ANTHROP\C

Product

Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet

Oct 30, 2024

Our latest model, the upgraded Claude 3.5 Sonnet, achieved 49% on SWE-bench Verified, a software engineering evaluation, beating the previous state-of-the-art model's 45%. This post explains the "agent" we built around the model, and is intended to help developers get the best possible performance out of Claude 3.5 Sonnet.

SWE-bench is an AI evaluation benchmark that assesses a model's ability to complete real-world software engineering tasks. Specifically, it tests how the model can resolve GitHub issues from popular open-source Python repositories. For each task in the benchmark, the AI model is given a set up Python environment and the checkout (a local working copy) of the repository from just before the issue was resolved. The model then needs to understand, modify, and test the code before submitting its proposed solution.

Each solution is graded against the real unit tests from the pull request that closed the original GitHub issue. This tests whether the AI model was able to achieve the same functionality as the original human author of the PR.

SWE-bench doesn't just evaluate the AI model in isolation, but rather an entire "agent" system. In this context, an "agent" refers to the combination of an AI model and the software scaffolding around it. This scaffolding is responsible for generating the prompts that go into the model, parsing the model's output to take action, and managing the interaction loop where the result of the model's previous action is incorporated into its next prompt. The performance of an agent on SWE-bench can vary significantly based on this scaffolding, even when using the same underlying AI model.

There are many other benchmarks for the coding abilities of Large Language Models, but SWE-bench has gained in popularity for several reasons:

1. It uses real engineering tasks from actual projects, rather than competition- or interview-style questions;
2. It is not yet saturated—there's plenty of room for improvement. No model has yet crossed 50% completion on SWE-bench Verified (though the updated Claude 3.5 Sonnet is, at the time of writing, at 49%);
3. It measures an entire "agent", rather than a model in isolation. Open-source developers and startups have had great success in optimizing scaffoldings to greatly improve the performance around the same model.

Note that the original SWE-bench dataset contains some tasks that are impossible to solve without additional context outside of the GitHub issue (for example, about specific error messages to return). SWE-bench-Verified is a 500 problem subset of SWE-bench that has been reviewed by humans to make sure they are solvable, and thus provides the most clear measure of coding agents' performance. This is the benchmark to which we'll refer in this post.

Achieving state-of-the-art

Tool Using Agent

Our design philosophy when creating the agent scaffold optimized for updated Claude 3.5 Sonnet was to give as much control as possible to the language model itself, and keep the scaffolding minimal. The agent has a prompt, a Bash Tool for executing bash commands, and an Edit Tool, for viewing and editing files and directories. We continue to sample until the model decides that it is finished, or exceeds its 200k context length. This scaffold allows the model to use its own judgment of how to pursue the problem, rather than be hardcoded into a particular pattern or workflow.

The prompt outlines a suggested approach for the model, but it's not overly long or too detailed for this task. The model is free to choose how it moves from step to step, rather than having strict and discrete transitions. If you are not token-sensitive, it can help to explicitly encourage the model to produce a long response.

The following code shows the prompt from our agent scaffold:

```
<uploaded_files>
{location}
```

```
</uploaded_files>
I've uploaded a python code repository in the directory {location}
(not in /tmp/inputs). Consider the following PR description:

<pr_description>
{pr_description}
</pr_description>

Can you help me implement the necessary changes to the repository so
that the requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files
described in the <pr_description>. This means you DON'T have to modify
the testing logic or any of the tests in any way!

Your task is to make the minimal changes to non-tests files in the
{location} directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to
familiarize yourself with its structure.
2. Create a script to reproduce the error and execute it with `python
<filename.py>` using the BashTool, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well

Your thinking should be thorough and so it's fine if it's very long.
```

The model's first tool executes Bash commands. The schema is simple, taking only the command to be run in the environment. However, the description of the tool carries more weight. It includes more detailed instructions for the model, including escaping inputs, lack of internet access, and how to run commands in the background.

