Emergent Web Server: An Exemplar to Explore Online Learning in Compositional Self-Adaptive Systems

Roberto Rodrigues Filho
Federal University of Goiás
Brazil
robertovito@ufg.br

Elvin Alberts
Vrije Universiteit Amsterdam
The Netherlands
e.g.alberts@student.vu.nl

Ilias Gerostathopoulos
Vrije Universiteit Amsterdam
The Netherlands
i.g.gerostathopoulos@vu.nl

Barry Porter
Lancaster University
United Kingdom
b.f.porter@lancaster.ac.uk

Fábio M. Costa
Federal University of Goiás
Brazil
fmc@inf.ufg.br

ABSTRACT

Contemporary deployment environments are volatile, with conditions that are often hard to predict in advance, demanding solutions that are able to learn how best to design a system at runtime from a set of available alternatives. While the self-adaptive systems community has devoted significant attention to online learning, there is less research specifically directed towards learning for open-ended architectural adaptation – where individual components represent alternatives that can be added and removed dynamically. In this paper we present the Emergent Web Server (EWS), an architecture-based adaptive web server with 42 unique compositions of alternative components that present different utility when subjected to different workload patterns. This artefact allows the exploration of online learning techniques that are specifically able to consider the composition of logic that comprises a given system, and how each piece of logic contributes to overall utility. It also allows the user to add new components at runtime (and so produce new composition options), and to remove existing components; both are likely to occur in systems where developers (or automated code generators) deploy new code on a continuous basis and identify code which has never performed well. Our exemplar bundles together a fully-functional web server, a number of pre-packaged online learning approaches, and utilities to integrate, evaluate, and compare new online learning approaches.

ACM Reference Format:
Roberto Rodrigues Filho, Elvin Alberts, Ilias Gerostathopoulos, Barry Porter, and Fábio M. Costa. 2022. Emergent Web Server: An Exemplar to Explore Online Learning in Compositional Self-Adaptive Systems. In *Proceedings of The 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2022).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

SEAMS 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Self-adaptive systems need to monitor their environment and state and change their behavior to improve their non-functional objectives: prolong their lifespan, optimize resource usage, and minimize cost of operation, to name a few. Such changes are typically enforced at runtime by suitable tactics developed by engineers at design time. However, contemporary self-adaptive systems must deal with conditions that are hard to predict in advance and devise tactics for. A promising alternative to handle such conditions is to have the systems themselves learn suitable tactics at runtime – a research direction known as *online learning*.

According to a recent survey on the application of machine learning in adaptive systems [10], online (or interactive) learning [12] has been targeted by a range of research that applies reinforcement learning (RL) to determine the best tactic at runtime. The majority of these approaches use Q-learning, a particular variant of model-free RL (e.g. [2, 14, 28]), while other approaches such as State-Action-Reward-State-Action (SARSA) [1, 27] and Multi-Armed Bandits (MAB) [17, 22, 23] have also been employed. Online learning has also been tackled from the perspective of online search and optimization, with approaches using, e.g., genetic algorithms [8, 15] to search the space of possible tactics at runtime by applying them and measuring their utility.

Drawing from these works and from our own experience, we have identified a number of distinct and overarching challenges for online learning in self-adaptive systems, namely: (1) How to select and tune a specific algorithm for a class of self-adaptive systems? (2) How to deal with the combinatorial explosion of state-action space in architecture-based self-adaptive systems? (3) How to deal with non-stationary environments that make learning and convergence hard? (4) How to evolve the knowledge built by an online learning algorithm to consider the addition of new tactics or removal of existing ones, as new components are added / removed?

To tackle the above challenges in the context of architecture-based self-adaptive systems, researchers need a common platform on which to apply, tune, evaluate and compare their approaches for online learning. Such a platform should provide the basic abilities of representing different self-adaptation tactics, applying them at runtime, and measuring their utility or reward. Ideally, it also should offer configuration options that help researchers focus their research on particular settings, e.g. non-stationary environments or runtime evolution of the available tactics.

In response, in this paper, we describe the Emergent Web Server (EWS), a self-adaptive web server that can play the role of such a platform. In EWS, tactics take the form of alternative architectural configurations – compositions of individual components that make up the logic of the web server. Each composition provides a different utility (measured as average response latency) when subjected to different workload patterns. EWS provides a total of 42 unique compositions, but this list can be extended by introducing new components to the system that enhance its envelope of adaptability.

