CS388: Natural Language Processing

Lecture 20: Language and Code



credit: Deepmind

TEXAS
The University of Texas at Austin

Slides credit: Greg Durrett

Announcements

- HW4 is out. Q1: 3rd error, we will accept any 2
- Quiz 4: Nov 17
- Final projects: start working on them, don’t leave it to the last week.



This Lecture

- Semantic parsing
  - Logical forms
  - Parsing to lambda calculus
  - Seq2seq semantic parsing
- Language-to-code
- Coding agents

Semantic Parsing

Model Theoretic Semantics

- Key idea: can ground out natural language expressions in set-theoretic expressions called *models* of those sentences
- Natural language statement S => interpretation of S that models it

  She likes going to that restaurant

  - Interpretation: defines who *she* and *that restaurant* are, make it able to be concretely evaluated with respect to a *world*
- This is a type of truth-conditional semantics: reduce a sentence to its truth conditions (configuration of the world under which it is true)
- Our modeling language is *first-order logic*
- Entailment (statement A implies statement B) reduces to: in all worlds where A is true, B is true

First-order Logic

- Powerful logic formalism including things like entities, relations, and quantifications

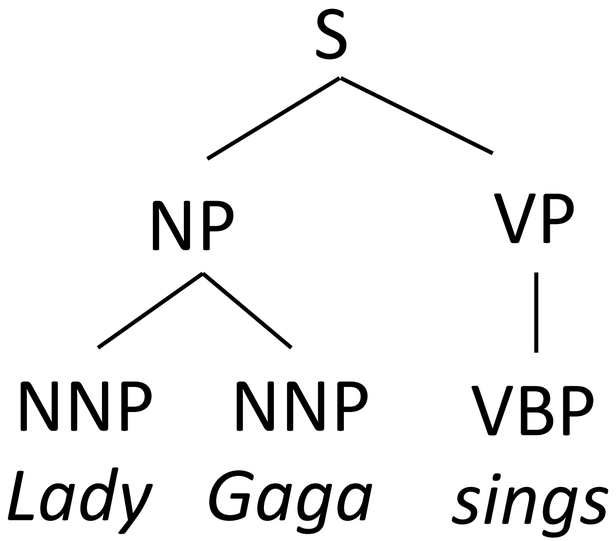
Lady Gaga sings

- sings is a *predicate* (with one argument), function f: entity ⟶ true/false
- sings(Lady Gaga) = true or false, have to execute this against some database (*world*)
- Quantification: “forall” operator, “there exists” operator

$\forall x \; \text{sings}(x) \vee \text{dances}(x) \longrightarrow \text{performs}(x)$

“Everyone who sings or dances performs”

Montague Semantics



Id	Name	Alias	Birthdate	Sings?
e470	Stefani Germanotta	Lady Gaga	3/28/1986	T
e728	Marshall Mathers	Eminem	10/17/1972	T

Database containing entities, predicates, etc.

- Richard Montague: operationalized this type of semantics and connected it to syntax
- Denotation: evaluation of some expression against this database

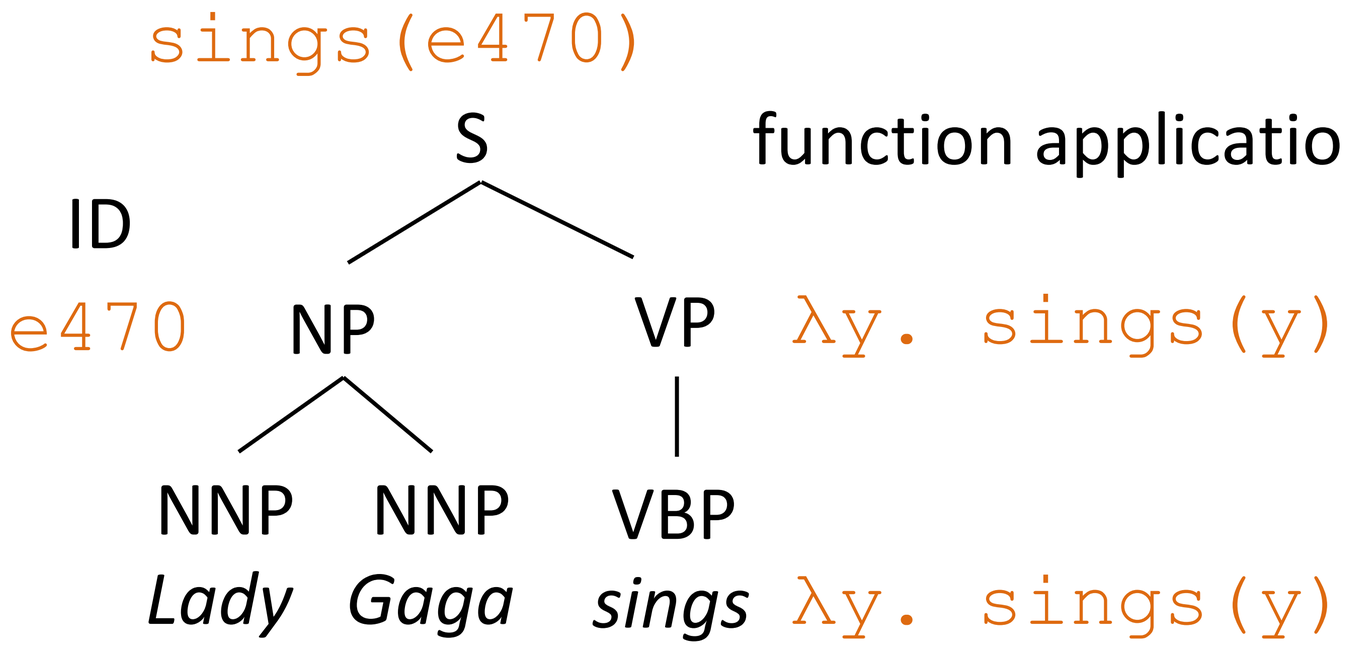
`[[`*Lady Gaga*`]] = e470`

denotation of this string is an entity

`[[sings(e470)]] = True`

denotation of this expression is T/F

Montague Semantics



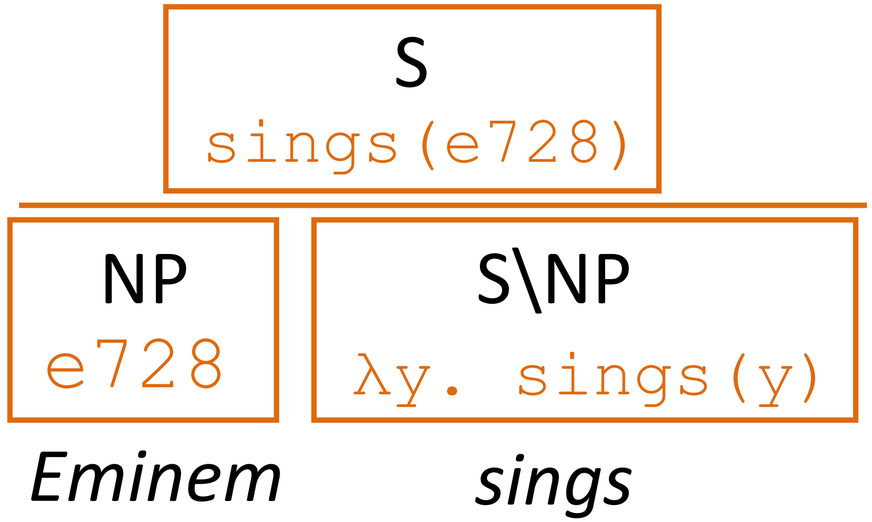
function application: apply this to e470

takes one argument (y, the entity)
and returns a logical form `sings(y)`

- We can use the syntactic parse as a bridge to the lambda-calculus representation, build up a logical form (our model) *compositionally*

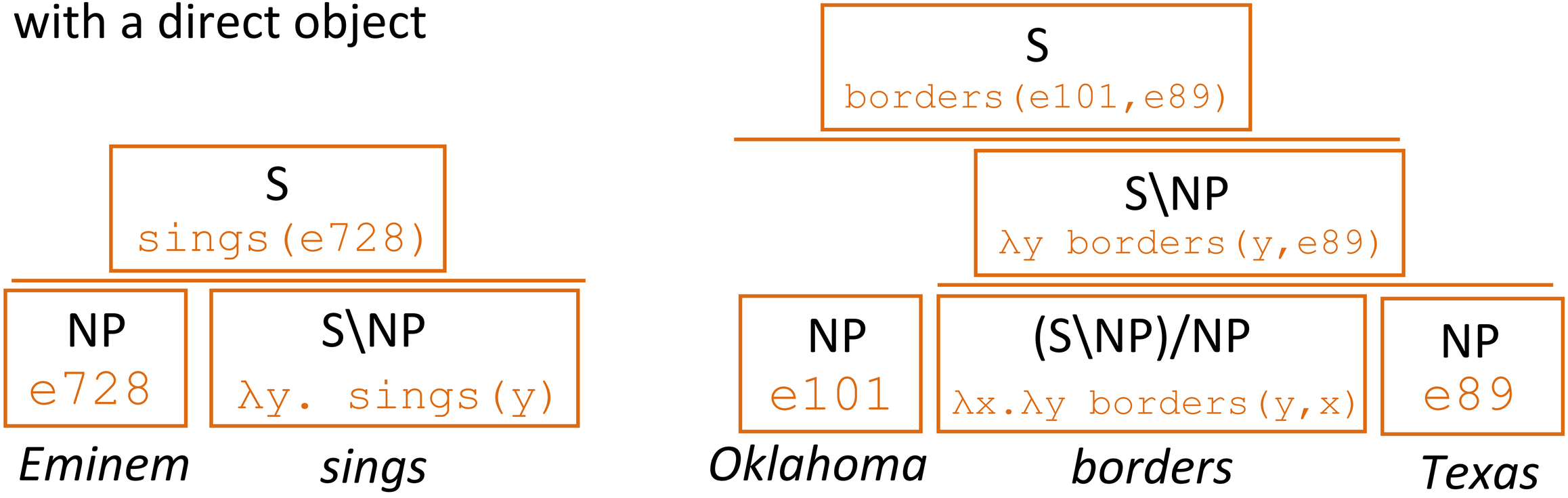
Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Parallel derivations of syntactic parse and lambda calculus expression
- Syntactic categories (for this lecture): S, NP, "slash" categories
- S\NP: "if I combine with an NP on my left side, I form a sentence" — verb
- When you apply this, there has to be a parallel instance of function application on the semantics side

