MiniChain: A Small Library for Coding with Large Language Models

Alexander M. Rush
srush.research@gmail.com
Hugging Face
Cornell Tech

Abstract

Programming augmented by large language models (LLMs) opens up many new application areas, but also requires care. LLMs are accurate enough, on average, to replace core functionality, yet make basic mistakes that demonstrate a lack of robustness. An ecosystem of prompting tools, from intelligent agents to new programming languages, has emerged with different solutions for patching LLMs with other tools. In this work, we introduce *MiniChain*, an opinionated tool for LLM augmented programming, with the design goals of ease-of-use of prototyping, transparency through automatic visualization, and a minimalistic approach to advanced features. The *MiniChain* library provides core primitives for coding LLM calls, separating out prompt templates, and capturing program structure. The library includes demo implementations of the main applications papers in the area, including chat-bots, code generation, retrieval-based question answering, and complex information extraction. The library is open-source and available at `https://github.com/srush/MiniChain`, with code demos available at `https://srush-minichain.hf.space/`, and video demo at `https://www.youtube.com/watch?v=VszZ1VnO7sk`.

1 Introduction

Large language models (LLMs) (Brown et al., 2020) are a transformative technology that make it possible to develop novel AI applications. Out of the box they perform extremely well across many different domains including code generation, question answering and decomposition, fact retrieval, information extraction, and dialogue to name a few, as well as entirely novel task domains. However, while demonstrating these novel behaviors, they also struggle in basic areas such as mathematical reasoning (Hendrycks et al., 2021), code execution (Liu, 2022), specific document lookup (Guu et al., 2020), and handling long contexts (Shaham et al., 2022).

The gap between the novel general-purpose abilities and low-level deficiencies in known areas, has motivated significant research into multi-stage systems, colloquially *chains*, that combine the use of LLMs with more basic computation blocks and calls to other classical tools. This intermediary software ecosystem describes compositional structure for how the LLM interacts with the scaffolding around it.

Despite the agreed upon problem, there are many different approaches being pursued simultaneously in the open-source community. Systems like AutoGPT (AutoGPT) utilize a fully autonomous agent to direct the choice of supplementary tooling. Other systems like LMQL (Beurer-Kellner et al., 2023) propose a new query language that is able to fully guide and constrict the LLM. In between, the popular LangChain (Chase, 2022) software provides a full-service toolkit for working with many of the different paradigms, from agents to vector-databases to chat bots with memory. The diversity of these systems indicates a broad and open challenge in designing tools that best facilitate programmer interaction with LLMs.

In this white paper, we propose a system with a different goal along this design space. *MiniChain*, is an implementation of the prompt chaining paradigm that can support widely used chaining patterns while remaining significantly simpler than comparable libraries. MiniChain is designed with three goals: a) ease-of-use, it should be indistinguishable from standard python code, b) transparency, it should be trivial for the user to introspect and follow all AI calls, c) minimal, it should not implement features that can be done easily with code.

The library itself and the underlying visualization tool are written in python and follow standard coding conventions. The library aims to be understandable by researchers and is contained in one file. However, it is also meant to be complete, and

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 311–317
December 6-10, 2023 ©2023 Association for Computational Linguistics

be possible to implement contemporary research on the topic. To demonstrate this, the whitepaper comes with an implementation of recent prompting research at https://srush-minichain.hf.space/. In addition there is a video demo describing its use at https://www.youtube.com/watch?v=VszZ1VnO7sk.

2 Related Work

Programming with interleaved LLM calls is a very recent phenomenon, and so there are relatively comparable systems. Of the related systems, many exist as open-source libraries or as demo code, and it is difficult to categorize their evolving features. Roughly, prompt chaining systems can be divided into five groups:

Toolkits for calling LLMs and managing state. Of these the most representative and important is LangChain (Chase, 2022), a Python library for building LLM applications supporting multiple paradigms including explicit chaining, agent-based modeling, and vector lookups. Dust (Dust) proposes a different toolkit approach using Rust as a backend.

