MockSAS: Facilitating the Evaluation of Bandit Algorithms in Self-Adaptive Systems

Elvin Alberts[1], Ilias Gerostathopoulos[1], and Tomas Bures[2]

[1] Vrije Universiteit Amsterdam, The Netherlands
[2] Charles University in Prague, Czech Republic

Abstract. To be able to optimize themselves at runtime even in situations not specifically designed for, self-adaptive systems (SAS) often employ online learning that takes the form of sequentially applying actions to learn their effect on system utility. Employing multi-armed bandit (MAB) policies is a promising approach for implementing online learning in SAS. A main problem when employing MAB policies in this setting is that it is difficult to evaluate and compare different policies on their effectiveness in optimizing system utility. This stems from the high number of runs that are necessary for a trustworthy evaluation of a policy under different contexts. The problem is amplified when several policies and several contexts are considered. It is however pivotal for wider adoption of MAB policies in online learning in SAS to facilitate such evaluation and comparison. Towards this end, we provide a Python package, MockSAS, and a grammar that allows for specifying and running mocks of SAS: profiles of SAS that capture the relations between the contexts, the actions, and the rewards. Using MockSAS can drastically reduce the time and resources of performing comparisons of MAB policies in SAS. We evaluate the applicability of MockSAS and its accuracy in obtaining results compared to using the real system in a self-adaptation exemplar.

Keywords: self-adaptive systems · multi-armed bandits · profiling

1 Introduction

Self-adaptive systems (SAS) are able to adapt to changes in their environment and internal state to ensure or optimize a number of adaptation goals related to e.g. application performance, resource consumption, and failure avoidance [12]. To make them capable to react to different situations at runtime, even to those unanticipated at design time, SAS can be equipped with an online learning loop that performs different adaptation actions, monitors their effect in terms of the overall utility of the system after each action is performed, and selects the actions that maximizes such utility [9, 15].

One possible way to implement online learning in SAS is to rely on multi-armed bandit (MAB) policies [13]. Such policies are a class of reinforcement learning algorithms that can be used for sequential decision making between a set of options. An MAB policy can be used to select an action out of a number

of discrete options (called *arms*) by balancing exploration – picking an arm to learn its utility (called *reward*) – and exploitation – picking the so-far best arm to maximize the cumulative utility. Since this process maps well to a SAS trying actions at runtime to observe their effect and optimize its performance, MAB policies are a natural candidate to consider for online learning in SAS. At the same time, different MAB policies – including ϵ-greedy, different variants of Upper Confidence Bound (UCB), and Explore-Then-Commit – exist and several have already been proposed for online learning in SAS [6, 14, 17, 1]. Each policy, or policy variant/configuration, performs better compared to other policies in certain operation scenarios (e.g. high variance in reward, frequent changes in context) and worse in others.

The problem we have encountered when employing MAB policies for online learning is that it is difficult to evaluate the effectiveness of different policy types (e.g. greedy, stochastic, contextual) in optimizing the overall system utility, and, consequently, to compare them based on such effectiveness and select the best policy for a given SAS [1]. The main problem stems from the fact that the evaluation needs to consider the stochasticity of the environment (e.g. different situations that the SAS may reside in), of the SAS itself (e.g. different processing times due to resource overload), and of the MAB policies themselves (e.g. randomization inherent in the logic of certain policies). To back up any evaluation/comparison results in this complicated setting with statistical confidence, one therefore needs to perform a huge number of runs covering all the foreseeable system and environment states. For instance, in a typical case one would need to compare e.g. five different MAB policies over three system contexts by performing 30 runs, each of which consists of 100 rounds (where each round evaluates an action and may last for one minute). This yields 5 * 3 * 30 * 100 = 45,000 mins or 31,25 days of evaluation time (in real time when the runs are not parallelized). Clearly, this is both a time-consuming and resource-intensive process – while at the same time essential for the wider adoption of MAB policies in SAS.

