

04834580 Software Engineering (Honor Track) 2025-26

Coding Agents

Sergey Mechtaev

mechtaev@pku.edu.cn

School of Computer Science, Peking University



A bare LLM does *one* thing: read a prompt, emit text, stop. It cannot run code, read a file it was not given, or check whether it was right.



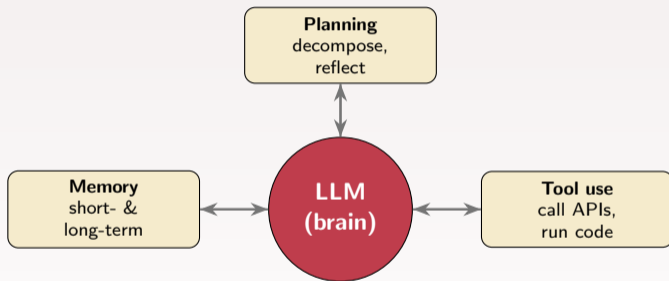
An **agent** wraps the model in a *loop* with **tools** and **feedback**. Now it can edit a repo, run the tests, read the failure, and try again — the engine behind today's SWE-bench leaders.

Definition (LLM-powered agent)

An **agent** is an autonomous system that uses an LLM as its reasoning core — its “brain” — to choose and execute a *sequence* of actions: invoking tools, observing the results, and revising its plan in pursuit of a goal.

*The model decides **what to do next**; the loop and the tools let it actually **do** it — turning a text predictor into something that acts on the world. [1]*

Lilian Weng's influential synthesis: **Agent = LLM + Planning + Memory + Tool use.**



- ▶ **Planning** — break a big goal into steps; learn from outcomes.
- ▶ **Memory** — recall context beyond the current window.
- ▶ **Tool use** — reach outside the model to the real world.

Real tasks are too big for one forward pass. Planning splits into:

Task decomposition

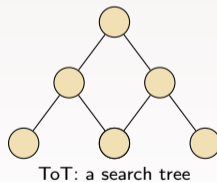
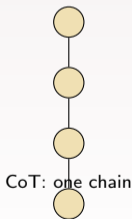
split a goal into a sequence of manageable sub-goals

Self-reflection

criticise past steps and refine the approach

Decomposition decides *how to break the work down*; reflection decides *how to recover when a step goes wrong*. Both can be *prompting patterns* over a frozen LLM.

- ▶ **Chain of Thought** [2] — “think step by step”: a single linear chain of sub-steps.
- ▶ **Tree of Thoughts** [3] — explore *several* branches per step; search the tree (BFS/DFS) and evaluate states.
- ▶ **LLM+P** [4] — hand the planning off to a *classical planner*: translate the problem to PDDL, solve, then translate the plan back.



Instead of planning everything up front, alternate **Thought** (reason in language) and **Action** (call a tool), reading each **Observation** before the next thought.

A ReAct trajectory

Thought: the test fails on empty input — I should look at the guard clause.

Action: `grep -n "def parse" src/reader.py`

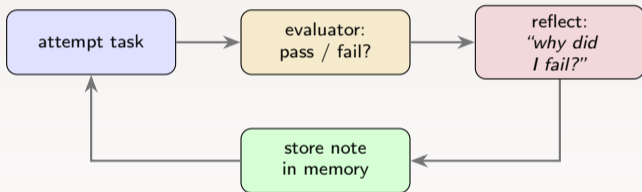
Observation: line 42: `def parse(s): return s.split(",")`

Thought: no check for the empty string; add one and re-run the test.

Action: `edit ...` → **Observation:** 1 passed

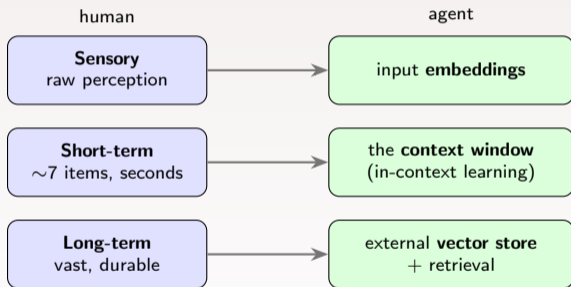
Reasoning makes the *actions* better-targeted; observations keep the *reasoning* grounded in reality instead of hallucination.

When an attempt fails, don't just retry blindly — have the agent write a **verbal self-critique** and store it, so the next attempt is informed.



