ETUDE – Evaluating the Inference Latency of Session-Based Recommendation Models at Scale

Barrie Kersbergen[1,2] Olivier Sprangers[2] Frank Kootte[1]
Shubha Guha[2] Maarten de Rijke[3] Sebastian Schelter[3,4]
[1]*bol.com* [2]*AIRLab, University of Amsterdam* [3]*University of Amsterdam* [4]*Ahold Delhaize*
bkersbergen@bol.com o.sprangers@uva.nl fkootte@bol.com
s.guha@uva.nl m.derijke@uva.nl s.schelter@uva.nl

***Abstract*—Session-based recommendation (SBR) targets a core scenario in e-Commerce: Given a sequence of interactions of a visitor with a selection of items, we want to recommend the next item(s) of interest to interact with. Unfortunately, SBR models are difficult to deploy in practice, as (i) session-based recommendations cannot be precomputed offline, but must be inferred online for ongoing user sessions with low latency, and (ii) there is a huge variety of SBR models available, typically designed by academic researchers, whose inference performance and deployment cost is unclear. As a result, data scientists must typically prototype and evaluate different deployment options in collaboration with devops teams – a tedious and costly process, which does not scale to multiple use cases.**

To alleviate this, we present ETUDE, an end-to-end benchmarking framework, which enables data scientists to automatically evaluate the inference performance of SBR models under different deployment options. With ETUDE, data scientists can declaratively specify workload statistics, hardware options, as well as latency and throughput constraints. Based on these, ETUDE automatically deploys and runs an inference benchmark in Kubernetes with a synthetically generated click workload. Subsequently, ETUDE provides the data scientists with measurements on the achieved throughput and latency, as a basis for deciding on feasible and cost-efficient deployment options.

We detail the design of ETUDE and present an experimental study for ten different SBR models in challenging settings resembling real-world workloads encountered at the large European e-Commerce platform bol.com. We determine performant and cost-efficient deployment options in terms of models and cloud instance types for a variety of online shopping use cases (ranging from grocery shopping to large e-Commerce platforms). Moreover, we identify severe performance bottlenecks in the open source TorchServe inference server from the PyTorch ecosystem and in the implementation of four SBR models from the open source RecBole library. We make the source code of our framework and experimental results publicly available.

I. INTRODUCTION

Session-based recommendation (SBR) targets a core scenario in e-Commerce. Given a sequence of interactions of a visitor with a selection of items, we want to recommend the next item(s) of interest to interact with. [1]–[11]. This machine learning problem is crucial for large e-Commerce platforms which offer millions of items such as bol.com [12], [13].

Challenges in deploying session-based recommendation systems. Scaling session-based recommender systems is a difficult undertaking, because the input space (sequences of item interactions) for the recommender system is exponentially large, which renders it impractical to precompute recommendations offline and serve them from a data store. Instead, session-based recommenders have to interactively react to changes in the ongoing user sessions, and compute next item recommendations with low latency [12], [14]. At bol.com for example, a SBR model has to handle at least 1,000 requests per second with a 90-th percentile latency of at most 50 milliseconds [12], [13]. Furthermore, deploying an SBR system often involves choosing from a large number of models [1]–[10], implemented in academic libraries like RecBole [15], which lack support for model deployment [16] and inference optimisations [17].

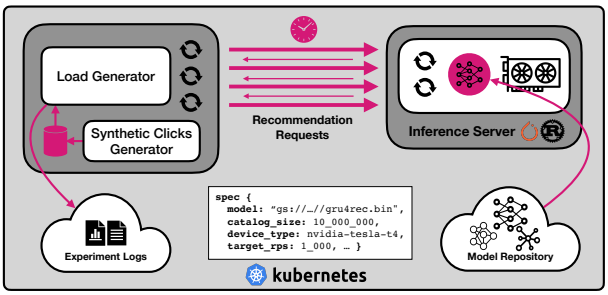

Fig. 1: High-level overview of ETUDE: Data scientists can automatically evaluate the inference performance of SBR models under declaratively specified deployment options in terms of hardware, workload statistics, as well as latency and throughput constraints.

For these reasons, it is currently very difficult for companies to deploy SBR models in challenging production scenarios. Data scientists typically have to prototype different deployment options in collaboration with devops teams, in order to manually evaluate a model's inference performance on a per use-case basis. This is often a tedious process lasting several weeks, which eats up the time of hard-to-hire to experts. Furthermore, this process may have to be repeated multiple times in retail corporations such as Ahold Delhaize[1], which have different brands and platforms in different countries, with varying product catalogs and visitor numbers.