To tackle the above problem, we propose to perform evaluations and comparisons of MAB policies on a *profile* of a SAS that captures the relations between the contexts, the actions, and the rewards. This profile acts similar to a mock object in unit testing: it helps isolating the behavior of the object under test (MAB policy) by simulating the behavior of the real object it depends on (SAS), when it is impractical to directly use the latter. To create such a profile, different metrics of the real system need to be measured under different contexts and actions and derive statistical distributions that can be used for generating rewards in each system state that corresponds to a (context, action) pair. However, it is important to note that this profiling needs to be done only once and can be reused for the evaluation of different MAB policies, saving time and resources. Additionally, it potentially allows non-measured contexts to be extrapolated and included in the profile should a complete recreation of the factors evaluating actions be not feasible.

The contribution of this paper is to describe a Python package, called MockSAS, and related grammar that we developed for (i) specifying profiles of SAS,

and (ii) using them in a loop with a MAB policy for fast evaluations. In particular, we provide a first version of the process that allows for defining profiles of SAS using the MockSAS grammar, and demonstrate its applicability on a self-adaptation exemplar (Section 3). Finally, we evaluate the accuracy of the MAB policy comparison results obtained when using a profile versus the real SAS on the same self-adaptation exemplar (Section 4).

2 Background

2.1 Multi-Armed Bandits

Multi-armed bandit (MAB) algorithms or *policies* are a class of reinforcement learning (RL) algorithms which deal with choosing between a set of K options called *arms* [13, 18]. Formally, this setting corresponds to a Markov Decision Process with a single state and K actions, while compared to the general RL setting, actions in an MAB policy are assumed to not influence future states of the environment. In this setting, an MAB policy balances *exploration* with *exploitation*: it tries to explore arms to gain knowledge about them while at the same time using the best-known arm regularly to exploit the knowledge it has already gained.

In particular, each arm has an associated reward (or payoff) whose value at a time t is not known prior to selecting (*"pulling"*) it. Arms are selected sequentially and their rewards are gradually revealed; an MAB policy prescribes with arm should be selected at each round. MAB policies try to minimize the *regret* they incur, i.e. the loss in performance by using them compared to the optimal policy of playing the best arm at each round. Equivalently, they try to maximize the cumulative reward. Formally, given K arms and sequences of rewards $X_{i,1}$,$X_{i,2}$,... associated with each arm i, the regret R_n of a policy after n plays I_1, ..., I_n is [5]

$$R_n = \max_{i=1,\dots,K} \sum_{t=1}^{n} X_{i,t} - \sum_{t=1}^{n} X_{I_t,t}$$

Different MAB policies can be categorized according to their assumptions on (i) their application horizon, (ii) the nature of the reward process, and (iii) the setting they are applied to. Depending on the application horizon, policies can be categorized as *anytime* and *fixed-horizon*. Depending on the assumed nature of the reward process, MAB policies are categorized into *stochastic*, *adversarial*, and *Markovian* [5]. Depending on their application setting, we distinguish between *contextual* (or associative) and *non-contextual* bandits.

2.2 Online learning with MABs in Self-Adaptive Systems

A SAS is typically composed of a managed system that is responsible for the business goals of the system and a managing system that is responsible for

its adaptation goals (e.g. optimizing its performance). As depicted in Fig. 1, the managing system typically follows the well-known Monitor-Analyze-Plan-Execute over Knowledge (MAPE-K) loop [8]. The Monitor phase collects data regarding the managed system and its environment, Analyze decides based on the data whether an adaptation is needed, Plan identifies the best adaptation action to perform and prepares a plan for it, and Execute performs the adaptation action to change the managed system. In this setting, online learning with MAB policies can be part of the Plan phase: when a SAS is in a situation for which the best action is unknown, it launches an online learning cycle. In this cycle, an MAB policy is used to select an action (arm) that is applied to the managed system. The reward for an action is calculated by combining different monitored attributes of the managed system into a utility value. Based on the reward for an action, a next action is selected from the policy and applied. An online learning cycle ends by either having the policy converge to an action or reaching a maximum number of rounds.

