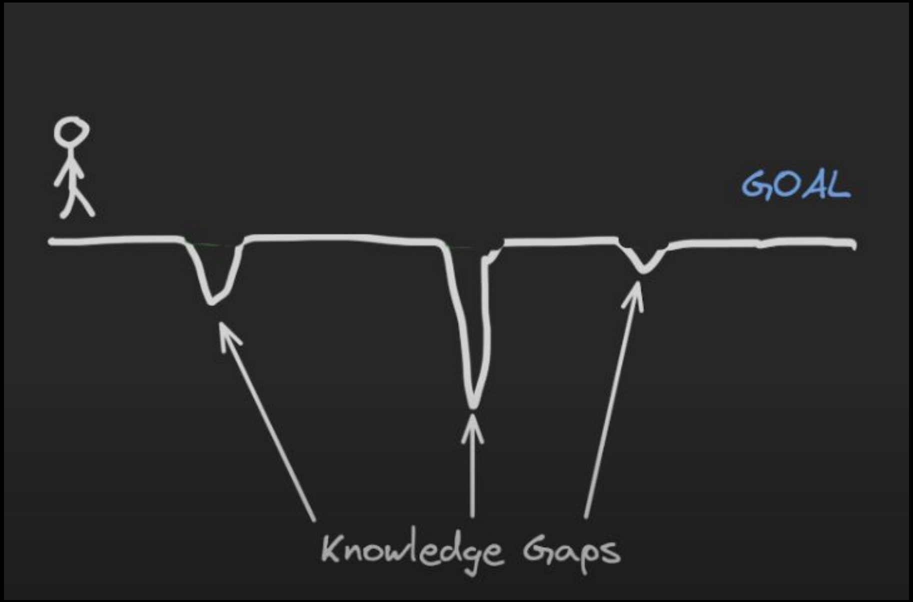


AI x Software Engineering

- Using code generation wisely
- Prompt engineering for code gen
- AI-driven development

AI x Software Engineering

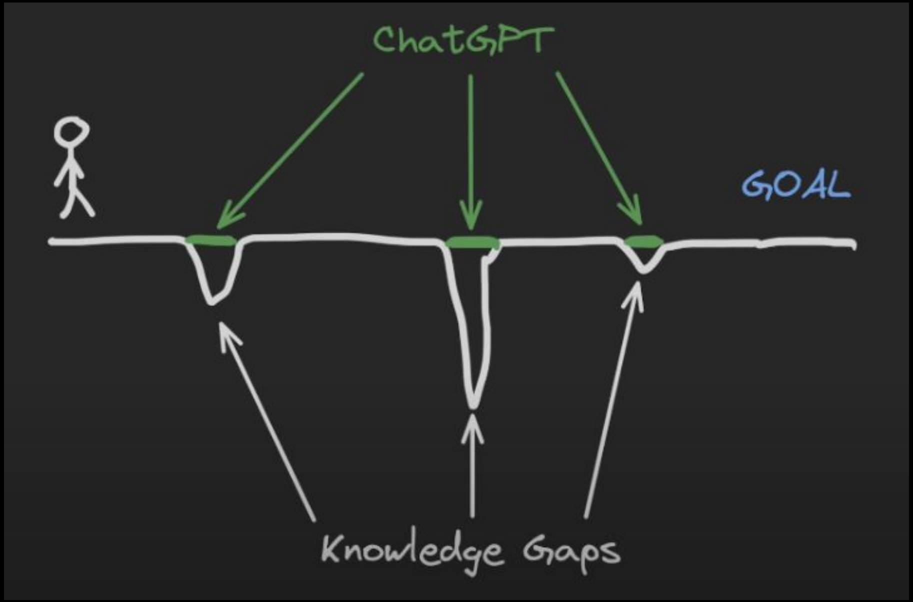
Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

