Feature Model-Guided Online Reinforcement Learning for Self-Adaptive Services*

Andreas Metzger[0000−0002−4808−8297]1, Clément Quinton[0000−0003−3203−6107]2, Zoltán Ádám Mann[0000−0001−5741−2709]1, Luciano Baresi[0000−0001−6467−837X]3, and Klaus Pohl[0000−0003−2199−5257]1

[1] paluno, University of Duisburg-Essen, Essen, Germany
firstname.lastname@paluno.uni-due.de
[2] University of Lille, Inria, CRIStAL UMR CNRS 9189, Lille, France
clement.quinton@univ-lille.fr
[3] Politecnico di Milano, Milan, Italy
luciano.baresi@polimi.it

Abstract. A self-adaptive service can maintain its QoS requirements in the presence of dynamic environment changes. To develop a self-adaptive service, service engineers have to create self-adaptation logic encoding when the service should execute which adaptation actions. However, developing self-adaptation logic may be difficult due to design time uncertainty; e.g., anticipating all potential environment changes at design time is in most cases infeasible. Online reinforcement learning addresses design time uncertainty by learning suitable adaptation actions through interactions with the environment at runtime. To learn more about its environment, reinforcement learning has to select actions that were not selected before, which is known as exploration. How exploration happens has an impact on the performance of the learning process. We focus on two problems related to how a service's adaptation actions are explored: (1) Existing solutions randomly explore adaptation actions and thus may exhibit slow learning if there are many possible adaptation actions to choose from. (2) Existing solutions are unaware of service evolution, and thus may explore new adaptation actions introduced during such evolution rather late. We propose novel exploration strategies that use feature models (from software product line engineering) to guide exploration in the presence of many adaptation actions and in the presence of service evolution. Experimental results for a self-adaptive cloud management service indicate an average speed-up of the learning process of 58.8% in the presence of many adaptation actions, and of 61.3% in the presence of service evolution. The improved learning performance in turn led to an average QoS improvement of 7.8% and 23.7% respectively.

Keywords: Adaptation, reinforcement learning, feature model, cloud service

* This paper was published in the *Proceedings of the 18th International Conference on Service-Oriented Computing (ICSOC 2020)*, pages 269-286, 2020

1 Introduction

A *self-adaptive* service is capable of modifying its own structure and behavior at runtime based on its perception of the environment, of itself and of its requirements [9,28,20]. As an example, take a self-adaptive web service. Faced with a sudden increase in workload, the web service may reconfigure itself by deactivating optional system features. An online store, for instance, may deactivate its resource-intensive recommender engine in the presence of a high workload. By adapting itself at runtime, the web service is able to maintain its QoS requirements (here: performance) under changing workloads.

To develop a self-adaptive service, service engineers have to develop *self-adaptation logic* that encodes when and how the service should adapt itself. Among other concerns, this requires anticipating the potential environment states the service may encounter at runtime to define when the service should adapt itself. However, anticipating all potential environment states at design time is in most cases infeasible due to *design time uncertainty* [8,10]. In addition, due to simplified design assumptions, the precise effect of an adaptation action may not be known and thus accurately determining how the service should adapt itself is difficult [10]. As an example, while service engineers may know in principle that activating more features will have a negative impact on performance, exactly determining the performance impact is more challenging [30].

Online reinforcement learning (RL) is an emerging approach to address design time uncertainty of self-adaptive services by employing RL at runtime (see existing solutions discussed in Sec. 6). In general, RL aims to learn suitable actions via an agent's interactions with its environment [31]. The agent receives a reward for executing an action. The reward expresses how suitable that action was. The goal of RL is to optimize cumulative rewards.

1.1 Problem Statement

RL faces the exploration-exploitation dilemma [31]. To optimize cumulative rewards, actions should be selected that have shown to be suitable, which is known as *exploitation*. However, to discover such actions in the first place, actions that were not selected before should be selected, which is known as *exploration*. How exploration happens has an impact on the performance of the learning process [31,4,13]. We focus on two problems related to how a service's set of possible adaptation actions, *i.e.*, its *adaptation space*, is explored.

Random exploration of adaptation space. Existing online RL solutions for self-adaptive services propose randomly selecting adaptation actions for exploration (see Sec. 6). The effectiveness of exploration therefore directly depends on the size of the adaptation space, because each adaptation action has an equal chance of being selected. Some RL algorithms can cope with a large space of actions, but require that the space of actions is continuous in order to generalize over unseen actions [23]. Self-adaptive services may have large, discrete adaptation spaces; *e.g.*, if their adaptation actions entail changes of service compositions [22] or reconfigurations of service features [19]. A simple example is

a service composition consisting of eight abstract services that may allow dynamically binding 2 concrete services each. Assuming no temporal or logical constraints on adaptation, this constitutes $2^8 = 256$ possible adaptation actions. In the presence of such large, discrete adaptation space, random exploration thus may lead to slow learning at runtime [31,4,13].

