AutoSys: The Design and Operation of Learning-Augmented Systems

Chieh-Jan Mike Liang, Hui Xue, Mao Yang, and Lidong Zhou, *Microsoft Research;* Lifei Zhu, *Peking University and Microsoft Research;* Zhao Lucis Li and Zibo Wang, *University of Science and Technology of China and Microsoft Research;* Qi Chen and Quanlu Zhang, *Microsoft Research;* Chuanjie Liu, *Microsoft Bing Platform;* Wenjun Dai, *Microsoft Bing Ads*

https://www.usenix.org/conference/atc20/presentation/liang-mike

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

AutoSys: The Design and Operation of Learning-Augmented Systems

Chieh-Jan Mike Liang[‡] Hui Xue[‡] Mao Yang[‡] Lidong Zhou[‡] Lifei Zhu[*‡] Zhao Lucis Li[⋆‡]
Zibo Wang[⋆‡] Qi Chen[‡] Quanlu Zhang[‡] Chuanjie Liu[○] Wenjun Dai[†]
[‡]*Microsoft Research* [*]*Peking University* [⋆]*USTC* [○]*Microsoft Bing Platform* [†]*Microsoft Bing Ads*

Abstract

Although machine learning (ML) and deep learning (DL) provide new possibilities into optimizing system design and performance, taking advantage of this paradigm shift requires more than implementing existing ML/DL algorithms. This paper reports our years of experience in designing and operating several production learning-augmented systems at Microsoft. AutoSys is a framework that unifies the development process, and it addresses common design considerations including ad-hoc and nondeterministic jobs, learning-induced system failures, and programming extensibility. Furthermore, this paper demonstrates the benefits of adopting AutoSys with measurements from one production system, Web Search. Finally, we share long-term lessons stemmed from unforeseen implications that have surfaced over the years of operating learning-augmented systems.

1 Introduction

Learning-augmented systems represent an emerging paradigm shift in how the industry designs modern systems in production today [33]. They refer to systems whose design methodology or control logic is at the intersection of traditional heuristics and machine learning. Due to the interdisciplinary nature, learning-augmented systems have long been widely considered difficult to build and require a team of engineers and data scientists to operationalize. To this end, this paper reports our years of experience in designing and operating learning-augmented systems in production at Microsoft.

The need of learning-augmented system design stems from the fact that heterogeneous and complex decision-makings run through each stage of the modern system lifecycle. These decisions govern how systems handle workloads to satisfy user requirements under a particular runtime environment. Examples include in-memory cache eviction policy, query plan formulation in databases, routing decisions by networking infrastructure, job scheduling for data processing clusters, document ranking in search engines, and so on.

Most of these decision-makings have been solved with explicit rules or heuristics based on human experience and comprehension. However, while heuristics perform well in general, they can be suboptimal as modern systems evolve. First, since many heuristics were designed at the time when computation and memory resources were relatively constrained, their optimality was often traded for execution cost. Second, since heuristics are typically designed for some presumably general cases, hardware/software changes and workload dynamics can break their intended usage or assumptions. Third, many modern systems have grown in complexity and scale beyond what humans can design heuristics for.

Recent advances in machine learning (ML) and deep learning (DL) have driven a shift in system design paradigm. Various efforts [6, 7, 16, 28, 34, 36, 48] have found success in formulating certain system decision-makings into ML/DL predictive tasks. Conceptually, from past benchmarks, ML/DL techniques can learn factors that impact the system behavior. For example, Cortez et al. [16] reported an 81% accuracy in predicting average VM CPU utilization, which translates to $\sim 6\times$ more opportunities for server oversubscription; Alipourfard et al. [7] reported near-optimal cloud configurations being predicted for running analytical jobs on Amazon EC2.

AutoSys is a framework that unifies the development process of several learning-augmented systems at Microsoft. AutoSys has driven decision-makings with ML/DL techniques, for several critical performance optimization scenarios. These scenarios range from web search engine, advertisement delivery infrastructure, content delivery network, to voice-over-IP client. Not only do these scenarios allow us to gain insights into the learning-augmented design, but they also reveal common design considerations that AutoSys should address.

Contributions. This paper makes the following key contributions, through reporting our years of experience in designing

This work was done when Lifei Zhu, Zhao Lucis Li, and Zibo Wang were interns at Microsoft Research.

and operating learning-augmented systems.

First, Section 2 analyzes the need for adopting the learning-augmented design, with concrete observations from modern systems in production. Due to its architectural similarities to most modern systems, this paper uses web search infrastructure (*Web Search*) as the target system scenario for performance optimization. We characterize sources of system complexity and operation complexity in modern systems, to contribute an understanding of the emergence of learning-augmented system design in industry.

Second, Section 3 describes the AutoSys framework that formulates a system decision-making as an optimization task. AutoSys incorporates proven techniques to address common design considerations in building learning-augmented systems. ***(1)*** To support scenario-specific decision-making, AutoSys employs a hybrid architecture – decentralizing inference plane for system-specific interactions such as actuations and exploration, and centralizing training plane for hosting an array of ML/DL algorithms with generalized abstractions. ***(2)*** To handle ad-hoc and nondeterministic jobs spawned by an optimization task, AutoSys employs a cross-layer solution – prioritizing jobs based on their expected gains towards solving the given optimization task and executing jobs in a container to satisfy heterogeneous job requirements in a resource-sharing environment. ***(3)*** To handle learning-induced system failures due to inference uncertainties, AutoSys incorporates a rule-based engine with hard rules authored by experts to check an inferred actuation's commands and assumptions.

Third, we report long-term lessons stemmed from the years of operations, and these lessons include higher-than-expected learning costs, pitfalls of human-in-the-loop, generality, and so on. Prior to sharing these lessons in Section 5, Section 4 quantifies benefits of the learning-augmented design, on Web Search's key application logic and data stores. Compared to years of expert tuning, Web Search exhibits an 11.5% reduction in CPU utilization for a keyword-based selection engine, 3.4% improvement in relevance score for a ranking engine, 16.8% reduction in key-value lookups for a datastore cluster, and so on. The core of AutoSys is open-sourced on GitHub (*https://github.com/Microsoft/nni*).

2 Background and Motivations

As system performance drives end-user experience and revenue, many modern systems are supported by large teams of engineers and operators. This section shares concrete observations in production, which have motivated the industry to transit to the learning-augmented system design. Particularly, we deep dive into one large-scale cloud system – the web-scale search service, or *Web Search*. Web Search is architecturally representative of modern systems, with fundamental building blocks of networking, application logic, and data stores.

2.1 Overview of Web-Scale Search

This section describes the Web Search design with respect to the fundamental building blocks of modern systems.

Distributed and Pipelined Infrastructure. Web Search realizes a multi-stage pipeline of networked services (c.f. Figure 1), to iteratively refine the list of candidate documents for a user search query. The first stage is *Selection service* which selects relevant documents from massive web indexes as candidates for subsequent Ranking service. It relies on both keyword-based and semantics-based matching strategies, i.e., KSE and SSE. Then, *Ranking service* orders these documents according to their expected relevance to the user query, by running the RE ranking engine. Finally, *Re-ranking service* adds additional web contents that are relevant to the user query, and it re-ranks search results. These additional contents are from sources such as stock and weather, and verticals such as news and images. Suppose the user query contains celebrity names, search results will likely have relevant news and images.

Application Logic. We present three applications implemented with rules, heuristics, and ML-based logic.

First, Selection service's Keyword-based engine (KSE) matches keywords in user queries and web documents, by looking up inverted web indexes. Queries are first classified into pre-specified categories. Each category corresponds to a physical execution plan, or a hand-crafted sequence of sub-plans to specify the document evaluation criteria. For example, one sub-plan can specify whether a query keyword should appear in the web document title/body/URL, and how many documents should be retrieved. Sub-plan knobs determine the trade-offs between search relevance and latency.