Next, we show the spec for the Bash Tool:

```
{
   "name": "bash",
   "description": "Run commands in a bash shell\n
* When invoking this tool, the contents of the \"command\" parameter
does NOT need to be XML-escaped.\n
* You don't have access to the internet via this tool.\n
* You do have access to a mirror of common linux and python packages
via apt and pip.\n
```

```
* State is persistent across command calls and discussions with the
user.\n
* To inspect a particular line range of a file, e.g. lines 10-25, try
'sed -n 10,25p /path/to/the/file'.\n
* Please avoid commands that may produce a very large amount of
output.\n
* Please run long lived commands in the background, e.g. 'sleep 10 &'
or start a server in the background.",
   "input_schema": {
       "type": "object",
       "properties": {
           "command": {
               "type": "string",
               "description": "The bash command to run."
           }
       },
       "required": ["command"]
   }
}
```

The model's second tool (the Edit Tool) is much more complex, and contains everything the model needs for viewing, creating, and editing files. Again, our tool description contains detailed information for the model about how to use the tool.

We put a lot of effort into the descriptions and specs for these tools across a wide variety of agentic tasks. We tested them to uncover any ways that the model might misunderstand the spec, or the possible pitfalls of using the tools, then edited the descriptions to preempt these problems. We believe that much more attention should go into designing tool interfaces for models, in the same way that a large amount of attention goes into designing tool interfaces for humans.

The following code shows the description for our Edit Tool:

```
{
   "name": "str_replace_editor",
   "description": "Custom editing tool for viewing, creating and
editing files\n
* State is persistent across command calls and discussions with the
user\n
* If `path` is a file, `view` displays the result of applying `cat -
n`. If `path` is a directory, `view` lists non-hidden files and
```

```
directories up to 2 levels deep\n
* The `create` command cannot be used if the specified `path` already
exists as a file\n
* If a `command` generates a long output, it will be truncated and
marked with `<response clipped>` \n
* The `undo_edit` command will revert the last edit made to the file
at `path`\n
\n
Notes for using the `str_replace` command:\n
* The `old_str` parameter should match EXACTLY one or more consecutive
lines from the original file. Be mindful of whitespaces!\n
* If the `old_str` parameter is not unique in the file, the
replacement will not be performed. Make sure to include enough context
in `old_str` to make it unique\n
* The `new_str` parameter should contain the edited lines that should
replace the `old_str`",
...
```

One way we improved performance was to "error-proof" our tools. For instance, sometimes models could mess up relative file paths after the agent had moved out of the root directory. To prevent this, we simply made the tool always require an absolute path.

We experimented with several different strategies for specifying edits to existing files and had the highest reliability with string replacement, where the model specifies `old_str` to replace with `new_str` in the given file. The replacement will only occur if there is exactly one match of `old_str`. If there are more or fewer matches, the model is shown an appropriate error message for it to retry.

The spec for our Edit Tool is shown below:

```
...
   "input_schema": {
       "type": "object",
       "properties": {
           "command": {
               "type": "string",
               "enum": ["view", "create", "str_replace", "insert",
"undo_edit"],
               "description": "The commands to run. Allowed options
are: `view`, `create`, `str_replace`, `insert`, `undo_edit`."
           },
```

```
            "file_text": {
                "description": "Required parameter of `create` command,
with the content of the file to be created.",
                "type": "string"
            },
            "insert_line": {
                "description": "Required parameter of `insert` command.
The `new_str` will be inserted AFTER the line `insert_line` of
`path`.",
                "type": "integer"
            },
            "new_str": {
                "description": "Required parameter of `str_replace`
command containing the new string. Required parameter of `insert`
command containing the string to insert.",
                "type": "string"
            },
            "old_str": {
                "description": "Required parameter of `str_replace`
command containing the string in `path` to replace.",
                "type": "string"
            },
            "path": {
                "description": "Absolute path to file or directory,
e.g. `/repo/file.py` or `/repo`.",
                "type": "string"
            },
            "view_range": {
                "description": "Optional parameter of `view` command
when `path` points to a file. If none is given, the full file is
shown. If provided, the file will be shown in the indicated line
number range, e.g. [11, 12] will show lines 11 and 12. Indexing at 1
to start. Setting `[start_line, -1]` shows all lines from `start_line`
to the end of the file.",
                "items": {
                    "type": "integer"
                },
                "type": "array"
            }
        },
        "required": ["command", "path"]
    }
}
```

Results

In general, the upgraded Claude 3.5 Sonnet demonstrates higher reasoning, coding, and mathematical abilities than our prior models, and the previous state-of-the-art model. It also demonstrates improved agentic capabilities: the tools and scaffolding help put those improved abilities to their best use.

