ORIE 4580/5580
Course Project: Modeling LLM-Query Serving Systems

(So that you get some sense of what is going on when you, and also all your classmates, ask an LLM to try and work on your Simulation project!)

1 Overview

The aim of this project is to use stochastic simulation to study the performance of modern Large Language Model (LLM) serving systems. These systems must handle highly variable user workloads, complex GPU interconnect architectures, and scheduling policies that balance latency, throughput, and efficiency. Your task will be to construct your own simulation model of such a system and use it to perform counterfactual studies to understand how scheduling, batching, and hardware/software co-design can improve throughput, tail latency, and cost.

Before diving into modeling, it is worth gaining a high-level understanding of how LLM inference works in real production systems. For this, you do not really need to know how an LLM (like GPT) works, but understand one important idea about LLM serving systems (such as ChatGPT) – the idea of the Key-Value cache. If you have never seen this, then before going on, take a look at this blogpost.

KV caches are the engineering reason why systems like ChatGPT work so well for everyone (it also explains why you often see that there is a wait after you ask a query, and then you start seeing the response being generated word-by-word). What is important for us is that each query goes through two distinct steps:

- **Prefill:** The model reads the entire prompt and prepares to emit the first token. This step is compute-heavy and tends to occupy more GPU resources.
- **Decode:** The model then generates the response one token at a time. Each token is relatively quick, but there can be hundreds of tokens, and many queries may be decoding concurrently.

These two steps have very different characteristics. Prefill iterations are much bigger, and saturate the compute resources of a typical GPU, but have high latency; decode iterations have much lower per-iteration latency but lower compute utilization, making batching highly effective for overall throughput.

When many users send queries at the same time, the service must decide *how* to process them efficiently and fairly—this is the scheduling problem. If we serve queries one-by-one, users can experience long waits and hardware may sit idle between steps. If we *batch* similar work together, we can raise throughput, but overly aggressive batching may increase the time until a user sees the first token (tail latency). Balancing these effects—high throughput versus low latency—is central to LLM scheduling.

Analogy (grading team resource allocation): As a (somewhat stretched) analogy for what is going on – imagine a grading team where one person prepares solutions and grading rubrics for assignments (slow, 2 hours per assignment), while other team members evaluate student submissions against the rubric (fast, 2 minutes per submission). You have assignments queued from multiple courses. How do you allocate team members? Do you have one person finish all of assignment A before starting B? Or do you have the solution writer work in parallel with graders evaluating earlier assignments? How many people should focus on solution prep versus grading? What if new assignments keep arriving?

This mirrors the prefill (setup, like solution preparation) versus decode (parallel grading) tradeoff in LLM serving. The key insight is that good resource allocation requires balancing slow, resource-intensive setup work with parallel throughput work—exactly the problem LLM schedulers face.

What you will do in this project:

- Build a stochastic simulation of an LLM-serving system under realistic arrivals and service stages (prefill/decodes).
- Explore simple scheduling and batching rules; measure their impact on throughput, mean latency, and tail latency under service-level objectives (SLOs).
- Perform counterfactual studies (e.g., with and without chunked-prefills; small versus large batches) and explain trade-offs to a non-expert audience.

By the end, hopefully you see how small local decisions (which query to serve next; whether to join a batch now or later) produce large global effects in capacity, cost, and user experience. In the process, you will also grapple with issues in simulating and reporting outcomes for steady-state systems, comparing different schedulers, and articulating trade-offs for decision makers.

In Section 3, we outline formal deadlines and deliverables; however in brief: the project should be done in **teams of 4–5 students** (not more, not less). The deliverables are in two parts: a basic simulation model, and the project report. Your initial simulation model (code + demo) is due by **December 8th** (the last day of classes); this should take about the same time as a long question in the assignments (plus the time to read this description and background reading). You must demonstrate your code to us in person. Your final reports (which is what should take more time and effort) are due on **December 17th at 11:59am ET**.