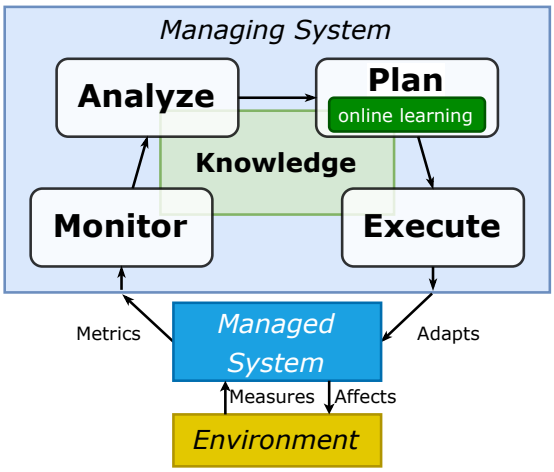

Fig. 1: MAPE-K with online learning within the Plan phase.

2.3 Running Example: SWIM

Simulator for Web Infrastructure and Management (SWIM) is a SEAMS artefact [16] representing the management of server infrastructure for a fictitious web location. The simulator allows real web traces to be replayed in simulated time, making it possible to adapt to hours of web traffic in a span of minutes. The adaptation goals of SWIM are to minimize the average response time for requests while at the same time also minimize the infrastructure cost and maximize the profit made by serving advertisements within the served requests. To reach these goals, SWIM is equipped with two adaptation mechanisms: (i) changing

the number of servers used to serve requests, and (ii) changing the dimmer value that controls the percentage of responses that contain advertisements. SWIM's overall utility is based on the degree of satisfaction of its adaptation goals. An adaptation action – arm in MAB – sets the number of servers or the dimmer value. In our experiment in Section 4 we only focus on the first type of actions.

3 Approach

3.1 MockSAS Process

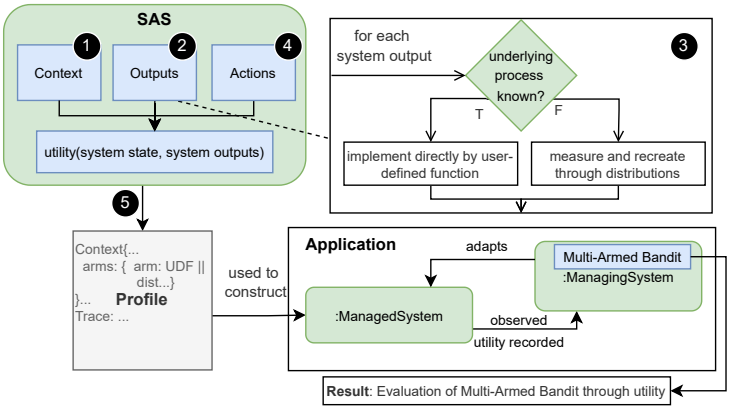

Fig. 2: Process for creating MockSAS profiles and evaluating MAB policies.

In order to facilitate the evaluation of MAB policies in online learning in SAS we have created MockSAS. Fig. 2 illustrates the process of using MockSAS to profile a SAS – in particular its *managed system* subsystem (Fig 1) – as well as how the profile is used in evaluating MAB policies. The required steps of the process for creating a profile are as follows:

1. Determine the system context(s) to be profiled. A profile consists of a series of contexts distinguished by features which when combined with the current system configuration constitute a system state. Contexts are operating scenarios of the system where the utility one or more system states differs significantly. For instance, in SWIM a context feature is the number of requests received per second.
2. Identify which outputs influence the calculation of the reward. Outputs are the metrics we assume the managed system generates depending on its state.
3. Of the identified outputs, determine which require profiling and are thus *profiled outputs*. Some outputs may have an established relationship with context features which can be directly implemented instead of profiled.

