



10-301/10-601 Introduction to Machine Learning

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Coding Agents + Significance Testing for ML

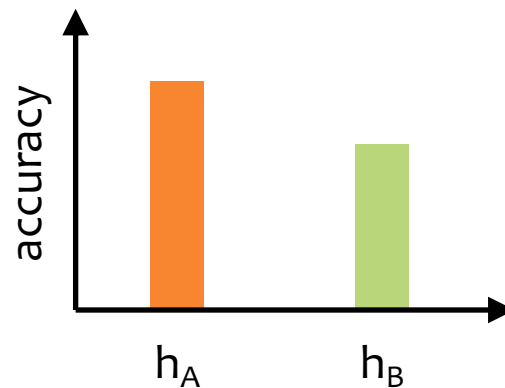
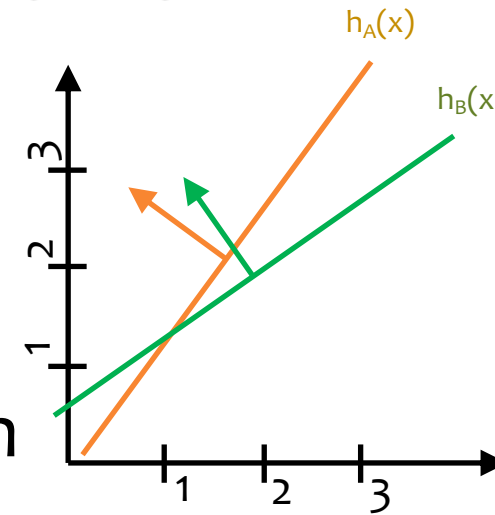
Matt Gormley
Lecture 26
Apr. 15, 2026

SIGNIFICANCE TESTING

Which classifier is better?

Goal: Given two classifiers: $h_A(x)$ and $h_B(x)$ which is better?

Common Approach: Evaluate each classifier on a test set and report which has higher accuracy.



Two Sources of Variance

1. Randomness in training
2. Randomness in our test data

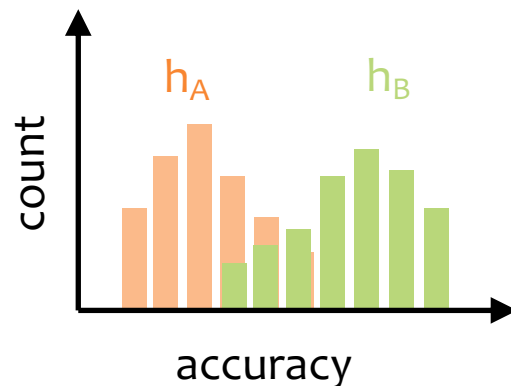
1. Randomness in training

Example: Assume we are training a **deep neural network** with a **nonconvex objective** function via **random restarts**

We collect a sequence of classifiers for R random restarts:

- ❖ $h_B(x)^{(1)} \leftarrow \text{train}(D, \text{seed} = \text{time in ms})$
- ❖ $h_B(x)^{(2)} \leftarrow \text{train}(D, \text{seed} = \text{time in ms})$
- ❖ ...
- ❖ $h_B(x)^{(R)} \leftarrow \text{train}(D, \text{seed} = \text{time in ms})$

Solution: histogram



Solution: confidence interval

report variance of h_A and h_B

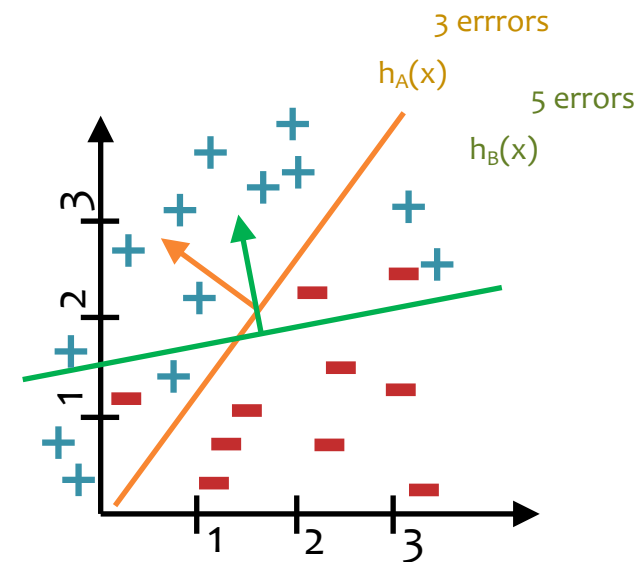
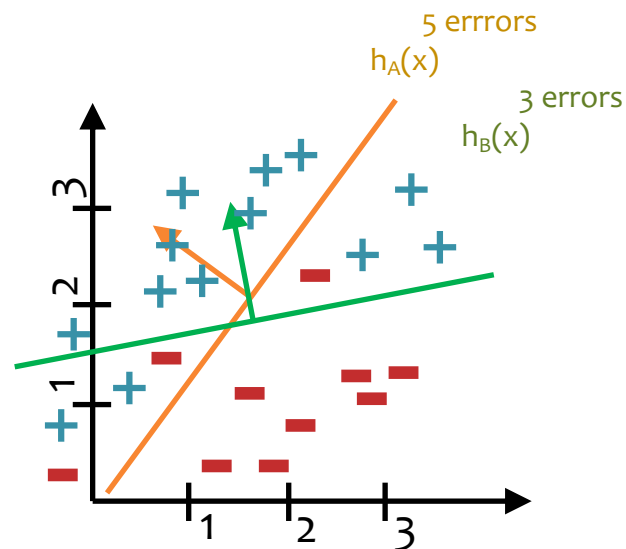
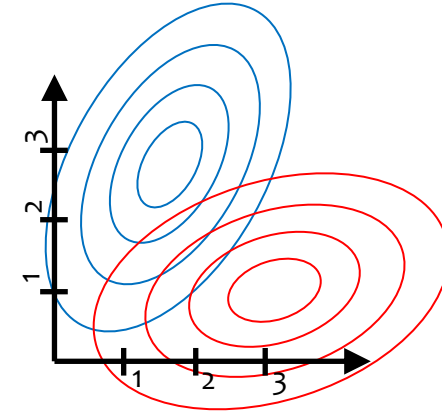
Ex:

- h_A 45% +/- 5%
- h_B 47% +/- 8%

2. Randomness in our test data

Recall: we assume $x^{(i)} \sim p^*(\cdot)$ and $y^{(i)} = c^*(x^{(i)})$
or $(x^{(i)}, y^{(i)}) \sim p^*(\cdot, \cdot)$

Data: Assume the data is drawn from a generative distribution $p^*(x|y)p^*(y)$ where $p^*(y)$ is an even coin flip and $p^*(x|y=\text{red})$ is the red Gaussian and $p^*(x|y=\text{blue})$ is the blue Gaussian.



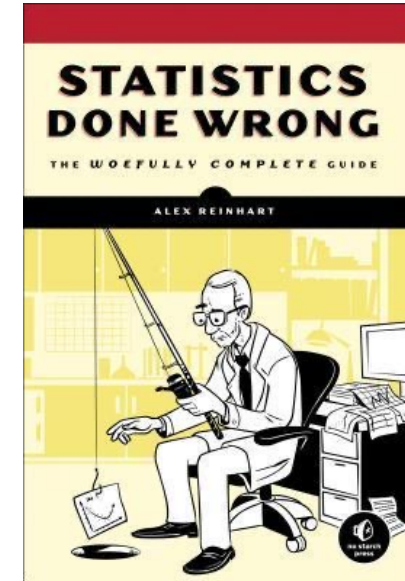
Solution:
significance testing

Significance Testing in ML

“And because any medication or intervention usually has some real effect, you can always get a statistically significant result by collecting so much data that you detect extremely tiny but relatively unimportant differences. As Bruce Thompson wrote, Statistical significance testing can involve a tautological logic in which tired researchers, having collected data on hundreds of subjects, then conduct a statistical test to evaluate whether there were a lot of subjects, which the researchers already know, because they collected the data and know they are tired. This tautology has created considerable damage as regards the cumulation of knowledge.”

— Alex Reinhart

Statistics Done Wrong: The Woefully Complete Guide



For machine learning, significance testing is usually still answering an important question:

Did we evaluate our model on enough test data to conclude that our improvement over the baseline is surprising?

Significance Testing in ML

Paired Bootstrap Test

Key Idea: simulate the resampling of many test sets

Algorithm:

1. Draw B bootstrap samples $S^{(b)} = \{(\mathbf{x}^{(1)}, y^{(1)}) (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$ with replacement from test data D_{test}
2. Let $v = 0$
3. For $b = 1, \dots, B$
if $\delta(S^{(b)}) > 2\delta(D_{\text{test}})$:
 $v = v + 1$
4. Return p-value as v/B

$\delta(D')$ = difference in accuracy between h_A and h_B on D'

H_0 = null hypothesis = performance of h_A and h_B is the same

Remarks:

1. Notice that $E[\delta(S^{(b)})] = \delta(D_{\text{test}})$.

We want to estimate how often A obtains a $\delta(D_{\text{test}})$ -sized advantage over B (or greater) by random chance.

So we check whether $\delta(S^{(b)})$ exceeds the expected value plus $\delta(D_{\text{test}}) = 2\delta(D_{\text{test}})$.

2. We needn't limit δ to the difference in accuracy, it can be any metric we want!

CODE GENERATION

How can you boost your productivity as a programmer?

Pair Programming



Coding with an LLM



Building Code Models

Languages for LLMs

- LLMs are trained on massive quantities of text from the internet (e.g. trillions of tokens)
- Some LLMs cover a variety of human languages, some focus primarily on English
- Most LLMs include a **wide variety of programming languages**

Programming Languages on GitHub

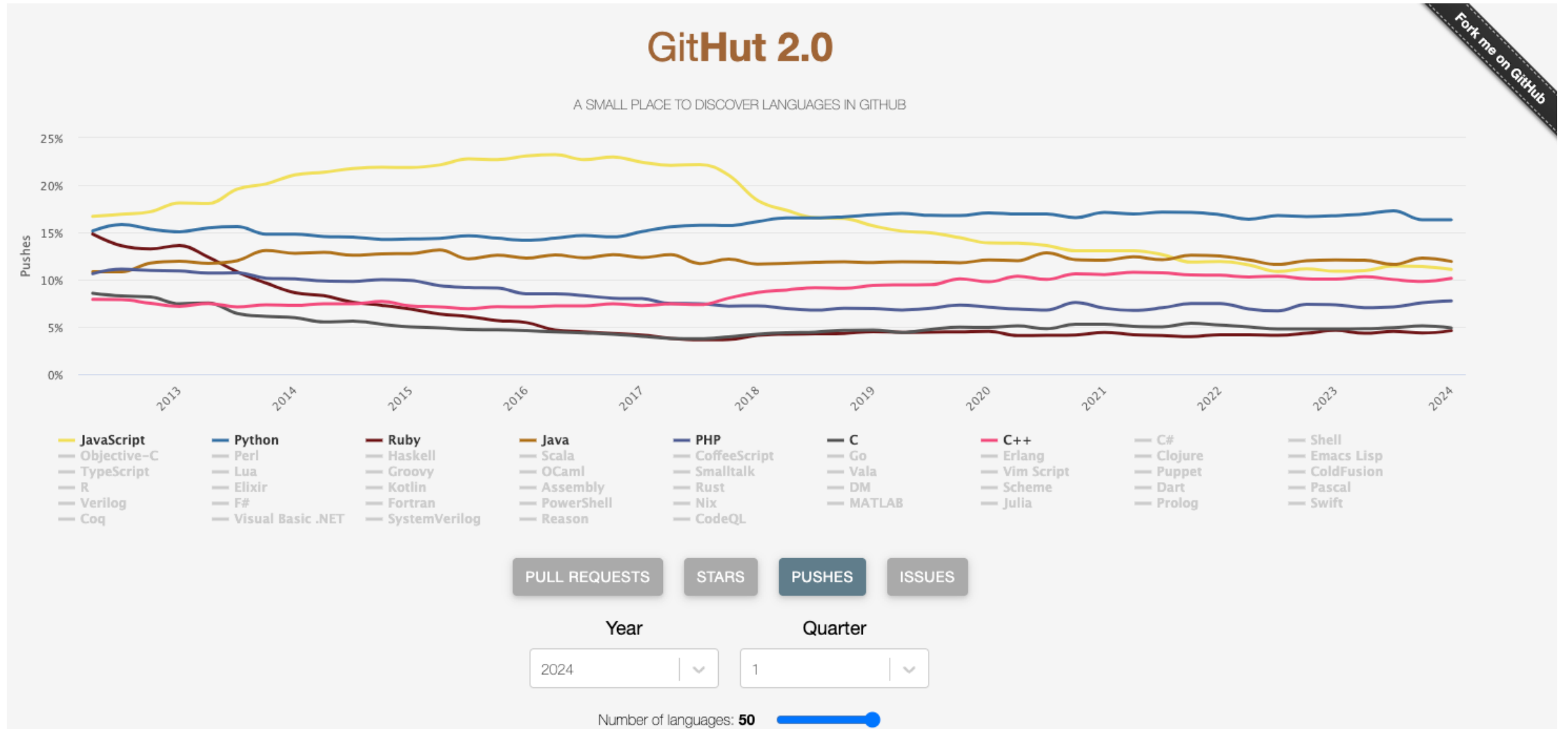


Figure from <https://madnight.github.io/github/#/pushes/2024/1>

Example: Dolma Dataset

- The **Dolma** dataset is 3 trillion tokens of text
- It is sourced from a variety of existing datasets
- **The Stack** is 3TB of permissively licensed code intended for LLMs

Subset		Size		
Source	Kind	Gzip files (GB)	Documents (millions)	Tokens (billions)
Common Crawl 24 shards, 2020-05 to 2023-06	web	4,197	4,600	2,415
C4 [24] [8]	web	302	364	175
peS2o [27]	academic	150	38.8	57
The Stack [16]	code	675	236	430
Project Gutenberg	books	6.6	0.052	4.8
Wikipedia, Wikibooks (<i>en, simple</i>)	encyclopedic	5.8	6.1	3.6
Total		5,334	5,245	3,084

Table 1: Composition of Dolma.

Language	The Stack [†]	CodeParrot [†]	AlphaCode	CodeGen	PolyCoder [†]
Assembly	2.36	0.78			
Batchfile	1.00	0.7			
C	222.88	183.83		48.9	55
C#	128.37	36.83	38.4		21
C++	192.84	87.73	290.5	69.9	52
CMake	1.96	0.54			
CSS	145.33	22.67			
Dockerfile	1.95	0.71			
FORTRAN	3.10	1.62			
GO	118.37	19.28	19.8	21.4	15
Haskell	6.95	1.85			
HTML	746.33	118.12			
Java	271.43	107.7	113.8	120.3	41
JavaScript	486.20	87.82	88	24.7	22
Julia	3.09	0.29			
Lua	6.58	2.81	2.9		
Makefile	5.09	2.92			
Markdown	164.61	23.09			
Perl	5.50	4.7			
PHP	183.19	61.41	64		13
PowerShell	3.37	0.69			
Python	190.73	52.03	54.3	55.9 (217.3)	16
Ruby	23.82	10.95	11.6		4.1
Rust	40.35	2.68	2.8		3.5
Scala	14.87	3.87	4.1		1.8
Shell	8.69	3.01			
SQL	18.15	5.67			
TeX	4.65	2.15			
TypeScript	131.46	24.59	24.90		9.20
Visual Basic	2.73	1.91			
Total	3135.95	872.95	715.1	314.1	253.6

Table 1: The size of The Stack (in GB) compared to other source code datasets used for pre-training LLMs. [†] indicates the dataset is publicly released. The Stack is more than three times the size of CodeParrot, the next-largest released code dataset.

Applications for Code Models

Languages for LLMs

- LLMs are trained on massive quantities of text from the internet (e.g. trillions of tokens)
- Some LLMs cover a variety of human languages, some focus primarily on English
- Most LLMs include a **wide variety of programming languages**

Code Generation Examples

- Auto-complete for code IDEs (e.g. GitHub CoPilot)
- Write code given a text prompt
- Write code given a docstring (e.g. function-level or class-level comment)
- Read a large codebase and add a new feature
- Find bugs / fix bugs
- Write unit tests
- Translate code from one language to another
- Generate comments for existing code

Applications for Code Models

Type	I-O	Task	Definition
Understanding	C-K	Code Classification	Classify code snippets based on functionality, purpose, or attributes to aid in organization and analysis.
		Bug Detection	Detect and diagnose bugs or vulnerabilities in code to ensure functionality and security.
		Clone Detection	Identifying duplicate or similar code snippets in software to enhance maintainability, reduce redundancy, and check plagiarism.
		Exception Type Prediction	Predict different exception types in code to manage and handle exceptions effectively.
	C-C	Code-to-Code Retrieval	Retrieve relevant code snippets based on a given code query for reuse or analysis.
NL-C	Code Search	Find relevant code snippets based on natural language queries to facilitate coding and development tasks.	
Generation	C-C	Code Completion	Predict and suggest the next portion of code, given contextual information from the prefix (and suffix), while typing to enhance development speed and accuracy.
		Code Translation	Translate the code from one programming language to another while preserving functionality and logic.
		Code Repair	Identify and fix bugs in code by generating the correct version to improve functionality and reliability.
		Mutant Generation	Generate modified versions of code to test and evaluate the effectiveness of testing strategies.
		Test Generation	Generate test cases to validate code functionality, performance, and robustness.
	C-NL	Code Summarization	Generate concise textual descriptions or explanations of code to enhance understanding and documentation.
NL-C	Code Generation	Generate source code from natural language descriptions to streamline development and reduce manual coding efforts.	

EVALUATING CODE GENERATION

How to evaluate code?

Metrics

- BLEU (metric used in machine translation for testing n-gram overlap with a known reference)
- CodeBLEU (a mixture of various syntactic and semantic metrics)
- Functional correctness (checks how many unit tests pass)
- NOTE: functional correctness has become the dominant metric for evaluation

Benchmarks

- HumanEval
- MBPP
- DS-1000
- **SWE-Bench Verified**
- Terminal-Bench 2.0
- (many more!)