2 Project Outline and Research Questions

Notation Summary

To make this document easier to follow, here is a summary of key notation used throughout:

λ	arrival rate (queries per second)
L_i	prompt length of query i (tokens)
B_i	output budget (max response length) of query i (tokens)
K	maximum batch size (number of jobs per batch)
b	token load in a batch (sum of tokens across all jobs)
$S(b)$	service time for batch with token load b (milliseconds)
c	setup/fixed cost per batch (ms)
a	marginal cost per token beyond b_0 (ms/token)
b_0	minimum batch size threshold (tokens)
TTFT	Time To First Token (latency until first response)
TBT	Time Between Tokens (latency between successive tokens)
P_{95}	95th percentile latency

Below is a simple baseline model for studying query handling in an LLM-serving system, based on the systems paper by Agarwal et al., as well as follow-up modeling papers by Li et al. and Bari et al.. You can use this as a starting point, then adapt or extend it – in particular, note that the parameters we suggest are completely made up, and you may need to explore to find parameters that lead to interesting behavior, and/or are realistic.

- **System components:** The system has an input queue of requests, a scheduler, and a set of identical GPU workers hosting one LLM with a KV-cache. Every request experiences two phases: *prefill* (heavy, parallelizable) and *decode* (light per token, repeated many times). The scheduler can form batches at the level of iterations (a 'prefill chunk' or a 'decode step') and dispatch them to GPUs.
- **Arrivals and sizes:** You can model query arrivals as a Poisson process with rate λ. Each query i has a prompt length L_i (tokens) and an output budget B_i (tokens). For testing, start with simple distributions (e.g., $L_i \sim$ mixture of small/medium/long prompts; B_i you can model as a Geometric distribution, with some fixed upper bound (token budget) B (say 16 or 32 tokens), and this may be correlated with prompt length.
- **What is a GPU worker?** In modern LLM services, most computation happens on Graphics Processing Units (GPUs). These are specialized chips originally designed for rendering graphics but now widely used for AI because they can perform thousands of operations in parallel. A single GPU can process multiple queries at once if they are batched together.

  You can model a single 'GPU worker' as a server that executes jobs in 'batches' of size up to K. All jobs in a batch of size b start and finish simultaneously, with service time determined by some service time $S(b)$ (see Service Model below). For simplicity, you can assume that a single query has at most K tokens (i.e., each prefill task involves $L_i \leq K$ tokens), so can be done in one batch.

Importantly, since each decode token is generated sequentially (token $t+1$ requires outputs from token t), **each batch can contain at most one decode job from any given query**. However, you can batch decode jobs from *different* queries in the same batch. You can also have any number of prefill tokens from the same query in a batch.

You should start with a single GPU worker, but can then move on to having multiple GPUs in your system.

- **Service model:** An important question now is what is the processing time of a single batch in a GPU worker. This is not fully understood, but for our purposes, a good starting point is the model suggested in the paper Li et al.. Based on fitting experimental data, the authors suggest that the processing time of a batch can be well approximated by a piecewise linear function that only depends on the token load b (i.e., the total token count across all jobs in the batch) as:

  $$S(b) = c + a \cdot \max(0, b - b_0)$$

  Further, they write (in Section 2.1):

  > Fig. 4 demonstrates how well a linear fit approximates $S(b)$ as b varies. The parameters (c, a, b_0) depend on the LLM model and configuration. For example, in Llama-3-70B running on two A100 80GB GPUs with NVLink under tensor parallel, we observe $c = 45.5$ ms, $a = 0.30$ ms/token, and $b_0 = 64$ tokens.

  The main points are: (1) there is a fixed 'setup cost' per batch, irrespective of the size, and (2) the additional service time is linear in batch size after a certain minimal batch size b_0 (see Bari et al. for a more refined formula, which also explains why this behavior occurs).

  For your simulation implementation, one approach is to model the setup time and per-token service times as random variables: $C \sim \text{Exp}(1/c)$ and per-token service $A \sim \text{Exp}(1/a)$, where $1/c$ and $1/a$ are the mean values (in milliseconds). This simplifies the model while preserving the piecewise-linear structure in expectation; you can also try and use a more general distribution (maybe a phase-type distribution) to see how sensitive this assumption is.