Second, Selection service's semantics-based engine (SSE) selects web documents with keywords semantically similar to the user query. The problem can be formulated as Approximate Nearest Neighbor (ANN) search [14, 47] in the vector space where keywords that share similar semantics are located in close proximity. The search strategy is an iterative process, and each step can take on one of the three possible actions: ***(1)*** identifying some anchors in the vector space by looking up the tree, ***(2)*** marking anchors' one-hop neighbors in the neighborhood graph as new anchors, and ***(3)*** terminating and returning the best anchors that we have seen. The action sequence determines how fast SSE returns semantically relevant document candidates.

Third, Ranking service's ranking engine (RE) implements a ranking algorithm based on high-performance LambdaMART [12], which uses Gradient Boosted Decision Trees (GBDT) [20]. GBDT is one of the sophisticated ranking algorithms hosted by Web Search, and each targets different query types, document types, languages, and query intentions. Since GBDT combines a set of sub-models to produce the final results, tuning RE requires data scientists to reason about how tuning each sub-model would impact the overall performance.

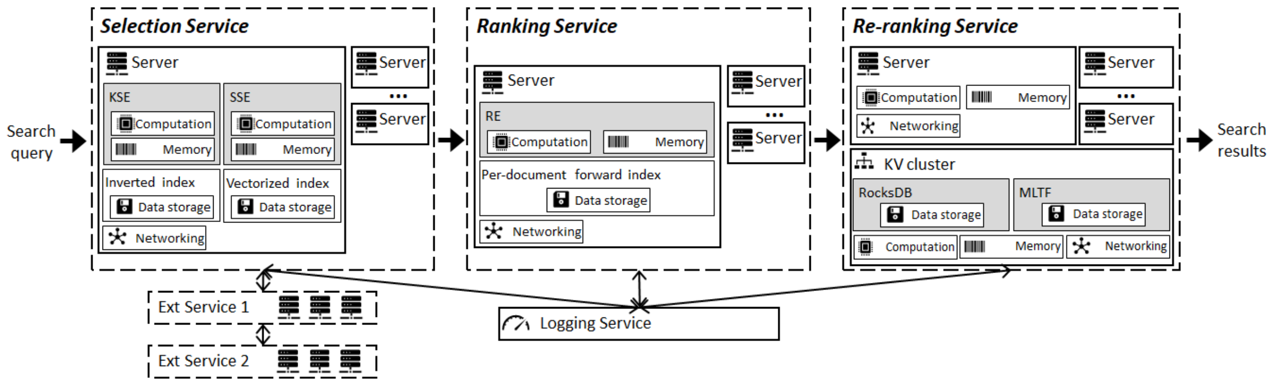


Figure 1: AutoSys drives transitions of several critical engines in Web Search to the learning-augmented design. These engines include *KSE* (Keyword-based Selection Engine), *SSE* (Semantics-based Selection Engine), *RE* (Ranking Engine), *RocksDB* key-value store engine, and *MLTF* (Multi-level Time and Frequency) key-value store engine. Since Web Search is architecturally similar to modern systems in general, AutoSys has also been applied to other production systems at Microsoft.

Data Store. One common data structure of web indexes is the key-value store. Web Search employs both open-sourced RocksDB, and customized solutions such as Multi-level Time and Frequency key-value store (MLTF). MLTF takes key access time and frequency as signals to decide cache evictions.

The index of SSE engine is organized in a mixed structure of space partition tree and neighborhood graph. Space partition tree is used to navigate the search to some coarse-grained subspaces while the neighborhood graph is used to traverse the keywords in these subspaces.

2.2 Sources of System Complexity

Heterogenous Classes of Decisions. Decision-makings in systems can be grouped into three classes: application logic, system algorithms, and system configurations. Each class requires human experts with different skill sets and experience.

First, application logic implements features that fulfill user requirements, so its decision-making process should adapt to user usage. In the case of Web Search, Ranking service hosts hundreds of lightweight and sophisticated ranking algorithms for different user query types and document types. Optimizing these ranking algorithms requires data scientists to have a deep understanding of how different ML/DL capabilities can be combined to match user preferences.

Second, the infrastructure implements system algorithms to better support application requirements with available resources. In the case of Web Search, Selection service has algorithms responsible for compiling user queries into physical execution plans that are specific to underlying hardware capabilities. Optimizing algorithms requires system designers to consider the relationship between application requirements and infrastructure capabilities.

Third, system configurations are knobs for operators to customize systems. Optimizing knobs requires a deep understanding of their combined effects on system behavior [50].

Multi-Dimensional System Evaluation Metrics. Optimizing multiple metrics can be non-trivial if they have different (and potentially conflicting) goals. For instance, Selection service has tens of metrics in different categories: resource usage, response latency, throughput, and search result relevance. Reasoning about the trade-offs among multiple metrics quickly becomes painstaking for humans, as the number of evaluation metrics increases. In some cases, system designers follow a rather conservative rule: improving some metrics without causing other metrics to regress. In fact, any software update in Web Search that can cause search quality degradation should not be deployed, even if it improves some crucial metrics such as the query latency for top queries.

Modern systems can also have meta-metrics that aggregate a set of metrics or measurements over a time period. One example is the "weekly user satisfaction rate" of Ranking service. To optimize these aggregated metrics, system operators need to understand their compositions.

End-to-End and Full-Stack Optimization. Modern systems are constructed with subsystems and components to achieve separation of concerns. Since the end-to-end system performance represents an aggregated contribution of all components, optimizing one component should consider how its outputs would impact others. For example, we have observed that Selection service may increase the number of potentially relevant pages returned, at the risk of increasing spam pages. If the subsequent Ranking service does not consider the possibilities of spam pages, it can hurt user satisfaction.

2.3 Sources of Operation Complexity

Environment Diversity and System Dynamics. While hardware upgrades and infrastructure changes can offer new capabilities, they potentially alter the existing system behavior. An example is how we altered the in-memory caching mechanism design, according to I/O throughput gaps between memory and mass storage medium for different data sizes. Furthermore, hardware upgrades might be rolled out in phases [41], and server resources can be shared with co-located tenants. Therefore, it is possible that instances of a distributed system face different resource budgets.

Modern systems have increasingly adopted tighter and more frequent software update cycles [39]. These software updates range from architecture, implementation, to even data. For example, Selection service has bi-annual major revisions to meet the increasing query volume and Web documents size, or even to adopt new relevance algorithms. And, Re-ranking service can introduce new data structures for new data types, or new caching mechanisms for the storage hierarchy. Finally, software behavior can change with periodic patches, bug fixes [54], and even the monthly index refresh.

Workload Diversity and Dynamics. Interestingly, there can be non-trivial differences among the workloads that individual system components actually observe. The reason is that subsystems can target different execution triggers, or depend on the outputs of others. For example, the list of candidate documents returned by Selection service predominately dictates the workload of Ranking service. And, if a large number of user queries do not have hits in web indexes, Ranking service would have low utilization. The same observation is applicable to the Re-ranking service.

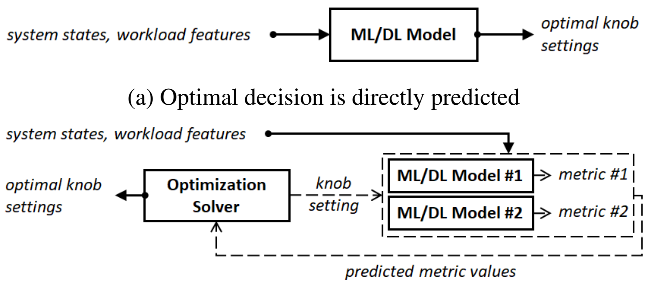
Furthermore, the workload can have temporal dynamics that are predictable and unpredictable, and an example is where the search keyword trend can shift with national holidays and breaking news, respectively.

