This post was written jointly by Marco Tulio Ribeiro and Scott Lundberg

Introduction

Open-source LLMs like Vicuna and MPT are popping up all over the place.
There's been discussion about how these models compare to commercial LLMs like ChatGPT or Bard, but most of the comparison has been around answers to simple questions.

As an example, the folks at LMSYSOrg did an interesting analysis (+1 for being automated and reproducible) comparing Vicuna-13B to ChatGPT on various short questions, which is great as a comparison of the models as simple chatbots. However, many interesting ways of using these LLMs typically require complex instructions and/or multi-turn conversations, and some prompt engineering. We think that in the 'real world', most people will want to compare different LLM offerings on *their* problem, with a variety of different prompts.

This blog post is an example of what such an exploration might look like, comparing two open source models (Vicuna-13B, MPT-7b-Chat) with ChatGPT on tasks of varying complexity.

Warmup: Solving equations

By way of warmup, let's start with a toy task (solving simple polynomial equations), where we can check the output for correctness and shouldn't need much prompt engineering.
This will be similar to the Math category in here, with the difference that we will evaluate models as correct / incorrect on ground truth, rather than using GPT-4 to rate the output.

Here is a simple function that generates a polynomial with distinct integer roots. This will give us both the input and the ground truth output.

```
import numpy as np
def random_polynomial(n_roots, low=1, high=100):
    roots = [np.random.randint(low, high) for _ in range(n_roots)]
    # unique roots only
    while len(set(roots)) != n_roots:
        roots = [np.random.randint(low, high) for _ in range(n_roots)]
    poly = np.polynomial.polynomial.Polynomial.fromroots(roots)
    a = poly.coef.copy()
    a = a[::-1]
    text= ''
    for i, coef in enumerate(a):
        if coef == 0:
            continue
        sign = ' + ' if coef > 0 else ' - '
        if i ==0:
            sign = ''
        elif coef < 0:
            coef = -coef
        if i == len(a) - 1:
```

```
            text += f'{sign}{coef}'
        else:
            if coef == 1:
                coef = ''
            power = f'^{len(a) - i - 1}' if len(a) - i - 1 > 1 else ''
            text += f'{sign}{coef}x{power}'
    text += ' = 0'
    return roots, text
roots, equation = random_polynomial(2, low=-10, high=10)
print('Roots', roots)
print(equation)
```

```
Roots [2, -7]
x^2 + 5.0x - 14.0 = 0
```

First, let's load the models (we use `guidance` throughout for easy comparison and control):

```
import guidance
import transformers
path = '/home/marcotcr/.cache/huggingface/hub/vicuna-13b'
mpt = guidance.llms.transformers.MPTChat('mosaicml/mpt-7b-chat',
device=1)
vicuna = guidance.llms.transformers.Vicuna(path, device_map='auto')
chatgpt = guidance.llms.OpenAI("gpt-3.5-turbo")
```

Quick digression on different syntaxes: each of these models have a their own *chat syntax*, e.g. here is how the same conversation would look like in Vicuna and MPT (where `[generated response]` is where the model would put its output):

Vicuna:

```
A chat between a curious user and an artificial intelligence
assistant. The assistant gives helpful, detailed, and polite answers
to the user's questions.
USER: Can you please solve the following equation? x^2 + 2x + 1 = 0
ASSISTANT: [generated response] </s>
```

MPT:

```
<|im_start|>system
- You are a helpful assistant chatbot trained by MosaicML.
- You answer questions.
- You are excited to be able to help the user, but will refuse to do
anything that could be considered harmful to the user.
- You are more than just an information source, you are also able to
write poetry, short stories, and make jokes."""
<|im_end|>
<|im_start|>user Can you please solve the following equation? x^2 + 2x
+ 1 = 0<|im_end|>
<|im_start|>assistant [generated response]<|im_end>
```

To avoid the tediousness translating between these, `guidance` supports a unified chat syntax that gets translated to the model-specific syntax when calling the model.
Here is the prompt we'll use for all models (note how we use `{{system}}`, `{{user}}` and `{{assistant}}` tags rather than model-specific separators):

```
find_roots = guidance('''
{{~#system~}}
{{llm.default_system_prompt}}
{{~/system}}

{{#user~}}
Please find the roots of the following equation: {{equation}}
Think step by step, find the roots, and then say:
ROOTS = [root1, root2...]
For example, if the roots are 1.3 and 2.2, say ROOTS = [1.3, 2.2].
Make sure to use real numbers, not fractions.
{{~/user}}

{{#assistant~}}
{{gen 'answer' temperature=0}}
{{~/assistant~}}''')
```

Let's try the prompt on a very simple example:

```
roots, equation = random_polynomial(2, low=-3, high=3)
print(roots)
print(equation)
```

```
[1, -2]
x^2 + x - 2.0 = 0
```

ChatGPT:

```
answer_gpt = find_roots(llm=chatgpt, equation=equation)
```

Vicuna:

```
answer_vicuna = find_roots(llm=vicuna, equation=equation)
```

MPT:

```
answer_mpt = find_roots(llm=mpt, equation=equation)
```

ChatGPT got it right, while Vicuna and MPT got it wrong (Vicuna didn't even follow the specified format). Let's write a simple regex to parse the output, so we can evaluate this on a few more expressions:

```
import re
def parse_roots(text):
    roots = re.search(r'ROOTS = \[(.*)\]', text)
    if not roots:
        return []
```

```
    roots = roots.group(1).split(',')
    try:
        roots = [float(r) for r in roots]
    except:
        roots = []
    return roots
def matches_groundtruth(roots, groundtruth):
    if len(roots) != len(groundtruth):
        return False
    gt = np.array(sorted(groundtruth))
    roots = np.array(sorted(roots))
    return np.all(np.abs(gt - roots) < 1e-3)

print('ChatGPT: ',
matches_groundtruth(parse_roots(answer_gpt['answer']), roots))
print('Vicuna: ',
matches_groundtruth(parse_roots(answer_vicuna['answer']), roots))
print('MPT: ', matches_groundtruth(parse_roots(answer_mpt['answer']),
roots))
```

```
ChatGPT:  True
Vicuna:  False
MPT:  False
```