Programming languages and domain specific languages (DSLs) to support programming with prompts. These include LMQL (Beurer-Kellner et al., 2023) a DSL for constraining model output, Microsoft's Semantic Kernel (Microsoft, a), a heavy-weight toolkit supporting many different prompting paradigm across languages, and Demonstrate-Search-Predict (Khattab et al., 2022) a DSL describing systems that integrate retrieval and LLM decision making.

Collections of tools designed for LLM usage. Llama-index (Liu, 2022) is a collection of data resources and software meant to help LLMs respond to targeted queries. Other approaches focus on collecting additional models to consult, e.g. HuggingGPT/JARVIS (Shen et al., 2023), or open APIs to utilize as in Taskmatrix (Liang et al., 2023).

LLM toolkits designed to provide prompts for specific tasks. These libraries, such as Promptify (Pal, 2022), collect good versions of prompts that help solve specific zero-shot or few-shot tasks. These toolkits are less about the chaining component, but provide clear and usable prompt templates for the individual prompts. Many of the other libraries also provide clear prompts as part of their system.

Autonomous agents with prompt-supported tools. The goal of these systems is less to be integrated into software, and more to propose a different, (and more chaotic) way to solve specific problems through repeated prompting to determine and solve subtasks with external tools. AutoGPT and BabyAGI (AutoGPT; Nakajima) are the most well-known systems in this category.

Of these systems, MiniChain can be seen as fitting between the first and second category. It is an embedded domain specific language in Python with a minimal toolkit supporting common prompt-chaining paradigms.

3 Programming with LLMs

Let us begin by considering a practical example of chaining language models. Large language models have trouble with computing mathematical equations due to the limitations of fixed depth transformer models. For example, at the time of this writing asking Google Bard [1] to "sum the numbers 10 through 15" yields a confident assertion that the answer is 60.

However, researchers have noted that they are extremely good at code generation, and can map natural language descriptions of math problems into usable code, e.g. (Gao et al., 2023). This code can then be run to produce an answer. This motivates the base case for *chaining*. Given a word problem, we a) describe to the LLM what we want to do and have it convert it to code, and then b) run the code in an interpreter to produce the result. We will think of both of these steps as "prompting", i.e. first prompt the LLM and then prompt the interpreter.

The *MiniChain* API has the user describe both of these steps using simple functions. First we describe how to ask to map the problem to code.

```
@prompt(OpenAI(),
        template_file="math.pmpt.tpl")
def math_prompt(model, query):
    "Call GPT with a Jinja template"
    return model(dict(question=query))
```

The function takes two arguments, a special `model` argument representing the LLM and a user argument for the `query` to convert to code. The key additional component is the `@prompt` decorator which tells us which model to use (in this case the default OpenAI model), as well as a template file with the details of the prompt to use.

[1] https://bard.google.com/u/2/

Now let's look at the prompt itself stored as a Jinja Template [2]. The prompt is a few-shot description of the task itself. It contains several examples of questions and code outputs, as well as a template "hole" question to fill in with the user question before generating. (This format is inspired by the PromptSource system (Bach et al., 2022))

```
Question:
A robe takes 2 bolts of blue fiber
and half that much white fiber.
How many bolts in total does it take?
Code:
2 + 2/2
Question:
{{question}}
Code:
```

Next we describe the code for running this output in a Python interpreter. We treat all external models in the same manner, so the interface to the python interpreter behaves the same as an LLM. Instead of defining the prompt in its own file, we use the option to write the Jinja code into the decorator.

```
@prompt(Python(),
      template="import math\n{{code}}")
def python(model, code):
    "Call Python interpreter"
    code = "\n".join(
       code.strip().split("\n")[1:-1])
    return model(dict(code=code))
```

Given these two prompt construction functions, in the last step we can apply the chaining to produce our output. The API takes a question and then produces an answer by running the two together.

```
def math_demo(query):
    "Chain them together"
    return python(math_prompt(query))
```

However, while this last step looks natural, the semantics are a bit more complex. The MiniChain library uses lazy streams throughout, so this last step does not call the LLMs, but produces a compound `Prompt` object. This object has access to the entire graph of prompt operations constructed in this chain, roughly analogous to the backprop graph in autodifferentiation libraries.