This is *verbal* reinforcement learning: no weight updates, just feedback turned into text the model reads on the retry. The natural fit for coding — the “evaluator” is simply **running the tests**.

Human memory inspires the agent design — each type maps onto a concrete mechanism.



The context window is fast but *small and forgotten* after the session; the vector store is the agent's **persistent** long-term memory.

To recall from a large store cheaply, embed everything as vectors and fetch the nearest ones to the query — **Maximum Inner Product Search**.



Exact search is too slow at scale, so we use **approximate nearest neighbour** indexes (HNSW, FAISS, ScaNN, ...): a little recall is traded for a huge speed-up.

For coding agents this becomes: index the repo, retrieve the functions, tests, and docs relevant to the issue.

Weights are frozen at the training cutoff and cannot compute, browse, or execute. **Tools** fill the gaps — and the LLM learns to decide *when* and *how* to call them.

- ▶ **Toolformer** [7] — the model *teaches itself* to insert API calls (calculator, search, ...) where they would improve the next-token prediction.
- ▶ **MRKL** [8] — a router sends each query to the right expert module, neural or symbolic.
- ▶ **HuggingGPT** [9] — the LLM is a *planner* that orchestrates other models as tools.

A coding agent's tools are the developer's tools: shell, editor, compiler, test runner, grep, git, debugger.

The same three components are also where things break.

- ▶ **Finite context** — the window caps how much history, code, and retrieved knowledge fit at once. Long tasks overflow it.
- ▶ **Long-horizon planning** — models still struggle to adjust a plan after an unexpected error; they lack robust trial-and-error over many steps.
- ▶ **Unreliable natural-language interface** — the agent's decisions are *text*; it may misformat a tool call or ignore an instruction, so scaffolds need heavy parsing and error handling.

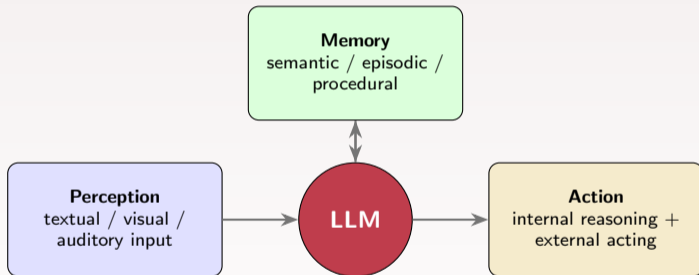
These are exactly the problems the software-engineering setting helps tame — next.

Of all agent domains, SE is uniquely well-suited — it removes the two hardest problems above.

- ▶ **The tools already exist** and are scriptable: compiler, test suite, git, linters, debuggers — decades of automation.
- ▶ **Feedback is verifiable.** Tests give an *objective* pass/fail signal — no fuzzy human judgement needed for the reward.
- ▶ **Tasks are well-scoped.** “Make these tests pass in this repo” is concrete, bounded, and checkable.

*Verifiable feedback closes the ReAct/Reflexion loop with ground truth — the model can be wrong and **find out**.*

Wang et al. (2025) survey **115 papers** on LLM-based agents in SE and organise them into one framework — mirroring the general agent anatomy, specialised for code.



Perception *reads* the inputs, Memory *supplies* context, Action *reasons and acts on the environment*.

Code is not just text — it has structure. The survey distinguishes input modalities:

- ▶ **Textual** — by far the most common:
 - ▶ *Token-based*: treat code as a plain token sequence (mainstream).
 - ▶ *Tree/graph-based*: ASTs, control-flow graphs — captures structure.
 - ▶ *Hybrid*: combine tokens with structure.
- ▶ **Visual** — UI sketches, UML diagrams as image input.
- ▶ **Auditory** — e.g. programming videos / speech.

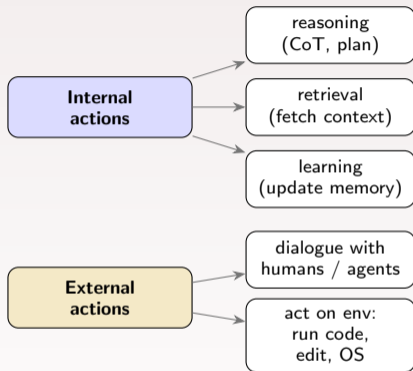
Trend: SE agents are overwhelmingly token-based today; structural, visual, and auditory perception are wide open.

Semantic — acknowledged world knowledge in an external *knowledge base*: project docs, libraries, API references, Stack Overflow. Retrieved to ground generation.