We evaluate the prompt on quadratic equations with roots between -20 and 20:

```
answers = {
    'ChatGPT': [],
    'Vicuna': [],
    'MPT': []
}
correct = {
    'ChatGPT': [],
    'Vicuna': [],
    'MPT': []
}
for _ in range(20):
    roots, equation = random_polynomial(2, low=-20, high=20)
    answer_gpt = find_roots(llm=chatgpt, equation=equation,
silent=True)
    answer_vicuna = find_roots(llm=vicuna, equation=equation,
silent=True)
    answer_mpt = find_roots(llm=mpt, equation=equation, silent=True)
    answers['ChatGPT'].append(answer_gpt)
    answers['Vicuna'].append(answer_vicuna)
    answers['MPT'].append(answer_mpt)

correct['ChatGPT'].append(matches_groundtruth(parse_roots(answer_gpt['
answer']), roots))
```

```
correct['Vicuna'].append(matches_groundtruth(parse_roots(answer_vicuna
['answer']), roots))

correct['MPT'].append(matches_groundtruth(parse_roots(answer_mpt['answ
er']), roots))

print('Frequency of correct answers:')
print('ChatGPT: ', np.mean(correct['ChatGPT']))
print('Vicuna: ', np.mean(correct['Vicuna']))
print('MPT: ', np.mean(correct['MPT']))
```

```
Frequency of correct answers:
ChatGPT:  0.8
Vicuna:  0.0
MPT:  0.0
```

ChatGPT gets the right roots 80% of the time, while Vicuna and MPT never get them right. Let's see a few errors:

```
answers['Vicuna'][0]
```

```
answers['MPT'][0]
```

Vicuna does the math wrong, while MPT does not even attempt to solve it step by step. ChatGPT also makes some mistakes, often involving some math substep:

```
error = np.where(np.array(correct['ChatGPT']) == False)[0][0]
answers['ChatGPT'][error]
```

Since Vicuna and MPT failed on quadratic equations, let's look at even simpler equations, such as `x - 10 = 0`:

```
find_solution = guidance('''
{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}

{{#user~}}
Please find the solution to the following equation: {{equation}}
Think step by step, find the solution, and then say:
SOLUTION = [value]
For example, if the solution is x=3, say SOLUTION = [3].
Make sure to use real numbers, not fractions.
{{/user}}

{{#assistant~}}
{{gen 'answer' temperature=0 max_tokens=300}}
{{~/assistant~}}''')

def parse_solution(text):
    solution = re.search(r'SOLUTION = \[(.*)\]', text)
```

```
    if not solution:
        return []
    try:
        return [float(solution.group(1))]
    except:
        return []

correct1d = {
    'ChatGPT': [],
    'Vicuna': [],
    'MPT': []
}
answers1d = {
    'ChatGPT': [],
    'Vicuna': [],
    'MPT': []
}
for i in range(20):
    print(i)
    roots, equation = random_polynomial(1, low=-20, high=20)
    answer_gpt = find_solution(llm=chatgpt, equation=equation,
silent=False)
    answer_vicuna = find_solution(llm=vicuna, equation=equation,
silent=False)
    answer_mpt = find_solution(llm=mpt, equation=equation,
silent=False)
    answers1d['ChatGPT'].append(answer_gpt)
    answers1d['Vicuna'].append(answer_vicuna)
    answers1d['MPT'].append(answer_mpt)

correct1d['ChatGPT'].append(matches_groundtruth(parse_solution(answer_
gpt['answer']), roots))

correct1d['Vicuna'].append(matches_groundtruth(parse_solution(answer_v
icuna['answer']), roots))

correct1d['MPT'].append(matches_groundtruth(parse_solution(answer_mpt[
'answer']), roots))

print('Frequency of correct answers:')
print('ChatGPT: ', np.mean(correct1d['ChatGPT']))
print('Vicuna: ', np.mean(correct1d['Vicuna']))
print('MPT: ', np.mean(correct1d['MPT']))
```

```
Frequency of correct answers:
ChatGPT:  1.0
Vicuna:  0.9285714285714286
MPT:  0.23076923076923078
```

Surprisingly, MPT still fails to solve these. Vicuna still makes some mistakes:

```
error = np.where(np.array(correct1d['Vicuna']) == False)[0][0]
answers1d['Vicuna'][error]
```

```
error = np.where(np.array(correct1d['MPT']) == False)[0][0]
answers1d['MPT'][error]
```

```
error = np.where(np.array(correct1d['MPT']) == False)[0][1]
answers1d['MPT'][error]
```

Discussion

This was a very toy task, but served as an example of how to compare models with different chat syntax using the same prompt.
For this particular combination of toy task and prompt, ChatGPT far surpasses Vicuna and MPT in terms of accuracy (measured exactly, because we have ground truth).
Let's now turn to more realistic tasks, where evaluating accuracy is not as straightforward.

Extracting snippets + answering questions about meetings

Let's say we want our LLM to answer questions (with the relevant conversation segments for grounding) about meeting transcripts.
This is an application where some users might prefer to use open-source LLMs rather than commercial ones, for privacy reasons (maybe I don't want to send all of my meeting data to OpenAI).
Here is a toy meeting transcript to start with:

```
meeting_transcript = '''John: Alright, so we're all here to discuss
the offer we received from Microsoft to buy our startup. What are your
thoughts on this?
Lucy: Well, I think it's a great opportunity for us. Microsoft is a
huge company with a lot of resources, and they could really help us
take our product to the next level.
Steven: I agree with Lucy. Microsoft has a lot of experience in the
tech industry, and they could provide us with the support we need to
grow our business.
John: I see your point, but I'm a little hesitant about selling our
startup. We've put a lot of time and effort into building this
company, and I'm not sure if I'm ready to let it go just yet.
Lucy: I understand where you're coming from, John, but we have to
think about the future of our company. If we sell to Microsoft, we'll
have access to their resources and expertise, which could help us grow
our business even more.
Steven: Right, and let's not forget about the financial benefits.
Microsoft is offering us a lot of money for our startup, which could
help us invest in new projects and expand our team.
John: I see your point, but I still have some reservations. What if
Microsoft changes our product or our company culture? What if we lose
control over our own business?
Steven: You know what, I hadn't thought about this before, but maybe
John is right. It would be a shame if our culture changed.
Lucy: Those are valid concerns, but we can negotiate the terms of the
```

```
deal to ensure that we retain some control over our company. And as
for the product and culture, we can work with Microsoft to make sure
that our vision is still intact.
John: But won't we change just by virtue of being absorbed into a big
company? I mean, we're a small startup with a very specific culture.
Microsoft is a huge corporation with a very different culture. I'm not
sure if the two can coexist.
Steven: But John, didn't we always plan on being acquired? Won't this
be a problem whenever?
Lucy: Right
John: I just don't want to lose what we've built here.
Steven: I share this concern too'''
```

