The Case for Learning-and-System Co-design

Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou
Microsoft Research

Abstract

While decision-makings in systems are commonly solved with explicit rules and heuristics, machine learning (ML) and deep learning (DL) have been driving a paradigm shift in modern system design. Based on our decade of experience in operationalizing a large production cloud system, Web Search, learning fills the gap in comprehending and taming the system design and operation complexity. However, rather than just improving specific ML/DL algorithms or system features, we posit that the key to unlocking the full potential of learning-augmented systems is a principled methodology promoting *learning-and-system co-design*. On this basis, we present the AutoSys, a common framework for the development of learning-augmented systems.

1 Introduction

Decision-making runs through each stage of system design, optimization and operation – these decisions govern how systems handle application requests under a particular operation environment, to satisfy user requirement. Examples include system configuration, database query plan formulation, routing decisions by networking infrastructure, job scheduling for data processing clusters, document ranking in search engines, and so on.

Most of these decision-makings are solved with explicit rules or heuristics based on human experience and comprehension. While heuristics perform well in general, they can be suboptimal for several reasons. First, the complexity of modern systems can grow beyond what humans can correctly reason about. Second, most heuristics are statically optimized for some assumed general cases, and any hardware/software changes and workload dynamics can break these assumptions.

Recently, machine learning (ML) and deep learning (DL) have driven a paradigm shift in system design and operation. As ML/DL excels in learning complex data patterns, it holds the potential in filling the gap of comprehending the complexity of modern systems operating in a highly dynamic environment. Various efforts [3, 4, 11, 17, 20, 23, 34] have found success in formulating certain system decision-makings into ML/DL tasks. Conceptually, trained models can provide guidance to system designers and operators [18], and augment (or even make decisions on behalf of) existing system logic [17]. We coin the term *learning-augmented systems*, to describe modern systems whose design methodology or control logic is at the intersection of traditional heuristics and ML/DL learning.

Due to the interdisciplinary nature, learning-augmented systems have long been considered difficult to build and require a team of engineers and data scientists. We argue that choosing off-the-shelf ML/DL algorithms is simply one step in the process – challenges arise from the fact that modern systems are not designed to be inherently learnable, nor to manage learning tasks. For the former, modern systems generally do not have a well-specified approach for ML/DL components to collect high-quality data for training. For the latter, modern systems lack the capability of validating ML/DL actuations and managing ML/DL model updates.

Rather than just improving specific ML/DL algorithms or system features, we posit that the key to unlocking the full potential of learning-augmented systems is a principled methodology promoting *learning-and-system co-design*. To this end, §2 first motivates the design and operation complexity of modern systems, based on our decade of experience in operationalizing one large production cloud system, Web Search. Then, §3 discusses design considerations towards making system learnable and learning manageable. On this basis, §4 presents the AutoSys framework to guide the development of learning-augmented systems.

2 Motivating Case Studies

To motivate and illustrate complexities in designing and operationalizing modern systems, this section describes one large-scale cloud system, *Web Search*. Like modern systems, Web Search is architecturally modularized into multiple stages, and interactions happen among services and components. Furthermore, since its system design requirements and components span computation, storage and networking, Web Search allows us to study how the learning-augmented design can be applied to these fundamental building blocks of modern systems.

Overview of Web Search. Web Search has a well-defined but yet challenging goal: given a user search query, it should return an ordered list of relevant search results with minimum latency, from hundreds of billions of documents of various types (e.g., texts, images and multimedia contents). One approach to realize Web Search is a telescoping framework [21] that iteratively refines the list of candidate documents considered for a user query. The telescoping framework divides Web Search into a multi-stage pipeline (c.f. Fig 1). The first stage is Selection service that addresses the problem of efficiently evaluating massive web indexes to select document candidates. This evaluation considers a multitude of document quality signals, which include search keyword matching, semantic similarity, past popularity, and