EWS is built on a powerful component-based language (Dana [21]) that allows for runtime component hot-swaps with low-overhead and promotes the design of systems via individual components that feature well-defined interfaces. At the same time, it comes with both an HTTP and a Python API that allow for (1) obtaining and changing the current composition, (2) obtaining a list of all available compositions, (3) adding and removing components on the fly, (4) adding and removing monitoring probes on the fly, (5) obtaining the overall utility over a window of time.

In this paper, we present the architecture of EWS (Sec. 3), its HTTP and Python APIs (Sec. 3.2 and Sec. 3.3), explain how it can be used and extended by other self-* researchers (Section 4) and our experience with using the exemplar (Section 5).

2 RELATED WORK

The self-adaptive systems community has already collected 27 exemplars that support research in self-adaptation[1]. Their domains span from cloud, web and service-based systems [4, 20, 29, 30] to traffic systems [9, 25], cyber-physical systems [16, 18, 19] and IoT [13, 24]. We specifically compare EWS to the SWIM exemplar [20] as the most closely related one, since it (i) belongs to the same application domain (web), and (ii) has also been used in evaluating online learning in adaptive systems (e.g. in [15]).

SWIM is a simulation of a multi-tier web application such as ZNN.com [7]. Its objectives are to (i) serve requests within a predefined latency threshold, (ii) minimize the infrastructure cost (cost of running servers), (iii) maximize the revenue by serving optional content (advertisement) with each request. Clearly, these three goals are in conflict: using more application servers may reduce the response latency at the expense of extra infrastructure cost, while serving more optional content increases revenue at the expense of increased latency. To resolve this conflict at runtime, SWIM provides two actuators – actions that can be performed at runtime: (i) increase/decrease the number of application servers, and (ii) increase or decrease a dimmer value that controls how much optional content is served. A configuration of SWIM (number of servers, dimmer value) can be evaluated at runtime, within an online learning context, by retrieving a utility value which combines the experienced latency, the perceived revenue and the infrastructure cost in a single value.

Like SWIM, EWS also offers the possibility of evaluating configurations at runtime, and can be configured for different load levels and load types. However, since in EWS individual components that comprise a web server (such as stream processors, cache components, or hash tables) can be combined together in different ways and even added and removed on the fly, EWS offers the ability to evaluate learning approaches that are specifically directed towards learning for open-ended architectural adaptation. This provides an even richer landscape for the evaluation of online learning approaches, which for example may analyse the specific utility of each component in a composition.

[1] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

3 EMERGENT WEB SERVER

EWS is a fully functioning component-based web server able to serve resources to clients issuing *HTTP 1.0*[2] requests. EWS has 42 unique compositions that can be (re)assembled at runtime to cope with changes in the incoming workload pattern. When changing from one composition to another, the EWS calculates a delta between the two compositions, then performs a series of individual component hot-swaps to reach the new composition; each such hot-swap is guaranteed not to miss or lose any client requests. Amongst its compositions, there are different variants of stream processors that use or don't use compression or caching in their logic, and different compression and cache replacement algorithms which combine to form specific EWS compositions. Furthermore, EWS provides a RESTful API that enables external systems to inspect and change the current web server composition, add or remove components at runtime and extract performance metrics from the executing server. It also provides a baseline online learning algorithm for comparison; researchers can use this exemplar to compare accuracy and convergence time amongst different online learning strategies. This section describes the EWS internal architecture, its RESTful API, and a Python module named PyEWS developed to facilitate the interaction with EWS in Python.

3.1 EWS Architecture

EWS was developed using a component-based model defined by the Dana programming language[3] that supports the change (i.e., replacement, addition and removal) of components at runtime. The Dana component-based model is somewhat similar to OSGi [11] and Fractal [5], but supports provably sound component hot-swaps as a result of how the language is designed [21]. Moreover, every part of a Dana system is inherently a hot-swappable component, from TCP sockets to graphical user interface widgets, providing ubiquity of the component-based paradigm throughout a system and likewise offering uniform reasoning for online learning about all parts of a system. In general, component-based software is composed of a set of small, reusable components executing in a single process. Each component implements an interface, which specify a list of function prototypes (with parameter- and return-types). Many different components are able to implement the same interface, providing a set of component variants (such as different sorting algorithms, or cache replacement algorithms). These variants are then used as basis to provide software adaptation tactics at runtime by swapping one component to one of its variants.

