Prompto: An open source library for asynchronous querying of LLM endpoints

Ryan Sze-Yin Chan[1], **Federico Nanni**[1], **Angus R. Williams**[2], **Edwin Brown**[1,3], **Liam Burke-Moore**[2], **Ed Chapman**[1], **Kate Onslow**[2], **Tvesha Sippy**[2], **Jonathan Bright**[2], **Evelina Gabasova**[1]

[1] Research Engineering Group & [2] Public Policy Programme (The Alan Turing Institute),
[3] Research Software Engineering Team (University of Sheffield)
Corresponding authors: `{rchan,fnanni}@turing.ac.uk`

Abstract

Recent surge in Large Language Model (LLM) availability has opened exciting avenues for research. However, efficiently interacting with these models presents a significant hurdle since LLMs often reside on proprietary or self-hosted API endpoints, each requiring custom code for interaction. Conducting comparative studies between different models can therefore be time-consuming and necessitate significant engineering effort, hindering research efficiency and reproducibility. To address these challenges, we present `prompto`, an open source Python library which facilitates asynchronous querying of LLM endpoints enabling researchers to interact with multiple LLMs concurrently, while maximising efficiency and utilising individual rate limits. Our library empowers researchers and developers to interact with LLMs more effectively and allowing faster experimentation, data generation and evaluation. `prompto` is released with an introductory video[1] under MIT License and is available via GitHub[2].

1 Introduction

The field of Natural Language Processing is going through a massive transition since the introduction of Transformer-based Large Language Models (LLMs) (Vaswani et al., 2017; Devlin et al., 2019; Radford et al., 2019) which have demonstrated exceptional generalisation capability on a wide range of language-related tasks.

While user-friendly interfaces like ChatGPT, Gemini and Claude have made LLMs more accessible to the public, researchers nowadays increasingly interact with such models through programmatic interfaces and APIs. La Malfa et al. (2023) noted several reproducibility issues with this *Language-Models-as-a-Service (LMaaS)* paradigm (Sun et al., 2022), where language models are centrally hosted and typically provided on a subscription or pay-per-use basis (e.g., the OpenAI API[3] and Google's Gemini API[4]). To address some of them, Biderman et al. (2024) suggested a series of best practices when evaluating LLMs, such as sharing your exact prompts, parameter inputs and code and always providing model outputs.

In addition to LMaaS, there are now also several *open* LLMs, defined here as those with broadly available model weights as in Kapoor et al. (2024), e.g. Llama (Llama Team, 2024), Gemma (Gemma Team, 2024), Aya (Aryabumi et al., 2024). These "open" models are often accompanied by a *model card* (Wolf et al., 2020) which provides instructions for executing them locally and for accessing their internals and weights. This allows users to create their own API endpoints for LLMs (e.g., via Ollama[5]) on their available hardware.

A significant challenge with both LMaaS and open LLMs is that conducting a comparative study across multiple models necessitates writing separate code to interact with each API and this obstacle further hinders already complex evaluation and reproducibility efforts. An additional problem is that APIs may have different constrains (e.g., *query-per-minute (QPM)* rate limits), adding even more complexity to the engineering design.

To address these limitations and simplify large-scale comparative studies across LLMs, we introduce `prompto`, an open source Python library for *asynchronous* querying of LLM endpoints in a consistent and highly efficient manner. `prompto` uses asynchronous programming to efficiently interact with endpoints by allowing users to send multiple requests to different APIs concurrently. This eliminates idle wait times and maximises efficiency, especially when dealing with different rate limits. In contrast, in traditional synchronous programming,

[1] https://youtu.be/lWN9hXBOLyQ
[2] https://github.com/alan-turing-institute/prompto
[3] https://openai.com/api/
[4] https://ai.google.dev/gemini-api
[5] https://ollama.com/

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (System Demonstrations), pages 106–115
April 30, 2025 ©2025 Association for Computational Linguistics

a user sends a single request to an endpoint and waits for a response from the API before sending another request, repeating for each query. `prompto` supports a range of LMaaS endpoints (e.g. OpenAI, Gemini, Anthropic) as well as self-hosted endpoints for querying local models (e.g. Ollama, Hugging Face's `text-generation-inference`[6] for serving models hosted on Hugging Face). The codebase is easily extensible to integrate new APIs and/or locally self-hosted models. For instance, we provide an example using Quart[7] in `prompto` to easily set up an endpoint for inferencing local models using `transformers` (Wolf et al., 2020).