4. Identify the arms (adaptation actions) per context whose effects on the reward should be measured. A system may have more adaptation actions than are applicable to the chosen contexts.
5. For each arm in each context, collect sufficient values of the profiled outputs to be able to identify their distribution.

```
1  SixtyRequests{
2    features: {
3          rq_rate: uniform(54,66)
4    }
5    arms: {
6      ThreeServers:    utilitySWIM(rq_rate, 1.0, normal(.064,.009), 13, 3)
7      SixServers:      utilitySWIM(rq_rate, 1.0, normal(.039,.001), 13, 6)
8      EightServers:    utilitySWIM(rq_rate, 1.0, normal(.039,.001), 13, 8)
9      ElevenServers:   utilitySWIM(rq_rate, 1.0, normal(.039,.001), 13, 11)
10     ThirteenServers: utilitySWIM(rq_rate, 1.0, normal(.039,.002), 13, 13)
11     } }
12
13 EightyRequests{
14   features: {
15     rq_rate: uniform(72,88)
16   }
17   arms: {
18     ThreeServers:    utilitySWIM(rq_rate, 1.0, normal(1.68,1.44), 13, 3)
19     SixServers:      utilitySWIM(rq_rate, 1.0, normal(.041,.001), 13, 6)
20     EightServers:    utilitySWIM(rq_rate, 1.0, normal(.039,.0004), 13, 8)
21     ElevenServers:   utilitySWIM(rq_rate, 1.0, normal(.039,.001), 13, 11)
22     ThirteenServers: utilitySWIM(rq_rate, 1.0, normal(.039,.001), 13, 13)
23   }
24 }
25 Trace: (SixtyRequests, 166) (EightyRequests,166)
```

Fig. 3: MockSAS profile of SWIM.

The listing in Figure 3 shows a profile created of SWIM using the implemented profile grammar included in MockSAS. We will use this listing to describe each part of the process in greater detail.

Step 1: Determining the system context(s) to be profiled. We choose to discretize the operating environment a SAS faces into contexts. The implication is that each distinct context influences the rewards per arm. The listing defines two contexts, `SixtyRequests` and `EightyRequests`. Contexts may or may not have measurable features which distinguish them. When these are present we specify them as features of the context such as in lines 2-4. These features in turn may

play a role in the reward as will be covered in steps 2 and 3. To determine system contexts thus requires observing how a system's reward changes over time. For SWIM it is straight-forward to distinguish contexts by the arrival rate of requests. Taking the context `SixtyRequests` as an example, for realism the request rate is a uniform distribution of 10% below and above the average of 60 request/s as defined on line 3. Within this range of requests/s the rewards achieved per arm are similar enough to be grouped as one context. Line 25 shows the trace of contexts; this simply specifies how many 'rounds' (adaptation opportunities) each context is active for and in what order. The total number of rounds across all contexts in a MockSAS trace determine the overall duration of the trace.

Step 2: Identifying outputs. A system may have numerous outputs that are continuously monitored. A subset of these outputs can be used to evaluate a system state's utility. The utility of the state resulting from an adaptation is its reward. For SWIM, a utility function was already defined and used in lines 6-10 and 18-22 to define the reward per arm. Should the system to be profiled not have a utility function defined, one can do so based on its requirements and the goals that stem from it [10]. Depending on a system utility function, different outputs may parameterize it. Those which do should be identified. The signature of the utility function used in our listing is `utilitySWIM(arrival_rate, dimmer,avg_response_time,max_servers,servers)`. The `arrival_rate` as covered is specified on line 3 of the listing. This is implemented through the user-defined function functionality of the grammar which allows any function written in Python to supply utility values. The `dimmer` and `servers` are both determined by the specific arm, the former is held constant and represents optional content served in responses. The `response_time` is the latency in serving those responses. Lastly, `max_servers` represents the maximum servers a SWIM adaptation can result in using, in our 13 as it is the max of the arms.

Step 3: Profiled outputs. There may be an established relationship between system outputs and the feature of a context. Further, some outputs may directly stem from the system state by way of the current configuration. If we look at line 6 of the listing, the second, penultimate, and last parameter of the utility function can be directly derived from the context features and the fact that the adaptation has three servers (value of the last parameter). In particular, 1.0 represents that the 'dimmer' value is kept constant, and 13 the maximum servers across adaptations. Similarly, the request rate can be used directly as it is a known context feature. Contrarily, the process behind response time is not precisely traceable for SWIM, so response time is a profiled output.