Non-Trivial System Knobs. Modern systems can expose a large number of controllable knobs to system operators. These knobs include software logic parameters, hardware configurations, actions of an execution sequence, engine selections, and so on. Operation complexity arises from the following observations. A set of knobs can have dependencies [54], i.e., the effect of one knob depends on the setting of another knob. In addition, knobs should be set with the prior knowledge of runtime workloads and system specifications [52]. In the presence of system and workload dynamics, operators need to periodically adjust knob settings for optimal performance. Finally, software parameters can take values of several types: continuous numbers (e.g., 0 - 1,000), discrete numbers (e.g., 1 and 2), and categorical values (e.g., `ON` and `OFF`).

3 AutoSys

We introduce the *AutoSys* framework to unify the development of learning-augmented systems. While AutoSys has driven performance optimization for several production systems at Microsoft, our discussions here focus on Web Search.

Optimization Tasks. In AutoSys, a system decision-making is formulated as an optimization task. In the case of system performance optimization, the output of an optimization task contains optimal values of system knobs. The input consists of system and workload characteristics (e.g., traffic arrival rate). During execution, an optimization task can trigger a sequence of jobs of the following types: ***(1)*** system exploration jobs, ***(2)*** ML/DL model training and inferencing jobs, and ***(3)*** optimization solver. Next, Figure 2 illustrates two use cases of optimization tasks.



(b) Optimal decision is indirectly predicted

Figure 2: Heterogenous classes of decision-makings can be formulated as ML/DL optimization tasks. This figure illustrates two common realizations of optimization tasks.

Figure 2a illustrates the first case where AutoSys predominately learns from human experts, who handcraft the training dataset containing preferable knob settings for some system states and workloads. In this case, the model takes in system states and workload features as inputs, and directly infers the optimal knob settings. As one example implementation, AutoSys can assign a high reward for these preferable knob settings, and the model can implement value functions to find a policy that maximizes the reward for unforeseen inputs.

Figure 2b illustrates the second case where AutoSys predominately learns from interactive explorations with the target system. By automatically generating system benchmark candidates, AutoSys collects measurements to train models. In this case, the model takes in a knob setting and predicts the expected value of performance metrics. Based on these model predictions, an optimization solver can infer optimal knob settings. An example implementation of model and solver is regression models and gradient descent. For cases where a sequence of step-wise actions is necessary such as Selection service's search query plans, the solver can be based on reinforcement learning.

3.1 Design Principles

AutoSys follows the design principles below, to address common considerations in building learning-augmented systems.

To support scenario-specific decision-makings, AutoSys implements a hybrid architecture. Specifically, a centralized training plane is shared across all target systems, and decen-

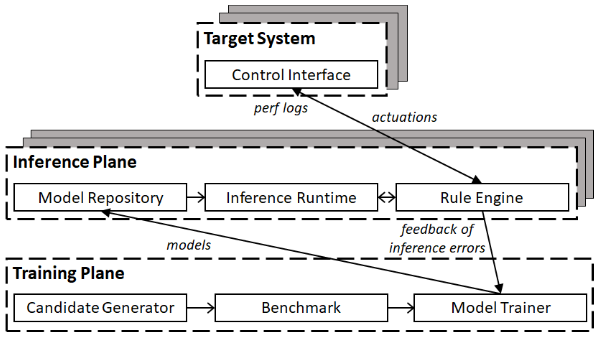


Figure 3: AutoSys framework. It centralizes training plane which hosts an array of ML/DL algorithms, and decentralizes inference plane for system-specific interactions such as actuations and exploration.

tralized inference planes are deployed for each target system. We observe that a centralized training plane promotes sharing data and trained models among scenarios – for example, this can help bootstrapping model training by initializing neural network weights and model hyper-parameters. Decentralized inference planes help distribute inference loads that grow with the system scale, and they also allow scenario-specific customizations such as verification rules.

To manage computation resources, AutoSys implements a cross-layer solution. Specifically, AutoSys abstracts scenarios as optimization tasks, and allows target systems to prioritize jobs spawned by their tasks. Unifying learning-augmented scenarios allows computation resources to be flexibly shared, especially since tasks are ad-hoc and non-deterministic. First, tasks are triggered in response to system dynamics, which might not exhibit a regular pattern. Second, jobs are determined at runtime according to the optimization task progress.

To handle learning-induced system failures, AutoSys implements a rule-based engine to validate actuations. Since most models mathematically encode knowledge learned, existing verification tools might not be applicable. On the other hand, rules are human-readable and human-verifiable.

3.2 Framework Overview

AutoSys executes an optimization task by spawning a number of jobs: ***(1)*** system exploration jobs, ***(2)*** ML/DL model training and inferencing jobs, and ***(3)*** optimization solver. Figure 3 shows the overall AutoSys framework to support these jobs.

Training Plane. The training plane implements features to support both system exploration jobs and ML/DL training jobs. Figure 4 shows the training plane workflow. The first step is candidate generation, which generates knob values to benchmark for the purpose of building up the training dataset. Considering the costs of running system benchmarks, the key is to balance the number of candidates and the model accuracy. Generation algorithms are wrapped in Tuner instances, and we

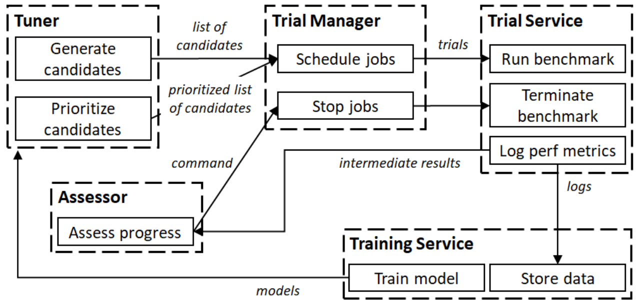


Figure 4: Workflow of training plane. System exploration jobs are wrapped in a Trial object, which collects system benchmark outputs for training models in the Training Service.

have implemented algorithms based on TPE [9], SMAC [26], Hyperband [30], Metis [31], and random search.

The second step is to benchmark configuration candidates. Trial Manager abstracts each system benchmark as a Trial object – the Trial object has fields holding ***(1)*** knob configurations, ***(2)*** execution meta-data: the command to run binaries and even ML/DL models (e.g., RE's hyper-parameter tuning), and metrics to log, ***(3)*** resource requirements (e.g., the number of GPU cores). In the case of KSE, SSE, MLTF, RocksDB engines, their Trial instances point to both the system executable and workload replay tool. The replay tool feeds a pre-recorded workload trace to the executable. In contrast, RE engine has a different goal of optimizing a ranking model's hyperparameters, its Trial instance contains the model and the dataset location. And, invoking `updateConfigs` updates model hyperparameters.

Table 2 presents the Trial Manager API. Invoking `startTrial` submits a Trial instance to Trial Service. At any time, `updateConfigs` can be called to change knob settings, and `getMetrics` can be called to retrieve metric measurements. A Trial can optionally assess its intermediate benchmark results, to decide whether it should terminate the benchmark early. To do so, Trial sends intermediate results to an Assessor instance implementing early-stopping criteria. If the criteria is met, Assessor can invoke Trial Manager's *stopJob*.

The third step is model training. Upon the completion of Trial instances, `getMetrics` outputs are merged with the corresponding knob settings to form a tabular dataset, for Training Service to train models. Historical results are optionally stored in data store, for model re-training or data sharing among similar scenarios.