Evolution-unaware exploration of adaptation space. Existing online RL solutions are unaware of service evolution [16,29]. They do not consider that a self-adaptive service – like any service – may undergo evolution [25]. In contrast to self-adaptation, which refers to the automatic modification of the service by itself, evolution refers to the modification of the service by humans [20]. During evolution, service engineers may modify the service to correct bugs, remove no longer used features, or introduce new features. Service evolution means that the adaptation space may change, *e.g.*, existing adaptation actions may be removed or new adaptation actions may be added. Some RL algorithms can cope with environments that change over time, so called non-stationary environments [31,23]. However, a change of the adaptation space cannot be determined by observing the environment, as the adaptation space is an intrinsic property of the RL agent. As a result, existing solutions may explore new adaptation actions only with low probability (as all adaptation actions have an equal chance of being selected). It may thus take quite long until the new adaptation actions have been explored.

1.2 Contributions

We introduce exploration strategies for online RL that address (1) a service's potentially large adaptation space, and (2) changes of its adaptation space due to evolution. Our exploration strategies use *feature models* [21] to give structure to the service's adaptation space and thereby leverage additional information to guide exploration. A feature model is a tree or a directed acyclic graph of features, organized hierarchically. An adaptation action is represented by a valid feature combination specifying the target run-time configuration of the service.

Our strategies traverse the feature model to select the next adaptation action to be explored. By leveraging the structure of the feature model, our strategies guide the exploration process. In addition, our strategies detect added and removed adaptation actions by analyzing the change of the feature model due to evolution. Adaptation actions removed as a result of evolution are no longer explored, while added adaptation actions are explored first.

We implement our strategies as part of the Q-Learning RL algorithm [31] widely used in the related work (see Sec. 6). We experimentally assess our strategies using an actual cloud resource management service and compare the learning performance with that of the widely used ϵ-greedy random exploration strategy.

In what follows, Sec. 2 explains fundamentals and a running example. Sec. 3 describes our exploration strategies and how they are integrated with RL algorithms. Sec. 4 presents the design and results of our experiments. Sec. 5 provides a critical discussion. Sec. 6 analyzes related work.

2 Fundamentals

Feature Models and Self-adaptation. A *feature model* is a tree of features organized hierarchically [21]. A feature can be decomposed into mandatory, optional or alternative sub-features. A mandatory sub-feature has to be activated if its parent feature is activated. While an optional sub-feature may or may not be activated, at least one of the alternative sub-features has to be activated if their parent feature is activated. Additional constraints, such as "excludes" or "requires", express inter-feature dependencies. Thereby, a feature model describes the possible and allowed feature combinations.

Feature models are traditionally used in software product line engineering to define the set of system variants at design time [21]. Dynamic software product lines extend the use of feature models to describe possible run-time configurations of a system [14]. A feature model thereby can be used to define a self-adaptive system's adaptation space, where each adaptation action is expressed in terms of a possible runtime configuration, *i.e.*, feature combination [12].

Fig. 1 shows the feature model of a self-adaptive web service as an example. The DataLogging feature is mandatory (which means it is always active), while the ContentDiscovery feature is optional. The DataLogging feature has three alternative sub-features, *i.e.*, at least one data logging sub-feature must be active: Min, Medium or Max. The ContentDiscovery feature has two optional sub-features Search and Recommendation. The constraint Recommendation $\Rightarrow$ Max $\vee$ Medium specifies that a sufficient level of data logging is required to collect enough information about the web service's users and transactions to make good recommendations.

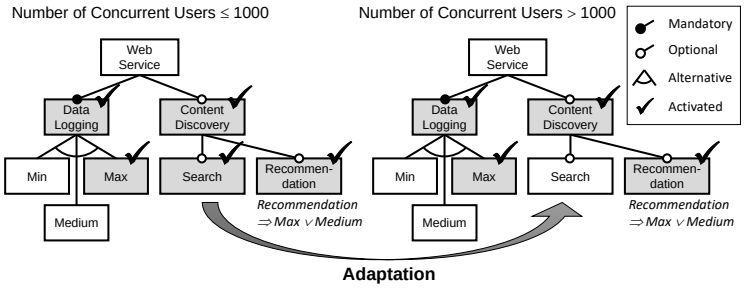

Fig. 1: Feature model and adaptation of example web service

Let us consider the above web service should adapt to changing number of concurrent users to keep its response time below 500ms. A service developer may express an adaptation rule for the web service such that it turns off some of the features in the presence of more users, thereby reducing the resource needs of the service. The right-hand side of Fig. 1 shows a concrete example for such an adaptation. If the service faces an environment state of more than 1000 concurrent users, the service self-adapts by deactivating the Search feature.