Fig. 1 shows all web server architectural compositions that are included in the base EWS exemplar. The dotted outer boxes represents the main interfaces of the EWS, and inside those boxes the solid-lined entities represent the components that implement the interface. Most interfaces have many a range of implementation

[2] RFC: https://datatracker.ietf.org/doc/html/rfc1945
[3] Dana: http://www.projectdana.com

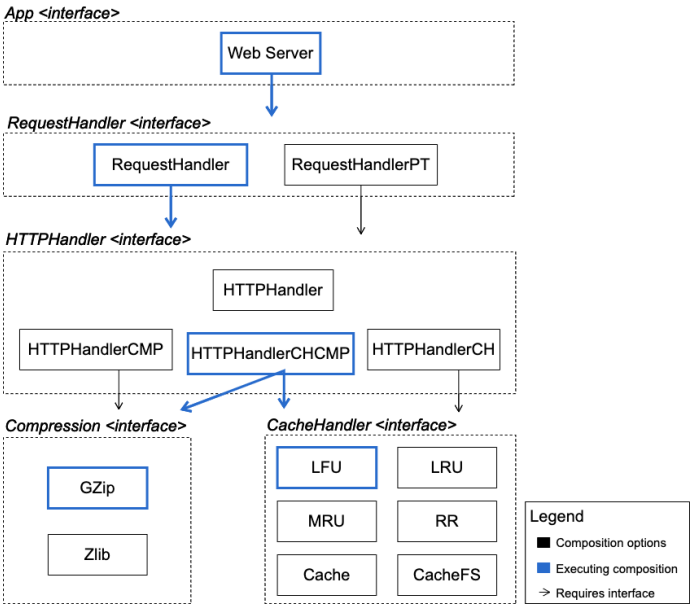

Figure 1: EWS architecture. Dotted-lined boxes represents interfaces. Solid-line boxes represents components. Arrows represent dependencies. The set of blue boxes and arrows represent a single composition.

variants, meaning that a change in the web server architecture can be made by replacing one component to its variant. For instance, when the *Compression* compression interface is part of the web server composition, either component *GZip* or *Zlib* is selected for use in the system. When Gzip is selected, a possible change in the system architecture consists of swapping the Gzip component to the Zlib alternative.

The web server has four main interfaces: **RequestHandler**, **HTTPHandler**, **Compression** and **CacheHandlder**. These interfaces make up the core functionalities of EWS. Component variants for these interfaces define the number of unique compositions EWS presents. We next describe each one of these interfaces:

RequestHandler. describes how incoming requests are handled. The components that implement this interface define different concurrency models to handle incoming requests. The component *RequestHandler* creates one thread for every new request, whereas the *RequestHandlerPT* keeps a pool of threads and assigns incoming requests to a queue on one thread in the pool.

HTTPHandler: is a stream processor which defines the main HTTP serving functionality. Its components implement the basic processing procedure of a HTTP GET request. The component *HTTPHandler* is the default implementation of the interface; when handling a client request, it searches for the requested resource in the file system, loads the resource to memory and sends it to the client. The rest of the components implement the same functionality with additional steps. *HTTPHandlerCH* is an alternative that checks if a requested resource is in a cache, responding with the cached version if so; if not, it locates the resource and adds it to the cache (potentially evicting other cached items when it does so) then responds to the client. *HTTPHandlerCMP*, in turn, compresses all the data before sending the response to the client. Finally, *HTTPHandlerCHCMP* is a combination of both cache and compression. Therefore, before sending the response to the client, it compresses the data, and after sending the response to the client, it caches the compressed response.

Compression: defines the functions used by *HTTPHandlerCMP* and *HTTPHandlerCHCMP*. The two different components that implement this interface provide different compression algorithms. *GZip* implements the *gzip* compression algorithm, whereas *Zlib* implements *deflate*.

CacheHandler: defines the functions used by *HTTPHandlerCH* and *HTTPHandlerCHCMP* that require a cache. The components that implement this interface provide caching with different cache replacement algorithms which evict one or more cached items when the cache is full and a new item is being added. Amongst the different component variants, there are: Least-frequently-used (LFU), Least-recently-used (LRU), Most-recently-used (MRU) and Random Replacement (RR).

Fig. 1 shows every possible EWS composition. A valid composition consists of selecting one component out of the possible variants for each of the interfaces that are part of the composition. For instance, the WebServer component is the root component of EWS and it requires the **RequestHandler** interface; that means that a component with that interface needs to be selected; once it is, both available components require the **HTTPHandler** interface. At that interface, there are four unique options. Once the *HTTPHandler* component is selected, the composition is finished since it has no further dependency. However, in case the other components at that interface are selected, then a component on the required interface must be selected. This results in a total of 42 unique compositions.

