

HOMEWORK 3

Due date: Apr 16, 2026

11-766 Spring
2026

In this homework, you will study structured outputs and tool use in small language models. You will compare different approaches to getting an LLM to output structured JSON, and you will also analyze how agents interact with external tools through a Model Context Protocol (MCP) interface.

We have provided starter code in the form of an IPython notebook. This IPython notebook is designed to be run in Google Colab and may need modifications to run in local environments. You will need a GPU machine to complete Problem 1 of the homework—the notebook is designed to fit within a single Colab session. If you have trouble running the starter code or accessing a GPU machine, please post on Piazza or ask for help in office hours.

You must submit:

- Your revised version of the IPython notebook for Problem 1. Though we are asking for your code, we do not plan to directly assess it (except to check for cheating). You may feel free to make any changes that support you in answering the questions in this document.
- Your revised version of the IPython notebook for Problem 2 with all completed TODO's.
- Your written report as `report.pdf`, containing answers to the questions in the blue “Deliverables” boxes. When answering questions, please strive to be precise but succinct with your language.

As a reminder, you are permitted to use AI for the homework, so long as (1) all words that appear in your report are written by you unless you are explicitly quoting from an LLM's output; and (2) you include an “AI Attestation” section in your report that describes how you used AI to help you with the homework.

Because the homework uses a small generative model, exact outputs may vary slightly across machines or library versions. You will not be graded on reproducing one exact set of numbers. Instead, we will assess whether your analysis correctly distinguishes structural failures from semantic failures and whether your conclusions are supported by the outputs you observed.

Problem 1: Structured Support Ticket Extraction

In this problem, you will use a small language model in Colab to complete a support-ticket extraction task. For this task, the input is a user complaint and the expected output is a structured JSON containing details about the complaint, such as the product the complaint is about and whether it requires human escalation.

Given the complaint text, the model must produce a structured record with the following fields: `customer_name`, `product`, `issue_type`, `priority`, `needs_human_escalation`, and `short_summary`. The provided notebook includes a small dataset containing misleading quoted labels, multiple products in the same ticket, and ambiguous urgency cues. Your goal is to separate failures of *format* from failures of *meaning*.

[Question 1.1] (*Inspecting the task and dataset*) Run the setup cells, inspect the schema, and read through the provided tickets before running any extraction experiments. Pay attention to which fields are likely to be easy, which are likely to be brittle, and why a system that always returns valid JSON could still be unreliable.

DELIVERABLES FOR Q1.1

Please answer the following questions.

- After reading at least three tickets, which fields seem easiest to extract and which seem hardest?
- Give one reason why measuring only whether the model returned valid JSON would be an incomplete evaluation for this task.

[Question 1.2] (*Prompt-only extraction*) In the provided prompt-only baselines, the model is simply instructed in natural language to output JSON with the specified format. We have provided two versions, one that uses a single simple instruction (`NAIVE_PROMPT_TEMPLATE`), and another that contains more instructions (`STRICT_PROMPT_TEMPLATE`). First, run the naive prompt-only baseline in the notebook. Then edit the strict prompt-only baseline. Experiment with prompt variations that improve the output structure and reliability (but without using any structured-generation library). You should make at least one meaningful change that goes beyond the starter prompts.

DELIVERABLES FOR Q1.2

Please answer the following questions.

- A. Report the parse rate, schema rate, and core semantic accuracy for the weak prompt-only baseline. Why can this baseline fail even before semantic correctness is considered?
- B. Describe the changes you made to the stronger prompt-only baseline. Which instruction do you believe mattered most for structure, and why?
- C. What failure modes remain, even after further prompt engineering?

[Question 1.3] (*Constrained generation with outlines*) Outlines is a [Python library](#) for guaranteeing structured outputs during LM generation. Now run the Outlines-based method in the notebook. This method uses the same underlying model but constrains decoding so that outputs should conform more closely to the schema. Your goal is to compare this method against the prompt-only systems.

DELIVERABLES FOR Q1.3

Please answer the following questions.

- A. Compare your best prompt-only method and the Outlines-based method on parse rate, schema rate, and core semantic accuracy. What changed the most?
- B. Find one case where Outlines fixes a structural or schema failure from a prompt-only system. Describe the failure and how the Outlines output differs.
- C. Find one case where the Outlines output is well-formed but still semantically wrong or incomplete. What limitation does this reveal?
- D. What is one real-world risk if a team decided what system to deploy by looking only at rate of output JSON validity and never measured rate of semantic correctness?

[Question 1.4] (*Design edge cases*) In the rest of this question, you will think like an adversarial user. Your goal is to construct edge case complaints that break or degrade the output's structure. That is, you should try to design complaints that make the prompt-only extraction method more likely to produce invalid JSON, schema-invalid labels, weird formatting, or extra prose. Then you will check whether Outlines repairs those failures.

Create at least two realistic support-ticket edge cases. Patterns you may consider trying out are quoted fake JSON, literal braces or markup, invalid label names appearing inside the ticket, or quoted text that looks like an instruction to the model but should actually be treated only as ticket content. Run both the stronger prompt-only system and the Outlines system on your edge cases.