Let's start by just trying to get ChatGPT to solve the task for us

```
query1 = 'How does Steven feel about selling?'
```

```
qa_attempt1 = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}

{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
Here is a meeting transcript:
----
{{transcript}}
----
Please answer the following question:
Question: {{query}}
Extract from the transcript the most relevant segments for the answer,
and then answer the question.
{{/user}}

{{#assistant~}}
{{gen 'answer' temperature=0}}
{{~/assistant~}}''')
qa_attempt1(llm=chatgpt, transcript=meeting_transcript, query=query1)
```

While the response is plausible, ChatGPT did not extract any conversation segments to ground the answer (and thus fails our specification). Let's try a different prompt:

```
qa_attempt2 = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
Here is a meeting transcript:
```

```
----
{{transcript}}
----
Consider the following question:
Question: {{query}}
Now follow these steps:
1. Extract from the transcript the most relevant segments for the
answer
2. Answer the question.
{{/user}}
{{#assistant~}}
{{gen 'answer' temperature=0}}
{{~/assistant~}}'''
qa_attempt2(llm=chatgpt, transcript=meeting_transcript, query=query1)
```

This is better, but maybe we want to specify the output format a little more, e.g. let's say we want each segment to have a summary, and to keep the character names. Let's try again:

```
qa_attempt3 = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
Here is a meeting transcript:
----
{{transcript}}
----
Based on the above, please answer the following question:
Question: {{query}}
Please extract from the transcript whichever conversation segments are
most relevant for the answer, and then answer the question.
Note that conversation segments can be of any length, e.g. including
multiple conversation turns.
Please extract at most 3 segments. If you need less than three
segments, you can leave the rest blank.

As an example of output format, here is a fictitious answer to a
question about another meeting transcript.
CONVERSATION SEGMENTS:
Segment 1: Peter and John discuss the weather.
Peter: John, how is the weather today?
John: It's raining.
Segment 2: Peter insults John
Peter: John, you are a bad person.
Segment 3: Blank
ANSWER: Peter and John discussed the weather and Peter insulted John.
{{/user}}
{{#assistant~}}
```

```
{{gen 'answer' temperature=0}}
{{~/assistant~}}''')
qa_attempt3(llm=chatgpt, transcript=meeting_transcript, query=query1)
```

ChatGPT did extract relevant segments, but it did not follow our output format (it did not summarize each segment, nor did it have the participant's names). Let's try again, with more explicit instructions:

```
qa_attempt4 = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
Here is a meeting transcript:
----
{{transcript}}
----
Based on the above, please answer the following question:
Question: {{query}}
Please extract from the transcript whichever conversation segments are
most relevant for the answer, and then answer the question.
Note that conversation segments can be of any length, e.g. including
multiple conversation turns.
Please extract at most 3 segments. If you need less than three
segments, you can leave the rest blank.

Your output should have the following structure:
CONVERSATION SEGMENTS:
Segment 1: a summary of the first conversation segment
(segment here)
Segment 2: a summary of the second conversation segment
(segment here)
Segment 3: a summary of the third conversation segment
(segment here)
ANSWER: the answer to the question, supported by the segments above.

As an example of output format, here is a fictitious answer to a
question about another meeting transcript.
CONVERSATION SEGMENTS:
Segment 1: Peter and John discuss the weather.
Peter: John, how is the weather today?
John: It's raining.
Segment 2: Peter insults John
Peter: John, you are a bad person.
Segment 3: Blank
ANSWER: Peter and John discussed the weather and Peter insulted John.
{{/user}}
{{#assistant~}}
```

```
{{gen 'answer' temperature=0}}
{{~/assistant~}}''')
qa_attempt4(llm=chatgpt, transcript=meeting_transcript, query=query1)
```

Finally, we got ChatGPT to use the format we wanted. The root of the problem we're facing is that the OpenAI API does not allow us to do partial output completion (i.e. we can't specify how the assistant begins to answer), and thus it's hard for us to **guide** the output. If, instead, we use one of the open source models, we can guide the output more clearly, forcing the model to use our structure.
For example, here is how we might modify `qa_attempt3`:

```
qa_guided = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}

{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
----
{{transcript}}
----
Based on the above, please answer the following question:
Question: {{query}}
Please extract the three segment from the transcript that are the most
relevant for the answer, and then answer the question.
Note that conversation segments can be of any length, e.g. including
multiple conversation turns. If you need less than three segments, you
can leave the rest blank.

As an example of output format, here is a fictitious answer to a
question about another meeting transcript:
CONVERSATION SEGMENTS:
Segment 1: Peter and John discuss the weather.
Peter: John, how is the weather today?
John: It's raining.
Segment 2: Peter insults John
Peter: John, you are a bad person.
Segment 3: Blank
ANSWER: Peter and John discussed the weather and Peter insulted John.
{{/user}}

{{#assistant~}}
CONVERSATION SEGMENTS:
Segment 1: {{gen 'segment1' temperature=0}}
Segment 2: {{gen 'segment2' temperature=0}}
Segment 3: {{gen 'segment3' temperature=0}}
ANSWER: {{gen 'answer' temperature=0}}
{{~/assistant~}}''')
qa_guided(llm=vicuna, transcript=meeting_transcript, query=query1)
```

With this guidance, we get the right format the first time (and all the time). Let's see how MPT does:

```
qa_guided(llm=mpt, transcript=meeting_transcript, query=query1)
```

While MPT follows the format, it ignores the question and takes snippets from the format example rather than from the real transcript.
From now on, we'll just compare ChatGPT and Vicuna.

Let's try another question:

```
query2 = 'Who wants to sell the company?'
```

```
qa_attempt4(llm=chatgpt, system_prompt=chatgpt_system,
transcript=meeting_transcript, query=query2)
```