Step 4: Identify arms. A system may have myriad potential adaptation actions. However, not every action is appropriate to every situation. Simultaneously, the fewer actions there are, the more efficiently a MAB policy can perform. Thus it is in the interest of the user to minimize the number of adaptation actions by identifying those appropriate. For the listed profile, five adaptation actions,

3, 6, 8, 11 and 13 servers are identified for each context. This is as for the low end of the `SixtyRequests` context 3 servers can be sufficient, while up to 13 are required to deal with the high end of the `EightyRequests` context.

Step 5: Profile arms per context. Now that the desired contexts and arms have been identified, they can be profiled for the necessary profiled outputs. In the case of SWIM this means subjecting each number of servers to the `SixtyRequests` and `EightyRequests` context and measuring the response times outputted. This was done for a sufficient amount of time to accurately determine the mean and standard deviation of the response time. We specify this through a normal distribution in each arm definition e.g. line 6 `normal(.064,.009)`. It should be noted that we did have prior knowledge the distribution would be Gaussian. We do not yet provide a general-purpose solution for profiled outputs of which the underlying distribution is not known prior although the grammar allows for specifying different distributions (e.g. normal, logistic, constant).

3.2 Application

MockSAS is implemented as a Python package which consists of abstractions of MAPE-K's managed system, managing system and environment and a grammar within which to specify profiles. The grammar is developed using the Lark Python library [7]. The source code of MockSAS can be found in a GitHub repository[3]. MockSAS accepts defined profiles as input. A `ManagedSystem` abstraction is constructed with a given profile as its parameter. The instance of the `ManagedSystem` is assigned to an instance of the `ManagingSystem` to be managed by it. This `ManagingSystem` is itself constructed with an instance of a MAB policy, provided by the MASCed_bandits library[4] developed in our previous work [2]. After the necessary objects have been constructed, the operation loop of the MockSAS begins. Initially the `ManagedSystem` observes its environment should one be specified by its profile. Then it notifies its `ManagingSystem` through the Observer pattern that an adaptation is required, simulating the result of the 'Analyze' step in MAPE-K. The `ManagingSystem` queries MAB policy for a new action. The policy takes as input the reward/utility observed for the previous action. This reward is gathered from the generator which has been created during parsing the profile. By using generators, the rewards as well as context features resulting from functions are only calculated on-demand making the implementation resource-efficient. The policy uses all the rewards received in its lifetime to advise which adaptation/arm should be enacted next. This loop continues until the end of the trace specified in the profile, or in the case of an indefinite trace until the user interrupts it.

[3] https://github.com/EGAlberts/MockSAS
[4] https://github.com/EGAlberts/MASCed_bandits/

4 Evaluation

4.1 Setup

We have designed one experiment in which MAB policies are evaluated and perform it using both the actual SWIM exemplar and our profile based on it as described in Figure 3. In doing so we seek to answer how accurately the evaluation of the policies done through MockSAS matches that of the actual SWIM exemplar.

Setting The experiment uses the trace as specified at the end of the profile in Figure 3 as its setting. There are 166 rounds of a server load of around 60 request/s followed by another 166 of around 80 requests/s. This number of rounds provides ample opportunity for all the policies we evaluate to determine the optimal arm (with the highest average reward). The range of response times per arm within each context have been measured prior to the experiment and used to define the profile used for the MockSAS portion. This is reflected in the `normal(mean,stdev)` function seen throughout the reward specification for arms in the profile.

MAB Policies For the experiments four different MAB policies are used; when considering different hyperparameter values in total 15 policies are run. We will now describe each policy and provide the rationale for its inclusion.