Episodic — experience from the *current/past cases*: similar code snippets, in-context examples, the history of previous reasoning iterations.

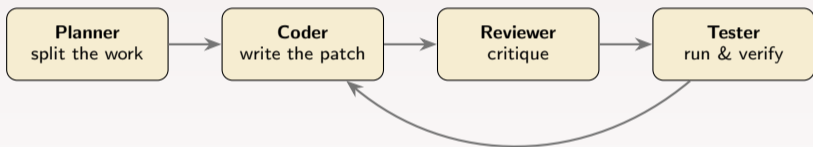
Procedural — *how to act*: *implicit* knowledge baked into the LLM weights, and *explicit* knowledge written into the agent's own code / prompts.

Mainstream technique today: retrieval (RAG) for semantic & episodic memory; procedural memory is the agent's design itself.



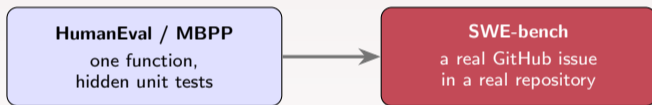
Internal actions refine the decision (think, recall, learn); **external** actions touch the world — talk to a human, or drive a *code interpreter / shell* and read the feedback. The external loop is what makes it an *agent*, not a chatbot.

A hard task can be split across *several* specialised agents that talk to one another — mirroring a software team.

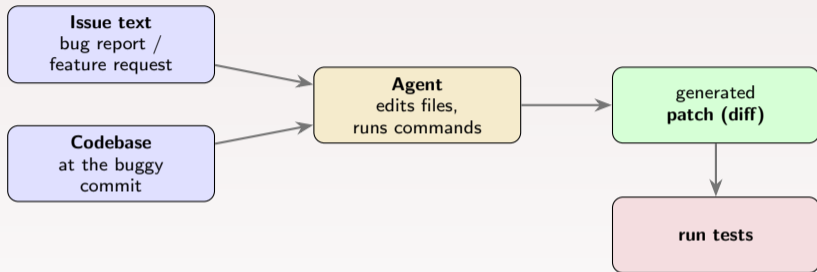


Roles bring division of labour and cross-checking — but at a cost: more LLM calls, communication overhead, and synchronisation. *When does a team beat one strong agent?* remains open.

Earlier benchmarks (HumanEval, MBPP) ask for one short, self-contained function. Real engineering is messier — so the bar moved.



SWE-bench [11]: thousands of real (issue, pull-request) pairs mined from popular Python projects. The task: *produce a patch that makes the project's own tests pass.*



Grading uses two test sets from the real merged PR:

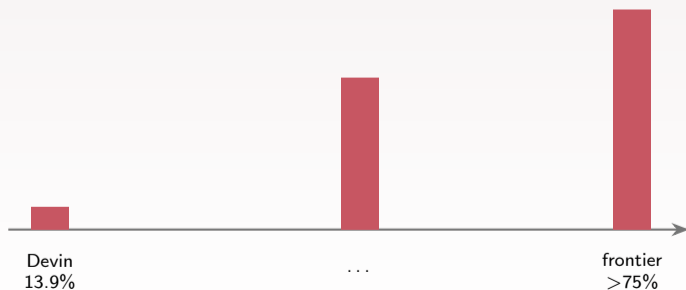
- ▶ FAIL_TO_PASS — were failing, must now pass (the fix works).
- ▶ PASS_TO_PASS — were passing, must stay passing (nothing broke).

Resolved *only if both* hold — a strict, automatic, ground-truth check.

One idea, several curated variants:

- ▶ **Full / Lite** — the original $\sim 2.3k$ tasks; Lite is a 300-task subset.
- ▶ **Verified** [12] — 500 tasks human-checked to be fair and solvable.
- ▶ **Multilingual** — beyond Python.
- ▶ **Pro** [13] — 731 harder, enterprise-scale tasks across many repos & languages.

SWE-bench Verified resolve rate, over ~ 2 years



The same LLM scores very differently depending on the *scaffold* wrapped around it — how it perceives and acts on the repo.

- ▶ **SWE-agent** [14] — an *agent-computer interface*: custom commands to view, edit, and navigate code, designed for the LLM.
- ▶ **OpenHands** [15] — a general open platform for software-developer agents.
- ▶ **Agentless** [16] — skip the autonomy: a *fixed* localize → repair → validate workflow, often competitive.
- ▶ **mini-SWE-agent** — a radically minimal scaffold: ~100 lines, *bash only*, no special tools.