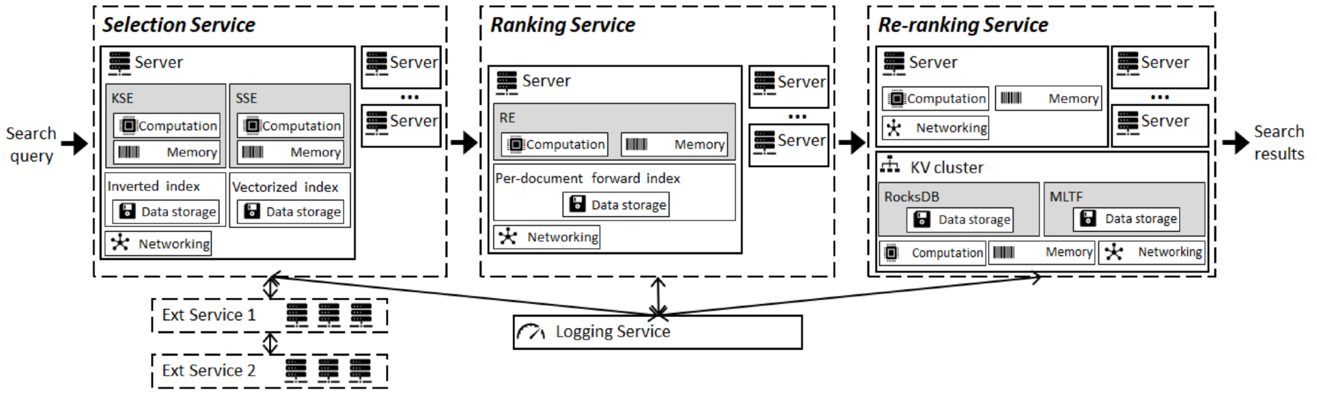

Figure 1. Like modern systems, Web Search is modularized into multiple stages. Interactions happen among services and service components. AutoSys has unified the design and operation for several critical Web Search engines: *KSE* (Keyword-based Selection Engine), *SSE* (Semantics-based Selection Engine), *RE* (Ranking Engine), *RocksDB* key-value store engine, and *MLTF* (Multi-level Time and Frequency) key-value store engine.

so on. Then, Ranking service needs to predict the relevance of each document candidate to user intent, and orders candidates accordingly. Interestingly, ranking algorithms receive frequent updates and tweaks to better adapt to user search interests. Finally, Re-ranking service adds and ranks relevant candidates of additional content types (e.g., images).

Realized gains from learning-augmented design. One decision-making opportunity that Web Search presents to learning is system behavior comprehension for configuration tuning. Decisions in modern systems can be abstracted as a function of controllable system knobs that include software logic parameters (e.g., the size of threading pools and caches), hardware configurations (e.g., CPU features), actions of an execution sequence, engine selections, and even hyper-parameters of ML/DL algorithms and models.

As Web Search has continuously evolved over the past decade, we highlight major optimization achievements enabled by learning – ***(1)*** learning reduces KSE engine's 99-percentile latency by another 19.4% - 29.7%, and CPU utilization by another 9.0% - 11.0%, as compared to expert-tuned counterparts. ***(2)*** Learning dynamically selects actions for individual steps of SSE engine's execution sequences, and achieves a 20.0% reduction in latency while keeping the search relevance score (or NDCG [2]) the same. ***(3)*** Learning improves RE engine's NDCG score by another 3.4%, as compared to expert-tuned counterparts. ***(4)*** For RocksDB key-value (KV) engine, learning achieves a comparable throughput while reducing months of manual efforts to two days of automatic tuning. ***(5)*** Learning reduces MLTF KV engine's 99-percentile latency by another 16.8%, as compared to expert-tuned counterparts.

2.1 Sources of System Complexity

Heterogenous classes of decision-makings. Different parts of modern systems can impose heterogenous classes of decision-makings: application logic, system algorithms, and system configurations.

