Simdex: A Simulator of a Real Self-adaptive Job-dispatching System Backend

Martin Kruliš
Charles University, Prague, Czech Republic
krulis@d3s.mff.cuni.cz

Tomáš Bureš
Charles University, Prague, Czech Republic
bures@d3s.mff.cuni.cz

Petr Hnětynka
Charles University, Prague, Czech Republic
hnetynka@d3s.mff.cuni.cz

ABSTRACT

Self-adaptive systems comprise a complex domain of computing systems that are intensively studied but sparsely employed in real applications. Furthermore, recent trends in computer science are steering towards machine learning which has yet to fully penetrate this domain. We would like to present Simdex — a realistic simulator of the self-adaptive backend that dispatches computing jobs among multiple workers. It is based on ReCodEx, a system for semi-automated evaluation of coding assignments that have been used for the past 5 years at our School of Computer Science. The simulator replays the workload logs recorded from ReCodEx over that period which provides a quite thorough evaluation and near-to-real feedback for the simulated scenarios. Furthermore, the design of the simulator is highly modular and allows the implementation of different self-adaptive controllers, including ones based on machine learning, as we demonstrate in our examples.

CCS CONCEPTS

• **Software and its engineering** → **Simulator / interpreter**; • **Computing methodologies** → **Self-organization**; • **Computer systems organization** → **Self-organizing autonomic computing**.

KEYWORDS

Job dispatching, simulator, self-adaptive, machine learning

ACM Reference Format:
Martin Kruliš, Tomáš Bureš, and Petr Hnětynka. 2022. Simdex: A Simulator of a Real Self-adaptive Job-dispatching System Backend. In *17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '22), May 18–23, 2022, PITTSBURGH, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3524844.3528078

1 INTRODUCTION

Machine learning is gradually becoming an important enabling technology for self-adaptive systems. It helps especially in cases when the model or process governing the environment, towards which the system self-adapts, is unknown. The lack of a model typically causes two problems: (a) the actual state of the environment is unknown, and thus it is difficult to choose an adaptation tactic correctly, and (b) it is not clear how the state of the model changes based on an action of the system. As a result, it is challenging to design adaptation rules to govern such a system.

Machine learning overcomes this problem by introducing the ability to extrapolate and remember relevant aspects of the environmental model that are based on observations of the environment (typically collected by sensors). Similarly, it can be used to predict associated hidden properties (e.g., energy consumption) that are not directly observable [9].

However, the greatest challenge of machine learning-based adaptation in existing systems lies in the fact that it cannot be easily trained without having real data. Pure simulation of a system typically does not give generalizable results because the simulator has to contain (and simulate) the model that the machine learning method is to learn. Thus, when researchers evaluate such an approach, they first create a model of an environment, hide it in a simulator, and then use the machine learning approach to rediscover the model. Though there is often no other way for the evaluation, this approach is plagued with threats to validity.

A proper evaluation requires that the data are collected from a real environment and thus reflect some hidden model with all the complex organic relationships between entities in the environment that would be dismissed or overly simplified in an artificial simulation. Unfortunately, this observation also limits the validity of existing artifacts (such as those collected so far at SEAMS), which are often based purely on simulation, for evaluating systems that employ machine learning. It is our belief that an artifact for evaluating machine learning-based adaptation methods should at least utilize real-world data (which provides the "real hidden model" part of the artifact) if the real system cannot be used directly for the evaluation (which is often the case). In this paper, we present such an artifact that contains both — a simulator and a dataset collected from actual users.

The artifact featured here is based on ReCodEx [7], a system for evaluation of coding assignments that is based on well-known principles of software testing common in software development. The part of ReCodEx that could benefit the most from self-adaptation methods is the backend of the system. It comprises a pool of workers and a broker that dispatches evaluation jobs to these workers. ReCodEx has been deployed at our faculty for nearly five years now (while the experience we have with similar systems extends over 15 years), and we were able to collect and compile a dataset of job-evaluation logs, which are used by the artifact for realistic workload simulations.

The job-dispatching scenario is well known in modern systems, making this use case easy to adopt and explain in scientific work.

Furthermore, it permits different combinations of optimization criteria to be tested, such as evaluation latency, throughput, or power consumption of the workers. It is difficult to write a generic job-dispatching strategy that would work optimally in a system where the workload parameters change over time, so we have adopted the self-adaptive approach based on a standard MAPE-K loop that monitors, analyze, and optionally executes modifications of the running system over time. The proposed simulator allows one to test various experimental self-adaptive strategies with selected evaluation metrics.