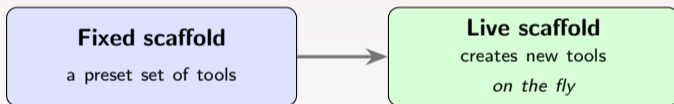
Hand-designing the “right” scaffold is itself a hard, open-ended search problem.

Today's agents have a **fixed** design — the tools and workflow are preset before the task is even seen.

- ▶ Manually designing an *optimal* scaffold means searching an essentially infinite space.
- ▶ **Self-improving** agents (Darwin-Gödel Machine [17], SICA) *evolve* their own code — but via costly **offline** training: one DGM run \approx **\$22,000** / >1000 GPU-hours.
- ▶ Worse, offline evolution **overfits** to the benchmark and the specific LLM — the learned agent may not generalise.

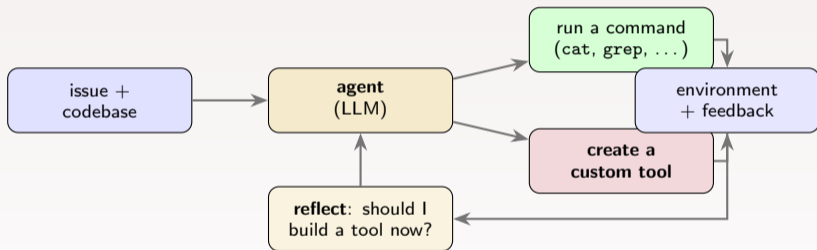
*Can an agent improve itself **while** it works — with no offline training at all?*

A software agent is just ... software. And modern LLMs are already good at writing software. So let the agent **rewrite its own capabilities at runtime**.



LIVE-SWE-AGENT elevates **tool creation** from a one-time preprocessing step to a *first-class decision*, taken repeatedly *during* problem solving — “live” self-evolution.

Start from the *minimal* mini-SWE-agent (bash only). At each step the agent chooses one of:



A lightweight **reflection** prompt after every observation asks whether a new tool would help. No change to the agent loop, no offline training, *agnostic to the LLM*.

Tools are just scripts the agent writes and then calls. They range from *general* utilities to *issue-specific* analyzers.

A self-made editing tool (sketch)

```
def edit_file(filename, operations): # replace / insert / delete
    ...
    if old_text in lines:
        lines = lines.replace(old_text, new_text)
        print("Successfully edited", filename) # <-- explicit feedback
    else:
        print("Warning:", old_text, "not found") # bash `sed` stays silent!
```

Examples observed: a **focused search** tool (skips `.git`, caps results), an **edit** tool that *reports* success/failure, even a MARC-file or Go analyzer for one specific repo. Tools are saved and **reused**.

Single attempt, *no* test-time scaling.

Benchmark	Best LLM	mini-SWE	Live-SWE-agent
SWE-bench Verified	Gemini 3 Pro	74.2%	77.4%
SWE-bench Verified	Claude 4.5 Sonnet	70.6%	75.4%
SWE-Bench Pro	Claude 4.5 Sonnet	43.6% [†]	45.8%

[†] best prior result on Pro is the hand-built SWE-agent.

77.4% on Verified and **45.8%** on Pro — topping all existing open *and* commercial agents at the time of writing, while adding only cents of cost over the bare scaffold.

Recall the first agent challenge — **finite context**. In SE it bites hard: to be safe, retrieval *over-approximates*, returning hundreds of lines (top- k files/functions) to capture a handful of relevant ones.



retrieved context — ~8% relevant vs. mostly noise

Two compounding costs:

- ▶ **\$ cost** — inference scales with context length.
- ▶ **Worse answers** — noise causes *lost-in-the-middle* [19] and *distraction* [20], diluting the fix signal.

Idea: distill the retrieved context to a concise form, then feed *that* to the resolver. But naive compression backfires:

- ▶ **Generic** compressors (e.g. LLMingua) treat code as prose — they break structure and snap definition–use links.
- ▶ **Code-specific** pruners (LongCodeZip, SWE-Pruner) keep what *looks similar* to the bug — and discard **fix ingredients** that don't.