[1]Ahold Delhaize is an international retail group composed of nineteen companies located across the United States, Europe and Indonesia.

Existing benchmarks are not designed for such industry scenarios unfortunately. The Session-Rec [18] benchmark for example measures prediction quality only, with Python-based single-threaded evaluation, and can therefore not assess real-world inference performance. Systems-focused benchmarks like MLPerf [19] are designed to evaluate the performance of inference systems, but not to help data scientists with the choice of models and deployment options in an end-to-end cloud setup.

The ETUDE benchmarking framework. In order to address the issues outlined above, we present the ETUDE benchmarking framework (Section II). ETUDE allows data scientists to automatically evaluate the inference performance of SBR models under different deployment options. Data scientists provide a set of trained SBR models and declaratively specify statistics of the underlying product catalog, hardware options for deployment (e.g., the type of GPU to use), together with latency and throughput constraints. Based on this, ETUDE automatically deploys and runs an inference benchmark in Kubernetes with a synthetically generated click workload, and provides the data scientists with measurements on throughput and latency, as a basis for deciding on feasible and cost-efficient deployment options.

We discuss the design of ETUDE in Section II, including its synthetic workload generator, its backpressure-aware load generator and our Rust-based inference server. We showcase ETUDE in Section III, where we evaluate ten recently proposed neural SBR models with different deployment options (CPU/GPU inference, small/large product catalogs, just-in-time-optimisation of the underlying models) in challenging settings resembling real-world workloads encountered at bol.com.

Contributions. Our contributions are as follows:

- We detail the design of our automated end-to-end SBR model benchmarking framework ETUDE (Section II).
- We present an experimental study for ten different SBR models in challenging settings resembling real-world workloads encountered at bol.com. We determine performant and cost-efficient deployment options in terms of models and cloud instance types for a variety of online shopping use cases (ranging from grocery shopping to large e-Commerce platforms) in Section III.
- We identify severe performance bottlenecks in the open source TorchServe inference server from the PyTorch ecosystem and in the implementations of four SBR models from the open source RecBole library (Section III).
- We make the source code of ETUDE and our experimental results available at https://github.com/bkersbergen/etudelib.

II. THE ETUDE BENCHMARKING FRAMEWORK

Design goals. We design ETUDE to address the previously outlined challenges. We assume that a data scientist (who is typically an ML expert, but not an expert in systems or devops) wants to assess whether their SBR model can be deployed in production, and what setup (in terms of the number and type of machines) is required to adhere to the latency and throughput constraints of their company. As illustrated in Figure 1, ETUDE helps the data scientist to automate and accelerate this process. Our framework enables fully automated benchmarking, where the data scientist only needs to provide their model and declaratively specifies statistics of the underlying product catalog, hardware options for deployment (e.g., the type of GPU to use), together with latency and throughput constraints. Thereby, ETUDE reduces time-to-deployment and experimentation cost for industry practitioners. Note that ETUDE only measures the inference performance of a model, not its prediction quality; we assume that the data scientist already evaluates the prediction quality during the model design phase.

Supported models. In general, ETUDE is compatible with any SBR model implemented in PyTorch [20]. For this paper, we discuss ten different neural SBR methods, as implemented in the open source RecBole library [15]. These include two recursive neural network-based methods: GRU4Rec [1], which utilises GRU neural networks with gating mechanisms for long-term dependencies in item interactions, and Repeat-Net [2], which employs an encoder-decoder architecture with a repeat-explore mechanism. Furthermore, we include two graph neural network-based methods: GC-SAN [3], which uses graph contextualised self-attention for session representation and SR-GNN [4], which combines graph models to predict user actions based on long-term preferences and current interests. We use three attention-based methods: NARM [5], which employs a hybrid encoder with attention to model sequential behavior, SINE [6], which introduces sparse-interest embeddings for session recommendations, and STAMP [7], which captures short-term attention and memory using gated self-attention. Finally, we also leverage three transformer-based methods: LightSANs [9], which uses transformers on session item embeddings for sequential recommendations, CORE [8], which ensures consistent session representations via weighted sum item embeddings, and SASRec [10], a self-attention-based recommendation model assigning weights to previous items in sessions.