Inspired by Biderman et al. (2024); He et al. (2024), the library promotes experiment reproducibility by facilitating the definition of all inputs/prompts within a single JSON Lines (JSONL) or CSV file. The file can encompass queries for various APIs and models, enabling parallel processing for even greater efficiency gains. The library's scalability allows it to handle large-scale experiments. `prompto` also provides built-in functionalities for automatic evaluation of the obtained responses, such as allowing the user to apply scoring functions to model outputs, and model graded evaluation or LLM-as-a-judge (Zheng et al., 2024).

In this paper, we present an overview of `prompto`, highlighting its modular design and flexibility. We present `prompto`'s functionalities, design choices, technical implementation and show its advantages in comparison with alternative approaches. We accompany its release with extensive documentation and a series of Jupyter Notebooks as tutorials, to allow the research community to easily explore all its functionalities. In addition to the library's provided examples[8], we provide an illustrative showcase of using our library and compare against a synchronous approach to query several LLM endpoints in parallel in Appendix A.

2 Related Work and Motivation

To support transparent and reproducible evaluation of large language models, a series of evaluation frameworks have been published in recent years, such as *Language Model Evaluation Harness* (`lm-eval`[9]) (Gao et al., 2023; Biderman et al., 2024), the UK AI Safety Institute's *Inspect* framework (`inspect-ai`[10]), Confident AI's *DeepEval* (`deeepeval`[11]), *LLM Comparator* (`llm-comparator`[12]).

Nevertheless, conducting evaluation experiments on multiple large language models in order to compare their performance remains an engineering challenge, as each endpoint requires dedicated code to be written. To address this, our `prompto` library allows users to specify queries for various APIs and models within a single JSONL file, without having to interact directly with the details of each endpoint's infrastructure. This prioritises flexibility in experiment design, simplifies the setup for researchers and especially promotes reproducibility as all model parameters and prompts can be documented within a single file. After running an experiment, we record all model responses into an output experiment file which can be further analysed or used downstream in other pipelines.

Furthermore, our `prompto` library prioritises efficiency when running experiments. When interacting with LLMs, a sequential and synchronous approach where requests are sent to an endpoint and then waited upon for a response is highly inefficient, especially when dealing with different rate limits or when conducting comparisons across multiple models. `prompto` and tackles this challenge by leveraging asynchronous programming.

The ability to interact with multiple LLMs concurrently and process requests asynchronously makes our library particularly well-suited for initial exploration and rapid comparison between different LLMs. Researchers can efficiently experiment with various models and gauge their performance on specific tasks. This might be particularly useful in experiments where an "expected" or "ideal" response is not known ahead of time since evaluation framework tools are often most useful for settings where there is pre-defined model response from which the performance of an LLM can be scored against (e.g. automated metrics such as BLEU (Papineni et al., 2002), HellaSwag (Zellers et al., 2019) or MMLU (Hendrycks et al., 2021)). `prompto` may also be preferred in settings where the responses are intended for human evaluation, for instance in

[6] https://github.com/huggingface/text-generation-inference

[7] https://github.com/pallets/quart

[8] Example usage of `prompto` can be found at https://alan-turing-institute.github.io/prompto/examples/

[9] https://github.com/EleutherAI/lm-evaluation-harness

[10] https://github.com/UKGovernmentBEIS/inspect_ai

[11] https://github.com/confident-ai/deepeval

[12] https://github.com/pair-code/llm-comparator

Rao et al. (2023); Rashkin et al. (2023); Dash et al. (2023); Williams et al. (2024).

Finally, prompto's applicability goes beyond beyond evaluation tasks, and can be leveraged to generate synthetic datasets from LLMs for model training and development (Wang et al., 2022; Xu et al., 2024a,b), or to create datasets specifically tailored for LLM distilation tasks, a technique for compressing knowledge from larger models into smaller, more efficient ones (Taori et al., 2023; Wu et al., 2023; Peng et al., 2023; Gu et al., 2023).