Our proposed testbed contains the following key features, which make it quite special:

- A highly modular simulator, where the user may inject own dispatcher and self-adaptive controller. The user may select a subset of prepared evaluation metrics or provide custom modules that compute arbitrary metrics.
- The simulator replays the log of jobs collected from an existing fully-deployed system, making the evaluation more sound and better resisting the threats to validity.
- The simulator is written in Python and fully ready to easily incorporate modern machine learning tools like TensorFlow [1].

Both the simulator and the dataset, along with the evaluation examples, are publicly available at GitHub[1].

The paper is organized as follows. Section 2 contains the explanation of the system that is being simulated. The testbed artifact that comprises the simulator and the dataset is detailed in Section 3. Three experiments based on two scenarios (described in Section 4) were implemented to evaluate the usability of the testbed. Section 5 overviews the related work and Section 6 concludes our paper.

2 BACKGROUND

Our simulator is based on a system called ReCodEx [7] which is a tool for semi-automated evaluation of coding assignments with a web frontend. This tool has been used at our university for several years, which allowed us to gather user experience as well as actual data that can be used in our simulator to evaluate experiments under realistic workloads. The system works as follows.

Teachers prepare coding exercises and assign them to selected groups of students (e.g., students visiting a particular course). The students solve the assignments individually and submit their solutions as source codes. The solutions are compiled and executed in a specialized sandbox that ensures secure evaluation (using the Linux kernel control groups [4] to create isolated containers) and also measures basic performance characteristics such as execution time and memory consumption. The execution of compiled solutions can be repeated multiple times for different testing scenarios (inputs, launch arguments, limits). Data gathered from these executions (correctness of the outputs, indicators of whether the performance meets given time and memory limits) are subsequently used to calculate the *correctness* of the solution, which is given to the student as feedback and may be used for grading as well [5].

One of the greatest benefits is that the solutions are evaluated almost instantaneously in most cases, and thus the students can receive the feedback in an interactive manner, which would not be possible if the code was evaluated manually by the teachers. Furthermore, the teachers do not have to bother with technical details such as testing whether the solution compiles or correctly handles corner cases, and they can focus on more high-level aspects such as the solution semantics or the quality of the student's coding style. The presence of time and memory limits also allows rough classification of the asymptotic behavior of the solutions by preparing inputs of incremental sizes and carefully comparing measured characteristics with expected values.

Another advantage of ReCodEx is that it can accommodate a wide variety of coding assignments ranging from trivial tasks for beginners to complex exercises required by advanced courses such as machine learning, parallel programming, or compiler design. Furthermore, ReCodEx supports 18 different programming languages ranging from mainstream such as Java, C++, or Python to more exotic ones like Haskell, Rust, or Scala. In addition, it provides means to adjust the compilation and testing process for individual exercises easily (e.g., combining Bison and Flex with C++ compilation).

For our research, we have focused on the process of evaluation of submitted solutions which can be generalized as job dispatching problem. The evaluation itself is performed by independent workers to ensure scalability and isolation; however, the aforementioned properties of the system makes the dispatching of individual evaluation jobs to the workers quite difficult as it need to satisfy multiple criteria such as throughput, latency, or technical constraints defined by the coding exercises. Furthermore, the workload of the system changes significantly over time which makes it particularly interesting from the perspective of self-adaptive systems.

2.1 Problem formalization

We have formalized the code evaluation process for the purposes of simulation as follows. When an evaluation job (or simply *job*) is created (i.e., when a user submits new source code), it is immediately passed to a *dispatcher* module which is implemented as the ZeroMQ[2] broker in ReCodEx. The dispatcher examines the job metadata and assigns the job to an appropriate worker. Each worker has a single queue and processes the jobs at its own pace in the first-come-first-served order. Once a job is assigned to a queue, it cannot be reassigned. Figure 1 depicts the schema of the system.