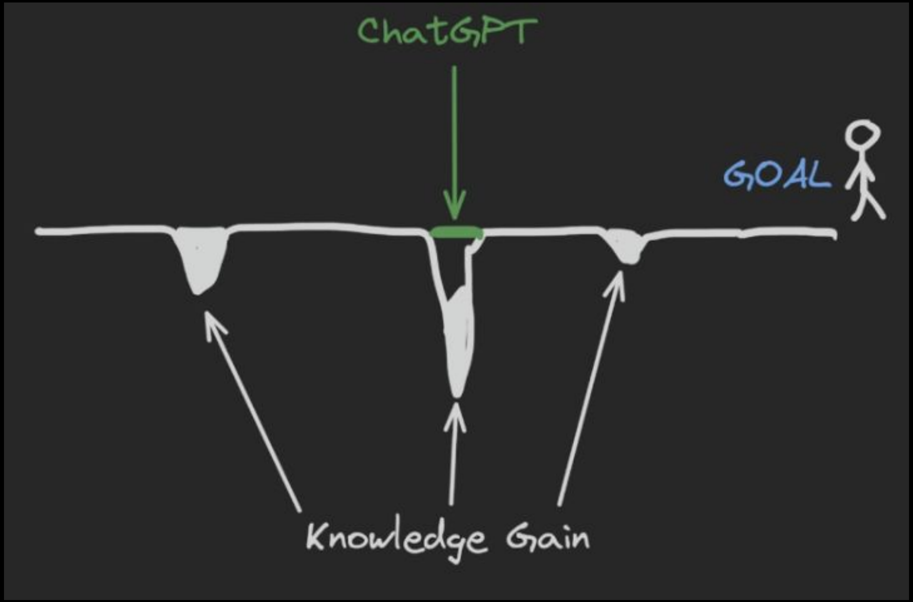
Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

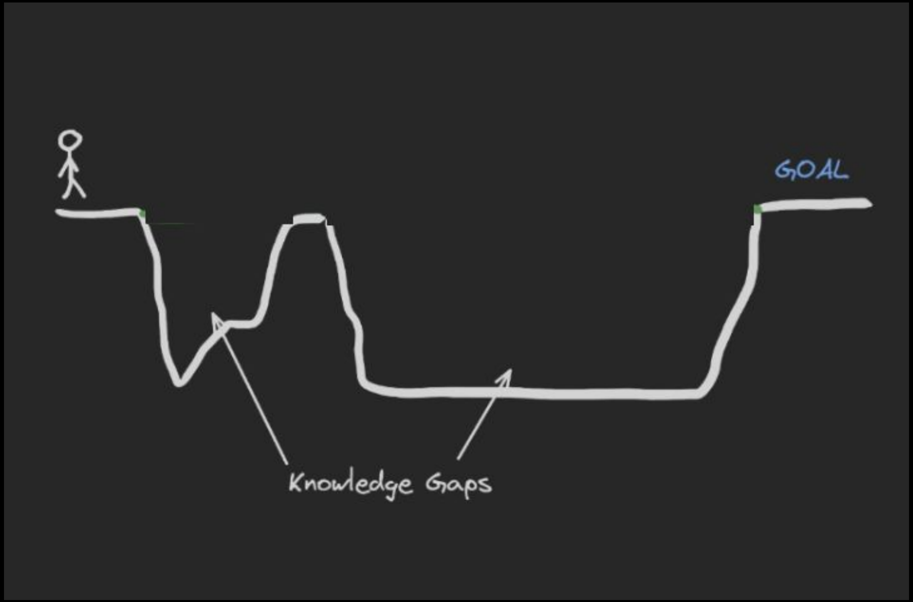
Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