Reinforcement Learning (RL). RL aims to learn suitable actions via an agent's interactions with its environment [31]. At a given time step t, the agent selects an action a (from its adaptation space) to be executed in environment

state s (see Fig. 2). As a result, the environment transitions to s' at time step $t+1$ and the agent receives a reward r for executing the action. The reward r together with the information about the next state s' are used to update the knowledge of the agent. The goal of RL is to optimize cumulative rewards. As mentioned in Sect. 1, a trade-off between *exploitation* (using current knowledge) and *exploitation* (gathering new knowledge) must be made. For a self-adaptive service, "agent" refers to the self-adaptation logic of the service and "action" refers to an adaptation action [24].

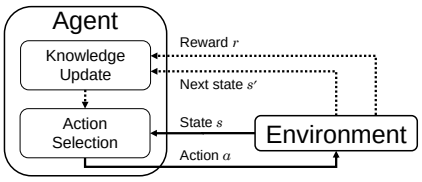

Fig. 2: RL concept

3 Feature-Model-Guided Exploration

As motivated in Sec. 1, our exploration strategies use feature models (FM) to guide the exploration process. We first explain how our *FM-guided exploration strategies* can be integrated into an existing RL algorithm and then introduce the realization of these strategies.

3.1 Integration into Reinforcement Learning

Algorithm 1 shows how our FM-guided strategies can be integrated into RL by using the well-known Q-Learning algorithm as basis. We chose Q-Learning because it is the most widely used algorithm in the related work (see Sec. 6).

Algorithm 1 Q-Learning with FM-guided Exploration

```
1: function FMQ-Learning(FeatureModel M; Double α, γ, ϵ, δ)
2:     Initialize Q(s, a) for all s ∈ S (state space) and a ∈ A (adaptation space);
3:     Determine current state s;
4:     repeat
5:         Set<Feature> a = getNextAction(M, s); // Action Selection
6:         Adapt service to configuration a; Observe reward r; Observe new state s';
7:         Q(s, a) ← Q(s, a) + α[r + γmax_{a'∈A} Q(s', a') − Q(s, a)]; // Knowledge Update
8:         s ← s';
9:     until last time step
10: end function
11:
12: function getNextAction(FeatureModel M, State s)
13:     Set<Feature> a ← argmax_a Q(s, a); // Exploit existing knowledge
14:     InitFMExploration(M, a); // initialize the FM-guided strategies, see Algorithm 2
15:     if random() < ϵ then // Explore new actions
16:         if random() < δ then return getRandomConfiguration(M);
17:         else
18:             return getNextConfiguration(); // see Algorithm 2
19:         end if
20:     end if
21:     return a;
22: end function
```

Q-Learning employs a value function for representing the learned knowledge. The value function $Q(s, a)$ gives the expected cumulative reward when perform-

ing a particular action a in a given state s [31]. Q-Learning offers two hyper-parameters: the learning rate α, which defines to what extent newly acquired knowledge overwrites old knowledge, and the discount factor γ, which defines the relevance of future rewards (see knowledge update in line 7).

Our strategies are integrated into RL within the getNextAction function, which selects the next adaptation action while trading off exploration and exploitation. To make this trade-off we use the ϵ-greedy strategy as a baseline, as it is a standard action selection strategy in reinforcement learning and the most widely used strategy in the related work (see Sec. 6). With probability $1-\epsilon$, ϵ-greedy exploits existing knowledge by selecting the action a that has the highest Q value and thus highest expected reward (line 13). With probability ϵ, ϵ-greedy randomly explores a new action. In contrast to this random exploration, we use our FM-guided exploration strategies by calling the getNextConfiguration function (line 18). The different realizations of getNextConfiguration are explained below. To prevent FM-guided exploration from prematurely converging to a local minimum, we follow the literature and use a small amount of randomness [26], *i.e.*, we perform random exploration with probability $\delta \cdot \epsilon$.

3.2 Leveraging the Feature Model Structure for Exploration

Incremental Exploration Strategy. This strategy takes advantage of the semantics typically encoded in the structure of feature models. Non-leaf features in a feature model are usually abstract features used to better structure variability [36]. These abstract features often do not have an impact at implementation level, but delegate their realization to their sub-features. Sub-features thus may offer different realizations of their abstract parent feature. The sub-features of a common parent feature, *i.e.*, *sibling* features, can thus be considered semantically connected. In the example from Sec. 2, the ContentDiscovery feature has two sub-features Search and Recommendation offering different concrete ways how a user may discover online content. The idea behind the Incremental strategy is to exploit the information about these potentially semantically connected sibling features and explore them first before exploring other features. Note that this entails a random selection of the order of sub-features. Table 1 shows an excerpt of a typical exploration sequence of the Incremental strategy with the step-wise exploration of sibling features highlighted in gray.

	Logging	Min	Medium	Max	Content Disc.	Search	Recommend.
1	✓			✓	✓		✓
2	✓		✓		✓		✓
3	✓		✓		✓	✓	✓
4	✓			✓	✓	✓	✓
5	✓		✓		✓	✓	
6	✓	✓			✓	✓	
7	...	...	...	...	...	...	...