Application logic implements features that fulfill user requirement, 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.

The infrastructure implements system algorithms that make decisions to better support applications 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.

Although system operators might have access to all knobs, identifying the optimal knob configuration requires them to comprehend the impact of combinatorial possibilities.

Multi-dimensional system evaluation metrics. Modern systems have increasingly been using multiple metrics to capture the system behavior. Optimizing multiple metrics can be non-trivial if these metrics have different (and potentially conflicting) goals. In some cases, system designers follow a conventional rule: improvement in some metrics must not cause other metrics to regress. Furthermore, modern systems can have meta-metrics that aggregate measurements over a time period (e.g., the weekly user satisfaction rate for Ranking service) or over a set of metrics. Reasoning about changes of multiple metrics quickly becomes painstaking for humans, as the number of evaluation metrics increases.

Interactions between subsystems and components. Subsystems and components form an integrated view of a large-scale modern system, as such an independently optimized subsystem might not necessarily benefit the greater good of the entire system. We illustrate with Selection service, where a new algorithm may increase the number of highly relevant pages being selected but also the number of spam pages. If the subsequent Ranking service does not update its algorithms accordingly, the display of spam pages can hurt user satisfaction.

2.2 Sources of Operation Complexity

Environment diversity and system dynamics. With the widespread adoption of cloud computing, the execution environment of many modern systems is structurally similar to a cluster of machines in datacenters. Optimally designing and operating systems under environment diversity and system dynamics can be challenging.

First, not only do hardware upgrades happen frequently, they might be rolled out in phases [27], and server resources can be shared with co-located tenants. Therefore, it is possible that subsystem instances and components face different resource budget. An example is how we changed the in-memory caching mechanism design, according to I/O throughput gaps between memory and mass storage medium for different data sizes.

Second, modern systems have increasingly adopted tighter and more frequent software update cycles [25]. These software updates range from architecture changes, feature updates, bug fixes, to even data. For example, Selection service has bi-annual major revisions to meet the increasing query volume and web document 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.

Workload diversity and dynamics. Workloads determine a system's runtime execution, which is a crucial signal in system design and operations. Workloads can have temporal dynamics that are predictable and unpredictable – we illustrate with how search keywords can change with national holidays and breaking news, respectively. Furthermore, individual subsystems (or components) can act on different features of the same workload. For example, unlike Selection, Re-ranking service is triggered only by certain query keywords such as celebrity names.

Non-trivial system knobs. Modern systems can expose a large number of controllable knobs, which include software logic parameters, hardware configurations, actions of an execution sequence, engine selections, and so on. Although knobs should be set with the prior knowledge of runtime workloads and system specifications [36], manual tuning efforts cannot scale with modern systems, especially in the presence of system and workload dynamics. In addition, modern systems can have a mix of different knob types: continuous numbers, discrete numbers, and categorical values. Finally, knobs can have dependencies [38]; i.e., the effect of one knob depends on the setting of another knob.

3 Learning-augmented System Design

The previous section discusses factors that push the design and operation complexity of modern systems beyond human reasoning and heuristic efforts. Our years of experience with Web Search suggests that adopting learning-augmented design goes beyond choosing off-the-shelf ML/DL algorithms. Instead, it requires a principled approach towards making ***(1)*** modern systems inherently learnable and ***(2)*** ML/DL learning manageable.

3.1 Principles on Making Systems Learnable

***P1:* Well-specified interfaces that abstracts system details for generality, and exposes system behavioral features for learning.** Modern systems selectively expose programming interfaces for functional needs, rather than controllable experiments. As a result, it is not easy for ML/DL-driven components to cross the system boundary, to complete the closed loop of system actuations and feedback. The problem exacerbates when we consider heterogeneous classes of decision-makings (c.f. §2.1) – without well-specified interfaces, ad-hoc solutions are implemented for each scenario separately. One common issue that we encountered during model training is how some systems distribute configurable parameters and error messages over a set of not-well documented configuration files and logs [29]. This approach can have undesirable consequences. First, directly modifying configuration files means that the system cannot enforce constraints or provide feedbacks on ML/DL actuations (e.g., invalid combinations of parameter values). Second, parsing raw logs can be time-consuming, especially if system components disagree on a unified logging format or excessively log [16]. This is common in systems where subsystems are maintained by different teams.