Architectural Description. EWS identifies each of the unique compositions that are available using a specific *architectural description string*. These description strings are used by the EWS when reporting the set of available compositions, the currently-in-use one, and also in allowing the user (or self-adaptive system controller) to select a composition to adapt to. The description string encodes the complete graph of a composition, and has two parts divided by the symbol "|"; the first part is a list of components that are part of the composition, the second part defines how the components are connected. While it is not necessary for a user to understand the description string format, being able to reason about it allows a learning algorithm to understand which components are in use and how each one may be contributing to overall utility.

Fig. 1 highlights a specific example EWS composition. The composition description corresponding to this is:

```
WebServer.o, RequestHandler.o, HTTPHandlerCHCMP.o,
GZip.o, LFU.o|0:RequestHandler:1, 1:HTTPHandler:2,
2:Compression:3, 2:CacheHandler:4
```

The list of all components in the composition is: *Web Server*, *RequestHandler*, *HTTPHandlerCHCMP*, *GZip*, *LFU*. The description also denotes how the components are connected: component 0 is connected to component 1 through interface **RequestHandler**.

1. HTTP **POST**: *meta/set_main*
   parameter: *{"comp" : "<path to comp>"}*
2. HTTP **GET**: meta/get_config
   response: *{"config" : "<architecture description>"}*
3. HTTP **GET**: meta/get_all_config
   response: *{"configs" : ["<arch desc1>", "<arch desc2>"]}*
4. HTTP **POST**: meta/set_config
   parameter: *{"config" : "<architecture description>"}*
5. HTTP **POST**: meta/add_comp
   parameter: *{"comps" : ["<comp path>", "<comp path>"]}*
6. HTTP **POST**: meta/remove_comp
   parameter: *{"comps" : ["<comp path>", "<comp path>"]}*
7. HTTP **POST**: meta/add_proxy
   parameter: *{"config" : "<architecture description>"}*
8. HTTP **POST**: meta/remove_proxy
   parameter: *{"config" : "<architecture description>"}*
9. HTTP **GET**: meta/get_perception
   response: *{"config" : "<architecture description>"}*

Figure 2: EWS REST API.

The numbers used in this part of the description are indices into the list of components in the first part of the description (the first component starts with 0, the second is 1 and so on). Hence, "0:RequestHandler:1" means: component at position 0, which is *WebServer.o*, is connected to the component at the position 1, which is the component *RequestHandler.o*.

Throughout our examples here, we note that we have isolated only the components that specifically relate to the EWS; in reality composition strings are much longer as they include Dana standard library components such as `TCP`, `FileSystem`, etc.

3.2 EWS REST API

EWS provides a RESTFul API to allow the development of external learning algorithms. The API provides a set of functions that allow external services to change EWS composition at runtime, fetch real-time performance metrics, and add / remove components from the web server architecture at runtime (where a newly added component will cause EWS to derive a set of additional compositions that are now available which involve that component). In this section we describe each of these functions and provide some context on how they are used.

Fig. 2 shows the complete list of functions on the EWS REST API. The first is "set_main" (Fig. 2 (1)). This function is provided with the file path of the root component of a system, and dynamically **discovers** all possible architectural compositions starting from that component. This is done by examining the interfaces that the root component depends on, finding all available components which implement those interfaces, and then examining the interfaces that *those* components depend on, and so on. The "set_main" function should only be called once when initiating EWS and expects one parameter, the EWS `WebServer` root component.

Once EWS is running, all of the other REST API functions can be used, and operate as follows.

The function "get_all_configs"(Fig. 2 (3)) returns a list of all available EWS compositions as architecture description strings. This function is often used by a machine learning algorithm to determine the list of adaptation tactics that are available.

The "*get_config*" function (Fig. 2 (2)) returns the architectural description string of the currently executing EWS composition.

The "set_config" function (Fig. 2 (4)) changes the currently operating composition to a different one. This function receives as parameter an EWS architectural description string and changes its composition at runtime (with no down time from the server). This function is often used by learning algorithms to explore different architectural compositions and learn their utility in each set of deployment environment conditions that is detected.

Besides examining available compositions and changing the currently active one, EWS also allows the dynamic addition of new components (and newly derived compositions which involve those components). Additions are made by using the "add_comp" function (Fig. 2 (5)). A file path to a component is provided as a parameter (assuming that component has previously been uploaded to the server via e.g. SFTP), and the system examines which interface this new component provides, and which interfaces it depends on, and derives a new set of additional compositions which are then available through "get_all_configs". If, for example, a development team (or automated code improvement system) determines that there may exist a better cache replacing algorithm that fits the workload pattern, a new cache component can be dynamically added at runtime. A new set of compositions is then generated and their efficacy can be learned by the ongoing online learning process.