Model	Claude 3.5 Sonnet (new)	Previous SOTA
SWE-bench Verified score	49%	45%

Examples of agent behavior

For running the benchmark, we used the SWE-Agent framework as a foundation for our agent code. In our logs below, we render the agent's text output, tool calls, and tool responses as THOUGHT, ACTION, and OBSERVATION, even though we don't constrain the model to a fixed ordering.

The code blocks below will walk through a typical case of the Sonnet 3.5 solving a SWE-bench problem.

In this first block, you can see part of the initial prompt given to the model, with `{pr_description}` filled in with the real value from a SWE-bench task. Importantly, this task contains steps to reproduce the issue, which will give the model a valuable starting point to investigate.

```
<uploaded_files>
/repo
</uploaded_files>
I've uploaded a python code repository in the directory /repo (not in
/tmp/inputs). Consider the following PR description:

<pr_description>
linear_model.RidgeClassifierCV's Parameter store_cv_values issue
#### Description
Parameter store_cv_values error on
sklearn.linear_model.RidgeClassifierCV

#### Steps/Code to Reproduce
import numpy as np
from sklearn import linear_model as lm

#test database
```

```
n = 100
x = np.random.randn(n, 30)
y = np.random.normal(size = n)

rr = lm.RidgeClassifierCV(alphas = np.arange(0.1, 1000, 0.1),
normalize = True,
                                         store_cv_values =
True).fit(x, y)

#### Expected Results
Expected to get the usual ridge regression model output, keeping the
cross validation predictions as attribute.

#### Actual Results
TypeError: __init__() got an unexpected keyword argument
'store_cv_values'

lm.RidgeClassifierCV actually has no parameter store_cv_values, even
though some attributes depends on it.
```

The model responds and first uses the Edit Tool to view the repository structure. You can see the model's text output and tool call arguments under THOUGHT and ACTION, and part of the tool's output under OBSERVATION:

```
THOUGHT:
I'll help you implement the changes to satisfy the PR description.
Let's follow the steps you outlined.

1. First, let's explore the repo structure:
ACTION (str_replace_editor):
command:
view

path:
/repo

OBSERVATION:
Here's the files and directories up to 2 levels deep in /repo,
excluding hidden items:
/repo
/repo/PULL_REQUEST_TEMPLATE.md
/repo/ISSUE_TEMPLATE.md
/repo/sklearn
/repo/sklearn/decomposition
```

```
/repo/sklearn/kernel_ridge.py
/repo/sklearn/model_selection
/repo/sklearn/covariance
/repo/sklearn/cluster
/repo/sklearn/base.py
/repo/sklearn/dummy.py
/repo/sklearn/naive_bayes.py
...
```

Now that the model has a better understanding of the repository structure, it uses the Edit Tool to create a new script that it will use to reproduce the issue and test its fix:

```
THOUGHT:
Based on the repository structure, we'll need to modify the Ridge
Classifier in the `sklearn/linear_model` directory.

Let's create a script to reproduce the error:
ACTION (str_replace_editor):
command:
create

path:
/repo/reproduce_error.py

file_text:
import numpy as np
from sklearn import linear_model as lm

#test database
n = 100
x = np.random.randn(n, 30)
y = np.random.normal(size = n)

rr = lm.RidgeClassifierCV(alphas=np.arange(0.1, 1000, 0.1),
normalize=True,
                         store_cv_values=True).fit(x, y)

OBSERVATION:
File created successfully at: /repo/reproduce_error.py
```

The model then uses the Bash Tool to execute the script it wrote, and successfully reproduces the issue from the task:

```
THOUGHT:
Let's verify that we can reproduce the error:
ACTION (bash):
command:
python3 /repo/reproduce_error.py

OBSERVATION:
Traceback (most recent call last):
  File "/repo/reproduce_error.py", line 10, in <module>
    store_cv_values=True).fit(x, y)
TypeError: __init__() got an unexpected keyword argument
'store_cv_values'
```