For modern systems to be inherently learnable, there needs to be well-defined control interfaces. In addition to being unified accessors to controllable knobs for ML/DL actuations, these interfaces should abstract observable system metrics and logs in a way that facilitates learning. While raw stack traces and core dumps can traditionally be helpful for system engineers and operators, they are not learnable. Instead, control interfaces should expose system behavioral features, to capture the cause (i.e., ML/DL actuations) and the effect (i.e., time series system metrics) in the temporal domain – this format allows the learning to carry out in a supervised fashion. Finally, data outliers should be removed to avoid skew or mislead the training process.

***P2:* Monitored ML/DL actuations for system failure prevention and detections.** Being stochastic in nature, ML/DL inference has some degrees of uncertainty, especially when models are not sufficiently trained to match the complexity of system component interactions (c.f. §2.1) and non-trivial system knobs (c.f. §2.2). This uncertainty can lead to learning-induced system failures or suboptimality. While failures are not unique to learning-augmented systems [6], handling them requires a different approach for the following reasons. First, while ensuring correctness has traditionally been a process of reviewing deterministic logic written by humans, many ML/DL models are not easily interpretable. ML/DL models mathematically encode insights learned from the training data, and it is not easy to determine how changing individual model weights would impact the probabilistic logic. Second, as models autonomously learn from the training data, it is difficult to construct a complete data set that would guarantee a model to fully learn a particular concept.

Since it is hard to guarantee the ML/DL inference correctness, modern systems need mechanisms to validate and monitor ML/DL actuations. The validation can take the form of basic sanity checks and sophisticated checks on inference assumptions such as the confidence interval size. On the other hand, monitoring system metrics can assist in detecting learning-augmented failures.

3.2 Principles on Making Learning Manageable

***P3:* Modularized learning for reducing learning complexity.** While it is possible to learn an entire cloud system with one monolithic model, doing so can impose a significant learning cost. First, suppose system designers properly expose all controllable knobs and observable metrics, the learning cost of modeling their relationship actually increases with the total number of possible combinations – if a system has m parameters and each parameter can take one of n values, the space to learn has a size of $O(n^m)$. Second, cloud systems exhibit various types of dynamics (c.f. §2.2), as they evolve or runtime environment changes. Unfortunately, changes to one software module can alter the system behavior, hence invalidating assumptions over which the monolithic model was previously trained.

One consideration in learning large modern systems is to break the learning task down to more manageable pieces. With modularized learning, a model learns the behavior of only a subsystem or a component. Conceptually, modularized learning shares the benefit of adaptivity with the software design principle of modularity [26]. In our case of Web Search, we learn to separately optimize individual subsystems, rather than the entire end-to-end system in one shot.

The challenge of modularized learning lies in determining the modularization that best minimizes the overall learning costs. Although one solution is to naively match each software module with one model, doing so neglects the fact that software modularity is typically based on the criteria of code functionality and maintainability, instead of learning. One issue arises when a system whose components consider metrics that are conflicting – we illustrate with Selection stage whose metrics include search latencies and relevance. While it is possible to isolate Selection's KSE engine and inverted index, the number of documents that KSE chooses to select improves search relevance, but it also depends on the inverted index query latencies. Furthermore, software modules can have data dependencies (i.e., one module's output becomes another module's input). Without knowledge on how subsequent modules use a module's output, it might be difficult to reason about the global optimum.