The function "remove_comp" (Fig. 2 (6)) removes one or more EWS components, resulting in some compositions being removed in which those components were involved. For instance, if the Gzip compression component is removed, all EWS compositions that use Gzip are eliminated from the available set of possible compositions. This can be a useful search space reduction strategy, which in turn can speed up learning convergence in newly-detected operating environment conditions, in cases where a learning algorithm has identified a set of compositions that have poor performance across a wide range of different operating environments.

In addition to being able to view the set of available compositions, select a composition, and add/remove components/compositions, an online learning algorithm must also be able to ascertain the current reward (or utility) of the chosen composition. EWS provides three functions through its REST API to support reward monitoring: the ability to inject and remove monitoring probes at selected points in a composition, and the ability to collect monitoring data acquired by those probes.

Monitoring probes can be inserted into EWS using the function "add_proxy" (Fig. 2 (7)). This function is provided with a file path to a component which will act as the monitoring probe, and path expression describing where that probe is to be inserted into any composition (e.g., in front of which interface).

In case there is already a monitoring probe present, users can remove it by using the function "remove_proxy" (Fig. 2 (8)). Once the probe is removed, the user can insert a new probe to extract new information from EWS. These two functions allow either a human user or a machine learning algorithm to experiment with measuring the self-adaptive from different locations to aid in understanding how different sub-elements of its architecture behave.

Finally, the function "get_perception" (Fig. 2 (9)) returns all monitoring data collected by monitoring probes, then clears the log of monitoring data in EWS. This function is usually called periodically

```python
from pyews.server_interface import ewsRESTInterface as eRI

# EWS startup
eRI.initialize_server(definitions["main_component"],
    definitions["proxy_JSON"])

# EWS available composition list,
# returns a list of Component objects
configurations = eRI.get_all_configs()

# EWS monitoring data - returns a Perception object
perception = eRI.get_perception()
```

Figure 3: EWS Python Interface.

by a learning algorithm in a fixed, predefined time interval to observe EWS performance. The monitoring probe we make available in the base EWS artifact collects the average response time to a request and also notes each request's MIME type. Response time is useful to determine the composition that has the highest utility, whilst the MIME type is useful to help identify patterns in the incoming workload and classify distinct operating environments – where each newly-detected distinct environment may trigger a new round of online learning for that environment, or a previously-detected environment may allow a learning algorithm to recall its prior learning state or best-choice for that environment.

Together, these functions facilitate the exploration of online learning algorithms in situations where both the performance of each architecture composition, and the set of possible operating environment characteristics, are unknown prior to deployment. They also allow the creation and exploration of new learning algorithms that can add or remove compositions that further optimise EWS or reduce its composition search space. Because the API is presented as REST server, this allows the exploration of learning algorithms written in any programming language which reduces the learning curve to use our exemplar.

3.3 EWS Python Interface

We also make available a Python Interface to facilitate interaction with EWS for Python developers. Given the current popularity of Python among machine learning practitioners, this is intended to broaden the usability of our exemplar.

The provided Python Interface acts as a client to the RESTful API provided by EWS and provides a set of native Python functions equivalent to those in the REST API, thereby avoiding the need for the user to manually parse JSON-formatted data when interacting with the EWS. We also provide an implementation of the ϵ-greedy learning algorithm, as an example to demonstrate how the EWS Python API can be used to implement online learning strategies.

Besides the methods defined by the EWS RESTful API in the previous section, the Python interface also provides the "$initialize\_server()$" and "$change\_configuration()$" methods, which are abstractions over "*meta/set_main*" and "*meta/add_proxy*" functions required to initialise the server, and the "meta/set_config" function to change EWS composition without using the architectural description and using *Component* Python objects instead.

Fig. 3 shows an example of how to initialise EWS in Python, get a list of all available compositions, and get monitoring data from the executing EWS. To use the Python interface, the user needs only to import the appropriate modules (as shown in Fig. 3), then initialise the EWS. In the example here we get a list of all available compositions, then get monitoring data from the executing EWS.

4 EXEMPLAR IN ACTION

This section describes the necessary steps to download, setup, execute, and interact with EWS using a comment-program that enables users to interact with the EWS API.