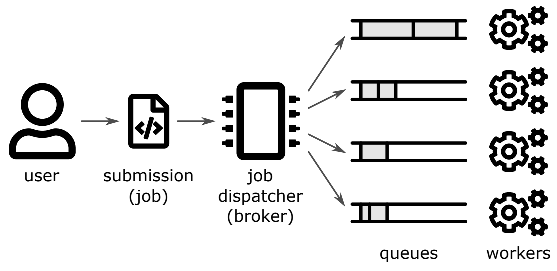

Figure 1: Schema of ReCodEx backend job dispatching

We have decided to ignore several issues that may arise in the real application to simplify the problem and the examples, namely:

[1]https://github.com/smartarch/simdex

[2]https://zeromq.org/

- Jobs and workers in ReCodEx are labeled with *worker group* tags that specify job requirements and worker capabilities (e.g., special HW or installed libraries). We have treated all workers as identical (i.e., a task can be assigned to any worker, and it will take the same amount of time to evaluate it) in the rest of the paper, although the *worker group* identifier is present in the data (so it may be used in future work).
- In a real setup, the workers may fail (e.g., be rebooted) in the midst of evaluation. We have completely ignored this possibility in the simulation, and the data do not comprise failed jobs since they are sporadic.
- There may be some overhead concerning the job dispatching and the evaluation results gathering. The simulation itself does not account for any overhead, but it can be incorporated by adding a small constant (e.g., 0.1s) to all job durations as compensation.

2.2 Self-adaptive approach

There are several objectives the system may need to optimize for, in particular:

- If a job is enqueued and other jobs are already in the queue, it is *delayed* until all previously submitted jobs are processed. One of the objectives may be the minimization of the average delay.
- If the workers are hosted on independent machines, we can shut them down when the system is underutilized to save power (or costs in the case of virtual machines).
- Besides the job delay, an exact yet quite detached metric, we can focus more on the user experience. In particular, short jobs should have a minimal delay to make the appearance of an interactive system, whilst long-running jobs may be delayed significantly more since it is unlikely the user would not wait for them eagerly. Unfortunately, this is a somewhat ambiguous matter since the threshold between short and long-running jobs is subjective and may differ significantly among users.

The presented objectives are highly affected by the job dispatching algorithm and worker management. Unfortunately, the actual workload of the system may not be easily predicted due to the wide variety of job types. Furthermore, the workload is heavily biased by daytime, assignment deadlines, users' sentiment, and other factors that are hard to interpret or predict. Therefore, it is challenging to design an explicit algorithm that optimally performs under all conceivable situations.

The simulation testbed presented in this paper provides an experimental platform to verify whether a self-adaptive approach would perform better than a static algorithm when optimizing for selected objectives. We aim to make this platform as flexible as possible to work with different optimization objectives and different self-adaptive strategies beyond the original needs of ReCodEx, whilst the evaluations themselves should remain comparable since they are based on the same real workload logs.

3 TESTBED

Our testbed comprises a simulator of the ReCodEx backend (i.e., job dispatcher, job queues, and workers) and a dataset extracted from the system logs (job metadata). The simulator replays the job logs whilst applying a user-provided self-adaptive dispatching strategy and using selected metric collectors to measure the quality of the tested strategy.

3.1 Simulator architecture

The simulator has a highly modular architecture. The two most important modules are *dispatcher* and *self-adaptive controller*, which are expected to be provided by the author of the experiment. The dispatcher implements a static dispatching algorithm — i.e., an algorithm that uses only a static configuration and the actual state of the workers to decide in which queue a job should be dispatched. The configuration is divided into two parts:

- worker queue properties,
- internal configuration of the dispatcher.

This separation was made only to make the simulator more convenient for various types of experiments and scenarios since some information are better kept with the workers (e.g., whether a worker is running) and some are better kept with the dispatcher (e.g., a model that helps predict the duration of newly spawned jobs). The overall schema of dispatching is depicted in Figure 2.

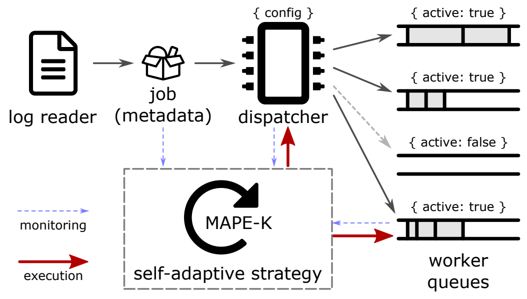

Figure 2: Self-adaptive simulator of job dispatching

The self-adaptive controller module analyses the system and optionally changes the behavior of the dispatcher by altering its internal configuration or properties of the worker queues. Although the controller can be implemented using various paradigms, we have relied on the classic MAPE-K loop [6] concept. In detail, the simulation runs in a single thread synchronously, and the interface of the self-adaptive controller is a single `do_adapt` method. It is invoked periodically in fixed intervals (as the simulation time advances) and just before each job is dispatched. The interface is used both for monitoring, analysis, and planning as well as for the execution of modifications. The controller module decides which steps of the MAPE-K loop are performed in each `do_adapt` invocation.

