Exploring Multi-armed Bandits Learning Strategies in the Emergent Web Server

Manoella Rockembach
Federal University of Santa Catarina - UFSC
manu.rockemba@gmail.com

Ítalo Manzine Amaral Duarte Garofalo
Federal University of Santa Catarina - UFSC
italomanzine@gmail.com

João Pedro Tavares Santos
Federal University of Santa Catarina - UFSC
j-pts@hotmail.com

Alison R. Panisson
Federal University of Santa Catarina - UFSC
alison.panisson@ufsc.br

Roberto Rodrigues-Filho
Federal University of Santa Catarina - UFSC
roberto.filho@ufsc.br

Abstract

Self-adaptive systems are designed to modify their architecture or behavior to uphold high-level objectives despite changes in their operating environments. A critical aspect of developing such systems involves creating strategies to handle unexpected events in the operating environments. While this remains an active area of research within the autonomic computing and self-adaptive systems community, one commonly adopted approach is leveraging machine learning techniques, particularly reinforcement learning, to address unforeseen challenges. In this paper, we conduct experiments using the Emergent Web Server exemplar, a publicly available self-adaptive web server, to investigate various monitoring metrics and implement a multi-armed bandit reinforcement learning approach. This approach enables the system to identify the optimal web server configuration for maximizing performance under varying workload patterns and operating conditions, enabling the system to react to unexpected events that rises from the operating environment with minimum human interference.

Keywords

Self-adaptive Systems, Emergent Web Server, Autonomic Computing, Multi-armed Bandits

1 Introduction

As the complexity of developing and managing contemporary systems continues to grow, the autonomic computing community emphasizes the importance of transferring more responsibility for system management to the systems themselves [1–3]. Autonomic computing and self-adaptive systems are paradigms defined to create systems capable of self-management, i.e., dynamically adapting their architecture or behavior to address changes as they occur.

A critical aspect of building such systems lies in designing strategies to handle unexpected events. Traditionally, autonomic computing systems relied on predefined, static rules to guide adaptation [4, 5]. These rule-based approaches require experts to anticipate all possible scenarios a system may encounter. However, in modern, dynamic operating environments [6], characterized by continual change, such approaches are limited. Eventually, these systems face scenarios where no rule exists to guide adaptation effectively, leading to suboptimal performance or failure. Recent advancements in machine learning have motivated researchers to explore techniques that minimize human intervention in defining adaptation rules.

A notable and early example of machine learning applied to self-adaptive systems is FUSION [7], where a machine learning-based approach was used to derive adaptation rules with minimal human supervision. However, these approaches require the system to be pre-trained on relevant scenarios. If the model is not exposed to all scenarios, it may encounter situations in its operating environment where it cannot respond effectively. Thus, this approach is limited in providing a safeguard mechanism to handle uncertainty.

We advocate that reinforcement learning techniques are better suited to address the challenges of adapting to unexpected operating environments. Reinforcement learning offers a framework for systems to learn and adapt dynamically, even when faced with unforeseen scenarios. This is due to its capability of improving and expanding its model as it interacts with the system in any given operating environment, enabling the agents to derive adaptation rules for all encountered scenarios, even unforeseen ones. Nevertheless, as noted by Gheibi et al. [8], no single machine learning approach has yet demonstrated a definitive advantage over others in this domain.

In this paper, we investigate a publicly available self-adaptive system provided by Rodrigues-Filho et al. [9]. Specifically, we explore the use of two multi-armed bandit algorithms, UCB1 [10] and Epsilon-Greedy [11], to address the exploration-exploitation trade-off inherent in reinforcement learning problems. Our results show that the UCB1 algorithm can identify the optimal web server composition at runtime without relying on predefined domain-specific knowledge or static adaptation rules, aligning with findings from other significant studies [12, 13].

The remainder of this paper is organized as follows. Section 2 reviews relevant related work. Section 3 describes the emergent web server exemplar and the foundations of our online learning approach: monitoring, adaptation mechanisms, and online learning. Section 4 presents an evaluation of the proposed online learning strategies. Finally, Section 5 concludes with final remarks.

2 Related Work

2.1 Learning Strategies for Self-adaptive Systems

Learning strategies are essential for equipping self-adaptive systems with the capability to handle dynamic and unpredictable environments. Unlike static adaptation models, learning-based approaches enable these systems to evolve their adaptation logic during runtime, improving their ability to meet user-defined goals, even under unforeseen conditions [7, 13].

Traditional self-adaptive systems rely heavily on analytical models defined at design time [1, 14–17]. These models estimate the system's response to various environmental conditions and guide adaptation decisions. However, they often fail when faced with unanticipated changes, as they are constrained by static assumptions [7]. Learning strategies address this limitation by allowing systems to refine their decision-making processes continuously, based on observed behavior and runtime data, as argued by Porter et al. [13].