- **ϵ-greedy**: ϵ-greedy is a classical solution to the exploration-exploitation trade off. The policy serves as a baseline, with a fixed rate of exploration being specified through its *epsilon* hyperparameter e.g. with $\epsilon = 0.8$ has an 80% chance of exploring in a given round. At each round it either does this exploration, or exploits by choosing the arm with maximum average reward so far. As a baseline, it is useful in evaluating more complex policies against what is a naive and simple solution.
- **UCB-Tuned**: UCB, Upper Confidence Bound, and its tuned variant is a well-cited MAB policy known for its relatively high performance guarantees [3]. The policy constructs a confidence bound on the true mean of each arm, using the distance between the means per arm to ascertain with high certainty which arm provides an optimal reward. It balances exploiting the arm with the highest certainty of being optimal against exploring arms about which there is more uncertainty. Its ubiquity and expected performance motivate its inclusion in our experiment. UCB-Tuned does not require a hyperparameter.
- **EXP3**: EXP3 [4] is a policy similar to UCB, with the key difference being that it does not assume there to be a probability distribution generating the rewards. Instead, rewards are presumed to be chosen by an adversary. Rather than ascertaining confidence, EXP3 instead uses a exponentially weighted distribution with weights provided by observed rewards per arm. At each

round it samples this distribution to choose the next arm. As EXP3 makes fewer assumptions than UCB it is of interest especially in application to SAS (when considering a desire for general-purpose solutions). We use one hyperparameter which is based on the total number of rounds the policy expects to face.

- **Discounted UCB (DUCB)**: DUCB [11] is a variation on UCB to handle *non-stationary* environments. For our purposes, these are environments with multiple contexts which cause an abrupt change in the expected rewards per arm. The policy tackles this by discounting rewards as they age, biasing more recent observations. As the environment we specify for the experiment contains multiple contexts, it is of interest to evaluate DUCB's performance relative to policies designed for stationary environments (all the other policies in our experiment). The hyperparameter *gamma* of DUCB controls the degree of discounting of old rewards; the closer it comes to 1, the closer DUCB approximates UCB-Tuned. From our previous work [2] we know that small variations of *gamma* can have a large effect on the performance of DUCB.

We have implemented these policies among others in a Python library MASCed_bandits in our previous work [2]. We make use of this library for the experiment on both platforms.

SWIM Exemplar As established through prior work [2, 1], we use an extension developed to SWIM which allows Python libraries to act as an 'AdaptationManager'. Through this extension, each MAB policy is used to implement the Plan phase of the MAPE-K-based adaptation logic for the system. Using this adaptation logic, 30 runs with differing randomization seeds are done per policy. The results of these runs are averaged and compiled in Table 1. The 30 runs are chosen as a tradeoff of statistical significance and time feasibility. Although SWIM can make use of simulated time, conducting 30*15 runs of more than 300 rounds each (with 60 simulated seconds per run), 120 minutes of real time were necessary to complete the experiment.

MockSAS We setup the experiment in MockSAS to recreate the conditions of the SWIM exemplar closely. A Python script simply loops through 30 different randomization seeds, similar to the SWIM case. Within each iteration, for each of the 15 policies a MockSAS object is instantiated, run for the duration of the trace, and its results are recorded. These results are averaged over the 30 runs and presented in Table 1. For the same number of runs as in SWIM, the experiment in MockSAS took 10 mins, only 8% of the time needed for SWIM.

4.2 Results

Table 1 compiles the results of the experiment on both platforms. The table is sorted by the median reward achieved by each policy on SWIM across the 30