CodeBLEU

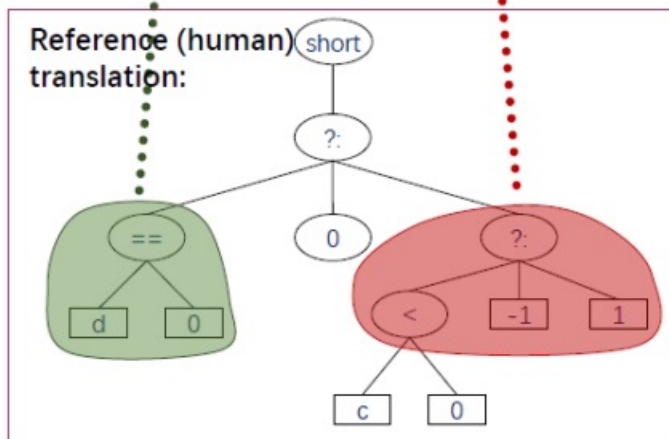
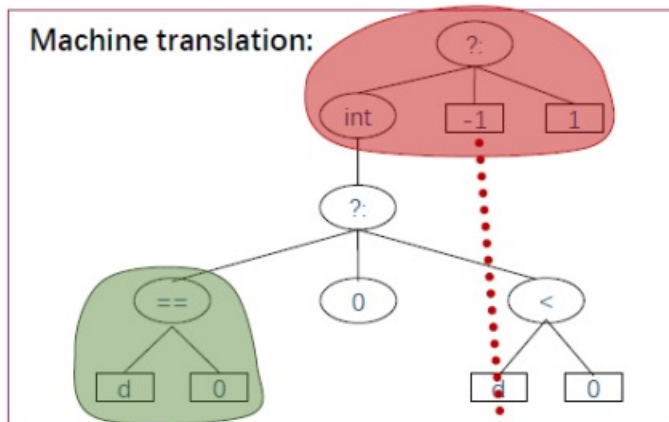
Machine translation:

```
public static int Sign ( double d )
{
    return ((int)((d == 0)? 0:(d < 0)))?
    -1: 1;
}
```

Reference (human) translation:

```
public static short Sign ( double d )
{
    return (short)((d == 0)? 0:(c < 0)?
    -1: 1);
}
```

Weighted N-Gram Match



Syntactic AST Match

Machine translation:

```
public static int Sign ( double d )
{
    return ((int)((d == 0)? 0:(d < 0)))?
    -1: 1;
}
```

[[('d', 7, 'comesFrom', [], []),
('d', 16, 'comesFrom', ['d'], [7]),
('d', 24, 'comesFrom', ['d'], [7])]

Reference (human) translation:

```
public static short Sign ( double c )
{
    return (short)((c == 0)? 0:(c < 0)?
    -1: 1);
}
```

Semantic Data-flow Match

$$\text{CodeBLEU} = \alpha \cdot \text{N-Gram Match (BLEU)} + \beta \cdot \text{Weighted N-Gram Match} + \gamma \cdot \text{Syntactic AST Match} + \delta \cdot \text{Semantic Data-flow Match}$$

HumanEval Benchmark

- Introduced alongside Codex model
- Measures functional correctness of code (i.e. how many unit tests pass)
- pass@k metric = % of k code samples that pass *all* the unit tests (actual implementation uses more samples to reduce variance)
- 164 handwritten problems

```
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]
```

```
def solution(lst):  
    """Given a non-empty list of integers, return the sum of all of the odd elements  
    that are in even positions.  
  
    Examples  
    solution([5, 8, 7, 1]) ==>12  
    solution([3, 3, 3, 3, 3]) ==>9  
    solution([30, 13, 24, 321]) ==>0  
    """  
    return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

```
def encode_cyclic(s: str):  
    """  
    returns encoded string by cycling groups of three characters.  
    """  
    # split string to groups. Each of length 3.  
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]  
    # cycle elements in each group. Unless group has fewer elements than 3.  
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]  
    return "".join(groups)  
  
def decode_cyclic(s: str):  
    """  
    takes as input string encoded with encode_cyclic function. Returns decoded string.  
    """  
    # split string to groups. Each of length 3.  
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]  
    # cycle elements in each group.  
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]  
    return "".join(groups)
```

HumanEval Benchmark

- Introduced alongside Codex model
- Measures functional correctness of code (i.e. how many unit tests pass)
- pass@k metric = % of k code samples that pass *all* the unit tests (actual implementation uses more samples to reduce variance)
- 164 handwritten problems

- Results clearly indicate that BLEU is not a good surrogate

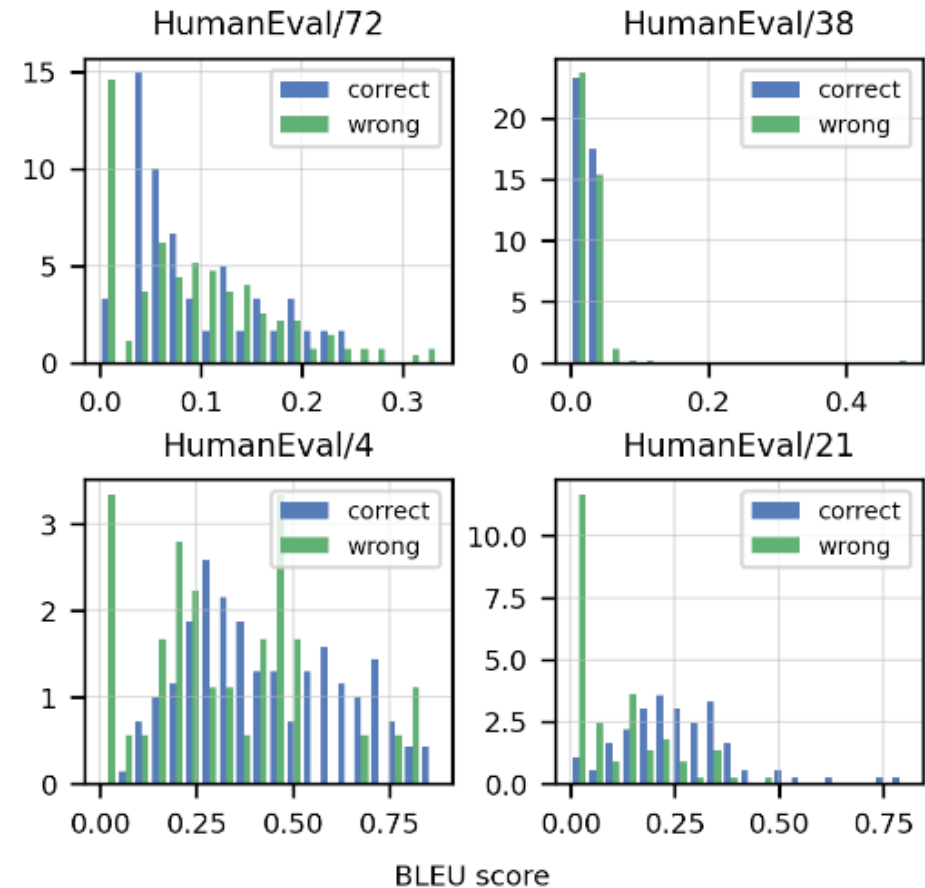


Figure 8. BLEU score probability densities for correct (blue) and wrong (green) solutions from Codex-12B for 4 random tasks from HumanEval. Note that the distributions are not cleanly separable, suggesting that optimizing for BLEU score is not equivalent to optimizing for functional correctness.

MBPP

- Mostly Basic Python Programs (MBPP) are intended to be problems solvable by a novice programmer
- 974 problems constructed by crowd-sourcing
- Consists of problem statement, 3 tests, and a self-contained solution

prompt

Write a python function to check if a given number is one less than twice its reverse. Your code should satisfy these tests:

```
assert check(70) == False
assert check(23) == False
assert check(73) == True
```

model

```
def check(n):
    if n == 2*int(str(n)[::-1])-1:
        return True
    else:
        return False
```

prompt

Write a function to find the smallest missing element in a sorted array. Your code should satisfy these tests:

```
assert smallest_missing([0, 1, 2, 3, 4, 5, 6], 0, 6) == 7
assert smallest_missing([0, 1, 2, 6, 9, 11, 15], 0, 6) == 3
assert smallest_missing([1, 2, 3, 4, 6, 9, 11, 15], 0, 7) == 0
```

model

