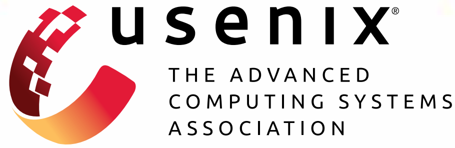


On Modular Learning of Distributed Systems for Predicting End-to-End Latency

Chieh-Jan Mike Liang, *Microsoft Research;* Zilin Fang, *Carnegie Mellon University;* Yuqing Xie, *Tsinghua University;* Fan Yang, *Microsoft Research;* Zhao Lucis Li, *University of Science and Technology of China;* Li Lyna Zhang, Mao Yang, and Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/nsdi23/presentation/liang-chieh-jan

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

On Modular Learning of Distributed Systems for Predicting End-to-End Latency

Chieh-Jan Mike Liang[‡] Zilin Fang[*] Yuqing Xie[∘] Fan Yang[‡]
Zhao Lucis Li[⋆] Li Lyna Zhang[‡] Mao Yang[‡] Lidong Zhou[‡]
[‡]*Microsoft Research* [*]*CMU* [∘]*Tsinghua University* [⋆]*University of Science and Technology of China*

Abstract

An emerging trend in cloud deployments is to adopt machine learning (ML) models to characterize end-to-end system performance. Despite early success, such methods can incur significant costs when adapting to the *deployment dynamics* of distributed systems like service scaling-out and replacement. They require hours or even days for data collection and model training, otherwise models may drift to result in unacceptable inaccuracy. This problem arises from the practice of modeling the entire system with *monolithic* models. We propose Fluxion, a framework to model end-to-end system latency with *modularized learning*. Fluxion introduces learning assignment, a new abstraction that allows modeling individual sub-components. With a consistent interface, multiple learning assignments can then be dynamically composed into an *inference graph*, to model a complex distributed system on the fly. Changes in a system sub-component only involve updating the corresponding learning assignment, thus significantly reducing costs. Using three systems with up to 142 microservices on a 100-VM cluster, Fluxion shows a performance modeling MAE (mean absolute error) up to 68.41% lower than monolithic models. In turn, this lower MAE allows better system performance tuning, e.g., a speed up for 90-percentile end-to-end latency by up to 1.57×. All these are achieved under various system deployment dynamics.

1 Introduction

Predicting cloud system performance is critical for improving the end-user experience. While this problem has been traditionally addressed with analysis and handcrafted performance models, an emerging trend is to incorporate machine learning (ML) techniques for performance modeling [7,9,23]. Such approaches [2,3,6,13,14,20,21,34,41] typically use *monolithic* ML models, to predict the performance (e.g., user request latency), given configurable knobs of the system component (e.g., cache size) and observable states (e.g., request rate).

In recent years, the microservice architecture has gained popularity in building distributed cloud systems [1,12,18,25,40]. Its per-service flexibility enables continuous integration and continuous delivery (CI/CD), and per-service horizontal scaling (e.g., replication) and vertical scaling (e.g., capacity adjustments) can handle load dynamics. Interestingly, the monolithic approach of modeling the entire system could still be applicable to such distributed systems, and doing so frees operators from explicitly modeling service dependencies.

Unfortunately, our first-hand experience at Microsoft suggests inherent limitations in effectively learning distributed system's *end-to-end* performance. There is a need to continually adapt performance models to the *deployment dynamics*. As services are independently scaled and replaced over time, deployment updates become frequent operations.

Continually updating monolithic models can incur significant time costs, especially if deployment dynamics are handled in an ad-hoc way. First, collecting a sufficient amount of training data can be time-consuming, as new system dynamics can take minutes and even hours to be fully warmed-up and stable [39]. Second, designing and training a new model can also be time-consuming, even with the help of automation tools [4,19,27]. For example, a system evaluated in §5 has 142 microservices, and requires a performance model that considers 1,034 service knobs and states. Collecting sufficient data points to train such a complex monolithic model takes us ∼46 hours, and model training takes additional ∼24 hours. Such a practice hinders the practicality of monolithic approach, for performance tuning in the real world.

Given the above challenges, we advocate *modularized learning* for microservice-based distributed systems. Our key observation is the locality of deployment dynamics, where changes happen at the granularity of system components (e.g., microservices). So, modularized learning models the performance of each service individually, and carefully composes these models on-demand to follow deployment dynamics.

Fluxion is a framework that realizes modularized learning

This work was done when Zilin Fang, Yuqing Xie, and Zhao Lucis Li were interns at Microsoft Research.

to model end-to-end latency while handling system deployment dynamics efficiently and effectively. Fluxion introduces *learning assignment*, an abstraction to model each service in a distributed system. Learning assignment is instantiated on each service instance to model its performance metrics (e.g., latency). It can accommodate any service and different ML modeling techniques (e.g., DNN or Gaussian process). Learning assignments can be composed into an *inference graph* to model a large complex system, similar to how services are composed into an end-to-end system. Changes in a service of the system only induce modeling errors in the corresponding learning assignments. Since the configurations and internal logic of other services remain the same, their corresponding learning assignments along with their modeling accuracy also remain unchanged. Therefore, Fluxion only needs to update the learning assignments corresponding to the changed service, thus significantly reducing the costs.

Three unique characteristics enable learning assignment to handle system deployment dynamics effectively. First, to capture the impact from other services, learning assignment defines external performance dependencies as part of modeling inputs. This enables the composability — multiple assignments can be composed into an inference graph, to follow service dependencies of a complex distributed system on the fly. Second, instead of considering only the performance metric of interest (e.g., p90 latency), learning assignments can take in a spectrum of performance metrics from upstream assignments (e.g., p50–p99 latencies) in inference graph. This allows a learning assignment to better observe (and capture) the impacts of system deployment dynamics from upstream assignments. Finally, to capture the temporal system dynamics, a learning assignment can host one or more ML models trained in different time periods and scales.

In summary, this paper makes the following contributions. ***(1)*** We propose a modular approach to modeling end-to-end latency for complex and dynamic distributed systems, exemplified by microservice systems. ***(2)*** The abstraction of learning assignment and the resulting inference graph effectively capture intrinsic system dynamics, as well as dependencies among services. ***(3)*** We conduct comprehensive experiments to demonstrate the significantly superior performance of Fluxion over existing approaches, under various system deployment dynamics. In some microservice systems spanning 100 VMs, Fluxion's performance model exhibits up to 68% lower MAE (mean absolute error). In turn, this enables better system performance optimization, or p90 latency speed up by up to 1.57$\times$ over the use of baselines. At the same time, Fluxion reduces the model training time by up to 99.98%.

2 Background and Motivations

2.1 Performance Prediction and Modeling

Performance models predict the *end-to-end system performance* (e.g., user request latencies), given *observable states*

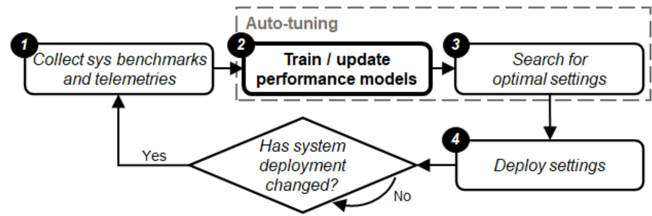


Figure 1: General workflow of auto-tuning. Fluxion focuses on *step #2*, or the efficiency in adapting performance models to system deployment dynamics.

(e.g., per-service request rates) and *configurable knobs* (e.g., per-service cache size and thresholds). Performance models can be analytically constructed with mathematical formulations, but doing so does not scale well with the size or complexity of large-scale distributed systems. A typical microservice system can have hundreds (and even thousands [40]) of services, and requests can traverse 40+ services [25].

Learned performance model. Advances in machine learning (ML) enable performance modeling to be learned, with regression techniques such as Gaussian process and DNN. Like previous efforts that model monolithic systems [2,3,9,13,21,23], it is possible to treat an entire microservice system as a black-box. To match system deployment scale, ML models can monolithically grow in size (e.g., adding DNN neurons). Furthermore, black-box modeling eliminates the need to explicitly consider service interactions and dependencies.