The interface for reading and changing worker queue properties is straightforward — each property is identified by a string key and may hold any value; thus, the interface is basically a getter and setter for a dictionary. The interface for the internal dispatcher configuration is designed by the author of the experiment (since both the dispatcher and the self-adaptive controller are created together). The technical details of the interfaces and the modules are specified in the simulator documentation.

3.2 Dataset

The dataset provided along with the simulator is basically a log of all jobs executed by the backend workers. A single record corresponds to a single evaluation of a source code submitted to the system. The job log record contains essential metadata, like the identifier of the exercise, the timestamp when the code was submitted, or the identifier of the user who made the submission. It also collects data from the real backend, most notably the duration of the evaluation process, which indicates how long the worker was occupied.

The dataset is split into two CSV files — *regular* jobs and *reference solutions* jobs. Students submit regular jobs while teachers submit reference solutions to verify the exercises are correctly set and determine the suitable time and memory limits. Although both job types are evaluated by the same backend in the real system, we feed only the regular jobs into the simulator for the sake of simplicity. However, we provide the reference jobs metadata so they can be used as additional inputs for the estimator that helps us predict regular job duration based on their exercise affiliation. The number of reference jobs is very low (compared to regular ones), and thus this simplification does not hinder our objective of simulating a realistic workload.

`solution_id`	identifier of the solution which corresponds to one instance of the submitted source code (one solution may be re-evaluated if necessary — i.e., multiple jobs may have the same solution ID)
`group_id`	identifier of a lab group of students the author is attending
`tlgroup_id`	identifier of top-level group which corresponds to a course the author is attending
`exercise_id`	identifier of an exercise being solved
`runtime_id`	identifier of the programming language and runtime environment used in the solution
`worker_group_id`	an abstraction that specifies requirements for special hardware or software the assigned worker must have installed
`user_id`	identifier of the author of the solution
`spawn_ts`	unix timestamp of the job creation
`limits`	sum of time limits imposed on all the tests of the corresponding exercise (which can give us rough estimate for the maximal job duration)
`cpu_time`	bool flag indicating whether CPU time (rather than wall time) was used for the time limits
`correctness`	relative correctness (in $[0, 1]$ range) based on which tests were passed
`compilation_ok`	bool flag indicating whether the solution passed the compilation phase (if not, no tests were actually executed)
`duration`	total time of the evaluation

Table 1: Overview of job metadata properties

The individual properties of a job are summarized in Table 1. The properties in the first part are available the moment the job is spawned, and the properties in the second part are added after the evaluation. All the properties are present in the logs right away, so the author of the simulation experiment must refrain from using the latter properties (like duration) in the dispatching algorithm if the simulation claims to be realistic.

All identifiers in the dataset have been transformed using a hashing function to ensure sufficient anonymization. Internally, the simulation transforms the string hashes to sequentially assigned integers when the data are loaded to simplify their processing further.

Perhaps the most crucial property is the duration of a job since it determines how long it will be occupying the worker. Please note that the duration values collected in the logs are not entirely accurate as they represent the sum of the time consumed by individual evaluation tasks; however, it does not take into account the overhead of dispatching the job, starting the tasks or assembling their results. Furthermore, the duration time depends heavily on the hardware configuration of the workers. Therefore, it may be valid to perform linear transformations of the duration values (where multiplicative constant may simulate hardware of different speeds and additive constant represents the overhead) to simulate realistic yet slightly different scenarios.

4 EXAMPLES AND EVALUATION

We have prepared two examples of experiments to demonstrate the capabilities of the simulator. The first one is a basic tutorial based on the simple utilization of worker queue properties to affect the dispatching process. The second one aims at demonstrating how to integrate machine learning into the self-adaptive process to improve user experience regarding the system latency.