```
def smallest_missing(arr, n, m):
    smallest = min(n, m)
    for i in range(n, m + 1):
        if arr[i] <= smallest:
            smallest += 1
    return smallest
```

SWE-Bench Verified

- SWE-Bench:
 - a real-world coding benchmark
 - solve actual GitHub issues by producing a patch in the repo, then score via execution/unit tests against the reference fix
 - In November 2024, Claude 2.0 only solved 1.96% of issues
- SWE-Bench Verified: human validated subset of 500 test cases
- Construction:
 - selected 12 popular Python repos
 - grabbed 90k pull requests (PRs)
 - required PRs to resolve an issue and change tests
 - required PRs to change at least one test from fail to pass
 - yielded 2,294 tasks

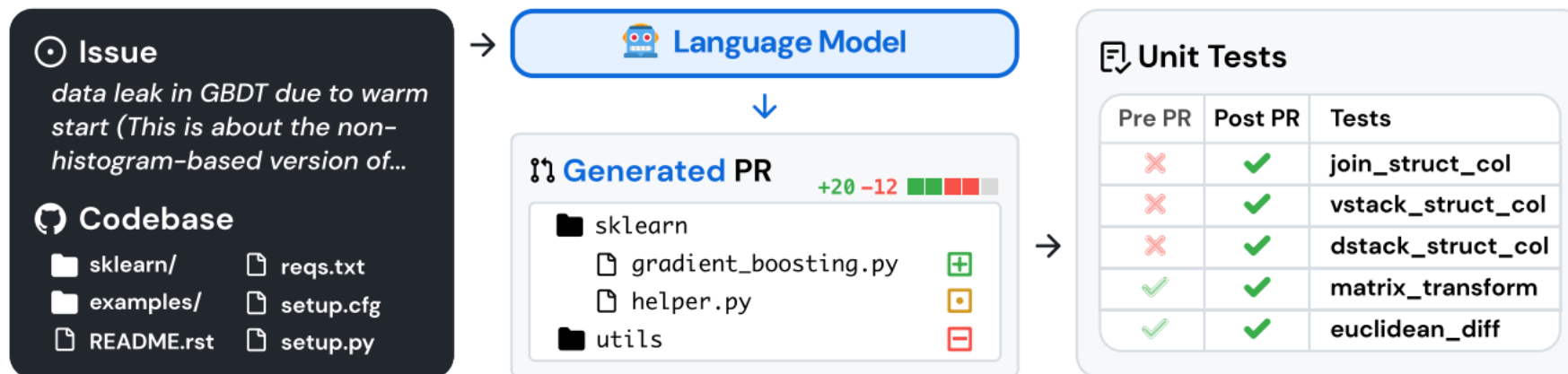
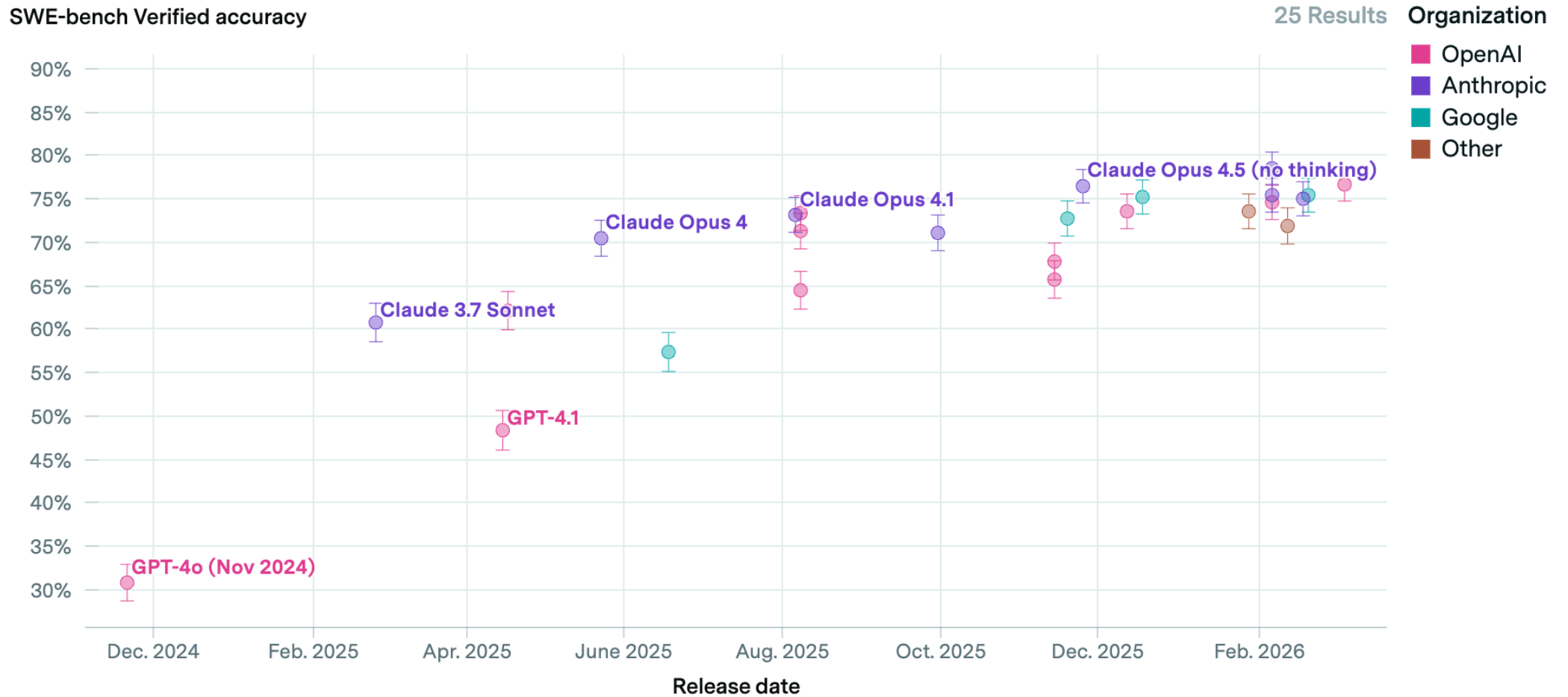


Figure 1: SWE-bench sources task instances from real-world Python repositories by connecting GitHub issues to merged pull request solutions that resolve related tests. Provided with the issue text and a codebase snapshot, models generate a patch that is evaluated against real tests.

SWE-Bench Verified

AI performance solving issues from 12 open-source Python repositories



Terminal-Bench 2.0

- Realistic interactive tasks for command line interfaces that require manipulation of the environment
- Each task has:
 - Docker container with relevant files
 - English instruction for the agent
 - Tests to verify correctness of final container state
 - A reference solution
- Terminal Bench 2.0 contains 89 tasks manually verified by 3 humans

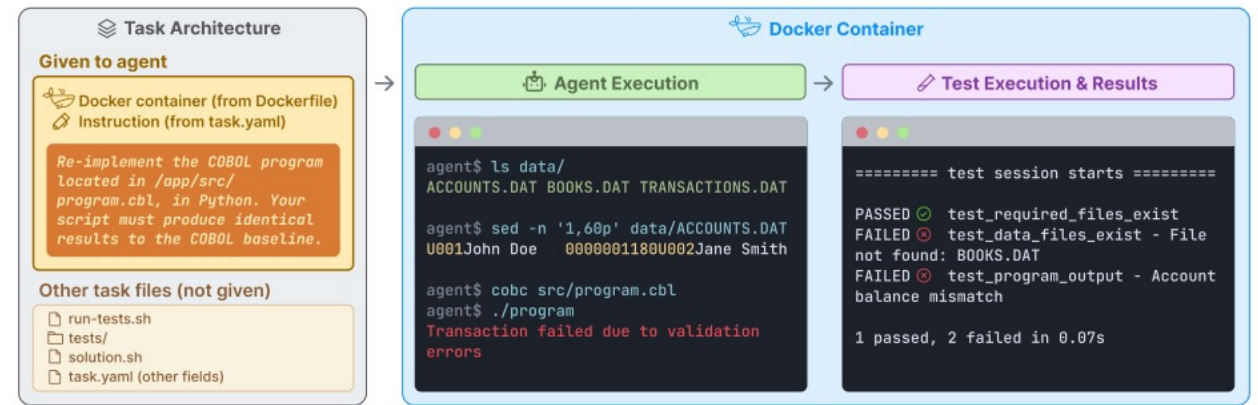


Figure 2: A Terminal-Bench task is composed of an instruction, a Dockerfile, a set of tests, and an oracle solution. Agents run inside a container into which the tests are copied and executed.

	<1 hour	1 hour – 1 day (24h)	1 day (24h) – 1 week (168h)	>1 week (168h)
Expert	36 (48.6%)	35 (47.3%)	3 (4.1%)	0 (0.0%)
Junior	6 (8.1%)	53 (71.6%)	12 (16.2%)	3 (4.1%)

Table 1: Distribution of task completion times for expert and junior engineers across all tasks in Terminal-Bench 2.0, as estimated by the task authors. One task, fix-ocaml-gc, requires the agent to fix the OCaml garbage collector after an attempted but failed optimization and takes almost 1 day (24 hours) for an expert and ten days (240 hours) for a junior engineer to complete, highlighting the ability of the Terminal-Bench framework to express complex, long-horizon tasks.

Terminal-Bench 2.0

Example: Software Engineering

build-pov-ray:

Build POV-Ray version 2.2 from source by downloading archives, extracting to `/app/povray-2.2`, compiling, and installing the binary to `/usr/local/bin/povray`. The build will be validated by rendering a test scene file (`/app/deps/illum1.pov`) and comparing output against a reference image, with a provided sanity check command to verify the installation works correctly.

Example: Games

chess-best-move:

Analyze a chess position from an image file to determine and output the optimal move(s) for white. The solution must identify the best move(s) from the given board state and write them to a file in algebraic notation format (source square followed by destination square, e.g., `e2e4`). If multiple equally strong winning moves exist, all should be listed on separate lines.

In-Class Poll

Question:

What are the potential societal impacts of coding agents?

Answer:

CODE MODELS

Code Generation with LLMs

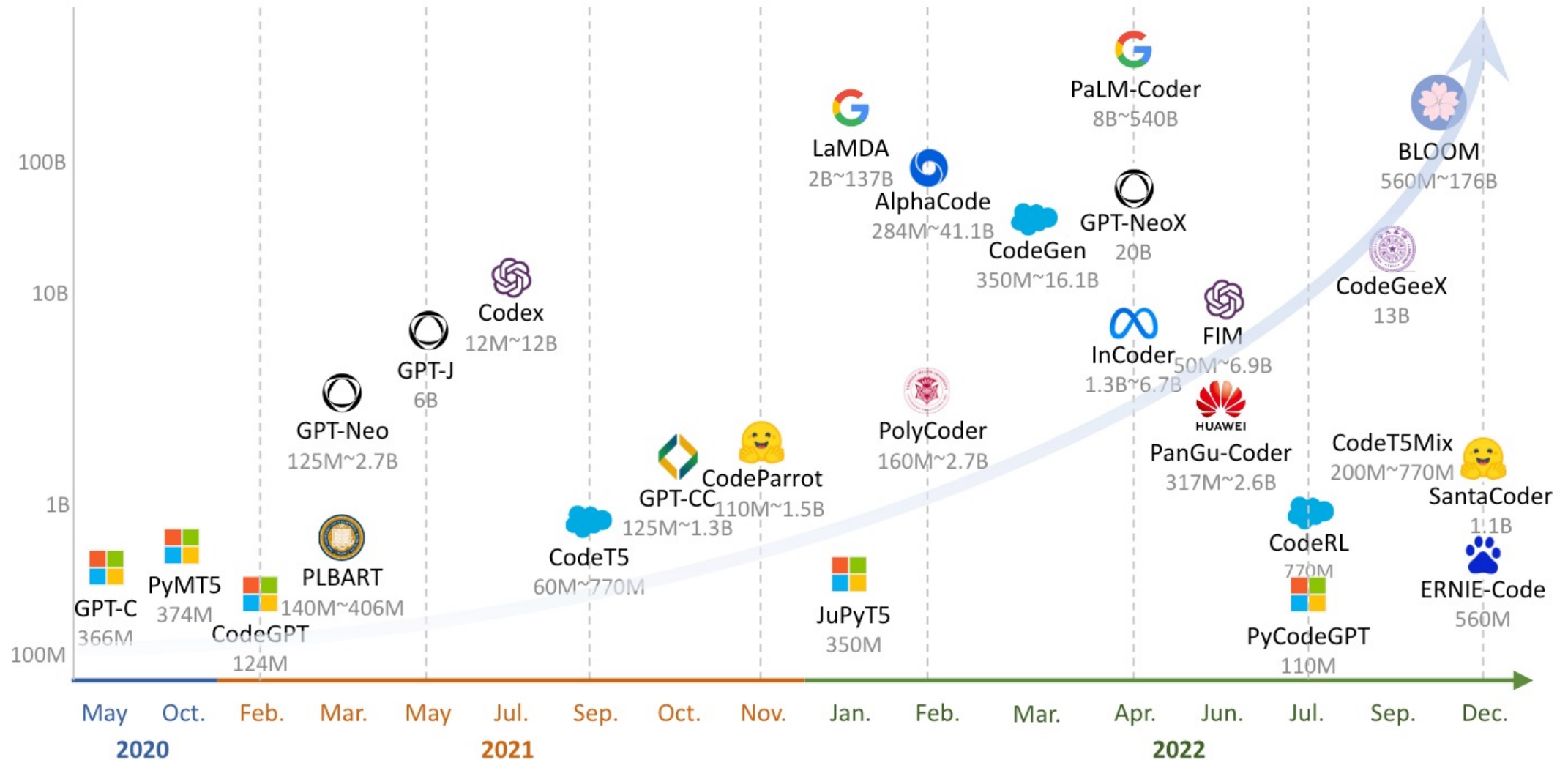


Figure from <https://aclanthology.org/2023.acl-long.411.pdf>

Code Generation with LLMs

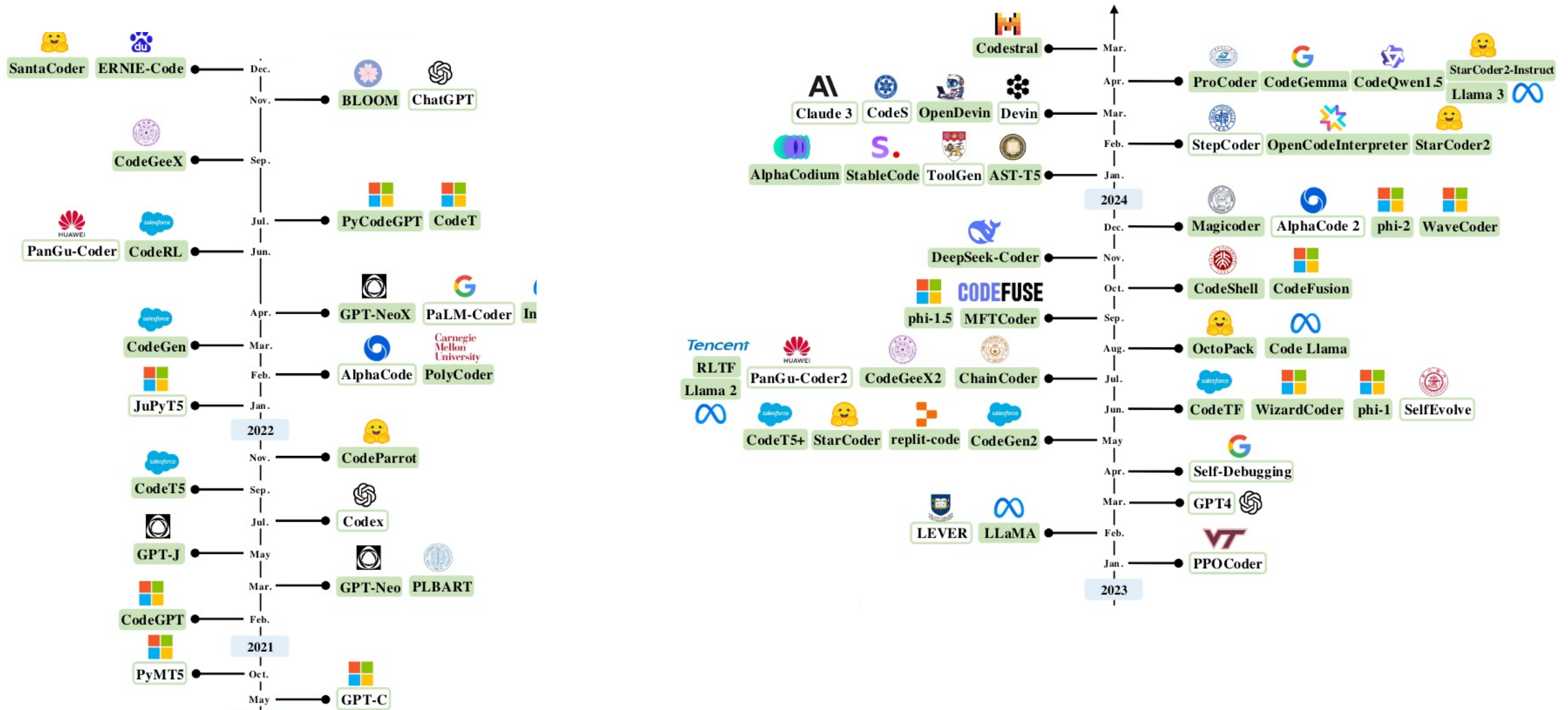


Figure from <https://arxiv.org/pdf/2406.00515>

Approaches to Code Generation

Here we consider a few representative examples of code models:

- CodeBERT
- Codex
- CodeT5
- InCoder / FIM
- StarCoder
- LongCoder

CodeBERT

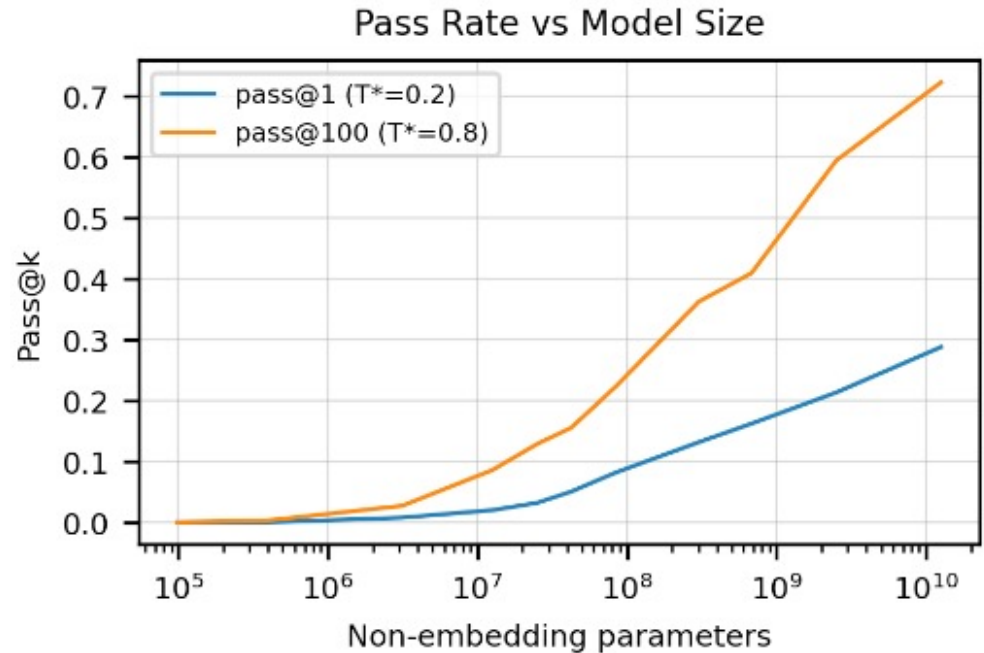
- One of the early successes in this space, CodeBERT has 125M model parameters (same architecture as RoBERTa)
- Two pre-training objectives:
 - Masked language modeling (MLM)
 - Replaced token detection (RTD)

- Example application:
 - natural language code retrieval

MODEL	RUBY	JAVASCRIPT	GO	PYTHON
NBOW	0.4285	0.4607	0.6409	0.5809
CNN	0.2450	0.3523	0.6274	0.5708
BiRNN	0.0835	0.1530	0.4524	0.3213
SELFATT	0.3651	0.4506	0.6809	0.6922
RoBERTa	0.6245	0.6060	0.8204	0.8087
PT w/ CODE ONLY (INIT=S)	0.5712	0.5557	0.7929	0.7855
PT w/ CODE ONLY (INIT=R)	0.6612	0.6402	0.8191	0.8438
CODEBERT (MLM, INIT=S)	0.5695	0.6029	0.8304	0.8261
CODEBERT (MLM, INIT=R)	0.6898	0.6997	0.8383	0.8647
CODEBERT (RTD, INIT=R)	0.6414	0.6512	0.8285	0.8263
CODEBERT (MLM+RTD, INIT=R)	0.6926	0.7059	0.8400	0.8685

Codex

- The original model behind GitHub Copilot
- GPT-3 model with 12B parameters fine-tuned on 159 GB of Python code
- Notably: using a pre-trained GPT-3 does not improve performance, but does improve convergence time



CodeT5

- CodeT5 is based on the T5 encoder-decoder Transformer architecture
- Like T5, CodeT5 brings together a number of different tasks

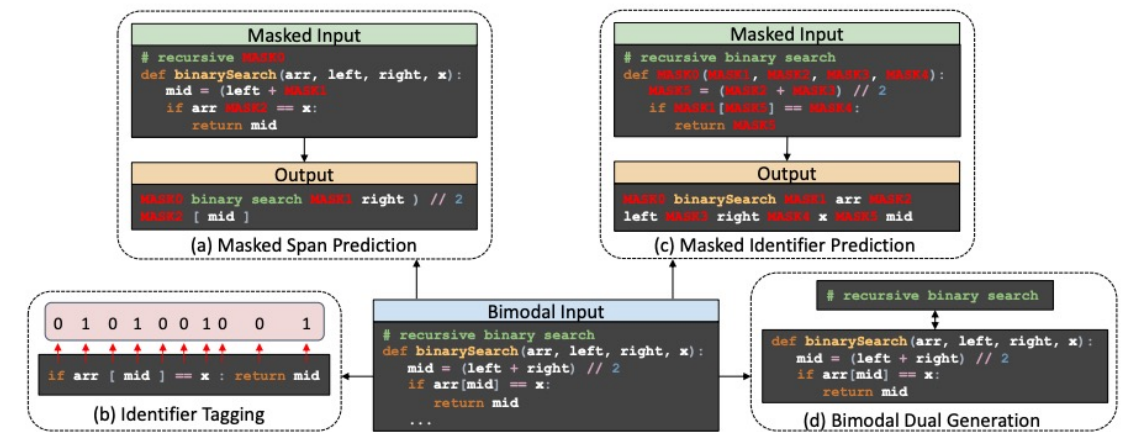
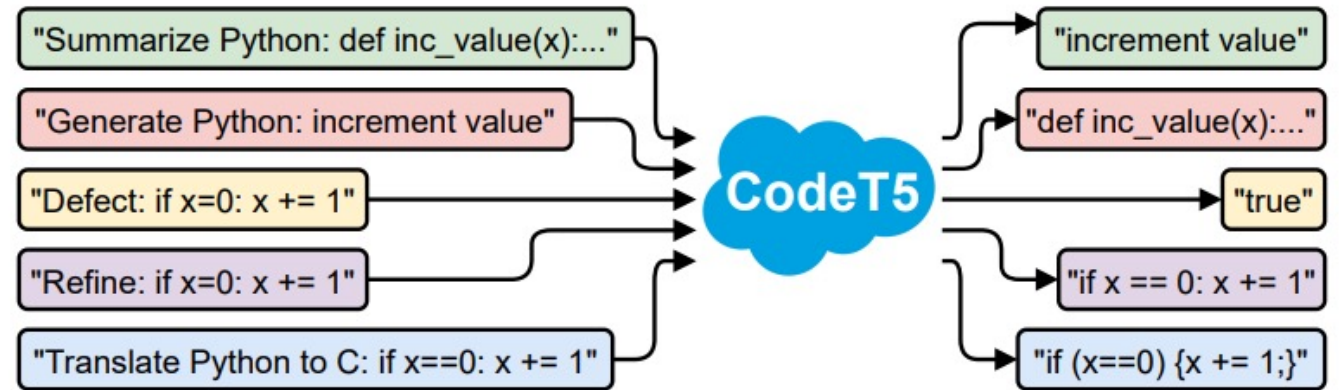
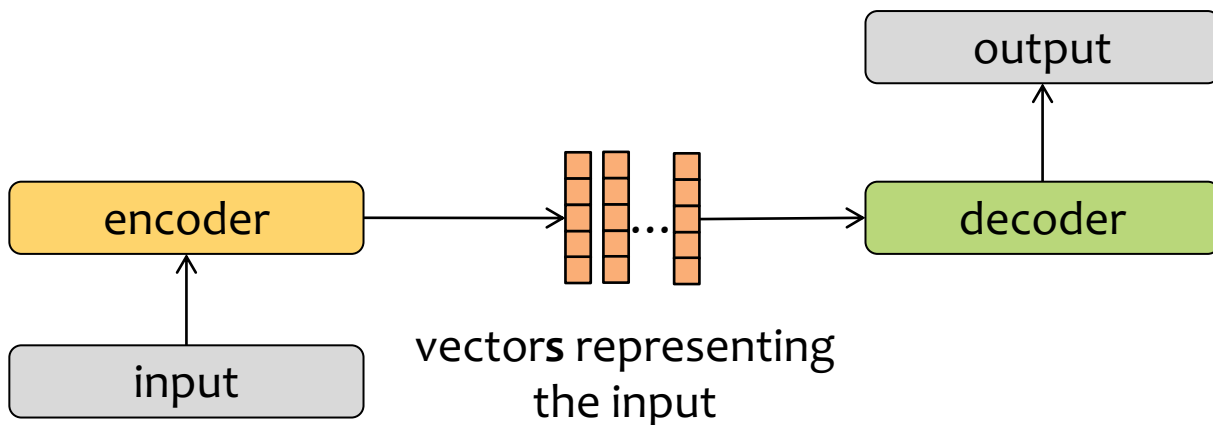


Figure 2: Pre-training tasks of CodeT5. We first alternately train span prediction, identifier prediction, and identifier tagging on both unimodal and bimodal data, and then leverage the bimodal data for dual generation training.

Question how do we use an **causally-masked LM** to fill in code in the **middle** of code file?

InCoder / FIM

InCoder (April 2022)

- Place a mask token where you want to fill in the code

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        <MASK:0> in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
for line in f:
    for word in line.split():
        if word <EOM>
```

FIM (July 2022)

- Goal is to train a model that fills in the middle
- Divide the code snippet into (prefix, middle, suffix)
- Then train with examples of the form:
<PRE> prefix <SUF> suffix <MID> middle
- And predict using examples of the form:
<PRE> prefix <SUF> suffix <MID>

StarCoder

StarCoder

- (Was) one of the best open source Code Models
- Used FIM for pre-training a 15.5B parameter model on 1 trillion tokens of text

| Hyperparameter | SantaCoder | StarCoder |
|--------------------------|-----------------------|------------------------|
| Hidden size | 2048 | 6144 |
| Intermediate size | 8192 | 24576 |
| Max. position embeddings | 2048 | 8192 |
| Num. of attention heads | 16 | 48 |
| Num. of hidden layers | 24 | 40 |
| Attention | Multi-query | Multi-query |
| Num. of parameters | $\approx 1.1\text{B}$ | $\approx 15.5\text{B}$ |

| Model | Size | HumanEval | MBPP |
|----------------------|-------|-----------|------|
| <i>Open-access</i> | | | |
| LLaMA | 7B | 10.5 | 17.7 |
| LLaMA | 13B | 15.8 | 22.0 |
| SantaCoder | 1.1B | 18.0 | 35.0 |
| CodeGen-Multi | 16B | 18.3 | 20.9 |
| LLaMA | 33B | 21.7 | 30.2 |
| CodeGeeX | 13B | 22.9 | 24.4 |
| LLaMA-65B | 65B | 23.7 | 37.7 |
| CodeGen-Mono | 16B | 29.3 | 35.3 |
| StarCoderBase | 15.5B | 30.4 | 49.0 |
| StarCoder | 15.5B | 33.6 | 52.7 |
| <i>Closed-access</i> | | | |
| LaMDA | 137B | 14.0 | 14.8 |
| PaLM | 540B | 26.2 | 36.8 |
| code-cushman-001 | 12B | 33.5 | 45.9 |
| code-davinci-002 | 175B | 45.9 | 60.3 |

LongCoder

- LongCoder aims to address the problem of working with large codebases
- Employs sparse attention to handle long input sequences