3 Library Design

Our prompto library is an open source Python package[13] which facilitates processing of experiments of language models stored as JSONL files. The library has commands to process an experiment file to query models and store results. A user can query multiple models asynchronously and in parallel in a single experiment file.

We detail key components of setting up an experiment and core commands of the library. For a comprehensive exploration of prompto's features, we kindly refer readers to the detailed documentation at https://alan-turing-institute.github.io/prompto/.

3.1 Pipeline data folder

For running an experiment in prompto, everything starts with setting up a *pipeline data folder* (illustrated in Figure 1) which has several subfolders:

- input: contains input experiment JSONL files as described in 3.2
- output: contains results of experiment runs
- media: contains any input media files for multimodal model experiments

When an experiment is ran, a folder will be created in the output folder with the experiment name (the experiment file name without the ".jsonl" extension). We move the input file (originally stored in the input folder) into the output folder and rename it to indicate it was the original input file for a particular run. The model responses are stored in a new "completed" JSONL file, where each prompt dictionary will have a new "response" key storing the model response. A log file is also created to store any logs from the experiment run. Each of these files are timestamped to when the experiment started, allowing users to run the same experiment multiple times without interfering with the same output files and to track any changes to the input files between runs. An illustration of this data folder can be found in Figure 1.

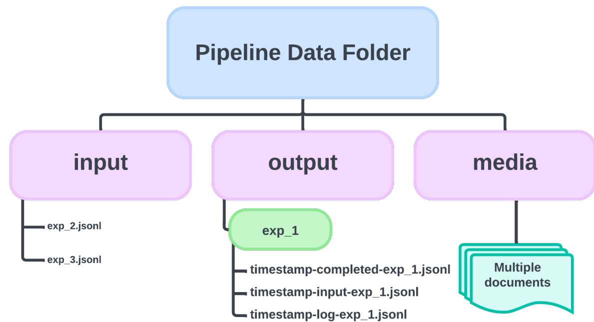

Figure 1: Illustration of the folder structure of the pipeline data folder after a exp_1.jsonl experiment file has been processed while exp_2.jsonl and exp_3.jsonl are experiment files yet to be processed in the input folder. After processing exp_1.jsonl, a exp_1 folder is created in the output folder which stores a completed experiment file, the original input file and a log file which are all timestamped to when the experiment was ran. The media folder stores any input media files for multimodal experiments.

3.2 Setting up a prompto experiment

A prompto *experiment file* is a JSON Lines (JSONL) file that contains the prompts for an experiment along with any other parameters or metadata. Each line in the JSONL file is a valid JSON value[14] which defines a particular input to a model. We refer to a single line in the JSONL file as a *prompt dictionary* (or prompt_dict) with keys:

- prompt: the prompt to the model
- api: the name of the API to query (e.g. "openai", "gemini", etc.)
- model_name: the name of the model to query (e.g. "gpt-4o", "gemini-1.5-flash", etc.)

The value of the prompt key is typically a *string* type that is passed to the model to generate a response, but for certain APIs or models, this could take different forms. For instance, for some API endpoints like OpenAI and Gemini, this could be a *list of strings* which we consider as a sequence of prompts to be sent as user messages in a chat interaction, or it could be a *list of dictionaries* each with "role" and "content" keys which can be used to define a *history* of a conversation. To foster the use of multimodal LLMs, prompto supports images and videos as inputs for several models. For instance,

[13] https://pypi.org/project/prompto/

[14] In practice, this means each line is a collection of key/-value pairs and in Python, this is realised as a dictionary.

for the OpenAI API, it is possible to prompt with images by passing them through the "content" keys. These can be URL links to images on the web, or be stored locally in the `media` folder of the pipeline data folder (see Section 3.1)[15].

There are also optional keys that can be included in a prompt dictionary such as "parameters" which define the parameters or *generation configuration* for a prompt such as *temperature*. Note that API services often have different names for the same generation parameter[16]. In `prompto`, these parameters are not unified and the generation parameters specified in the prompt dictionary are passed directly to the API used. A "group" key could be defined to pass a user-specified grouping of the prompts which can be useful for processing groups of prompts in parallel.

Note that CSV files can also be used as input where they get converted to JSONL to be processed. In this setting, the keys in the prompt dictionary discussed above are columns in the CSV file.