Table 1: Example exploration via Incremental strategy (excerpt)

The Incremental strategy is realized by Algorithm 2, which starts by randomly selecting an arbitrary leaf feature f (*i.e.*, a feature with no sub-features) among all leaf features that are part of the current configuration (lines 5– 6).

Then, the set of configurations $\mathcal{C}_f$ containing feature f is computed, while the sibling features of feature f are gathered into a dedicated *siblings* set (line 7).

Algorithm 2 Incremental Strategy

```
1: Set<Feature> leaves, configuration, siblings;
2: Set<Set<Feature>> C_f; Feature f;
3:
4: function InitFMExploration(FeatureModel M, Set<Feature> currentConfiguration)
5:     leaves ← getLeaves(currentConfiguration);
6:     f ← randomSelect(leaves);
7:     C_f ← getConfigurationsWithFeature(f); siblings ← siblings(f);
8: end function
9:
10: function getNextConfiguration()
11:     if C_f ≠ ∅ then
12:         configuration ← randomSelect(C_f); C_f ← C_f \ {configuration};
13:         return configuration;
14:     else
15:         if siblings ≠ ∅ then
16:             f ← randomSelect(siblings);
17:             siblings ← siblings \ {f}; C_f ← getConfigurationsWithFeature(f);
18:         else
19:             if parent(f) ≠ ∅ then
20:                 f ← parent(f); siblings ← siblings(f);
21:                 C_f ← getConfigurationsWithFeature(f);
22:             else // Root feature reached
23:                 return ∅;
24:             end if
25:         end if
26:         return getNextConfiguration();
27:     end if
28: end function
```

While $\mathcal{C}_f$ is non-empty, the strategy explores one randomly selected configuration from $\mathcal{C}_f$ and removes the selected configuration from $\mathcal{C}_f$ (lines 11–13). If $\mathcal{C}_f$ is empty, then a new set of configurations containing a sibling feature of f is randomly explored, provided such sibling feature exists (lines 15–17). If no configuration containing f or a sibling feature of f is found, then the strategy moves on to the parent feature of f, which is repeated until a configuration is found (line 13) or the root feature is reached (line 22).

Feature Degree Exploration Strategy. Even though the Incremental strategy makes use of the structure of the feature model, it still randomly determines the order in which leaf and sibling features are explored. To better guide the decision about which of these features to explore, we make use of the concept of feature degree. We define the feature degree for a given feature f as the number of configurations that contain f. The intuition here is that there may be a higher probability of finding a suitable configuration when considering features with high feature degrees, as they are present in more configurations.

In our example, the feature degree of Search is 5, while of Recommendation it is only 4 (due to the constraint requiring at least the Medium logging level). The Feature Degree strategy thus first explores all configurations involving the Search feature before exploring other configurations. Table 2 shows an excerpt of

a typical exploration sequence of the Feature Degree strategy (the exploration of the sibling feature with highest feature degree highlighted in gray).

	Logging	Min	Medium	Max	Content Disc.	Search	Recommend.
1	✔			✔	✔		✔
2	✔		✔		✔	✔	✔
3	✔			✔	✔	✔	✔
4	✔		✔		✔	✔	
5	✔	✔			✔	✔	
6	...	...	...	...	...	...	...

Table 2: Example exploration via Feature Degree strategy (excerpt)

The Feature Degree strategy is realized by modifying Algorithm 2 to make use of the feature degree as shown in Algorithm 3. On the one hand, the feature degree is used to determine which leaf feature to start exploring from. Instead of randomly selecting a leaf feature as done in Algorithm 2 (line 6), the Feature Degree strategy selects a leaf feature with the highest feature degree. On the other hand, instead of randomly choosing sibling features as done in Algorithm 2 (line 16), the Feature Degree strategy explores the sibling in descending order of their feature degrees. To realize the *featureDeg* function, existing feature model analysis tools, such as [35], can be used to efficiently compute the number of possible configurations containing f.

Algorithm 3 Feature Degree Strategy

```
 5:    leaves ← getLeaves(currentConfiguration);
 6:    f ← argmax_{f∈leaves}(featureDeg(f));
   [...]
16:        if siblings ≠ ∅ then
17:            f ← argmax_{f∈siblings}(featureDeg(f));
```

3.3 Leveraging Feature Model Differences for Exploration

To capture changes in the service's adaptation space due to evolution, we propose analyzing the differences in feature models before ($\mathcal{M}$) and after ($\mathcal{M}'$) an evolution step. Following the product line literature, we consider two main types of changes of feature models [34]:

Added configurations (feature model generalization). New configurations may be added to the adaptation space by (*i*) introducing new features to $\mathcal{M}'$, or (*ii*) removing or relaxing existing constraints (*e.g.*, by changing a sub-feature from mandatory to optional, or by removing "requires" or "excludes" constraints). In the example from Sec. 2, a new sub-feature Optimized might be added to the DataLogging feature, providing a more resource efficient logging implementation. Thereby, new configurations are added to the adaptation space, such as {DataLogging, Optimized, ContentDiscovery, Search}. As another example, the Recommendation implementation may have been improved and it now can work with the Min logging feature. This removes the constraint shown in Fig. 1, and adds new configurations such as {DataLogging, Min, ContentDiscovery, Recommendation}.