Fix ingredient

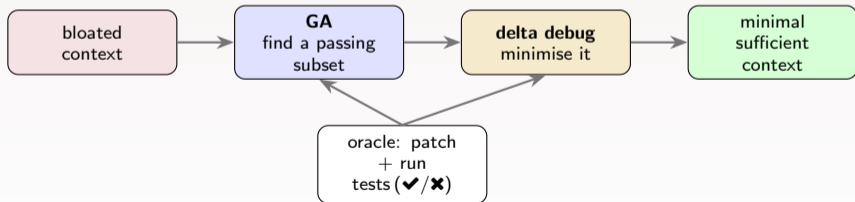
A code element (variable, expression, type, project API) needed to *construct the patch* — even if it bears no lexical resemblance to the bug itself.

*The reframe [21]: compress for **sufficiency**, not for relevance.*

Definition (Minimal sufficient context)

A subset \hat{C} of the retrieved context is **sufficient** if the LLM can produce a *test-passing* patch from it, and **minimal** if removing *any single* element makes it insufficient.

Found by **Oracle-Guided Code Distillation** — search where the “oracle” is *generate a patch and run the tests*:



This is expensive — but it runs *offline*, to manufacture training data.

Distil thousands of (bloated \rightarrow minimal-sufficient) pairs, then fine-tune a *small* cross-encoder (Qwen3-Reranker-0.6B + LoRA) to **score each code segment** keep/drop. At inference: no oracle, no tests — just score + budget-aware greedy selection.

Resolver LLM	no comp.	SWEzze	tokens
DeepSeek-V3.2	47.8%	52.2%	-56%
Qwen3-Coder-Next	40.0%	42.0%	-52%
GPT-5.2	53.2%	57.8%	-71%

On SWE-bench Verified (Agentless): a stable $\sim 6\times$ compression that cuts total tokens by **52–71%** and lifts resolution by **5.0–9.2%**.

*Less context, but the **right** context \Rightarrow cheaper **and** more accurate.*

- [1] Lilian Weng.
LLM powered autonomous agents.
<https://lilianweng.github.io/posts/2023-06-23-agent/>, 2023.
Lil'Log.
- [2] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al.
Chain-of-thought prompting elicits reasoning in large language models.
Advances in neural information processing systems, 35:24824–24837, 2022.
- [3] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan.
Tree of thoughts: Deliberate problem solving with large language models.
In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

- [4] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone.
LLM+P: Empowering large language models with optimal planning proficiency.
arXiv preprint arXiv:2304.11477, 2023.
- [5] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.
ReAct: Synergizing reasoning and acting in language models.
In International Conference on Learning Representations (ICLR), 2023.
- [6] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao.
Reflexion: Language agents with verbal reinforcement learning.
In Advances in Neural Information Processing Systems (NeurIPS), 2023.

- [7] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom.
Toolformer: Language models can teach themselves to use tools.
In Advances in Neural Information Processing Systems (NeurIPS), 2023.

- [8] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, et al.
MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning.
arXiv preprint arXiv:2205.00445, 2022.

- [9] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang.
HuggingGPT: Solving ai tasks with chatgpt and its friends in hugging face.
In Advances in Neural Information Processing Systems (NeurIPS), 2023.

- [10] Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng.
Agents in software engineering: survey, landscape, and vision.
Automated Software Engineering, 32(2):70, 2025.
- [11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan.
SWE-bench: Can language models resolve real-world GitHub issues?
arXiv preprint arXiv:2310.06770, 2023.
- [12] OpenAI.
Introducing SWE-bench verified.
<https://openai.com/index/introducing-swe-bench-verified/>, 2024.
- [13] Scale AI.
SWE-Bench pro: Can ai agents solve long-horizon software engineering tasks?
arXiv preprint arXiv:2509.16941, 2025.

- [14] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press.
SWE-agent: Agent-computer interfaces enable automated software engineering.
In Advances in Neural Information Processing Systems (NeurIPS), 2024.
- [15] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, et al.
OpenHands: An open platform for ai software developers as generalist agents.
arXiv preprint arXiv:2407.16741, 2024.
- [16] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang.
Agentless: Demystifying llm-based software engineering agents.
arXiv preprint arXiv:2407.01489, 2024.
- [17] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune.
Darwin Gödel machine: Open-ended evolution of self-improving agents.
arXiv preprint arXiv:2505.22954, 2025.

- [18] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.
- [19] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics (TACL)*, 12, 2024.
- [20] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning (ICML)*, 2023.
- [21] Haoxiang Jia, Earl T. Barr, and Sergey Mehtaev. Compressing code context for LLM-based issue resolution, 2026. Manuscript. Artifact: <https://zenodo.org/records/19248411>.