3.3 Running an experiment in `prompto`

An experiment can be ran in the terminal using the `prompto_run_experiment` command line interface (CLI) by specifying the path to input experiment file with the `--file` or `-f` argument. The user can also specify the path to the data folder (Section 3.1) with the `--data-folder` or `-d` argument[17]:

```
prompto_run_experiment
  --file path/to/experiment.jsonl \
  --data-folder data
```

For interacting with LLM endpoints, we often need to specify API keys or other variables. These can be set as environment variables, or they can be specified in a `.env` file[18] as key-value pairs:

```
OPENAI_API_KEY=...
GEMINI_API_KEY=...
```

The `prompto_run_experiment` CLI then starts asynchronously sending requests using `asyncio`[19] for each prompt dictionary which includes details of the input, the API to send to and the model name to query. There are several other arguments to the command for providing flexibility of how this process occurs[20] such as the maximum queries to send per minute (`--max-queries` or `-mq`), the maximum number of retries if errors (such as rate limit or connection errors) occur (`--max-attempts` or `-ma`). To adhere to strict API rate limits, we use the `max-queries` (per minute) argument to determine how frequently to send our requests asynchronously. If `max-queries` is set to 10, we simply send our requests every $60/10 = 6$ seconds. Since we use an asynchronous approach, we do not need to wait for a response before sending another request, ensuring that prompts are sent at correct intervals. To handle any errors (e.g. API failures or timeouts), we use the `max-attempts` argument to determine the maximum number of attempts for a prompt. In the case of an unexpected error, prompts are added to the back of the queue and are retried a maximum number of times. If the maximum is reached, we return and log the error in the response.

For example, to run an experiment where we asynchronously send 50 queries per minute (one query every 1.2 seconds) with 5 maximum retries, we can simply run the following command:

```
prompto_run_experiment \
  --file path/to/experiment.jsonl \
  --data-folder data \
  --max-queries 50 \
  --max-attempts 5
```

Furthermore, `prompto` supports sending requests to different APIs or models in parallel (`--parallel` or `-p`). Our implementation of the "parallel" processing again uses asynchronous programming allowing multiple queues of prompts to be managed concurrently within a single thread of execution rather than utilising multiple CPU cores and so is more resource efficient. If parallel processing of different APIs or models is requested, the user can fully customise how queues of prompts are constructed and set different rate limits for each queue. This is particularly useful when querying multiple API endpoints with different rate limits[21].

[15]A full example with multimodal prompting in `prompto` with the OpenAI API can be found at https://alan-turing-institute.github.io/prompto/examples/openai/openai-multimodal/.

[16]For example, to specify the maximum output tokens to generate, the OpenAI API uses `max_tokens` while Gemini API uses `max_output_tokens`.

[17]By default, we assume the data folder is called `data/` in the current working directory.

[18]By default, we look for a `.env` file in the current working directory, but a path can be specified using `--env` or `-e`.

[19]https://docs.python.org/3/library/asyncio.html

[20]Full details of each of these arguments can be found in our documentation at https://alan-turing-institute.github.io/prompto/docs/pipeline/

[21]Full details of parallel processing can be found at https://alan-turing-institute.github.io/prompto/docs/rate_limits/

Users can use all of prompto's functionalities directly in Python rather than using the CLI. For instance, to run an experiment in Python:

```
from prompto import Settings, Experiment

experiment_settings = Settings(
    data_folder="data",
    max_queries=50,
    max_attempts=5,
)
experiment = Experiment(
    file_name="path/to/experiment.jsonl",
    settings=experiment_settings,
)

await experiment.process()
```

In this example code, we are utilising the `Settings` and `Experiment` classes of the `prompto` library. The `Settings` class defines the settings of the experiment and stores all the paths to the relevant data folders, whereas the `Experiment` class stores all the relevant information for a single experiment and takes in a `Settings` object during initialisation. The `Experiment` class has an async method called `process` which runs the experiment.

For a more illustrative walkthrough of running experiments, the documentation includes examples of typical workflows with associated notebooks[8].