Ugh, it seems that we didn't fix the formatting issue with ChatGPT yet. Not only does it not summarize the segments, it also doesn't really answer the question. Let's try again, putting a one-shot example as a conversation round this time:

```
qa_attempt5 = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: What were the main things that happened in the meeting?
Here is a meeting transcript:
----
Peter: Hey
John: Hey
Peter: John, how is the weather today?
John: It's raining.
Peter: That's too bad. I was hoping to go for a walk later.
John: Yeah, it's a shame.
Peter: John, you are a bad person.
----
Based on the above, please answer the following question:
Question: {{query}}
Please extract from the transcript whichever conversation segments are
most relevant for the answer, and then answer the question.
Note that conversation segments can be of any length, e.g. including
multiple conversation turns.
Please extract at most 3 segments. If you need less than three
segments, you can leave the rest blank.
{{/user}}
{{#assistant~}}
CONVERSATION SEGMENTS:
Segment 1: Peter and John discuss the weather.
Peter: John, how is the weather today?
John: It's raining.
```

```
Segment 2: Peter insults John
Peter: John, you are a bad person.
Segment 3: Blank
ANSWER: Peter and John discussed the weather and Peter insulted John.
{{~/assistant~}}
{{#user~}}
You will read a meeting transcript, then extract the relevant segments
to answer the following question:
Question: {{query}}
Here is a meeting transcript:
----
{{transcript}}
----
Based on the above, please answer the following question:
Question: {{query}}
Please extract from the transcript whichever conversation segments are
most relevant for the answer, and then answer the question.
Note that conversation segments can be of any length, e.g. including
multiple conversation turns.
Please extract at most 3 segments. If you need less than three
segments, you can leave the rest blank.
{{/user}}
{{#assistant~}}
{{gen 'answer' temperature=0}}
{{~/assistant~}}''')
```

```
qa_attempt5(llm=chatgpt, transcript=meeting_transcript, query=query2)
```

It works this time (it also works on the original query):

```
qa_attempt5(llm=chatgpt, transcript=meeting_transcript, query=query1)
```

Let's see how Vicuna does on this one.

```
qa_guided(llm=vicuna, transcript=meeting_transcript, query=query2)
```

Vicuna just worked.
Let's now try both of these prompts on a different meeting transcript, the beginning of an interview with Elon Musk:

```
# Full transcript: https://www.rev.com/blog/transcripts/elon-musk-
interview-with-the-bbc-4-11-23-transcript
transcript2 = '''Interviewer: In Sachs that used to be in content
moderation. And we've spoken to people very recently who were involved
in moderation. And they just say there's not enough people to police
this stuff. Particularly around hate speech in the company. Is that
something that you-
Elon Musk: What hate speech are you talking about? I mean, you use
Twitter?
Interviewer: Right.
Elon Musk: Do you see a rise in hate speech? Just your personal
```

anecdote, do you? I don’t.
Interviewer: Personally, my For You, I would see I get more of that kind of content. Yeah. Personally. But I’m not going to talk for the rest of Twitter.
Elon Musk: You see more hate speech personally?
Interviewer: I would say see more hateful content in that.
Elon Musk: Content you don’t like, or hateful. Describe a hateful thing?
Interviewer: Yeah, I mean just content that will solicit a reaction. Something that may include something that is slightly racist or slightly sexist. Those kinds of things.
Elon Musk: So you think if it’s something is slightly sexist it should be banned?
Interviewer: No, I’m not saying anything. I’m saying-
Elon Musk: I’m just curious. I’m trying to understand what you mean by “hateful content.” And I’m asking for specific examples. And you just said that if something is slightly sexist, that’s hateful content. Does that mean that it should be banned?
Interviewer: Well, you’ve asked me whether my feed, whether it’s got less or more. I’d say it’s got slightly more.
Elon Musk: That’s why I’m asking for examples. Can you name one example?
Interviewer: I honestly don’t…
Elon Musk: You can’t name a single example?
Interviewer: I’ll tell you why. Because I don’t actually use that For You feed anymore. Because I just don’t particularly like it. Actually a lot of people are quite similar. I only look at my following.
Elon Musk: You said you’ve seen more hateful content, but you can’t name a single example. Not even one.
Interviewer: I’m not sure I’ve used that feed for the last three or four weeks. And I honestly couldn’t-
Elon Musk: Then how could you see the hateful content?
Interviewer: Because I’ve been using it. I’ve been using Twitter since you’ve taken it over for the last six months.
Elon Musk: Then you must have at some point seen the For You hateful content. I’m asking for one example.
Interviewer: Right.
Elon Musk: And you can’t give a single one.
Interviewer: And I’m saying-
Elon Musk: Then I say, sir, that you don’t know what you’re talking about.
Interviewer: Really?
Elon Musk: Yes. Because you can’t give a single example of hateful content. Not even one tweet. And yet you claimed that the hateful content was high. That’s false.
Interviewer: No. What I claimed-
Elon Musk: You just lied.
Interviewer: No, no. What I claimed was there are many organizations that say that that kind of information is on the rise. Now whether it has on my feed or not...'''

```
query3 = 'The interviewer says his claim was not about his personal
feed. Is this true?'

qa_attempt5(llm=chatgpt, transcript=transcript2, query=query3)

qa_guided(llm=vicuna, transcript=transcript2, query=query3)
```

Again, both work fine. Another question:

```
query4 = 'Does Elon Musk insult the interviewer?'
qa_attempt5(llm=chatgpt, transcript=transcript2, query=query4)

qa_guided(llm=vicuna,  transcript=transcript2, query=query4)
```

Vicuna, has the right format and even the right segments, but it surprisingly generates a completely wrong answer, when it says "Elon musk does not accuse him of lying or insult him in any way".
We tried a variety of other questions and conversations, and the overall pattern was that Vicuna was comparable to ChatGPT on most questions, but got the answer wrong more often than ChatGPT did.

Application: Using Bash

Now we try to get these LLMs to iteratively use a bash shell to solve individual tasks. Whenever they issue a command, we run it and paste the output back into the prompt, until the task is solved.

```
# A bash session with state
import pty
from subprocess import Popen
import os
import time
class BashSession:
    def __init__(self):
        self.master_fd, self.slave_fd = pty.openpty()
        self.p = Popen('bash',
              preexec_fn=os.setsid,
              stdin=self.slave_fd,
              stdout=self.slave_fd,
              stderr=self.slave_fd,
              universal_newlines=True)
        self.run('ls')
    def run(self, command):
        command = command + '\n'
        os.write(self.master_fd, command.encode())
        time.sleep(0.2)
        return '\n'.join(os.read(self.master_fd,
10240).decode().split('\n')[1:-1])
```