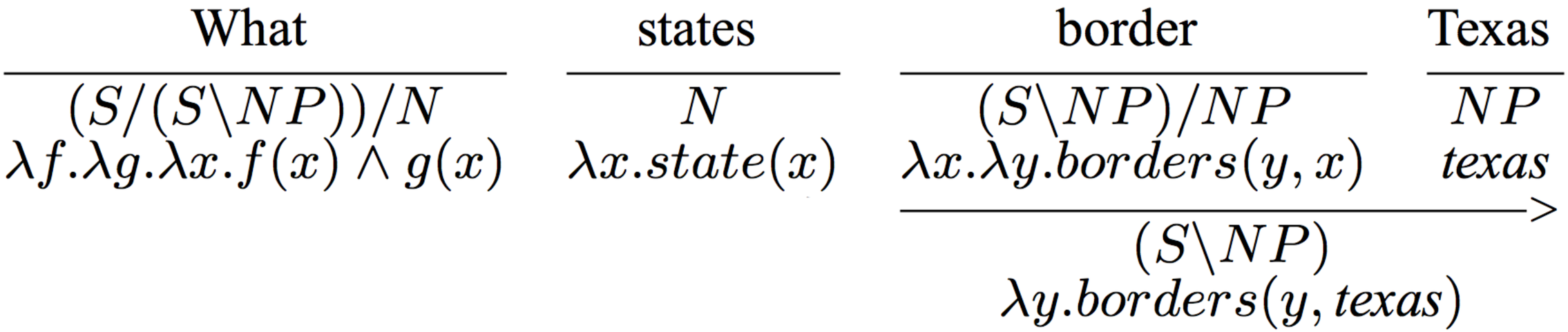


Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Syntactic categories (for this lecture): S, NP, “slash” categories
  - S\NP: “if I combine with an NP on my left side, I form a sentence” — verb
  - (S\NP)/NP: “I need an NP on my right and then on my left” — verb with a direct object



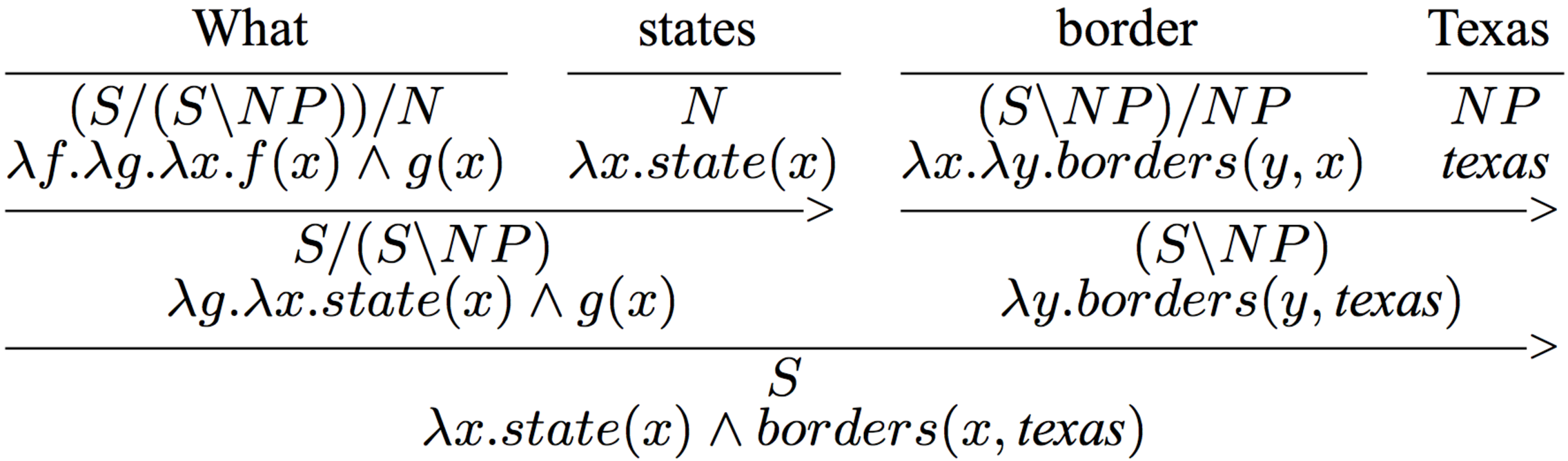
CCG Parsing



- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)

Zettlemoyer and Collins (2005)

CCG Parsing



- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)
- **Why are we talking about this in this lecture? Because this lambda calculus expression is basically executable code**.

Zettlemoyer and Collins (2005)

CCG Parsing

- These question are *compositional*: we can build bigger ones out of smaller pieces

What states border Texas?

What states border states bordering Texas?

What states border states bordering states bordering Texas?

Zettlemoyer and Collins (2005)



Training CCG Parsers

- Training data looks like pairs of sentences and logical forms

What states border Texas `λx. state(x) ∧ borders(x, e89)`

What borders Texas `λx. borders(x, e89)`

…

- Unlike PCFGs, we don't know which words yielded which fragments of CCG
- Very hard to build a conventional parser for this problem

Zettlemoyer and Collins (2005)



Semantic Parsing as Translation

“what states border Texas”

↓

```
lambda x ( state ( x ) and border ( x , e89 ) ) )
```

- Write down a linearized form of the semantic parse, train seq2seq models to directly translate into this representation (similar to code generation like GitHub Copilot)
- What are some benefits of this approach compared to grammar-based?
- What might be some concerns about this approach? How do we mitigate them?

Jia and Liang (2016)

Applications

- GeoQuery (Zelle and Mooney, 1996): answering questions about states (~80% accuracy)
- Jobs: answering questions about job postings (~80% accuracy)
- ATIS: flight search
- Can do well on all of these tasks if you handcraft systems and use plenty of training data: these domains aren't that complex and models these days can produce well-formed outputs



Ray Mooney:
UT Austin legend



Code Generation

- Suppose we are going to generate source code like in Codex/GitHub Copilot. What differs from generating natural language?

- What differs from semantic parsing?

- In spite of these differences, the “obvious” thing is to do some pre-training and see how far we get!

Generating Code

CodeT5

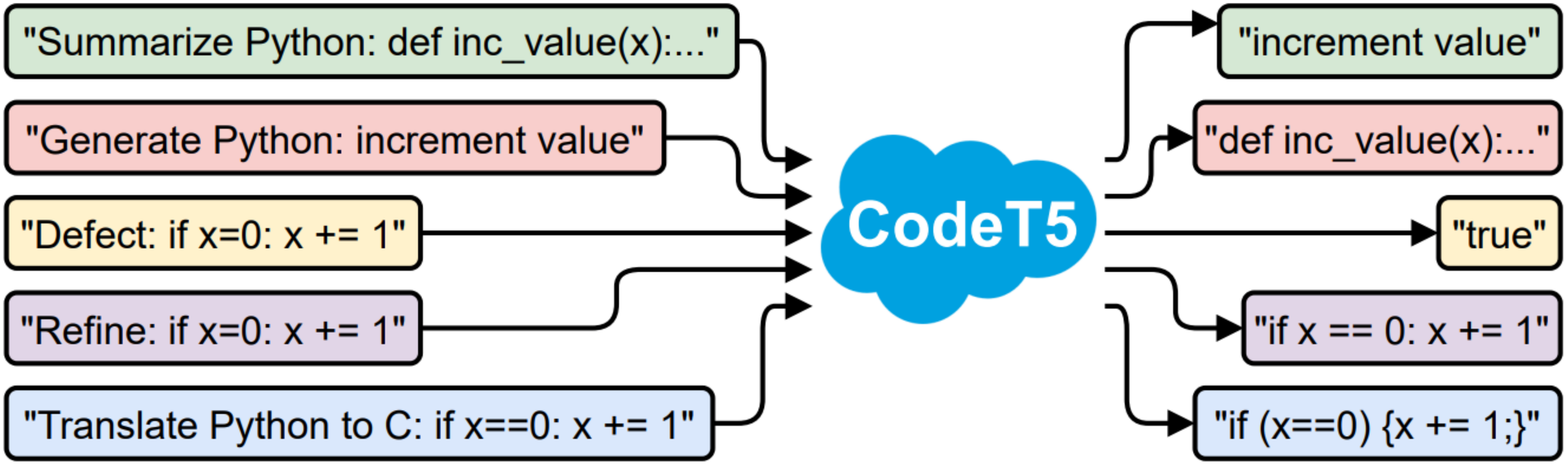
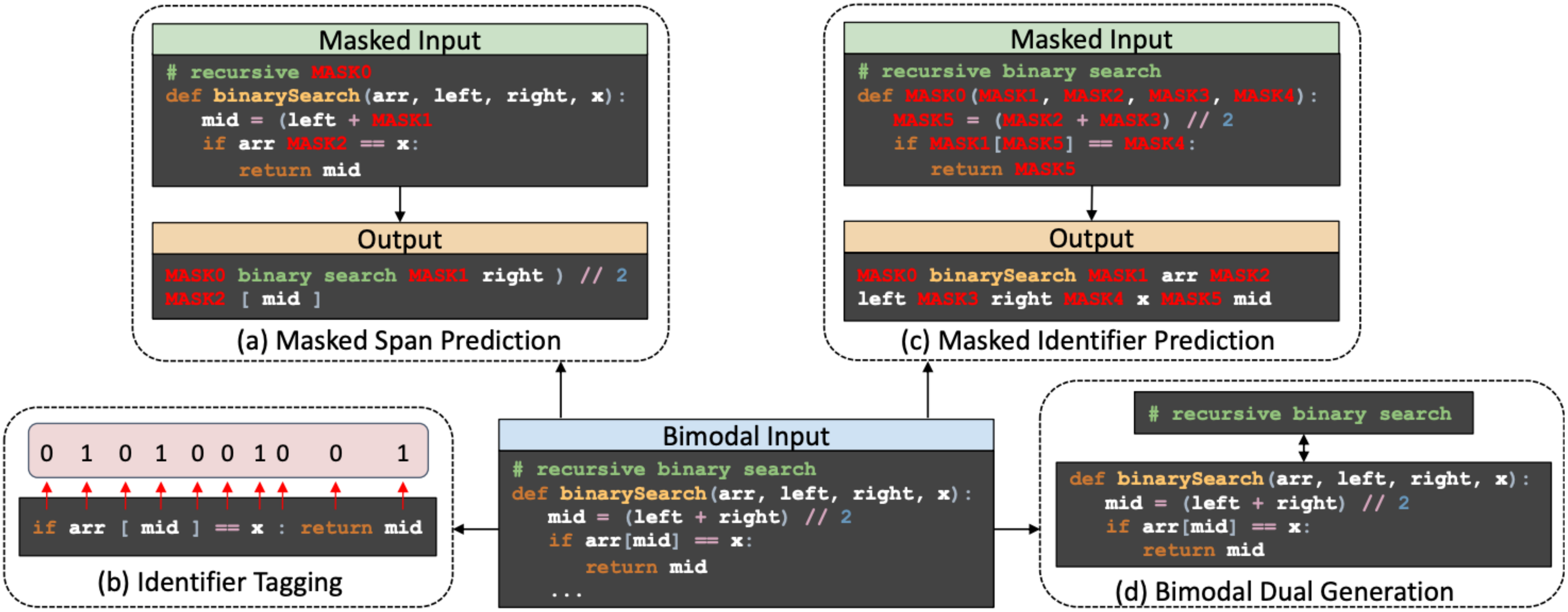


Figure 1: Illustration of our CodeT5 for code-related understanding and generation tasks.