By incorporating learning into the adaptation cycle, self-adaptive systems can:

(1) **Detect Patterns**: Identify emergent behaviors or anomalies that were not accounted for during system design.
(2) **Adapt Dynamically**: Adjust their behavior to align with current operational conditions without human intervention.
(3) **Optimize Performance**: Improve resource usage and response times by exploring and fine-tuning configuration options.

In the following sections, we present a broad overview of learning-based strategies for self-adaptive systems discussed in the literature.

2.1.1 ***FUSION: A Learning-Centric Framework.*** The FeatUre-oriented Self-adaptatION (FUSION) framework [7] exemplifies a learning-based strategy for self-adaptive systems. FUSION's key innovation lies in its feature-oriented approach, where each feature represents a distinct system capability. The framework uses a continuous learning cycle to understand the impact of enabling or disabling specific features on system goals, such as performance, security, or reliability. The FUSION learning cycle involves: *i)* **observation**: collecting metrics during runtime to identify discrepancies between expected and actual outcomes; and, *ii)* **induction**: learning new relationships between features and goals, refining the system's adaptation logic to address emergent conditions.

2.1.2 ***Complementing Learning with Control Theory.*** Control Theory [15] provides a foundational structure for maintaining system stability via feedback loops. When combined with learning strategies, these feedback mechanisms become more robust, as the system can refine control parameters based on runtime observations. For instance, FUSION can augment a control-theoretic system by: *i)* learning optimal parameter settings for controllers, such as thresholds for workload distribution; and *ii)* identifying new adaptation strategies when feedback loops alone are insufficient, e.g., during novel traffic patterns or hardware failures.

This hybrid approach allows self-adaptive systems to not only stabilize their behavior but also evolve over time to handle increasingly complex scenarios.

2.1.3 ***Reinforcement Learning.*** This branch of machine learning focuses on making decisions to maximize cumulative rewards in a given situation. Unlike supervised learning, which relies on a training dataset with predefined answers, reinforcement learning involves learning through experience. In this method, the system finds its best configuration in an ever-changing, complex environment by calculating a numerical value of reward that represents its overall performance in the environment it is in.

This paradigm can be explained/illustrated through the multi-armed bandit problem, a very versatile concept that is well known for its applications in recommending systems and ad selection based on user activity on the web [12, 18]. A multi-armed bandit is composed by a set of arms that represent the systems available actions, each time an arm pulled/chosen results in a different value of reward, which depends (only) on the selected arm. The multi-armed bandits' objective is to maximize the total reward obtained, and it does so by balancing the exploitation of arms already known to perform well with the exploration of under-tested arms to ensure that it does not fail to find the best arm.

The most prominent strategies behind the decision-making of a multi-armed bandit are the Epsilon-greedy [11] and UCB [10] algorithms. The epsilon-greedy algorithm is the simpler of the two mentioned, and it works as follows: each time the algorithm needs to decide whether to explore or exploit, it has an ϵ probability of choosing to explore a random arm and a $1 - \epsilon$ probability of choosing to exploit the arm with the highest cumulative reward up until that point. The UCB algorithm, on the other hand, is based on the principle of optimism in the face of uncertainty. It selects the arm with the highest upper confidence bound, balancing the estimated reward and the uncertainty of the estimate.

2.2 Monitoring Systems for Self-adaptive Systems

Effective monitoring is a pillar of self-adaptive systems, as it provides the essential data required for evaluating system performance and triggering adaptation mechanisms. In this context, the monitoring system leverages the RESTful API of the EWS (Emergent Web Server) to collect runtime data efficiently.

2.2.1 ***Monitoring Architecture and Workflow.*** In Self-adaptive Systems, a module is responsible for executing periodic calls to the EWS API. These calls gather key performance metrics, such as **response time, resource utilization, and throughput**. By ensuring that these metrics are collected regularly, the monitoring system provides an up-to-date view of the system's operational state, enabling timely identification of potential goal violations or performance bottlenecks. The main components are:

- **Data Retrieval**: Executes structured API calls to fetch specific performance metrics.
- **Data Organization**: Formats the retrieved data into structured representations, making it ready for further analysis.

- **Error Handling**: Manages API call failures or inconsistencies in the data to ensure the robustness of the monitoring process.

A critical design goal of the monitoring system is to ensure that data collection imposes minimal overhead on the running system, i.e., achieving continuous and lightweight monitoring. The periodic API calls are designed to be lightweight, avoiding significant interruptions to system processes. Additionally, the modular nature of the python module allows for easy customization, enabling the monitoring system to scale with the complexity and needs of the self-adaptive system.

2.2.2 ***Monitoring Role in Decision-Making.*** The data collected by the monitoring system serves as the foundation for adaptation decisions. Metrics are continuously evaluated against predefined objectives, such as maintaining response times below a certain threshold or optimizing resource allocation. When discrepancies are detected, the monitoring system flags potential violations, prompting the adaptation logic to plan and execute corrective actions.

