Publicly Shared

Benchmarking LLM Workloads for Performance Evaluation and Autoscaling in Kubernetes

Author: Ashok Chandrasekar
Shared with dev@kubernetes.io for commenter
Shared with wg-serving@kubernetes.io for editing

Benchmarking Landscape

Model Server Benchmarks

There are lots of different tools to benchmark model servers and inference workloads available today. Each model server has their own set of tools like vLLM, Triton, TGI, and Jetstream. These are focussed on single model server benchmark which helps measure throughput and latency with a specific dataset and in most cases works well / only with their corresponding model server. They provide little in the way of traffic shaping for autoscaling.

Hosted Offering Benchmarks

There are also other benchmarking tools like LLMPerf which helps to benchmark hosted offerings like Vertex, Sagemaker, Anthropic, etc. These are meant to benchmark specific API endpoints and provide latency and throughput numbers so they can be compared against each other and a leaderboard can be provided among them. These offer less customizability and don't address traffic shaping needs of autoscaling.

Accelerator Benchmarks

There are hardware accelerator benchmarks like MLPerf which provides a loadgen and a way to benchmark and compare different hardware accelerators. It is more on the academic side and is useful for individual chip manufacturers and cloud providers to claim infrastructure performance. They are not suitable for Kubernetes and autoscaling benchmarking needs.

Generic Web Benchmarks

There are generic http / web server benchmarks like Locust, Apache Bench, etc. which help generate http traffic against the inference servers. These are the most useful when it comes to autoscaling. These tools help control the concurrency of requests and it can be increased or decreased as the test happens which allows you to introduce spikes which are good to benchmark autoscaling. A comparison of these can be seen below.

Tool	Language	License	RPS control	Distributed Load Testing	Containerization	Spike / Burst Traffic
Locust	Python	MIT	No	Yes	Yes	Manual
Apache Bench	C	Apache	Yes	No	No	Manual
Apache JMeter	Java	Apache	Yes	Yes	No	Manual
Gatling	Java / JS	Apache	-	Yes	No	Yes
K6	Go / JS	AGPL	No	Yes	Yes	Automated

Of the available tools, Locust is the most useful in terms of being written in python and having a good support ecosystem. But there are some missing features still which makes it a not so good choice.

1. GenAI features need to be implemented as extensible tasks which makes it harder to both implement because of the friction involved and scale because of how resource requirements are harder to figure out when these extensions are implemented.
2. The request rate cannot be controlled which is a key requirement for benchmarking any model server. Only the number of users to send requests from can be controlled which makes it not very useful for GenAI benchmarking, since each user can send a new request only after the existing one completes and since AI inference requests take in the order of seconds to minutes, this makes it hard to hit any specific request rate.
3. gRPC support is lacking.
4. There are multiple components to run including a locust master and one or more workers. So deploying and managing it is a little complicated which adds friction to someone who wants a benchmark they can easily run.
5. Load spikes need to be manually introduced. So, hands off benchmarking is harder to do.

Benchmarking Tooling for Kubernetes

Requirements

We need a benchmarking tool that we can run in a Kubernetes cluster for the following reasons.

1. With a specific model server and accelerator, how does the performance look?
    a. This is needed for model catalogs and any pre-baked configurations that we want to publish.
2. To identify, what is the saturation point for model server compute (the inflection point where we cannot get more throughput with a higher request rate, but start to incur additional latency) so we can autoscale effectively?
3. To measure the baseline for autoscaling performance so it is easier to compare different metrics and thresholds that we can use for autoscaling and pick the one that performs better.
4. To measure the performance of Instance Gateway and other future orchestration enhancements we might build in Kubernetes.

Other general requirements include the following:

1. To be able to repeatedly rerun the tool against vastly different inference stacks / infrastructure. This can include hosted offerings like Vertex, Sagemaker, etc. too in addition to different model servers running on Kubernetes.
2. To be able to extend the tool and to easily integrate with tokenizers and other client libraries in the GenAI ecosystem.

Options

To solve the above requirements, we have two options.

1. **Benchmark as Data:**
   In this model, the benchmarking tool will target ease of use so it can be easily deployed and performance data can be obtained. It would be useful especially for newer users and users who are focussed on getting the supported benchmarking data on their Kubernetes clusters. It is less extensible and any new support would need to be implemented in the tool first before it can be used.

2. **[Preferred] Benchmark as Code:**
   In this model, the benchmarking tool will target extensibility to support different use cases and it will be a tool that ML engineers can use for a variety of inference benchmarking needs. This will be useful to the model server teams and will have use outside of the Kubernetes ecosystem. Additional effort will be needed in making it easier to deploy and run on Kubernetes. But that is relatively easier to solve.