Time complexities for inference. We derive the asymptotic time complexities for inference with the models. These complexities depend on common hyperparameters, such as the catalog size C and the number of items to recommend k. Additionally, they are influenced by model-specific hyperparameters, such as the hidden size for recursive neural networks or the embedding dimension of transformers and graph neural networks, to which we collectively refer as d. In a typical SBR scenario with a relatively short session length, the asymptotic complexity for inference with all models is $O\left(C(d + \log k)\right)$, despite the different neural architectures. This is because all models conduct a maximum inner product search for the top-k similar items in the d-dimensional learned vector representations of all C items within the catalog. The embedding dimension d is typically chosen heuristically based on the value of C and k is set to a small value, which means that the inference time is dominated by the catalog size C across all models.

Synthetic session generation. A design goal of ETUDE is to enable load testing and benchmarking without having to replay sensitive real-world click data. Therefore, we run experiments with synthetic sessions, which preserve key statistical properties of the underlying workload. Users only have to provide two statistics: the exponent α_l of a power law distribution fitted to the distribution of session lengths in the click log, and the exponent α_c of a powerlaw distribution fitted to the distribution of click counts. These statistics can be estimated once from a real click log and reused for experiments later.

We detail how to generate N synthetic clicks for a catalog with C items based on these exponents in Algorithm 1. For each synthetic session, we first sample a length l from a power law distribution with exponent α_l (Line 10), and subsequently select l items via inverse transform sampling (Line 14) from the empirical cumulative distribution (CDF) of C click counts generated upfront by sampling from a power law distribution with exponent α_c (Line 7). This algorithm is fast enough for online generation (our implementation is able to generate over one million clicks per second on a single core for a catalog size C of ten million items).

Algorithm 1 Synthetic workload generation from the marginal statistics of a real clicklog.

```
1  function GENERATE_SYNTHETIC_SESSIONS(C, N, α_l, α_c)
2    Input: catalog size C, number of clicks N, exponents α_l and α_c for the
3           distribution of session lengths and click counts.
4    Output: Synthetic sessions Q.
5    Q ← ∅
6    n, s, t ← 0
7    I ← sample C click counts from power law dist. with exponent α_c
8    while n < N :
9      s ← s + 1                                  // increment session identifier
10     l ← sample session length from power law dist. with exponent α_l
11     n ← n + l                                  // increment number of clicks generated
12     for 0 to l :                               // Generate l clicks
13       t ← t + 1
14       i ← sample item id from the empirical CDF of I     // Choose item
15       Q ← Q ∪ (s, i, t)                        // Add synthetic click on item i in session s
     return Q
```

Load generator. Our goal is to measure the latency of a deployed model for a given target throughput (in terms of requests per second). However, ETUDE should still provide insightful results in situations where the model cannot handle the desired throughput (and for example times out the majority of requests). Therefore, we design a custom load generator for ETUDE, which slowly ramps up the load to a specified target throughput while keeping track of backpressure. It will refrain from sending more work once too much backpressure is built up, which allows us to gracefully shutdown experiments in such cases and determine the throughput threshold where a model fails to handle the load.

As detailed in Algorithm 2, the load generator ramps up the load to a target throughput r over the timespan d, while replaying the synthetic sessions Q. The load generator operates in "ticks" of one second and keeps track of the current number of pending requests. The main loop to handle a single tick starts in Line 3. In each tick, the current number r_c of requests

Algorithm 2 Backpressure-aware load generator, which replays the synthetic sessions Q for a target throughput r with a ramp-up over the duration d.

```
1  function GENERATE_LOAD(r, d, Q)
2    t, p ← 0          // Tick counter and atomic counter for pending requests
3    for r_c ← TIMEPROP_RAMPUP(r, d) :                    // Main tick loop
4      terminate in case deadline d reached
5      t ← t + 1
6      for i ← 0 ... r_c :                                // Request generation loop
7        terminate in case deadline d reached
8        while p ≥ r_c :                                  // Backpressure handling
9          if no time left for current tick t
10           go to next tick t + 1
11         wait 1 millisecond
12       if no time left for current tick t
13         go to next tick t + 1
14       SCHEDULE_REQUEST_ASYNC(p, Q)
15       d_t ← milliseconds till next tick
16       wait for d_t/(r_c − i) milliseconds       // Evenly spread out requests
17     wait until next tick t + 1
```

to send per second is ramped up by the TIMEPROP_RAMPUP function, proportionally to the time spent with respect to the desired benchmark duration d, so that we reach the target throughput r eventually. Requests replay clicks from the synthetic session log Q, and are sent asynchronously (Line 14) and dynamically spread out over the duration of a tick (Line 16). During the request generation loop, the count of currently pending requests p is used to handle backpressure: if this count reaches the current throughput target ($p \geq r_c$), the generator pauses for a millisecond to wait for the load to be handled by the server (Lines 9 and 12). Note that p is decreased when responses are received asynchronously (not shown in the pseudo code).