For training performance models, each training data point consists of inputs (i.e., per-service knob settings and observable states) and an output (i.e., a system performance measurement). Data are collected through benchmarks in controlled environment (e.g., testbeds or isolated sections in production), or telemetries in production. Trained models may be evaluated with testing data, which are collected in the same fashion. One common evaluation metric for performance modeling is MAE (mean absolute error) [37], or the average magnitude of errors in a set of test predictions made by the model.

Performance auto-tuning scenario. As Fig 1 illustrates, performance models can drive auto-tuning [2, 3, 21–23]. The goal is to guide non-system-experts to set system knobs, to optimize for a performance metric. Auto-tuning relies on optimizers (*step #3*), which iteratively search for global optimum in the modeled space. Each iteration algorithmically selects new knob setting (for performance model to predict), based on performance predictions from previous iterations.

2.2 High Costs to Maintain ML Models

We observe significant time costs in continually keeping ML-based performance models updated to system deployment dynamics. Deployment dynamics make performance models drift over time and impact modeling accuracy, hence auto-tuning outcomes (c.f. §5). Main sources of deployment

dynamics include ***(1)*** system scaling, i.e., replicating or reclaiming service instances, and ***(2)*** continuous integration and delivery (CI/CD), i.e., replacing existing services.

We discuss the breakdown of time costs below.

***Problem #1*: Collecting training data points for performance modeling can be time-consuming.** As the model complexity grows, the number of training data points necessary also grows. However, each benchmark requires the system to be fully warmed-up and stable [39]. In our cases, one benchmark can take up to 15 minutes, and collecting sufficient data points can require ∼46 hours.

The problem exacerbates as distributed cloud systems require a significantly higher model complexity than previously considered. Unlike monolithic systems with ∼20 knobs and states to consider, this number can quickly add up to hundreds and thousands for microservice-based systems. For example, Train-Ticket [31] has 41 services. At initialization, it has 242 model inputs: 148 configuration knobs (e.g., MongoDB's `eviction_dirty_target`), 94 states (e.g., Docker's `cpu-limit` and per-service requests per second). As Train-Ticket scales-out by a factor of 6, there is a total of 210 service instances, and 1,020 model inputs.

***Problem #2*: Designing and training new models for performance modeling can be time-consuming.** Keeping performance models updated goes beyond simply fine-tuning models with recent benchmarks. In many cases, the required changes lie in the model structure. One motivating example is how replicating services essentially alters the deployment, with respect to the available knobs, service states, and service dependencies. As a result, we need a new model of different input dimension and even different modeling technique.

Although AutoML toolkits can automate this process to some extent, our experience suggests that it can take at least 20 hours to produce a reasonably accurate ML model for performance modeling. Furthermore, it is not feasible to pre-train all monolithic models for all possible deployment setups. Since services can be independently replaced and arbitrarily scaled, the number of possible deployment setups is unbounded.

2.3 Modularized Learning

The principle of modularity has been proven in engineering scalable and elastic systems. It provides opportunities to realize ML-based performance modeling in an agile and accurate way. Instead of monolithically modeling the end-to-end latency of distributed systems with one performance model, we propose *modularized learning* that breaks down this model to align with a deployment's modular units.

Challenges. To practice modularity, it is natural to independently model each system component, e.g., a microservice. Given system components can have vastly different configuration knobs and states, different types of ML models can be chosen for individual system components. The key challenges are: ***(1)*** to represent different system components, possibly using different ML techniques, with a *consistent interface*; and ***(2)*** to have a *composability criteria* that combines these component representations into an end-to-end ML model for performance modeling.

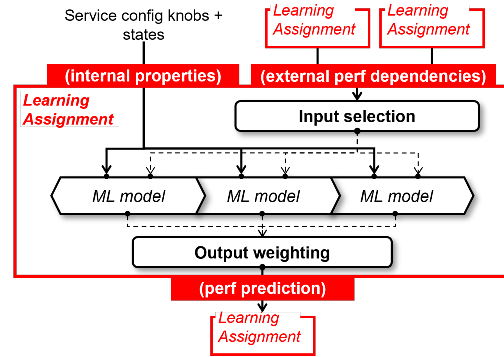


Figure 2: Learning assignment is a wrapper for ML models. It has a consistent interface for composability.

3 Fluxion Framework

Fluxion is a framework that realizes modularized learning for distributed systems. To address the two challenges discussed in §2.3, Fluxion introduces *learning assignment* to abstract away model and component heterogeneity and provide a unified interface to model service-level latency (§3.1). Moreover, Fluxion presents *inference graph* to dynamically compose assignments into an end-to-end performance model (§3.2).

3.1 Learning Assignments

A learning assignment is a basic modeling unit. It hosts one or more ML models that collectively model a modular unit in distributed systems. Since system deployment dynamics typically happen at the unit of services (e.g., scaling and replacing services), assignments are instantiated on a *per-service-instance* and *per-performance-metric*[1] basis. When deployment dynamics happen, this mapping of modular units allows Fluxion to localize changes to some learning assignments. Fig 2 illustrates the internal structure of an assignment, and we elaborate the details next.

Interface. The learning assignment interface is designed to abstract service-level performance for ML models. Modeling individual services is different from modeling the entire system monolithically. The former needs to take service dependencies into account. E.g., a service's observed latency inherently includes the latency of its downstream services [10].

Therefore, in addition to internal configuration knobs and internal observable states, the assignment inputs further include external performance dependencies (e.g., downstream service latency). The assignment output is a service-level

[1]E.g., p50 and p90 latencies require two learning assignments.

performance metric. This interface addresses *challenge #1* — the consistent service-level performance metrics as input and output, together with the heterogeneous internal states and knobs, are sufficiently general to model different services and to host ML models of different modeling techniques, and a learning assignment's output can be connected to another assignment as an external performance dependency.

Exposing a spectrum of performance metrics. To predict the end-to-end latency more accurately, a learning assignment may expect the external performance dependencies to be a spectrum of performance metrics from dependent services. For example, to predict end-to-end p90 latency, a learning assignment may require p50–p90 latencies of the downstream services (and even their CPU utilization and disk throughput), so as to decide which metrics are appropriate for consideration. This allows a learning assignment to better observe the extent of impacts that system deployment dynamics impose on its downstream services.

However, not all performance metrics are *highly* relevant to the one being predicted. An example is the bottom-percentile latencies *vs.* the top-percentile latencies. Including irrelevant performance metrics incurs additional costs. The first is the training costs. As unnecessary inputs add noises to the training dataset, some ML models would need more data points to distinguish and learn from the relevant inputs. And collecting training data points can be time-consuming (c.f. §2.2). The second is the unnecessary learning assignments introduced to predict these irrelevant metrics.

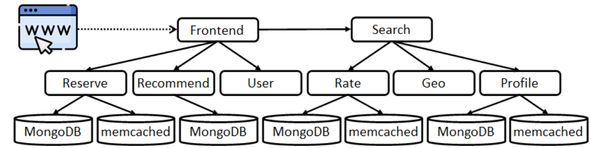
To this end, learning assignment introduces "input selection" (shown in Fig 2) to prune unnecessary metrics and the corresponding learning assignments. The problem of input selection can be formulated as follows. Given a set of k learning assignments (m_k) as inputs to a specific learning assignment, we want to find the k-dimensional binary weight vector (w_k^*). w_k^* should minimize the prediction error of an weighted sum of m_k, over a batch of n data points (inputs X, and outputs Y). This is formulated as the following equation:

$$w_k^* = \underset{w_k \in W}{\operatorname{argmin}} \left(\sum_{i=1}^{n} Y_i - f(w_k, m_k(X_i))\right)^2. \quad (1)$$

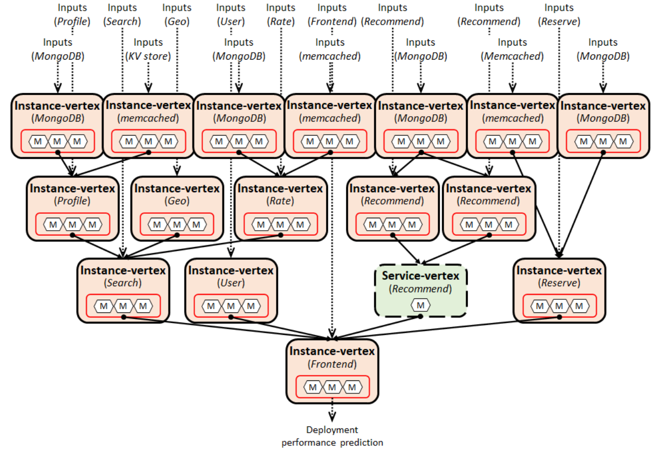
The goal is to search for w_k^*, or the optimal k-dimensional binary vector. f is a predefined function that aggregate outputs of k learning assignments, and it can be hand-written code or non-linear functions such as neural networks. We can expand f as follows:

$$f(w, m(X)) = \frac{\sum_{i=1}^{k} w^{(i)} m^{(i)}(X)}{\sum_{i=1}^{k} w^{(i)}}. \quad (2)$$

In practice, k can be large. For example, RocksDB has $k = 99$ performance metrics. To select three relevant metrics need to search 941,094 possible combinations. To find a solution effectively, §4.1 presents one generic approach in the current implementation of Fluxion.