- Key idea: code analogue of T5 that should be able to map language to source code

Wang et al. (2021)

CodeT5



- Predict (a) spans; (c) identifiers; (d) language from code and vice versa
- What's different from normal T5?

Wang et al. (2021)

CodeT5

- Pre-trained on data from several language and NL
- Applied to several generation tasks: code summarization, generation, and translation (between programming languages)

	PLs	W/ NL	W/o NL	Identifier
CodeSearchNet	Ruby	49,009	110,551	32.08%
	JavaScript	125,166	1,717,933	19.82%
	Go	319,132	379,103	19.32%
	Python	453,772	657,030	30.02%
	Java	457,381	1,070,271	25.76%
	PHP	525,357	398,058	23.44%
Our	C	1M	-	24.94%
	CSharp	228,496	856,375	27.85%
	Total	3,158,313	5,189,321	8,347,634

- Also used for classification like bug detection (can be fine-tuned like BERT-style models)

Wang et al. (2021)

CodeT5

- Generation task from CONCODE (Iyer et al., 2018):

```
public class SimpleVector implements Serializable {
  double[] vecElements;
  double[] weights;

  NL Query: Adds a scalar to this vector in place.
  Code to be generated automatically:
  public void add(final double arg0) {
    for (int i = 0; i < vecElements.length; i++){
      vecElements[i] += arg0;
    }
  }
```

- What do you think about this evaluation?

Methods	EM	BLEU	CodeBLEU
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CodeT5-small	21.55	38.13	41.39
+dual-gen	19.95	39.02	42.21
+multi-task	20.15	35.89	38.83
CodeT5-base	22.30	40.73	43.20
+dual-gen	**22.70**	**41.48**	**44.10**
+multi-task	21.15	37.54	40.01

Table 3: Results on the code generation task. EM denotes the exact match.

Wang et al. (2021)



Codex

- GPT-3 additionally fine-tuned on code (although they state that pre-training on NL isn't really helpful)
  - Modified tokenizer to handle whitespace better. Otherwise, no real modifications!
- Up to 12B parameter models fine-tuned on Python
- One challenge is evaluation. How to go beyond BLEU/EM?

Mark Chen et al. (2021)

HumanEval

- Generate standalone Python functions from docstrings **and execute them!**

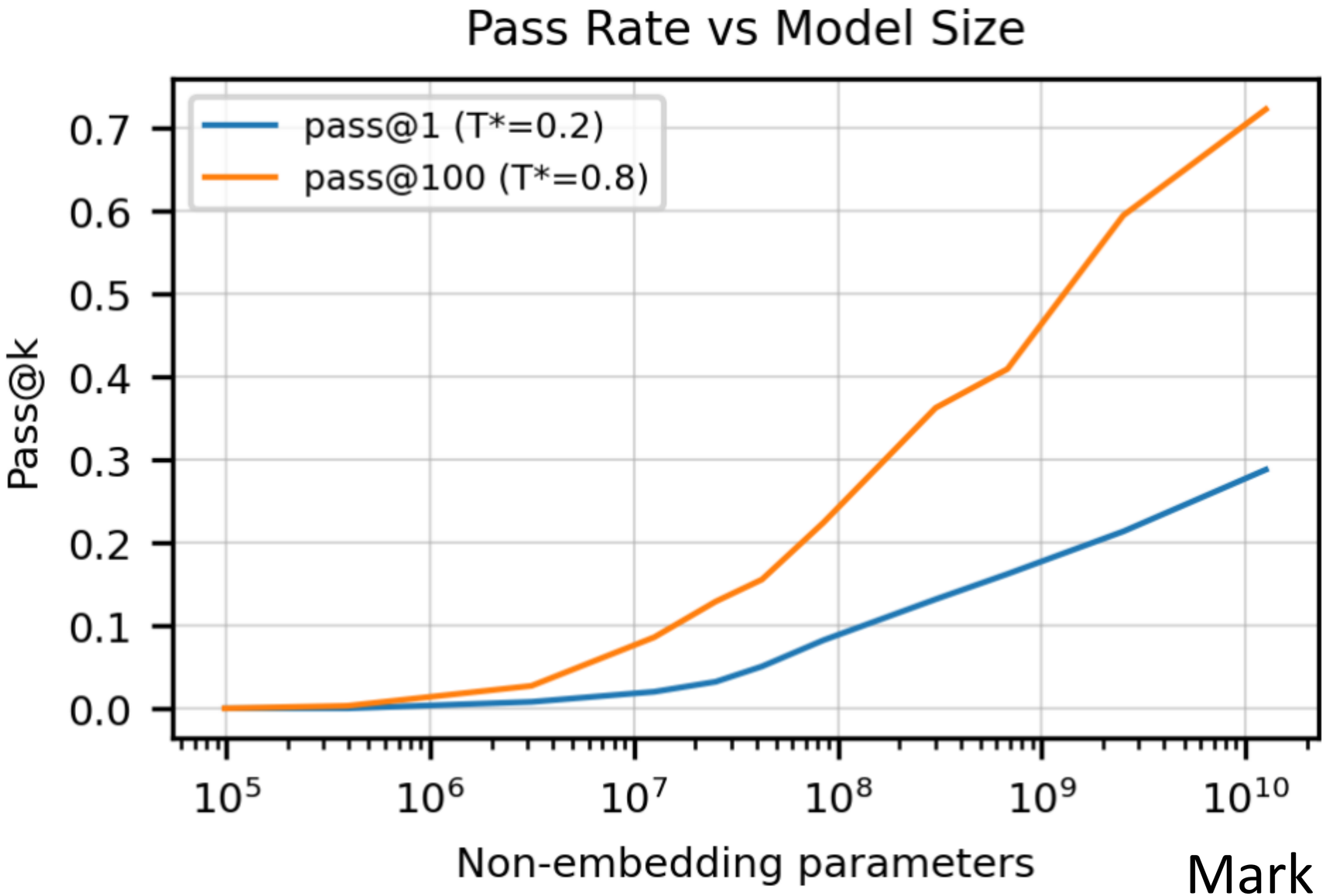
```
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    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)
```

- Handwritten benchmarks evaluated for correctness (“pass@k”: generate k, see if one of them works)

Mark Chen et al. (2021)

HumanEval

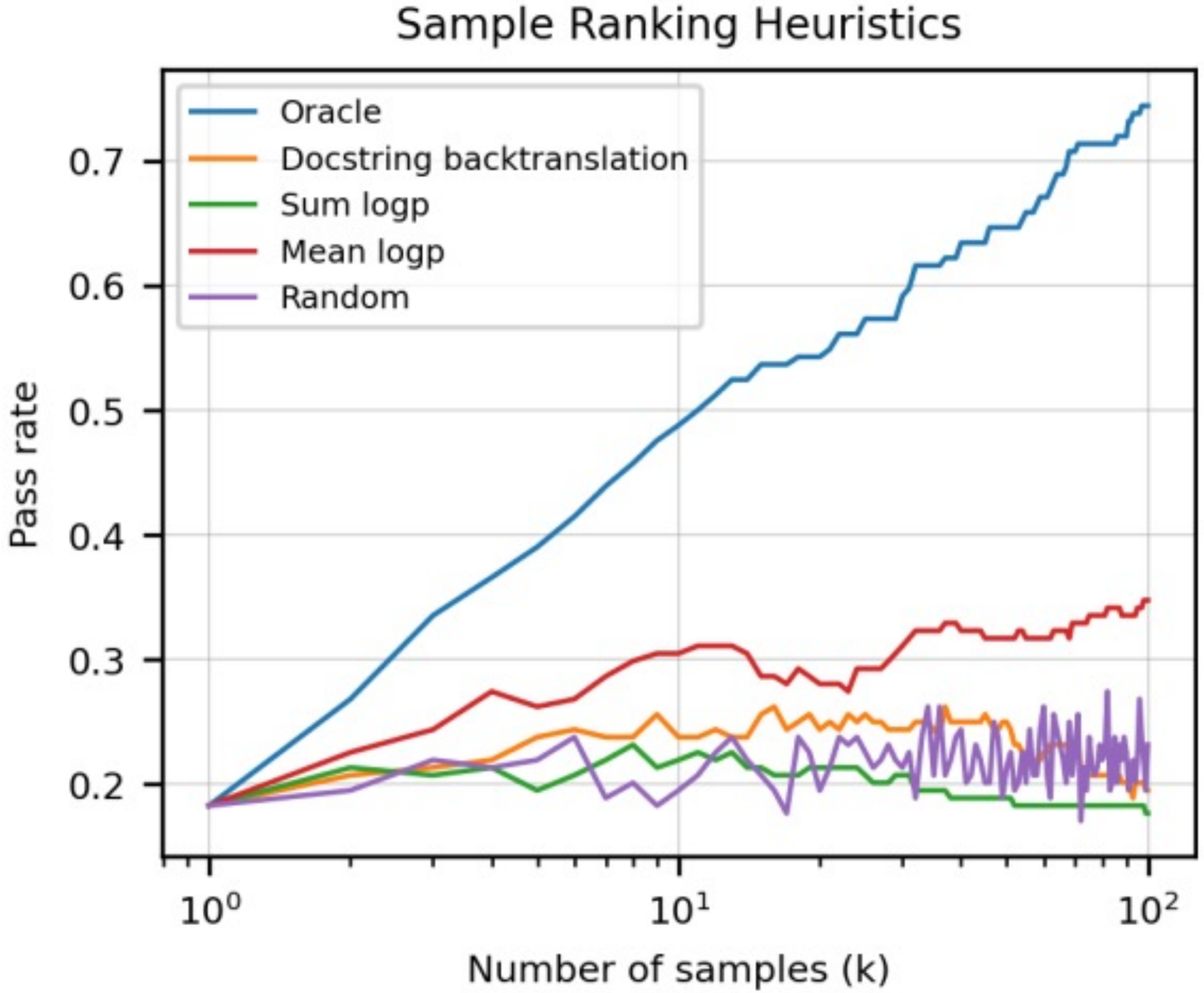


Mark Chen et al. (2021)