Step 1.: Download and execute EWS. The quickest way to get EWS running is to execute its Docker image. We make available a docker image on DockerHub that is ready to run. To execute the container, the user is required to have Docker installed and running. After ensuring that Docker itself is working, type the command:

```
docker run --name=ews -p 2011-2012:2011-2012 -d
    robertovrf/ews:1.0
```

Step 2.: Get access to the container terminal. To interact with EWS, we need to get access to the container terminal so that we can type commands and view output. We do this using the command:

```
docker exec -it ews bash
```

Step 3.: Interacting with EWS. There are two ways to interact with EWS. One way is to write your code using our Python interface or EWS RESTful API, and the other is through a command prompt. As we have already described the APIs, here we describe how to use the command prompt. First, the user must start the InteractiveEmergentSys.o component, which provides access to the command prompt. In the container terminal, type the command:

```
dana -sp ../repository InteractiveEmergentSys.o
```

Once the *InteractiveEmergentSys.o* component is executing, the user can interact with EWS using the terminal. To start, the user can type "help" to get a list of the available commands. For a more detailed description of how to execute EWS, please refer to EWS GitHub repository[4].

5 PUBLISHED RESULTS AND CHALLENGES

The EWS is intended to allow researchers to experiment with online learning for open-ended architectural self-adaptation, where new components can be dynamically added and removed. A wide range of challenges are yet to be explored in this space; in this section we briefly summarise existing results that have used the EWS, then describe major open challenges.

Existing results using the EWS have explored different multi-armed bandit reinforcement learning algorithms [26]. This type of learning algorithm consider a set of actions (i.e., EWS compositions) that when selected return a reward (or cost – i.e., response time), where the reward that is observed obeys a probability distribution function which captures natural variance in reward over time. The goal of this learning strategy is to learn which action to select in order to minimise cost (i.e., response time).

One of the first results published using EWS was at OSDI'16 [22]. This work explored the general idea of autonomously learning how to compose systems at runtime and finding optimal compositions as the system executes – and while its operating conditions experience

[4] https://github.com/robertovrf/emergent_web_server

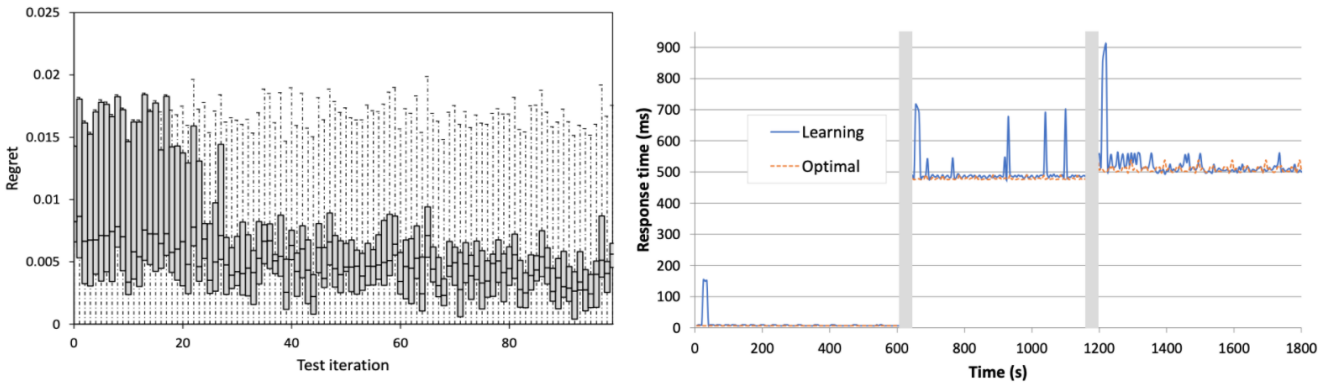

Figure 4: Thompson sampling: (a) shows online learning reducing regret as it explores different web server composition. UCB1: (b) shows response time decreasing as the algorithm converges towards the optimal solution for different environments.

regular change that may not have been predictable at design-time. In this particular study we used the exemplar to explore an online machine learning technique based around Thompson sampling [6], combined with Bayesian regression to locate the optimal EWS composition for a variety of different workload patterns. The use of continuous regression allowed the learning approach to understand the marginal contribution of each individual component towards the overall reward of a chosen composition (without directly measuring each component), and to then make estimates on the likely reward of untried compositions based on which components those compositions used. The results showed that this approach was able to very quickly narrow the search space to locate the ideal composition, needing to sample as few as 6 compositions or at most 20 compositions in order to have a high-confidence model of which of the 42 available compositions was best suited to the current deployment conditions. An example result is illustrated in Fig. 4 (a), showing on the x-axis the selection of a composition at a given time and its corresponding reward on the y-axis (expressed as 'regret' from a known ground truth); this example shows a particularly challenging deployment environment which required around 20 compositions to be tried before a high-confidence choice is made.