(a) Execution graph



(b) Inference graph

Figure 3: A simplified inference graph of a microsevice application, Hotel Reservation. Vertices represent services, and directed edges capture performance dependencies among services. Graph inputs include configuration knobs and observable service states. (M) represents models. In this example, the Recommend service is scaled out to two instances.

Capturing system's temporal dynamics. A learning assignment can host multiple ML models trained in different time periods and scales. Doing so promotes the reuse of previously learned models, in order to better capture temporal dynamics and predict recurring patterns. An example is the daily variations in incoming request rates. In this case, we can train a new model with data points collected each day.

Learning assignment introduces "output weighting" (shown in Fig 2) to weight over outputs from different time periods/scales and reduce all models' outputs to one succinct value. The formulation of output weighting is similar to that of input selection, except that k assignments are changed to k internal models and w_k^* here is a k-dimensional vector of continuous numbers between 0 and 1 inclusively. And the output is computed as the weighted sum of all k models' outputs.

Similarly, to find a good combination of weights is computationally expensive. §4.1 presents one generic approach in the current implementation of Fluxion.

3.2 Inference Graph of Learning Assignments

Inference graph is a set of interconnected learning assignments. It represents the performance modeling of an end-to-

end system. From external user's perspective, inference graph has the same input/output semantics as monolithic models. In other words, inference graph exposes the following performance modeling inputs: all services' configuration knobs and observable states. Its output predicts for an end-to-end performance metric: the end-to-end tail latency in our case. Inference graph addresses *challenge #2*, i.e., the composability criteria to compose learning assignments.

3.2.1 Inference Graph Construction

Fig 3 illustrates a simplified inference graph for a microservice system, Hotel Reservation [11].

Graph vertices. An inference graph has two types of vertices: instance-vertices and service-vertices (c.f. Fig 3b). Instance-vertices correspond to the learning assignments modeling the performance of service instances. They expose learning assignment inputs (e.g., service configuration knobs, observable states, and external performance dependencies) and output (e.g., a service-level performance metric). Service-vertices aggregate instance-vertices that model the same service and performance metric. In Fig 3b, the Recommend service (marked in dash-line) is scaled out to two instances.

Graph edges. Graph edges are directed dataflows, and they satisfy services' external performance dependencies. To draw the edges correctly, we leverage the observation that the performance dependency of two services is the *reverse* of their execution dependency. Since an upstream service invokes RPC calls to its downstream service, the upstream service's latency would depend on the latency of downstream service. Hence an edge should be connected from the learning assignment (i.e., vertex) representing the downstream service to the one representing the upstream service (i.e., the reverse direction of the service execution order).

Handling deployment dynamics through inference graph updating. Since graph vertices and edges have a strong correspondence to services in the system deployment, orchestration-induced dynamics can be localized to certain regions of the inference graph. This implies that other regions can remain unchanged. Fluxion provides APIs to update graph for common orchestration operations (c.f. §4.1.2). First, scaling-out a service is conceptually equivalent to replicating the corresponding instance-vertices, to match the number of deployed instances. Similarly, scaling-in a service removes some of the corresponding instance-vertices. Second, upgrading a service (or even a migration from MySQL to PostgreSQL) is conceptually equivalent to replacing old service's instance-vertices with new service's.

3.2.2 Graph Inferencing to Predict End-to-End Latency

Graph inferencing is performed through graph traversal. The traversal starts from graph vertices that do not have external performance dependencies, or services that do not invoke any downstream services (e.g., the top vertices in Fig 3b). At each vertex, the learning assignment output metric is computed with its ML models. Following graph edges, the output is then passed to subsequent vertices as an external performance dependency. The traversal stops at the last vertex in the graph, or typically the gateway service in a deployment. The output of this last vertex is the output of the graph, and it predicts the end-to-end system performance.

3.2.3 Inference Graph Re-training

Inference can be performed immediately after graph is composed, but in cases where the prediction error rate is high, re-training can mitigate the problem. Fluxion identifies two major error sources. And, it can effectively reduce the re-training costs by taking advantage of the inherent modularity in the graph, rather than re-designing and re-training the entire monolithic model. Particularly, graph prediction errors can be traced back to some subsets of vertices.

Graph error source #1: New learning assignments. New learning assignments are required when new services are deployed to microservice systems or existing services are being updated. From our experience, their high MAE is typically due to insufficient training, especially by non-ML-experts. This case can be mitigated with the use of AutoML toolkits. Another possibility is service-vertices. Since they aggregate instance-vertices, scaling out/in a service requires them to have a new model with a new input dimension. Localizing new assignments is trivial, and the information is available through recording graph manipulations over time.

Graph error source #2: Unforeseen prediction inputs. Unforeseen prediction inputs can happen when the system receives unforeseen request types, or unexpected request ratios. This is a situation monolithic models also have to handle. As different requests stress different service-to-service execution paths, a service can observe unfamiliar states (e.g., requests per second and CPU utilization) or even downstream service performance. In the monolithic approach, the solution is to re-train the monolithic models. Modularized learning gives new optimization opportunities. Fluxion only needs to retrain the learning assignments being impacted to better handle these prediction inputs.

Identifying the impacted assignments, i.e., vertices in the inference graph, to address *error source #2* can be non-trivial. This step is not simply about identifying vertices whose learning assignments have the largest prediction MAE. A well-trained learning assignment can still output unexpected predictions if it receives erroneous inputs from another. The reason is that local errors of individual assignments can propagate. This is an artifact of how the inference graph output is a function of all its models. As graph traversal passes the prediction of a learning assignment to another assignment,

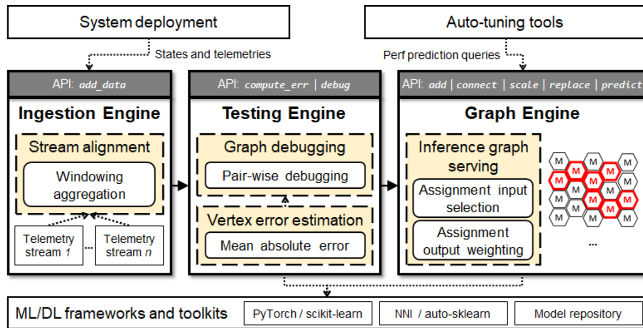


Figure 4: Fluxion architecture overview. It is implemented as three engines. Graph Engine hosts inference graphs. Testing Engine implements features for graph uncertainty estimation and graph debugging. Ingestion Engine ingests and buffers streams of telemetries, for ML model training and testing.

prediction errors propagate and accumulate.

Unfortunately, it is not trivial to analytically derive formulas describing the error accumulation. As ML models apply non-linear transformations to inputs, this non-linearity also transforms input errors to prediction errors. Furthermore, a learning assignment can have multiple model inputs. Not only can different model inputs be weighted differently, but they can also be of predictions from different assignments.

Pair-wise debugging approach. Fluxion implements a pair-wise debugging approach for *error source #2*. This is an iterative process, and each iteration selects one vertex. The core idea is to evaluate the likelihood that the vertex's prediction error is due to its own learning assignment or parent vertices.