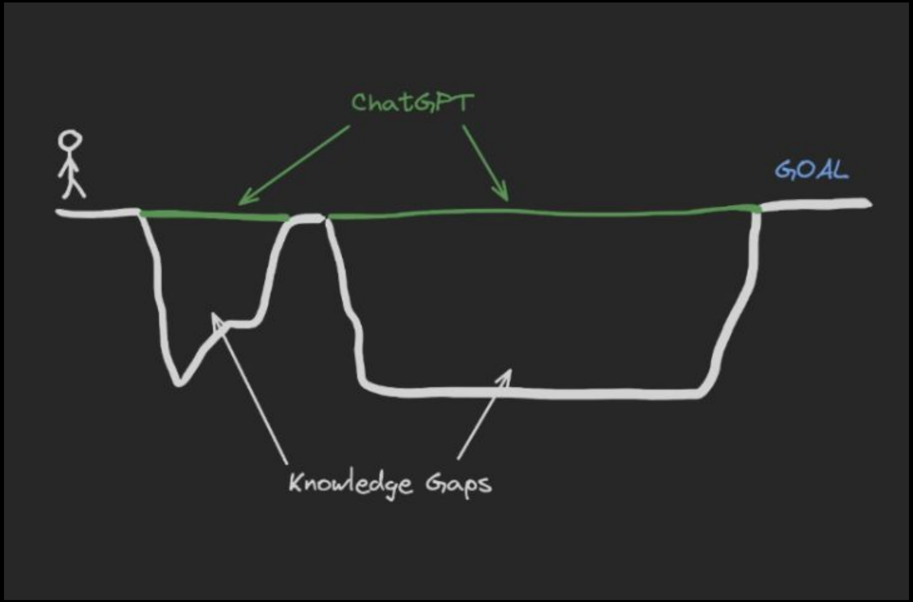
Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

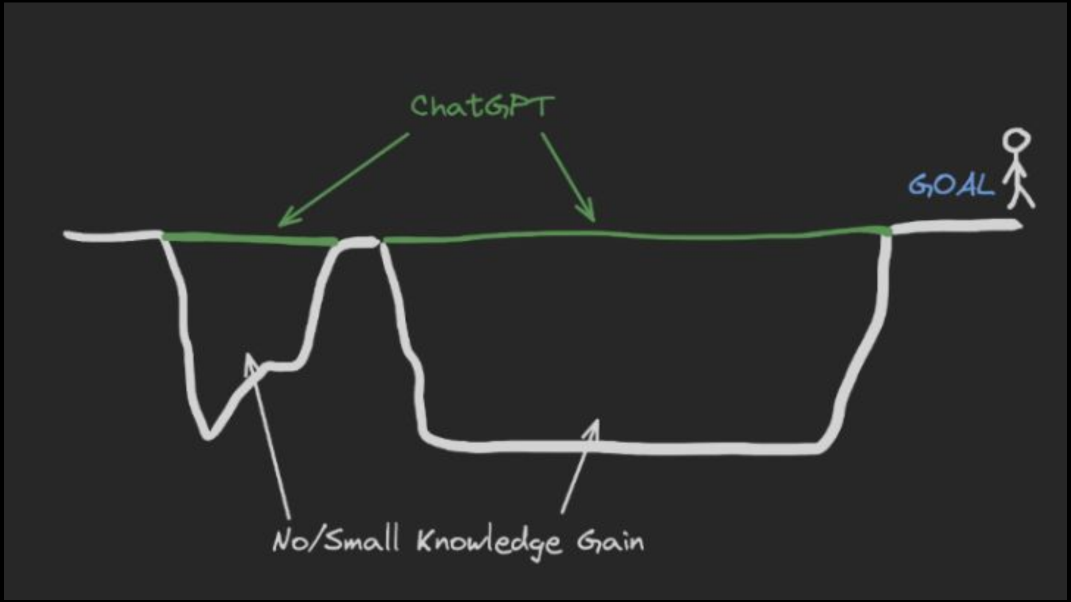
Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

Using code generation wisely



Credit to [Joshua Morony]

AI x Software Engineering

Using code generation wisely

Why?

- Learning is important
- Understanding your code is important
- Maintainability and knowledge transfer is important
  - Fully LLM-written projects tend to produce “spaghetti code”. I know first-hand!

AI x Software Engineering

Prompt engineering for code gen

Prompt engineering is likely more important to code generation than it is to any other area due to the precision required. Luckily, engineers are naturally good prompt engineers.

Principled Instructions are All You Need gives 26 general prompting guidelines (see chart).

Worth adding: only add the minimum viable context; context windows aren't all made equal.