To compute the final output for a user query we need to instantiate and run the chain.

```
math_demo("""What is the sum of the
           powers of 3 (3^i) that
           are smaller than 100?
""").run()
```

4 Visualizing the LLM Interactions

Since the system uses lazily instantiated chain of prompts with explicit prompt templates, it has full transparency into each step of the prompting process. This design makes it easy to extract and control the intermediate states of the system before and during runtime.

One particular benefit of this transparency is that it facilitates automatic interactive visualization and debugging. MiniChain includes a full visualization library built in based on the Gradio visualization library [3]. The visualization does not require any additional code, beyond what was shown in the previous section, and can be launched with the following command.

```
show(math_demo,
     examples=[...],
     subprompts=[math_prompt, python],
     out_type="markdown")
```

This command creates an automatic interactive visualization UI, which is shown in Figure 1. Starting from a text input, it kicks off and runs each step of the lazy chain showing the intermediate steps and output from the system.

The visualization shows each of the prompts, responses, model uses and the chain structure. Expanding the '. . . ' will show additional low level information like the raw template, variables used, and the commands for calling the underlying LLM model.

The visualization mode of the library also supports additional modalities. *Minichain* supports the ability to utilize models that are non-text based, such as images, video, and audio. In Figure 2, we have the model first write a story and then chain that with a Stable Diffusion model that draws the output into an image.

In addition to being lazy, the graph used in MiniChain is by default streaming. This means that the visualization can display partial outputs from a call to an LLM in real-time. For a slow model like GPT-4, showing intermediate results as

[2] https://jinja.palletsprojects.com/en/3.0.x/

[3] https://www.gradio.app/

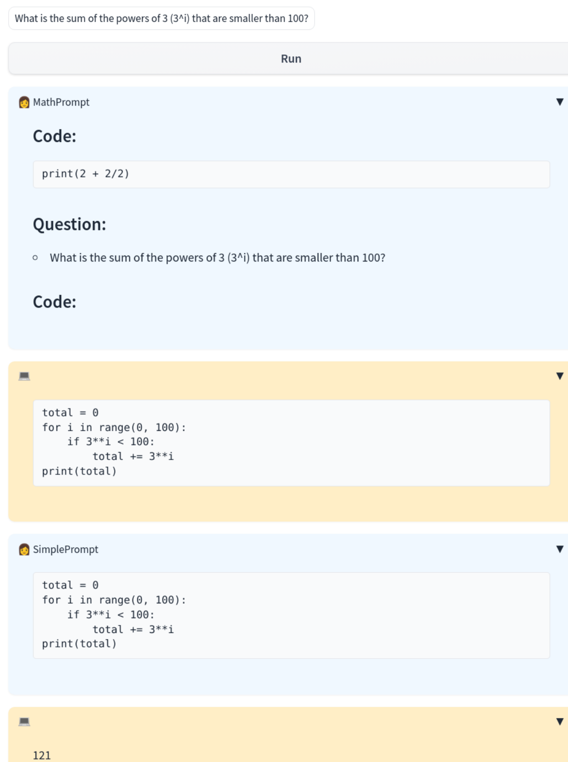

Figure 1: Automatic interactive visualization of the Math prompt showing input and outputs.

they are generating presents a much improved user experience. Practically this is configured through a customization of the `prompt` call. For example, in one of the examples we extract a table from a document as a CSV file. This code will convert the output to a well-formatted HTML table in real-time.

```
def to_html(out):
    return "..."

@prompt(OpenAI(),
        template_file="table.pmpt.txt",
        gradio_conf=GradioConf(
         block_output=gr.HTML,
         postprocess_output = to_html)
        )
def extract(model, passage):
    return model(dict(passage=passage))
```

5 Use-Cases

MiniChain is an opinionated library, and one of the goals is to not build additional features that are not related to chaining into the library. We argue for this minimality, by showing how popular prompt paradigms can be implemented without custom

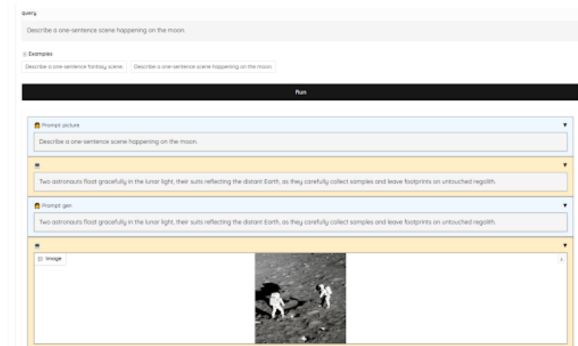

Figure 2: Visualization of a tool-use prompt-chain with image generation.

support.