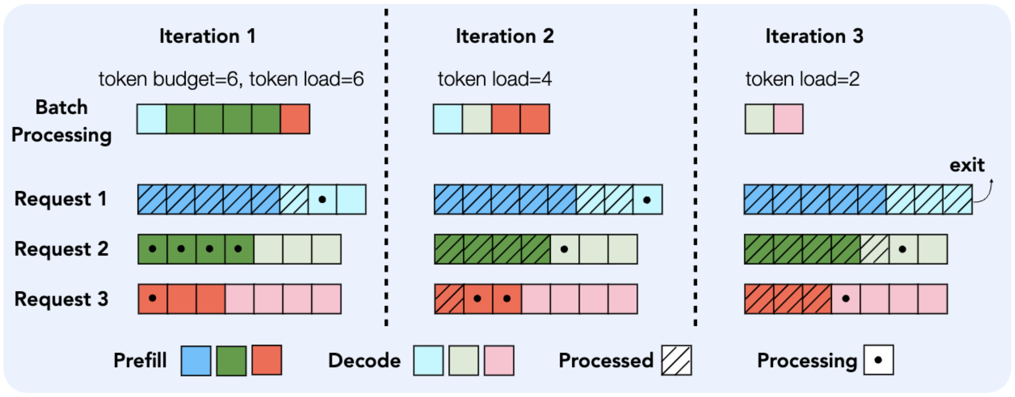
  Parameter guidance: For initial testing, you can use realistic values from the Llama-3 example above. To explore different hardware characteristics, you can experiment with scaled versions (e.g., $c = 20$–50 ms, $a = 0.1$–0.5 ms/token). Document your parameter choices and test sensitivity to changes in c and a, as these directly impact the throughput-latency tradeoff.

- **SLOs and metrics:** (i.e., what to measure) A well-designed system keeps each active GPU busy while also keeping response times low. This leads to two main goals: (1) achieving high **throughput**, measured in queries per second, and (2) maintaining low

latency, which directly affects user experience. Latency is typically measured using two Service-Level Objectives (SLOs):

- **Time To First Token (TTFT):** the delay between a query's arrival and the generation of the first output token (at the end of the prefill phase), capturing how long a user waits before the LLM starts responding.
- **Time Between Tokens (TBT):** the time between successive output tokens, indicating the rate at which the response is streamed to the user.

Moreover, most LLM-server systems track both mean and **tail latency** (i.e., the latency for the lowest 95% of queries, often denoted as P_{95}). Finally, on the server side, systems track metrics like GPU utilization, batch occupancy, and pipeline bubbles (idle gaps between iterations).



- **Scheduling Strategies:** The secret sauce that makes the above system work well is in designing good policies for scheduling/batching tasks in the available GPUs. The above diagram, reproduced from Li et al., illustrates how batching works for a single GPU. The important things to remember are:
  - For a given query, you need to first process all prefill tokens to generate the first output token, and then serve each decode token, each generating one output token.
  - **Baseline: Traditional Prefill Execution.** In traditional LLM systems, the prefill phase processes the entire prompt (i.e., all L_i prefill tokens) in a single batch on one GPU before moving to decode. This is simple to implement and matches the approach in older production systems.
  - **Advanced Feature (Optional): Chunked-Prefill.** Modern systems like Sarathi-Serve improve on this by splitting the prefill phase into smaller token chunks and interleaving them with decode iterations from other queries. For instance, instead of processing all 512 prompt tokens at once, you could process 64 tokens at a time, allowing short prompts to start decoding sooner. The L_i prefill tokens can be executed all at once (traditional), or separately in chunks of any size. These chunks can also be executed on different GPUs and in parallel; however all tokens must complete before moving on to decode tokens. Chunked-prefill prevents long prompts from blocking short queries, improves fairness, and reduces tail latency without

sacrificing too much throughput. Implementing this is an advanced extension (see Chunked-Prefill Experiments below).

- Decode tokens must be executed sequentially. If you have to generate B_i tokens, then this must be done one token at a time, with at most one decode job from each query in any GPU batch. You can load decode tokens from different queries on the same GPU, but again, must wait for earlier tokens to finish before loading new tokens in a batch.

This however still gives you a lot of freedom in terms of policies. An excellent summary is given in Section 1.1.2 in Bari et al., which we abridge below:

Schedulers differ in how they prioritize *prefill-phase* and *decode-phase* requests, and in their batching granularity. Below are some key approaches in the literature:

- **Decode-prioritizing (request-level) schedulers:** Frameworks such as Faster-Transformer and the request-level mode of TensorRT-LLM process a set of requests to completion before admitting any new requests. Since new prefill-phase requests never interrupt ongoing decode-phase requests, these schedulers perform well on *Time To First Token (TTFT)*. However, they exhibit low throughput when there is an imbalance in the total (prompt and output) length of requests, which can lead to under-utilization of the GPU.
- **Prefill-prioritizing (iteration-level) schedulers:** Iteration-level batching, first introduced by Orca, enables dynamic admission and completion of requests at each forward pass. vLLM increased the maximum number of concurrent requests to 128 and aggressively admits new prefill-phase requests to improve throughput. However, this eager admission policy can delay decode iterations—especially for long prompts—leading to higher TBT latencies.
- **Hybrid schedulers:** Sarathi-Serve introduces a token-budgeted, chunked-prefill strategy to balance throughput and TTFT, effectively reducing decode-iteration stalls that are common in prefill-prioritizing schedulers.