P4:* Resource management for system exploration and model maintenance.** Building up the key predictive capability to optimize modern systems requires resources to run two types of jobs: ***(1) system exploration benchmarks, and ***(2)*** ML/DL model training. Maintaining learning-augmented systems can spawn a large amount of these jobs, for the following two reasons. First, to match the system scale and complexity, modularizing learning can divide a cloud system into a large number of ML/DL models. Second, cloud system dynamics imply that model training should not be considered as a one-shot process. If these dynamics change assumptions over which models were previously trained (c.f. §2.2), it is necessary to re-train or incrementally update these models.

For learning-augmented systems to better maintain models, there need to be mechanisms to manage limited resources for running pending system exploration benchmarks and ML/DL model training jobs. Specifically, in the case of learning-augmented systems, resource management consists of two distinct tasks – prioritizing system exploration benchmarks based on how they are expected to help the predictive model accuracy, and scheduling jobs of heterogenous requirements for the resource pool.

4 AutoSys Framework

As individual target systems impose different classes of decisions and requirements, applying learning-augmented design can involve scenario-specific considerations and changes. Interestingly, from our years of experience with different parts of Web Search, we observed similarities in the software development. On this basis, we introduce *AutoSys*, a framework of learning-augmented design to unify this development process. Figure 2 shows three major components in AutoSys: Training Plane, Inference Plane, and target systems.

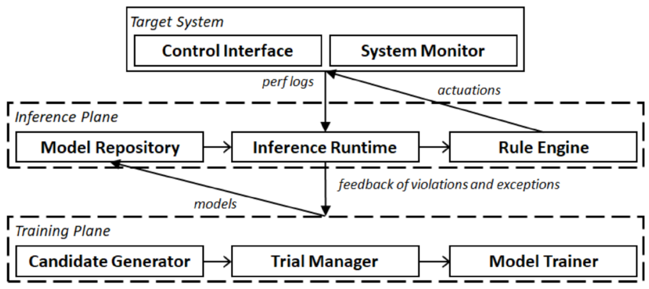

Figure 2. AutoSys framework.

4.1 Training Plane

Training Plane addresses model training requirements introduced by learning-augmented systems (c.f. ***P3*** and ***P4***). In addition to training new models as system operators instruct, it continuously decides whether existing models should be redesigned and updated according to Inference Plane's feedback of system runtime logs and failure debugging.

Candidate Generator. Candidate Generator iteratively generates training inputs towards learning regions with low inference accuracy or high inference uncertainty. This concept of iterative strategy is similar to Bayesian optimization [31], and implemented by various algorithms [7, 15, 18] hosted by AutoSys. Model fuzzing is also similar to software fuzzing [8], where the goal is to efficiently identify program inputs that can trigger software failures.

Trial Manager. Trial Manager abstracts each candidate as a Trial instance. 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)*** scheduling meta-data including priority and resource requirements (e.g., the number of GPU cores). Trials can impose heterogenous resource requirements to support their corresponding decision-making scenarios. For 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 has 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 the number of CPU cores.

Model Trainer. In the case of learning-augmented systems, model architecture design should consider ***(1)*** inference accuracy, which determines how much system components/tasks can benefit from learning, ***(2)*** inference cost, which impacts the gain in metrics such as system throughput, and ***(3)*** training cost, which determines the feasibility of adapting system dynamics. While these considerations partially align with neural architecture search (NAS), we highlight two differences. First, existing NAS efforts [10, 28, 33] mainly focus on engineering an optimal model for a narrow selection of tasks and datasets. Second, they always assume to build a new model, rather than leveraging the knowledge of system state changes to fine-tune existing models.

It is also crucial for Model Trainer to deal with noisy data points. While ignoring this noise can impact the training data quality (hence the learning accuracy), modeling it as additive white Gaussian noise does not properly deal with non-Gaussian noise sources in the wild (e.g., CPU scheduling and OS updates). Instead, AutoSys hosts algorithms [18] that proactively identify potentially noisy data points.