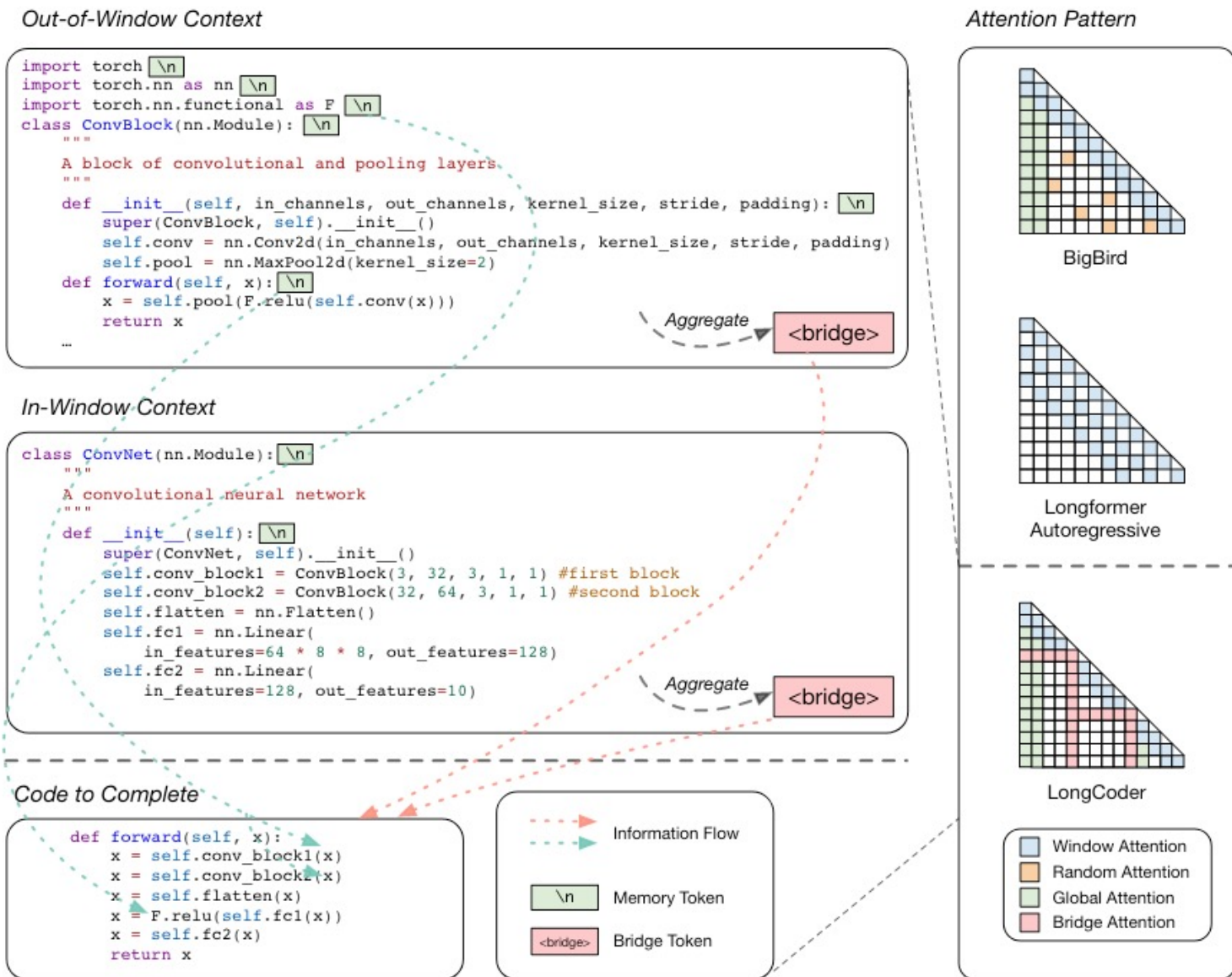
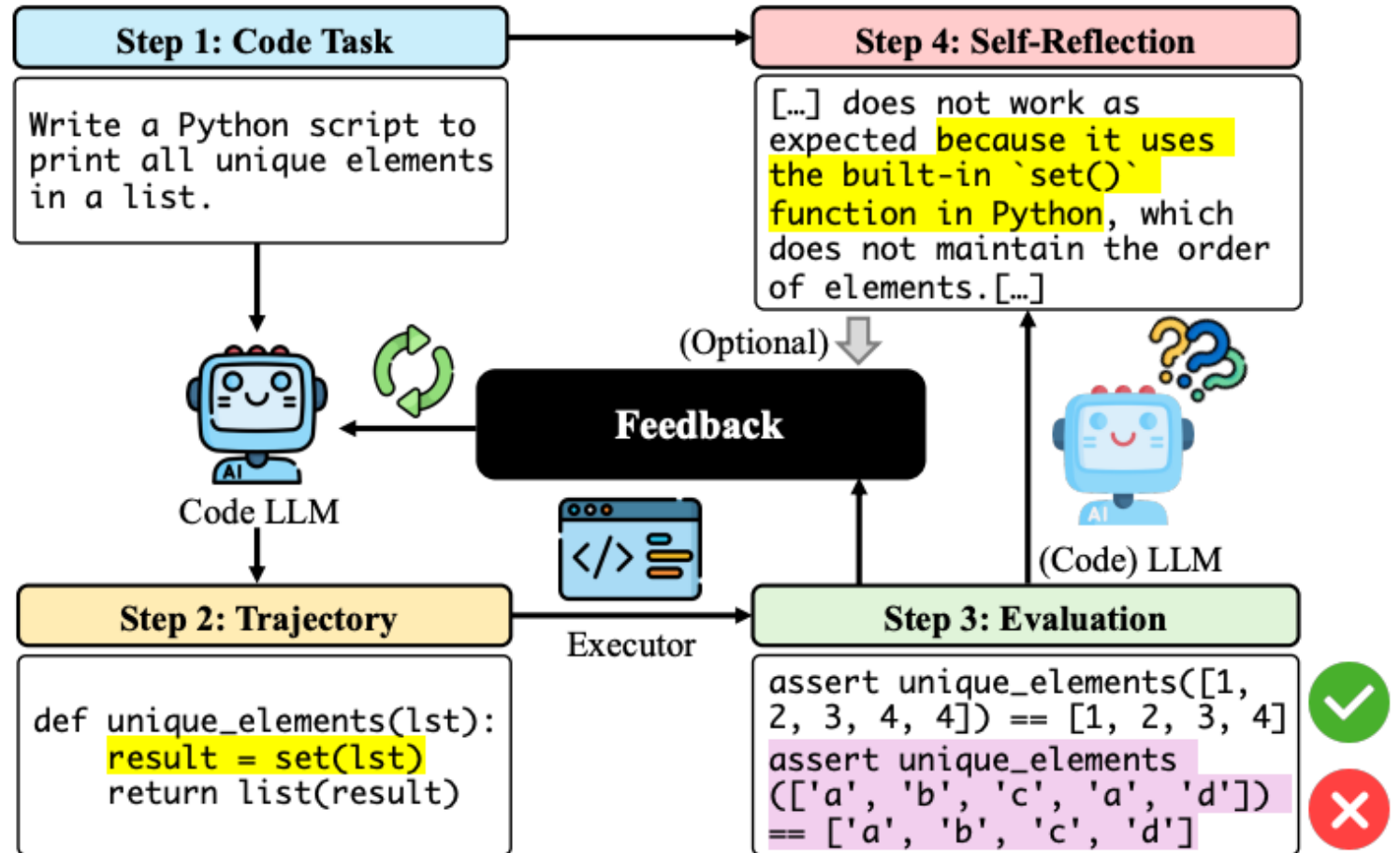


Figure 1. (Left) An example of how LongCoder facilitates completion with longer context. The memory tokens save potentially useful information (including package imports, class and function definitions) for global access despite whether they are within the sliding window. The bridge tokens aggregate local information by attending to a fixed length of tokens. The information flow within the window is omitted for clarity. (Right) Attention patterns used in BigBird (Zaheer et al., 2020), Longformer (Beltagy et al., 2020) and LongCoder. Best viewed in color.

CODE MODEL-SPECIFIC TECHNIQUES

Iterative Self-Refinement

- Normally self-correction with an LLM (e.g. for reasoning problems) does not work
- However, when performing self-correction on a code model, we may also have access to unit test output
- This output from unit tests can lead to great success in iterative self-refinement at test time



TOOL CALLING

Tool Calling

- Tool calling lets an LLM choose and invoke external functions instead of only generating text.
- A tool usually has a name, a description, and a schema for its arguments.
- The model decides when a tool is needed, outputs a structured call, gets the tool result back, then continues reasoning with that result.
- This is useful for actions or grounded data, like checking weather, querying a database, sending email, or doing calculations.
- The model is not “doing” the external action by itself; it is requesting that the surrounding system run the tool.

```
1 // Available tool:
2 {
3   "name": "get_weather",
4   "description": "Get a weather forecast for a city and date",
5   "parameters": {
6     "type": "object",
7     "properties": {
8       "city": { "type": "string" },
9       "date": { "type": "string" }
10    },
11    "required": ["city", "date"]
12  }
13 },
14
15 // Model tool call:
16 {
17   "tool_name": "get_weather",
18   "arguments": {
19     "city": "Pittsburgh",
20     "date": "2026-04-09"
21   }
22 },
23
24 // Tool result:
25 {
26   "forecast": "Cloudy, high 52°F, low 39°F"
27 }
```

Kimi K2

- Synthesizes tool-use SFT data at scale: build a large tool-spec repository (real MCP tools + synthetic tools), then generate agents, tasks, and successful tool-calling trajectories
- Post-trains with joint RL in real and synthetic environments, so the model improves tool selection/sequencing from interaction outcomes, not just imitation
- Serves tools at inference by giving the model the available tool list each request; it then decides when/how to call them autonomously

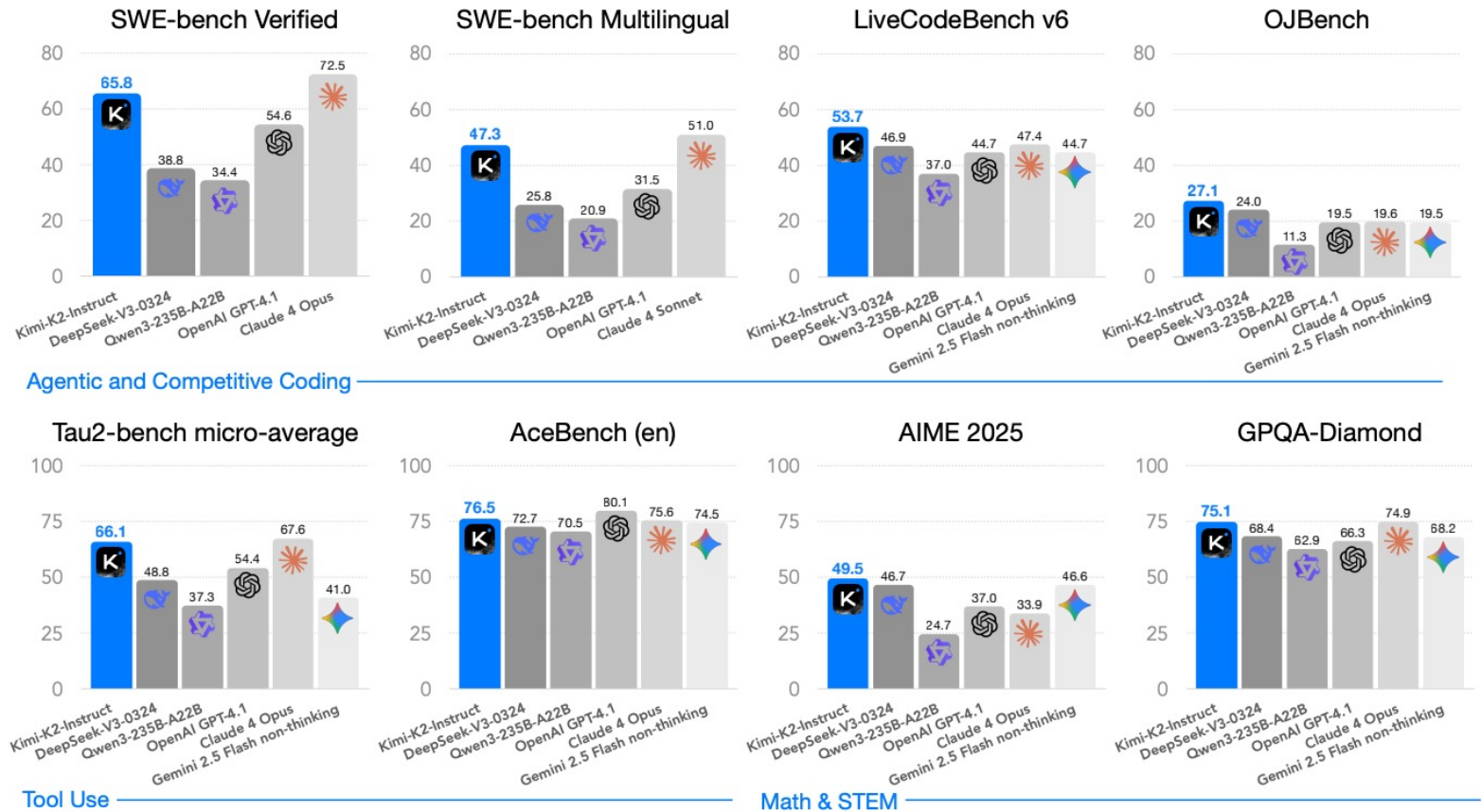
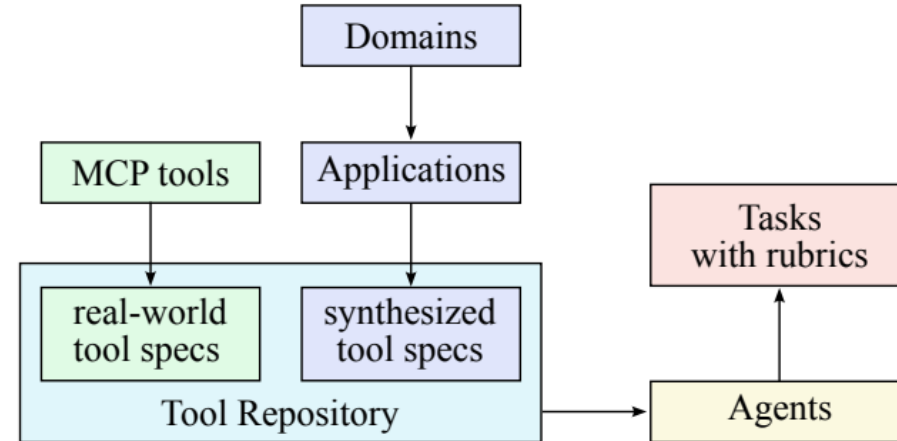


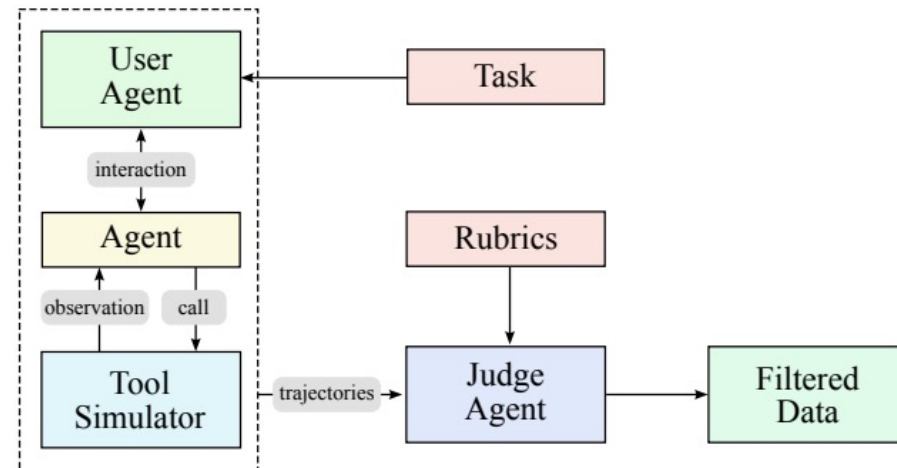
Figure 1: Kimi K2 main results.²

Kimi K2

- Synthesizes tool-use SFT data at scale: build a large tool-spec repository (real MCP tools + synthetic tools), then generate agents, tasks, and successful tool-calling trajectories
- Post-trains with joint RL in real and synthetic environments, so the model improves tool selection/sequencing from interaction outcomes, not just imitation
- Serves tools at inference by giving the model the available tool list each request; it then decides when/how to call them autonomously



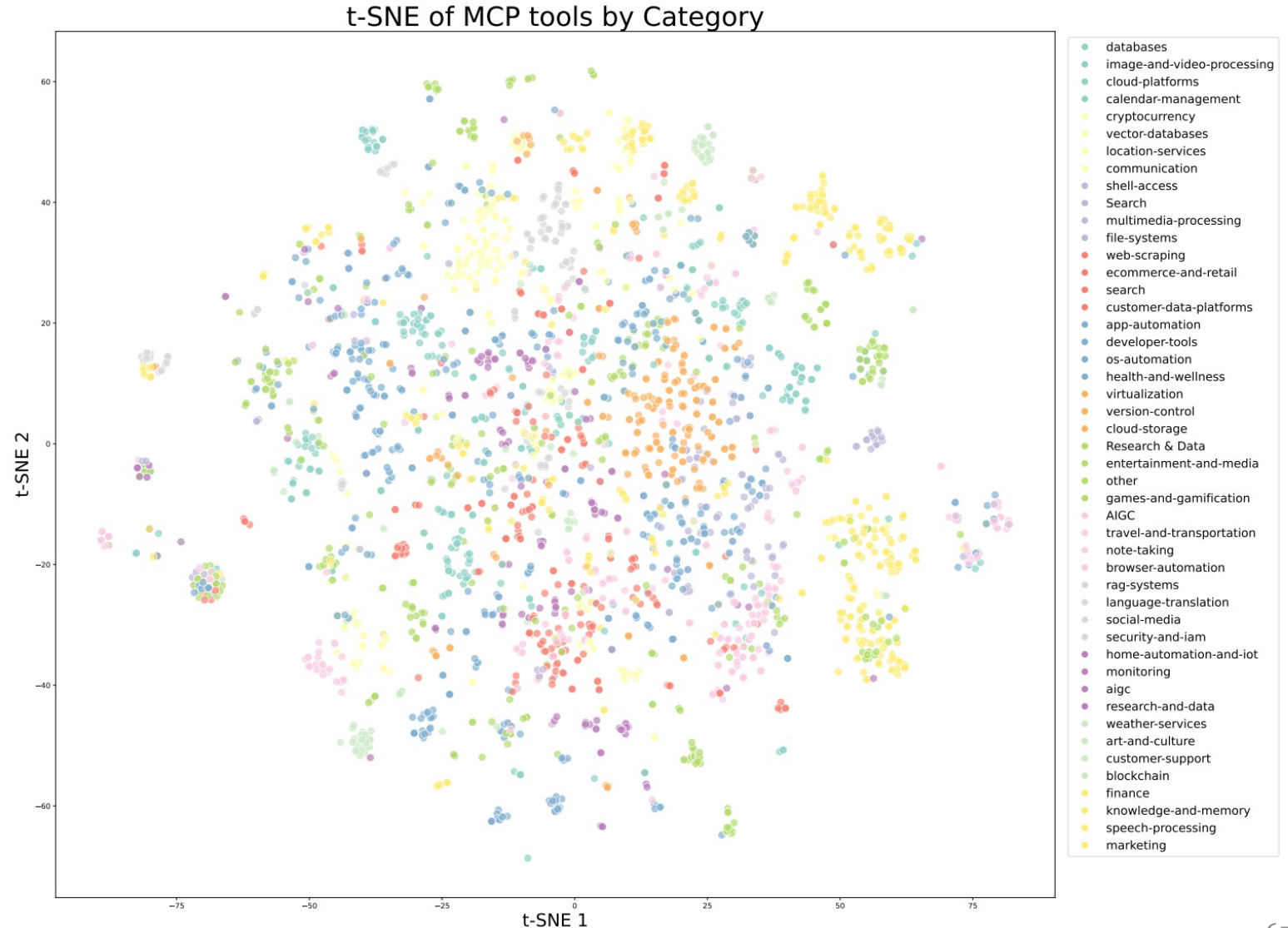
(a) Synthesizing tool specs, agents and tasks