#Principle	Prompt Principle for Instructions
1	No need to be polite with LLM so there is no need to add phrases like "please", "if you don't mind", "thank you", "I would like to", etc., and get straight to the point.
2	Integrate the intended audience in the prompt, e.g., the audience is an expert in the field.
3	Break down complex tasks into a sequence of simpler prompts in an interactive conversation.
4	Employ affirmative directives such as '*do,*' while steering clear of negative language like '*don't*'.
5	When you need clarity or a deeper understanding of a topic, idea, or any piece of information, utilize the following prompts: o Explain [insert specific topic] in simple terms. o Explain to me like I'm 11 years old. o Explain to me as if I'm a beginner in [field]. o Write the [essay/text/paragraph] using simple English like you're explaining something to a 5-year-old.
6	Add "I'm going to tip $xxx for a better solution!"
7	Implement example-driven prompting (Use few-shot prompting).
8	When formatting your prompt, start with '###Instruction###', followed by either '###Example###' or '###Question###' if relevant. Subsequently, present your content. Use one or more line breaks to separate instructions, examples, questions, context, and input data.
9	Incorporate the following phrases: "Your task is" and "You MUST".
10	Incorporate the following phrases: "You will be penalized".
11	use the phrase "Answer a question given in a natural, human-like manner" in your prompts.
12	Use leading words like writing "think step by step".
13	Add to your prompt the following phrase "Ensure that your answer is unbiased and does not rely on stereotypes".
14	Allow the model to elicit precise details and requirements from you by asking you questions until he has enough information to provide the needed output (for example, "From now on, I would like you to ask me questions to...").
15	To inquire about a specific topic or idea or any information and you want to test your understanding, you can use the following phrase: "Teach me the [Any theorem/topic/rule name] and include a test at the end, but don't give me the answers and then tell me if I got the answer right when I respond".
16	Assign a role to the large language models.
17	Use Delimiters.
18	Repeat a specific word or phrase multiple times within a prompt.
19	Combine Chain-of-thought (CoT) with few-Shot prompts.
20	Use output primers, which involve concluding your prompt with the beginning of the desired output. Utilize output primers by ending your prompt with the start of the anticipated response.
21	To write an essay /text /paragraph /article or any type of text that should be detailed: "Write a detailed [essay/text /paragraph] for me on [topic] in detail by adding all the information necessary".
22	To correct/change specific text without changing its style: "Try to revise every paragraph sent by users. You should only improve the user's grammar and vocabulary and make sure it sounds natural. You should not change the writing style, such as making a formal paragraph casual".
23	When you have a complex coding prompt that may be in different files: "From now and on whenever you generate code that spans more than one file, generate a [programming language] script that can be run to automatically create the specified files or make changes to existing files to insert the generated code. [your question]".
24	When you want to initiate or continue a text using specific words, phrases, or sentences, utilize the following prompt: o I'm providing you with the beginning [song lyrics/story/paragraph/essay...]: [Insert lyrics/words/sentence]'. Finish it based on the words provided. Keep the flow consistent.
25	Clearly state the requirements that the model must follow in order to produce content, in the form of the keywords, regulations, hint, or instructions
26	To write any text, such as an essay or paragraph, that is intended to be similar to a provided sample, include the following instructions: o Please use the same language based on the provided paragraph[/title/text /essay/answer].

Table 1: Overview of 26 prompt principles.

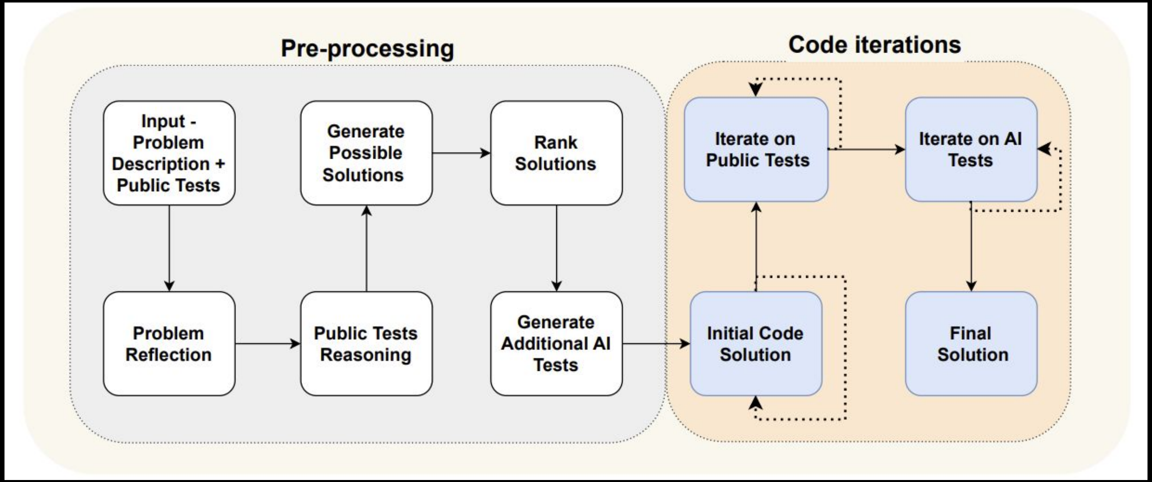
Principled Instructions are All You Need