Inference Plane. The inference plane implements features to support ML/DL inferencing jobs and optimization solver. Taking the current system states and workload characteristics as inputs, these jobs infer optimal actuations. These jobs are typically triggered by events, which are predefined by system operators to support service level agreements. For example, if the target system's workload changes (e.g., an increase in search queries per second), performance drops (e.g., a drop

Name	Description
tuner.**updateSearchSpace**(*args*)	Specify search space. *args* is a list of system knobs' names, value types, and value ranges.
candidates = *tuner*.**generateCandidates**()	Generate and return a list of configuration candidates.
tuner.**generateModel**()	Train the Tuner instance's model.

Table 1: Tuner instance API

Name	Description
trial = **submitTrial**(*args*)	Submit and deploy a benchmark trial. *args* include a configuration candidate, execution meta-data, and scheduling meta-data.
trial.**startTrial**()	Start a benchmark trial.
trial.**stopJob**()	Stop a benchmark trial.
trial.**updateConfigs**(*args*)	Update a trial's configuration candidate. *args* is a configuration candidate.
measurements = *trial*.**getMetrics**()	Return perf measurements of a trial.

Table 2: Trial Manager API

in tail latencies), or models update, inference plane initiates optimization tasks to make system tuning decisions. Furthermore, inference plane can be configured to directly actuate the target system, or simply inform system operators as suggestions. Finally, the inference plane can relay online system performance measurements to the training plane, for training.

3.3 Ad-hoc and Nondeterministic Jobs

In contrast to traditional systems, learning-augmented systems introduce jobs that are difficult for system operators to provision beforehand. First, optimization tasks are *ad-hoc* – they are triggered in response to adapting to system and workload dynamics, which might not exhibit a regular pattern. Second, optimization tasks have *nondeterministic* requirements – they spawn system exploration jobs and ML training jobs according to the optimization progress at runtime. To this end, AutoSys implements mechanisms to prioritize, schedule, and execute these jobs.

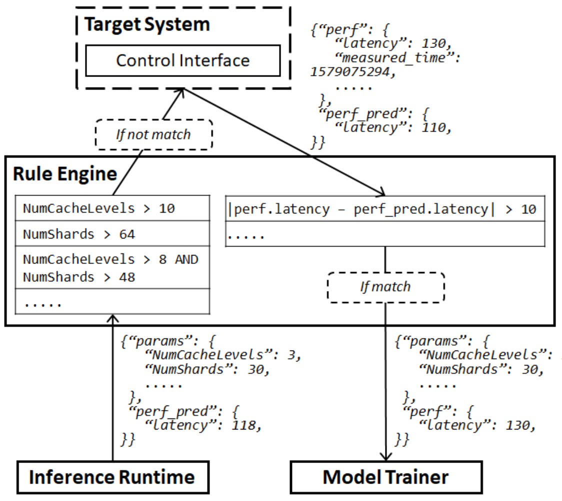
Job Prioritization. The tuner can prioritize the list of candidates in `generateCandidates`, to highlight benchmarks that are expected to subsequently improve model accuracy the most. Unnecessary benchmarks waste time and resources, especially for systems that require warm-up (e.g., in-memory cache warm-up). In this mode, training plane iterates between candidate generation, candidate prioritization, and model training.

We illustrate some of the candidate prioritization strategies that AutoSys Tuners have implemented. First, since the Metis Tuner uses Gaussian process (GP) regression model, it leverages GP's capability to estimate the confidence interval of its inference. A larger confidence interval represents lower inference confidence. And, it prioritizes candidates by sorting their confidence interval in descending order. Second, the TPE Tuner maintains two mixture models to learn the distribution of top-performing knob combinations [9]. It computes how likely a candidate belongs to this distribution, and prioritizes by sorting likelihood scores in descending order.

Job Scheduling. Trial Manager schedules Trials according to priorities and available resources, and passes this information to underlying infrastructure [3, 49]. Trials can impose heterogeneous resource requirements to support their corresponding decision-making scenarios. Taking system exploration jobs as an example – benchmarking ML/DL-learned system components in RE benefits from access to ML/DL acceleration hardware such as GPUs, but benchmarking MLTF KV engine must take place with SSDs. We note that scheduling learning jobs also have similar considerations. For learning approaches based on neural networks, their training jobs can be scheduled to machines with GPUs and TPUs [1] for acceleration. Some learning approaches such as Metis maintain a collection of ML models, and their training and inference time can significantly benefit from multiple CPU cores.

Job Execution. In addition to natively running system exploration jobs (i.e., Trial instances) on real machines, AutoSys also supports containers such as Docker [2]. The container image packages a Trial's software dependencies including the target system's binaries and libraries. Containers benefit AutoSys in the following ways. First, containers can be started and stopped to share hardware resources among multiple target systems. Second, previous efforts reported that containers exhibit a much lower overhead than virtual machines [19]. This improves the benchmark fidelity, hence AutoSys's training data quality. Second, the capability of deploying an image on heterogeneous machines easily enables benchmarking a target system under different hardware environments.

In addition to allocating short-lived containers that run only one benchmark, AutoSys also offers long-lived containers. For Trials with multiple benchmarks, long-lived containers effectively amortize the cost of initializing and loading the image. Furthermore, if consecutive benchmarking jobs need to share states (e.g., warmed-up caches) and data (e.g., weight sharing for tuning ML/DL model hyperparameters), contents in memory can be retained and reused.



```
{"perf": {
    "latency": 130,
    "measured_time":
1579075294,
    .....
 },
 "perf_pred": {
    "latency": 110,
}}
```

```
NumCacheLevels > 10
NumShards > 64
NumCacheLevels > 8 AND
NumShards > 48
.....
```

```
|perf.latency - perf_pred.latency| > 10
.....
```

```
{"params": {
    "NumCacheLevels": 3,
    "NumShards": 30,
    .....
 },
 "perf_pred": {
    "latency": 118,
}}
```

```
{"params": {
    "NumCacheLevels": 3,
    "NumShards": 30,
    .....
 },
 "perf": {
    "latency": 130,
}}
```

Figure 5: Workflow of rule engine in inference plane.

3.4 Learning-Induced System Failures

Being stochastic in nature, ML/DL inference exhibits some degrees of uncertainty, and this uncertainty can lead to learning-induced system failures or suboptimality. While failures are not unique to learning-augmented systems [8], handling them requires a different approach for the following reasons. First, ML/DL models mathematically encode knowledge learned from the training data, the meaning of their internal weights is not interpretable to humans and common formal verification techniques. Second, as models autonomously learn from the training data, it is difficult to assess whether a dataset would guarantee a model to fully learn a particular concept.

Since it is hard to formally verify ML/DL correctness, AutoSys opts to validate ML/DL outputs with a rule-based engine. These validation rules are authored by operators, and the rule engine functions as a blacklist. Each rule specifies conditions of a violation to catch, and it has the following format: $(Predicate_1\ AND|OR\ Predicate_2 ...)$. A predicate is one variable value comparison with operators such as $==$, $!=$, $>=$, $>$, $<=$, and $<$. Conceptually, AutoSys maintains the following two rulesets for each target system.

First, ruleset validates ML/DL actuations, or inference runtime outputs as illustrated in Figure 5. In addition to validating parameter value constraints, if certain system states have been known to cause failures (from either past experience or bug reports), system operators can prevent those configurations from being applied. Rules can also encode knob dependencies – an example is the multi-tenant setup where the total memory allocated to all tenants must not exceed a budget, and the blacklist rule can be written as $capacity_1 + capacity_2 > 1024$.