System operators provide a test dataset, which contains recent deployment benchmarks. Fluxion then computes the test MAE for each graph vertex. And, it computes the Pearson correlation, or $r(mae_{parent},\ mae_{child})$, for each directly connected pair of graph vertices. With all rs computed, the debugging procedure follows a depth-first traversal. It starts from the last graph vertex (i.e., the vertex that produces the graph output), and performs the following steps — ***(step #1)*** with respect to the current vertex, we rank all parent vertices by their r in descending order. ***(step #2)*** If the top-ranked parent vertex has an r larger than 0, we traverse the edge to it and repeat step #1. ***(step #3)*** Otherwise, if the top-ranked parent vertex does not have an r larger than 0, we stop and return the current vertex as the debugging result.

4 Implementation

Current implementation has 13,012 SLOC, supports PyTorch and scikit-learn models, and integrates NNI [27]. Fig 4 shows an overview. A model repository stores models in serialized form and retrieves them with the unique model ID.

4.1 Graph Engine (GE)

GE serves inference graphs. During auto-tuning, the optimizer queries GE for performance predictions, just like how it would query monolithic models. GE implements the input selection strategy and the output weighting strategy (c.f. §3.1), and it offers APIs to manipulate inference graphs.

Input selection strategy. Our implementation is based on Thompson sampling with Beta distribution [30]. For a learning assignment, each external performance dependency has a Beta distribution, to estimate the probability of being selected for w_k^*. The probability density is governed by two variables, α and β. A larger α increases the mean probability, and a larger $(\alpha+\beta)$ decreases the probability variance.

αs and βs are initialized to 0, or system operators can manually specify a larger α to favor certain performance metrics. Then, the input selection strategy repeatedly updates αs and βs as follows. Each round starts by randomly generating a combination of models — specifically, we generate random numbers from each β-distribution and select k performance metrics with the largest number. This combination is then evaluated for the prediction accuracy. If the current round results in a higher accuracy then the first round, we increment each performance metric's α value, otherwise the β value.

This process of updating α and β repeats for a user-defined number of rounds, or if the random selection converges for several rounds. Upon termination, we again generate random numbers from each β-distribution and select k performance metrics with the largest number.

Output weighting strategy. Our implementation is based on differential evolution for stochastic minimization [33]. It makes no assumptions on the search space distribution, and it is easy-to-use due to few hyperparameters. Differential evolution is initialized with a population of starting points in the search space. In rounds of mutation-recombination-selection, it moves towards the optimum.

A subset of the initial population can be based on cached w_k^*s. So, the N initial population $(w_1^0, w_2^0, \dots w_N^0)$ consists of uniformly random candidates and previously computed w_k^*. At each round G, differential evolution creates mutant vectors storing combinations of individuals (w_i^G, w_j^G, w_k^G) that are randomly chosen from the current population. Mutant vectors are then mixed with a pre-determined population candidate, to produce a new trial candidate. With data points selected by the Ingestion Engine, we then evaluate this trial candidate. If the new trial candidate yields a better prediction accuracy, it is added to the population.

This process of updating population repeats for a user-defined number of rounds, or if the population converges for several rounds. Upon termination, the population candidate that best maximizes Equation 2 is returned.

4.1.1 Learning Assignment APIs

LA.init(X_names, y_name) sets the labels for assignment inputs and output.

LA.add(X, y, del) adds a new model, and trains the new model with input `X` and output data `y`. `del` specifies whether existing models should be deleted.

LA.input_selection(X, y) runs the input selection strategy with the inputs `X` and output `y`.

LA.output_weighting(X, y) runs the output weighting strategy with the inputs `X` and output `y`.

LA.predict(X) predicts for `X`.

4.1.2 Graph Update APIs

GE.add(s_name, p_name, a_ptr) adds a new instance-vertex in the graph, to represent one instance of the service `s_name` for the performance metric `p_name`. `a_ptr` points to the learning assignment instance.

GE.connect(s1_name, s2_name) specifies the performance dependency from the service `s1_name` to `s2_name`. Since a service can have multiple performance metrics, `GE.connect` adds directed edges for all combinations of `s1_name`'s and `s2_name`'s metrics. It matches learning assignments' input and output labels to make the connection.

GE.scale(s_name, num_inst) replicates/removes instance-vertices to match `num_inst`, for all `s_name`'s performance metrics. And, for each performance metric, it adds a service-vertex to aggregate instance-vertices. These service-vertices are initialized to an input dimension of `num_inst`, and they can be trained by invoking `GE.fit`.

GE.replace(s_name, p_name, a_ptr) updates instance vertices to reference the new learning assignment `a_ptr`. Service-vertice needs to be initialized by `GE.fit`.

GE.fit(s_name, p_name, time_window) creates a learning assignment, for the service-vertex of `s_name` service and `p_name` metric. Then, it retrieves data points within the last `time_window` seconds from IE, and invokes `LA.add`.

4.2 Testing Engine (TE)

TE implements functionalities to support graph testing. First, `TE.compute_err` computes the per-vertex test error, with the test dataset given in the argument. The current implementation uses the mean absolute error. The test dataset is in a tabular format; each row represents one system benchmark, and columns record service config knob settings and performance measurements. Second, `TE.debug` starts the graph debugging strategy and returns a learning assignment's name.

4.3 Ingestion Engine (IE)

IE ingests and buffers per-service telemetry streams for ML model training and testing. A stream contains one time-series data type, which can be a performance metric, a configuration knob, or an observable state. Data are published to IE by `IE.add_data`. They are in JSON format with the following fields: `stream_uri`, `type`, `seq_num`, and `val`. The `type` field can be "continuous", "discrete", or "choices". The `seq_num` field is an incrementing integer such as the Unix timestamp.

5 Evaluation

We evaluate and demonstrate the superior performance of Fluxion, with three complex microservice systems on up to 100 VMs, under deployment dynamics like service scale-out and replacement. Our major results include:

(1) Fluxion consistently maintains a lower performance modeling MAE (mean absolute error). Considering the case of gradually scaling Hotel Reservation from 15 to 142 services, Fluxion's MAE is 29.14% lower on average and up to 68.41% lower than comparison baselines.

(2) Fluxion's lower MAE enables better end-to-end system latency. Considering the case of switching from baselines to Fluxion, auto-tuning optimizers achieve a speedup of 1.24× on average (and up to 1.44×), for TrainTicket's 90th-percentile latency.

(3) Using a 30-day Azure trace, results show that Fluxion can capture system dynamics in the temporal dimension. By recognizing and adapting to the recurring patterns, the daily training time is reduced by up to 99.98%.

5.1 Microservice Systems

We evaluate the effectiveness of Fluxion, by measuring both the performance modeling accuracy improvement (or MAE *reduction*), and the resulting latency improvement for microservice-based systems. Our evaluations are based on case studies — as microservice systems are orchestrated to exhibit deployment dynamics, we run auto-tuning to continually optimize their tail latency, i.e., the 90th-percentile latency.

Microservice system setup. We deploy three systems: ***(1)*** TrainTicket [31], with 41 unique services, ***(2)*** Hotel Reservation, with 15 unique services from DeathStarBench [11], ***(3)*** Boutique, with 11 unique services from Google [15]. Services are managed by Kubernetes, and they can be replicated and replaced. KubeDNS is used for round-robin load-balancing. Services log per-request latencies for all remote procedure calls, and measurements are centrally stored in an InfluxDB. Appendix lists each system's knobs. For databases, we select top knobs that have been identified to impact read/write latency in production, by Microsoft engineers.

Experiment setup. Our comparison baselines are performance models of monolithic Gaussian process (GP) and multi-layer perceptron (DNN) models. These baselines are common in recent performance optimization efforts [2, 3, 6, 7, 9, 13, 14, 20, 21, 23, 34, 41]. GP uses Matern(5/2) kernel [3]. We

use NNI [27] to tune DNN hyper-parameters: number of hidden layers, hidden layer size, and initial learning rate. We construct Fluxion graphs with APIs in §4.1.1, and learning assignments use GP. Externally, baselines and Fluxion graph have the same inputs and output (c.f. Appendix).

Our testbeds are 3 clusters on Azure — 100-VM cluster (with Intel E5-2673 CPU and 54GB RAM) for Train Ticket and Hotel Reservation; 6-VM cluster (with Intel 8272CL CPU and 8GB RAM) for Hotel Reservation; a 9-VM cluster (with Intel 8171M CPU and 16GB RAM) for Boutique.