3.4 The `prompto` pipeline

Our `prompto` library also has the functionality to run a pipeline which continually looks for new experiment JSONL files in the input folder using the `prompto_run_pipeline` command which takes in the same settings arguments as the `prompto_run_experiment` command:

```
prompto_run_pipeline \
  --data-folder data \
  --max-queries 50 \
  --max-attempts 5
```

This command initialises the process of continually checking the input folder for new experiments to process. Consequently, there is no need to pass in a path to an experiment file to process as with `prompto_run_experiment`. If a new experiment is found, it is processed and the results and logs are stored in the output folder as explained in Section 3.3. The pipeline will continue to check for new experiments until the process is stopped. In the case where there are several experiments in the input folder, the pipeline will process the experiments in the order that the files were last modified.

In Python, it is possible to initialise this process in the following way:

```
from prompto import Settings,
    ExperimentPipeline

experiment_settings = Settings(
    data_folder="data",
    max_queries=50,
    max_attempts=5,
)
experiment_pipeline = ExperimentPipeline(
    settings=experiment_settings,
)

experiment_pipeline.run()
```

3.5 Automatic evaluation

A common use case for `prompto` is to evaluate different models, where we first need to obtain a large number of responses and then subsequently evaluate those responses. `prompto` provides functionality to automatically evaluate model responses.

One approach is to apply a simple *scoring function* to responses. A scoring function is typically lightweight such as performing string matching to some *target* output or some regex pattern matching. We have some built in scoring functions in the library such as a `match()` which determines whether the model response matches some expected response pre-defined by the user, or `includes()` which determines if the model responses includes a substring which is defined by the user.

In future work, we hope to expand this functionality to more advanced and computationally expensive scoring functions such toxicity classifiers (Inan et al., 2023; Hanu and Unitary team, 2020). While these can already be implemented as custom scorers in the library, scorers are applied independently to each output meaning batching is not fully utilised in the current implementation. A potential avenue is to utilise continuous batching strategies as in Kwon et al. (2023) to efficiently batch requests as responses are retrieved.

To automatically apply scoring functions to model outputs during an experiment run, one can simply use the `--scorers` argument to specify a comma-separated list of scoring functions:

```
prompto_run_experiment \
  --file path/to/experiment.jsonl \
  --data-folder data \
  --scorers match,includes
```

Another common approach to automatic evaluation is using an LLM to judge the outputs of a model

(Zheng et al., 2024). To perform an LLM-as-a-judge evaluation, the user must provide a prompt template which will be used to generate prompts to the Judge LLM for evaluation. In `prompto`, we treat this evaluation as simply as another `prompto` experiment where we obtain a new set of prompts using some judge evaluation template which includes a model response. An LLM-as-a-judge evaluation can be performed automatically after running an experiment. Full details and guidelines for running LLM-as-a-judge evaluations and all other evaluations features available with `prompto` can be found at https://alan-turing-institute.github.io/prompto/docs/evaluation/.

3.6 Rephrasing prompts

It is often useful to be able to rephrase/paraphrase a given prompt, particularly in evaluation settings since performance of models can vary significantly with choice of prompt or wording (Gonen et al., 2023; Mirzadeh et al., 2024; Hughes et al., 2024). In `prompto`, we provide functionality to simply use another language model to rephrase a given prompt. A *rephrasal experiment* can be constructed by using a template to prompt a model to rephrase/paraphrase. The responses from the rephrasal experiment (along with the original prompts), can be sent to another model to obtain responses for later evaluation or for any other purpose.

It's important to note that without safeguards, rephrasing may introduce semantic drift, leading to inconsistent evaluation results, so it is recommended that users also construct an evaluation of the rephrased prompts using methods discussed in Section 3.5 such as an LLM-as-a-judge evaluation. Full details and some examples of utilising the rephrasing functionality in `prompto` can be found at https://alan-turing-institute.github.io/prompto/docs/rephrasals/.

3.7 Adding new APIs and models to `prompto`

We have designed `prompto` to be easily extensible to integrate new LLM API endpoints and models. In particular, a user can add a new API by creating a new class which inherits from the `AsyncAPI` class from the `prompto` library. The user must then implement an async method `query` which asynchronously interacts with the model endpoint. Full details on implementing a new model endpoint can be found at https://alan-turing-institute.github.io/prompto/docs/add_new_api/.