5.1 Retrieval Augmentation

The process of chaining becomes more interesting if we want to allow intermediate processing and computation in addition to prompted calls to large language models. One popular use-case is to store dense embeddings in a vector database in order to support retrieval augmented question answering using large language models. In this section we consider the example of a question answering about the Olympics based [4]. The chain will a) compute the embedding of a passage, b) use it to lookup a corresponding Wikipedia article, c) use this article to answer the question.

We begin with a similar prompt as before, using an embedding template to process the question into a vector embedding with an LLM.

```
@prompt(HuggingFaceEmbed(),
        prompt="emb.pmpt.tpl")
def embed(model, query):
    return model(dict(question=query))
```

Next we need to look up this embedding in our vector database. Oftentimes it is overkill to have a full vector database, so we can simply use a local in-memory matrix to do this lookup. Here we use HuggingFace Datasets (Lhoest et al., 2021) to create our "vector database" by adding a FAISS index (Johnson et al., 2017).

```
d = datasets.load_from_disk("oly.data")
d.add_faiss_index("embeddings")
```

To use this vector lookup in the chain, we need to inject Python code into our LLM chain. We do this

[4] Used as an example in https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-2-create-qa.ipynb

with the `transform()` annotation, which allows us to lift a non-prompt to a function in the LLM chain. This function returns the k nearest neighbors.

```
@transform()
def neighbors(inp, k):
    return d.get_nearest_examples(
        "embeddings", np.array(inp), k)
```

Finally we introduce a prompt that uses the nearest neighbors and the original question to answer and construct the chain itself.

```
@prompt(OpenAI(),
        template_file="qa.pmpt.tpl")
def answer(model, query, neighbors):
    return model(dict(question=query,
                      docs=neighbors))

def qa(query):
    n = neighbors(embed(query), 3)
    return answer(query, n)
```

5.2 Agents and Tool Use

There has been significant excitement over the development of Agent based systems for LLMs that have the ability to process a confirmation and make use of various tools, such as AutoGPT (AutoGPT). While MiniChain does not have agent based behavior built in, it does have API features that make these systems possible to construct. A "tool" in MiniChain is just represented as having multiple models that can be called in a prompt. Prompts can be setup so that they can dynamically select which tool they should call next. This enables transparency in the visualization, while also maintaining each of use.

More tangibly, if we have a set of tools from some repository such as TaskMatrix (Liang et al., 2023), then we can have the model decide using plain python which to utilize.

```
tools = {tool1, tool2, ... }
@prompt(tools)
def tool_use(model, selector, command):
    return model(command,
        tool_num=tools[selector])
```

To build an Agent-based system all that is required is to combine this with a prompt and parsing command to determine which tool should be used. Here's an example of the prompt instructions given to the model and a parsing function.

```
Thought: Do I need to use a tool? Yes
Action: the action to take, should be
one of
[{% for tool in tools.keys()%}
{{tool[0]}},
{% endfor %}]
Action Input: the input to the action
```

```
@transform()
def tool_parse(out):
    lines = out.split("\n")
    if lines[0].endswith("Yes"):
        return lines[1], line[2]
    else:
        return Break()
```

Where `Break()` is a command to stop the chain from processing.

5.3 Chatbots and Memory

Given the fixed-length memory of LLMs, it is important to utilize available model context for higher-level tasks such as chat-like behavior. As such, libraries for chaining have devoted significant overhead to abstractions of memory to maintain previous contextual information.

MiniChain does not directly handle this problem, and instead relies on Python. Since chains are lazy and immutable, they do not provide any mechanism to maintain explicit state. To simulate mutability, the user needs to update and maintain their own history. Python support for easy immutable containers makes this relatively straightforward.