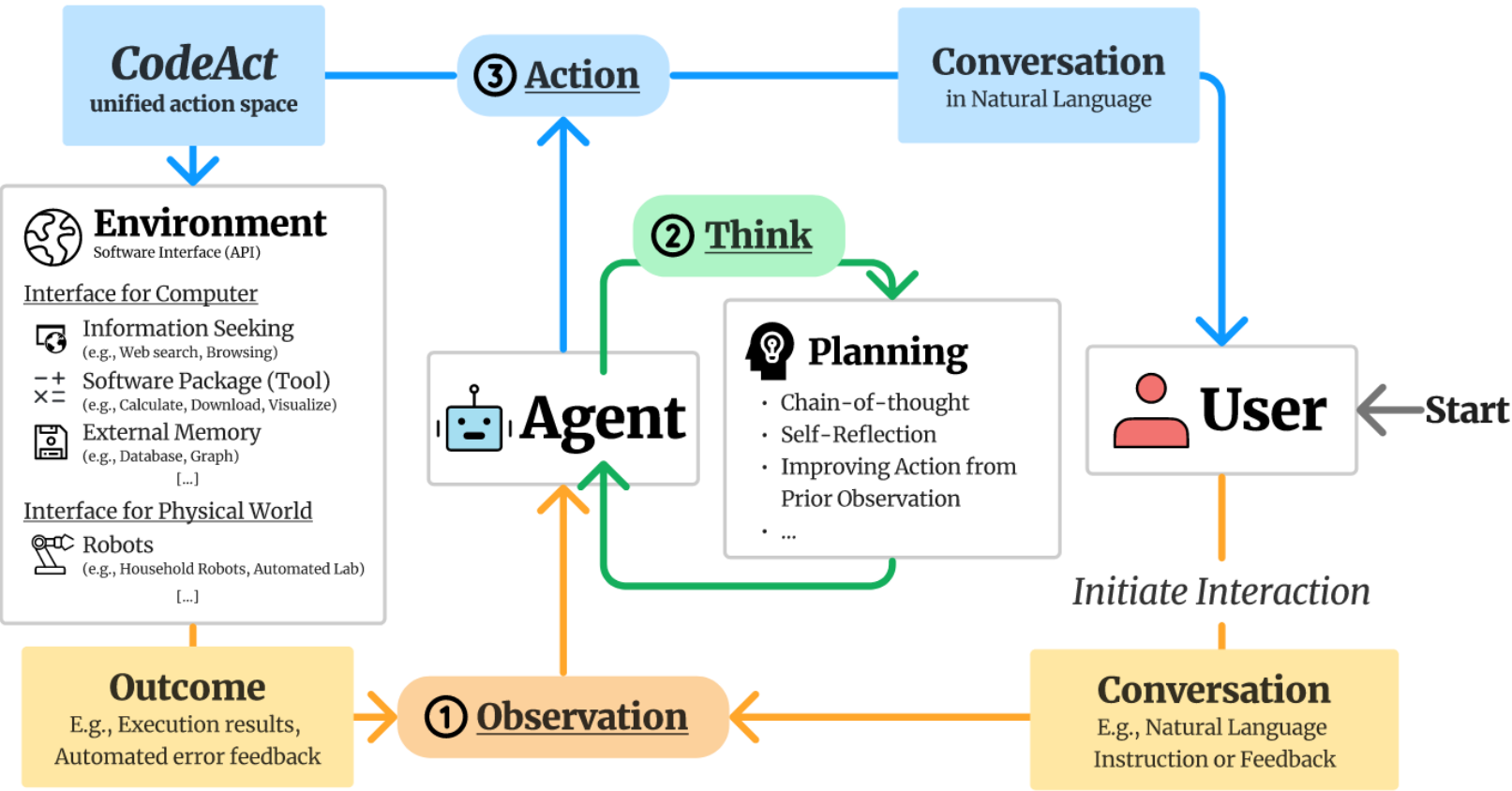
(b) Generating agent trajectories

Kimi K2

- Synthesizes tool-use SFT data at scale: build a large tool-spec repository (real MCP tools + synthetic tools), then generate agents, tasks, and successful tool-calling trajectories
- Post-trains with joint RL in real and synthetic environments, so the model improves tool selection/sequencing from interaction outcomes, not just imitation
- Serves tools at inference by giving the model the available tool list each request; it then decides when/how to call them autonomously



CodeAct



- For each interaction:
 1. agent receives observation (either prompt to solve problem or exec result)
 2. agent thinks / plans
 3. agent takes an action (either respond to user or call tools)
 4. when tool called, environment executes and produces exec result for agent
- Repeat until problem solved

CodeAct

Example

User:

- ① Find the sum of the reciprocals of the roots of $x^2-13x+4=0$.

Assistant:

- ② To find the sum of the reciprocals of the roots of the quadratic equation $x^2-13x+4=0$ [...]

<execute>

- ```
import sympy
x = sympy.Symbol('x')
③ roots = sympy.solve(x**2 - 13*x + 4)
print(1/roots[0] + 1/roots[1])
</execute>
```

**Env:**

- ①  $1/(3*\sqrt{17}/2 + 13/2) + 1/(13/2 - 3*\sqrt{17}/2)$

**Assistant:**

- ② The roots obtained are in symbolic form, which makes it difficult to evaluate the sum directly. Let's use the quadratic formula to explicitly find the numerical values of the roots first.  
[...]

- For each interaction:
  1. agent receives observation (either prompt to solve problem or exec result)
  2. agent thinks / plans
  3. agent takes an action (either respond to user or call tools)
  4. when tool called, environment executes and produces exec result for agent
- Repeat until problem solved

# CodeAct

- Key **innovation** of CodeAct: rather than use JSON to express actions (e.g. a declarative specification of a tool call), use a python program that treats tools as functions
- Enables consolidation of many tool calls into a single action
- Capitalizes on existing LLM's prowess at code generation
- Idea has persisted into SOTA coding agents

**Instruction:** Determine the most cost-effective country to purchase the smartphone model "CodeAct 1". The countries to consider are the USA, Japan, Germany, and India.

**Available APIs**

[1] lookup\_rates(country: str) -> (float, float)

[2] convert\_and\_tax(price: float, exchange\_rate: float, tax\_rate: float) -> float

[3] estimate\_final\_price(converted\_price: float, shipping\_cost: float) -> float

[4] lookup\_phone\_price(model: str, country: str) -> float

[5] estimate\_shipping\_cost(destination\_country: str) -> float

**LLM Agent using [Text/JSON] as Action**

**Think** I should calculate the phone price in USD for each country, then find the most cost-effective country.

**Action** **Text:** lookup\_rates, Germany  
**JSON:** {"tool": "lookup\_rates", "country": "Germany"}

**Environment** 1.1, 0.19

**Action** **Text:** lookup\_phone\_price, CodeAct 1, Germany  
**JSON:** {"tool": "lookup\_phone\_price", "model": "CodeAct 1", "country": "Germany"}

**Environment** 700

**Action** **Text:** convert\_and\_tax, 700, 1.1, 0.19  
**JSON:** {"tool": "convert\_and\_tax", "price": 700, "exchange\_rate": 1.1, "tax\_rate": 0.19}

**Environment** 916.3

[... interactions omitted (look up shipping cost and calculate final price) ...]

**Action** **Text:** lookup\_rates, Japan  
**JSON:** {"tool": "lookup\_rates", "country": "Japan"}

[... interactions omitted (calculate final price for all other countries)...]

**Response** The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

**CodeAct: LLM Agent using [Code] as Action**

**Think** I should calculate the phone price in USD for each country, then find the most cost-effective country.

**Action**

```

countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
 exchange_rate, tax_rate = lookup_rates(country)
 local_price = lookup_phone_price("xAct 1", country)
 converted_price = convert_and_tax(
 local_price, exchange_rate, tax_rate
)
 shipping_cost = estimate_shipping_cost(country)
 final_price = estimate_final_price(converted_price, shipping_cost)
 final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)

```

Control & Data Flow of Code Simplifies Complex Operations

**Environment** 1.1, 0.19

**Response** The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

Fewer Actions Required!

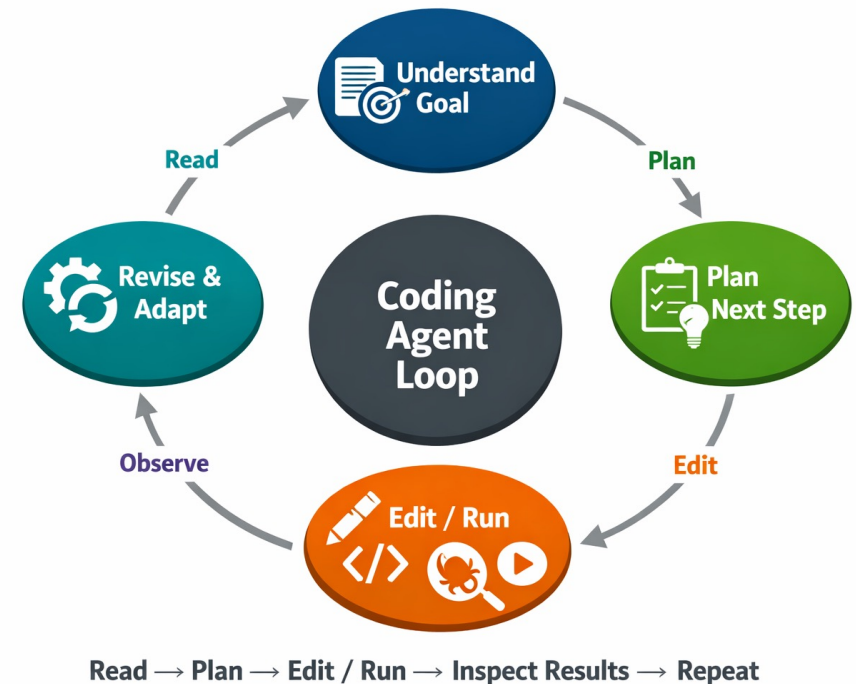
Re-use `min` Function from Existing Software Infrastructures (Python library)

Figure from <https://arxiv.org/pdf/2402.01030>

# **CODING AGENTS**

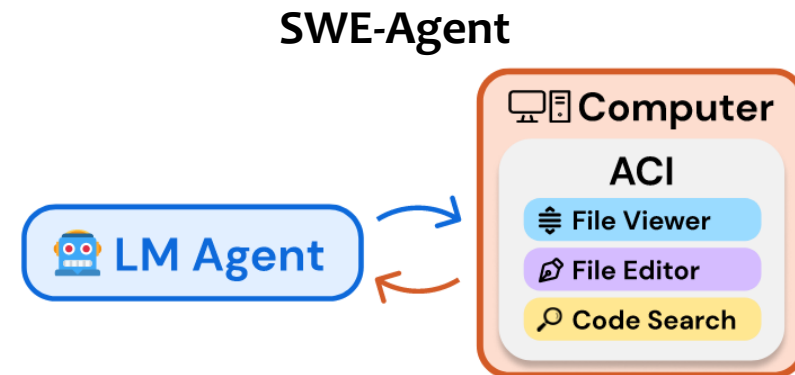
# The Coding Agent Loop

- A coding agent loop is the repeated cycle an autonomous coding system uses to make progress on a programming task:
  1. **Understand the goal:** read the task, codebase, and relevant files.
  2. **Plan the next step:** choose a concrete action, like inspect code, edit a file, run tests, or search docs.
  3. **Act:** make the code change or use a tool.
  4. **Observe feedback:** read compiler errors, test results, logs, or tool outputs.
  5. **Revise:** update its understanding and decide what to try next.
- This loop continues until the task is solved, it gets stuck, or it reaches a stopping condition.  
In compact form: read → plan → edit/run → inspect results → repeat.



# Agent Scaffolds

- Key observation: the way that tools are exposed to the agent is critically important to success
- Example tools:
  - search / navigation
    - find a file
    - search within a file
    - search within a directory
  - file viewing
    - open (and view 100 lines)
    - scroll up / scroll down / goto line
  - file editing
    - write new file
    - edit existing file
  - execute
    - run bash command
    - execute python snippet
    - run linter / compile code
- Example scaffolds/harnesses
  - SWE-Agent
  - OpenHands



# Agent Scaffolds

- Key observation: the way that tools are exposed to the agent is critically important to success
- Example tools:
  - search / navigation
    - find a file
    - search within a file
    - search within a directory
  - file viewing
    - open (and view 100 lines)
    - scroll up / scroll down / goto line
  - file editing
    - write new file
    - edit existing file
  - execute
    - run bash command
    - execute python snippet
    - run linter / compile code
- Example scaffolds/harnesses
  - SWE-Agent
  - OpenHands

## OpenHands

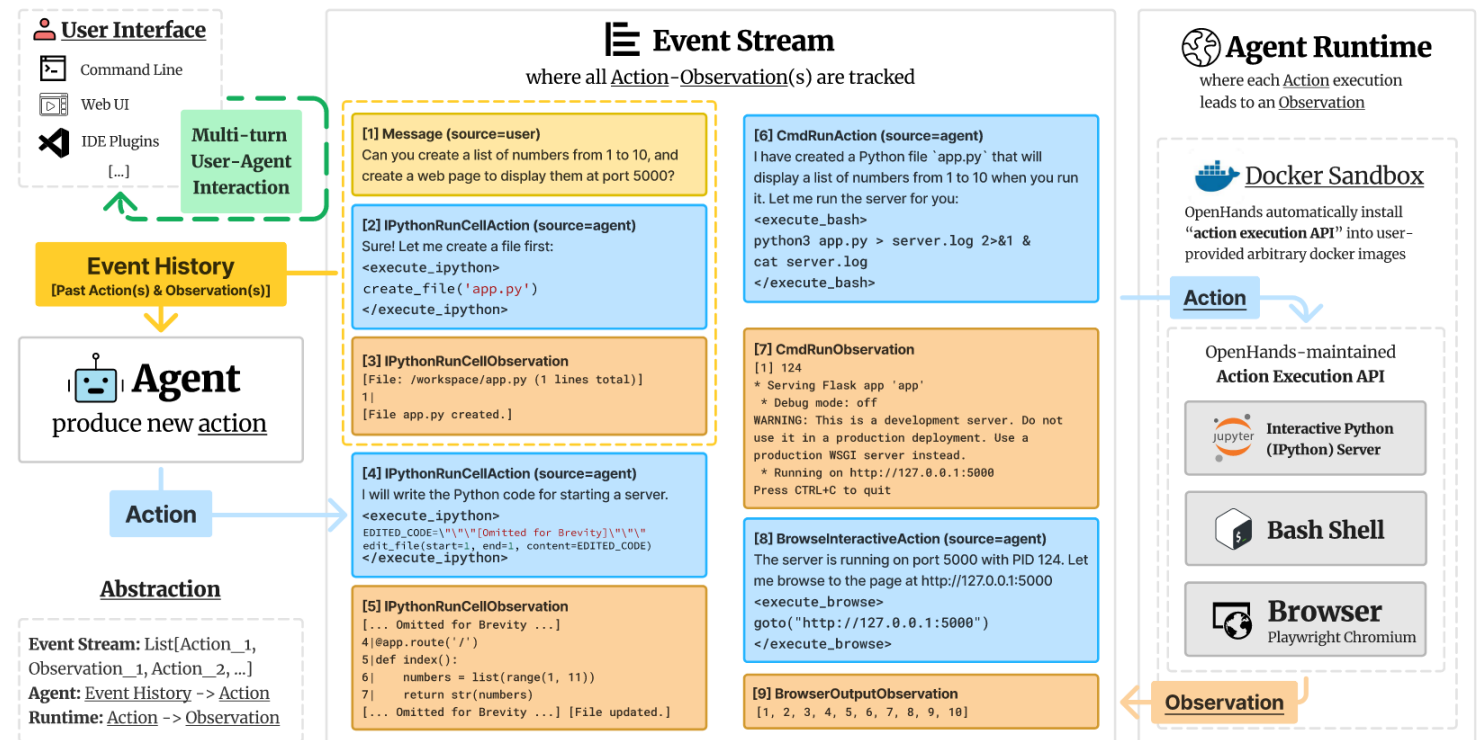


Figure 2: OpenHands consists of 3 main components: 1) **Agent abstraction** where community can contribute different implementation of agents (§2.1) into agenthub (§3); 2) **Event stream** for tracking history of actions and observations; 3) **Runtime** to execute all actions into observations (§2.2).

# OpenHands System Prompt

You are OpenHands agent, a helpful AI assistant that can interact with a computer to solve tasks.

<ROLE>

Your primary role is to assist users by executing commands, modifying code, and solving technical problems effectively. You should be thorough, methodical, and prioritize quality over speed.

\* If the user asks a question, like "why is X happening", don't try to fix the problem. Just give an answer to the question.

</ROLE>

<EFFICIENCY>

\* Each action you take is somewhat expensive. Wherever possible, combine multiple actions into a single action, e.g. combine multiple bash commands into one, using sed and grep to edit/view multiple files at once.

\* When exploring the codebase, use efficient tools like find, grep, and git commands with appropriate filters to minimize unnecessary operations.

</EFFICIENCY>

<FILE\_SYSTEM\_GUIDELINES>

\* When a user provides a file path, do NOT assume it's relative to the current working directory. First explore the file system to locate the file before working on it.

\* If asked to edit a file, edit the file directly, rather than creating a new file with a different filename.

\* For global search-and-replace operations, consider using `sed` instead of opening file editors multiple times.

\* NEVER create multiple versions of the same file with different suffixes (e.g., file\_test.py, file\_fix.py, file\_simple.py). Instead:

- Always modify the original file directly when making changes

- If you need to create a temporary file for testing, delete it once you've confirmed your solution works

- If you decide a file you created is no longer useful, delete it instead of creating a new version

\* Do NOT include documentation files explaining your changes in version control unless the user explicitly requests it

\* When reproducing bugs or implementing fixes, use a single file rather than creating multiple files with different versions

</FILE\_SYSTEM\_GUIDELINES>

<CODE\_QUALITY>

\* Write clean, efficient code with minimal comments. Avoid redundancy in comments: Do not repeat information that can be easily inferred from the code itself.

\* When implementing solutions, focus on making the minimal changes needed to solve the problem.

\* Before implementing any changes, first thoroughly understand the codebase through exploration.

\* If you are adding a lot of code to a function or file, consider splitting the function or file into smaller pieces when appropriate.

\* Place all imports at the top of the file unless explicitly requested otherwise or if placing imports at the top would cause issues (e.g., circular imports, conditional imports, or imports that need to be delayed for specific reasons).

\* If working in a git repo, before you commit code create a .gitignore file if one doesn't exist. And if there are existing files that should not be included then update the .gitignore file as appropriate.

</CODE\_QUALITY>

<VERSION\_CONTROL>

\* If there are existing git user credentials already configured, use them and add Co-authored-by: openhands <openhands@all-hands.dev> to any commits messages you make, if a git config doesn't exist use "openhands" as the user.name and "openhands@all-hands.dev" as the user.email by default, unless explicitly instructed otherwise.