Second, ruleset checks the actuation feedback, or target system outputs as illustrated in Figure 5. One use case in our deployments is to check discrepancies between actual system states and ML/DL inference. Specifically, if the predicted performance (of an actuation) significantly differs from the actual performance measurement, this feedback is relayed to the training plane as additional training data. An example rule is $|perf.latency - perf_pred.latency| > 10$ in Figure 5.

3.5 Extensibility

Since tuning scenarios can vary in requirements, we design AutoSys to be extensible through the Tuner abstraction. Tuner is agnostic to specific candidate generation algorithms, and it provides APIs to wrap the underlying ML/DL details (c.f. Table 1). After a Tuner instance is instantiated, users can specify its search space by invoking `updateSearchSpace` method. The method argument is a list of system knobs' names, value types, and value ranges. Invoking `generateCandidates` generates a list of configuration candidates to be benchmarked for model training. Then, after benchmarks complete, AutoSys invokes `generateModel` to train and update the Tuner instance's model in Training Service.

We have implemented Tuners based on algorithms including TPE [9], SMAC [26], Hyperband [30], Metis [31], and random search. Our implementation of `updateSearchSpace` allows system operators to specify each parameter's expected value type: `choice` (e.g., categorical values for KSE engine's `RankingStreams` parameter), `uniform` (e.g., continuous number within a range for RE engine's `LearningRate`), `randint` (e.g., integers between within a range for RocksDB engine's `WriteBufferSize` parameter), and so on. Finally, for model-less algorithms such as random search, it is not necessary to implement `generateModel`.

3.6 Implementation

Our current implementation comprises ∼18,205 lines of Python code (Tuner: 5,427, Assessor: 1,392, Trial Manager: 35, Trial Service: 28), ∼12,852 lines of TypeScript code (Trial Manager: 3,283, Trial Service: 6,638), and ∼13,344 lines of code in other languages. We have implemented Tuners for an array of popular optimization algorithms such as TPE (Tree-structured Parzen Estimator) [9], SMAC (Sequential Model-based Algorithm Configuration) [26], Hyperband [30], Metis [31], anneal, naïve evolution, grid search, and random search. We have also implemented two early-stopping algorithms based on median stop [23] and curve fitting [18]. Our current implementation supports the following Trial Service realizations: local machine, remote servers, several Kubernetes-based platforms, and several internal experimental platforms. We have open-sourced the core of AutoSys on GitHub (*https://github.com/Microsoft/nni*).

4 Production Deployment Measurements

This section presents production measurements of Web Search, and Table 3 summarizes key results. The goal is to

	Search space size	Tuning time	Key results (vs. long-term expert tuning)
Keyword-based Selection Engine (KSE)	$O(1000^n)$	1 week	Up to 33.5% and 11.5% reduction in 99-percentile latency and CPU utilization, respectively
Semantics-based Selection Engine (SSE)	Action sequences	1 week	Up to 20.0% reduction in average latency
Ranking Engine (RE)	$O(10^n)$	1 week	3.4% improvement in NDCG@5
RocksDB key-value cluster (RocksDB)	$O(100^n)$	2 days	Lookup latency on-par with years of expert tuning
Multi-level Time and Frequency key-value cluster (MLTF)	$O(100^n)$	1 week	16.8% reduction on avg in 99-percentile latency

Table 3: Summary of adopting learning-augmented design to tune various systems of Web Search (c.f. Section 2.1). We compare key results to the previous practice of manual tuning by human experts over the years. n represents the number of parameters.

quantify benefits of adopting learning-augmented system design in terms of ***(1)*** tuning effort reduction and ***(2)*** system performance improvement.

4.1 Tuning Application Logic

This subsection considers both cases of tuning application logic in a single step and in a sequence of step-wise actions. Specifically, we present measurements from Selection service's KSE engine and SSE engine.

Performance Gain for KSE Engine. KSE engine exposes the following key knobs. Each execution plan consists of a hand-crafted sequence of sub-plans. Each sub-plan has a categorical parameter, *RankingStreams* (`title`, `body`, `anchor`, and `URL`), that specifies document fields that a query keyword should appear in. In addition, it has an integer parameter, *MaxSeekCount* (1 - 1000), that dictates the maximum number of documents the sub-plan should examine. These parameters determine the trade-off between Selection service effectiveness and latency – while a large *MaxSeekCount* potentially increases the number of document candidates for ranking, it also increases the Selection service latency. Depending on the number of execution plans, there can be up to 20 controllable knobs and parameters.

Metrics of interests include per-query latency, CPU utilization, and relevance. As Selection outputs an un-ranked list, we use the popular NCG (Normalized Cumulative Gain) score to quantify the overall relevance, and this is a variant of NDCG [5] that does not consider position-based discounting. Importantly, the higher the NCG, the more likely users will click the corresponding search result. For KSE, the optimization target is to reduce latency and CPU utilization, while keeping relevance score the same.

We optimized KSE for the image and video domain. Web Search divides the image and video domain into several segments: generic, tail (e.g., lower popularity), regions (e.g., US market), and so on. With production workloads, we ran the TPE (Tree-structured Parzen Estimator) Tuner for a week, on a machine with 2.1 GHz CPU (with 8 cores) and 16 GB RAM. Compared to years of expert-tuning, we highlight the following improvements. For the image domain, AutoSys lowered KSE 99-percentile latency by another 16.9% - 33.5%, and CPU utilization by another 9.0% - 11.0%. For the video domain, compared to expert-tuned configurations, AutoSys lowered KSE 99-percentile latency by another 19.4% - 29.7%, and CPU utilization by another 10.1% - 11.5%. These improvements represent a Selection latency reduction up to 33 msec; for reference, many companies have reported an ~1.0% revenue gain from reducing the end-to-end search engine latency by 100 msec.

Performance Gain for SSE Engine. SSE engine optimization concerns with deciding the action for each step of the execution sequence. In contrast to other engines in Web Search, SSE requires a correct ordering of actions. The problem can be formulated as the Approximate Nearest Neighbor (ANN) search [14, 47] in the vector space. As we mentioned before, given a user query, each step of SSE chooses one of the three possible actions: ***(1)*** identifying some anchors in the vector space by looking up the tree, ***(2)*** making anchors' one-hop neighbors in the neighborhood graph as new anchors, and ***(3)*** terminating and returning the best anchors that we have seen. At each step, SSE provides the following system states and environment features for the decision-making: the number of distance calculations between candidates and the query so far, the number of tree searches so far, whether top K candidates have been updated in the last T actions, and the average distance between current top K candidates and the query.

We implemented a reinforcement learning Tuner with tabular based models (Q-tables). Reinforcement learning excels in discovering the action sequence that would maximize the overall reward. We define the reward of step t as a trade-off between relevance gain and latency cost: $R_t = \alpha \times relevance\_gain_t - \beta \times latency\_cost_t$ (α and β are hyper-parameters). Under Web Search production workload, we observed that learned execution sequences are able to achieve an average of 20.0% reduction in latency while keeping the relevance score the same.

Additional Consideration: Actuation Granularity. While setting up AutoSys for KSE, we encountered the question of actuation granularity – *should AutoSys generate coarse-grained actuations (i.e., one actuation for all system instances) or fine-grained actuations (i.e., one actuation for each system instance, user segment, region, and so on)*? Coarse-grained actuations impose less computation loads, but fine-grained actuations potentially offer higher performance gains. Unfortunately, the real-world value of learning-augmented design diminishes with either high learning cost

	RS	MS_1	MS_2	MS_3	MS_4
Image-generic	U,T,B	300	6	9,00	160
Image-tail	U,T,B	484	10	6,30	130
Image-US	U,T,B,C	245	50	4,80	90
Video-generic	U,T,C	220	160	6,50	10
Video-tail	U,T,C	125	30	6,40	180

Table 4: Optimal decisions should vary with workload diversity. This table illustrates the optimal configuration of key KSE knobs for several segments of the image and video search domain. *RS* and *MS* are the abbreviated name for *RankingStreams* and *MaxSeekCount* parameters, respectively.

or low performance gain.