HumanEval

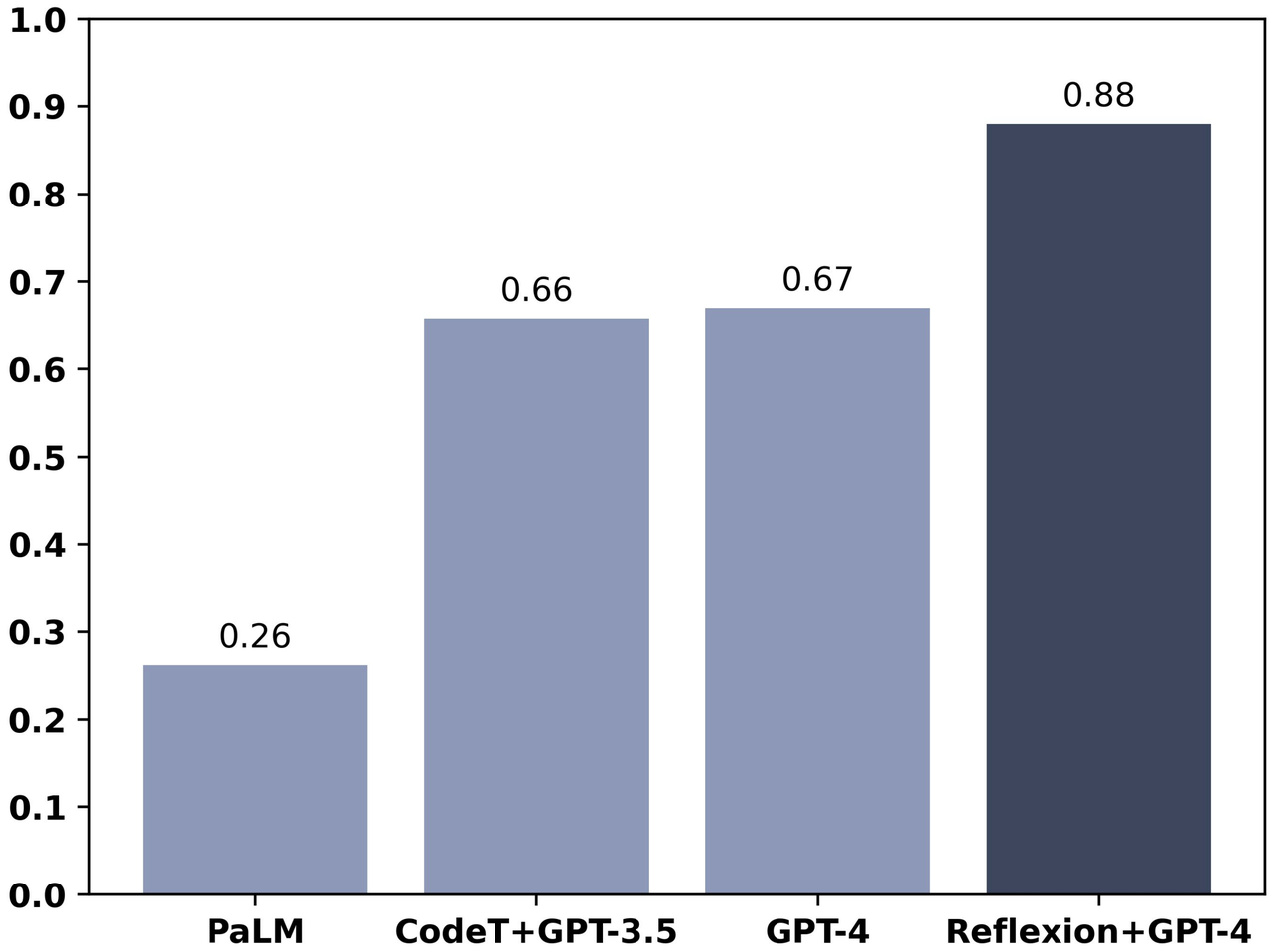
- Another setting: can we generate a bunch of samples and then pick the correct one? This would be useful for rejection sampling

- Other experiments: additional fine-tuning on competitive programming problems, docstring generation



Reflexion

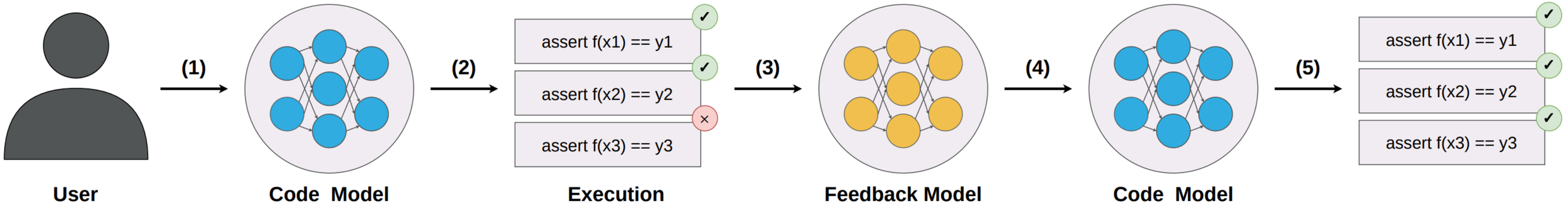
- Follow-up to ReACT (some same authors)
- Interact with environment, generate a “reflection” about that interaction, then condition on that interaction for the next round
- Very strong results on HumanEval!



https://twitter.com/johnjnay/status/1639362071807549446

Shinn et al. (2023)

Self-Repair



(1)

Given is a string s representing the day of the week today. s is one of SUN, MON, TUE, WED, THU, FRI, or SAT. After how many days is the next Sunday (tomorrow or later)?

```
# UNIT TESTS
# (EXECUTABLE)
assert f('MON') == 6
assert f('WED') == 4
assert f('SUN') == 7
```

(2)

```
def f(s):
  return (7 - ['SUN', ... , 'FRI', 'SAT'].index(s)) % 7
```

(3)

Given input 'SUN', the program returned 0, but the expected output was 7.

(4)

The code does not account for the case where the input is 'SUN' and the output should be 7. This can be fixed by removing the modulo operation.

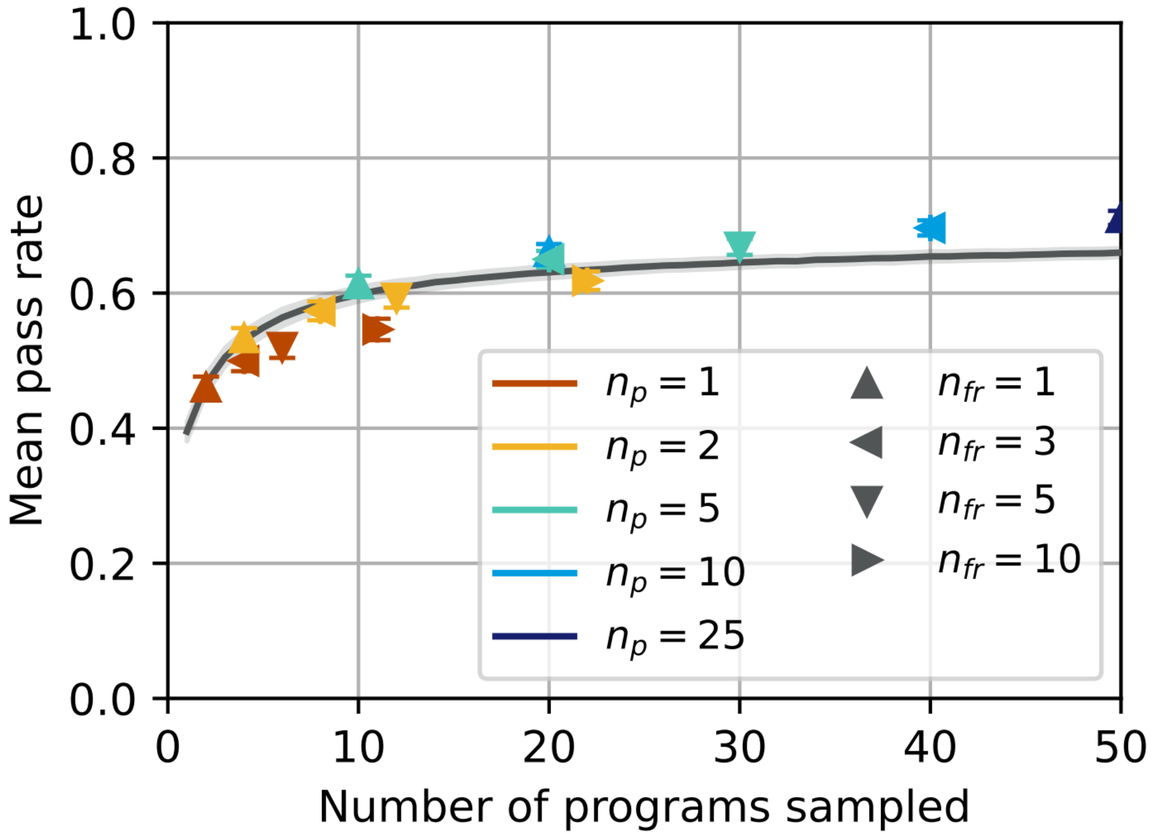
(5)