By combining the capabilities of the EWS API and requests, the monitoring system not only ensures a reliable flow of runtime data but also supports the dynamic nature of self-adaptive systems. This integration allows for real-time evaluation of system behavior and facilitates informed, data-driven adaptations to meet evolving demands and conditions

3 Online Learning for Emergent Web Server

The online learning approach for an emergent web server consists of several components for monitoring, learning, and adapting the system. The following sections discuss each of these components.

3.1 Adaptation

The adaptation component is the core of the system's ability to autonomously adjust its behavior. This component leverages the algorithm to automatically modify system parameters, reconfiguring the server's composition based on the monitored metrics. The adaptation component is composed by an *adaptation mechanism* and an *adaptation logic*.

3.1.1 ***Adaptation Mechanism.*** The adaptation mechanism is responsible for triggering the reconfiguration of the EWS compositions. It employs the algorithm to identify optimal adaptation strategies and operates as follows:

(1) **Identifying the current composition**: The system retrieves the current composition of the EWS and evaluates the associated performance metrics.
(2) **Proposing a new composition**: Based on the analysis, the algorithm suggests new values for the hyperparameters of the Multi-Armed Bandit (MAB) algorithms that govern the selection of compositions.
(3) **Switching configurations**: If a new composition is determined to be more efficient, the EWS transitions to the suggested configuration while ensuring uninterrupted processing of requests.

3.1.2 ***Adaptation Logic.*** The adaptation logic is driven by the Dynamic Bayesian Optimisation for Multi-Arm Bandits (DBO-MAB) algorithm [19], which employs an incremental optimization approach:

(1) **Data collection**: At each monitoring cycle, the metrics are analyzed by the DBO-MAB.
(2) **Evaluating the current composition**: The algorithm assesses the performance of the current composition based on the collected metrics and determines whether adjustments to the MAB algorithm parameters are necessary.
(3) **Exploration vs. Exploitation**: The algorithm balances exploration (testing new compositions) with exploitation (refining the current composition) using Bayesian Optimization. This balance ensures optimal performance by dynamically adjusting the hyperparameters over time.
(4) **Hyperparameter updates**: When changes are needed, the algorithm proposes new hyperparameter values for composition selection. The EWS is then reconfigured to reflect these updates.

3.1.3 ***Benefits of the Approach.*** This approach enables the system to efficiently adapt to continuously evolving workload patterns while maintaining optimal performance in real time. By dynamically reconfiguring itself based on monitored metrics, the system ensures that it remains responsive to changing demands and achieves sustained performance improvements.

3.2 Monitoring

Effective monitoring is crucial for the Emergent Web Server (EWS) to adapt dynamically to changing workloads and maintain optimal performance. The monitoring system collects runtime data by leveraging the EWS's RESTful API and Docker container statistics, enabling informed decision-making for adaptation strategies.

3.2.1 ***Monitoring Architecture and Workflow.*** The monitoring component is implemented as a Python module that periodically gathers key performance metrics such as response time, CPU usage, memory usage, and throughput. This data collection is achieved through structured API calls and Docker statistics. The data collection process is as follows:

(1) **Response Time Monitoring:** The module uses the EWS API to retrieve the average response time for client requests.
(2) **Resource Utilization Monitoring:** CPU and memory usage are obtained from the Docker container running the EWS using the Docker SDK for Python.
(3) **Throughput Measurement:** Network I/O statistics are collected to calculate the throughput of the server.

3.2.2 ***Implementation Details.*** The monitoring system is designed to be lightweight and efficient, ensuring minimal overhead on the EWS. Key functionalities of the monitoring process include:

- **Accessing Docker Statistics:** Connects to the EWS Docker container to retrieve real-time statistics on resource utilization.
- **Calculating CPU and Memory Usage:** Processes the collected data to compute the current CPU usage percentage and memory consumption of the server.

- **Retrieving Response Time:** Accesses the EWS perception data to obtain the average response time for handling client requests.

3.2.3 ***Integration with Adaptation Mechanism.*** The collected metrics are fed into the adaptation component to make informed decisions about reconfiguring the EWS. By continuously monitoring the system, the adaptation logic can adjust configurations in real-time to optimize performance.

3.2.4 ***Role in Learning Strategies.*** The monitoring component provides essential data for the learning algorithms, such as Epsilon-Greedy and UCB1, which are used to select the most efficient configurations. Accurate and timely metrics enable these algorithms to evaluate the performance of different configurations and make decisions that balance exploration and exploitation.

3.2.5 ***Ensuring Minimal Overhead.*** The monitoring system is designed to be non-intrusive, including the following characteristics:

- **Efficient Data Collection:** Uses lightweight API calls and avoids unnecessary data processing.
- **Asynchronous Operations:** Performs monitoring tasks asynchronously where possible to prevent blocking critical operations.
- **Configurable Intervals:** Allows adjustment of monitoring frequency based on system requirements.