Methodology. We send workloads of requests, with wrk2 [38] and Locust [24] (c.f. Appendix). We periodically induce the following stresses to trigger orchestrations, hence deployment dynamics. First, we change the requests per second (RPS) or ratio of request types, to stress different services and paths. This stress then triggers Kubernetes' HPA (Horizontal Pod Autoscaler) to replicate or reclaim multiple services, to maintain an average service CPU utilization of 60%. Second, we replace a service. Third, we scale-out the entire system.

After each orchestration operation (i.e., deployment dynamics), we first ensure all modeling approaches' inputs match system knobs. This step involves training new baselines, and also re-composing Fluxion's graph. In addition, to evaluate how different approaches would improve with further training, we collect new training dataset. Each iteration performs one random benchmark and measures per-request latencies for ∼10 minutes. To compare prediction MAE (mean absolute error), we collect an additional 100 random benchmarks as the testing dataset. MAE is computed as the average error between a benchmark's actual latency and predicted latency.

5.1.1 Performance Modeling Error Reduction

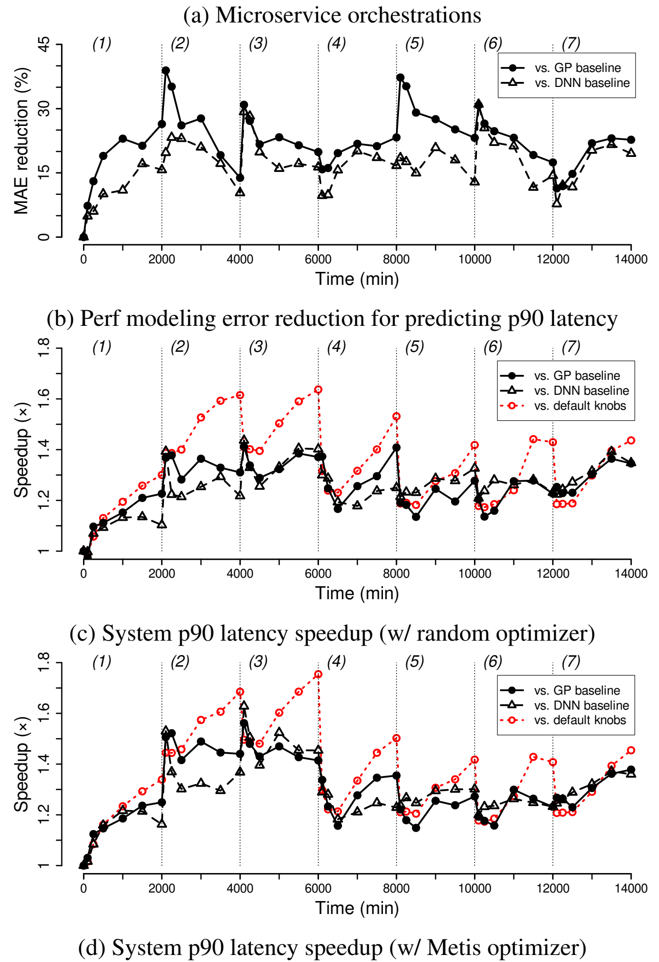
We evaluate how well Fluxion reduces the MAE of predicting the end-to-end latency, as compared to baselines. In the presence of deployment dynamics, a consistently lower MAE suggests a more robust performance modeling.

For TrainTicket, Fig 5b shows that Fluxion consistently maintains a lower MAE (i.e., MAE reduction is always greater than 0) for predicting 90th-percentile latency. It achieves 7.30–38.92% and 4.88%–29.22% lower MAE than monolithic GP and DNN baselines, respectively; this translates to an average MAE reduction of 2,181.89 μs and 1,547.27 μs. Similarly, for Hotel Reservation on the 100-VM cluster, Fig 7b shows that Fluxion achieves 34.82–60.05% and 27.24–57.39% lower MAE than monolithic GP and DNN baselines, respectively; this translates to an average MAE reduction of 7,298.78 μs and 6,858.30 μs. For Hotel Reservation on the 6-VM cluster, Fig 6b shows that Fluxion achieves 10.04–68.41% and 10.87–66.14% lower MAE than baselines; this translates to an average MAE reduction of 2,814.09 μs and 2,205.70 μs. Finally, Fig 8b shows Fluxion's lower MAE, for Boutique.

Right after deployment dynamics happen (i.e., orchestration operations), the MAEs of all approaches increase. However, compared to baselines, the *relative* MAE reduction of Fluxion becomes significantly higher after an orchestration operation. Since monolithic baselines require entirely new models, their modeling accuracy can improve only after collecting sufficient data points and training. For example, as we gradually scale-out Hotel Reservation from 15 to 142 services, we need new GP and DNN baselines to accommodate the input dimension that grows from 63 to 1,034. Furthermore, Table 6a shows an operation (Step #6) that replaces all Memcached services by Redis. Although the input dimension here does not change, we need new baselines because Redis

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "TrainTicket"	49	242
(2) Stress RPS (=100)	Scale-out *Station* (2×)	50	247
(3) Change req ratio	Scale-out *Station* (2×), *Route* (2×)	51	252
(4) Change req ratio	Scale-out *Station* (2×), *Route* (2×), *Order* (2×)	53	262
(5) Stress RPS (=250)	Scale-out *Station* (5×), *Route* (6×), *Order* (4×)	62	307
(6) Stress scale	Scale-out *all* services (3×)	105	510
(7) Replace services	*TiDB* replaces *MySQL*	49	242

(a) Microservice orchestrations



(b) Perf modeling error reduction for predicting p90 latency

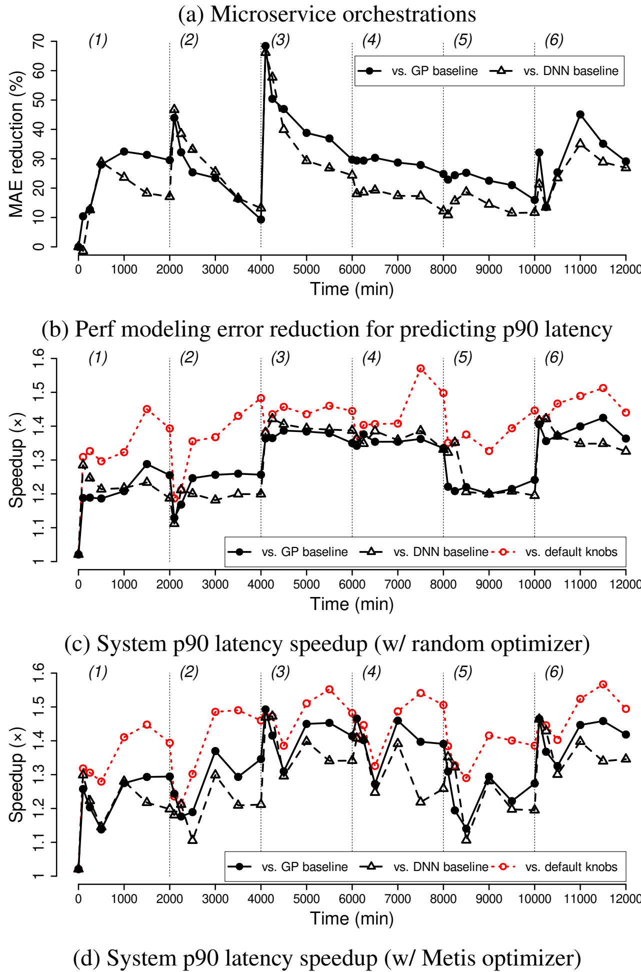
(c) System p90 latency speedup (w/ random optimizer)

(d) System p90 latency speedup (w/ Metis optimizer)

Figure 5: TrainTicket on 100-VM cluster (Intel 2673).