4.1 Example 1: Saving power

The first example draws on a very common problem of saving operational costs (e.g., consumed electric power) whilst maintaining reasonable availability of provided services (mainly the latency of job evaluation). It is based on the assumption that the workers are hosted on independent servers, and each server can be shut down and woken up by the self-adaptive module.

In the simulator, the state of the workers is determined by the `'active'` property of the worker queues. If the queue is active, the dispatcher may enqueue jobs in it; otherwise, it is closed for business, and the corresponding worker is assumed to be shut down.

For the sake of simplicity, we have decided that a queue may be changed to an inactive state only if it does not contain any jobs. Furthermore, the change of the active state is instantaneous in the simulation, which is not entirely realistic from the hardware point of view, but it represents a simple abstraction that demonstrates the main principles of the simulator whilst upholding the objective of this example.

4.1.1 Experimental setup. The dispatcher uses a straightforward static algorithm that operates only with the properties of the queues. For each job, the following steps are performed:

(1) select all active queues $Q_a \subseteq Q$ (Q_a set is expected to have at least one item)
(2) choose q as the shortest queue of Q_a (i.e., the queue with the least jobs)

(3) dispatch job into the queue q

The self-adaptive controller monitors the number of jobs in individual queues and activates/deactivates queues based on the workload. The decision process has only two rules:

(1) If at least one queue contains more than one job (i.e., at least one job is being delayed) and some of the queues in the pool are still inactive, select one inactive queue and make it active.
(2) Otherwise, if no queue contains more than one job and more than one queue is idle (has no jobs), deactivate one idle queue. This rule also ensures that there is always at least one queue active at all times (assuming at least one queue was active at the beginning).

4.1.2 Results. To measure the improvement of a self-adaptive strategy, we need to compare it with a referential baseline. The simulator can be executed without the self-adaptive module (simply omitting it in the configuration) and run only with the static dispatcher. In this case, we use the exact same dispatcher, but (without the SA module) the number of active workers is set at the beginning and fixed for the whole experiment. The two selected baseline runs were conducted with 1 worker (minimum) and 4 workers (maximum) active (the self-adaptive strategy also uses a pool of 4 workers).

There are two criteria being measured — the delay of the jobs and the overall power consumption. In case of delay, we gather statistics for all jobs and then compute average and maximal delay. The power consumption is computed as the time the servers were running (the queues were set as active) relatively to the consumption of one worker (e.g., value 2.0 means that two workers were active on average).

method	avg. delay	max. delay	power
1-worker	311.66 s	36545 s	1.0
4-worker	9.123 s	11130 s	4.0
self-adaptive	14.17 s	13284 s	1.05

Table 2: Results of power saving self-adaptive strategy and two referential baselines

The *1-worker* baseline has the lowest possible power consumption, but the job delays get rather high as they are forced into a bottleneck. The *4-worker* baseline has the lowest delays possible (if the 4 is maximum), but it consumes 4× more power. The *self-adaptive* strategy seems to find a rather good compromise between the two criteria as it uses only 5% more power than the theoretical minimum, and the job delays are only slightly higher than in the case of the *4-worker* setup.

We would like to emphasize that the main objective of this experiment was not to design the best solution possible but rather to demonstrate the simulator using straightforward and comprehensible algorithms. It is not difficult to devise a more elaborate (but also more complex) solution that would yield better results.

4.2 Example 2: Machine learning

The second example aims at improving user experience by focusing on the latency of job evaluation in particular. The inspiration for the metric comes from an observation that simpler assignments are evaluated quickly (so the results will be available almost immediately), whilst complex assignments may take a long time to evaluate (so the users will not wait for them actively). In other words, we calculate the acceptable delay of a job based on its expected duration (the longer the duration, the longer the acceptable delay).

Since the metric would be difficult to understand in terms of absolute numbers, we have defined three categories that are used to label the quality of each evaluation:

- **on time** — the delay is acceptable within the expectations (the delay is less than 1.5× expected duration)
- **delayed** — the job was noticeably delayed, but the delay was still reasonable (the delay is less than 3× expected duration)
- **late** — the delay was quite large and the user may perceive this as a problem (the delay is above 3× expected duration)

The above definitions operate with the term *expected duration*. The expected duration is computed as an average of all preceding jobs of the same exercise and runtime that passed the compilation phase. This value will give us smoother estimates as it filters out failed jobs (which are terminated quickly) and moderates the impact of solutions of lesser quality (which might take longer to evaluate). The actual constants that define the categories were selected based on practical experience from ReCodEx as well as to keep the individual categories reasonably sized w.r.t. data characteristics.