First, let's do ChatGPT. Again, since we can't specify the output format, we rely on a description of the format and on a one-shot example:

```
import re
terminal = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
Please complete the following task:
Task: list the files in the current directory
You can give me one bash command to run at a time, using the syntax:
COMMAND: command
I will run the commands on my terminal, and paste the output back to
you. Once you are done with the task, please type DONE.
{{/user}}
{{#assistant~}}
COMMAND: ls
{{~/assistant~}}
{{#user~}}
Output: guidance project
{{/user}}
{{#assistant~}}
The files or folders in the current directory are:
- guidance
- project
DONE
{{~/assistant~}}
{{#user~}}
Please complete the following task:
Task: {{task}}
You can give me one bash command to run at a time, using the syntax:
COMMAND: command
I will run the commands on my terminal, and paste the output back to
you. Once you are done with the task, please type DONE.
{{/user}}
{{#geneach 'commands'}}
{{#assistant~}}
{{gen 'this.command' temperature=0}}
{{~/assistant~}}
{{#user~}}
Output: {{set 'this.output' (await 'output')}}
{{~/user}}
{{/geneach}}''')
def run_task_chatgpt(task):
    t = terminal(llm=chatgpt, task=task)
    session = BashSession()
    for _ in range(10):
        # Extract command
        command = re.findall(r'COMMAND: (.*)', t['commands'][-1]
['command'])
```

```
        if not command or 'DONE' in t['commands'][-1]['command']:
            break
        command = command[0]
        output = session.run(command)
        t = t(output=output)
    return t

run_task_chatgpt(task)
```

Now, let's try a simple task.
We created a dummy repo in `~/work/project`, with file `license.txt` (not the standard `LICENSE` file name).
Without communicating this to ChatGPT, let's see if it can figure it out:

```
task = 'Find out what license the open source project located in
~/work/project is using.'
run_task_chatgpt(task)
```

Indeed, ChatGPT follows a very natural sequence, and solves the task.

For the open source models, we write a simpler (guided) prompt where there is a sequence of command-output:

```
guided_terminal = guidance('''{{#system~}}
{{llm.default_system_prompt}}
{{~/system}}
{{#user~}}
Please complete the following task:
Task: list the files in the current directory
You can run bash commands using the syntax:
COMMAND: command
OUTPUT: output
Once you are done with the task, use the COMMAND: DONE.
{{/user}}
{{#assistant~}}
COMMAND: ls
OUTPUT: guidance project
COMMAND: DONE
{{~/assistant~}}
{{#user~}}
Please complete the following task:
Task: {{task}}
You can run bash commands using the syntax:
COMMAND: command
OUTPUT: output
Once you are done with the task, use the COMMAND: DONE.
{{~/user}}
{{~#assistant~}}
{{#geneach 'commands'~}}
COMMAND: {{gen 'this.command' stop='\\n'}}
OUTPUT: {{shell this.command}}{{~/geneach}}
```

```
{{~/assistant~}}''')
class StatefulShellOpenSource:
    def __init__(self):
        self.session = BashSession()
    def __call__(self, command):
        if 'DONE' in command:
            raise StopIteration
        output = self.session.run(command)
        return output.strip()

shell = StatefulShellOpenSource()
task = 'Find out what license the open source project located in
~/work/project is using.'
t = guided_terminal(llm=vicuna, task=task, shell=shell)
```

Vicuna is not able to solve the task this time. Let's see how MPT does:

```
shell = StatefulShellOpenSource()
t = guided_terminal(llm=mpt, task=task, shell=shell)
```

Surprisingly, MPT works this time while Vicuna doesn't.

Besides privacy (we're not sending the session transcript to OpenAI), open source-models have a significant advantage: the whole prompt is a single LLM run (and we even accelerate it by not having it geneate the output structure tokens like `COMMAND:`).

In contrast, we have to make a new call to ChatGPT for each command, which is slower and more expensive.

Let's try a different bash task with ChatGPT and Vicuna:

```
task = 'Find all jupyter notebook files in ~/work/guidance that are
currently untracked by git'
run_task_chatgpt(task)
```

Once again, we run into a problem with ChatGPT not following our specified output structure (and thus making it impossible for us to use inside a program, without a human in the loop).
We fix this **particular** problem by changing the message when there is no output below, but we can't fix the general problem of not being able to *force* ChatGPT to follow our specified output structure.

```
    t = terminal(llm=chatgpt, task=task)
    session = BashSession()
    for _ in range(10):
        # Extract command
        command = re.findall(r'COMMAND: (.*)', t['commands'][-1]
['command'])
        if not command or 'DONE' in t['commands'][-1]['command']:
            break
        command = command[0]
```

```
        output = session.run(command)
        if not output:
            output = 'No output'
        t = t(output=output)
```

Now, ChatGPT *was* able to solve the problem after this small modification. Let's see how Vicuna does:

```
shell = StatefulShellOpenSource()
t = guided_terminal(llm=vicuna, task=task, shell=shell)
```

Vicuna follows our output structure, but unfortunately runs the wrong command to do the task.
We observed the same pattern for other tasks as well, and found ChatGPT to be more reliable.

```
shell = StatefulShellOpenSource()
t = guided_terminal(llm=mpt, task=task, shell=shell)
```

```
╭──────────────────────────────── Traceback (most recent call last) ─────────────────────────────────╮
│ /tmp/ipykernel_3685168/2592723886.py:2 in <module>                                                 │
│                                                                                                    │
│   1 shell = StatefulShellOpenSource()                                                              │
│ ❱ 2 t = guided_terminal(llm=mpt, task=task, shell=shell)                                           │
│   3                                                                                                │
│                                                                                                    │
│ /home/marcotcr/work/guidance/guidance/_program.py:216 in __call__                                  │
│                                                                                                    │
│   213 |   |   else:                                                                               │
│   214 |   |   |   loop = asyncio.new_event_loop()                                                  │
│   215 |   |   |   loop.create_task(new_program.update_display.run())                               │
# start the display updat   |
│ ❱ 216 |   |   |   loop.run_until_complete(new_program.execute())                                   │
│   217 |   |                                                                                        │
│   218 |   |   return new_program                                                                   │
│   219                                                                                              │
```

```
/home/marcotcr/.virtualenvs/guidance/lib/python3.9/site-packages/nest_
asyncio.py:84 in           
 run_until_complete

    81 |   |   |   if f is not future:
    82 |   |   |   |   f._log_destroy_pending = False
    83 |   |   |   while not f.done():
 ❱  84 |   |   |   |   self._run_once()
    85 |   |   |   |   if self._stopping:
    86 |   |   |   |   |   break
    87 |   |   |   if not f.done():

/home/marcotcr/.virtualenvs/guidance/lib/python3.9/site-packages/nest_
asyncio.py:120 in          
 _run_once

   117 |   |   |   |   break
   118 |   |   |   handle = ready.popleft()
   119 |   |   |   if not handle._cancelled:
 ❱ 120 |   |   |   |   handle._run()
   121 |   |   handle = None
   122 |
   123 |   @contextmanager

 /usr/lib/python3.9/asyncio/events.py:80 in _run
```