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "Hotel Reservation"	15	63
(2) Stress RPS (=500)	Scale-out *Reservation* (2×)	17	72
(3) Change req ratio	Scale-out *Reservation* (2×), *Rate* (2×)	19	80
(4) Stress RPS (=800), change req ratio	Scale-out *all* services (3×)	37	139
(5) Stress scale	Scale-out *all* services (4×)	48	177
(6) Replace services	*Redis* replaces *Memcached*	15	63

(a) Microservice orchestrations



(b) Perf modeling error reduction for predicting p90 latency

(c) System p90 latency speedup (w/ random optimizer)

(d) System p90 latency speedup (w/ Metis optimizer)

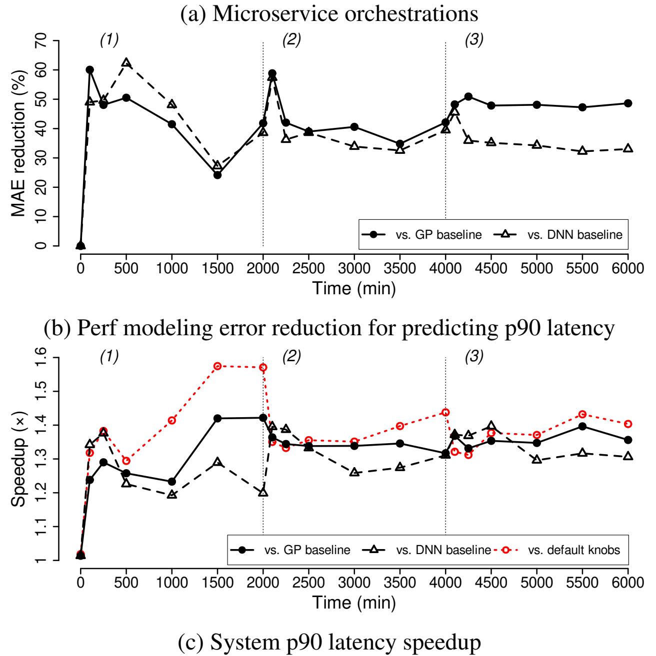
Figure 6: Hotel Reservation on 6-VM cluster (Intel 8272CL).

brings a different set of configuration knobs to performance modeling.

On the other hand, Fluxion is able to localize the inference graph regions (or some learning assignments) that require updating, without changing the rest of the graph. For example, when Hotel Reservation scales-out from 15 to 142 services, graph updates replicate all services' instance-vertices and train their service-vertices. The former incurs no costs, and the latter represents only 15 of the 157 learning assignments in the inference graph. Similar observations can be made for TrainTicket (c.f. Fig 5) and Boutique (c.f. Fig 8).

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "Hotel Reservation"	15	63
(2) Stress RPS (=1,200), change req ratio	Scale-out *all* services (6×)	70	253
(3) Stress scale	Scale-out *all* services (6×), *all Memcached* (10×)	142	1,034

(a) Microservice orchestrations



(b) Perf modeling error reduction for predicting p90 latency

(c) System p90 latency speedup

Figure 7: Hotel Reservation on 100-VM cluster (Intel 2673).

5.1.2 Model Adaptation Time Reduction

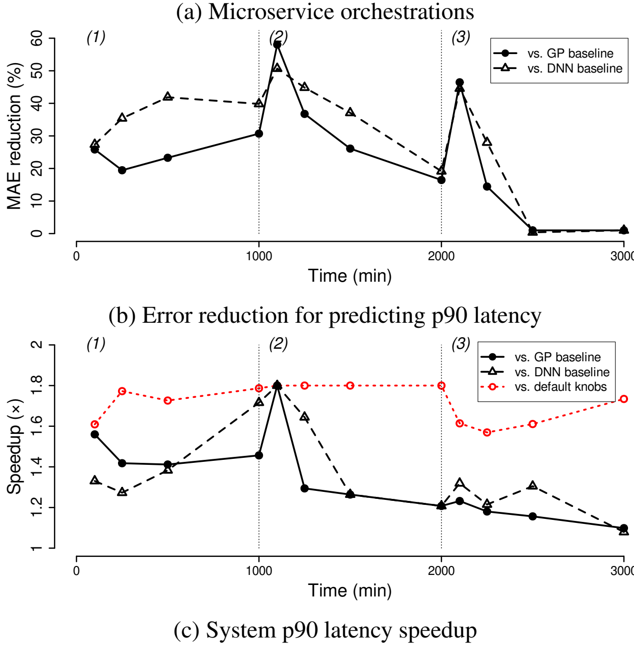
After each orchestration operation, all modeling approaches are updated to match inputs to system knobs. While baselines need to be re-trained, Fluxion minimizes the overhead by localizing updates to some inference graph regions. If none or only a small number) of regions need updating, graph can immediately achieve low prediction MAE, without collecting training data points. Since collecting data points can be time-consuming (c.f. §2.2), if more training data points are necessary, the modeling accuracy will take longer to improve.

We take a deep dive into TrainTicket. After an orchestration operation, both monolithic GP and DNN baselines generally need at least 300–400 data points, in order to train new monolithic models that have an MAE close to what Fluxion can achieve with only 10–25 data points. If each system benchmark takes ∼10 minutes, this is a reduction of 2,900–3,750 minutes (or up to 30× reduction).

Furthermore, we highlight the case where Hotel Reservation is scaled-out to 142 services. With 500 data points, monolithic GP and DNN models achieve an MAE of 29,576.57μs and 22,575.57μs, respectively. On the other hand, with only 10 data points, Fluxion can already achieve an MAE of 17,716.25μs. Since Fluxion needs to update only a small subset of the learning assignments in the graph, it requires

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=100)	Deploy "Boutique"	11	63
(2) Stress RPS (=200), stress scale	Scale-out *all* services (2×)	22	126
(3) Stress RPS (=300), stress scale	Scale-out *all* services (3×)	33	189

(a) Microservice orchestrations



(b) Error reduction for predicting p90 latency

(c) System p90 latency speedup

Figure 8: Boutique on 9-VM cluster (Intel 8171M).

much fewer training data points.

5.1.3 End-to-end System Latency Speedup

Intuitively, a better performance model should enable better performance optimization. Based on Fluxion's MAE reduction shown in §5.1.1, this subsection now quantifies how it can better reduce end-to-end system latencies. To do so, we couple performance models with auto-tuning optimizers (c.f. §2.1): ***(1)*** a random optimizer that selects the best knob setting from randomly generated 100,000 settings, and ***(2)*** Metis [21].

Fig 5c and 5d show TrainTicket's 90th-percentile latency speedup from using Fluxion, over baselines. With Fluxion, the random optimizer achieves a speedup up to 1.41× and 1.43×, over GP and DNN baselines, respectively; the Metis optimizer achieves a speedup up to 1.52× and 1.62×. While the choice of optimizer can impact the auto-tuning outcome, using Fluxion can result in better performance optimization. Even as we introduce deployment dynamics, the speedup is always greater than 1. We note that figures also plot the speedup over default knobs, to demonstrate the benefits of performance optimization.

Fig 7c shows similar observations for Hotel Reservation on the 100-VM cluster — with Fluxion, the random optimizer achieves a 90th-percentile latency speedup up to 1.43× and 1.40×, as compared to relying on GP and DNN baselines, respectively. Even for the last orchestration step where Hotel Reservation is scaled-out to 142 services, the speedup can be up to 1.40× and 1.39×. Furthermore, compared to the default knob setting, Fluxion achieves a maximum speedup of 1.57×.

Fig 6c and Fig 6d illustrate the results for Hotel Reservation on the 6-VM cluster. Fluxion helps the Metis optimizer to achieve a speedup up to 1.49× and 1.47×, over GP and DNN baselines, respectively. Fig 8c shows Boutique, where the random optimizer achieves a speedup up to 1.81× and 1.79×, over GP and DNN baselines, respectively.

5.2 Microbenchmarks

5.2.1 Exposing a Spectrum of Performance Metrics

We evaluate the benefits of exposing a spectrum of metrics for external performance dependencies. To do so, we compare the following inference graphs that predicting Hotel Reservation's 90th-percentile latency. In the first inference graph, all services' learning assignments consider (50, 80–99)th-percentile latencies as external performance dependencies from downstream services. In the second inference graph, they consider only the target performance metric, or the 90th-percentile latency. The last inference graph considers only (50, 85, 90, 95)th-percentile latencies, which are suggested by Fluxion's input selection strategy.