To balance the trade-off, we experimented with three levels of granularity: system-wide, per-instance, and per-search-segment. Due to restrictions imposed by the production environment, we performed exploration jobs on a random selection of 3 instances in each search segment. Results suggested that per-search-segment granularity best balances the trade-off for KSE. Table 4 illustrates the best-performing knob configurations for five popular segments, and there is a noticeable variance in their knob settings.

4.2 Tuning ML Algorithms

This subsection considers tuning system components that host ML/DL algorithms. Specifically, we present measurement from Ranking service's RE engine.

Performance Gain for RE Engine. RE engine runs a set of decision trees, or random forest. Its key controllable knobs include *LearningRate*, *NumberOfLeaves*, *MinimumDocsPerLeaf*, *NumberOfTrees*, and so on. *LearningRate* takes a continuous number (0.01 - 0.99), for adjusting gradient descent speed that trades off between learning convergence time and accuracy. *NumberOfLeaves* takes an integer (10 - 5,000), for adjusting the maximum number of base tree leaves, which relates to the model's learning capability. *MinimumDocsPerLeaf* takes an integer (5 - 1,000), for adjusting the minimum number of documents in a leaf. *NumberOfTrees* takes an integer (5 - 100), for adjusting the number of decision trees. In total, there are approximately 5×10^8 possible parameter combinations in the configuration space.

The optimization metric is NDCG (c.f. Section 4.1), and we ran the TPE (Tree-structured Parzen Estimator) Tuner in AutoSys for one week, on a machine with 2.1 GHz CPU (with 8 cores) and 16 GB RAM. Compared to years of expert-tuning, we highlight the following improvements (evaluated on a production workload containing 150K queries and 2.5M URLs): AutoSys improved NDCG@1 (i.e., top 1 result's NDCG score) by another 2.9%, NDCG@2 (i.e., top 2 results' NDCG score) by another 3.4%, NDCG@3 by another 3.4%, NDCG@4 by another 3.4%, and NDCG@5 by another 3.4%. We note that ranking relevance has a direct correlation with conversion rate (e.g., ads clicking).

Additional Consideration: Human-in-the-Loop. We note that solving the combinatorial optimization from a total of 5×10^8 possibilities is theoretically doable, but it might not be practically feasible. With RE, we took advantage of human knowledge of the engine design, and reduced the value range of several parameters during the process. Interestingly, we have encountered cases where information from humans unintentionally misled learning or caused unexpected consequences, and Section 5 shares these cases.

4.3 Tuning Data Store

Through both RocksDB engine and MLTF engine in Reranking service, we demonstrate data store optimization.

Performance Gain for RocksDB Engine. From years of operation, Web Search operators selected the following key knobs to optimize RocksDB read and write throughputs (in MB per second): *WriteBufferSize* (1 - 96 MB), *BlockSize* (128 - 2,000 KB), *Level0FileNumCompactionTrigger* (2 - 64), and *MaxBackgroundJobs* (1 - 45). Details of these knobs are available online [4]. We used the Metis Tuner because gaussian process models have been shown to be effective for tuning databases [6].

We allocated a two-day computation budget (on an 8-core 2.1 GHz CPU), for AutoSys to search for the optimal configuration with respect to a 5-day trace of production Web Search traffic. AutoSys improves the maximum write throughput to 50.36 MB per second, which matches the throughput achieved by Web Search operators' years of manual tuning. This result demonstrates that AutoSys can significantly reduce the amount of human efforts.

Performance Gain for MLTF Engine. MLTF (Multi-level Time and Frequency) KV engine has the following key knobs. There are *NumCacheLevels* (1 - 10) cache levels. A cached object can move up a level if it has been queried *CachePromotionThreshold* (1 - 1,000) times. Top *NumInevictableLevels* (0 - 9) cache levels can be specified as being inevictable. Furthermore, MLTF does not immediately admit large keys with an object size larger than *AdmissionThreshold* (1 - 1,000) bytes, but it first holds them in a shadow buffer of *ShadowCapacity* (1 - 10) MB. Then, keys in the shadow buffer are moved to the cache only if they have been queried more than *ShadowPromotionFreq* (1 - 1,000) times. The dataset partition on each server is divided into *NumShards* (1 - 64) shards. The metric of interests is the 99-percentile query latency. Due to the noise in latency measurements, we used the Metis Tuner. We collected measurements from one production cluster whose servers have a 512 MB in-memory cache and SSD.

With measurements from a 14-day window, we try to answer the question, *can AutoSys continuously maintain optimal system performance over time?* AutoSys generated a new configuration every two hours to adapt to workload dynamics.

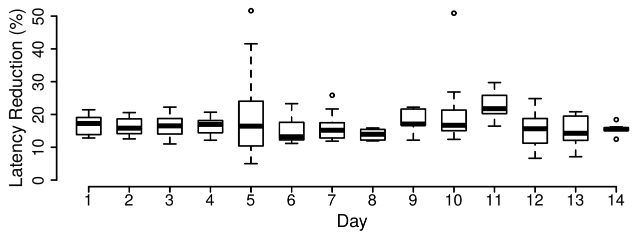


Figure 6: The figure summarizes the 99-percentile latency reduction over a 14-day window, as compared to the static default configuration. AutoSys periodically tuned MLTF every two hours, so we have 12 actuation feedbacks per day.

Figure 6 summarizes the latency reduction for each day, as compared to the default configuration from human operators. We highlight the following observations. First, AutoSys lowered the 99-percentile latency by an average of 16.8%. Second, if the workload changes too frequently, AutoSys might not be able to always update the system configure in time.

Additional Consideration: System Measurement Quality. Noise and outliers are the common factors that system engineers typically consider in terms of measurement quality. In fact, as mentioned above, they are the reason that we used the Metis Tuner for MLTF. Interestingly, while setting up AutoSys for RocksDB, we encountered another factor of system measurement quality – imbalanced measurements. In imbalanced datasets, data points are not roughly equally distributed among all classes of behavior (e.g., read vs. write requests). For example, in one RocksDB scenario, the workload trace is significantly skewed, and writes significantly dominate reads. As a result of using this trace for training, AutoSys optimization tasks frequently produced actuations that sacrificed read throughput for write throughput. These actuations might not be acceptable in the real world.

5 Long-Term Lessons Learned

Although AutoSys addresses common design considerations (c.f. Section 3), unforeseen implications have surfaced over years of operation. They reveal roadblocks in the transition to learning-augmented system design, from the perspective of system operators.

5.1 Higher-Than-Expected Learning Costs

While we anticipated model training would incur some costs, these costs sometimes exceed our expectations due to how operators set up models. The common approach is to model an entire system deployment as one black box. Since ML/DL models directly learn from the observed system execution behavior, operators are freed from worrying about non-trivial component interactions and resource contentions within the deployment. This benefit of simplicity is attractive because interactions and contentions are unavoidable in modern systems – if multiple service instances are deployed on the same server, they would contend for computation and I/O resources, especially on over-subscribed servers. Even for single-instance servers, instances share network resources, job dispatcher, etc.

The first unapparent trade-off of modeling an entire system deployment as one black box is *re-training cost*. Compared to traditional systems, most modern systems are designed to be elastic. Individual instances can be created and destroyed on demand, and they can run on heterogeneous hardware as required. Unfortunately, any changes to the deployment setup would invalidate model assumptions and cause the trained model to be irrelevant. In our example above, system setup changes include the number of co-located instances on a server and instance migration. Re-training models for modern systems can be costly. Complex systems require complex ML/DL models, which tend to be difficult to train and require a large amount of training data.