3.2.6 ***Benefits of the Monitoring Approach.*** By implementing an effective monitoring system, the EWS can:

- Continuously assess its performance under varying workloads.
- Provide critical data for adaptation and learning components.
- Ensure that resource utilization is optimized while maintaining high-quality service.

3.3 Learning

Considering the nature of the self-adaptive system — a web server that handles multiple types of requests at varying frequencies — the demands placed on the software may change over time, rendering the current configuration suboptimal. An exhaustive search for the best configuration, which would cause the system to remain in suboptimal states for an extended period, is clearly undesirable. Therefore, it is crucial to strike a balance between exploring untested areas of the search space and exploiting solutions already known to be effective.

Therefore, in this study, we evaluate two online reinforcement learning strategies that were implemented based on the multi-armed bandit problem with the objective to find the configuration, among all 42 possibilities, that best suits the running systems current workload in an agile manner. To illustrate our multi-armed problem, the EWS is a bandit with 42 playable arms (the available component configurations), and the reward given by playing an arm (choosing a configuration for the web server) is defined by the metrics produced by the EWS (the average response time of the current configuration to client requests).

The learning strategies for the EWS implemented in this work are the aforementioned Epsilon-greedy and UCB1 algorithms. Each was built into respective classes (class UCB1 and class EpsilonGreedy) with very similar structure. For this project, a third class (class ewsAPI) was constructed to connect with the EWS API in order to retrieve the response time metric from the remote procedure call (proxy). The following subsections describe the learning models developed in detail.

3.3.1 ***Core iteration loop.*** The main iteration loop for both classes (methods `run_UCB1()` and `run_epsilon_greedy()`) work in very similar manners:

(1) The outer loop runs the experiment for a specified number of iterations;
(2) In each iteration the `select_arm` method is used to select which configuration to test next;
(3) The chosen configuration is applied to the system through the API;
(4) There is a warm-up period to allows the system to stabilize after the configuration change before taking measurements;
(5) Then a specified number of response time samples for the set configuration is collected through the API;
(6) Finally, it processes the results:
  (a) Calculates the average response time from all samples.
  (b) Converts the response time to milliseconds and makes it negative to create a reward (negative because lower response times are better in this context).
  (c) Uses the update method to adjust the cumulative reward of the set configuration.

3.3.2 ***Reward definition for the learning strategies.*** Considering the objective of the learning strategies is to find the configuration that returns the lowest response time for the servers current workload, the reward value considered is the negative number of the average response time in miliseconds. We do this because lower response times are better in this context, considering the multi-armed bandit problem is used to maximize the reward and only flipping the metrics sign gives a more direct representation of the systems performance.

With a new configuration set to rule the EWS, its total reward value is reprocessed after a new average response time is collected. It increments the count of times the configuration has been selected and updates the cumulative reward value of the chosen configuration based on the obtained reward and recalculates the average reward value for that configuration: dividing the total cumulative reward value of the configuration by the number of times it was tested, as shown in Algorithm 1.

```
def update(self, chosen_arm, reward):
    self.counts[chosen_arm] += 1
    self.total_pulls += 1

    # Update total reward and average value
    self.total_reward[chosen_arm] += reward
    self.values[chosen_arm] = self.total_reward[chosen_arm]
                                / self.counts[chosen_arm]
```

Algorithm 1: Episilon-greedy and UCB1 update method.

```python
def select_arm(self):
    # First try all arms at least once
    for arm in range(self.n_arms):
        if self.counts[arm] == 0:
            return arm

    # Epsilon-greedy selection
    if np.random.random() < self.epsilon:
        # Explore: choose random arm
        return np.random.randint(self.n_arms)
    else:
        # Exploit: choose best arm
        return np.argmax(self.values)
```

Algorithm 2: Epsilon-greedy `select_arm` method.

3.3.3 ***Epsilon-greedy arm selection.*** For the epsilon-greedy arm selection logic, an initial exploration is performed to ensure that each configuration is tested at least once. If a configuration has not yet been tested, it is selected.

After all configurations have been tested at least once, the configuration selection follows the epsilon-greedy logic presented in Algorithm 2.

A value between 0.0 and 1.0 is randomly chosen, and then, if the chosen value is less than the epsilon value (0.1 by default), the method selects a configuration randomly, allowing for exploration of new options, otherwise, if the chosen value is greater than the epsilon value (0.1 by default), the method selects the configuration with the highest average reward, focusing on exploiting the best-known option.

3.3.4 ***UCB1 arm selection.*** Just like the epsilon-greedy arm selection method, this method performs an initial exploration to ensure that each configuration is tested at least once. If a configuration has not yet been tested, it is selected.