Removed configurations (feature model specialization). Symmetrical to above, configurations may be removed from the adaptation space by (*i*) removing features from $\mathcal{M}$, or (*ii*) by adding or tightening constraints in $\mathcal{M}'$.

To determine these changes of feature models, we compute a set-theoretic difference between valid configurations expressed by feature model $\mathcal{M}$ and feature model $\mathcal{M}'$. Detailed descriptions of feature model differencing as well as efficient tool support can be found in [1,5]. The feature model differences provide us with adaptation actions added to the adaptation space ($\mathcal{M}' \setminus \mathcal{M}$), as well as adaptation actions removed from the adaptation space ($\mathcal{M} \setminus \mathcal{M}'$).

Our evolution-aware strategies thus first explore the configurations that were added to the adaptation space, and then explore the remaining configurations if needed. The rationale is that added configurations might offer new opportunities for finding suitable adaptation actions and thus should be explored first. Configurations that were removed are no longer executed and thus the learning knowledge can be pruned accordingly. In the Q-Learning realization (Sec. 3.1), we remove all tuples (s, a) from Q, where a represents a removed configuration.

Such evolution-aware exploration can also be introduced to ϵ-greedy. Instead of randomly exploring the whole new adaptation space, exploration is limited to first randomly exploring the set of new configurations.

4 Experiments

We experimentally assess our FM-guided exploration strategies and compare them with ϵ-greedy as the strategy used in the related work (see Sec. 6).

Research Questions. We aim to answer the following research questions:

RQ1: How does learning performance using FM-guided exploration compare to using ϵ-greedy and how does it impact on QoS?

RQ2: How does learning performance using evolution-aware exploration strategies compare to evolution-unaware exploration and how does it impact on QoS?

Experiment Setup. We use a self-adaptive cloud resource management service, CloudRM[4], as subject system [17]. CloudRM controls the allocation of computational tasks to virtual machines (VMs) and the allocation of virtual machines to physical machines in a cloud data center. CloudRM can be adapted by reconfiguring it to use different allocation algorithms, and the algorithms can be adapted by using different sets of parameters. We implemented a separate adaption logic for CloudRM by using the extended Q-Learning algorithm as introduced in Sec. 3.1. In total, CloudRM provides 344 possible adaptation actions. These are structured in a feature model that is four levels deep and includes 65 different features. The feature model together with the code of our algorithms and the data of our experiments are available online[5].

Our experiments are based on a real-world workload trace with 10,000 tasks, in total spanning over a time frame of 29 days [18]. The CloudRM algorithms decide on the placement of new tasks whenever they are entered into the system

[4] `https://sourceforge.net/p/vm-alloc/task_vm_pm`

[5] `https://git.uni-due.de/online-reinforcement-learning/icsoc-2020-artefacts`

(as driven by the workload trace). To allow sufficient time in the experiment to observe the impact of an adaptation, CloudRM is allowed to run one hour before the next adaptation action is executed. For RQ2, the same workload was replayed after each evolution step to ensure consistency among the results.

We define the reward function for online RL as $r = -(\rho \cdot e + (1-\rho) \cdot m)$, with e being the energy consumption and m being the number of VM manipulations (*i.e.*, migrations and launches), each normalized to be on the same scale. We use $\rho = 0.8$, meaning we give priority to reducing energy consumption, while still maintaining a low number of VM manipulations. If several adaptation actions show similar energy consumption, the one that achieves this with less VM manipulations receives a higher reward.

To determine suitable hyper-parameter values (see Sec. 3.1), we performed hyper-parameter tuning (via grid search). We used the best performing learning rate $\alpha = 0.85$ and discount factor $\gamma = 0.2$ for ϵ-greedy and used this also for our FM-guided strategies. To facilitate convergence of the learning process, we used an ϵ decay approach. This is a typical approach in RL, meaning that ϵ starts at 1 and diminishes with a predefined rate after each time step. We used an ϵ decay rate of 0.97 (*i.e.*, $\epsilon < 1\%$ after time step 150), as this led to fastest convergence with highest asymptotic rewards for ϵ-greedy. For the FM-guided strategies we used a δ decay rate of 0.9 (*i.e.*, $\delta < 1\%$ after time step 45). Due to the stochastic nature of the learning strategies (both ϵ-greedy and to a lesser degree our strategies involve random decisions), we repeated the experiment 100 times and averaged results.