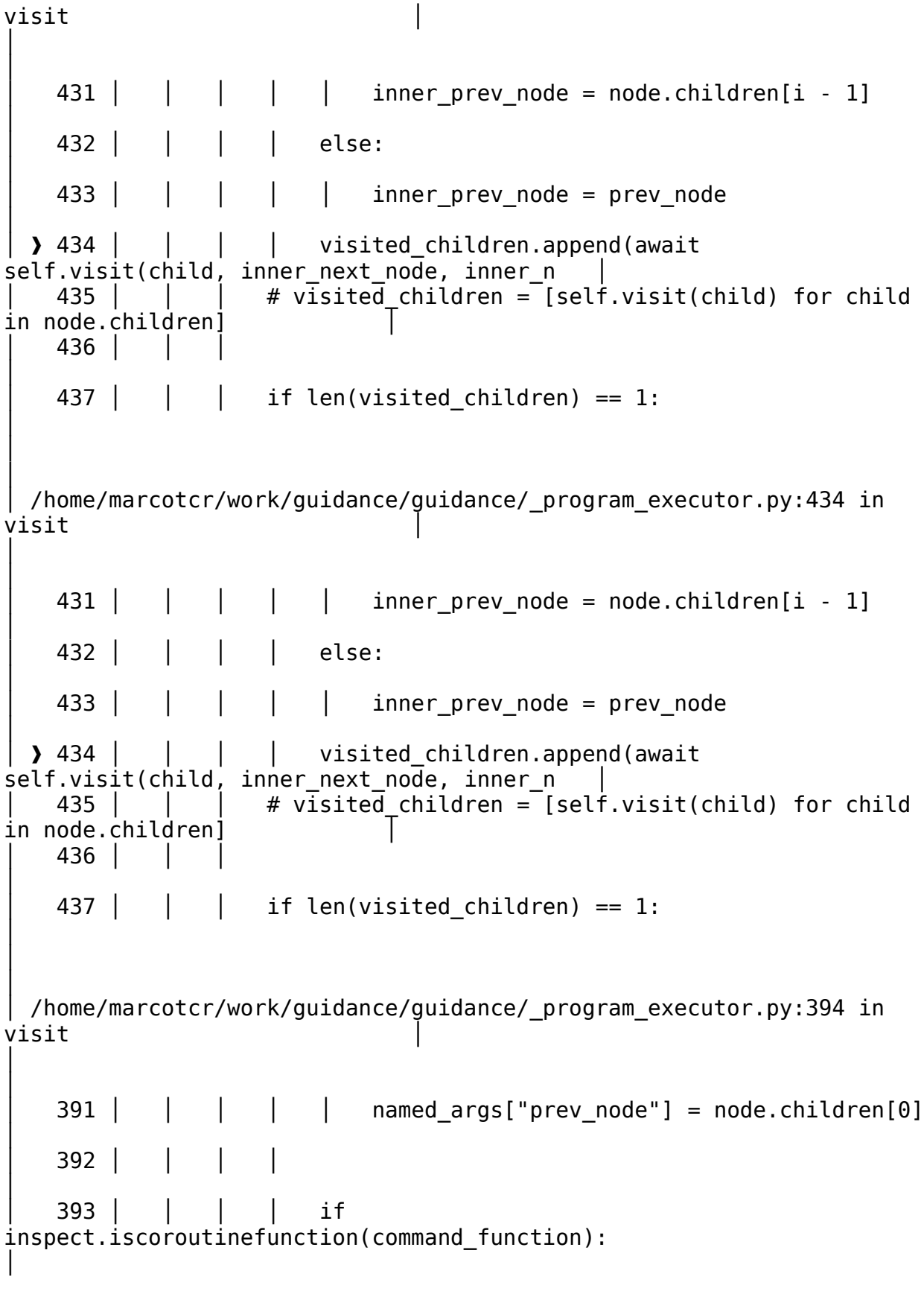
```
    77 |
    78 |   def _run(self):
    79 |   |   try:
 ❱  80 |   |   |   self._context.run(self._callback, *self._args)
    81 |   |   except (SystemExit, KeyboardInterrupt):
    82 |   |   |   raise
    83 |   |   except BaseException as exc:


/home/marcotcr/.virtualenvs/guidance/lib/python3.9/site-packages/nest_
asyncio.py:196 in step     |
   193 |   def step(task, exc=None):
   194 |   |   curr_task = curr_tasks.get(task._loop)
   195 |   |   try:
 ❱ 196 |   |   |   step_orig(task, exc)
   197 |   |   finally:
   198 |   |   |   if curr_task is None:
   199 |   |   |   |   curr_tasks.pop(task._loop, None)


 /usr/lib/python3.9/asyncio/tasks.py:256 in __step


   253 |   |   |   if exc is None:
   254 |   |   |   |   # We use the `send` method directly, because
coroutines                    |
   255 |   |   |   |   # don't have `__iter__` and `__next__`
methods.                              |
 ❱ 256 |   |   |   |   result = coro.send(None)
```

```
│
│   257 │   │   │   else:
│   258 │   │   │   │   result = coro.throw(exc)
│   259 │   │   except StopIteration as exc:
│
│  /home/marcotcr/work/guidance/guidance/_program.py:299 in execute
│
│   296 │   │
│   297 │   │   # run the program and capture the output
│   298 │   │   with self.llm.session(asynchronous=True) as
llm_session:                                │
│ ❱ 299 │   │   │   await self._executor.run(llm_session)
│   300 │   │   self._text = self._executor.prefix
│   301 │   │
│   302 │   │   # delete the executor and so mark the program as not
executing                     │
│
│  /home/marcotcr/work/guidance/guidance/_program_executor.py:94 in run
│
│    91 │   │   │   # self.whitespace_control_visit(self.parse_tree)
│    92 │   │   │
│    93 │   │   │   # now execute the program
│ ❱  94 │   │   │   await self.visit(self.parse_tree)
│    95 │   │   except Exception as e:
│    96 │   │   │   print(traceback.format_exc())
│    97 │   │   │   print("Error in program: ", e)
│
│  /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
```