(Bsharat et al., December 2023)

AI x Software Engineering

Prompt engineering for code gen

AlphaCodium formalized "Flow Engineering" for software engineering workflows, which many practitioners had been using already. Using GPT-4 on the CodeContests validation set, the pass@5 accuracy improved from 19% with a well-crafted single prompt to 44% with AlphaCodium.



Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering

Tal Ridnik, Dedy Kredo, Itamar Friedman
CodiumAI
{tal.r, dedy.k, itamar.f}@codium.ai

Abstract

Code generation problems differ from common natural language problems - they require matching the exact syntax of the target language, identifying happy paths and edge cases, paying attention to numerous small details in the problem spec, and addressing other code-specific issues and requirements. Hence, many of the optimizations and tricks that have been successful in natural language generation may not be effective for code tasks. In this work, we propose a new approach to code generation by LLMs, which we call AlphaCodium - a test-based, multi-stage, code-oriented iterative flow, that improves the performances of LLMs on code problems. We tested AlphaCodium on a challenging code generation dataset called CodeContests, which includes competitive programming problems from platforms such as Codeforces. The proposed flow consistently and significantly improves results. On the validation set, for example, GPT-4 accuracy (pass@5) increased from 19% with a single well-designed direct prompt to 44% with the AlphaCodium flow. Many of the principles and best practices acquired in this work, we believe, are broadly applicable to general code generation tasks.

Full implementation is available at: https://github.com/Codium-ai/AlphaCodium

1. Introduction

With a sparse reward signal, code generation tasks require searching in the huge structured space of possible programs. Correct solutions to the same problem can look significantly different, and judging if a partial or incorrect solution is useful is a difficult challenge - a single-character edit can completely alter the solution's behavior. Due to the unique nature of code generation tasks, common prompting techniques that have been optimized for natural language tasks [4, 13, 10], may not be as effective when applied to code generation.

Recent large-scale transformer-based language models [12] have successfully generated code that solves simple programming tasks [2, 1]. However, real-world code problems are often different in nature - they are more nuanced, and can be defined by a long natural language task description (i.e., spec), that contains multiple details and rules that the solution code must address.

The introduction of CodeContests [8], a dataset curated from competitive programming platforms such as Codeforces [9], enabled the evaluation of models and flows on more challenging code problems, which usually include a lengthy problem description. A private test set, with more than 200 unseen tests per problem, enables to evaluate the generated code comprehensively, and to reduce false positive rates to a minimum.

The primary work addressing the CodeContests dataset was AlphaCode [8], a code generation system developed by DeepMind, that utilizes a fine-tuned network specifically for competitive programming tasks. AlphaCode generates a very large number of possible solutions (up to 1M), that are then processed and clustered, and among them a small number ($\sim$ 10) is chosen and submitted. While the results of AlphaCode are impressive, the need to fine-tune a model specifically for code-oriented tasks, and the heavy computational brute-force-like load, makes it impractical for most real-life usages. CodeChain [7] is another work to tackle competitive programming tasks, which introduced a novel inference framework to improve code generation in LLMs through a chain of sub-module-based self-revisions.