Once every configuration has been tested at least once, the method proceeds to calculate the UCB (Upper Confidence Bound) for each arm that has been tested, which is the result of two main factors:

- `average_reward`: The average reward obtained by the arm so far.

$$\text{average\_reward} = \frac{\text{total\_reward}[arm]}{\text{counts}[arm]}$$

- `exploration_factor`: A term that encourages exploration, becoming smaller as an arm is pulled more frequently, balancing exploration and exploitation. It is calculated as:

$$\text{exploration\_factor} = \sqrt{\frac{2 \cdot \ln(\text{total\_pulls})}{\text{counts}[arm]}}$$

- Finally, the two terms are summed:

$$\text{ucb\_values}[arm] = \frac{\text{total\_reward}[arm]}{\text{counts}[arm]} + \sqrt{\frac{2 \cdot \ln(\text{total\_pulls})}{\text{counts}[arm]}}$$

Afterward, the best configuration is selected, i.e., the configuration with the highest UCB value in the current iteration, as shown in Algorithm 3.

```python
def select_arm(self):
    # First try all arms at least once
    for arm in range(self.n_arms):
        if self.counts[arm] == 0:
            return arm

    # Calculate UCB values for each arm
    ucb_values = np.zeros(self.n_arms)
    for arm in range(self.n_arms):
        if self.counts[arm] > 0:
            average_reward = self.values[arm]
            exploration_factor = math.sqrt(2 *
                    math.log(self.total_pulls) / self.counts[arm])
            ucb_values[arm] = average_reward + exploration_factor

    return np.argmax(ucb_values)
```

Algorithm 3: UCB1 `select_arm` method.

4 Evaluation

4.1 EWS Metric Analysis

In this paper, critical performance metrics such as CPU usage, memory usage, throughput, and reward are collected and calculated using various functions within the monitor component. This subsection provides a detailed explanation of how these metrics are handled, along with code snippets for clarity.

4.1.1 ***CPU Usage.*** The CPU usage percentage of the Docker container is calculated using the *calculate_cpu_percent* function presented in Algorithm 4. These metrics related to CPU usage are obtained by the following process:

- *Retrieve CPU Statistics*: The function calculates the difference in total CPU usage (`cpu_delta`) and the difference in system CPU usage (`system_delta`) between the current and previous readings.
- *Calculate CPU Percentage*: If `system_delta` is greater than zero, it computes the CPU usage percentage using the equation:

$$\text{cpu\_percent} = \left(\frac{\text{cpu\_delta}}{\text{system\_delta}}\right) \times \text{num\_cpus} \times 100.0$$

- *Error Handling*: If a `KeyError` occurs, it logs the error and returns 0.0.

```python
def calculate_cpu_percent(stats):
    try:
        cpu_delta = stats['cpu_stats'] ['cpu_usage']['total_usage'] -
                  stats['precpu_stats']['cpu_usage']['total_usage']
        system_delta = stats ['cpu_stats'] ['system_cpu_usage'] -
                       stats ['precpu_stats'] ['system_cpu_usage']
        num_cpus = stats ['cpu_stats'].get('online_cpus', 1)
        if system_delta > 0.0:
            cpu_percent = (cpu_delta / system_delta) *
                          num_cpus * 100.0
        else:
            cpu_percent = 0.0
            return cpu\_percent
    except KeyError as e:
        logging.error(f"Error calculating CPU usage: {e}")
    return 0.0
```

Algorithm 4: CPU Usage Calculation

4.1.2 ***Memory Usage and Throughput.*** Memory usage and network throughput are collected using the *get_container_stats* function demonstrated in Algorithm 5. This algorithm implements the following operations and process:

- *Access Docker Container*: Connects to the Docker container specified by `container_name`.
- *Collect Statistics*: Retrieves real-time statistics using `container.stats(stream=False)`.
- *Memory Usage Calculation*:
  - `mem_usage`: Current memory usage converted from bytes to megabytes.
  - `mem_limit`: Maximum memory limit for the container.
- *Throughput Calculation*:
  - Sums up the *rx_bytes* (received bytes) and *tx_bytes* (transmitted bytes) from all network interfaces.
  - Converts the total bytes to megabytes and computes the difference from `prev_throughput`.
- *Return Metrics*: Returns the CPU percentage, memory usage, and throughput.

```python
def get_container_stats(docker_client, prev_throughput,
                                        container_name='ews'):
  try:
     container = docker_client.containers.get(container_name)
     stats = container.stats(stream=False)

     # CPU usage
     cpu_percent = calculate_cpu_percent(stats)

     # Memory usage in MB
     mem_usage = stats['memory_stats']['usage'] / (1024 ** 2)
     mem_limit = stats['memory_stats']['limit'] / (1024 ** 2)

     # Network I/O for throughput calculation
     net_io = stats['networks']
     net_input = sum(net['rx_bytes'] for net in net_io.values())
     net_output = sum(net['tx_bytes'] for net in net_io.values())
     throughput = (net_input + net_output) / (1024 ** 2) -
                     prev_throughput  # In MB

     logging.info(f"CPU: {cpu_percent:.2f}%, Memory:
                {mem_usage:.2f}/{mem_limit:.2f} MB, Throughput:
                {throughput:.2f}MB")
     return cpu_percent, mem_usage, throughput
  except Exception as e:
     logging.error(f"Error obtaining container statistics: {e}")
     return 0.0, 0.0, 0.0
```