From here on, the model uses the Edit Tool to change the source code in the repository and reruns its script to verify whether the change has resolved the issue:

```
THOUGHT:
The error is reproducible. Looking at the code, I can see that the
issue is that while the `RidgeClassifierCV` class inherits from
`_BaseRidgeCV`, it fails to pass the `store_cv_values` parameter from
its constructor through to the base class constructor.

Let's fix this:
ACTION (str_replace_editor):
command:
str_replace

path:
/repo/sklearn/linear_model/ridge.py

old_str:
    def __init__(self, alphas=(0.1, 1.0, 10.0), fit_intercept=True,
                 normalize=False, scoring=None, cv=None,
class_weight=None):
        super(RidgeClassifierCV, self).__init__(
            alphas=alphas, fit_intercept=fit_intercept,
normalize=normalize,
            scoring=scoring, cv=cv)
        self.class_weight = class_weight

new_str:
    def __init__(self, alphas=(0.1, 1.0, 10.0), fit_intercept=True,
                 normalize=False, scoring=None, cv=None,
```

```
class_weight=None,
                 store_cv_values=False):
        super(RidgeClassifierCV, self).__init__(
            alphas=alphas, fit_intercept=fit_intercept,
normalize=normalize,
            scoring=scoring, cv=cv, store_cv_values=store_cv_values)
        self.class_weight = class_weight
```

In this particular example, the model worked for 12 steps before deciding that it was ready to submit. The task's tests then ran successfully, verifying that the model's solution addressed the problem. Some tasks took more than 100 turns before the model submitted its solution; in others, the model kept trying until it ran out of context.

From reviewing attempts from the updated Claude 3.5 Sonnet compared to older models, updated 3.5 Sonnet self-corrects more often. It also shows an ability to try several different solutions, rather than getting stuck making the same mistake over and over.

Challenges

SWE-bench Verified is a powerful evaluation, but it's also more complex to run than simple, single-turn evals. These are some of the challenges that we faced in using it—challenges that other AI developers might also encounter.

1. **Duration and high token costs.** The examples above are from a case that was successfully completed in 12 steps. However, many successful runs took hundreds of turns for the model to resolve, and >100k tokens. The updated Claude 3.5 Sonnet is tenacious: it can often find its way around a problem given enough time, but that can be expensive;

2. **Grading.** While inspecting failed tasks, we found cases where the model behaved correctly, but there were environment setup issues, or problems with install patches being applied twice. Resolving these systems issues is crucial for getting an accurate picture of an AI agent's performance.

3. **Hidden tests.** Because the model cannot see the tests it's being graded against, it often "thinks" that it has succeeded when the task actually is a failure. Some of these failures are because the model solved the problem at the wrong level of abstraction (applying a bandaid instead of a deeper refactor). Other failures feel a little less fair: they solve the problem, but do not match the unit tests from the original task.

4. **Multimodal.** Despite the updated Claude 3.5 Sonnet having excellent vision and multimodal capabilities, we did not implement a way for it to view files saved to the filesystem or referenced as URLs. This made debugging certain tasks (especially those from Matplotlib) especially difficult, and also prone to model hallucinations. There is definitely low-hanging fruit here for developers to improve upon—and SWE-bench has launched a new evaluation focused on multi-modal tasks. We look forward to seeing developers achieve higher scores on this eval with Claude in the near future.

The upgraded Claude 3.5 Sonnet achieved 49% on SWE-bench Verified, beating the previous state-of-the-art (45%), with a simple prompt and two general purpose tools. We feel confident that developers building with the new Claude 3.5 Sonnet will quickly find new, better ways to improve SWE-bench scores over what we've initially demonstrated here.

Acknowledgements

Erik Schluntz optimized the SWE-bench agent and wrote this blog post. Simon Biggs, Dawn Drain, and Eric Christiansen helped implement the benchmark. Shauna Kravec, Dawn Drain, Felipe Rosso, Nova DasSarma, Ven Chandrasekaran, and many others contributed to training Claude 3.5 Sonnet to be excellent at agentic coding.

Claude

API

Team

Pricing

Research

Company

Customers

News

Careers

Press Inquiries

Support

Status

Availability

Twitter

LinkedIn

YouTube

Terms of Service – Consumer

Terms of Service – Commercial

Privacy Policy

Usage Policy

Responsible Disclosure Policy

Compliance

Privacy Choices