[CS 294-264] Agent Assignment

# Deadline

* **Fri, Oct 3rd, 2025, 11:59 p.m. PST**
* This is the **only homework** of the semester!

# Why an Agent Assignment?

We believe this assignment will help you tackle harder, more realistic problems more quickly, and position us at the forefront of using AI tools to accelerate systems research.

AlphaEvolve, OpenEvolve, and SATLUTION help to discover new algorithms by editing code. They edit code instead of pseudocode because real code can be executed, tested, and verified for correctness. However, the coding approaches used by such systems are pretty straightforward. They just make a single call to an LLM with a prompt to edit the code. Code modified by a single call to an LLM is restrictive:

1. The code generated may contain some minor errors that an agent can correct in a couple of iterations.
2. They can only handle small pieces of code
3. They are limited to Python

Therefore, if we need to evolve an algorithm that spans across multiple files, possibly written in languages other than Python, we need to have a better AI-based code editor. Note that system code is often written in C/C++ for efficiency. AI is not as practical for generating correct and safe code in C/C++ as it is in Python.

The assignment will ask you to write an effective and powerful agent that can address the above limitations of generating and editing code. This, in turn, will enable you to evolve big systems projects that are currently beyond the reach of OpenEvolve. The assignment will allow you to write an agent that can incorporate various advanced search techniques, such as parallel search, reflective prompt optimization, etc. Hopefully, hands-on experience with these search techniques will allow you to rapidly evolve OpenEvolve.

In the project, you will write a software engineering agent (SWE agent) that enables you to edit code in a repository given some natural language description about the edits. We decided to ask you to write an SWE agent because we have an excellent benchmark suite for SWE tasks, called SWE-Bench. The benchmark has not been saturated.

# What is a ReAct Agent?

A **ReAct agent** (short for *Reasoning + Acting*) is a type of agent that alternates between two steps:

* **Reasoning:** The agent reasons in natural language about the problem, keeping track of what it knows, what it needs to do, and why.
* **Acting:** The agent calls an external tool (like a calculator, database, or shell command) to gather more information or take an action in the real world.

This loop continues until the agent reaches a final answer. The advantage of ReAct is that the agent doesn’t need to know everything internally — it can *reason step by step* and *use tools to extend its abilities*.

# ReAct Agent Specification

## **Problem statement**

Write a **MINIMAL** ReAct agent that solves software engineering tasks from [SWE-Bench](https://www.swebench.com/).

* Implement baseline, and improve it with optimizations
* You can use any coding agent to complete the homework; however, you must own the code and understand it.

We provide the skeleton code in 👉 [[GITHUB](https://github.com/lynnliu030/cs294-264-hw-FA25)], follow the code and find **TODO(student)** on where you should implement things.

## **Message History Tree**

Your agent must:

* Keep **a message history tree** (tree dicts with role, content, timestamp, id, parent / children)
  + role: "system", "user", "assistant", “tool”, etc.
  + content: the text content of the message (e.g. can be Markdown)
  + timestamp: when the message was created
  + unique\_id: unique identifier for each message (must be a counter starting at 1)
  + children: list of pointers (unique\_id) to the child messages in the history
  + parent: pointer (unique\_id) to the parent message

The structure of this tree will be:

1. **System prompt node** 
   * The root of the history tree is unique and contains the system prompt.
   * A system prompt has three parts
     + *Main content* is fixed: “You are a Smart ReAct agent.”
     + *List of available tools (with docstrings)*
       - A tool description will contain the signature of the callable tool and its docstring describing its functionality. This should be constructed automatically.
     + *An output format description*
2. **User prompt node** 
   1. Root node has only one child node that contains the user prompt or the task description.
3. **Instruction node** 
   1. User prompt node’s only child.
   2. Represents the mandatory instructions for the agent to follow, and a list of ideas and insights the agent has learned so far.
   3. The agent can update the content of the node by providing a tool to backtrack (see later)

### NOTE

* The path from the root node (having no parent) to a leaf node (having no children) concatenated together forms the context of the agent.
* The format of a message sent to the LLM is the following:

------------------

|MESSAGE(role="user", id=97, step=9)|

Hello LLM.



* Any node in the history tree (including the instruction node) can have multiple children.
  + A child is added when the agent backtracks or continues from a node.
* Nodes cannot be deleted, but their content can be modified.
  + Example: update mandatory instructions or set content to an empty string (ignored in context).
* The agent must maintain two pointers
  + Root node (system prompt)
  + Current node (most recent message from LLM or tool)

## **Agent Implementation**

A ReAct agent must be a class that implements a special function called

run(): str

which will run the agent and return the final result as a string. If the agent encounters any unhandled exception, it will be re-raised by the function.