Algorithm 5: Memory Usage and Throughput Collection

4.1.3 ***Reward Calculation.*** The *calculate_reward* function computes a reward based on the collected metrics, as shown in line 34 of the Algorithm 6. This function uses the following strategy and weights:

- *Weights Assignment*:
  - Response Time: 40%
  - Throughput: 30%
  - CPU Usage: 20%
  - Memory Usage: 10%
- *Normalization*:
  - Inverse is used for metrics where lower values are better (response time, CPU usage, memory usage).
  - Throughput is used directly since higher throughput is better.
- *Reward Calculation*: Computes a weighted sum of the normalized metrics to reflect overall performance.

4.1.4 ***Data Collection During Algorithm Execution.*** The metrics used by the adaptation system are collected during each iteration of the algorithm using the *run_algorithm* function presented in Algorithm 6. The strategy applied by this algorithm can be specified as follows:

```python
def run_algorithm(algorithm_name, selector_class, configs,
                                    total_iterations, docker_client):
  throughput = 0
  selector = selector_class(len(configs))
  logging.info(f"Starting algorithm {algorithm_name}")

  try:
    for iteration in range(1, total_iterations + 1):
        # Select configuration based on the algorithm
        # (e.g., UCB1, Greedy)
        arm = selector.select_arm()
        selected_config = configs[arm]
        logging.info(f"Iteration {iteration}: " +
                           "Changing to configuration {arm}")

        # Apply the selected configuration
        eRI.change_configuration(selected_config)
        time.sleep(5)  # Wait for the system to stabilize

        # Collect response time from the perception
        perception = eRI.get_perception()
        avg_response_time =
          perception.metric_dict.get('response_time').average_value()
          if perception else None
        logging.info(f"Avg Response Time: {avg_response_time}")

        # Collect CPU usage, memory usage, and throughput
        cpu_percent, mem_usage, throughput =
                get_container_stats(docker_client, throughput)

        # Calculate reward if all metrics are available
        if avg_response_time and cpu_percent is not None
                                        and mem_usage is not None:
            reward = calculate_reward(avg_response_time, throughput,
                                            cpu_percent, mem_usage)
        else:
            logging.warning("Insufficient metrics to calculate reward.")
            reward = 0

        # Update the algorithm with the obtained reward
        selector.update(arm, reward)

        # Optionally, store metrics for analysis
        # ...

  except Exception as e:
    logging.error(f"Error in algorithm {algorithm_name}: {e}")
```

Algorithm 6: Algorithm Execution and Data Collection

- *Configuration Selection*: The algorithm selects a configuration based on the current policy (e.g., UCB1, Greedy).
- *System Stabilization*: It waits a short period to allow the system to adapt to the new configuration.
- *Metric Collection*: It collects new metrics from the following sources:
  - *Response Time*: Obtained from the EWS perception data.

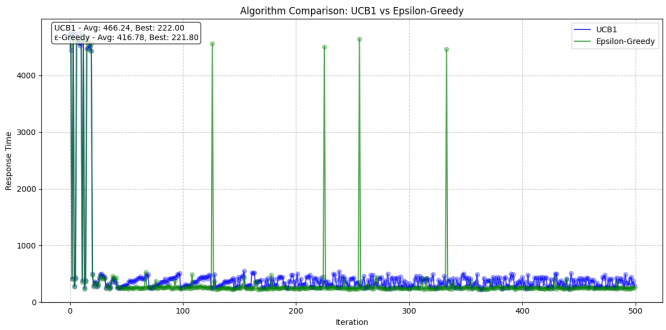

Figure 1: Text requests - 500 iterations

  - *CPU Usage, Memory Usage, Throughput*: Retrieved using *get_container_stats*.
- *Reward Calculation*: Uses the collected metrics to calculate the reward via *calculate_reward*.
- *Algorithm Update*: The reward is fed back into the selection algorithm to influence future decisions.
- *Iteration*: The process repeats for the specified number of iterations.

We extended the exemplar to collect and process the various metrics introduced. This enhancement enables the use of these metrics in different learning approaches by formulating a reward function that incorporates them. In this study, however, we focused on the response time metric in our cost function, with our learning strategies aimed at minimizing it.

4.2 Epsilon Greedy vs UCB1

With both implementations, Epsilon Greedy and UCB1 algorithms, we performed a set of experiments to evaluate the performance of both approaches. The experiments were designed to monitor the performance of the system running the different algorithms under different types of workloads. These workloads are characterized by the types of requests made by clients to the server.