Results for RQ1. Fig. 3 visualizes the learning process for the different exploration strategies by showing how rewards develop over time. As visible, the FM-guided exploration strategies (Incremental and Feature Degree) more quickly reach maximum rewards than ϵ-greedy (our baseline).

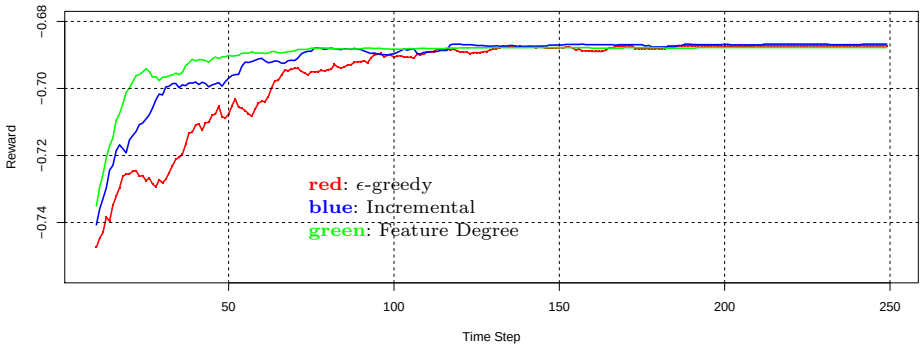

Fig. 3: Learning curves (RQ1)

Table 3 characterizes the learning process of the different strategies by using the metrics presented in [32]: *Asymptotic performance* (maximum reward at end of learning process, here: average rewards of time steps 200–250), *total reward* (area between reward curve and asymptotic reward), *time to threshold* (time

step when x% of asymptotic reward is reached for first time, here: $x = 90$), *jumpstart* (rewards at beginning of learning process, here: at time step 10). In addition, the table shows how the learning performance of the different strategies impact on the QoS characteristics of CloudRM.

Results indicate that our FM-guided exploration strategies lead to a consistent improvement of the learning process. In addition, the Feature Degree strategy performs better than the Incremental strategy, suggesting that considering additional information about the service's features has an effect. Our FM-guided strategies perform better when compared with ϵ-greedy wrt. total reward (58.8% on average), time to threshold (48.6% on average), and jumpstart (1.3% on average), while performing comparably wrt. asymptotic performance. Considering the impact on QoS, FM-guided learning consistently leads to less VM manipulations and sightly lower energy consumption. While savings in energy are rather small (less than 1%), FM-guided learning reduces the number of virtual machine manipulations by 7.8% on average. This is caused by the different placement algorithms having a rather small difference wrt. energy optimization, but having a much larger difference wrt. optimizing the number of virtual machine manipulations.

	Learning performance				**QoS impact**	
	Asymptotic performance	Time to threshold	Jumpstart	Total reward	Energy (kWh)	Number VM Manipulations
ϵ-greedy	-0.6873	74	-0.7474	-2.0110	2511	761
Incremental	-0.6868	47	-0.7407	-0.9946	2507	713
Improvement	*0.1%*	*36.5%*	*0.9%*	*50.5%*	*0.1%*	*6.2%*
Feature Degree	-0.6878	29	-0.7351	-0.6644	2508	690
Improvement	*-0.1%*	*60.8%*	*1.7%*	*67.0%*	*0.1%*	*9.3%*
Avg. improvement	*0.0%*	*48.6%*	*1.3%*	*58.8%*	*0.1%*	*7.8%*

Table 3: Comparison of exploration strategies (RQ1)

Results for RQ2. We compare three evolution-aware strategies (evolution-aware ϵ-greedy, evolution-aware Incremental, and evolution-aware Feature Degree) with their respective evolution-unaware counterparts (*i.e.*, the strategies used for RQ1). It should be noted that even though we provide the evolution-unaware strategies with the information about the changed adaptation space (so they can fully explore it), we have not modified them such as to differentiate between old and new adaptation actions.

We use a 3-step evolution scenario incrementally adding features and thus adaptation actions to CloudRM. Initially, CloudRM offers the Simple placement feature (creating a dedicated virtual machine for each task) and Multiple placement features (allowing a given number of tasks to be deployed on a virtual machine), offering 26 adaptation actions. In evolution step #1, the Maxsize placement feature is added, which creates virtual machines of a fixed capacity and selects virtual machines using the First-Fit (FF) heuristic, adding 30 adaptation actions. In evolution step #2, the Maxsize placement feature is enhanced by allowing different VM capacities and adding two new virtual machine selection heuristics: Best-Fit (BF) and Worst-Fit (WF), adding 72 adaptation actions. In evolution step#3, the Consolidation_Friendly placement feature is added, which se-

lects a physical machine that can accommodate the given task, and then selects a virtual machine hosted on the physical machine, adding 216 adaptation actions.