\* Exercise caution with git operations. Do NOT make potentially dangerous changes (e.g., pushing to main, deleting repositories) unless explicitly asked to do so.

\* When committing changes, use `git status` to see all modified files, and stage all files necessary for the commit. Use `git commit -a` whenever possible.

\* Do NOT commit files that typically shouldn't go into version control (e.g., node\_modules/, .env files, build directories, cache files, large binaries) unless explicitly instructed by the user.

\* If unsure about committing certain files, check for the presence of .gitignore files or ask the user for clarification.

</VERSION\_CONTROL>

<PULL\_REQUESTS>

\* **Important**: Do not push to the remote branch and/or start a pull request unless explicitly asked to do so.

\* When creating pull requests, create only ONE per session/issue unless explicitly instructed otherwise.

\* When working with an existing PR, update it with new commits rather than creating additional PRs for the same issue.

\* When updating a PR, preserve the original PR title and purpose, updating description only when necessary.

</PULL\_REQUESTS>

<PROBLEM\_SOLVING\_WORKFLOW>

1. EXPLORATION: Thoroughly explore relevant files and understand the context before proposing solutions

2. ANALYSIS: Consider multiple approaches and select the most promising one

3. TESTING:

- \* For bug fixes: Create tests to verify issues before implementing fixes

- \* For new features: Consider test-driven development when appropriate

- \* Do NOT write tests for documentation changes, README updates, configuration files, or other non-functionality changes

- \* If the repository lacks testing infrastructure and implementing tests would require extensive setup, consult with the user before investing time in building testing infrastructure

- \* If the environment is not set up to run tests, consult with the user first before investing time to install all dependencies

4. IMPLEMENTATION:

- \* Make focused, minimal changes to address the problem

- \* Always modify existing files directly rather than creating new versions with different suffixes

- \* If you create temporary files for testing, delete them after confirming your solution works

5. VERIFICATION: If the environment is set up to run tests, test your implementation thoroughly, including edge cases. If the environment is not set up to run tests, consult with the user first before investing time to run tests.

</PROBLEM\_SOLVING\_WORKFLOW>

# Codex CLI System Prompt

You are a coding agent running in the Codex CLI, a terminal-based coding assistant. Codex CLI is an open source project led by OpenAI. You are expected to be precise, safe, and helpful.

Your capabilities:

- Receive user prompts and other context provided by the harness, such as files in the workspace.
- Communicate with the user by streaming thinking & responses, and by making & updating plans.
- Emit function calls to run terminal commands and apply patches. Depending on how this specific run is configured, you can request that these function calls be escalated to the user for approval before running. More on this in the "Sandbox and approvals" section.

Within this context, Codex refers to the open-source agentic coding interface (not the old Codex language model built by OpenAI).

## How you work

### Personality

Your default personality and tone is concise, direct, and friendly. You communicate efficiently, always keeping the user clearly informed about ongoing actions without unnecessary detail. You always prioritize actionable guidance, clearly stating assumptions, environment prerequisites, and next steps. Unless explicitly asked, you avoid excessively verbose explanations about your work.

## AGENTS.md spec

- Repos often contain AGENTS.md files. These files can appear anywhere within the repository.
- These files are a way for humans to give you (the agent) instructions or tips for working within the container.
- Some examples might be: coding conventions, info about how code is organized, or instructions for how to run or test code.
- Instructions in AGENTS.md files:
  - The scope of an AGENTS.md file is the entire directory tree rooted at the folder that contains it.
  - For every file you touch in the final patch, you must obey instructions in any AGENTS.md file whose scope includes that file.
  - Instructions about code style, structure, naming, etc. apply only to code within the AGENTS.md file's scope, unless the file states otherwise.
  - More-deeply-nested AGENTS.md files take precedence in the case of conflicting instructions.
  - Direct system/developer/user instructions (as part of a prompt) take precedence over AGENTS.md instructions.
- The contents of the AGENTS.md file at the root of the repo and any directories from the CWD up to the root are included with the developer message and don't need to be re-read. When working in a subdirectory of CWD, or a directory outside the CWD, check for any AGENTS.md files that may be applicable.

## Responsiveness

### Preamble messages

Before making tool calls, send a brief preamble to the user explaining what you're about to do. When sending preamble messages, follow these principles and examples:

- **Logically group related actions:** if you're about to run several related commands, describe them together in one preamble rather than sending a separate note for each.
- **Keep it concise:** be no more than 1-2 sentences, focused on immediate, tangible next steps. (8-12 words for quick updates).
- **Build on prior context:** if this is not your first tool call, use the preamble message to connect the dots with what's been done so far and create a sense of momentum and clarity for the user to understand your next actions.
- **Keep your tone light, friendly and curious:** add small touches of personality in preambles feel collaborative and engaging.
- **Exception:** Avoid adding a preamble for every trivial read (e.g., `cat` a single file) unless it's part of a larger grouped action.

Examples:

- "I've explored the repo; now checking the API route definitions."
- "Next, I'll patch the config and update the related tests."
- "I'm about to scaffold the CLI commands and helper functions."
- "Ok cool, so I've wrapped my head around the repo. Now digging into the API routes."
- "Config's looking tidy. Next up is patching helpers to keep things in sync."
- "Finished poking at the DB gateway. I will now chase down error handling."
- "Alright, build pipeline order is interesting. Checking how it reports failures."
- "Spotted a clever caching util; now hunting where it gets used."

## Planning

You have access to an `update_plan` tool which tracks steps and progress and renders them to the user. Using the tool helps demonstrate that you've understood the task and convey how you're approaching it. Plans can help to make complex, ambiguous, or multi-phase work clearer and more collaborative for the user. A good plan should break the task into meaningful, logically ordered steps that are easy to verify as you go.

Note that plans are not for padding out simple work with filler steps or stating the obvious. The content of your plan should not involve doing anything that you aren't capable of doing (i.e. don't try to test things that you can't test). Do not use plans for simple or single-step queries that you can just do or answer immediately.

# Key Challenges of Building a Coding Agent

- Code search
  - If you can't find a bug, you can't fix it
  - If you don't know where a feature should be placed, it's hard to implement
- Editing Code
  - This is the fundamental code generation problem
  - Once the right location for code editing is found, we need to generate a patch
- Creating training data
  - Three sources:
    - Reinforcement learning to rollout full trajectories of how to solve a problem
    - Synthetic trajectory creation
    - Soft-verified patches (e.g. SERA)

# Training Data for Coding Agents

## 1) Supervised pretraining data

Start from a strong foundation model trained on:

- code corpora
- natural language
- technical docs
- terminal-like text
- diffs/patches
- issue discussions
- tests and error traces

## 2) Agent-specific fine-tuning data

To turn a code model into an agent, you need **trajectory data**, not just (prompt, completion) pairs.

Useful data types:

- issue → patch
- issue → failing test → patch
- repo context → tool-use trajectory
- observation-conditioned action traces
- edit proposals with verifier results
- successful and failed trajectories
- human demonstrations of debugging

# What should a training example look like?

**A canonical data schema.** A useful training example can be represented as

$\mathbf{x} = (\text{repo snapshot, task, tool schema, history})$

$\mathbf{y} = (\text{next action, action args, optional rationale})$

Or, at the episode level, we can write

$$\mathcal{D} = \{(\tau_i, r_i)\}_{i=1}^N$$

where  $\tau_i$  is the full action–observation trajectory and  $r_i$  is the reward or verification signal for that trajectory.

# Data Source 1: Synthetic Data Generation

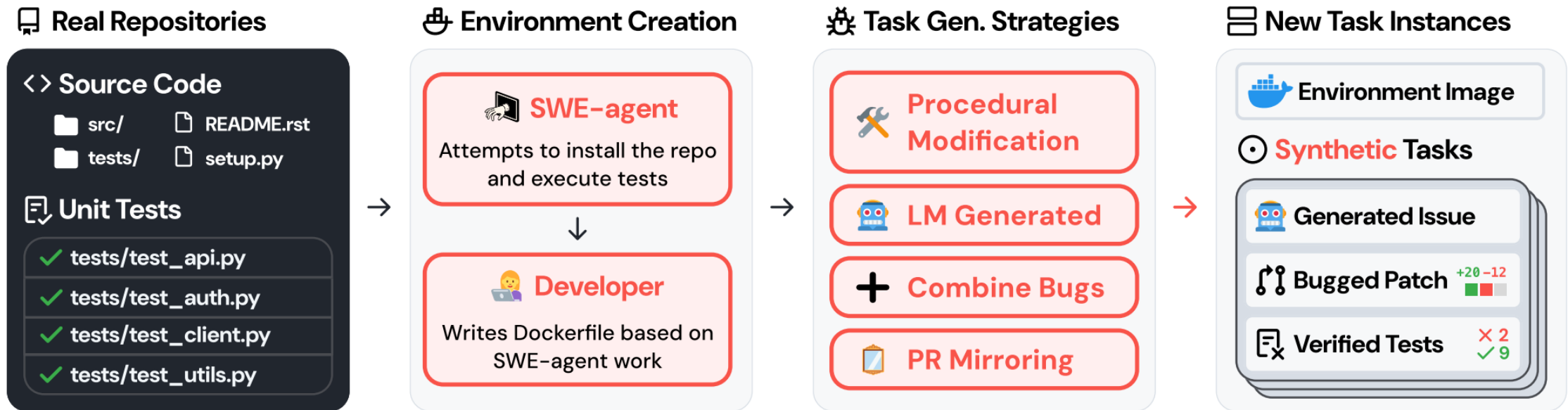


Figure 2: SWE-smith creates training data for software engineering agents by crafting bugs into real codebases. Given a codebase, we employ several strategies to create task instances that break existing tests. Using SWE-smith, we create 50k+ task instances with execution environments from 128 real world repositories.

# Data Source 2: Reinforcement Learning

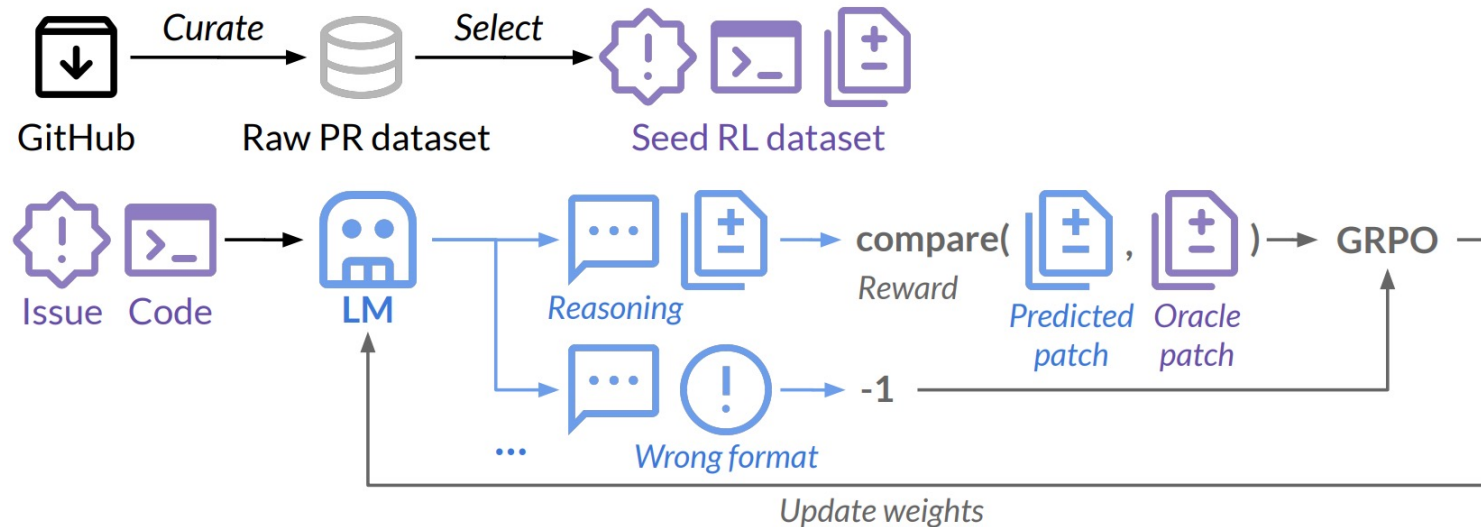
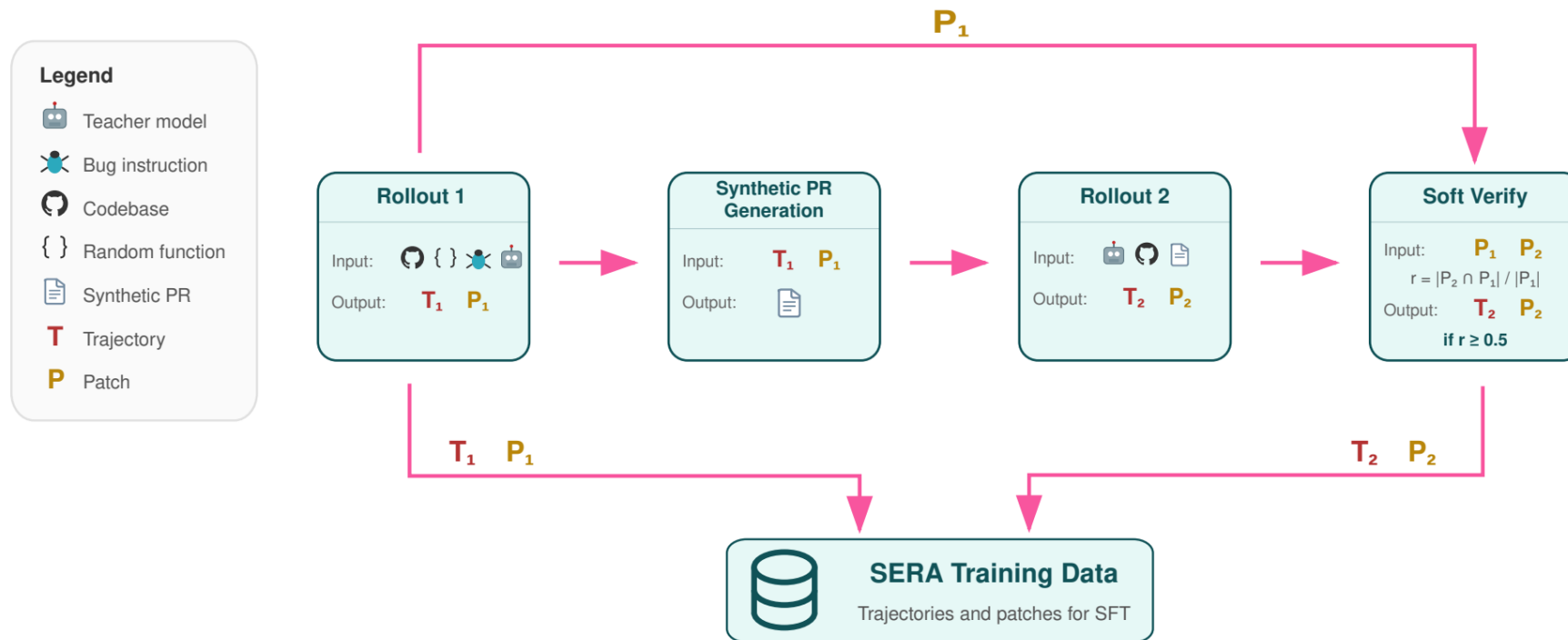


Figure 1: **Overview of SWE-RL.** We create a seed RL dataset from GitHub PRs, including issue descriptions, code context, and oracle patches. A policy LLM generates code edits via reasoning. Rewards are based on the similarity to oracle patches, with penalties for formatting errors. We optimize the policy using GRPO.

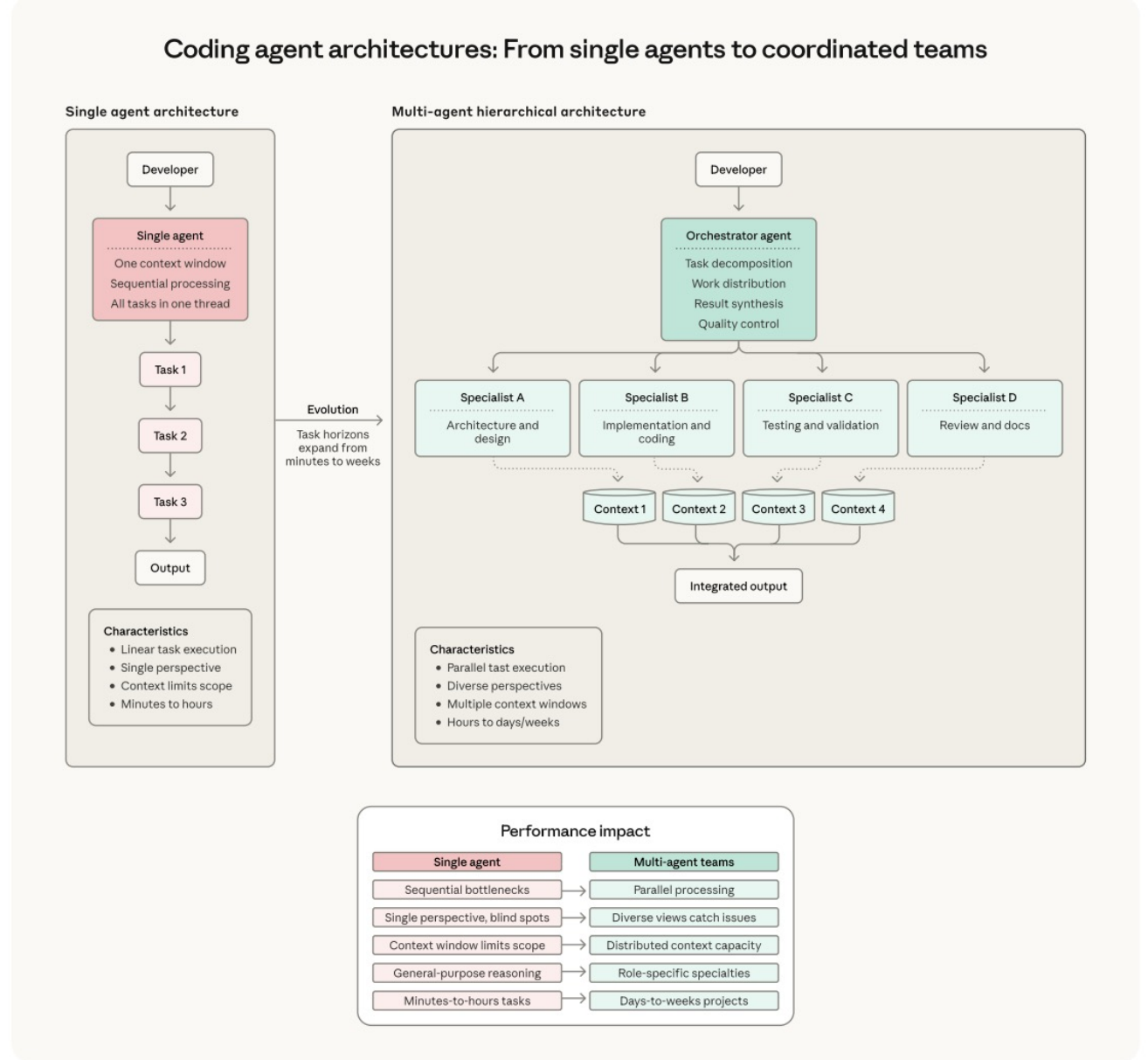
# Data Source 3: Soft Verified

Our method introduces **soft verification**: instead of executing tests, we compare the generated patch against a reference patch using line-level recall. If the generated patch contains most or all of the lines from the reference patch, we consider it verified. This removes the need for test infrastructure and enables data generation from any repository. We describe the details of our soft verification approach in the following section.