4.2 Inference Plane

Inference Plane addresses decision-making requirements due to the learning-augmented design (c.f. ***P2*** and ***P4***).

Inference Runtime. With modularized modules, a system is characterized by a set of models. And, inferences over these models follow the paradigm of dataflow programming: a directed graph of data flowing between operations. In our case, operations are models, and directed edges are wirings connecting a model's output to other models' inputs. Furthermore, numerical constraints can be specified as assertions over a group of model inputs and outputs. An usage case of constraints is describing resource sharing among tenants co-located on the same server. We note that Inference Plane should also translate the final output into separate actuations for individual system modules.

Rule Engine. To harden against potential learning-induced failures, AutoSys wraps ML/DL decision algorithms with a deterministic rule-checking engine. Unlike ML/DL models, explicit rules are authored by operators, and they are human-readable. Our instantiation of AutoSys implements blacklists, and each rule specifies conditions of a violation to catch.

First, rules can do basic sanity checks – in addition to validating parameter value constraints, system operators can prevent applying certain system states that have been known/reported to cause failures. Second, rules can check knob dependencies – an example is the multi-tenant setup where the total memory allocated to all tenants must not exceed a budget. Third, rules can check for discrepancies between predicted and actual performance measurements. Since causes for large discrepancies include inaccurate ML/DL inference and unexpected system behavior, Training Plane can subsequently train and update ML/DL models.

4.3 Target System

The target system is wrapped by a control interface over which Inference Plane can actuate system components, and system runtime can share structural data of state information and performance metrics (c.f. ***R1***). System Monitor analyzes real-time streams of system runtime logs to detect potential health problems. Our instantiation of AutoSys incorporates both rule-based detection and unsupervised anomaly detection. In the case of Web Search, the former encodes hard cases such as the lower bound of performance metrics, and the latter monitors for unexpected correlations among multiple metrics. Finally, failure recovery is configured with rules that specify per-failure actions to restore good system states, e.g., reboots.

5 Related Work

Infusing ML/DL into system designs is emerging as a new interdisciplinary area attracting a significant amount of research efforts and industry investments. Unlike AutoSys promoting a learning-and-system co-design, many efforts heavily focus on system challenges. Anticipating growing system scale and complexity, Self-* [12] stated a vision of autonomic computing that satisfies a collection of "self-*" properties, and proposed a conceptual model. Recently, Berkeley shared their views of system challenges for artificial intelligence (AI) [32], and there is a synergy between this view and our view – some AutoSys framework components such as continual learning require support from AI infrastructure.

Recent efforts of building learning-augmented systems motivate the need for a learning-and-system co-design. They demonstrate limitations of the current methodology such as the assumption of static databases. These efforts include learning index structures and memory access patterns [14, 17], optimizing data query evaluations [24], performance tuning [4, 18], adaptive caching [5], placing deep learning computational graphs onto hardware device [22, 23], etc.

Some AutoSys components are inspired by decades of research and experience in the system community. Related to control interfaces, interfaces and methods for controlling and exploring systems state are used by implementation-level model checking (e.g., MaceMC [13] and Modist [35]). Fuzz testing has been effectively used in generating inputs to induce unexpected software behavior [8, 19, 37], and there is a rich literature on software testing and system debugging. Inspired by composable AI [32], we propose modularized learning to scalably model large-scale systems.

Finally, some AutoSys components are inspired by research in the ML/DL community. Examples include online learning [9], continual learning [30], etc.

6 Conclusion

To realize the full potential of learning-augmented systems, we propose AutoSys, a framework of learning-and-system co-design to address operational and learning challenges. Drawing from our experience with Web Search, we discuss these challenges and formulate requirements that AutoSys should enable. Looking forward, we believe that AutoSys will accelerate the development of learning-augmented systems beyond Web Search, and further foster the paradigm shift in modern system design.