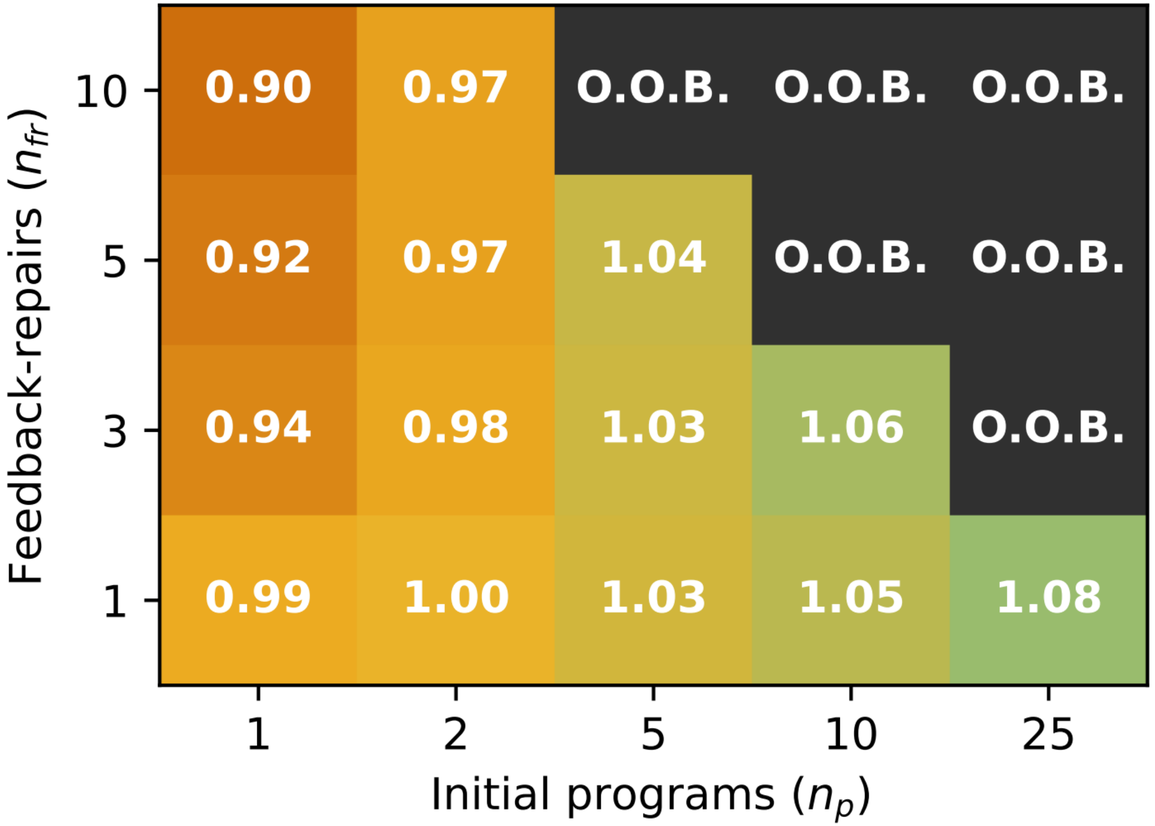
```
def f(s):
  return (7 - ['SUN', ... , 'FRI', 'SAT'].index(s)) # % 7
```

Is Self-Repair a Silver Bullet for Code Generation?

Theo Olausson et al. (2023)

Self-Repair



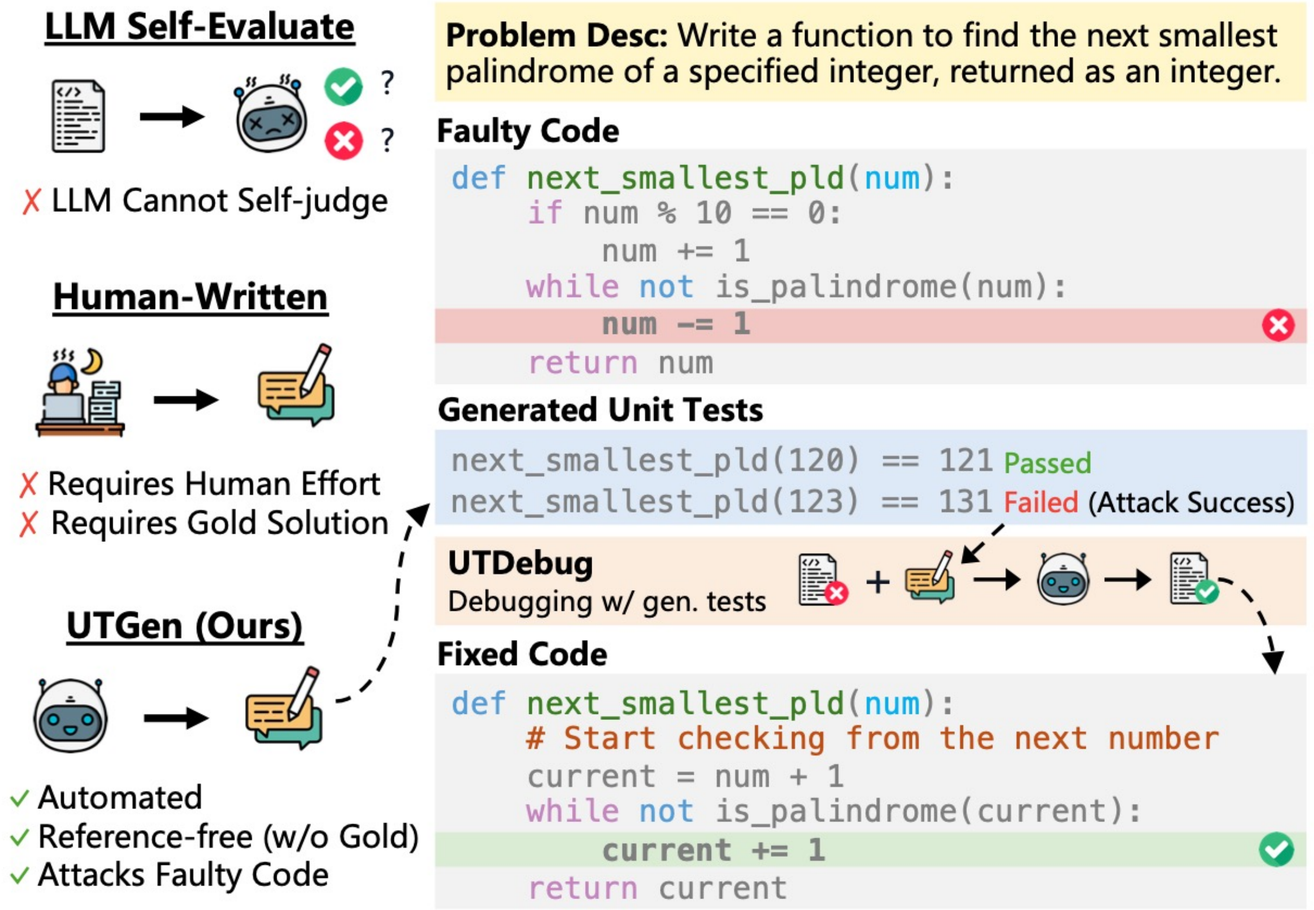


- GPT-4 results on “APPS” dataset, frequently math-y programming puzzles
- Getting many initial programs and trying to repair each one once is the best strategy

Theo Olausson et al. (2023)

Generating unit tests

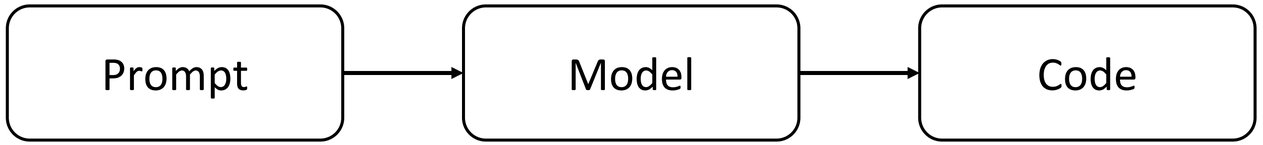
- UTGen: learn failing tests + correct outputs without gold code
- UTDebug: vote on outputs; undo edits that don't pass more tests
- Debugging: +12.35% pass@1 on MBPP+ (hard)



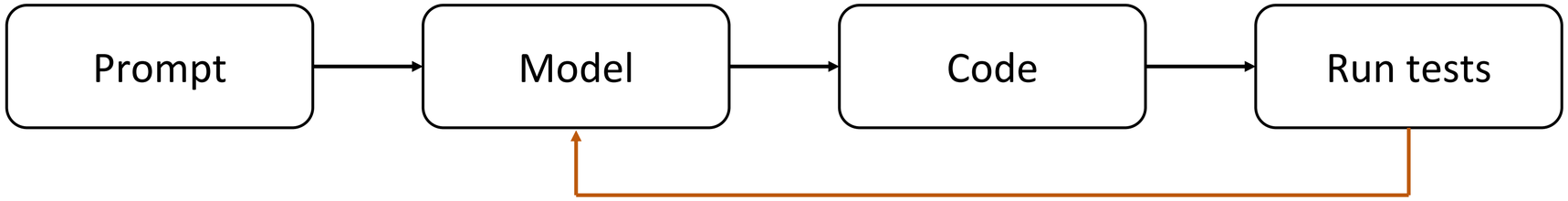
Prasad et al. (2025), Learning to Generate Unit Tests for Automated Debugging

Agent Era

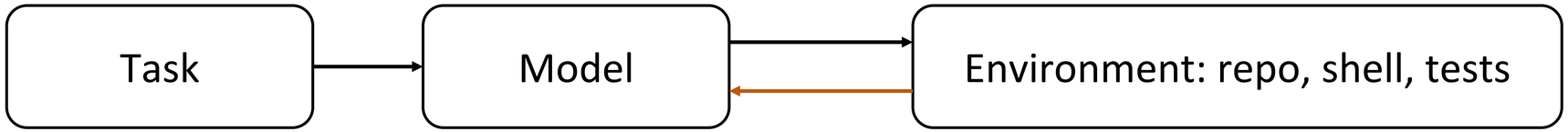
One-shot (Codex)



Self-repair loop

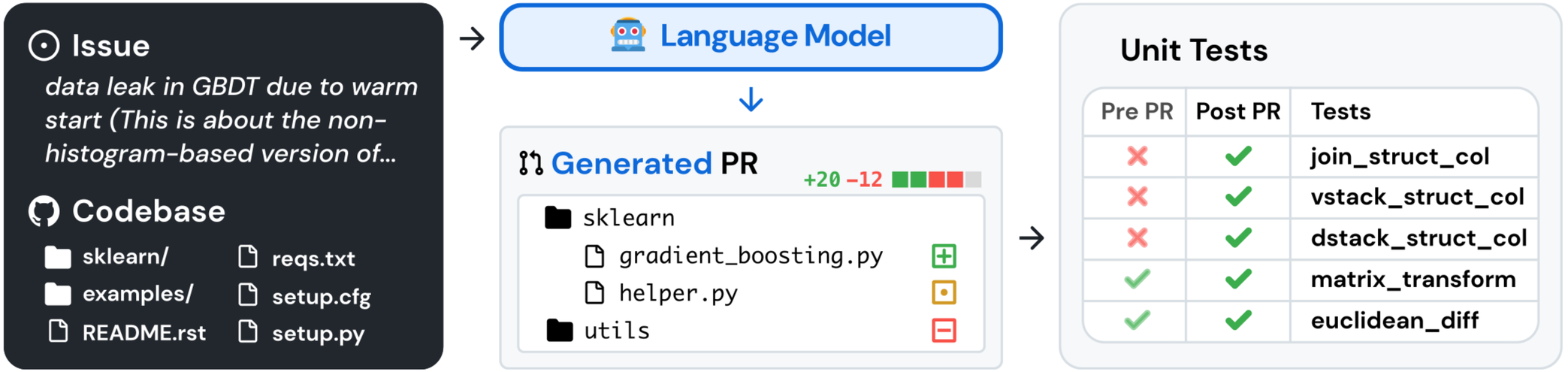


Agent loop



- Self-repair: fixed loop, written by us
- Agent: model picks each action and when to stop

SWE-Bench



- Input: GitHub issue + snapshot of the whole repo
- Output: a patch that edits the repo
- Graded by tests from the PR that fixed the issue

Jimenez et al. (2024), SWE-bench: Can Language Models Resolve Real-World GitHub Issues?