Considering the above options, we propose to go with the **benchmark as code** approach. With this approach, it is better to build a python based benchmarking tool. This makes it easier

to integrate and use client libraries in the GenAI ecosystem like tokenizers directly. It can be directly adopted by the model server implementers and GenAI developers who are already familiar and would prefer to work in the python ecosystem. We can use the vLLM benchmarking script as a starting point which we can then expand to support all the needed functionalities.

Design

This section describes the high level design for the tool. It includes the following components.

Dataset Preprocessor

Dataset Preprocessor takes in a known dataset like ShareGPT or OpenOrca as the input and pre-processes them by making sure the prompt length and generation length are aligned with the user input to support different options like fixed input / output length tests, variable length tests (larger input / smaller output and the vice versa). This allows us to support different GenAI use cases like chat completion, summarization, code completion, etc. depending on the dataset and the benchmarking user's inputs.

LoadGen

LoadGenerator is the component which generates different traffic patterns based on user input. This can include a fixed RPS test for a predetermined amount of time or include a way to generate bursts in traffic or other traffic patterns as desired for autoscaling and other use cases.

Request Processor

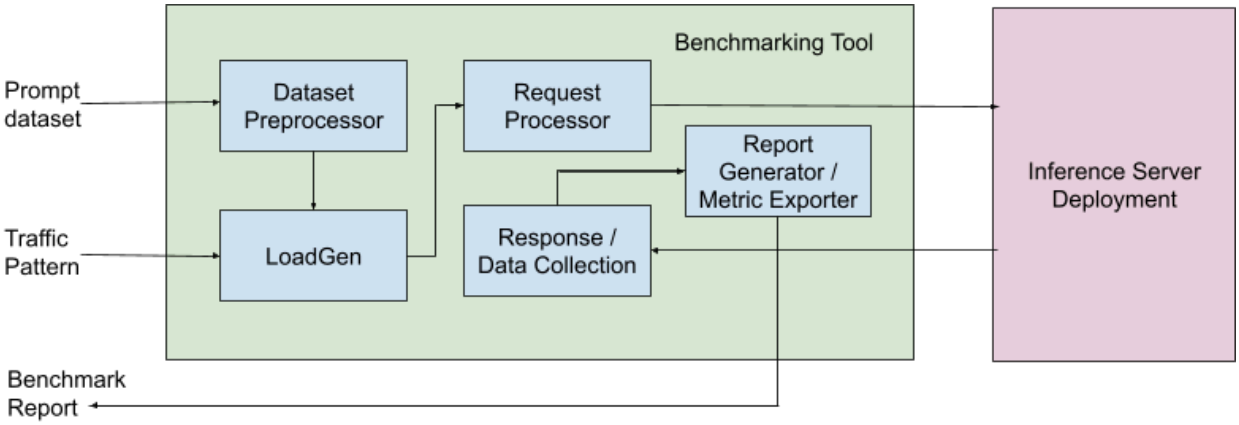
Request Processor provides a way to support different model servers and their corresponding request payload with different configurable parameters. This makes our tool model server agnostic and provides a generic way to benchmark different model servers and produce apples to apples comparison between them. This component will also support different protocols like http and grpc and options like request streaming which is important to produce time to first token (TTFT) metric.

Response Processor / Data Collector

Response Processor / Data Collector component allows us to process the response and measure the actual performance of the model server in terms of request latency, TPOT, TTFT and throughput.

Report Generator / Metrics Exporter

Report Generator / Metrics Exporter generates a report based on the data collected during benchmarking. It can also export the different metrics that we collected during benchmarking as metrics into Prometheus which can then be consumed by other monitoring or visualization solutions.



This addresses specific things we need for LLM and GenAI benchmarking in Kubernetes.

1. Support to use a prompt dataset as input.
2. Support for different model servers with their own API format and protocol.
3. Support to generate different loads / traffic patterns needed for autoscaling.
4. Support for a benchmarking report generation with the required LLM metrics to measure performance.

Benchmarking API for Kubernetes

To take this one step further, we can also build a benchmarking operator in Kubernetes which would make this easier to run in a continuous fashion in any Kubernetes cluster to benchmark any model server along with autoscaling and load balancing.

Something like:

```
{
  metadata: {
    name: llama3-405b-2024-07-01,
    namespace: llm,
  },
  spec: {
    endpoint: llm-1.svc.local,
    port: 8000,
    performance: {
      traffic-shape: {
        req-rate: 10 qps,
        model-type: instruction-tuned-llm/diffusion,
        dataset: share-gpt,
        input-length: 1024,
        max-output-length: 1024,
        total-prompts: 1000,
        traffic-spike: {
          burst: 10m,
```

```
          req-rate: 20 qps,
        }
      }
    }
  },
  status: {
    status: success,
    results: gcs-bucket-1/llama3-405b-2024-07-01,
  }
}
```