4.2.1 Experimental setup. Unlike in the previous example, all queues remain active for the whole time since the power consumption optimization is not an objective. Each queue has optionally a `'limit'` property which is configured statically and used by the dispatcher to restrict the jobs assignment into the queues based on its expected duration. In particular, we use 4 queues, 3 of which have the limit set to 30 seconds, and one has no limit.

The dispatcher employs the following decision process to assign a job:

(1) compute a duration estimate e of the incoming job based on given *estimation model*
(2) select all active[3] queues Q_a
(3) filter out candidates $Q_c \subseteq Q_a$ which have their *limit* greater than e or which have no *limit* at all
(4) choose q as the shortest queue of Q_c (i.e., the queue with the least jobs)
(5) dispatch job into the queue q

The setup should either make sure that Q_c is always not empty (e.g., by providing at least one queue without limit) or implement some form of fallback for the third step (e.g., that Q_c hold the queues where the limit is exceeded by e the least).

The most important part of this demonstration is the *estimation model* used in step 1. We assume that this model is trained by the self-adaptive controller and expresses the focal point of the configuration that affects the behavior of the system. In the example, we have implemented two models based on the machine-learning paradigm:

- *Simple statistical model*, which is based on the same principle as the evaluation metric — i.e., utilization of history of previously processed jobs (including reference solutions). It

[3]Although in our setup all queues are active the whole time, the algorithm is ready for future alterations.

computes an average duration from jobs that passed compilation categorized by their exercise and runtime affiliations.

- *Neural-network model* uses a standard artificial neural network that is trained as a regression predictor. The neural network uses the same inputs (exercise and runtime identifiers) in hot-one encoding, one hidden layer (ReLU activation), and the final layer with exponential activation.

The neural-network model was implemented in TensorFlow [1], a popular framework for machine learning. One of the intentions of this example is to demonstrate how to utilize well-established tools in combination with our simulator.

4.2.2 Results. The proposed self-adaptive strategies were compared with a static baseline (i.e., dispatcher without a self-adaptive controller) that uses job time limits (divided by 2) to estimate their duration.

As a second baseline, we have run an experiment with an *oracle model* that does not use machine learning but violates the data causality and reads the actual job duration from the log before the job is duly processed (i.e., it simulates an ultimate prediction model by looking into the future). This experiment has no realistic basis, but it gives us the upper bound for other models and helps us determine how much the job duration estimator affects the quality of the user experience.

method	on time	delayed	late	avg. delay
no-SA (base)	382, 615	2461	13, 226	89.47 s
statistics	387, 502	1928	8872	62.09 s
NN regression	387, 216	1928	9158	55.4 s
oracle (base)	388, 244	1940	8118	52.18 s

Table 3: User-experience results for two machine-learning strategies

Table 3 summarizes the results of the two demonstrated machine-learning models and the two baselines. Both implemented models show similar improvement in the user-experience quality over the baseline. Also, both models are quite close to the theoretical maximum for a given worker configuration established by the oracle baseline.

5 RELATED WORK

We went through more than five years of the SEAMS artifacts, and we have not found any closely similar ones. Nevertheless, there are several artifacts that bear at least some similarities. Curiously, all of them target the adaptation (such as scaling, load balancing, etc.) in service-oriented and web-based systems.

K8-Scalar [3] is a workbench for evaluating self-adaptive approaches to auto-scaling container-orchestrated services. It focuses on containerization and monitoring within a cluster but similarly to our artifact, it adapts to the incoming requests to the system. These requests are generated as specified in a workload profile, which contrasts with our artifact, which also contains an extensible real-life dataset.

Another similar artifact is SWIM [8] which simulates a generic multi-tier web application with a load-balancer, multiple web servers, and a database management system. SWIM allows the creation of different adaptation modules that control the adaptation of the system for the incoming requests. It also contains two datasets of recorded real-life workload. These datasets are in the form of a series of numbers specifying how long to wait before sending the next request. This contrasts with our dataset that contains data collected over several years and offers rich meta-data of the requests.

Hogna [2] is a platform for the deployment and self-management of web applications in a cloud. It primarily offers its users to provide their own analyzer and planner components and implement adaptation like load balancing, etc. No dataset is included, and a load generator is employed.