This gives you a simple initial model for LLM serving. It also gives you a sense of different scheduling policies, and the metrics of interest. You should first build a simulator that incorporates all these aspects, and then you can use this to figure out appropriate parameters, as well as compare different policies to understand the tradeoffs in this system.

For your demo, we expect you to show us (at least) a system with a single GPU worker, a single type of jobs (maybe fixed L_i, B_i), and comparisons of a basic scheduler which processes each query to completion, as well as maybe a basic prefill-prioritization policy (without prefill chunking), and plots showing average throughput, TTFT and TBT over time. Also you should validate your model by testing the case with no batching, no setup time (i.e., $c = 0$) and all exponential service times, and comparing it to results from queueing theory.

Below are some ideas for tasks and experiments you can perform once you have a working simulation model. These are intended to help you validate your simulator, explore design trade-offs, and communicate insights effectively; you do not need to do them all though. (You may want to skim through the earlier cited papers to understand the terminology/issues)

- **Validation and Benchmarking:** Identify special cases where system performance can be computed analytically (e.g., single GPU, no batching, exponential service times, or two GPUs with one used for prefills and one for decode tokens). Use these to verify correctness of your simulation.
- **Impact of Batching:** Compare throughput and tail latency under different batching strategies (e.g., fixed-size batches vs. dynamic batching). In **fixed-size batching**, the system waits until it has accumulated b tokens of work (or reaches a timeout), then immediately dispatches a batch to a GPU. In **dynamic batching**, batches are formed on-the-fly based on current queue depth and time thresholds, potentially dispatching smaller batches if waiting would exceed latency targets.
- **Scheduling Policies:** Implement and compare decode-prioritizing, prefill-prioritizing, and hybrid schedulers. Quantify trade-offs in TTFT (Time To First Token), TBT (Time Between Tokens), and overall throughput.
- **Chunked Prefills:** Study the effect of chunked-prefill strategies (i.e., breaking the prefill phase of a query into token chunks, which can be executed separately) on latency and throughput. Does chunking reduce stalls in decode iterations? How sensitive are results to chunk size?
- **Hardware Scaling:** Extend your model to multiple GPUs with realistic interconnect constraints. Explore how scaling affects performance under different workload intensities.
- **Workload Sensitivity:** Vary arrival rates, prompt lengths, and output budgets. Which workload characteristics most strongly influence tail latency? Can you identify 'bottlenecks/phase transitions' for the system?
- **Counterfactual Studies:** Evaluate hypothetical design changes—e.g., doubling GPU memory, changing batch admission rules, or introducing token-budgeted scheduling. Which changes yield the largest improvement in SLOs?
- **Visualization and Reporting:** Produce plots of latency distributions (mean, P_{95}), GPU utilization over time, and batch occupancy. Use these to explain trade-offs to a non-expert audience.

As with any real-world project, these suggestions are open-ended. You are encouraged to adapt the above ideas, propose new experiments, and explore additional metrics that highlight the complexity of LLM-serving systems. (Of course, you should also feel free to focus on the basic model, with simple policies, but write a good report, with proper measurements and well presented plots). The focus overall is on correctness and insights, rather than volume of things you do.

3 Deadlines and Deliverables

The project should be done in **teams of 4–5 students** (not more, not less). Please form your teams promptly and upload them on Gradescope after returning from Thanksgiving break (template provided by course staff). If you do not have a team, please tell us on Ed.

Your deliverable is in two parts: a basic simulation model (which should take about the same time as a long question in the assignments, plus the time to read this description and background reading), and the project report (which is where most of the time and effort should be involved).