The experiments used to evaluate the performance of the learning algorithms, as the orchestrating tool for the adaptation of the EWS, involved two different types of workloads: text and image. These experiments considered scenarios with: (i) single type of request workflow, i.e., scenarios in which only text or only images were requested, and (ii) simultaneous type of request workflow, i.e., scenarios in which both text and images were requested simultaneously.

Figure 1 presents the results for a scenario involving a single type of request workflow, where only text requests were considered. The experiment consisted of 500 interactions. Both algorithms demonstrated similar performance during the initial exploration phase. Subsequently, the UCB1 algorithm avoided selecting configurations with very low performance but continued to explore configurations with similar performance. In contrast, the Epsilon-Greedy algorithm prioritized the best configuration while occasionally selecting other configurations randomly, sometimes leading to the selection of configurations with very low performance. Ultimately, based on the average response time, the Epsilon-Greedy algorithm outperformed UCB1. This outcome reflects the absence of changes in request behavior, with Epsilon-Greedy prioritizing the best configuration identified during the exploration phase (while only occasionally exploring other configurations), whereas UCB1 more frequently explored configurations with comparable performance.

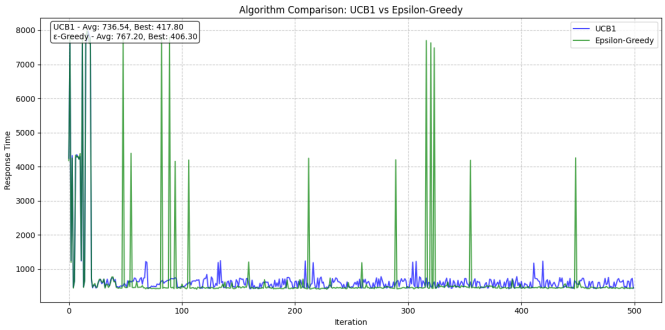

Figure 2: Image and Text requests - 500 iterations

Figure 2 presents the results for a scenario involving a simultaneous request workflow, where both text and image requests were made. Some configurations exhibited significantly lower response times for image requests compared to text requests. The experiment was conducted over 500 interactions. As in the previous scenario, both algorithms demonstrated similar performance during the initial exploration phase. However, in this case, the UCB1 algorithm outperformed the Epsilon-Greedy algorithm in terms of average response time. This advantage can be attributed to UCB1's ability to focus on exploring configurations with good performance while avoiding those with lower response times. Additionally, UCB1 adjusts to changes in request behavior by not staying too long with configurations that eventually begin to perform poorly. In contrast, Epsilon-Greedy occasionally continued to select configurations randomly, which deteriorated in performance under the changing environment.

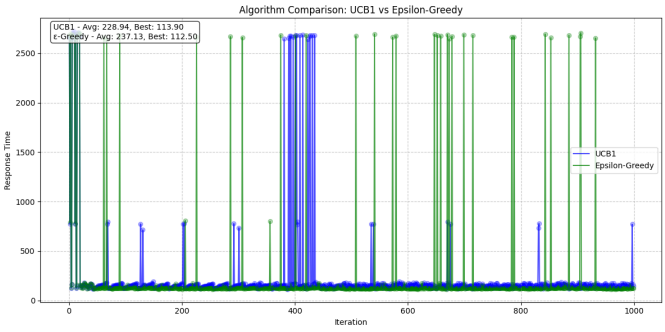

Figure 3: Image requests - 1000 iterations

Figure 3 presents the results for a scenario involving a single-type request workflow, where only image requests were considered. The experiment consisted of 1000 interactions. As in the previous scenarios, both algorithms demonstrated similar performance during the initial exploration phase. However, the UCB1 algorithm

outperformed the Epsilon-Greedy algorithm under similar environmental conditions observed in the previous experiment, where some configurations began to perform poorly following changes in request behavior.

The developed algorithms leverage the concept of multi-armed bandits to balance exploration and exploitation, adaptively optimizing EWS's configuration based on workload patterns. The goal of dynamically select the most efficient component composition for the web server at runtime was achieved, balancing the need to test different configurations (exploration) with choosing the best-known configuration (exploitation).

5 Final Remarks

In this paper, we experimented with multi-armed bandit algorithms in the context of the Emergent Web Server, a publicly available exemplar designed to explore learning strategies for runtime adaptation rule discovery.

We extended the exemplar to include additional monitoring metrics, enabling future experimentation with new reward/cost functions to explore diverse learning approaches. Furthermore, we evaluated the system using two multi-armed bandit algorithms: UCB1 and Epsilon-Greedy.