4 Conclusions and Future Work

We present `prompto`, an open source library designed to facilitate researchers to efficiently query large language models which reside on proprietary or self-hosted API endpoints. Using `prompto` is simple and all inputs/prompts can be easily defined in a single JSONL file, which can encompass prompts for various APIs and models. Outputs are then stored in output JSONL files allowing researchers to easily share experiment outputs promoting reproducibility. Consequently, our library enables faster experimentation and evaluation. `prompto` is an ongoing open source project and we welcome contributions from the community to ensure support for a wide range of LLM endpoints and new features.

There are a number of interesting avenues for extending `prompto`. In the future, we hope to continue focusing on extending the evaluation pipelines discussed in Section 3.5. For evaluation, integration of larger-scale NLP pipelines can be useful in order to efficiently apply more advanced scoring functions such as toxicity classifiers and other safety risk classifiers for automatic safety evaluation. Furthermore, integrating prompto with data visualisation libraries can bring additional plotting features to the library to enable deeper exploration of model performance.

Another direction of work is to extend the rephrasing pipelines in `prompto` (see Section 3.6). A particular area of interest is extending the pipeline for translation of prompts. This can be useful for synthetic data generation applications. The current pipeline enables the use of LLMs for translation which have demonstrated remarkable potential in handling multilingual machine translation (MMT) (Zhu et al., 2024), however extending this pipeline to dedicated MMT models, such as NLLB (Costa-jussà et al., 2022; NLLB Team, 2024), could particularly be effective for translating to lower resourced languages. Moreover, a rephrasing pipeline can also be important in evaluation settings to generate a wider set of prompts to test a model. However, further research is required in order to do this effectively.

Lastly, future work includes expanding our tool's accessibility by developing direct local model integration capabilities. The current implementation requires setting up an API endpoint for querying, but we envision simplifying this process by enabling direct model instantiation (e.g.,

using `transformers` (Wolf et al., 2020)). While toolkits for deploying and serving LLMs, such as `text-generation-inference` or `vLLM` (Kwon et al., 2023), already simplify setting up efficient endpoints, this approach would reduce overhead for researchers who wish to query local models and utilise offline batched inference, allowing for more streamlined experimentation and rapid prototyping.

Acknowledgments

We thank Yi-Ling Chung, Florence Enock, Kobi Hackenburg for useful discussions on this library, and also thank Marie Chappell for their support on the project. This work was partially supported by the Ecosystem Leadership Award under the EPSRC Grant EPX03870X1, The AI Safety Institute & The Alan Turing Institute.

References

Viraat Aryabumi, John Dang, Dwarak Talupuru, Saurabh Dash, David Cairuz, Hangyu Lin, Bharat Venkitesh, Madeline Smith, Kelly Marchisio, Sebastian Ruder, et al. 2024. Aya 23: Open weight releases to further multilingual progress. *arXiv preprint arXiv:2405.15032*.

Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, Leo Gao, Jonathan Tow, Baber Abbasi, Alham Fikri Aji, Pawan Sasanka Ammanamanchi, Sidney Black, Jordan Clive, et al. 2024. Lessons from the Trenches on Reproducible Evaluation of Language Models. *arXiv preprint arXiv:2405.14782*.

Marta R Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, et al. 2022. No Language Left Behind: Scaling Human-Centered Machine Translation. *arXiv preprint arXiv:2207.04672*.

Debadutta Dash, Rahul Thapa, Juan M Banda, Akshay Swaminathan, Morgan Cheatham, Mehr Kashyap, Nikesh Kotecha, Jonathan H Chen, Saurabh Gombar, Lance Downing, et al. 2023. Evaluation of GPT-3.5 and GPT-4 for supporting real-world information needs in healthcare delivery. *arXiv preprint arXiv:2304.13714*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2023. A framework for few-shot language model evaluation.

Gemma Team. 2024. Gemma: Open Models Based on Gemini Research and Technology. *arXiv preprint arXiv:2403.08295*.

Hila Gonen, Srini Iyer, Terra Blevins, Noah Smith, and Luke Zettlemoyer. 2023. Demystifying prompts in language models via perplexity estimation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10136–10148, Singapore. Association for Computational Linguistics.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. MiniLLM: Knowledge Distillation of Large Language Models. *arXiv preprint arXiv:2306.08543*.

Laura Hanu and Unitary team. 2020. Detoxify. `https://github.com/unitaryai/detoxify`.