MAB Policy	Hyper-parameter	SWIM Median Reward	MockSAS Median Reward	SWIM Position	MockSAS AVG Position	AVG Match
UCB-Tuned	-	0.75	0.76	1	1.13	86.67%
ϵ-greedy	$\epsilon = 0.2$	0.73	0.73	2	2.97	36.67%
DUCB	$\gamma = 0.997$	0.71	0.73	3	2.53	46.67%
	$\gamma = 0.995$	0.69	0.71	4	3.83	56.67%
ϵ-greedy	$\epsilon = 0.4$	0.68	0.69	5	5.4	33.33%
EXP3-FH	$h = 333$	0.67	0.66	6	8.03	10.0%
DUCB	$\gamma = 0.992$	0.66	0.69	7	5.53	6.67%
	$\gamma = 0.99$	0.65	0.68	8	7.07	23.33%
ϵ-greedy	$\epsilon = 0.6$	0.65	0.66	9	8.73	53.33%
	$\epsilon = 0.8$	0.61	0.62	10	11.27	16.67%
DUCB	$\gamma = 0.97$	0.6	0.63	11	10	13.3%
	$\gamma = 0.95$	0.58	0.62	12	11.6	60.0%
ϵ-greedy (Random)	$\epsilon = 1.0$	0.57	0.58	13	14.9	3.33%
DUCB	$\gamma = 0.92$	0.57	0.6	14	13	10.0%
	$\gamma = 0.89$	0.56	0.59	15	14	6.67%

Table 1: MockSAS v SWIM Experiment Results

runs, where the reward refers to the average reward achieved by the policy for a run irrespective of which choices were made. This entails that not only choosing the optimal arm is emphasized (as a convergence measure would) but also how lucrative its non-optimal choices were. The 'MockSAS average position' refers to the position each policy had at the end of a run relative to one another. This indicates in what range of positions policies were evaluated. This can be contrasted with the position held statically after the 30 SWIM runs ('SWIM Position' column). If a policy mismatches SWIM yet has proximity in its average position that inconsistency is less significant. The average match percentages in the last column acts towards the same end. Whenever at the end of a run a policy's position was the same as in SWIM, this is considered a match. For instance, in 86.67% of runs (26 out of 30), UCB-Tuned is 'correctly' ranked by MockSAS at the same position as in SWIM (i.e. first).

The results show that when it comes to the top performing policy, UCB-Tuned, there is heavy agreement between MockSAS and SWIM. Similarly, DUCB's variants enjoy the same relatively high agreement. This can be explained by their being based on the original UCB. These are also the only two policies which enjoy determinism in their decision logic. This may go towards explaining their consistency with SWIM as they also have internal consistency i.e. the variance in their performance is generally smaller than that of non-deterministic policies. This argument is further supported by the fact that the policy which makes the most random choices, $\epsilon - greedy$ with a 1.0 (maximal) exploration factor matches position for SWIM the fewest times. The margin in median reward between the lowest performing policies is quite small across both platforms, which

also causes less consistency. The distributions which generate randomness in the reward could not be seeded and can account for the small margins seen. These margins can also unpredictably influence the internal rankings of each platform.

From the result we can conclude that for a majority of deterministic policies MockSAS accurately reflects evaluation by the real system SWIM. For the remaining deterministic policies their extreme proximity in median reward to non-deterministic policies obfuscates the matching in overall position. With the elimination of these from ranking even those deterministic policies with low matching to SWIM would be accurately ranked. For the random policies further work is required to control for the diverging randomization processes between the two platforms. Should the experiment be run with equivalent seeds for each of the 30 runs for the MAB policies, the consistency may see an increase.

5 Conclusion

In this paper we propose a process for creating profiles of SAS to facilitate the evaluation of online learning policies, specifically MAB policies. We demonstrate how one can profile a SAS through the running example of SWIM. To enable this process we implemented a Python package MockSAS, with an associated grammar to parse defined profiles. MockSAS simulates the interaction between the MAB policies and the managed system of SWIM and reports on their performance. We used this for our experiment where we measure how accurately the profile of SWIM in MockSAS evaluates MAB policies compared to the real system SWIM. Our results show that for deterministic policies the MockSAS profile can accurately reproduce the same ranking of policies as SWIM. While for non-deterministic policies, more has to be done to control the random process before its accuracy can be truly determined.

For future work, a general-purpose solution should be found for handling profiled outputs, even when the type of underlying distribution is unknown. As it stands, we make use of detailed knowledge of SWIM to recreate the response time but this cannot always be assumed to be possible. Besides this, we see a possibility of extending the use of MockSAS (or only its process) for other online learning solutions than MAB policies.