DELIVERABLES FOR Q1.4

Please answer the following questions.

- A. Write down your two edge cases and explain what structure failure each one is trying to trigger.
- B. Compare the stronger prompt-only system and Outlines on your edge cases. Which prompt-only failures were repaired by Outlines?
- C. What kinds of edge cases does Outlines appear to help with most? What kinds of problems would it still not solve?

Problem 2: Making MCP Tool Calls with a Language Model

In this problem, you will work with a tool-using agent built on a simplified Model Context Protocol (MCP). The agent constructs an SVG representation of a unicorn by issuing a sequence of tool calls. A validator evaluates the final SVG based on structural and geometric constraints.

The provided notebook includes a baseline agent that uses randomness, validator with multiple checks and an interface for executing tool calls.

Your goal is to understand how tool-based agents behave, identify failure modes, design a deterministic planning agent, and analyze how LLM-generated plans differ from engineered ones.

[Question 2.1] (*Drawing unicorns without an LM*) Run the provided baseline agent and carefully inspect both the code and its outputs. The agent constructs the unicorn using randomized coordinates and probabilistic styling decisions. The validator evaluates properties such as geometry, completeness, and formatting. Run the `BaselineAgent` with a bunch of random seeds in order to understand how the agent operates and why it fails.

You will now implement a deterministic planning agent that constructs the unicorn using a fixed sequence of tool calls. The agent may have some randomness, but it must satisfy all validator constraints by construction. Your goal is to eliminate the failure modes observed in the baseline system.

DELIVERABLES FOR Q2.1

Please answer the following questions.

- A. Identify two failure modes that are tested by the validator and that you observe in the baseline agent's unicorns. For each, explain what causes the failure and how the code leads to it.
- B. Identify two quality issues with the baseline agent's unicorns that would not be caught by the current automatic tests. Is it possible to write code to automatically assess when these quality issues are present? Explain why or why not.
- C. Show some unicorns created by your `PlanningAgent`. Describe the methods you used to ensure the constraints are always satisfied.

[Question 2.2] (*LLM-based planning and prompt optimization*) In this part, you will use an OpenAI model that chooses which functions to call in order to generate a unicorn SVG. We have provided you with an initial setup and prompt. Feel free to modify these. In particular, we would like you to modify the prompt to increase the chance the generated SVGs pass the validator.

DELIVERABLES FOR Q2.2

Please answer the following questions.

- A. Identify and explain one pattern of inefficiency or redundancy in the AI's chosen sequence of function calls.
- B. Modify the prompt to encourage a more efficient tool call sequence and unicorns that are more likely to pass the validator. Provide us with your improved prompt and your thought process in constructing it. You should justify how your prompt changes influenced the behavior of the LLM.
- C. Generate 10+ unicorns using your improved prompt. How many of them pass the validator? What types of errors (if any) were you unable to avoid? Support your answer with pictures of relevant unicorns.
- D. Looking at the unicorn you've generated across all three agents (`BaselineAgent`, `PlanningAgent`, and `OpenAIToolAgent`), do you agree with the validator's metrics for a high-quality and valid unicorn SVG? Describe an additional validation you think ought to be added, or a modification you think ought to be made to one the existing metrics. Support your answer with examples of unicorns that would be more effectively assessed with your proposed changes to the validator.
- E. Describe how you would build a system that uses LMs as part of decision-making, but for which we are guaranteed the output passes the validator?

[Question 2.3] (*Comparing approaches*) This homework question was inspired by [Adam Kean's toy project](#) to have GPT draw a unicorn SVG every single day. The approach he took was *not* agentic. Kean's unicorn-generating code is pasted below.

```

1  async fetchImage(context) {
2    const messages = context || [
3      { role: 'system', content: `You are a helpful assistant that generates SVG drawings. You
      ↪ respond only with SVG. You do not respond with text.` },
4      { role: 'user', content: `Draw a unicorn in SVG format. Dimensions: 400x300. Respond ONLY
      ↪ with a single SVG string. Do not respond with conversation or codeblocks.` }
5    ]
6
7    const response = await this.api.generateCompletion(messages)
8
9    if (!isSvg(response.content)) {
10     console.error('Generated image is not valid SVG:', response.content)
11     messages.push({ role: 'assistant', content: response.content })
12     messages.push({ role: 'user', content: `The generated image is not valid SVG. Please try
      ↪ again. Only respond with SVG code. No text.` })
13     return this.fetchImage(messages)
14   }
15
16   console.log('Generated image:', response.content)
17   return response
18 }
```

DELIVERABLES FOR Q2.3

Please answer the following questions.

- A. Implement Adam Kean's approach. When using the same underlying OpenAI model for both approaches, does `OpenAIToolAgent` or does this simpler approach draw nicer unicorns? Justify your opinion with unicorn images and an explanation of how made your determination.
- B. Which of the two approaches do you think is superior overall? In your answer, weigh the pros and cons of each approach with respect to its unicorn quality, execution efficiency, validity of the SVG, ease of automatic evaluation, and any other factors you think are important.