Chaoqun He, Renjie Luo, Shengding Hu, Ranchi Zhao, Jie Zhou, Hanghao Wu, Jiajie Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2024. UltraEval: A lightweight platform for flexible and comprehensive evaluation for LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 247–257, Bangkok, Thailand. Association for Computational Linguistics.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations*.

John Hughes, Sara Price, Aengus Lynch, Rylan Schaeffer, Fazl Barez, Sanmi Koyejo, Henry Sleight, Erik Jones, Ethan Perez, and Mrinank Sharma. 2024. Best-of-n jailbreaking. *arXiv preprint arXiv:2412.03556*.

Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, et al. 2023. Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations. *arXiv preprint arXiv:2312.06674*.

Sayash Kapoor, Rishi Bommasani, Kevin Klyman, Shayne Longpre, Ashwin Ramaswami, Peter Cihon, Aspen Hopkins, Kevin Bankston, Stella Biderman, Miranda Bogen, et al. 2024. On the Societal Impact of Open Foundation Models. *arXiv preprint arXiv:2403.07918*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E.

Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Raza Nazar, Anthony G Cohn, Nigel Shadbolt, and Michael Wooldridge. 2023. Language models as a service: Overview of a new paradigm and its challenges. *arXiv preprint arXiv:2309.16573*.

Llama Team. 2024. The Llama 3 Herd of Models. *Meta AI*.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models.

NLLB Team. 2024. Scaling neural machine translation to 200 languages. *Nature*, 630(8018):841.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction Tuning with GPT-4. *arXiv preprint arXiv:2304.03277*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Arya Rao, Michael Pang, John Kim, Meghana Kamineni, Winston Lie, Anoop K Prasad, Adam Landman, Keith Dreyer, and Marc D Succi. 2023. Assessing the utility of ChatGPT throughout the entire clinical workflow: development and usability study. *Journal of Medical Internet Research*, 25:e48659.

Hannah Rashkin, Vitaly Nikolaev, Matthew Lamm, Lora Aroyo, Michael Collins, Dipanjan Das, Slav Petrov, Gaurav Singh Tomar, Iulia Turc, and David Reitter. 2023. Measuring Attribution in Natural Language Generation Models. *Computational Linguistics*, 49(4):777–840.

Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. 2022. Black-Box Tuning for Language-Model-as-a-Service. In *International Conference on Machine Learning*, pages 20841–20855. PMLR.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. `https://github.com/tatsu-lab/stanford_alpaca`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. *Advances in Neural Information Processing Systems*, 30.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Models with Self-Generated Instructions. *arXiv preprint arXiv:2212.10560*.

Angus R Williams, Liam Burke-Moore, Ryan Sze-Yin Chan, Florence E Enock, Federico Nanni, Tvesha Sippy, Yi-Ling Chung, Evelina Gabasova, Kobi Hackenburg, and Jonathan Bright. 2024. Large language models can consistently generate high-quality content for election disinformation operations. *arXiv preprint arXiv:2408.06731*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Minghao Wu, Abdul Waheed, Chiyu Zhang, Muhammad Abdul-Mageed, and Alham Fikri Aji. 2023. LaMini-LM: A Diverse Herd of Distilled Models from Large-Scale Instructions. *arXiv preprint arXiv:2304.14402*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024a. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024b. Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing. *arXiv preprint arXiv:2406.08464*.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, et al. 2024. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*.

Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. 2024. Multilingual machine translation with

large language models: Empirical results and analysis. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2765–2781, Mexico City, Mexico. Association for Computational Linguistics.

A Experiments

We provide some illustrative examples[22] of using our prompto library and compare against a traditional synchronous approach to querying LLM endpoints. We show that our our pipeline provides significant speedups, especially when querying different models either on separate LLM endpoints (Appendix A.2) or available on the same API service (Appendix A.3).

In each of the following examples, we consider the task of generating instruction-following data using the Self-Instruct approach of Wang et al. (2022) and Taori et al. (2023). We take a sample of 100 prompts from the instruction-following data[23] from Taori et al. (2023) and apply the same prompt template. We then use these as prompt inputs to different models using prompto. Full details of how we obtained a sample of prompts along with notebooks to run each of the following experiments can be found at https://alan-turing-institute.github.io/prompto/examples/system-demo/.