As an example, let us consider a chat example with a model that needs to remember the last N responses it has made to the user. We store this in a state data structure.

```
@dataclass(frozen=True)
class State:
    memory: List[Tuple[str, str]]
    human_input: str = ""
    # ...
```

We can then use this memory with a chain where at each step we update the state.

```
@prompt(OpenAI(), template_file="...")
def chat_response(model, state):
    return model(state)

@transform()
```

```python
def update(state, outp):
    return state.push(outp.split()[-1])
```

To construct the chain the main loop is just a for-loop passing the new state back to the next iteration of the chat_response.

6 Experimental Features

6.1 Auto-Prompts from Types

Utilizing LLMs in code requires having some certainty as to the intermediate types of the variables being produced. In practice even powerful models like GPT-4 have trouble consistently producing outputs of the expected form. MiniChain implements methods for notating and describing types. Specifically it includes a type_to_prompt method that allows users to specify specific types that they want the system to extract. It then uses this type specification to describe to the model the format.

```python
@dataclass
class Player:
    player: str
    stats: List[Stat]

@prompt(OpenAI(), template_file="...")
def stats(model, passage):
    return model(dict(passage=passage,
        typ=type_to_prompt(Player)))

@transform()
def to_data(s:str):
    return [Player(**j)
            for j in json.loads(s)]
```

A similar approach was recently implemented in TypeChat (Microsoft, b), a system that uses Javascript type annotations to automatically produce prompts and ensure adherence.

6.2 Back-tracking

LLMs in code is inherently a non-deterministic process. Even at temperature 0, many LLMs do not return deterministic results [5]. This behavior increases the importance of error checking within the chain itself. The lazy nature of *Minichain* makes it feasible to support handling errors through an explicit failure mechanism, and even allow the chain to back-up and retry its search again. Previous nodes in the chain will be able check the cause of the future failure and even update their prompt. This mechanism can be used to implement error correction by pointing out the failures of the output and revising based on failed output.

[5] https://twitter.com/BorisMPower/status/1608522707372740609

7 Conclusion

We describe *MiniChain*, a software toolkit for prompt chaining. The library focuses on creating an explicit chain of prompts each of which are simple Python functions. Each prompt separates out the descriptive language from the actual chain logic, and transformation logic can be specified with standard Python code. The system is transparent, which allows automatic interactive visualization with different modalities, streaming, and detailed debugging. The core language is minimal, but powerful enough to implement core prompt paradigms, such as retrieval, chat-bots, and tool-use. We hope this work demonstrates some of the possibilities of prompt programming, and encourages others to think about the APIs of these systems and how LLMs will integrate into software.

References

AutoGPT. Auto-GPT: An experimental open-source attempt to make GPT-4 fully autonomous.

Stephen H Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V Nayak, Abheesht Sharma, Taewoon Kim, M Saiful Bari, Thibault Fevry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged S Al-shaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir Radev, Mike Tian-Jian Jiang, and Alexander M Rush. 2022. PromptSource: An integrated development environment and repository for natural language prompts.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI):1946–1969.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are Few-Shot learners.

Harrison Chase. 2022. LangChain.

Dust. dust: Design and deploy large language model apps.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model Pre-Training. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3929–3938. PMLR.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-Search-Predict: Composing retrieval and language models for knowledge-intensive NLP.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander M Rush, and Thomas Wolf. 2021. Datasets: A community library for natural language processing.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. 2023. TaskMatrix.AI: Completing tasks by connecting foundation models with millions of APIs.

Jerry Liu. 2022. LlamaIndex.

Microsoft. a. semantic-kernel: Integrate cutting-edge LLM technology quickly and easily into your apps.

Microsoft. b. TypeChat: TypeChat is a library that makes it easy to build natural language interfaces using types.

Yohei Nakajima. babyagi.

Ankit Pal. 2022. Promptify: Structured output from LLMs. https://github.com/promptslab/Promptify.

Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. 2022. SCROLLS: Standardized CompaRison over long language sequences.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI tasks with ChatGPT and its friends in hugging face.