The second unapparent trade-off of modeling an entire system deployment as one black box is *exploration cost*. Considering optimizing the job completion time for a cluster of hundreds (or even thousands) workers and job dispatchers, if we consider individual nodes' CPU utilization, the model would already have hundreds of inputs to learn. Furthermore, preparing training datasets for this model scale can be challenging: ***(1)*** testbeds rarely match the target system's hyper scale in the real world, and ***(2)*** exploratory actuations on critical systems in production are prohibitive.

To mitigate higher-than-expected learning costs, one ongoing effort is to take advantage of the target system's software modularity [40]. Software modularity emphasizes separating code functionality to promote maintainability. Similarly, instead of modeling an entire system deployment with a monolithic model, we modularize the learning task into composable units of learning assignments. One realization is to dedicate a model to learn a subsystem or a component. Considering a content-aggregation application that queries two local key-value stores for images and videos in series, we can have separate latency-predicting models for these stores, M_{image} and M_{video}. And, the end-to-end latency can be computed by aggregating outputs of M_{image} and M_{video}. Due to the separation, each model has less to learn and can be re-trained independently. Furthermore, if the application is updated to aggregate new content types, additional models (e.g., M_{text}) can be added without updating M_{image} and M_{video}.

We acknowledge that software design modularity might not always be the appropriate level of modularity, especially that software modularity is typically based on the criteria of code functionality and maintainability, rather than learning complexity. This process is currently a manual trial-and-error process for individual systems, and we are accumulating experience to standardize the methodology.

5.2 Pitfalls of Human-in-the-Loop

Senior engineers and operators likely have a wealth of knowledge and experience on the target system, which can guide AutoSys optimization tasks. This subsection describes cases where information from humans unintentionally misled learning or caused unexpected consequences.

First, human experts can inject biases into training datasets, by providing a large number of labeled data points for certain search space regions. This is possible if human experts are already familiar with these regions. As a result, models would exhibit an uneven distribution of uncertainties. For optimization algorithms that tend to exploit regions with lower uncertainties, e.g., Expected Improvement (EI) [44], decisions would likely lean towards regions labeled by human experts. While the academic community has investigated data bias in the context of classification (e.g., images [46]), learning-augmented systems also rely on regression. Our current practice is to advise operators to mix human-labeled datasets with random exploration.

Second, human experts can write conflicting specifications for optimization tasks. Specifically, human experts can help AutoSys narrow down the search space by specifying the *valid* value ranges of each configuration knob, and an example is RE described in Section 4.2. At the same time, they can specify *invalid* configurations for the rule engine to check optimization task outputs. Due to human errors, if the invalid space completely covers the valid space, any outputs would effectively be rejected by AutoSys. Our current practice is to run a tool to check this overlapping condition.

5.3 Closed-Loop System Control Interfaces

We have worked with many production systems that lack closed-loop control interfaces. The closed loop refers to how AutoSys actuates a system to achieve optimality, based on the current system feedback. To this end, not only do modern systems need interfaces to accept external actuations, but they should also have well-defined interfaces that abstract system measurements and logs in a way of facilitating learning.

We describe common issues that motivate this need. First, some systems distribute configurable parameters and error messages over a set of not-well documented configuration files and logs [42]. And, directly modifying configuration files means that the system can not enforce value checks or provide immediate feedbacks. Second, parsing raw logs can be time-consuming, especially if system components disagree on a unified logging format or excessively log [27]. Third, many system feedbacks are not natively learnable, e.g., stack traces and core dumps.

To this end, we have been customizing closed-loop control interfaces for individual systems. Our current practice consists of the following steps. We ensure interfaces contain accessors for all configurable knobs and also accessors for system metrics. The latter output system measurements in the format of time-series values, which capture system measurements since the last AutoSys actuation. Furthermore, control interfaces implement mechanisms to remove system-specific data outliers (e.g., Gaussian noise and spikes), to improve the quality of system benchmark measurements as training data.

5.4 Applicability to Other Systems

This subsection summarizes our experience in applying AutoSys to systems other than Web Search. AutoSys works extremely well in a well-controlled learning environment where high-quality workload traces can be easily collected from the target system, and training can take place offline on high-fidelity testbeds or simulators. Interestingly, many critical scenarios already have the infrastructure to satisfy these strict requirements for debugging purposes.

Many target systems have a more relaxed learning environment. First, real-time exploration can be slow, especially for systems that require warm-up (e.g., in-memory cache). For Tuners based on Bayesian optimization or reinforcement learning, training can take a long time. Our current practice is to run multiple Trials for multiple concurrent benchmarks, at each iteration of exploration. Second, online in-situ exploration with production systems can be restricting and even prohibitive. our current practice is to construct base models offline by running exploration on testbeds or simulators, and then fine-tune models online with live traffic. This practice is useful, especially for systems where individual instances exhibit different workload characteristics. Finally, Section 5.1 discusses cases of frequent model retraining, due to various types of dynamics.

6 Related Work

There are efforts on exploring and demonstrating the potential of learning in solving certain system challenges. Building on these efforts, AutoSys takes a step towards unifying the development of learning-augmented systems. Anticipating growing system scale and complexity, Self-* [22] stated a vision of autonomic computing that satisfies a collection of "self-*" properties, and proposed a conceptual model. Recent efforts include learning index structures and memory access patterns [25, 28], optimizing data query evaluations [37], system performance tuning [7, 31], database configuration tuning [6, 13], placing deep learning computational graphs onto hardware device [35, 36], anomaly detection [21, 29, 55], etc.

There are efforts on building general-purpose predictive service. Resource Central [16] is a predictive service to drive Azure's VM scheduler, and it builds random forest and XGBoost models from past VM telemetry, rather than interactive explorations. Vizier [23] is a general-purpose black-box optimization service, and it has enabled tasks such as parameter tuning at Google. Vizier implements Bayesian optimization to learn the search space through interactive explorations. Clipper [17] is a general-purpose low-latency prediction serving system which introduces a modular architecture to simplify model deployment across frameworks. However, these efforts do not consider some of the challenges in operationalizing learning-augmented systems such as interactive explorations, learning-induced system failures, and so on.

Some AutoSys components are inspired by decades of research and experience in the system community. Many efforts heavily focus on system challenges to support learning tasks [15, 38, 49], and Berkeley shared their views of system challenges for artificial intelligence (AI) [45]. Related to control interfaces, interfaces and methods for controlling and exploring systems state are used for implementation-level model checking (e.g., MaceMC [24] and Modist [51]). One approach to drive automating system performance tuning is interactive exploration. Fuzz testing has been effectively used in generating inputs to induce unexpected software behavior [10, 32, 53], and there is a rich literature on software testing and system debugging. Inspired by the idea of composable AI [45], we are exploring how assembling previously trained models can scalably model large-scale systems.

Finally, some AutoSys components are inspired by research in the ML/DL community. Examples include online learning [11], continual learning [43], and so on.

7 Conclusion

This paper reports our years of experience in designing and operating learning-augmented systems at Microsoft. To unify the development process of these systems, we introduce the AutoSys framework that addresses common design considerations. Furthermore, we present production measurements and discuss long-term lessons learned from operating one such system, Web Search. Going forward, we will study how learning-augmented systems should evolve models over time, and how end-to-end and full-stack system optimization can be safely carried out in practice.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Ana Klimovic, for their extensive comments and suggestions. We also thank our colleagues at Microsoft, for their help in deploying AutoSys: Mingqin Li (Bing Platform), Wei Li (Bing Platform), Haidong Wang (Bing Platform), Yuanchi Yan (Bing Platform), Chao Zhang (Bing), Qiang Zhang (Bing), and Wenjin Zhang (Bing).