Further examples including examples with prompting multimodal models with images and videos can be found on our GitHub repo and documentation[8].

A.1 Querying LLM endpoints asynchronously vs synchronously

We first compare prompto against a synchronous approach for three different LLM endpoints: the OpenAI API for gpt-3.5-turbo, Gemini API for gemini-1.5-flash and Ollama API for Llama 3. For using prompto for OpenAI and Gemini APIs, we send requests at 500 queries per minute (QPM). For Ollama, we send only at 50 QPM due to the limitations of the machine which we are running the Ollama server from[22]. We report run-times of the two approaches for obtaining 100 responses to sample prompts in Table 1.

For OpenAI and Gemini, even with a small sample of 100, we observe a significant speed up (of around 9 times or 12 times, respectively). Additionally, note that for OpenAI and Gemini APIs, there are tiers for which the query per minute rate limit is much higher than 500 QPM so further gains could be reached. For Ollama, we only observe a modest improvement in run-time since the Ollama API handles async requests by queuing the prompts for computation, so requests still get completed one-at-a-time. Therefore, in this setting prompto is simply putting prompts in a queue for the Ollama server. As noted in Section 4, batch inference strategies for local models and using alternative toolkits for serving LLMs such as vLLM can improve efficiency for querying local models and there is ongoing work to bring these to prompto.

Table 1: Run-time in seconds to obtain responses from 100 prompts from each API using a synchronous approach and using prompto.

	OpenAI	Gemini	Ollama
sync	126.31	163.49	271.45
prompto	13.92	14.09	268.59
speedup	9.07	11.60	1.01

A.2 Querying different LLM endpoints in parallel vs synchronously

As mentioned in Section 3.3, prompto supports sending requests to different APIs in parallel for greater efficiency gains. Further, the user can easily specify how many queries to send to each API or model per minute. To illustrate, we consider the same APIs and models as in Section A.1, but query the models in parallel. We compare against the baseline synchronous approach which we expect to be close to the sum of the individual run-times of the synchronous approaches in Table 1. We report run-times of the two approaches to obtain a total of 300 prompts (100 to each model/API) in Table 2.

We observe a 2 times speedup when using prompto with parallel processing. The prompto run-time is close to how long it took to process the Ollama requests in Section A.1 which is expected since the run-time of querying different APIs or models in parallel is simply dictated by the slowest API or model to query.

Table 2: Run-time in seconds to obtain responses from 100 prompts from each API in one run using a synchronous approach and enabling parallel processing of APIs with prompto.

	Overall
sync	558.74
prompto	269.06
speedup	2.08

[22] All experiments were ran on a 2021 MacBook Pro with M1 Pro and 32 GB of memory.

[23] https://github.com/tatsu-lab/stanford_alpaca/blob/main/alpaca_data.json

A.3 Querying different models from the same endpoint in parallel vs synchronously

Lastly, we illustrate how a user can query from different models available at the same endpoint using `prompto`. For this experiment, we consider the OpenAI API to query from three different models: `gpt-3.5-turbo`, `gpt-4` and `gpt-4o`. As with the previous two experiments, we will compare the run-times of querying the models individually as well as using `prompto` with parallel processing (i.e. sending requests to each model in parallel). For each model, we send requests at a rate of 500 QPM. We report run-times of this experiment in Table 3.

For each model and with parallel processing of the models, using `prompto` offers a signficant speedup in obtaining responses. Focusing on the synchronous run-times, we can see GPT-4o and GPT-4 are slower to obtain responses for than GPT-3.5-Turbo. Comparing using `prompto` with parallel processing against synchronously querying each model, we obtain an approximately 35 times speedup with `prompto`. As with the previous experiment in Section A.2, the run-time of `prompto` with parallel processing is roughly the time to query the slowest model, GPT-4 in this case.

Table 3: Run-time in seconds to obtain responses from 100 prompts from different models from the OpenAI API using a synchronous approach and using `prompto` and enabling parallel processing of models.

	GPT-3.5	**GPT-4**	**GPT-4o**	**Overall**
sync	130.73	392.21	241.24	705.38
`prompto`	14.29	19.79	18.11	19.30
speedup	9.15	19.82	13.32	36.55