We implement Algorithm 2 in Java, using the asynchronous HTTP client from Apache HttpComponents 5.2.1 and integrate our synthetic clicklog generation. Our implementation additionally ensures that the load generator respects the order of the sessions, e.g., it will only send the next interaction for a session if a response for the previous interaction was received.

Inference server. We focus on efficiently serving PyTorch [20] models in ETUDE, which is the implementation framework of choice for the vast majority of state-of-the-art SBR models [15]. We spend several weeks evaluating the open source inference server TorchServe [21] for PyTorch models, which unfortunately fails to satisfy our latency and performance requirements. We experience severe performance issues with TorchServe, which we attribute to the overhead of using several Python processes, orchestrated by a Java frontend. We experimentally validate this finding in Section III-A, where we showcase that TorchServe fails to handle 1,000 requests/second efficiently even if no model inference is performed.

As a consequence, we implement our own light-weight inference server for ETUDE in Rust, based on Actix [22], a high-performance web server leveraging non-blocking IO, the Rust bindings [23] for the C++ API of PyTorch and a plugin enabling request batching for GPU inference [24].

Our inference server can deploy serialised PyTorch models from Google storage buckets and serve them with CPU or GPU inference. Furthermore, it allows users to configure the number of worker threads and details of the request batching. As validated in Section III-A, the latency overhead of this inference server is extremely low.

Benchmark execution. We detail how to concretely execute benchmark experiments with ETUDE. We automate the cloud infrastructure management via a `make infra` command, which provisions and configures essential components such as a Kubernetes cluster, Google Storage and the addition of service accounts required for deployments. Importantly, this setup is a one-time operation, which can be reused for multiple experiments.

Experiment deployment and execution. The definition and execution of a single experiment proceeds as follows. ETUDE users declaratively specify the model(s) to deploy and the type of hardware to use. Furthermore, they specify the catalog size C, the statistics for click generation and the target throughput to which the load generator should ramp up. Subsequently, the execution is triggered via the command `make run_deployed_benchmark`. ETUDE will then deploy the model onto a dedicated machine in Kubernetes. Once the model deployment is finished (determined via Kubernetes's readiness probes), a ClusterIP service interface is deployed for allowing access to the serving machine. Next, the load generator is deployed on an another machine, from which it sends the corresponding recommendations request for recommendations to the model inference server via the service interface. The load generator measures the end-to-end response latencies for its recommendation requests and the inference server additionally communicates metrics like the inference duration via HTTP response headers. The observed measurements are written to a Google storage bucket upon termination of the experiment.

III. EXPERIMENTAL STUDY

We validate our design decisions and showcase how ETUDE can be used to determine performant and cost-efficient deployment options for a variety of e-Commerce scenarios.

If not declared otherwise, we use the following settings. We run our experiments in the Google Cloud Platform (GCP) via the Google Kubernetes Engine v1.27.3-gke.100, operating in Autopilot mode with Google Cloud SDK 442.0.0. We leverage general purpose `e2` instances [25] with 5.5 vCPUs from an Intel Xeon CPU@2.20GHz and 32 GB RAM. For the GPU experiments, we either use an NVidia-Tesla-T4 with 16GB RAM attached to an `e2` instance or a preconfigured instance with an NVidia-Tesla-A100 with 40GB GPU memory, 12 vCPUs and 85GB of RAM.

We choose the embedding dimensions of the models via the common heuristic of rounding up the fourth root of the catalog size C [26] (which is in line with the original embedding sizes used in the corresponding research papers), and randomly initialise the weights of the SBR models (which need not be trained in order to measure inference performance). For synthetic session generation, we leverage marginal statistics from a real bol.com click log. On our inference server, we apply request batching for GPUs for up to 1,024 requests, and empty the underlying buffer every two milliseconds. We make our code and experimental results available at https://github.com/bkersbergen/etudelib/blob/main/experiments.md.

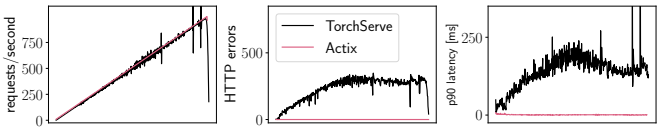

Fig. 2: Infrastructure test for answering 1,000 requests/s without model inference. TorchServe already fails at handling "empty" requests efficiently.

