The goal of this assignment is to get you familiar with the common framework that we use for training and inference.

Setting up the code

The starting code base is provided in https://github.com/llmsystem/llmsys_s25_hw5.git. You will need to first install the requirements, **please create a conda env with Python 3.9 to avoid dependency issue**:

```
module load cuda/12.6.1
 module load gcc/10.2.0
pip install datasets
pip install "sglang[all]>=0.4.4.post3" --find-links
https://flashinfer.ai/whl/cu124/torch2.5/flashinfer-python
```

Set up enviornment

We recommend using PSC for this homework, you can request an interactive session with 2 GPUs like this:

```
srun --partition=GPU-shared --gres=gpu:2 --time=4:00:00 --pty bash
```

To avoid storing model on your home directory (which will cause Disk Quota Exceed errors), you can set the default path for huggingface cahce.

```
export HF_HOME=/ocean/projects/cis240137p/<access-id>/huggingface
export HF_HUB_CACHE=/ocean/projects/cis240137p/<access-id>/huggingface_cache
```

Problem 1: Training with DeepSpeed ZeRO & LoRA (5)

For this problem, we will continue the training from our example, but with lora enabled so we can fit it in 2 GPUs. You will need to request for LLaMa model access here https://huggingface.co/meta-llama/Llama-2-7b-hf. Update `deepspeed/run_llama2_7b_lora.sh` to use LoRA, you can explore with different configuration that makes the training possible with 2 V100 GPUs/ 16GB GPU memory.

```
cd deepspeed
pip install -r requirements.txt
huggingface-cli login
bash run_llama2_7b_lora.sh
```

Problem 2: Inference with SGLang (5)

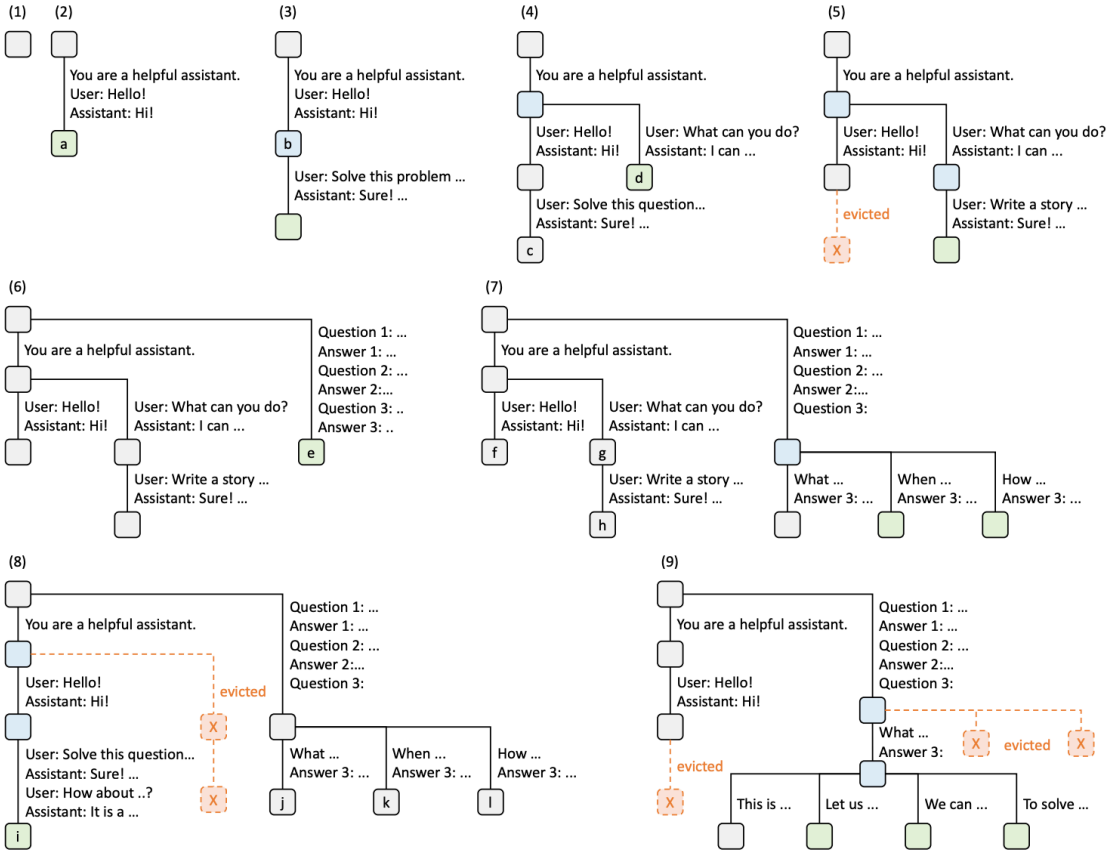
SGLang is an efficient system designed to provide higher throughput. SGLang consists of both the frontend language primitives and the runtime accleration. Spcifically for the runtime, there were two main methods:

- Efficient KV Cache Reuse with RadixAttention

To enable reusing the KV cache across multiple calls and instances, SGLang introduce RadixAttention where it retains the cache for prompts and generation results in a radix tree, instead of discarding after the generation. The radix tree, as a more space-efficient alternative to traditional trie tree, allows for faster prefix search, reuse, insertion, and eviction.

A LRU eviction policy and a cache-aware scheduling policy is also implemented to enhance the cache hit rate.

The LRU eviction policy evicts the least recently used leaf, to allow reuse of the common ancestors util the ancestors are also evicted.

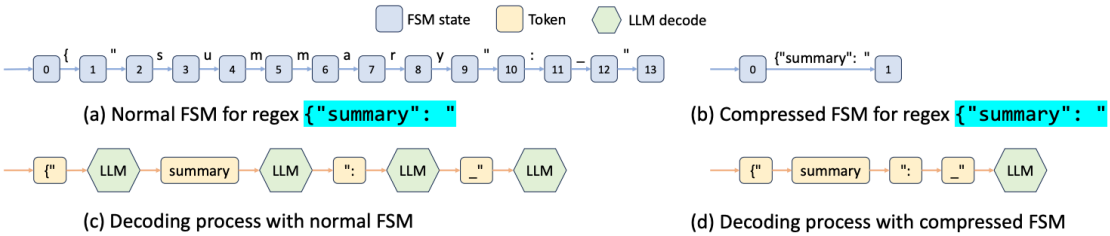


- Cache-aware scheduling

During batch-processing, the requests are sorted by matched prefix length in the cache. Requests with longer matched prefixes are prioritized as opposed to a FIFO.

- Efficient Constrained Decoding with Compressed Finite State Machine

SGLang creates a fast constrained decoding runtime with a compressed FSM, by analyzing the FSM and compresses adjacent singular-transition edges in the FSM into single edges, it enables multi-tokens to be decoded at once.



SGLang's backend in built on flashinfer.

To use SGLang for inference is very simple. Please try to fill out the TODOs in `sglang/run_sglang.py`. You should be able to finish all generations in under 15 minutes. You are also welcomed to explore different parameter settings to make the run faster!

```
cd sglang
python run_sglang.py
```