We delve into the case when Hotel Reservation is scaled-out by a factor of 6. If we consider all (50, 80–99)th-percentile latencies, the graph MAE is 9,722.53μs. This is a 10.87% lower MAE, as compared to the second inference graph's MAE of 10,779.22μs. Even across a sequence of orchestrations on the 6-VM cluster, the first inference graph MAE is at least 1.39% lower than the second inference graph. We note that the trade off is the inference graph size — the first graph has 2,170 vertices and 17,647 edges, but the second graph has only 110 vertices and 167 edges. Since each vertex references a learning assignment, this trade off can have a significant implication in terms of the training costs, i.e., 660 more learning assignments to train.

The third latency graph tries to include only highly relevant metrics. In the case of scaling-out Hotel Reservation above, this graph achieves a MAE of 10,091.87μs, or 6.81% lower than considering only the 90th-percentile latency. Furthermore, compared to considering all (50, 80–99)th-percentile latencies, the third latency graph reduces the graph size to 419 vertices and 1,055 edges. Although there is a modest 3.80% increase in MAE, the number of necessary learning assignments reduces by 561.

5.2.2 Capturing System's Temporal Dynamics

We evaluate how well learning assignments can re-use previously trained models to adapt to recurring patterns in temporal dynamics. Our evaluations are based on a case study, which

auto-tunes per-service VM resource allocations by predicting incoming VM request's max CPU utilization and lifetime. Particularly, as the VM utilization pattern changes over time, models are trained and added to learning assignments, which then use output weighting to compute outputs.

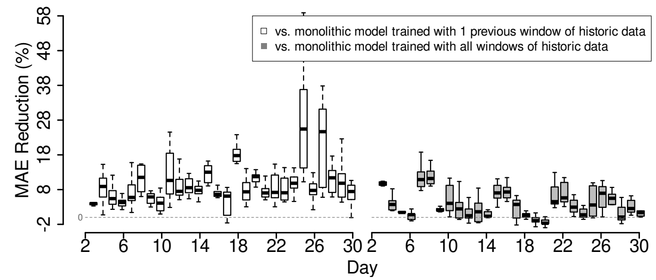
Experiment methodology. We use the 30-day Azure VM utilization trace as the workload, and this production trace was recorded in 2019 [26]. It records resource utilization measurements (e.g., 5-min CPU utilization and VM lifetime) and VM metadata (e.g., VM size and encrypted subscription/deployment ID) for 2,695,548 VM requests.

Following Cortez et al. [7], our baselines are monolithic random forest and extreme gradient boosting tree (XGBoost), for modeling VM's max CPU utilization and lifetime, respectively. And, these metrics are bucketized. Model inputs include encrypted subscription/deployment ID, requested VM size/category, hour of the day, day of the week. We re-train monolithic baselines at the beginning of each day in the trace, with data points from the previous day or all past days. Furthermore, to ensure comparison baselines are properly trained, we tune their hyper-parameters with NNI [27].

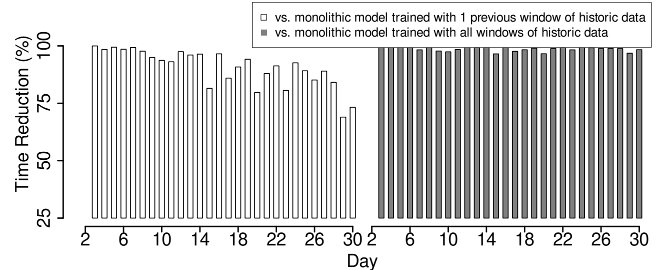
For Fluxion, we set up two learning assignments to represent VM's max CPU utilization and lifetime. We then discretize the continuous trace into non-overlapping batches by days. At the end of each day in the trace, Fluxion evaluates the mean absolute error (MAE). A new model is trained and added only if this MAE is below 85%. The learning assignment gradually accumulates models for future re-uses, and its output is the weighted sum of all its models' predictions. We re-compute the weights by invoking `GE.fit()` every two hours, with recent data points.

Performance modeling error reduction. Fig 9a and Fig 10a suggest that Fluxion can significantly reduce the daily modeling MAE. One reason is that `GE.fit()` can quickly re-adjust weights with the output weighting strategy, rather than going through expensive model training. In summary, compared to baselines trained with the previous day of data, the daily MAE reduction is 2.04%–11.25% and 3.97%–25.43%, for predicting max CPU utilization and lifetime, respectively. Compared to baselines trained with all historic data, the daily MAE reduction is 0.64%–6.49% and -1.16%–11.20%, for predicting max CPU utilization and lifetime, respectively. We note that there are days (e.g., day #20) where Fluxion has a slightly higher MAE than baselines. The reason is that these days exhibit a pattern that significantly drifts from previous days.

Model adaptation cost reduction. Fig 9b and Fig 10b suggest that Fluxion significantly reduces the daily training time, i.e., the time spent on model training and `GE.fit()`. We note that this reduction varies by days, as Fluxion does not need to train new models every day. By re-using models, it trains only a total of 20 and 23 models for predicting VM's max CPU utilization and lifetime, respectively. Furthermore, `GE.fit()` is relatively lightweight — invoking `GE.fit()` 12 times a day



(a) Perf modeling error reduction



(b) Adaptation time reduction

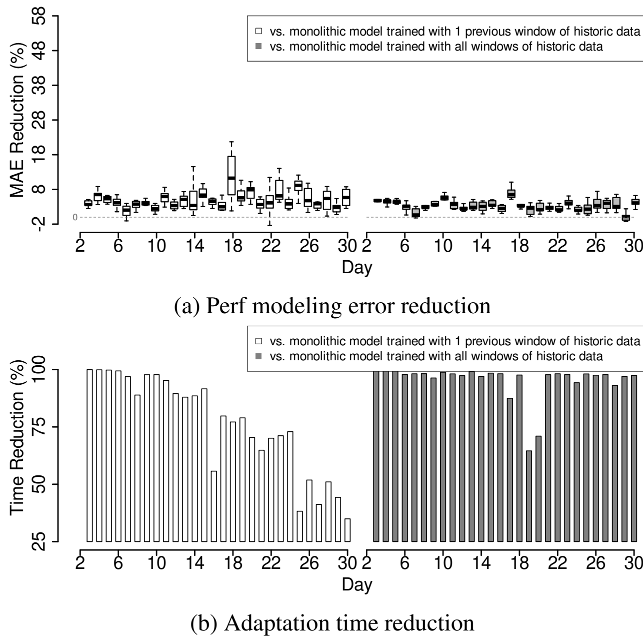
Figure 9: Benefits of learning assignments in predicting VM lifetime in the Azure 30-day trace, compared to baselines. The output weighting strategy promotes the re-use of models trained at different time periods. We evaluate the prediction error every two hours to produce daily boxplots.

takes ∼1,620 and ∼1,480 seconds, for modeling VM's max CPU utilization and lifetime, respectively. So, compared to monolithic random forest and XGBoost baselines trained with the previous day of data, Fluxion reduces the daily training time by 34.93–99.97% and 68.99–99.98%, respectively. Compared to monolithic random forest and XGBoost baselines trained with all historic data, Fluxion reduces the daily training time by 64.55–99.96% and 96.51–99.98%, respectively.

5.2.3 Graph Re-training

As mentioned in §3.2.3, there are two error sources. As previous evaluation has shown Fluxion's benefit on *error source #1* (new learning assignments), this section focuses on *error source #2* (unforseen prediction inputs). As the incoming request pattern changes, different service-to-service execution paths are stressed. We evaluate how well Fluxion can identify the learning assignments at fault in the inference graph.

We conduct experiments by altering the ratio of four request types in the wrk2 workload generator: `search`, `recommend`, `reserve`, and `user`. After we scale-out Hotel Reservation by a factor of 6, we increase the ratio of `search`, `recommend`, and `user` requests from 10% to 30%. At this point, the graph MAE for predicting the 90th-percentile latency is 12,444.63μs. Then, the first iteration of graph debugging identifies MongoDB's learning assignment. After adding a new model trained with recent data points, the graph MAE reduces to



(a) Perf modeling error reduction

(b) Adaptation time reduction

Figure 10: Benefits of learning assignments in predicting VM's max CPU utilization in the Azure 30-day trace, compared to baselines. Output weighting promotes the re-use of models trained at different time periods. We evaluate the prediction error every two hours to produce daily boxplots.