The class should also provide a method

add\_functions(functions: List[Callable])

which will add the functions (i.e., tools) that the agent could call to get information from the environment.

The constructor of the agent class

\_\_init\_\_(self, name: str, parser: ResponseParser)

should take a name and a response parser object. You MUST NOT use XML or JSON format for function calls. This is because the LLM must escape special XML and JSON characters, respectively, and some LLMs (such as Gemini) can make mistakes. For example, a simple textual format could be

Let me think step-by-step:

...

----FUNTION\_CALL----

name

----ARG----

name

value

.

.

.

----ARG----

.

.

.

----FUNCTION\_CALL\_END----



You should use string rfind to extract the function call. Note that an LLM response must end a function call. rfind helps you skip the reasoning text from parsing. The LLM must produce a single function call at the end of its response. You can use -—--FUNCTION\_CALL\_END–—- as the stopping token.

## **Tool Implementation**

The following tools are implemented.

import subprocess

def run\_bash\_cmd(command: str, description: str):

"""Run the command in a bash shell and return the output or throw a ValueError exception if the process returns non-zero exit code

Args;

command (str): the shell command to run

description (str): A single-line short natural language description of what the command achivees

Returns:

The output of running the shell command

"""

def finish(result: str):

"""The agent must call this function with the final result when it has solved the given task.

Ev

Args:

result (str); the result generated by the agent

Returns:

The result passed as an argument. The result is then returned by the agent's run method.

"""



You need to implement an additional tool:

def add\_instructions\_and\_backtrack(instructions: str, at\_message\_id: str):

"""The agent should call this function if it is making too many mistakes or is stuck in finding a solution,

The function changes the content of the instruction node with 'instructions' and backtracks at the node with id 'at\_message\_id'.

"""

TODO: implement this



### Optional Tool Implementations

You will notice that LLM could make repeated mistakes in invoking specific commands. In such situations, you should add custom functions that could run a sequence of commands in addition to run\_bash\_cmd.

For example, the LLM might try to use the sed command to edit a file, but you will find that it often fails and is not efficient.

You may want to write a custom function such as

def replace\_in\_file(file\_path: str, from\_line: str, to\_line: str, content: str)

which replaces the lines from\_line to to\_line (both inclusive) with the content.

Similarly, you can define

def show\_file(file\_path: str)

which will call run\_bash\_cmd with f”cat -n {file\_path}” to show the file contents with line numbers.

# Evaluation

You need to evaluate the agent on the [SWE-bench **verified**](https://www.swebench.com/SWE-bench/) **subset** and report the performance. We set up a GitHub repository for this homework. Please follow the README to run the evaluation.

Goal: improve the accuracy of the agent by creating custom functions and the most optimized user prompt containing instructions on how to solve a generic SWE issue.

* You can refer to <https://agents.md/> for examples of thousands of SWE prompts for various SWE tasks and workflows.

For evaluation:

1. Backend LLM should be **GPT 5-mini (medium reasoning)**
2. You must restrict the number of steps to 100, i.e., MAX\_STEPS=100.
3. Baseline: report accuracy without using add\_instructions\_and\_backtrack
4. Report improved accuracy with add\_instructions\_and\_backtrack and any customized tools you want to add

# Submission Instructions

You will submit three things to the submission server [here](http://vassar.millennium.berkeley.edu:8080/):

1. **Code artifact**
   1. A zip uploaded to the portal
   2. Must contain everything needed to build and run an end-to-end evaluation
   3. Do not commit secrets/keys
2. **Report (pdf)** 
   1. You should report your accuracy number, the custom tools you created and the reason behind making them, and the lessons you learned.
3. **JSON results** for leaderboard
   1. You should submit the final SWEBench harness evaluation results to our leaderboard, which should have the following structure:

{

"total\_instances": 20,

"submitted\_instances": 20,

"completed\_instances": 19,

"resolved\_instances": 9,

"unresolved\_instances": 10,

"empty\_patch\_instances": 1,

"error\_instances": 0,

"completed\_ids": ["astropy\_\_astropy-7166", ...],

"resolved\_ids": ["astropy\_\_astropy-7166", ...],

"unresolved\_ids": ["django\_\_django-10973", ...],

"schema\_version": 2

}



You might find the checklist [here](?tab=t.olqrgkmhsjzp#heading=h.kik0z2vpfbv7) helpful.

## **More Information Coming…**