Yet another similar artifact is a Hadoop-Benchmark [11] for evaluating adaptation strategies in applications deployed in Hadoop. Again, adaptation primary targets load balancing, throughput, etc., and existing Hadoop benchmarks are used as datasets.

Tele Assistance System (TAS) [10] is also a related artifact. It offers a platform for the evaluation of adaptation strategies in service-oriented systems, which are composite services composed of multiple atomic services. Adaptation targets choices with the composite services, how the atomic ones should be chosen based on requests, reliability of the atomic services, their cost, etc. Instead of a dataset, a generator of requests (and their QoS parameters) is employed.

6 CONCLUSION

The paper presents a testbed artifact that comprises a simulator of a job-dispatching system and a dataset collected from logs of a real system. The simulator is designed to be highly modular, so it can be easily adapted for various experiments with self-adaptive strategies. Since it replays actual logs, it provides a stable and accurate evaluation of these experiments.

We have implemented two examples: The first is a quite frequent scenario where workers are being shut down or woken up based on the actual workload to save resources. The second is a more complex scenario that focuses on improving the user experience of the system in terms of job processing latency. It was used to demonstrate how to easily combine the testbed with machine learning models and with the TensorFlow framework.

The implemented examples demonstrated the versatility and usefulness of the testbed artifact for further research in this domain. The greatest concern of current implementation (albeit in Python) is performance, especially when the TensorFlow model was used in the dispatcher to make predictions of the job duration. The issue at hand is that the jobs are being dispatched one by one, which limits the performance of TensorFlow since it is oriented on batch processing. We are planning to address this issue in our future work.

The whole artifact and its documentation are publicly available on GitHub (https://github.com/smartarch/simdex).

ACKNOWLEDGMENT

This work was supported by the Czech Science Foundation project 20-24814J and by Charles University institutional funding SVV 260451.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al.

2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 2015. Hogna: A Platform for Self-adaptive Applications in Cloud Environments. In *Proceedings of SEAMS 2015, Florence, Italy*. IEEE Press, 83–87. https://doi.org/10.1109/SEAMS.2015.26

[3] Wito Delnat, Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. 2018. K8-scalar: A Workbench to Compare Autoscalers for Container-orchestrated Database Clusters. In *Proceedings of SEAMS 2018, Gothenburg, Sweden*. ACM, 33–39. https://doi.org/10.1145/3194133.3194162

[4] Tejun Heo. 2015. The Linux kernel Documentation: Control Groups v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[5] Pavel Jezek, Michal Malohlava, and Tomas Pop. 2013. Automated evaluation of regular lab assignments: A bittersweet experience?. In *Proceedings of CSEE&T 2013, San Francisco, CA, USA*. IEEE, 249–258. https://doi.org/10.1109/CSEET.2013.6595256

[6] Jeffrey Kephart and David Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

[7] Martin Kruliš, Jan Buchar, Martin Polanka, and Petr Stefan. 2016. ReCodEx: Code Examiner. https://github.com/ReCodEx

[8] Gabriel A. Moreno, Bradley Schmerl, and David Garlan. 2018. SWIM: An Exemplar for Evaluation and Comparison of Self-adaptation Approaches for Web Applications. In *Proceedings of SEAMS 2018, Gothenburg, Sweden*. ACM, 137–143. https://doi.org/10.1145/3194133.3194163

[9] Henry Muccini and Karthik Vaidhyanathan. 2019. A Machine Learning-Driven Approach for Proactive Decision Making in Adaptive Architectures. In *Companion Proceedings of ICSA 2019, Hamburg, Germany*. 242–245. https://doi.org/10.1109/ICSA-C.2019.00050

[10] Danny Weyns and Radu Calinescu. 2015. Tele Assistance: A Self-adaptive Service-based System Examplar. In *Proceedings of SEAMS 2015, Florence, Italy*. IEEE Press, 88–92. https://doi.org/10.1109/SEAMS.2015.27

[11] B. Zhang, F. Krikava, R. Rouvoy, and L. Seinturier. 2017. Hadoop-Benchmark: Rapid Prototyping and Evaluation of Self-Adaptive Behaviors in Hadoop Clusters. In *Proceedings of SEAMS 2017, Buenos Aires, Argentina*. 175–181. https://doi.org/10.1109/SEAMS.2017.15