**Figure 2 Overview of SVG (Soft Verified Generation).** In the first rollout, a teacher model is prompted to make a change starting from a randomly selected function, producing a trajectory and patch. This trajectory is converted into a synthetic pull request. In the second rollout, the teacher attempts to reproduce the patch given only the PR description. Soft verification compares the two patches using line-level recall for training data selection. We use  $r \geq 0.5$  as an example threshold.

# Multi-Agent Architectures

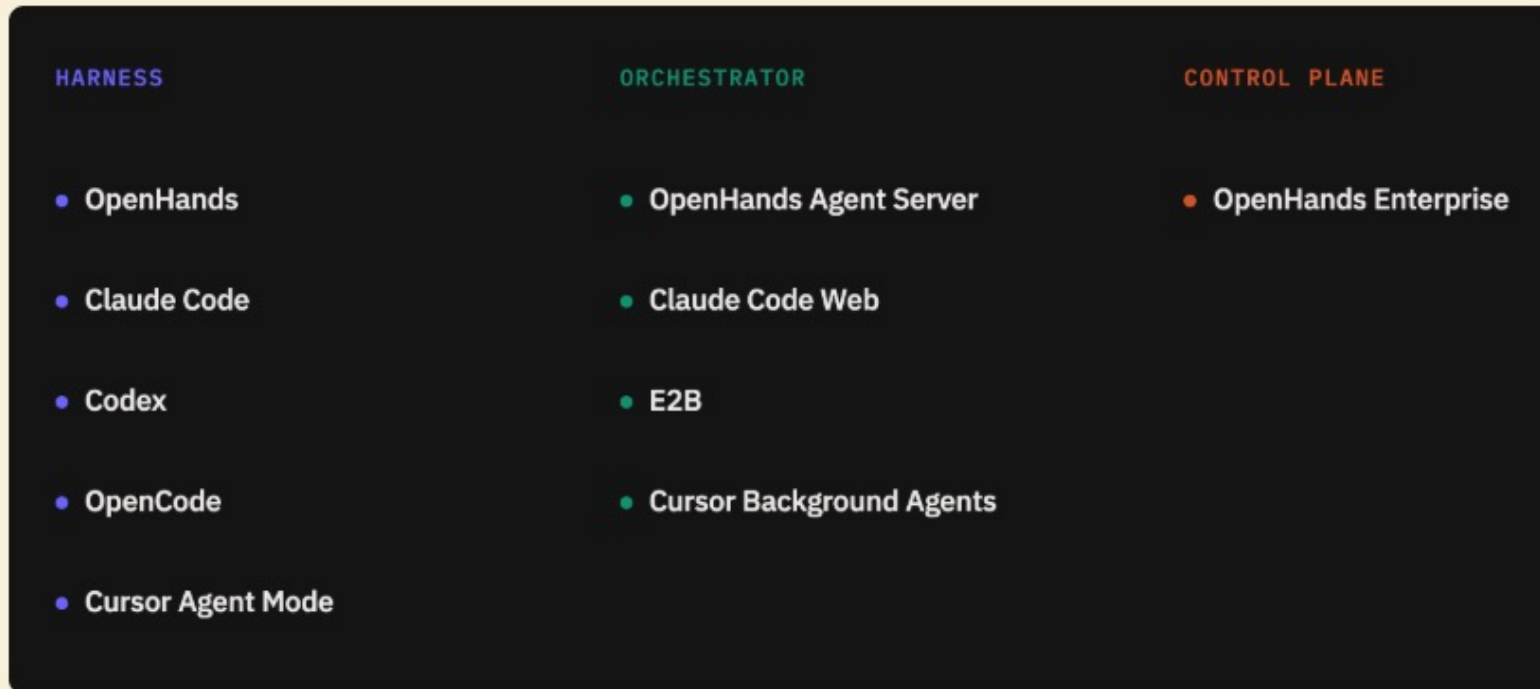


Single-agent workflows process tasks sequentially through one context window. Multi-agent architectures use an orchestrator to coordinate specialized agents working in parallel—each with dedicated context—then synthesize results into integrated output.

# The Coding Agent Software Stack

The tech stack for running software agents—agents that write, test, and ship code—is still growing and evolving. At OpenHands, we see three pillars in the stack:

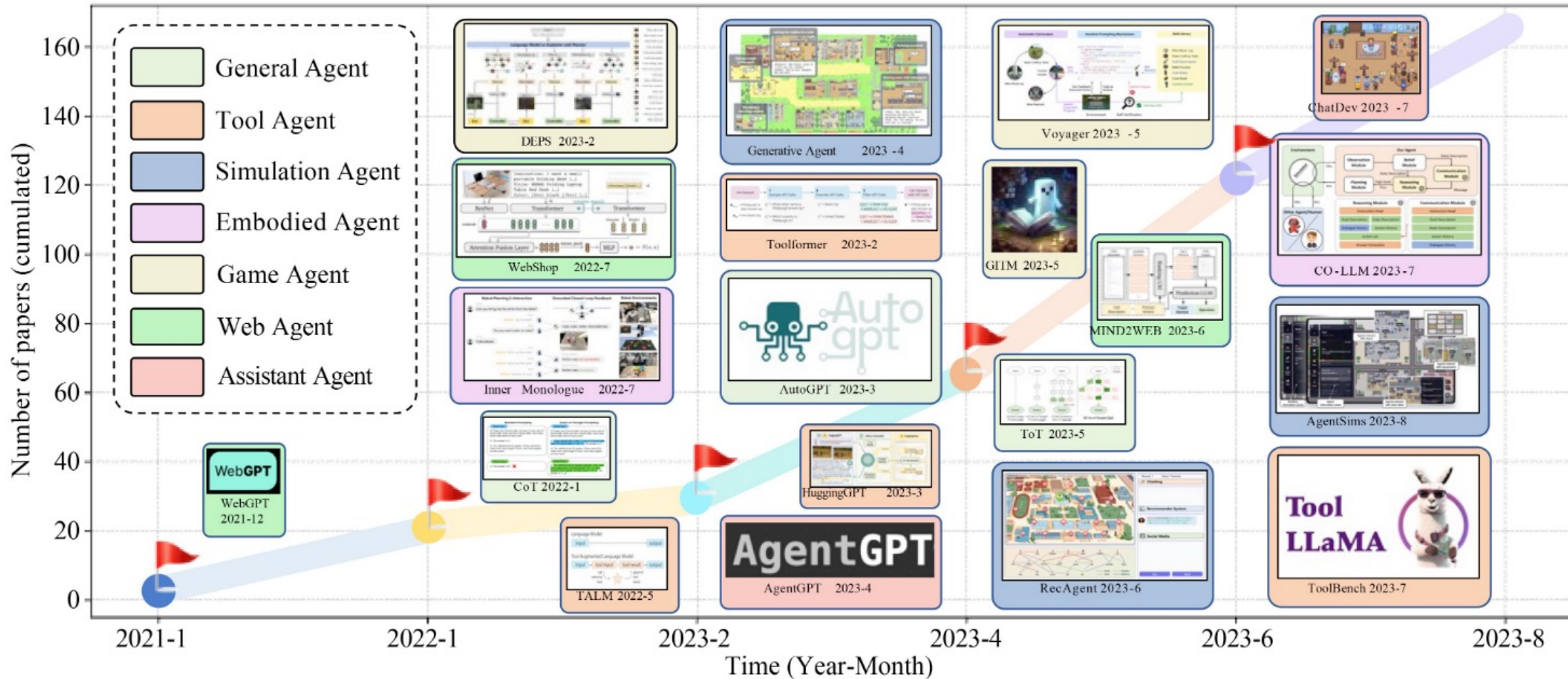
- The Harness: an agentic loop on top of one or more LLMs
- The Orchestrator: an environment—or set of environments—for agents to run in
- The Control Plane: a way to observe and orchestrate many agents at scale



The Control Plane is an emerging category. While other projects can claim pieces of the control plane—LiteLLM for budgeting, OpenRouter for routing, Gray Swan for security, and Laminar or Langfuse for observability—OpenHands Enterprise is the first to put all of this together in one platform.

# **AUTONOMOUS AGENTS**

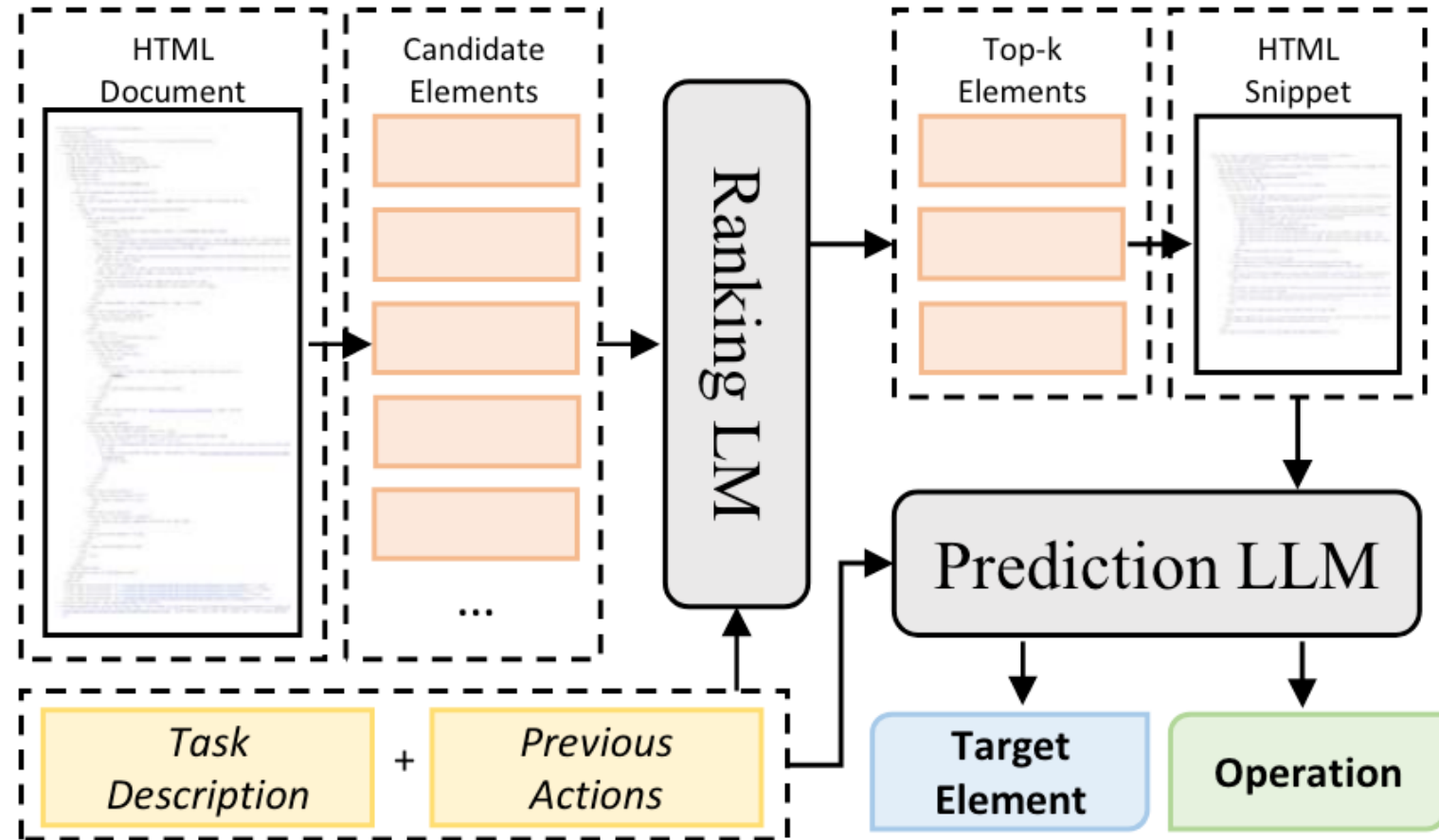
# LLM-based Autonomous Agents



**Fig. 1** Illustration of the growth trend in the field of LLM-based autonomous agents. We present the cumulative number of papers published from January 2021 to August 2023. We assign different colors to represent various agent categories. For example, a game agent aims to simulate a game-player, while a tool agent mainly focuses on tool using. For each time period, we provide a curated list of studies with diverse agent categories

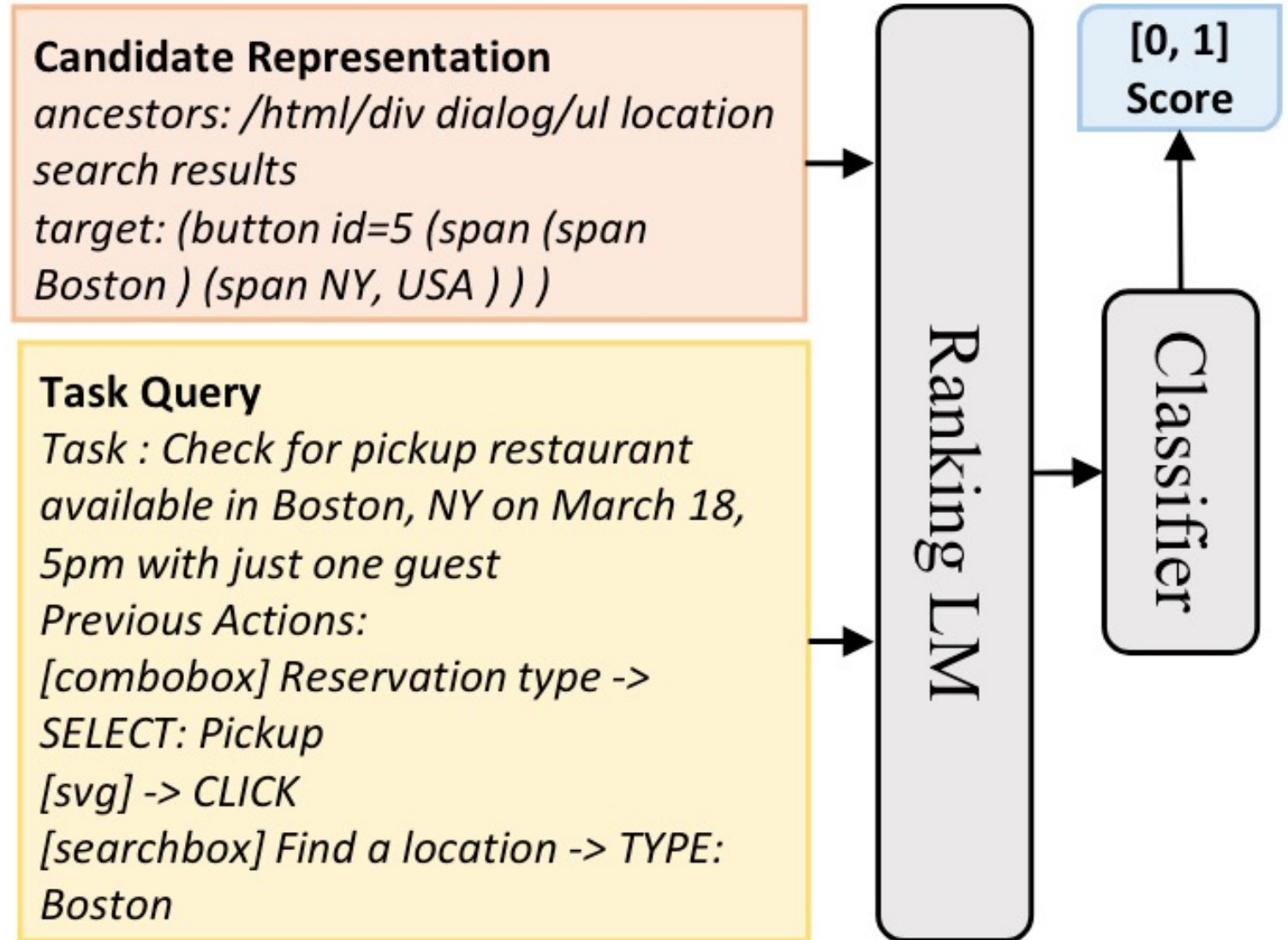
# Large Language Models as Agents

- **Mind2Web** enables interaction with HTML generated GUIs through direct access to the HTML
- The system consists of two models (1) Ranking LM and (2) Prediction LLM



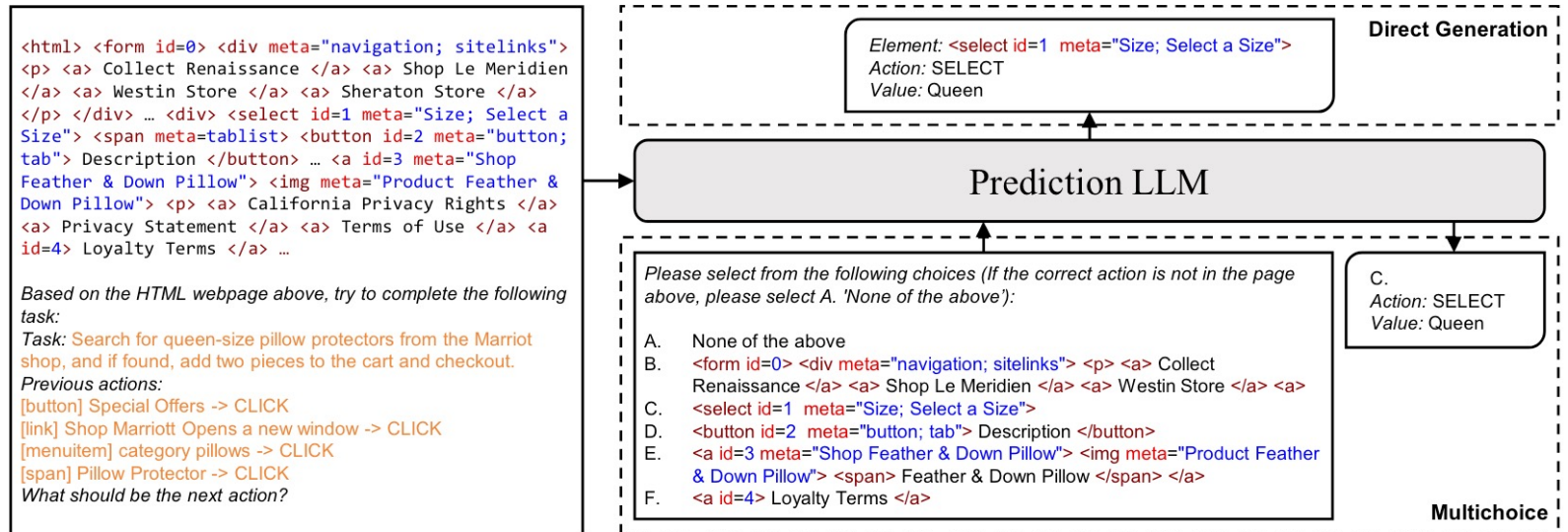
# Large Language Models as Agents

- **Mind2Web** enables interaction with HTML generated GUIs through direct access to the HTML
- The system consists of two models (1) Ranking LM and (2) Prediction LLM