A. Validation of Design Choices

First, we validate our design choice of leveraging an Actix-based Rust runtime instead of the open source TorchServe project as serving engine.

Experimental setup. In order to measure the serving performance of TorchServe independent of the model inference overhead, we deploy TorchServe on a 2 vCPU `e2` machine with 2GB of memory, and implement a Python model that returns a empty response and does not conduct any computation. Next, we configure our load generator to ramp up to 1,000 requests per second over the duration of ten minutes, and measure the response latencies. We deploy our Actix-based inference server analogously and also make it return a static answer.

Results and discussion. We plot the results of our experiment in Figure 2. The load ramps up to 1,000 requests per second over 10 minutes, and we observe early on that TorchServe cannot keep up with the load and starts to return a large number of HTTP errors (due to reaching the internal timeout of 100ms). It handles the remaining requests with a p90 latency between 100 and 200ms. Our Actix-based inference server easily handles the load with a p90 latency of around one millisecond for serving the static content and does not throw any HTTP errors. These results are a strong indication that TorchServe's design causes severe latency overheads and that it is not suitable for low-latency, high throughput use cases like session-based recommendation. This insight is further supported by the documentation on benchmarking and tuning TorchServe, which only uses workloads with a small number of requests (1,000 in total) and low concurrency (10 requests at the same time) [27], [28].

We also run a validation experiment for the synthetic click generation, where we compare the latency measurements achieved by replaying a real click log from bol.com to the measurements achieved when using a synthetic workload generated based on statistics from the real click log. We find that the achieved latencies resemble each other closely.

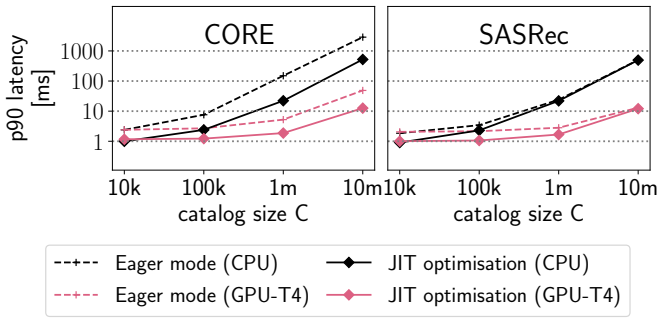

Fig. 3: Microbenchmark results confirming the dependency of the inference latency on the catalog size and the benefits of applying PyTorch's JIT optimisations.

B. Micro-Benchmark

We run a micro-benchmark to confirm our theoretical insight about the impact of the catalog size on prediction latency, the potential of accelerator hardware and the benefits of PyTorch's just-in-time (JIT) optimisation [29].

Experimental setup. We conduct our micro-benchmark on a single machine with synthetic click data generated according to the marginal statistics of session lengths from bol.com and varying catalog sizes. We send recommendation requests in a serial manner (one request after another, waiting for model responses), measure the prediction time and report the p90 latency. We repeat the experiment with two different instances (CPU and GPU-T4), different execution types (eager execution without optimisation and JIT-optimised versions of the models) and catalog sizes of 10,000, 100,000, 1,000,000, and 10,000,000 distinct items.

Results and discussion. We observed similar results for all models and plot a selection of the resulting prediction latencies in relation to the catalog sizes on a logarithmic scale in Figure 3. The results confirm our theoretical analysis from Section II about the strong dependence of the runtime on the catalog size, as we observe a linear scalability of the prediction latency with the catalog size. We see clear benefits of using GPUs for medium to large catalog sizes: starting from catalogs with one million items, the prediction latency of the GPU is more than an order of magnitude lower than the latencies achieved with CPUs only (and the CPU already requires more than 50ms per prediction for catalogs with one million items). Interestingly, this relation does not hold for small catalogs with 10,000 items, in six out of ten cases, the CPU latency is on par with or lower than the GPU latency here. Furthermore, we find that JIT-optimisation is always beneficial and never hurts performance. We identify an issue with the LightSANs model implementation though, which cannot be JIT-optimised by PyTorch due to dynamic code paths.

C. End-to-End Benchmark

Our goal for the final experiment is to showcase how ETUDE can identify well performing models and cost-efficient deployment setups for a variety of e-Commerce use cases.