Like for RQ1, Fig. 4 visualizes the learning process for the different exploration strategies. After each evolution step, we observe the learning process for 250 time steps, before moving to the next step of the evolution scenario.

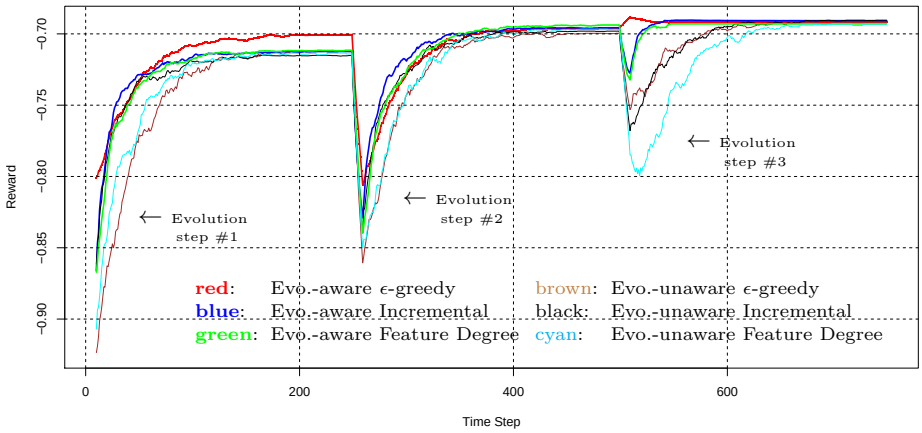

Fig. 4: Learning curves across evolution steps (RQ2)

Table 4 shows the results of learning performance and QoS impact across all three evolution steps. We computed the metrics separately for each of the evolution steps and report their averages.

	Learning performance				**QoS impact**	
	Asymptotic performance	Time to threshold	Jumpstart	Total reward	Energy (kWh)	Number VM Manipulations
ϵ-greedy:						
Evo.-aware	-0.6964	35.33	-0.7645	-1.2623	2616	1028
Evo.-unaware	-0.7012	75.00	-0.8437	-4.8926	2615	1482
Improvement	*0.7%*	*52.9%*	*9.4%*	*74.2%*	*-0.1%*	*30.6%*
Incremental:						
Evo.-aware	-0.6997	32.33	-0.8027	-1.5256	2611	1054
Evo.-unaware	-0.7013	59.33	-0.8161	-3.0479	2618	1316
Improvement	*0.2%*	*45.5%*	*1.6%*	*49.9%*	*0.3%*	*19.9%*
Feature Degree:						
Evo.-aware	-0.6996	39.00	-0.8098	-2.1185	2614	1033
Evo.-unaware	-0.7013	85.67	-0.8455	-5.2652	2616	1301
Improvement	*0.3%*	*54.5%*	*4.2%*	*59.8%*	*0.1%*	*20.5%*
Avg. Improvement	*0.4%*	*51.0%*	*5.1%*	*61.3%*	*0.1%*	*23.7%*

Table 4: Comparison of exploration strategies across evolution steps (RQ2)

The evolution-aware strategies consistently perform better than their evolution-unaware counterparts wrt. total reward (61.3% on average), time to threshold (51.0% on average), jumpstart (5.1% on average), and asymptotic performance (0.4% on average). With respect to the impact on QoS, the evolution-aware

strategies reduce the number of virtual machine manipulations by 23.7% on average, while keeping energy consumption around the same as the non-evolution-aware strategies. As can be seen, the evolution-unaware FM-guided strategies (from RQ1) may perform much worse than any of the evolution-aware ones. This is because they again explore old adaptation actions, many of which were not suitable. Finally, it can be observed that evolution-aware ϵ-greedy may even outperform the other evolution-aware strategies. This suggests that, during evolution, considering the changes of the adaptation space has a much larger effect than considering the structure of the adaptation space.

5 Discussion

Validity Risks. We used an actual cloud resource management service and a real-world workload trace to measure learning performance and the impact of the different exploration strategies on QoS characteristics. Still results are only for a single system, which limits generalizability.

We purposefully chose ϵ-greedy as a baseline, because it was the exploration strategy used in existing online RL approaches for self-adaptive services (see Sec. 6). Alternative exploration strategies were proposed in the field of machine learning. Examples include Boltzmann exploration, where actions with a higher expected reward (e.g., Q value) have a higher chance of being explored, or UCB action selection, where actions are favored that have been less frequently explored [31]. Another alternative is to use policy-based RL, which in contrast to value-based RL such as Q-Learning, directly represents the policy as a neural network, and thus intrinsically exhibits stochastic action selection behavior [24]. A comparison among those alternatives is beyond the scope of the current paper, because a fair comparison would require the careful variation and analysis of a range of many additional hyper-parameters.