Our results demonstrate the effectiveness of UCB1 in identifying and converging on the optimal composition of the web server across various operating environments. UCB1 proved more efficient in terms of convergence accuracy and speed, requiring less time to stabilize on the optimal composition. In contrast, while Epsilon-Greedy successfully converged to the correct composition, its inherent exploration behavior caused periodic deviations. UCB1, on the other hand, consistently maintained the optimal composition in most scenarios, resulting in a better average response time for the system.

For future work, we plan to explore other multi-armed bandit algorithms, such as Thompson Sampling, and compare our results with those reported in [13]. Additionally, investigating the tuning of the exploration constant in the UCB1 algorithm could be valuable for improving its convergence in scenarios where it failed to stabilize. We also aim to explore multi-agent approaches, such as those reported in [20], incorporating sophisticated communication protocols. In particular, we consider argumentation-based dialogue protocols, which enable agents to deliberate and argue about the optimal system configuration [21, 22].

References

[1] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[2] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pages 1–32. Springer, 2013.

[3] Barry Porter, Penn Faulkner Rainford, and Roberto Rodrigues-Filho. Self-designing software. *Communications of the ACM*, 68(1):50–59, 2025.

[4] Olga Kouchnarenko and Jean-François Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *International Workshop on Formal Aspects of Component Software*, pages 234–253. Springer, 2013.

[5] Paul Grace, Danny Hughes, Barry Porter, Gordon S. Blair, Geoff Coulson, and Francois Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. *SIGOPS Oper. Syst. Rev.*, 42(4):123–136, April 2008. ISSN 0163-5980. doi: 10.1145/1357010.1352606. URL https://doi.org/10.1145/1357010.1352606.

[6] Gordon Blair. Complex distributed systems: The need for fresh perspectives. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1410–1421, 2018. doi: 10.1109/ICDCS.2018.00142.

[7] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 7–16, 2010.

[8] Omid Gheibi, Danny Weyns, and Federico Quin. Applying machine learning in self-adaptive systems: A systematic literature review. *ACM Trans. Auton. Adapt. Syst.*, 15(3), August 2021. ISSN 1556-4665. doi: 10.1145/3469440. URL https://doi.org/10.1145/3469440.

[9] Roberto Rodrigues Filho, Elvin Alberts, Ilias Gerostathopoulos, Barry Porter, and Fábio M. Costa. 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 '22, page 36–42, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393058. doi: 10.1145/3524844.3528079. URL https://doi.org/10.1145/3524844.3528079.

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

[11] Chris Dann, Yishay Mansour, Mehryar Mohri, Ayush Sekhari, and Karthik Sridharan. Guarantees for epsilon-greedy reinforcement learning with function approximation. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 4666–4689. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/dann22a.html.

[12] Barry Porter and Roberto Rodrigues Filho. Distributed emergent software: Assembling, perceiving and learning systems at scale. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 127–136, 2019. doi: 10.1109/SASO.2019.00024.

[13] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. REX: A development platform and online learning approach for runtime emergent software systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 333–348, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter.

[14] Paul Grace, Danny Hughes, Barry Porter, Gordon S. Blair, Geoff Coulson, and Francois Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, page 123–136, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580135. doi: 10.1145/1352592.1352606. URL https://doi.org/10.1145/1352592.1352606.

[15] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 299–310, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568272. URL https://doi.org/10.1145/2568225.2568272.

[16] Chenyang Lu, Ying Lu, T.F. Abdelzaher, J.A. Stankovic, and Sang Hyuk Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9): 1014–1027, 2006. doi: 10.1109/TPDS.2006.123.

[17] Wei Zhang, Timothy Wood, and Jinho Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 5–14, 2016. doi: 10.1109/ICAC.2016.28.

[18] Frederico Meletti Rappa, Roberto Rodrigues-Filho, Alison R. Panisson, Leandro Soriano Marcolino, and Luiz Bittencourt. Multi-armed bandits for self-distributing stateful services across networking infrastructures. In *NOMS 2024 IEEE/IFIP Network Operations and Management Symposium*, 2024.

[19] Mohammad Alsomali, Roberto Rodrigues-Filho, Leandro Soriano Marcolino, and Barry Porter. An online incremental learning approach for configuring multi-arm bandits algorithms. In *ECAI*, July 2024.

[20] Bernardo Pandolfi Costa, Heitor Henrique da Silva, Analucia S. Morales, Luiz F. Bittencourt, Alison R. Panisson, and Roberto Rodrigues-Filho. A multi-agent approach to self-distributing systems. In *International Conference on Advanced Information Networking and Applications (AINA)*, 2025.

[21] Alison R Panisson, Felipe Meneguzzi, Renata Vieira, and Rafael H Bordini. Towards practical argumentation-based dialogues in multi-agent systems. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 2, pages 151–158. IEEE, 2015.

[22] Alison R Panisson, Felipe Meneguzzi, Renata Vieira, and Rafael H Bordin. Towards practical argumentation in multi-agent systems. In *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 98–103. IEEE, 2015.