Experimental setup. We define five end-to-end use case scenarios, detailed in the first three columns of Table I: *Grocery shopping (small)*, *Grocery shopping (large)*, *Fashion*, *e-Commerce*, and *Platform* with catalog sizes from 10,000 up to 20,000,000 items and a target throughput ranging from 100 to 1,000 requests per second. These scenarios are inspired by experiences from our various brands and use cases at Ahold Delhaize, and are in line with publicly reported catalog sizes [30]–[32].

We conduct an end-to-end benchmark for the JIT-optimised variants of all ten models in all scenarios with three different instance types (CPU, GPU-T4, and GPU-A100). We ramp up the load to 1,000 requests per second over a period of ten minutes and measure the response latency. We execute each configuration three times and ignore the runs with the lowest and highest latencies, amounting to around four hundred runs.

Results and discussion. We plot detailed results for a selection of scenarios in Figure 4 and discuss various aspects of our findings.

Issues with selected SBR models. We encounter serious issues with three additional SBR model implementations from the RecBole library [15]: SR-GNN, GC-SAN, RepeatNet are not able to handle most of our use cases (or only handle them with unacceptably low performance). We inspect their implementations to determine the root causes for this finding. The RepeatNet model contains expensive tensor multiplications of very sparse matrices which are implemented with dense operations and representations (and therefore incur high overheads), and the SR-GNN and GC-SAN models contain NumPy operations in their inference functions which require repeated data transfers between CPU and GPU at inference time. We filed bug reports for these issues with the RecBole project.

Impact of catalog size and accelerator hardware. We observe that catalog sizes of 10,000 and 100,000 can be handled well with CPU instances only, where most models achieve more than 500 requests per second at a 50ms p90 latency. The situation changes for catalogs with one million items, where the performance of CPU instances drops to around 200 milliseconds. At the same time, we see that this setup is easily handled by instances with GPUs, where the T4 card already handles more than 700 requests per second at a 50ms p90 latency. Only GPU instances are able to handle the load for catalogs with 10 million items, and for the platform setting with 20 million items, the high-end A100 cards are required.

Cost-efficient deployment options. The most performant setup may not necessarily be the most cost-efficient one. The monthly costs (given a one year commitment) for different machines vary [33], a CPU instance for example costs $108.09 in GCP, an instance with an additional T4 GPU costs $268.09 per month and the instance with the A100 GPU has a hefty price tag of $2,008.80. There may be cases where it is more beneficial to linearly scale out the recommender system with cheaper hardware than to use a high-end device.

Scenarios			Deployment Options			SBR Models					
Use case	Catalog size	Throughput	Instance type	Amount	Cost/month	CORE	GRU4Rec	NARM	SASRec	SINE	STAMP
Groceries (small)	10,000	100 req/s	**CPU**	**1**	**$108**	✓	✓	✓	✓	✓	✓
			GPU-T4	1	$268	✓	✓	✓	✓	✓	✓
Groceries (large)	100,000	250 req/s	**CPU**	**1**	**$108**	✓	✓	✓	✓	✓	✓
			GPU-T4	1	$268	✓	✓	✓	✓	✓	✓
Fashion	1,000,000	500 req/s	CPU	3	$324				✓		✓
			GPU-T4	**1**	**$268**	✓	✓	✓	✓	✓	✓
			GPU-A100	1	$2,008	✓	✓	✓	✓	✓	✓
e-Commerce	10,000,000	1,000 req/s	**GPU-T4**	**5**	**$1,343**		✓	✓	✓	✓	✓
			GPU-A100	2	$4,017	✓	✓	✓	✓	✓	✓
Platform	20,000,000	1,000 req/s	**GPU-A100**	**3**	**$6,026**		✓	✓		✓	✓

TABLE I: Cost-efficient deployment options for SBR models in varying e-Commerce scenarios with costs per month, derived from ETUDE measurements. Boldface indicates the most cost-efficient deployment option for a scenario. Empty cells for models indicate that they are not able to handle the target throughput with the given deployment option.