visit
431 | | | | | inner_prev_node = node.children[i - 1]
432 | | | | else:
433 | | | | | inner_prev_node = prev_node
❱ 434 | | | | visited_children.append(await self.visit(child, inner_next_node, inner_n
435 | | | # visited_children = [self.visit(child) for child in node.children]
436 | | |
437 | | | if len(visited_children) == 1:
/home/marcotcr/work/guidance/guidance/_program_executor.py:434 in visit
431 | | | | | inner_prev_node = node.children[i - 1]
432 | | | | else:
433 | | | | | inner_prev_node = prev_node
❱ 434 | | | | visited_children.append(await self.visit(child, inner_next_node, inner_n
435 | | | # visited_children = [self.visit(child) for child in node.children]
436 | | |
437 | | | if len(visited_children) == 1:
/home/marcotcr/work/guidance/guidance/_program_executor.py:394 in visit
391 | | | | | named_args["prev_node"] = node.children[0]
392 | | | |
393 | | | | if inspect.iscoroutinefunction(command_function):

```
| ❱ 394 |   |   |   |   |   command_output = await
command_function(*positional_args, **named_ar   |
|   395 |   |   |   |   else:
|
|   396 |   |   |   |   |   command_output =
command_function(*positional_args, **named_args)      |
|   397
|
|
| /home/marcotcr/work/guidance/guidance/library/_assistant.py:8 in
assistant                          |
|
|    5 |
|
|    6 |   This is just a shorthand for {{#role
'assistant'}}...{{/role}}.                           |
|    7 |   '''
|
| ❱  8 |   return await role(name="assistant",
block_content=block_content, partial_output=part    |
|    9 assistant.is_block = True
|
|   10
|
|
|  /home/marcotcr/work/guidance/guidance/library/_role.py:12 in role
|
|
|    9 |   # send the role-start special tokens
|
|   10 |   partial_output(parser.program.llm.role_start(name))
|
|   11 |
|
| ❱ 12 |   out = await parser.visit(block_content[0],
next_node=next_node, prev_node=prev_node,    |
|   13 |
|
|   14 |   # send the role-end special tokens
|
|   15 |   partial_output(parser.program.llm.role_end(name))
|
|
|
| /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
visit                              |
```

```
│
│
│   431 │   │   │   │   inner_prev_node = node.children[i - 1]
│
│   432 │   │   │   │   else:
│
│   433 │   │   │   │   │   inner_prev_node = prev_node
│
│ ❱ 434 │   │   │   │   visited_children.append(await
self.visit(child, inner_next_node, inner_n   │
│   435 │   │   │   # visited_children = [self.visit(child) for child
in node.children]            │
│   436 │   │   │
│
│   437 │   │   │   if len(visited_children) == 1:
│
│
│
│ /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
visit                            │
│
│   431 │   │   │   │   inner_prev_node = node.children[i - 1]
│
│   432 │   │   │   │   else:
│
│   433 │   │   │   │   │   inner_prev_node = prev_node
│
│ ❱ 434 │   │   │   │   visited_children.append(await
self.visit(child, inner_next_node, inner_n   │
│   435 │   │   │   # visited_children = [self.visit(child) for child
in node.children]            │
│   436 │   │   │
│
│   437 │   │   │   if len(visited_children) == 1:
│
│
│
│ /home/marcotcr/work/guidance/guidance/_program_executor.py:394 in
visit                            │
│
│   391 │   │   │   │   named_args["prev_node"] = node.children[0]
│
│   392 │   │   │   │
│
│   393 │   │   │   │   if
inspect.iscoroutinefunction(command_function):
│
│ ❱ 394 │   │   │   │   command_output = await
```

```
command_function(*positional_args, **named_ar   │
│   395 │   │   │   else:
│
│   396 │   │   │   │   command_output =
command_function(*positional_args, **named_args)      │
│   397
│
│
│ /home/marcotcr/work/guidance/guidance/library/_geneach.py:74 in
geneach                           │
│
│    71 │   │   │   if len(data) > 0 and join != "":
│
│    72 │   │   │   │   partial_output(join)
│
│    73 │   │   │
│
│ ❱  74 │   │   │   await parser.visit(block_content[0]) # fills out
parser.prefix                     │
│    75 │   │   │   block_variables = parser.variable_stack.pop()
["this"]                          │
│    76 │   │   │   data.append(block_variables)
│
│    77 │   │   │   if hidden:
│
│
│ /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
visit                             │
│
│   431 │   │   │   │   inner_prev_node = node.children[i - 1]
│
│   432 │   │   │   else:
│
│   433 │   │   │   │   inner_prev_node = prev_node
│
│ ❱ 434 │   │   │   visited_children.append(await
self.visit(child, inner_next_node, inner_n   │
│   435 │   │   │   # visited_children = [self.visit(child) for child
in node.children]                 │
│   436 │   │   │
│
│   437 │   │   │   if len(visited_children) == 1:
│
│
│ /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
```

```
visit                          |
|
|
|   431 |   |   |   |   |   inner_prev_node = node.children[i - 1]
|
|   432 |   |   |   |   else:
|
|   433 |   |   |   |   |   inner_prev_node = prev_node
|
| ❱ 434 |   |   |   |   visited_children.append(await
self.visit(child, inner_next_node, inner_n   |
|   435 |   |   |   # visited_children = [self.visit(child) for child
in node.children]            |
|   436 |   |   |
|
|   437 |   |   |   if len(visited_children) == 1:
|
|
|
| /home/marcotcr/work/guidance/guidance/_program_executor.py:213 in
visit                          |
|
|
|   210 |   |   |
|
|   211 |   |   |   # visit our children
|
|   212 |   |   |   self.block_content.append([])
|
| ❱ 213 |   |   |   visited_children = [await self.visit(child,
next_node, next_next_node, prev_   |
|   214 |   |   |   self.block_content.pop()
|
|   215 |   |   |   out = "".join("" if c is None else str(c) for c in
visited_children)            |
|   216
|
|
|
| /home/marcotcr/work/guidance/guidance/_program_executor.py:213 in
<listcomp>                      |
|
|
|   210 |   |   |
|
|   211 |   |   |   # visit our children
|
|   212 |   |   |   self.block_content.append([])
|
| ❱ 213 |   |   |   visited_children = [await self.visit(child,
```

```
next_node, next_next_node, prev_   |
|   214 |   |   |   self.block_content.pop()
|
|   215 |   |   |   out = "".join("" if c is None else str(c) for c in
visited_children)           |
|   216
|
|
|
| /home/marcotcr/work/guidance/guidance/_program_executor.py:434 in
visit                           |
|
|   431 |   |   |   |   |   inner_prev_node = node.children[i - 1]
|
|   432 |   |   |   |   else:
|
|   433 |   |   |   |   |   inner_prev_node = prev_node
|
| ❱ 434 |   |   |   |   visited_children.append(await
self.visit(child, inner_next_node, inner_n   |
|   435 |   |   |   # visited_children = [self.visit(child) for child
in node.children]            |
|   436 |   |   |
|
|   437 |   |   |   if len(visited_children) == 1:
|
|
|
| /home/marcotcr/work/guidance/guidance/_program_executor.py:289 in
visit                           |
|
|   286 |   |   |   |   try:
|
|   287 |   |   |   |   |   if
inspect.iscoroutinefunction(command_function):                        |
|   288 |   |   |   |   |   |   await asyncio.sleep(0) # give other
coroutines a chance to run     |
| ❱ 289 |   |   |   |   |   |   command_output = await
command_function(*positional_args, **name   |
|   290 |   |   |   |   |   else:
|
|   291 |   |   |   |   |   |   command_output =
command_function(*positional_args, **named_args   |
|   292 |   |   |   |   except StopIteration as ret:
|
|
|
| /home/marcotcr/work/guidance/guidance/library/_gen.py:93 in gen
```

```
│
│
│    90 │   │   │   if logprobs is not None:
│    91 │   │   │   │   logprobs_list =
parser.get_variable(variable_name+"_logprobs", [])         │
│    92 │   │   │   │   logprobs_list.append([])
│
│ ❱  93 │   │   for resp in gen_obj:
│
│    94 │   │   │   await asyncio.sleep(0) # allow other tasks to run
│
│    95 │   │   │   #log("parser.should_stop = " +
str(parser.should_stop))                          │
│    96 │   │   │   if parser.should_stop:
│
│
│ /home/marcotcr/work/guidance/guidance/llms/_transformers.py:352 in
_stream_then_save             │
┬
│
│   349 │
│   350 │   def _stream_then_save(self, streamer, key, thread):
│   351 │   │   list_out = []
│ ❱ 352 │   │   for out in streamer:
│   353 │   │   │   list_out.append(out)
│   354 │   │   │   yield out
│   355 │   │   thread.join() # clean up the thread
│
│
│ /home/marcotcr/work/guidance/guidance/llms/_transformers.py:656 in
__next__                      │
┬
│
│   653 │   │   return self
│   654 │
│   655 │   def __next__(self):
│
│ ❱ 656 │   │   value = self.out_queue.get(timeout=self.timeout)
```

```
│   657 │   │   if value is None:                                                │
│   658 │   │   │   raise StopIteration()                                        │
│   659 │   │   else:                                                            │
│                                                                                │
│ /usr/lib/python3.9/queue.py:171 in get                                         │
│                                                                                │
│   168 │   │   │   │   │   raise Empty                                          │
│   169 │   │   │   elif timeout is None:                                        │
│   170 │   │   │   │   while not self._qsize():                                 │
│ ❱ 171 │   │   │   │   │   self.not_empty.wait()                                │
│   172 │   │   │   elif timeout < 0:                                            │
│   173 │   │   │   │   raise ValueError("'timeout' must be a non-               │
negative number")                                                              │
│   174 │   │   │   else:                                                        │
│                                                                                │
│ /usr/lib/python3.9/threading.py:312 in wait                                    │
│                                                                                │
│    309 │   │   gotit = False                                                   │
│    310 │   │   try:    # restore state no matter what (e.g.,                   │
KeyboardInterrupt)                                                             │
│    311 │   │   │   if timeout is None:                                         │
│ ❱  312 │   │   │   │   waiter.acquire()                                        │
│    313 │   │   │   │   gotit = True                                            │
│    314 │   │   │   else:                                                       │
│    315 │   │   │   │   if timeout > 0:                                         │
└────────────────────────────────────────────────────────────────────────────────┘
KeyboardInterrupt
```