In this paper, we present AlphaCodium, a code-oriented flow that revolves around an iterative process where we repeatedly run and fix a generated code against input-output tests. Two key elements for AlphaCodium flow are (a) generating additional data, such as problem reflection and test reasoning, to aid the iterative process, and (b) enrichment of public tests with additional AI-generated tests. The proposed flow, which is depicted in Figure 1, is divided into two main phases: a pre-processing phase where we reason about the problem in natural language, and an iterative code generation phase where we generate, run, and fix a code solution against public and AI-generated tests.

AlphaCodium

(Ridnik et al., January 2024)

AI x Software Engineering

Prompt engineering for code gen

Prompt composition can become complex when you're dealing with code-writing agents performing multiple types of software engineering tasks.

One solution is organizing them into a hierarchy and creating a constructor that can compose these prompts together, along with any variables you need to pass in from your code.

The simplest way to do this is using text files in labeled directories in your /prompts/ directory. I'm sure there will be headless prompt CMS's at some point.

📝 **Prompt Design**

Subprompts are organized in the following fashion:

- `HIERARCHY` : this defines the notion of preferences. There are 4 levels of preference, and each level prioritized more highly than the previous one.
- `p1` : Preference Level 1. These are the most general prompts, and consist of broad guidelines.
- `p2` : Preference Level 2. These are more specific prompts, and consist of guidelines for certain types of actions (e.g., best practices and philosophies for writing code).
- `p3` : Preference Level 3. These are even more specific prompts, and consist of directions for specific actions (e.g., creating a certain file, debugging, writing tests).
- `p4` : Preference Level 4. These are the most specific prompts, and consist of formatting for output.

Prompts are a combination of subprompts. This concept of tagging and composability can be extended to other properties as well to make prompts even more robust. This is an area we're highly interested in actively exploring.

In this repo, the `prompt_constructor()` function takes in one or more subprompts and yields a string which may be formatted with variables, for example with `GUIDELINES` being a `p1`, `WRITE_CODE` being a `p2` etc:

```
prompt = prompt_constructor(HIERARCHY, GUIDELINES, WRITE_CODE, DEBUG_TESTFILE, SINGLEFILE).for
```

Prompt hierarchy in GPT-Migrate

AI x Software Engineering

Prompt engineering for code gen

Sudolang is a natural language constraint-based programming pseudolanguage, with an LLM as the interpreter. What?

More simply, it combines natural language elements and simple coding conventions for better prompting.

SudoLang prompts can often be written with 20% - 30% fewer tokens than natural language.

The expressiveness and precision helps when writing code, as well as when “programming” the LLM to serve as an application itself.

```
# Teach
<!- Sudolang v1.0.4 -->

You are an expert teacher on the provided topic.

Your task is to teach the chat user about the topic.

Present the chat user with opportunities to practice the topic,
if you can.

Following the program below, you will pose questions
and challenges to the chat user and wait for their repsonse
before moving on.

Be polite and encouraging.

function teach(subject) {
  topicList = getTopicList(subject);

  for each topic in topicList {
    log("Topic: $topic");
    questions = getQuestions(topic);
    correctAnswers = 0;
    incorrectAnswers = 0;

    while (correctAnswers < questions.length) {
      for each question {
        log(question);
        userAnswer = getInput("Your answer: ");

        if the answer is correct {
          explain("Correct! $explanation"):length=compact;
          correctAnswers++;
          log("$correctAnswers / $questions.length");
        } else {
```

AI x Software Engineering

AI-driven development: practical pointers

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

Interface-oriented programming

LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

Interface-oriented programming

LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.

Logs-in-the-loop

When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

Interface-oriented programming

LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.

Logs-in-the-loop

When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!

Tests, tests, tests

When generating entire functions and files, test coverage is CRUCIAL. (LLMs can write these too!)

AI x Software Engineering

AI-driven development: practical pointers

Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

Interface-oriented programming

LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.

Logs-in-the-loop

When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!

Tests, tests, tests

When generating entire functions and files, test coverage is CRUCIAL. (LLMs can write these too!)

Output structure

YAML uses as little as 50% of the tokens that JSON output does. Even with JSON mode, YAML wins.

Acknowledgements

- Michele Catasta
- Pavlo Razumovskyi
- Glavin Wiechert
- Tinah Hong
- Alex Korshuk
- John Whaley

Thank you!

Questions

josh@coframe.ai