- Your initial simulation model (code + demo) is due by **December 8th** (the last day of classes). You can demonstrate your code to us in person. We will upload a sign-up sheet with 15-minute time slots; all teams should enter at least 4 time-slots they can attend, and we will make the assignments. **All teams must have at least one member present at their demo slot.** If your team cannot attend any of the scheduled time slots, notify instructors by **December 5th** to arrange an alternative. We will also provide a basic model after December 5, in case teams have trouble with theirs
- For your final report, we encourage *reproducible science*. Present your report in a main Jupyter notebook, with supporting code in separate `.py` files and any generated data in `data/` directory. All imports should reference local files using relative paths. The instructor should be able to download your submission folder and run the notebook end-to-end without any setup.
- Your final reports are due on **December 17th at 11:59am ET**. Please submit a single folder (or .zip file) containing:
  - `main_report.ipynb`: Your complete analysis and findings (required).
  - `simulation.py` (or multiple supporting `.py` files): Code for your simulator.
  - `data/`: CSV or pickle files with results from your experiments.
  - `README.md`: Brief instructions on how to run the notebook and reproduce results.

  You may optionally include one additional PDF document (supplementary visualizations, etc.), but put all main content in the notebook.

3.1 Report Guidelines

Your project report should be made up of two sections: a *main findings section* describing your analysis in a way that is accessible to non-experts, and a *technical appendix* to allow other simulation specialists to repeat/test your analysis. The main part of the report should be designed for non-specialists. Your project report should not be very long. The main report (all text sections combined) should be **approximately 10–12 pages if printed** (at 12pt font, 1.5 spacing). This corresponds to roughly 2000–3000 words of text plus figures. Prioritize clarity: concise writing is better than padding. Technical appendices (code, detailed derivations, supplementary plots) are not included in this page count and can be as long as needed.

Below are possible sections you may want to consider in your project report. If you feel uncomfortable with any portion of the outline suggested below except the Executive Summary, then feel free to modify it:

- **Executive summary:** This is the part that decision-makers (engineering managers, platform leads, etc.) will read. Give a synopsis of what you did, what you discovered, and how the tools you developed can be used further. Think what you would want to know if you were a high-level manager and this was all that you read. This section should be short (at most two pages of text).

- **Modeling approach, assumptions, parameters:** Describe the LLM-serving simulation model in general terms. This section is for intelligent non-specialists, who want to ensure that your model is reasonable and captures all of the important aspects (prefill vs decode, batching, GPU utilization, SLOs). Describe what inputs are needed to drive your model and how you obtained them. You may defer technical parts to the appendices.
- **Model Details:** Describe the model in technical detail, in a way accessible to experts. Include iteration timeline diagrams and scheduler pseudo-code (see Notation Summary and Scheduling Strategies sections for examples of the level of detail expected). Explain in broad strokes how you checked that your model was correct as implemented (refer to validation tests in the appendix). Also outline the benchmark scheduling policies that you use to test your system.
- **Model analysis:** Describe how you used your model to study scheduling policies. For each analysis, clearly explain what you are testing, and then present the output (with confidence bounds, axis labels and legends). Analyze the results and make recommendations. Think about sensitivity analysis for quantities that are uncertain or approximate, and how robust your recommendations are to such changes.

  Steady-state considerations: Run your simulation long enough that statistics stabilize. As a rule of thumb, simulate at least 10,000 queries. Discard the first 1,000–2,000 queries as warm-up to reduce initialization bias. Document your warm-up period and replication strategy in the technical appendix.
- **Conclusions:** Briefly recap your findings and recommendations.
- **Technical Appendices:** Intended for readers who have taken ORIE 4580/5580. Separate model and implementation details; document functions/classes with docstrings and provide interactive elements (e.g., animations of batch timelines, GPU workloads, query queues, etc.). Include your validation tests and any supplementary plots.

3.2 General Recommendations

- An open-ended project like this rarely has completely specified deliverables, and often it is up to you to decide how much to do, and how to manage your time and effort. If you have doubts over any particular details of the problem, make assumptions about it, and use your report to outline and justify these assumptions. We will be available to help with this, but may choose not to specify all details.
- Please try to plan your project and design your code to be modular. Use classes/functions wherever possible to ensure that the code is easy to build on and extend. Also make sure the code is well commented and documented, as this will help a lot when working with teammates.
- Please remember that this is a team project – make sure you work well as a team to get things done. Good code commenting, dividing responsibilities, version control, planning meetings, etc. are particularly important here, since you are working separately.