References

1. Alberts, E., Gerostathopoulos, I.: Measuring convergence inertia: Online learning in self-adaptive systems with context shifts. In: Proceedings of the 2022 International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, to appear (Oct 2022)
2. Alberts, E.G.: Adapting with Regret: Using Multi-armed Bandits with Self-adaptive Systems. Master's thesis, University of Amsterdam (2022), https://scripties.uba.uva.nl/search?id=727497
3. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning **47**(2), 235–256 (May 2002). https://doi.org/10.1023/A:1013689704352

4. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: The Nonstochastic Multiarmed Bandit Problem. SIAM Journal on Computing **32**(1), 48–77 (Jan 2002). https://doi.org/10.1137/S0097539701398375
5. Bubeck, S.: Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. Foundations and Trends® in Machine Learning **5**(1), 1–122 (2012). https://doi.org/10.1561/2200000024
6. Cabri, G., Capodieci, N.: Applying Multi-armed Bandit Strategies to Change of Collaboration Patterns at Runtime. In: 2013 1st International Conference on Artificial Intelligence, Modelling and Simulation. pp. 151–156. IEEE, Kota Kinabalu, Malaysia (Dec 2013). https://doi.org/10.1109/AIMS.2013.31
7. Erezsh: Lark parser. https://github.com/lark-parser/lark (2022)
8. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer **36**(1), 41–50 (2003)
9. Kim, D., Park, S.: Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 76–85 (May 2009). https://doi.org/10.1109/SEAMS.2009.5069076, iSSN: 2157-2321
10. Kim, D., Park, S.: Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 76–85 (2009). https://doi.org/10.1109/SEAMS.2009.5069076
11. Kivinen, J., Szepesvári, C., Ukkonen, E., Zeugmann, T. (eds.): Algorithmic Learning Theory: 22nd International Conference, ALT 2011, Espoo, Finland, October 5-7, 2011. Proceedings, Lecture Notes in Computer Science, vol. 6925. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24412-4
12. Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C.: A survey on engineering approaches for self-adaptive systems. Pervasive and Mobile Computing **17**, 184–206 (Feb 2015). https://doi.org/10.1016/j.pmcj.2014.09.009
13. Lattimore, T., Szepesvári, C.: Bandit Algorithms. Cambridge University Press, 1 edn. (Jul 2020). https://doi.org/10.1017/9781108571401, https://www.cambridge.org/core/product/identifier/9781108571401/type/book
14. Lewis, P.R., Esterle, L., Chandra, A., Rinner, B., Yao, X.: Learning to be Different: Heterogeneity and Efficiency in Distributed Smart Camera Networks. In: 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems. pp. 209–218. IEEE, Philadelphia, PA, USA (Sep 2013). https://doi.org/10.1109/SASO.2013.20, http://ieeexplore.ieee.org/document/6676508/
15. Metzger, A., Quinton, C., Mann, Z.A., Baresi, L., Pohl, K.: Feature-Model-Guided Online Learning for Self-Adaptive Systems. arXiv:1907.09158 [cs] **12571**, 269–286 (2020). https://doi.org/10.1007/978-3-030-65310-1_20, arXiv: 1907.09158
16. Moreno, G.A., Schmerl, B., Garlan, D.: SWIM: an exemplar for evaluation and comparison of self-adaptation approaches for web applications. In: Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems. pp. 137–143. ACM, Gothenburg Sweden (May 2018). https://doi.org/10.1145/3194133.3194163
17. Porter, B., Rodrigues Filho, R.: Distributed Emergent Software: Assembling, Perceiving and Learning Systems at Scale. In: 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). pp. 127–136 (Jun 2019). https://doi.org/10.1109/SASO.2019.00024, iSSN: 1949-3681

18. Slivkins, A.: Introduction to Multi-Armed Bandits. Foundations and Trends® in Machine Learning **12**(1-2), 1–286 (Nov 2019). https://doi.org/10.1561/2200000068, http://www.nowpublishers.com/article/Details/MAL-068, publisher: Now Publishers, Inc.