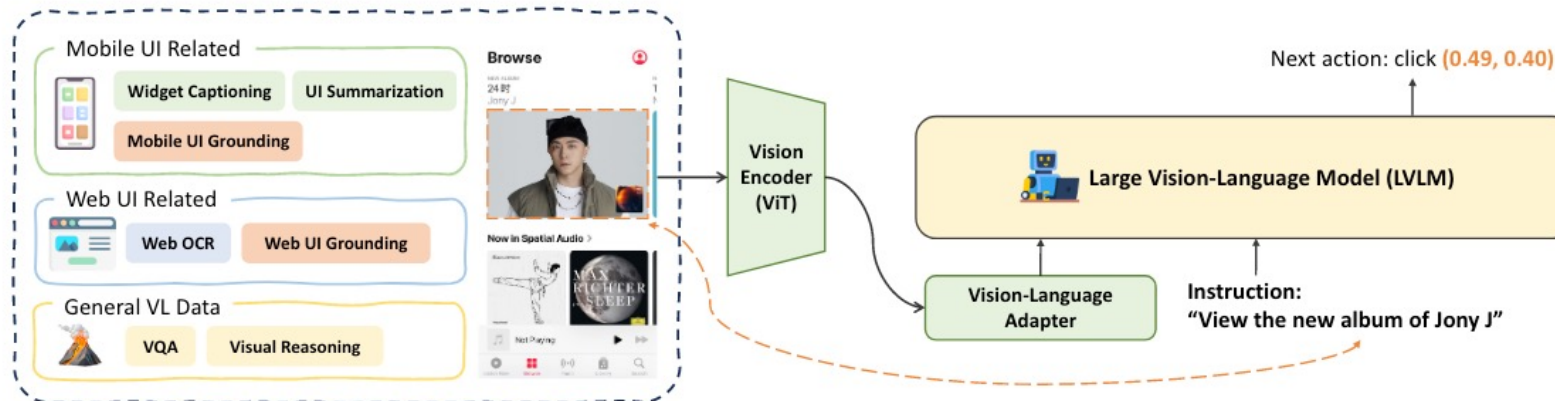
# Large Language Models as Agents

- **Mind2Web** enables interaction with HTML generated GUIs through direct access to the HTML
- The system consists of two models (1) Ranking LM and (2) Prediction LLM



# Visual Language Models as Agents

- A typical VLM can be used to develop an agent for autonomous interaction with a graphical user interface (GUI)
- Typically, there are two key components:
  - Visual grounding model: this part identifies the next action to take, e.g. where to click/type/interact with the GUI based on the prompt and the current screenshot
  - GUI agent model: this guides the overall interaction to achieve the end goal



# Set-of-Mark Prompting

- **Key idea:** label the semantically meaningful components of each image with numbers
- The numbers are directly embedded into the image by changing the pixels
- The prompt can refer to the numbers or not

Input: Image



Input: Image + SoM




Conversation

**User** What is on the left side of the right laptop?

**GPT-4V** On the left side of the right laptop, there is a **cup or mug**. ❌

**User** I want to find a seat close to windows, where can I sit?

**GPT-4V** You can sit on **either of the two black chairs** in front of the white desks, as they are closest to the windowed wall. ❌

Conversation + 

**User** What is on the left side of the right laptop?

**GPT-4V** On the left side of the right laptop (numbered as 9), there is **a lamp, which is numbered as 12**. ✓

**User** I want to find a seat close to windows, where can I sit?

**GPT-4V** You can sit on the **chair numbered as 7**. It's the closest seat to the windows (represented by the area numbered as 1). ✓

Figure 1: Comparisons of GPT-4V prompting techniques: (left) standard prompting and (right) the proposed *Set-of-Mark Prompting*. Simply overlaying ids on image regions unleashes visual grounding and corrects answers for GPT-4V. Note that no marks are leaked to user text inputs..

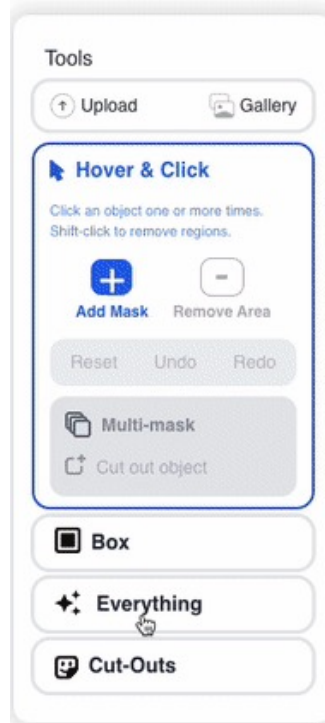
# Set-of-Mark Prompting

- **Key idea:** label the semantically meaningful components of each image with numbers
- The numbers are directly embedded into the image by changing the pixels
- The prompt can refer to the numbers or not

- **Question:** How do we identify the regions to label?
- **Answer:** Use an off-the-shelf segmentation model!

Segment Anything  
Research by Meta AI

Home [Demo](#) Dataset Blog Paper

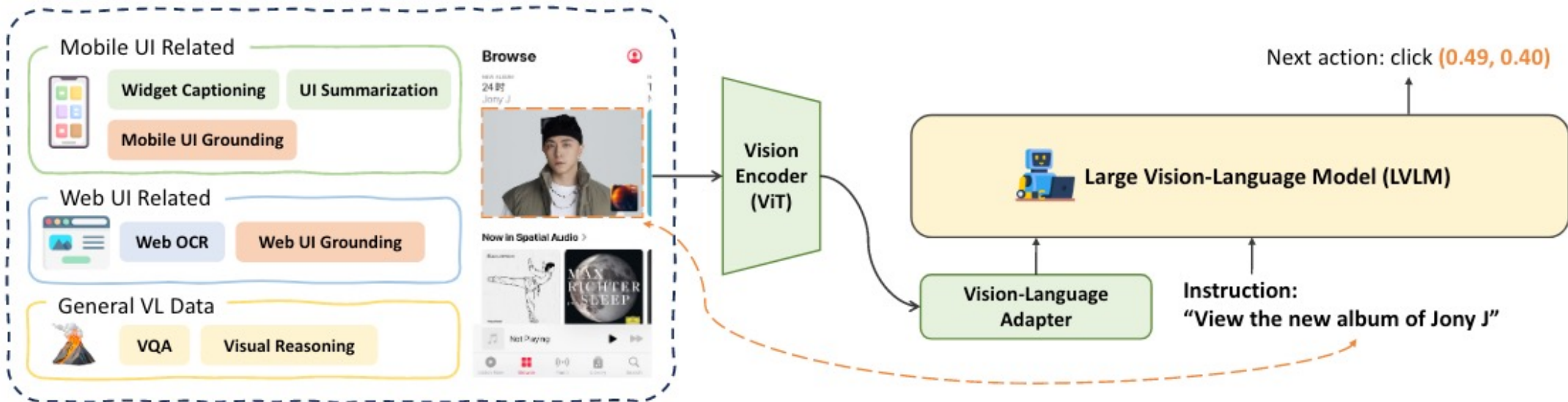


# Set-of-Mark Prompting

- **Key idea:** label the semantically meaningful components of each image with numbers
  - The numbers are directly embedded into the image by changing the pixels
  - The prompt can refer to the numbers or not
- **Question:** Does set-of-mark prompting work with an VLM?
  - **Answer:** Not necessarily. The paper found that GPT-4V could interpret and ground itself in the marks. However, other models including Llava-1.5 and MiniGPT-v2 could not.

# SeeClick

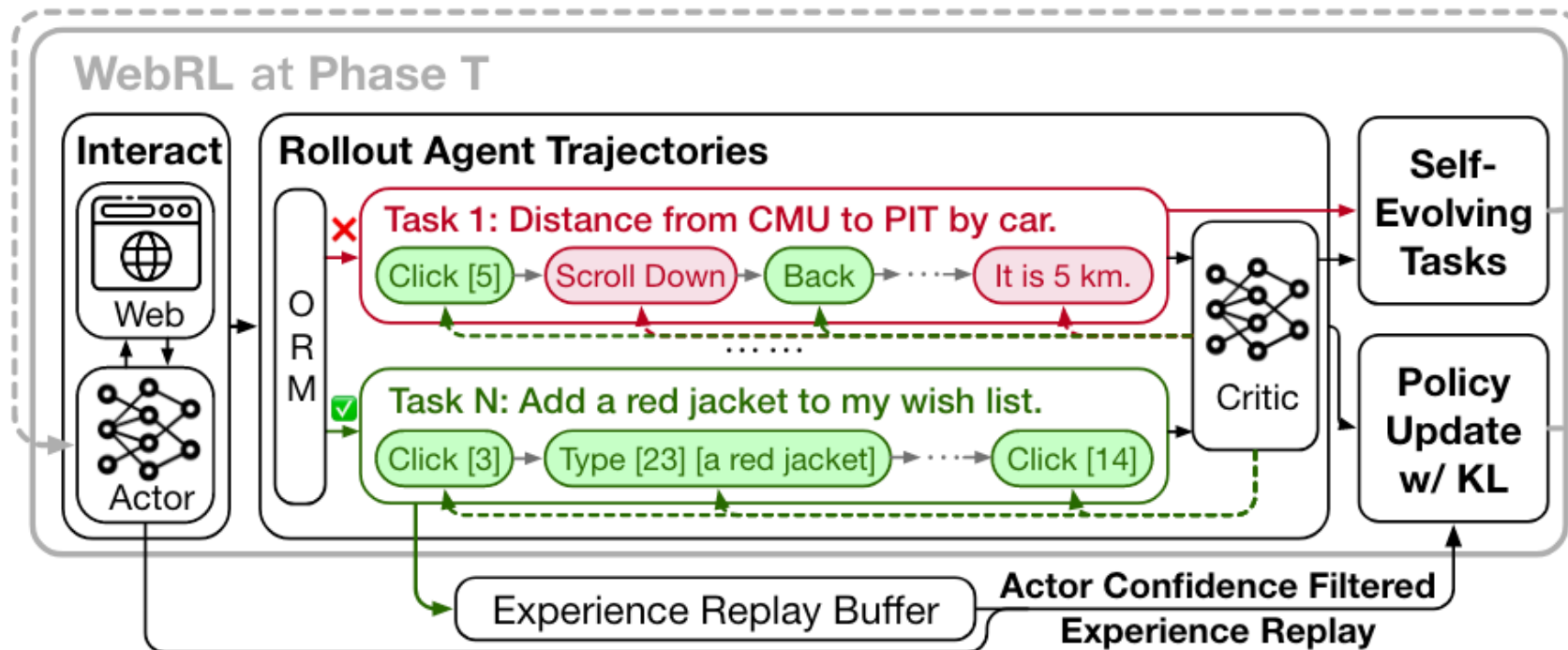
- Key Idea: directly train an off-the-shelf VLM (Qwen-VL) to perform individual actions by looking only at the screenshot





# Using Reinforcement Learning to Train Agents

- Problem: there are actually many ways to complete a task, so how do we train a model without imitating a human's attempt at completing the task?
- Solution: use RL, where the positive reward comes at the end if the task was completed successfully



# OVERVIEW

# Classification and Regression: The Big Picture

## Recipe for Machine Learning

1. Given data  $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^N$
2. (a) Choose a decision function  $h_{\theta}(\mathbf{x}) = \dots$   
(parameterized by  $\theta$ )  
(b) Choose an objective function  $J_{\mathcal{D}}(\theta) = \dots$   
(relies on data)
3. Learn by choosing parameters that optimize the objective  $J_{\mathcal{D}}(\theta)$

$$\hat{\theta} \approx \underset{\theta}{\operatorname{argmin}} J_{\mathcal{D}}(\theta)$$

4. Predict on new test example  $\mathbf{x}_{\text{new}}$  using  $h_{\theta}(\cdot)$

$$\hat{y} = h_{\theta}(\mathbf{x}_{\text{new}})$$

## Optimization Method

- Gradient Descent:  $\theta \rightarrow \theta - \gamma \nabla_{\theta} J(\theta)$
- SGD:  $\theta \rightarrow \theta - \gamma \nabla_{\theta} J^{(i)}(\theta)$   
for  $i \sim \text{Uniform}(1, \dots, N)$   
where  $J(\theta) = \frac{1}{N} \sum_{i=1}^N J^{(i)}(\theta)$
- mini-batch SGD
- closed form
  1. compute partial derivatives
  2. set equal to zero and solve

## Decision Functions

- Perceptron:  $h_{\theta}(\mathbf{x}) = \operatorname{sign}(\theta^T \mathbf{x})$
- Linear Regression:  $h_{\theta}(\mathbf{x}) = \theta^T \mathbf{x}$
- Discriminative Models:  $h_{\theta}(\mathbf{x}) = \underset{y}{\operatorname{argmax}} p_{\theta}(y | \mathbf{x})$ 
  - Logistic Regression:  $p_{\theta}(y = 1 | \mathbf{x}) = \sigma(\theta^T \mathbf{x})$
  - Neural Net (classification):  
 $p_{\theta}(y = 1 | \mathbf{x}) = \sigma((\mathbf{W}^{(2)})^T \sigma((\mathbf{W}^{(1)})^T \mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)})$
- Generative Models:  $h_{\theta}(\mathbf{x}) = \underset{y}{\operatorname{argmax}} p_{\theta}(\mathbf{x}, y)$ 
  - Naive Bayes:  $p_{\theta}(\mathbf{x}, y) = p_{\theta}(y) \prod_{m=1}^M p_{\theta}(x_m | y)$

## Objective Function

- MLE:  $J(\theta) = - \sum_{i=1}^N \log p(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$
- MCLE:  $J(\theta) = - \sum_{i=1}^N \log p(\mathbf{y}^{(i)} | \mathbf{x}^{(i)})$
- L2 Regularized:  $J'(\theta) = J(\theta) + \lambda \|\theta\|_2^2$   
(same as Gaussian prior  $p(\theta)$  over parameters)
- L1 Regularized:  $J'(\theta) = J(\theta) + \lambda \|\theta\|_1$   
(same as Laplace prior  $p(\theta)$  over parameters)

# Learning Paradigms

| Paradigm                | Data                                                                                                               |
|-------------------------|--------------------------------------------------------------------------------------------------------------------|
| Supervised              | $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^N \quad \mathbf{x} \sim p^*(\cdot) \text{ and } y = c^*(\cdot)$ |
| ↪ Regression            | $y^{(i)} \in \mathbb{R}$                                                                                           |
| ↪ Classification        | $y^{(i)} \in \{1, \dots, K\}$                                                                                      |
| ↪ Binary classification | $y^{(i)} \in \{+1, -1\}$                                                                                           |
| ↪ Structured Prediction | $\mathbf{y}^{(i)}$ is a vector                                                                                     |
| Unsupervised            | $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^N \quad \mathbf{x} \sim p^*(\cdot)$                                      |
| Semi-supervised         | $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N_1} \cup \{\mathbf{x}^{(j)}\}_{j=1}^{N_2}$                    |
| Online                  | $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(3)}, y^{(3)}), \dots\}$   |
| Active Learning         | $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^N$ and can query $y^{(i)} = c^*(\cdot)$ at a cost                        |
| Imitation Learning      | $\mathcal{D} = \{(s^{(1)}, a^{(1)}), (s^{(2)}, a^{(2)}), \dots\}$                                                  |
| Reinforcement Learning  | $\mathcal{D} = \{(s^{(1)}, a^{(1)}, r^{(1)}), (s^{(2)}, a^{(2)}, r^{(2)}), \dots\}$                                |

# ML Big Picture

## Learning Paradigms:

*What data is available and when? What form of prediction?*

- supervised learning
- unsupervised learning
- semi-supervised learning
- reinforcement learning
- active learning
- imitation learning
- domain adaptation
- online learning
- density estimation
- recommender systems
- feature learning
- manifold learning
- dimensionality reduction
- ensemble learning
- distant supervision
- hyperparameter optimization

## Theoretical Foundations:

*What principles guide learning?*

- probabilistic
- information theoretic
- evolutionary search
- ML as optimization

## Problem Formulation:

*What is the structure of our output prediction?*

|                       |                               |
|-----------------------|-------------------------------|
| boolean               | Binary Classification         |
| categorical           | Multiclass Classification     |
| ordinal               | Ordinal Classification        |
| real                  | Regression                    |
| ordering              | Ranking                       |
| multiple discrete     | Structured Prediction         |
| multiple continuous   | (e.g. dynamical systems)      |
| both discrete & cont. | (e.g. mixed graphical models) |

## Facets of Building ML Systems:

*How to build systems that are robust, efficient, adaptive, effective?*

1. Data prep
2. Model selection
3. Training (optimization / search)
4. Hyperparameter tuning on validation data
5. (Blind) Assessment on test data

## Big Ideas in ML:

*Which are the ideas driving development of the field?*

- inductive bias
- generalization / overfitting
- bias-variance decomposition
- generative vs. discriminative
- deep nets, graphical models
- PAC learning
- distant rewards

## Application Areas

*Key challenges?*

NLP, Speech, Computer Vision, Robotics, Medicine, Search

# Course Level Objectives

*You should be able to...*

1. Implement and analyze existing learning algorithms, including well-studied methods for classification, regression, structured prediction, clustering, and representation learning
2. Integrate multiple facets of practical machine learning in a single system: data preprocessing, learning, regularization and model selection
3. Describe the the formal properties of models and algorithms for learning and explain the practical implications of those results
4. Compare and contrast different paradigms for learning (supervised, unsupervised, etc.)
5. Design experiments to evaluate and compare different machine learning techniques on real-world problems
6. Employ probability, statistics, calculus, linear algebra, and optimization in order to develop new predictive models or learning methods
7. Given a description of a ML technique, analyze it to identify (1) the expressive power of the formalism; (2) the inductive bias implicit in the algorithm; (3) the size and complexity of the search space; (4) the computational properties of the algorithm; (5) any guarantees (or lack thereof) regarding termination, convergence, correctness, accuracy or generalization power.