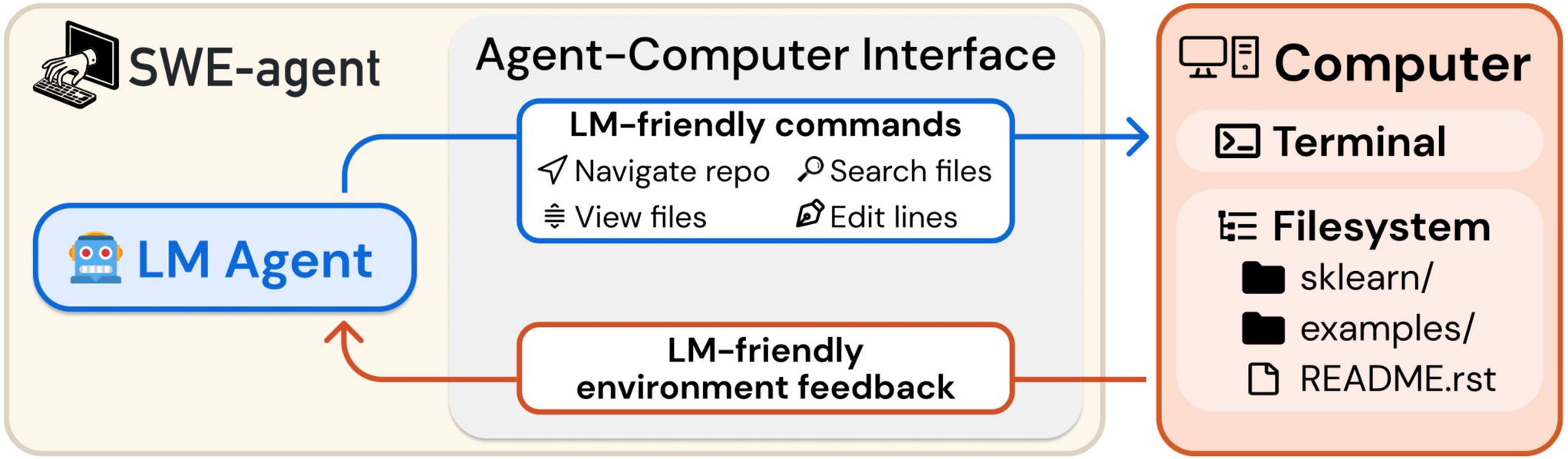
SWE-Bench

	HumanEval	SWE-bench (avg. per task)
Input	Function signature + docstring	Issue (195 words) + repo (3,010 files, 438K lines)
Output	One function	Patch: 1.7 files, 3.0 functions, 32.8 lines
Grading	Unit tests	Repo tests: fail-to-pass + pass-to-pass

- Best model at release (Claude 2 + retrieval): 1.96% resolved

Chen et al. (2021), Evaluating Large Language Models Trained on Code
Jimenez et al. (2024), SWE-bench: Can Language Models Resolve Real-World GitHub Issues?

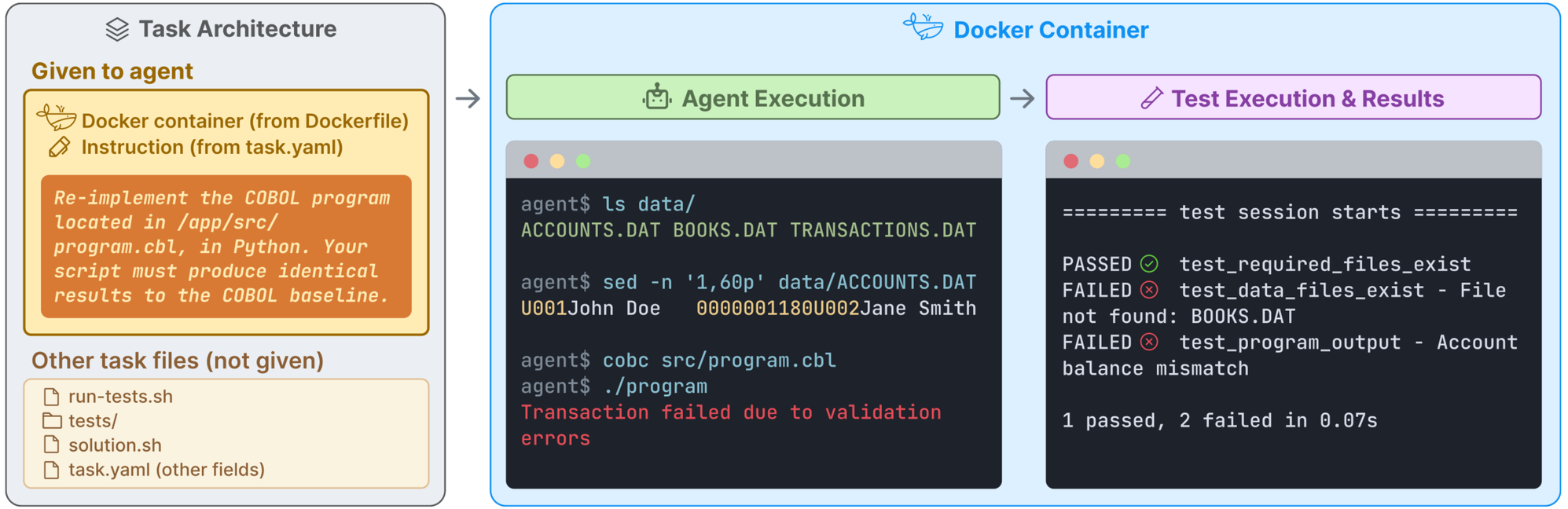
SWE-Bench



- SWE-agent: tools built for the model to view, search, edit, run
- Full SWE-bench: 3.8% (best retrieval) → 12.5% (GPT-4 Turbo agent)
- SWE-bench Verified: 500 tasks checked by human engineers

Yang et al. (2024), SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering
OpenAI (2024), Introducing SWE-bench Verified

Terminal agents

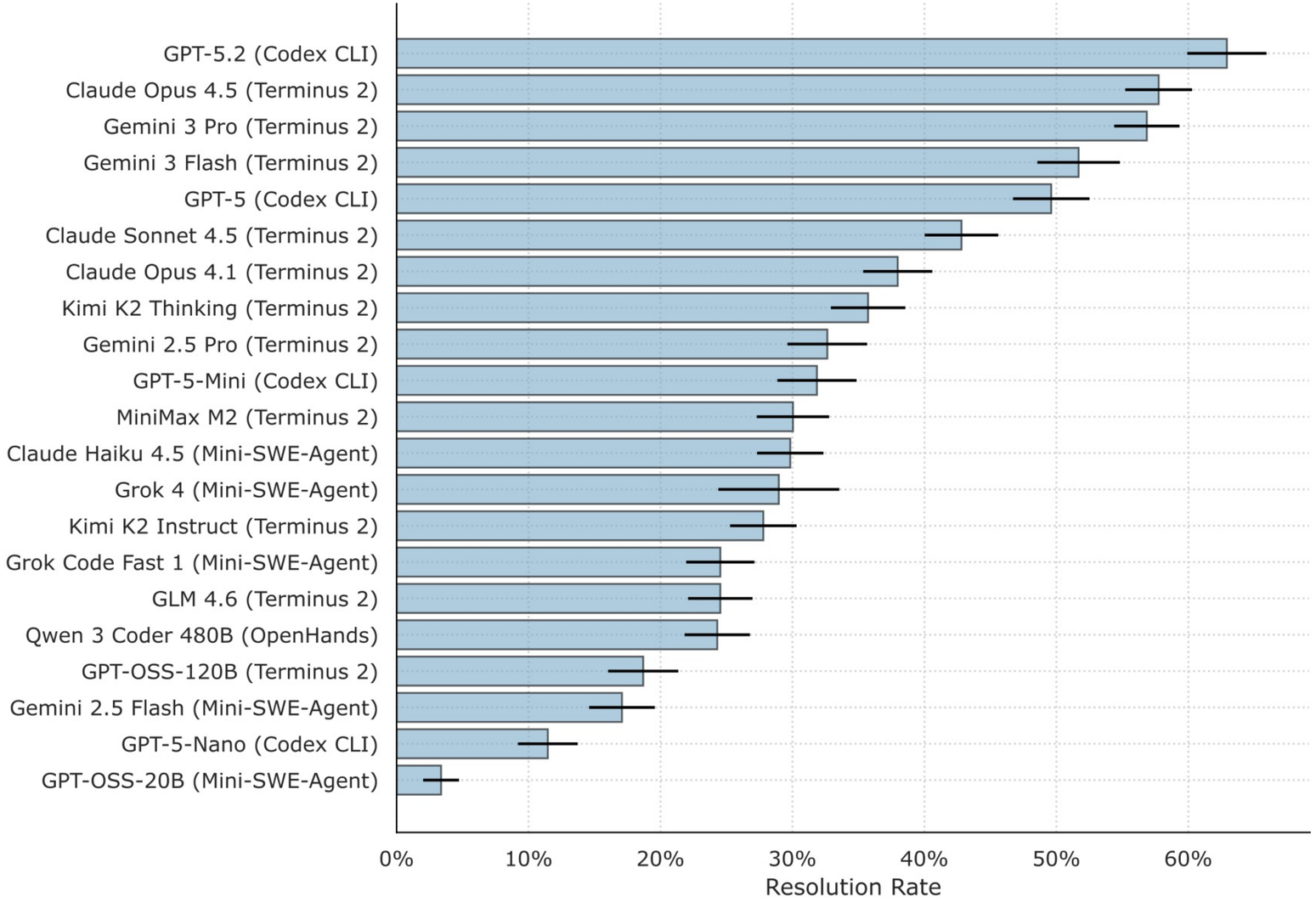


- Task: instruction + Docker container + time limit
- Tests check the final container state, not the commands
- 89 tasks: train models, build Linux, reverse-engineer binaries