Unfortunately, MPT fails this time, but calling the same command again and again (we had to interrupt to stop execution.)

Takeaways

In addition to the examples above, we tried various other inputs for both tasks (question answering and bash), and also tried a variety of other tasks involving summarization, question answering, and "creative" generation.
We also tried various toy string manipulation tasks, where we can evaluate accuracy automatically.
Here is a summary of our findings:

- **Quality on task**: For *every* task we tried, **ChatGPT is still stronger** than Vicuna on the task itself. MPT performed poorly on almost all tasks (perhaps we are using it wrong?), while Vicuna was often close to ChatGPT (sometimes very close, sometimes much worse as in the last example task above).

- **Ease of use**: It is much more painful to get ChatGPT to follow a specified output format, and thus it is harder to use it inside a program (without a human in the loop). Further, we always have to write regex parsers for the output (as opposed to Vicuna, where parsing a prompt with clear syntax is trivial).
  We are typically able to solve the structure problem adding more few-shot examples, but it is tedious to write them, and sometimes ChatGPT goes off-script anyway (we also end up with prompts that are longer, clumsier, and *uglier*).
  Being able to specify the output structure is a significant benefit of open-source models, which sometimes make them a better option even if they are a little worse on the task itself. Vicuna is often close enough to ChatGPT, and for easier tasks they may be indistinguishable.
- **Efficiency**: having the model locally means we can solve tasks in a single LLM run (`guidance` keeps the LLM state while the program is executing), which is faster and cheaper. This is particularly true when any substeps involve calling other APIs or functions (like search, terminal, etc), which always requires a new call to the OpenAI API. `guidance` also accelerates generation by not having the model generate the output structure tokens, which sometimes makes a big difference.

Now, it may be that these findings don't generalize, and are instead specific to the tasks and inputs we tried (or to the kinds of prompts we tend to write).
That may very well be true, but we think that anyone who tries to use LLMs for real-world tasks will have to go through a similar process to figure out which LLM makes more sense for their use case / preferred prompt style.
(This is of course not including considerations of cost, privacy, model versioning, etc)

In summary, our assessment is that MPT is not ready for real-world use yet (unless we're using it wrong), and that Vicuna is a viable alternative to ChatGPT (3.5) for many tasks (in part due to the ability to specify the output structure, since ChatGPT seems to still have stronger on-task performance).
We should acknowledge that we are biased by having used OpenAI models a lot in the past

few years, having written various papers that depend on GPT-3 (e.g. here, here), and a big paper that is basically saying "GPT-4 is awesome, here are a bunch of cool examples". While Vicuna is somewhat comparable to ChatGPT (3.5), we believe GPT-4 is a *much* stronger model, and are excited to see if open source models can approach *that*. `guidance` plays quite well with OpenAI models, but it really shines when you can specify output structure and accelerate generation, which right now you can only do with open source models.

Disclaimer: this post was written jointly by Marco Tulio Ribeiro and Scott Lundberg. It strictly represents our personal opinions, and not those of our employer (Microsoft).