11,024.66μs (or a 11.41% reduction). Subsequent iterations identify the following services: User, Recommendation, and Geo; updating these services reduces MAE to 10,190.89μs (or a 18.11% reduction), 9,483.48μs (or a 23.79% reduction), and 9,352.04μs (or a 24.85% reduction). Beyond this point, the graph MAE reduction starts to exhibit a diminishing return.

6 Related Work

Auto-tuning. Performance tuning for distributed systems is a problem that has continuously received attentions. Instead of relying on heuristics, previous efforts have demonstrated the feasibility of ML-based auto-tuning, for systems ranging from databases [2,20,21,36], storage [6], VM instances [3,7,13,14,41], cloud services [23], and big data analytics [34].

The focus of this paper is not to apply auto-tuning to new system scenarios, nor to propose new ML techniques. Rather, we are motivated by the limitations of driving auto-tuning with monolithic performance models, especially in the presence of deployment dynamics. We take the first step at abstractions and pieces to systematically bring the concept of modularity to performance modeling. Furthermore, one key question addressed is how this process should be standardized and generalized, without being coupled to specific modeling techniques and systems.

Ensemble of models. The system community has proposed model ensemble as a research opportunity to improve the development speed and adoption in the real world. Stoica et al. [32] describe this opportunity as composable AI systems. Their goal is to query multiple models in different patterns to balance the tradeoff between accuracy, latency, and throughput of a model serving system. In contrast, Fluxion focuses on providing performance modeling for modern systems.

Ensemble learning is a popular machine learning approach that combines multiple models to achieve a higher prediction accuracy on a given dataset [16,17,28]. Representative efforts include bagging [5], boosting [8] and so on. Unlike Fluxion, these techniques do not consider system deployment dynamics and the inherent modularity of ML-based performance model for large complex distributed systems.

Previous research efforts have also applied ensemble learning, to realize incremental learning [29,35]. They inspire our design for learning assignments to keep a list of models.

7 Discussion

We discuss overarching issues regarding modularity level. Fluxion's current design closely follows the system modularity of services, but a finer or coarser modularity level might also seem viable. For Fluxion, the main difference would be in the number of learning assignments. Having said that, we choose the modularity level of services, in order to align with what deployment orchestrations typically operate on. While developers could carefully craft a monolithic system that outperforms service-based counterpart, doing so would complicate everyday CI/CD orchestrations in production. Therefore, we advocate developers to follow the well-known principle of engineering cohesive and loosely coupled services. And, for training, these services should expose appropriate knobs and performance feedback.

8 Conclusion

We report the design and implementation of Fluxion. Fluxion applies the principle of modularity to make performance modeling practical for distributed systems such as microservices. Even under deployment dynamics, empirical results show that Fluxion consistently maintains a higher performance modeling accuracy than monolithic models. This in turn enables auto-tuning tools to better reduce end-to-end system latencies.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Ravi Netravali, for their extensive comments and suggestions.

References

[1] Adam Gluck. Introducing Domain-Oriented Microservice Architecture, 2020.

[2] 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.

[3] 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.

[4] auto-sklearn. auto-sklearn. http://github.com/automl/auto-sklearn.

[5] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[6] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *ATC*. USENIX, 2018.

[7] 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.

[8] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[9] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. On the Use of ML for Blackbox System Performance Prediction. In *NSDI*. USENIX, 2021.

[10] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical Scalable ML-Driven Performance Debugging in Microservices. In *ASPLOS*. ACM, 2021.

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. Unveiling the Hardware and Software Implications of Microservices in Cloud and Edge Systems. *IEEE Micro*, 2020.

[12] Giulio Santoli. Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo, 2016.

[13] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *SIGKDD*. ACM, 2017.

[14] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *CNSM*. IEEE, 2010.

[15] Google. Online Boutique. http://github.com/GoogleCloudPlatform/microservices-demo.

[16] L.K. Hansen and P. Salamon. Neural Network Ensembles. In *Transactions on Pattern Analysis and Machine Intelligence*. IEEE, 1990.

[17] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive Mixtures of Local Experts. In *Neural Computation*. MIT, 1991.

[18] Jeremy Cloud. Decomposing Twitter: Adventures in Service Oriented Architecture, 2013.

[19] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: An Efficient Neural Architecture Search System. In *KDD*. ACM, 2019.

[20] Feifei Li. Cloud-native Database Systems at Alibaba: Opportunities and Challenges. In *VLDB*, 2019.

[21] 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.

[22] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, and Lidong Zhou. The Case for Learning-and-System Co-design. In *SIGOPS Operating Systems Review*. ACM, 2019.

[23] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. AutoSys: The Design and Operation of Learning-Augmented Systems. In *ATC*. USENIX, 2020.

[24] Locst. Locust - A Modern Load Testing Framework. https://locust.io/.

[25] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC*. ACM, 2021.

[26] Microsoft. Azure Public Datasets. http://github.com/Azure/AzurePublicDataset.

[27] Microsoft. NNI. http://github.com/Microsoft/nni.

[28] Robi Polikar. Ensemble Based Systems in Decision Making. *IEEE Circuits and Systems Magazine*, 2006.

[29] Robi Polikar, Lalita Udpa, Satish S. Udpa, and Vasant Honavar. Learn++: An Incremental Learning Algorithm for Supervised Neural Networks. In *Transactions on Systems, Man, and Cybernetics: Systems*. IEEE, 2001.

[30] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A Tutorial on Thompson Sampling, 2017.

[31] Software Engineering Laboratory of Fudan University. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket.

[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] R Storn and K Price. Differential Evolution - a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 1997.

[34] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*. USENIX, 2016.

[35] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining Concept-Drifting Data Streams Using Ensemble Classifiers. In *KDD*. ACM, 2003.

[36] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. UDO: Universal Database Optimization using Reinforcement Learning. *VLDB*, 2021.

[37] Cort J. Willmott and Kenji Matsuura. Advantages of the Mean Absolute Error (MAE) over the Root Mean Square Error (RMSE) in Assessing Average Model Performance. In *Climate Research*. Inter-Research, 2005.

[38] wrk2. wrk2. http://github.com/giltene/wrk2.

[39] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the Cache Warm? Manufacturing a Rule of Thumb. In *HotCloud*. USENIX, 2020.

[40] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *SoCC*. ACM, 2018.

[41] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *SoCC*. ACM, 2017.

A Appendix

This appendix provides further information regarding our experiment setup in §5.

- Table 1, 2, and 3 list features (i.e., configuration knobs and states) that we use as performance model inputs, for our three microservice systems. These tables break down these features by microservices.
- Our Fluxion inference graphs use Gaussian Process (GP) models in learning assignments. These GP models use the Matern(5/2) kernel.
- We use NNI to automatically tune hyperparameters for the DNN baseline: the number of hidden layers (3–7), each hidden layer size (100–2,048), and the initial learning rate (0.001–0.1). We budget 24 hours of NNI for each baseline.
- We rely on scripts provided by microservice systems, to generate different request payloads. These requests are then sent by wrk2 or Locust, as recommended by each system. They are available here: Hotel Reservation (https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation/wrk2), Boutique (https://github.com/GoogleCloudPlatform/microservices-demo/tree/main/src/loadgenerator), and TrainTicket (https://github.com/FudanSELab/train-ticket/issues/131).

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All MySQL	`innodb_thread_concurrency` (8–128) `innodb_buffer_pool_size` (512–3,072) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All MongoDB	`eviction_dirty_target` (10–99) `eviction_dirty_trigger` (1–99) `cache` (50–200) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All TiDB	`storage.scheduler_worker_pool_size` (2–32) `rocksdb.write_buffer_size` (64–256) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 1: Performance model inputs, for TrainTicket.

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All Redis	`hash_max_ziplist_entries` (32–4,096) `maxmemor_samples` (1–10) `maxmemory` (1–16) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS (read) RPS (write)
"*Email*"	`max_workers` (1–20) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
"*Recommend*"	`max_workers` (1–20) `max_response` (1–5) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
"*Ad*"	`max_ads_to_serve` (1–10) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 2: Performance model inputs, for Boutique.

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All Memcached	`memory-limit` (30–100) `threads` (1–16) `slab-growth-factor` (1.1–2.2)	RPS (read)
All MongoDB	`eviction_dirty_target` (10–99) `eviction_dirty_trigger` (1–99) `cache` (50–200) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 3: Performance model inputs, for Hotel Reservation.