A second study, published at SASO'19 [23], shows the application of the UCB1 online learning algorithm [3] to locate the EWS composition that has the lowest response time, with example results shown in Fig. 4 (b) (y-axis). Each of the vertical bars separating the graphs is an entire experiment where EWS was subjected to different workload patterns; while the graph shows that the algorithm converges towards the optimal composition, we also observe that sub-optimal actions are sometimes chosen repeatedly to ensure they continue sub-optimal. This demonstrates some of the difficulty in properly tuning online learning algorithms for unknown environments and reward ranges; this remains an open challenge.

Our existing results demonstrate that it is possible to learn an ideal action, at runtime, with no prior knowledge about the possible EWS compositions nor the operating environment on which EWS is executing. Despite these results, a range of further research is needed to understand online learning in open-ended architectural self-adaptation. Our studies to date have examined two different learning approaches, which have very different characteristics; further study is needed to understand which online learning approaches tend to work best in these scenarios, particularly where the search space of permutations grows much larger. While it is possible to infer information about individual component utility across possible compositions, for example, it may also be viable to transfer knowledge between deployment environment conditions that have at least some shared features. Second, using online learning in non-stationary deployment environments has some very difficult challenges; without prior knowledge, an online learning system is likely to need to simultaneously learn both how to classify the deployment environment in a way that correlates usefully to different reward levels, and how to learn which compositions are best suited to each such environment. Doing this under environment conditions that may change at any moment has received relatively little study. Finally, there is very little research which studies *changes* to the available set of tactics post-deployment, such as the addition of a new component, or the ability to remove components that have shown little promise in utility gain in order to narrow a search space.

6 CONCLUSION

In this paper we have described the Emergent Web Server (EWS) artefact. This artefact is an exemplar designed for researchers to explore problems related to online learning for compositional self-adaptive systems. It has 42 unique architectural compositions and, at runtime, EWS can change from one composition to another seamlessly, with no downtime. EWS provides both a RESTful API and a Python interface that enables users to interact with it. Through the API, users can get monitoring data from the executing EWS, change its composition at runtime, get a list of all available compositions, and add or remove components. EWS is an easy to use exemplar that can be used to study a set of open challenges; we hope the community may find it a useful tool in future research.

REFERENCES

[1] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. 2008. Adaptive Action Selection in Autonomic Software Using Reinforcement Learning. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. 175–181. https://doi.org/10.1109/ICAS.2008.35 ISSN: 2168-1872.

[2] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. 2017. A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, Madrid, Spain, 64–73. https://doi.org/10.1109/CCGRID.2017.15

[3] Peter Auer. 2002. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research* 3, Nov (2002), 397–422.

[4] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 2015. Hogna: A Platform for Self-Adaptive Applications in Cloud Environments. In *Proceedings of the 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '15)*. IEEE Computer Society, USA, 83–87. https://doi.org/10.1109/SEAMS.2015.26

[5] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani. 2009. Component-based architecture: the Fractal initiative. *annals of telecommunications - annales des télécommunications* 64, 1 (2009), 1–4. https://doi.org/10.1007/s12243-009-0086-1

[6] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of thompson sampling. *Advances in neural information processing systems* 24 (2011), 2249–2257.

[7] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2009. Evaluating the effectiveness of the Rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, Vancouver, BC, 132–141. https://doi.org/10.1109/SEAMS.2009.5069082

[8] Erik M. Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. 2019. Planning as Optimization: Dynamically Discovering Optimal Configurations for Runtime Situations. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 1–10. https://doi.org/10.1109/SASO.2019.00010 ISSN: 1949-3681.

[9] Ilias Gerostathopoulos and Evangelos Pournaras. 2019. TRAPPed in Traffic? A Self-Adaptive Framework for Decentralized Traffic Optimization. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Montreal, QC, Canada, 32–38. https://doi.org/10.1109/SEAMS.2019.00014

[10] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review. *ACM Transactions on Autonomous and Adaptive Systems* 15, 3 (Sept. 2021), 1–37. https://doi.org/10.1145/3469440

[11] R.S. Hall and H. Cervantes. 2004. An OSGi implementation and experience report. In *First IEEE Consumer Communications and Networking Conference, 2004. CCNC 2004.* 394–399. https://doi.org/10.1109/CCNC.2004.1286894