References

[1] Cloud TPU. https://cloud.google.com/tpu/, 2018.

[2] Eugene Agichtein, Eric Brill, and Susan Dumais. Improving Web Search Ranking by Incorporating User Behavior Information. In *SIGIR*. ACM, 2016.

[3] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 2017.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX, 2017.

[5] Suzanna Becker, Sebastian Thrun, and Klaus Obermayer. *Advances in Neural Information Processing Systems 15: Proceedings of the 2002 Conference*, volume 15. MIT Press, 2003.

[6] Theophilus Benson, Aditya Akella, and Aman Shaikh. Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services. In *SIGCOMM*. ACM, 2011.

[7] James Bergstra, Remi Bardenet, Yoshua Bengio, and Balazs Kegl. Algorithms for Hyper-Parameter Optimization. In *NIPS*, 2011.

[8] D. L. Bird and C. U. Munoz. Automatic Generation of Random Self-checking Test Cases. *IBM Systems Journal*, 1983.

[9] Leon Bottou and Yann Le Cun. Large Scale Online Learning. In *NIPS*, 2003.

[10] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understandingand Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*. ACM, 2017.

[12] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* Storage: Brick-based Storage with Automated Administration. Technical report, CMU, 2003.

[13] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP*. ACM, 2011.

[14] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning Memory Access Patterns. *CoRR*, 2018.

[15] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*. Springer, 2011.

[16] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding Customer Problem Troubleshooting from Storage System Logs. In *FAST*. USENIX, 2009.

[17] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*. ACM, 2018.

[18] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC*. USENIX, 2018.

[19] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Lyna Zhang, Borje Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *MobiCom*. ACM, 2014.

[20] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *SIGCOMM*. ACM, 2017.

[21] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. High Accuracy Retrieval With Multiple Nested Ranker. In *SIGIR*, 2006.

[22] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A Hierarchical Model for Device Placement. In *ICLR*, 2018.

[23] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. *CoRR*, 2017.

[24] Corby Rosset Damien Jose Gargi Ghosh Bhaskar Mitra and Saurabh Tiwary. Optimizing Query Evaluations using Reinforcement Learning for Web Search. In *SIGIR*. ACM, 2018.

[25] Iulian Neamtiu and Tudor Dumitras. Cloud Software Upgrades: Challenges and Opportunities. In *MESOCA*. IEEE, 2011.

[26] D.L. Parnas. On the Criteria To Be Used in Decomposing System into Modules. In *ACM Communication*. ACM, 1972.

[27] David A. Patterson. Technical Perspective: The Data Center Is The Computer. *ACM Communication*, 2008.

[28] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient Neural Architecture Search via Parameter Sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[29] Ariel Rabkin and Randy Katz. Static Extraction of Program Configuration Options. In *ICSE*. ACM, 2011.

[30] Mark B. Ring. CHILD: A First Step Towards Continual Learning. In *Machine Learning*. Springer, 1997.

[31] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, 2016.

[32] Ion Stoica, Dawn Song, Raluca Ada Popa, David A. Patterson, Michael W. Mahoney, Randy H. Katz, Anthony D. Joseph, Michael Jordan, Joseph M. Hellerstein, Joseph Gonzalez, Ken Goldberg, Ali Ghodsi, David E. Culler, and Pieter Abbeel. A Berkeley View of Systems Challenges for AI. Technical report, Berkeley, 2017.

[33] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware Neural Architecture Search for Mobile. *arXiv preprint arXiv:1807.11626*, 2018.

[34] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 2018.

[35] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*. USENIX, 2009.

[36] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP*. ACM, 2011.

[37] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. Systematically Testing Background Services of Mobile Apps. In *ASE*. ACM, 2017.

[38] Sai Zhang and Michael D. Ernst. Which Configuration Option Should I Change? In *ICSE*. ACM, 2014.