Merrill et al. (2026), Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces

Terminal agents

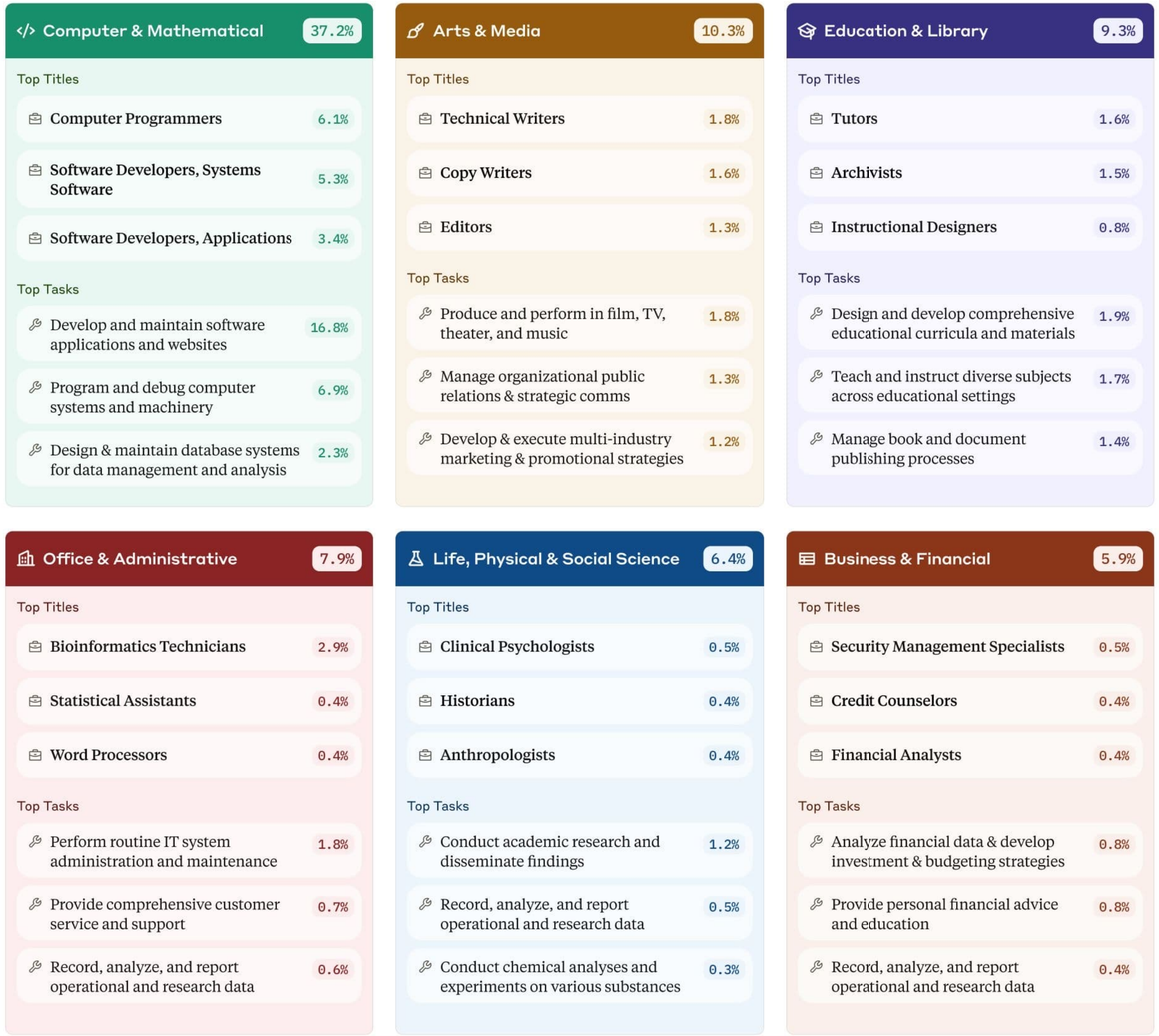
- Best: 63% (Codex CLI + GPT-5.2); best open-weight: 36%
- Top command failure: tool not installed or not on PATH
- Some runs: 2 hours and ~100M tokens on one task



Merrill et al. (2026), Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces

Code agent progress

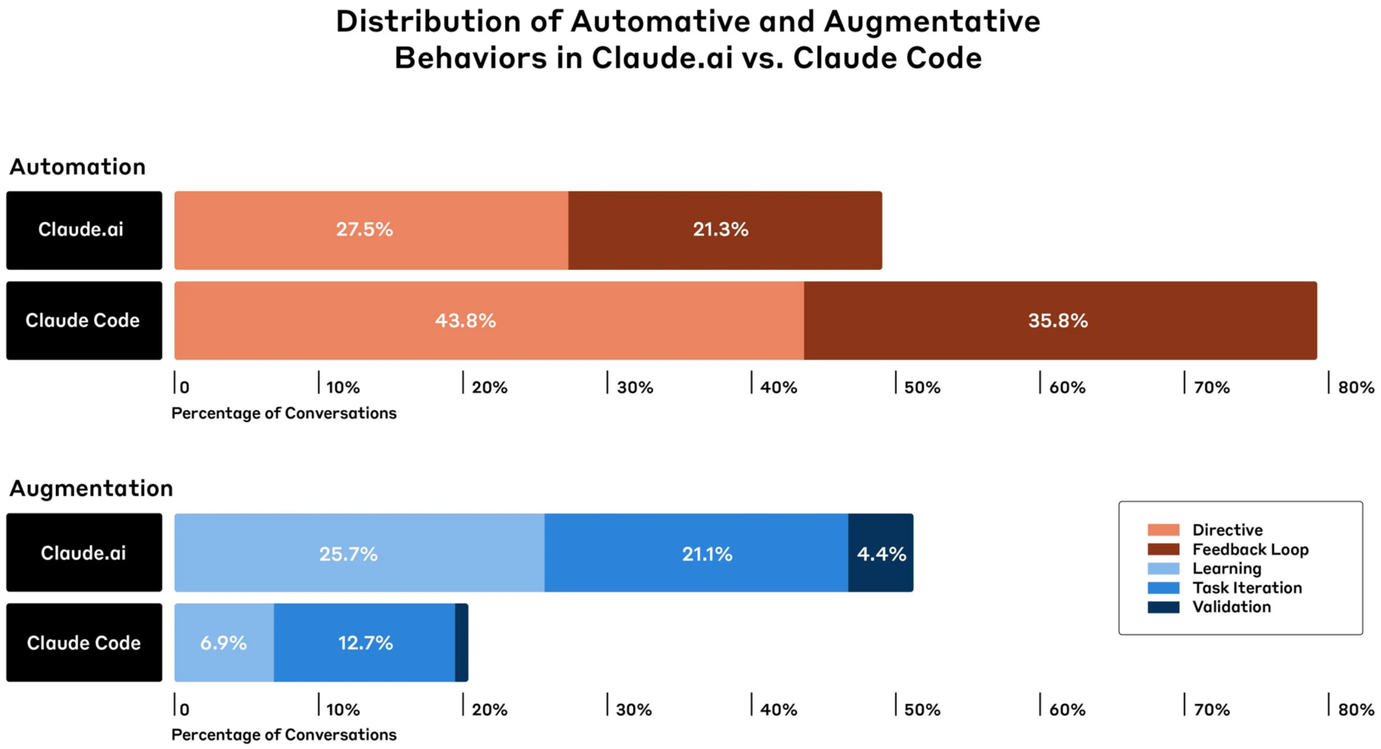
- 37% of Claude.ai conversations: computer & math tasks
- This was BEFORE CC



Anthropic (2025), The Anthropic Economic Index
Anthropic (2025), Anthropic Economic Index: AI's Impact on Software Development
Becker et al. (2025), Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity

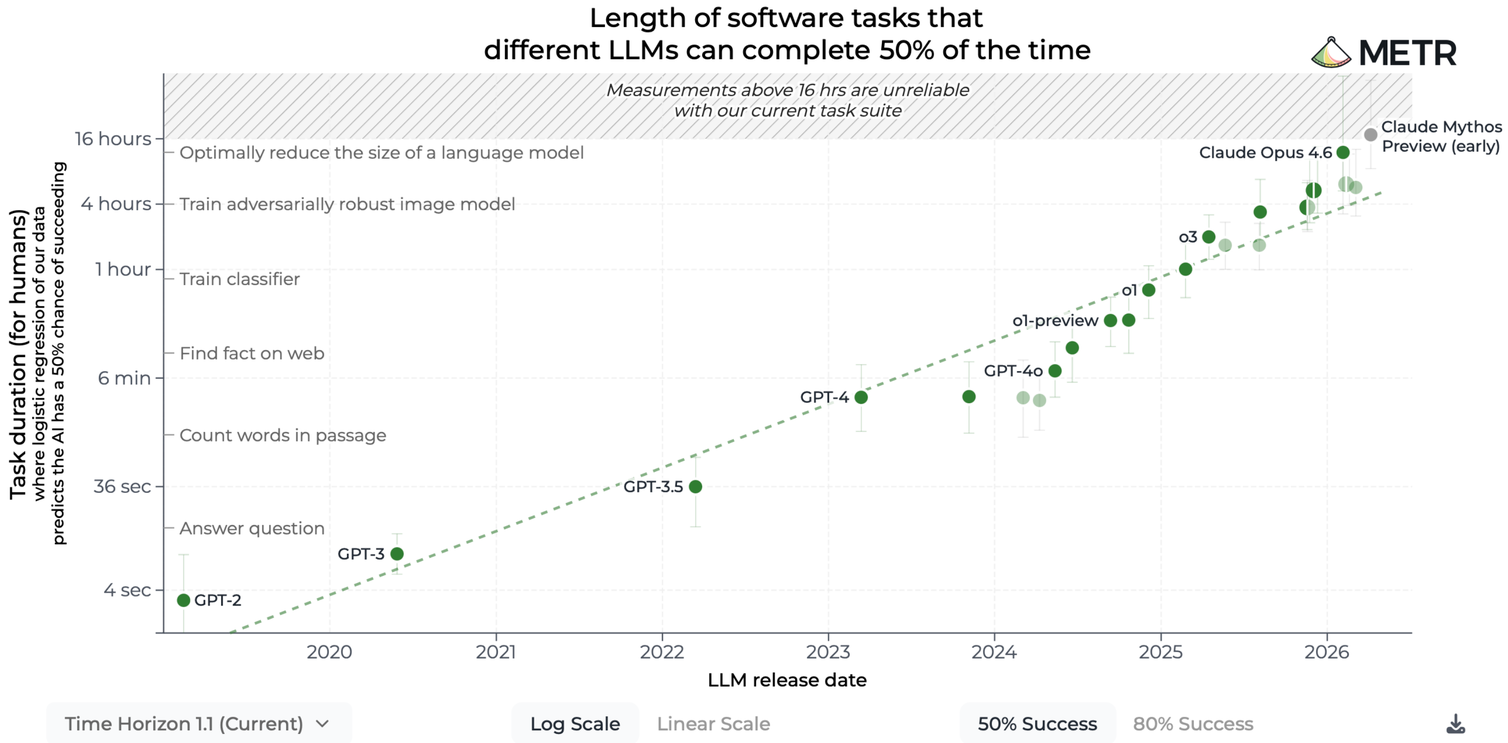
Augment vs. Automate

- Claude Code: 79% automation vs. 49% on Claude.ai



Anthropic (2025), The Anthropic Economic Index
Anthropic (2025), Anthropic Economic Index: AI's Impact on Software Development
Becker et al. (2025), Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity

Code agent progress

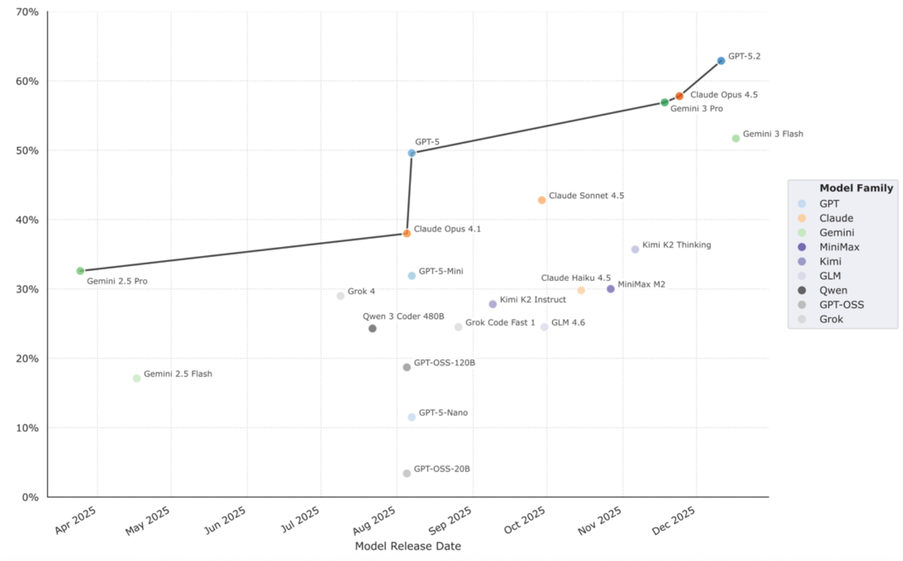


- Task length agents finish at 50% success: doubles every ~7 months
- Measured in human expert time; o3 (2025): ~110 minutes

Kwa et al. (2025), Measuring AI Ability to Complete Long Tasks

Code agent progress

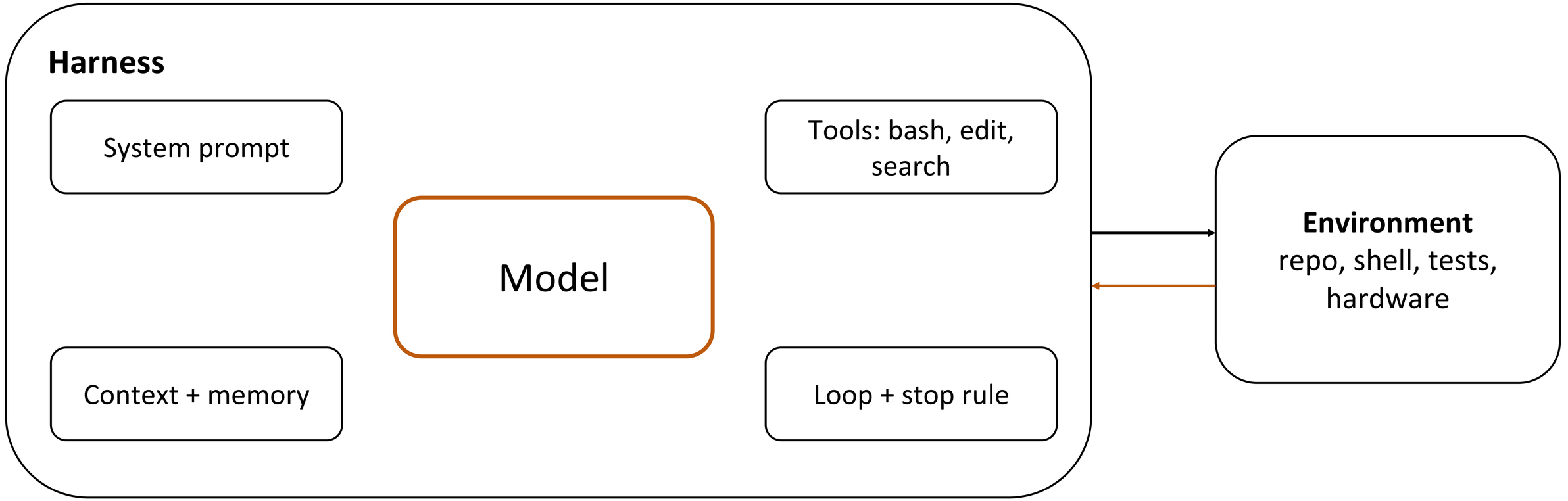
- SWE-bench Verified: top scores ~97%, close to saturated
- Terminal-Bench 2.0: best score nearly doubled in 8 months



	Systems (88)	Harness	Score	Cost / Test	Tokens / Test	Duration / Test
1	Claude Opus 5	Mini-SWE-agent	97.00 % ±0.76	$ 1.29	1.09 m	9 m 37 s
2	DeepSeek V4 Pro 0813 $	Mini-SWE-agent	96.40 % ±0.83	$ 0.10	764.04 k	4 m 00 s
3	GPT-5.6 Sol	Mini-SWE-agent	96.20 % ±0.86	$ 1.15	512.39 k	3 m 02 s
4	Grok 4.6	Mini-SWE-agent	95.60 % ±0.92	$ 0.78	968.97 k	10 m 04 s
5	GPT-5.6 Terra	Mini-SWE-agent	95.40 % ±0.94	$ 0.40	618.88 k	3 m 00 s
6	GLM 5.3	Mini-SWE-agent	95.40 % ±0.94	$ 0.34	958.22 k	14 m 01 s
7	Claude Fable 5	Mini-SWE-agent	95.00 % ±0.98	$ 2.05	722.57 k	5 m 56 s
8	Kimi K3	Mini-SWE-agent	93.40 % ±1.11	$ 0.76	883.95 k	10 m 19 s
9	GPT-5.6 Luna	Mini-SWE-agent	93.00 % ±1.14	$ 0.04	780.98 k	3 m 21 s
10	GLM 5.3 Flash	Mini-SWE-agent	92.00 % ±1.21	$ 0.02	792.80 k	1 h 04 m

Epoch AI (2026), SWE-bench Verified; Vals AI (2026), SWE-bench Verified Leaderboard
Merrill et al. (2026), Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces

Model vs. harness



- Model: weights; tokens in, tokens out
- Harness: everything else (Claude Code, Codex CLI, SWE-agent)

Anthropic (2024), Building Effective Agents
Anthropic (2025), Effective Harnesses for Long-Running Agents

Model vs. harness

- Same model, better interface: 11% → 18% on SWE-bench Lite
- Same model, other harness: +17% (Gemini 2.5 Pro, Terminal-Bench)
- Same harness, stronger model: +52% (Codex CLI, GPT-5.2 vs. Nano)
- Leaderboards fix one minimal harness to compare models

Model Name	Agent Name	Resolution Rate	Input Tokens	Output Tokens
GPT-5.2	Codex CLI	62.9% ± 3.0%	137.5M	2.3M
Claude Opus 4.5	Terminus 2	57.8% ± 2.5%	3.9M	1.3M
Gemini 3 Pro	Terminus 2	56.9% ± 2.5%	5.1M	2.2M
GPT-5.2	Terminus 2	54.0% ± 2.9%	12.4M	2.6M
Claude Opus 4.5	Claude Code	52.1% ± 2.5%	256.9M	0.8M
Claude Opus 4.5	OpenHands	51.9% ± 2.9%	151.4M	1.4M

Yang et al. (2024), SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering
Merrill et al. (2026), Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces

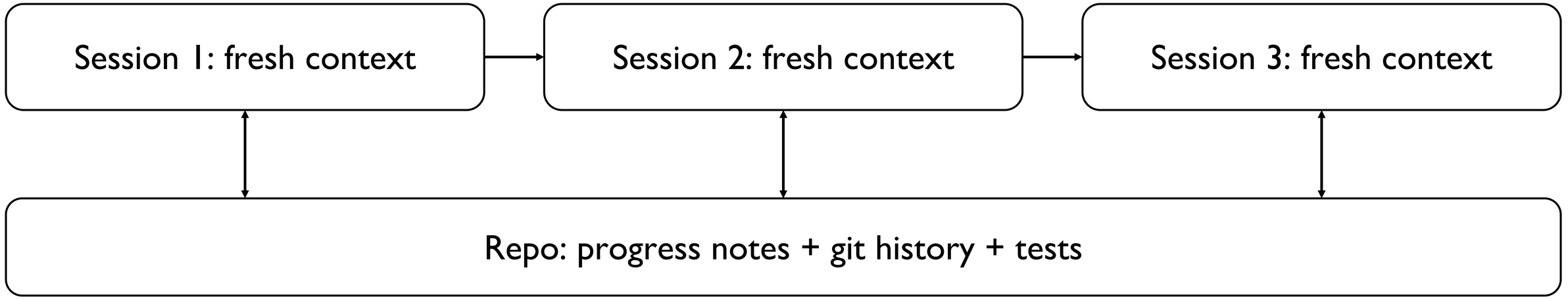
Post-training for code

$$\max_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)], \qquad R(\tau) = \mathbb{1}[\text{tests pass}]$$

- Reward is checked by code, not by people
- SWE-RL: reward = match to human patch; 70B → 41% Verified
- DeepSWE: RL inside the agent loop; 23% → 42% in 200 steps
- Bottleneck is environments: Qwen3-Coder runs 20K in parallel
- CPU is often the bottleneck here (loading docker, loading environment, executing)
- Models learn to game the checker (edit tests, patch the grader)

Wei et al. (2025), SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution
Agentica & Together AI (2025), DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL
Qwen Team (2025), Qwen3-Coder: Agentic Coding in the World
Von Arx et al. (2025), Recent Frontier Models Are Reward Hacking

Harness engineering



- Humans design the environment; agents do optimization
- OpenAI: ~1M lines, none written by hand, in 5 months
- Context tools: compaction, note files, sub-agents
- Skill files, AGENTS.md, custom tools, tests

Lopopolo (2026), Harness Engineering: Leveraging Codex in an Agent-First World
Anthropic (2025), Effective Context Engineering for AI Agents
Anthropic (2025), Effective Harnesses for Long-Running Agents



Takeaways

- Language was being interpreted into logical forms that looked like code for a long time (including in formal semantics)
- Rather than doing this with parsers, now we just use seq2seq models
  - Powerful enough models will almost always generate code that compiles. You don't need special constraints on the output.
- …and because of pre-training, rather than using customized DSLs, we just use source code because models have seen more of it
- Coding agents work remarkably well and are changing the way we code