[12] Steven C.H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2021. Online learning: A comprehensive survey. *Neurocomputing* 459 (2021), 249–289. https://doi.org/10.1016/j.neucom.2021.04.112

[13] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. 2017. DeltaIoT: a self-adaptive internet of things exemplar. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '17)*. IEEE Press, Buenos Aires, Argentina, 76–82. https://doi.org/10.1109/SEAMS.2017.21

[14] Pooyan Jamshidi, Amir Sharifloo, Claus Pahl, Hamid Arabnejad, Andreas Metzger, and Giovani Estrada. 2016. Fuzzy Self-Learning Controllers for Elasticity Management in Dynamic Cloud Architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. IEEE, Venice, Italy, 70–79. https://doi.org/10.1109/QoSA.2016.13

[15] Cody Kinneer, David Garlan, and Claire Le Goues. 2021. Information Reuse and Stochastic Search: Managing Uncertainty in Self-* Systems. *ACM Transactions on Autonomous and Adaptive Systems* 15, 1 (Feb. 2021), 1–36. https://doi.org/10.1145/3440119

[16] Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. 2015. An Architecture Framework for Experimentations with Self-Adaptive Cyber-physical Systems. In *Proceedings of the 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '15)*. IEEE Computer Society, USA, 93–96. https://doi.org/10.1109/SEAMS.2015.28

[17] Peter R. Lewis, Lukas Esterle, Arjun Chandra, Bernhard Rinner, Jim Torresen, and Xin Yao. 2015. Static, Dynamic, and Adaptive Heterogeneity in Distributed Smart Camera Networks. *ACM Transactions on Autonomous and Adaptive Systems* 10, 2 (June 2015), 8:1–8:30. https://doi.org/10.1145/2764460

[18] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. 2019. Dragonfly: a Tool for Simulating Self-Adaptive Drone Behaviours. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Montreal, QC, Canada, 107–113. https://doi.org/10.1109/SEAMS.2019.00022

[19] Gabriel Moreno, Cody Kinneer, Ashutosh Pandey, and David Garlan. 2019. DART-Sim: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Smart Cyber-Physical Systems. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Montreal, QC, Canada, 181–187. https://doi.org/10.1109/SEAMS.2019.00031

[20] 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 the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '18)*. Association for Computing Machinery, Gothenburg, Sweden, 137–143. https://doi.org/10.1145/3194133.3194163

[21] Barry Porter and Roberto Rodrigues Filho. 2021. A Programming Language for Sound Self-Adaptive Systems. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 145–150. https://doi.org/10.1109/ACSOS52086.2021.00036

[22] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. REX: a development platform and online learning approach for runtime emergent software systems. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 333–348.

[23] Barry Porter and Roberto Rodrigues Filho. 2019. Distributed Emergent Software: Assembling, Perceiving and Learning Systems at Scale. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 127–136. https://doi.org/10.1109/SASO.2019.00024 ISSN: 1949-3681.

[24] Michiel Provoost and Danny Weyns. 2019. DingNet: A Self-Adaptive Internet-of-Things Exemplar. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Montreal, QC, Canada, 195–201. https://doi.org/10.1109/SEAMS.2019.00033

[25] Sanny Schmid, Ilias Gerostathopoulos, Christian Prehofer, and Tomas Bures. 2017. Self-adaptation based on big data analytics: a model problem and tool. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '17)*. IEEE Press, Buenos Aires, Argentina, 102–108. https://doi.org/10.1109/SEAMS.2017.20

[26] Steven L. Scott. 2010. A modern Bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry* 26, 6 (2010), 639–658. https://doi.org/10.1002/asmb.874 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/asmb.874

[27] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *2006 IEEE International Conference on Autonomic Computing*. 65–73. https://doi.org/10.1109/ICAC.2006.1662383

[28] Karthik Vaidhyanathan. 2021. *Data-Driven Self-Adaptive Architecting Using Machine Learning*. Ph. D. Dissertation. GSSI Gran Sasso Science Institute.

[29] Thomas Vogel. 2018. mRUBiS: an exemplar for model-based architectural self-healing and self-optimization. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '18)*. Association for Computing Machinery, Gothenburg, Sweden, 101–107. https://doi.org/10.1145/3194133.3194161

[30] Danny Weyns and Radu Calinescu. 2015. Tele Assistance: A Self-Adaptive Service-Based System Examplar. In *Proc. of SEAMS '15*. IEEE. http://homepage.lnu.se/staff/daweaa/papers/2015SEAMS.pdf