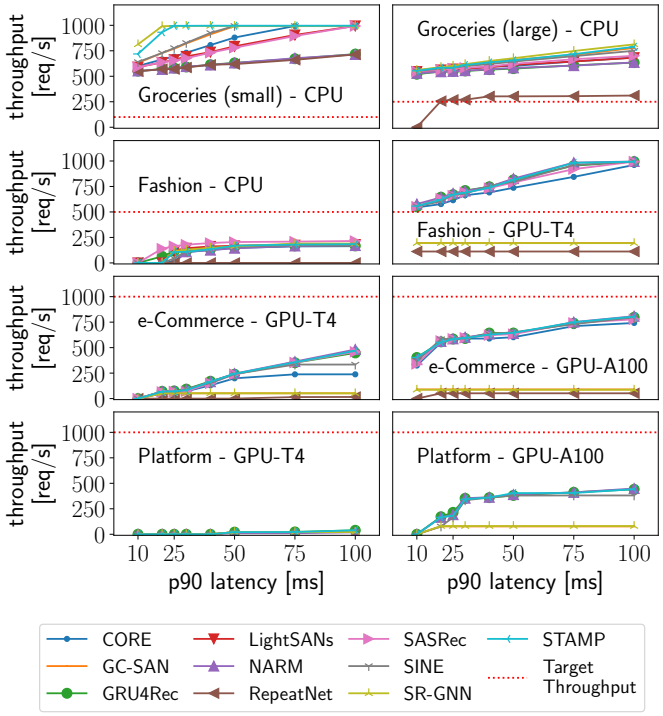

Fig. 4: Observed latency and throughput of different SBR models in deployment scenarios with varying instances types.

Such cost decisions can be made based on ETUDE's experimental results: Table I lists the monthly costs for the best models per scenario for different setups. Note that we applied a latency threshold of 50 milliseconds in the 90th quantile and ignored the four models for which we found implementation errors. We find that (i) both grocery shopping scenarios can be handled very cost-efficiently with a single CPU machine for $108 per month (for all models); (ii) the specialised e-commerce scenario can be handled with a single GPU-T4 instance (for all models) for $268 per month, and two models (SASRec and STAMP) are also comparatively cheap to deploy with only CPUs (at a cost of $324 per month); (iii) the general e-Commerce and platform scenario require GPUs: the platform scenario with a large catalog of 20 million items can only be efficiently handled with three high-end GPU-A100 instances at the high cost of $6,026 per month. Interestingly, for the general e-Commerce scenario, it is significantly cheaper to deploy five GPU-T4 instances ($1,343) than to leverage two more powerful GPU-A100 instances (for $4,017).

IV. CONCLUSION

We presented ETUDE, a framework to automatically evaluate the inference performance of SBR models under declaratively specified deployment options in terms of hardware, workload statistics and latency and throughput constraints.

In the past, we have seen recommendation teams refrain from building online SBR systems due to the outlined serving challenges. As a consequence, they designed static recommender systems with precomputed recommendations for the last item of a session only, which often exhibit low prediction quality due to the missing session context. ETUDE is currently helping such teams to reduce risk, as they can test newly designed models early on challenging workloads, and to improve their model implementations by identifying bugs which impact performance.

In the future, we plan to extend ETUDE with more inference runtimes such as ONNX [34] or TensorRT [35] and to support additional cloud environments such as Microsoft Azure or Amazon Web Services. Furthermore, we will explore the incorporation of techniques to trade-off prediction quality with inference latency, such as model quantisation [36] or approximate nearest neighbor search [37], as well as the automatic choice of appropriate instance types for declaratively specified workloads. Finally, our findings also indicate that there is a need to design custom neural models for high cardinality catalogs. This indicated by the enourmous costs for deploying models on catalogs with twenty million items, which can be handled much cheaper with non-neural approaches [13].

REFERENCES

[1] Y. K. Tan, X. Xu, and Y. Liu, "Improved recurrent neural networks for session-based recommendations," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 17–22.

[2] P. Ren, Z. Chen, J. Li, Z. Ren, J. Ma, and M. de Rijke, "Repeatnet: A repeat aware neural recommendation machine for session-based recommendation," in *AAAI*, vol. 33, no. 01, 2019, pp. 4806–4813.

[3] C. Xu, P. Zhao, Y. Liu, V. S. Sheng, J. Xu, F. Zhuang, J. Fang, and X. Zhou, "Graph contextualized self-attention network for session-based recommendation." in *IJCAI*, vol. 19, 2019, pp. 3940–3946.

[4] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan, "Session-based recommendation with graph neural networks," in *AAAI*, vol. 33, no. 01, 2019, pp. 346–353.

[5] J. Li, P. Ren, Z. Chen, Z. Ren, T. Lian, and J. Ma, "Neural attentive session-based recommendation," in *CIKM*, 2017, pp. 1419–1428.

[6] Q. Tan, J. Zhang, J. Yao, N. Liu, J. Zhou, H. Yang, and X. Hu, "Sparse-interest network for sequential recommendation," in *WSDM*, 2021, pp. 598–606.