References

[1] Cloud TPU. http://cloud.google.com/tpu/.

[2] Docker. http://www.docker.com.

[3] Resource Scheduling and Cluster Management for AI. http://github.com/microsoft/pai.

[4] RocksDB Tuning Guide. http://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[5] AGICHTEIN, E., BRILL, E., AND DUMAIS, S. Improving Web Search Ranking by Incorporating User Behavior Information. In *SIGIR* (2016), ACM.

[6] AKEN, D. V., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD* (2017), ACM.

[7] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (2017), USENIX.

[8] BENSON, T., AKELLA, A., AND SHAIKH, A. Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services. In *SIGCOMM* (2011), ACM.

[9] BERGSTRA, J., BARDENET, R., BENGIO, Y., AND KEGL, B. Algorithms for Hyper-Parameter Optimization. In *NIPS* (2011).

[10] BIRD, D. L., AND MUNOZ, C. U. Automatic Generation of Random Self-checking Test Cases. *IBM Systems Journal* (1983).

[11] BOTTOU, L., AND CUN, Y. L. Large Scale Online Learning. In *NIPS* (2003).

[12] BURGES, C. J. From RankNet to LambdaRank to LambdaMART: An Overview.

[13] CAO, Z., TARASOV, V., TIWARI, S., AND ZADOK, E. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *ATC* (2018), USENIX.

[14] CHEN, Q., WANG, H., LI, M., REN, G., LI, S., ZHU, J., LI, J., LIU, C., ZHANG, L., AND WANG, J. SPTAG: A Library for Fast Approximate Nearest Neighbor Search. http://github.com/microsoft/SPTAG, 2018.

[15] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., SHEN, H., COWAN, M., WANG, L., HU, Y., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI* (2018), USENIX.

[16] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource Central: Understandingand Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP* (2017), ACM.

[17] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI* (2017), USENIX.

[18] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curve. In *IJCAI* (2015).

[19] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. Tech. rep., IBM Research, 2014.

[20] FRIEDMAN, J. H. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* (2001).

[21] GABEL, M., SCHUSTER, A., BACHRACH, R.-G., AND BJORNER, N. Latent Fault Detection in Large Scale Services. In *DSN* (2012), IEEE.

[22] GANGER, G. R., STRUNK, J. D., AND KLOSTERMAN, A. J. Self-* Storage: Brick-based Storage with Automated Administration. Tech. rep., CMU, 2003.

[23] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A Service for Black-Box Optimization. In *SIGKDD* (2017), ACM.

[24] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP* (2011), ACM.

[25] HASHEMI, M., SWERSKY, K., SMITH, J. A., AYERS, G., LITZ, H., CHANG, J., KOZYRAKIS, C., AND RANGANATHAN, P. Learning Memory Access Patterns. *CoRR* (2018).

[26] HUTTER, F., HOOS, H., AND LEYTON-BROWN, K. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION* (2011), Springer.

[27] JIANG, W., HU, C., PASUPATHY, S., KANEVSKY, A., LI, Z., AND ZHOU, Y. Understanding Customer Problem Troubleshooting from Storage System Logs. In *FAST* (2009), USENIX.

[28] KRASKA, T., BEUTEL, A., CHI, E. H., DEAN, J., AND POLYZOTIS, N. The Case for Learned Index Structures. In *SIGMOD* (2018), ACM.

[29] LAPTEV, N., AMIZADEH, S., AND FLINT, I. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *KDD* (2015), ACM.

[30] LI, L., JAMIESON, K., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. In *ICML* (2018).

[31] LI, Z. L., LIANG, C.-J. M., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC* (2018), USENIX.

[32] LIANG, C.-J. M., LANE, N. D., BROUWERS, N., ZHANG, L. L., KARLSSON, B., LIU, H., LIU, Y., TANG, J., SHAN, X., CHANDRA, R., AND ZHAO, F. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *MobiCom* (2014), ACM.

[33] LIANG, C.-J. M., XUE, H., YANG, M., AND ZHOU, L. The Case for Learning-and-System Co-design. In *SIGOPS Operating Systems Review* (2019), ACM.

[34] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural Adaptive Video Streaming with Pensieve. In *SIGCOMM* (2017), ACM.

[35] MIRHOSEINI, A., GOLDIE, A., PHAM, H., STEINER, B., LE, Q. V., AND DEAN, J. A Hierarchical Model for Device Placement. In *ICLR* (2018).

[36] MIRHOSEINI, A., PHAM, H., LE, Q. V., STEINER, B., LARSEN, R., ZHOU, Y., KUMAR, N., NOROUZI, M., BENGIO, S., AND DEAN, J. Device Placement Optimization with Reinforcement Learning. *CoRR* (2017).

[37] MITRA, C. R. D. J. G. G. B., AND TIWARY, S. Optimizing Query Evaluations using Reinforcement Learning for Web Search. In *SIGIR* (2018), ACM.

[38] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., AND STOICA, I. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI* (2018), USENIX.

[39] NEAMTIU, I., AND DUMITRAS, T. Cloud Software Upgrades: Challenges and Opportunities. In *MESOCA* (2011), IEEE.

[40] PARNAS, D. On the Criteria To Be Used in Decomposing System into Modules. In *ACM Communication* (1972), ACM.

[41] PATTERSON, D. A. Technical Perspective: The Data Center Is The Computer. *ACM Communication* (2008).

[42] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *ICSE* (2011), ACM.

[43] RING, M. B. CHILD: A First Step Towards Continual Learning. In *Machine Learning* (1997), Springer.

[44] RYZHOV, I. O. On the Covergence Rates of Expected Improvement Methods. In *Operations Research* (2014).

[45] STOICA, I., SONG, D., POPA, R. A., PATTERSON, D. A., MAHONEY, M. W., KATZ, R. H., JOSEPH, A. D., JORDAN, M., HELLERSTEIN, J. M., GONZALEZ, J., GOLDBERG, K., GHODSI, A., CULLER, D. E., AND ABBEEL, P. A Berkeley View of Systems Challenges for AI. Tech. rep., Berkeley, 2017.

[46] TORRALBA, A., AND EFROS, A. A. Unbiased Look at Dataset Bias. In *CVPR* (2011).

[47] WANG, J., AND LI, S. Query-driven Iterated Neighborhood Graph Search for Large Scale Indexing. In *SIGMM* (2012), ACM.

[48] WANG, M., CUI, Y., WANG, X., XIAO, S., AND JIANG, J. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network* (2018).

[49] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., YANG, F., AND ZHOU, L. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI* (2018), USENIX.

[50] XU, T., JIN, L., FAN, X., ZHOU, Y., PASUPATHY, S., AND TALWADKE, R. Hey, You Have Given Me Too Many Knobs. In *FSE* (2015), ACM.

[51] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI* (2009), USENIX.

[52] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP* (2011), ACM.

[53] ZHANG, L. L., LIANG, C.-J. M., LIU, Y., AND CHEN, E. Systematically Testing Background Services of Mobile Apps. In *ASE* (2017), ACM.

[54] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *ICSE* (2014), ACM.

[55] ZHANG, X., LIN, Q., XU, Y., QIN, S., ZHANG, H., QIAO, B., DANG, Y., YANG, X., CHENG, Q., CHINTALAPATI, M., WU, Y., HSIEH, K., SUI, K., MENG, X., XU, Y., ZHANG, W., SHEN, F., AND ZHANG, D. Cross-dataset Time Series Anomaly Detection for Cloud Systems. In *ATC* (2019), USENIX.