[7] Q. Liu, Y. Zeng, R. Mokhosi, and H. Zhang, "Stamp: short-term attention/memory priority model for session-based recommendation," in *KDD*, 2018, pp. 1831–1839.

[8] Y. Hou, B. Hu, Z. Zhang, and W. X. Zhao, "Core: Simple and effective session-based recommendation within consistent representation space," *SIGIR*, 2022.

[9] X. Fan, Z. Liu, J. Lian, W. X. Zhao, X. Xie, and J.-R. Wen, "Lighter and better: low-rank decomposed self-attention networks for next-item recommendation," in *SIGIR*, 2021, pp. 1733–1737.

[10] W.-C. Kang and J. McAuley, "Self-attentive sequential recommendation," in *ICDM*. IEEE, 2018, pp. 197–206.

[11] M. Ludewig, N. Mauro, S. Latifi, and D. Jannach, "Performance comparison of neural and non-neural approaches to session-based recommendation," in *RecSys*, 2019, pp. 462–466.

[12] B. Kersbergen and S. Schelter, "Learnings from a retail recommendation system on billions of interactions at bol.com," *ICDE*, 2021.

[13] B. Kersbergen, O. Sprangers, and S. Schelter, "Serenade-low-latency session-based recommendation in e-commerce at scale." SIGMOD, 2022.

[14] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," *SIGIR*, pp. 103–112, 2014.

[15] W. X. Zhao, Y. Hou, X. Pan, C. Yang, Z. Zhang, Z. Lin, J. Zhang, S. Bian, J. Tang, W. Sun *et al.*, "RecBole 2.0: Towards a More Up-to-Date Recommendation Library," in *CIKM*, 2022, pp. 4722–4726.

[16] "How to do online serving · issue #1816. — github.com," https://github.com/RUCAIBox/RecBole/issues/1816, [Accessed 16-10-2023].

[17] "Questions about optimization and efficiency · issue #1855. — github.com," https://github.com/RUCAIBox/RecBole/issues/1855, [Accessed 16-10-2023].

[18] M. Ludewig, N. Mauro, S. Latifi, and D. Jannach, "Empirical analysis of session-based recommendation algorithms: A comparison of neural and non-neural approaches," *User Modeling and User-Adapted Interaction*, vol. 31, pp. 149–181, 2021.

[19] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MlPerf Inference Benchmark," in *ISCA*. IEEE, 2020, pp. 446–459.

[20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[21] TorchServe, "TorchServe - A performant, flexible and easy to use tool for serving PyTorch models in production." 2023. [Online]. Available: https://pytorch.org/serve/

[22] Actix, "Actix Web - a powerful, pragmatic, and extremely fast web framework for Rust," 2023. [Online]. Available: https://actix.rs

[23] tch-rs, "Rust bindings for the C++ api of PyTorch." 2023. [Online]. Available: https://github.com/LaurentMazare/tch-rs

[24] P. Walsh, "batched-fn - a Rust server plugin for deploying deep learning models with batched prediction," 2023. [Online]. Available: https://github.com/epwalsh/batched-fn

[25] Google, "General-purpose machine family for Compute Engine," 2023. [Online]. Available: https://cloud.google.com/compute/docs/general-purpose-machines#e2_machine_types

[26] "Introducing TensorFlow Feature Columns — developers.googleblog.com," https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html, [Accessed 28-09-2023].

[27] "Torchserve benchmark results." [Online]. Available: https://github.com/pytorch/serve/tree/master/benchmarks#installation-1/

[28] H. Shojanazeri, "Torchserve performance tuning, animated drawings case-study." [Online]. Available: https://pytorch.org/blog/torchserve-performance-tuning/

[29] PyTorch, "JIT Optimisation," 2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.jit.optimize_for_inference.html

[30] Albert Heijn, "Van land tot klant: onze ketens," 2023. [Online]. Available: https://www.ah.nl/over-ah/duurzaamheid/onze-ketens

[31] Zalando, "Extending the life of fashion," 2023. [Online]. Available: https://corporate.zalando.com/en/our-impact/extending-life-fashion

[32] Bol.com, "Facts and figures about bol.com," 2023. [Online]. Available: https://pers.bol.com/en/facts-figures/

[33] Google, "Compute Engine Pricing," 2023. [Online]. Available: https://cloud.google.com/compute/all-pricing

[34] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," 2019. [Online]. Available: https://github.com/onnx/onnx

[35] NVIDIA, "Tensorrt, an SDK for high-performance deep learning inference," 2023. [Online]. Available: https://developer.nvidia.com/tensorrt

[36] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[37] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.