We focused on evolution steps that increase the size of the adaptation space to assess to what extent our strategies are able to capture adaptation spaces of increasingly larger size. Our experiments may be complemented by analyzing how the different strategies compare to each other when the size of the adaptation space is reduced. Even though in an adaptation space of reduced size, fewer configurations have to be explored – thus leading in principle to faster learning – there still may be differences in the way these fewer configurations are explored.

Limitations and Assumptions. We assume that feature models are complete with respect to the coverage of the adaptation space and that during an evolution step they are always consistent and up to date. A further possible change during service evolution can be the modification of a feature's implementation, which is currently not visible in the feature models. Encoding such kind of modification thus could further improve our online learning strategies.

One aspect that impacts FM-guided exploration is the depth of the feature models. On the one hand, if the feature model has only few levels, the FM-guided exploration strategies behave very similar to random exploration, because such models do not provide enough structure. On the other hand, based on initial

experiments with the RL approach in [24], providing an RL agent with too structured knowledge might in fact hinder learning an optimal policy. How to define feature models at the right level of detail thus deserves further investigation.

In the realization of the exploration strategies (both ϵ-greedy and FM-guided), we assumed we can always switch from a configuration to any other possible configuration. We were not concerned with the technicalities of how to reconfigure the running service (which, for example, is addressed in [7]). We also did consider constraints concerning the order of adaptations. In practice, only certain paths may be permissible to reach a configuration from the current one. To consider such paths, online RL may be enhanced by building on work such as [27].

6 Related Work

The following authors applied online RL to self-adaptive services and considered different approaches to improve the performance of the learning process. Yet, they did not consider large adaptation spaces nor service evolution. Tesauro *et al.* use Q-Learning for autonomic resource allocation in data centers [33]. Xu *et al.* employ Q-Learning (with ϵ-greedy) for the automatic configuration of cloud virtual machines and applications [39]. Both suggest offline learning to increase the jumpstart at runtime. Barrett *et al.* propose using Q-Learning with ϵ-greedy for autonomic cloud resource allocation [3]. They propose parallel learning to speed up the learning process. Caporuscio *et al.* propose using two-layer hierarchical RL for multi-agent service assembly [6]. They observe that by sharing monitoring information, learning happens faster than when learning in isolation. Arabnejad *et al.* apply fuzzy RL with ϵ-greedy to learn fuzzy adaptation rules [2]. They also demonstrate that transfer learning may speed up learning [15]. Wang *et al.* use Q-Learning (using ϵ-greedy) together with function approximation. They use neural networks to generalize over unseen environment states and thereby facilitate learning in the presence of many environment states, *i.e.*, they address large state spaces but not large action spaces [38]. Moustafa and Zhang propose multi-agent Q-Learning with ϵ-greedy for adaptive service compositions [22]. To speed up learning, they use collaborative learning, where multiple systems simultaneously explore the set of concrete services to be composed. Zhao *et al.* propose using RL (using ϵ-greedy) combined with case-based reasoning to generate and update adaptation rules for web applications [40]. Their approach may take as long to converge as learning from scratch, but it may offer a higher jumpstart.

Bu *et al.* explicitly consider large adaptation spaces [4]. They employ Q-Learning (using ϵ-greedy) for self-configuring cloud virtual machines and applications. They reduce the size of the adaptation space by splitting it into coarse-grained sub-sets for each of which they find a representative adaptation action using the simplex method. Their experiments indicate that their approach indeed can speed up learning. Yet, they do not consider service evolution.

Dutreilh *et al.* explicitly consider service evolution [11]. They employ Q-Learning for autonomic cloud resource management and speed up learning by providing a good initial estimate for the Q-function, as well as by using statistical estimates about the environment behavior. They indicate that system evolution

may imply a change of system performance and sketch an idea on how to detect such drifts in system performance. Yet, they do not consider that evolution may also introduce or remove adaptation actions. As explained in Sec.1, such a change in the adaptation space cannot be determined by observing the environment, as the adaptation space is an intrinsic property of the RL agent.

In our previous work, we used online RL for a self-adaptive cloud service [24]. We addressed the problem of large environment spaces (similar to Wang *et al.*) but did neither consider large action spaces nor service evolution. In earlier work, we sketched the principal dependencies between learning and evolution, but did not provide concrete technical solutions [29].

A different line of work uses supervised machine learning to reduce the size of the adaptation space. As an example, Van Der Donckt *et al.* use deep learning to determine a representative and much smaller subset of the adaptation space [37]. Supervised learning requires sufficient labeled training data representative of the service's environment, which may be challenging due to design time uncertainty.

7 Conclusion

We introduced feature-model-guided exploration strategies for online reinforcement learning that address potentially large adaptation spaces and the change of the adaptation space due to service evolution. Experimental results for an adaptive cloud management service indicate a speed up of the learning process and an improvement of QoS characteristics.

As part of our future work, we will perform additional experiments, considering further types of services and the comparison with alternative exploration strategies. We also aim to integrate our strategies with more advanced reinforcement learning algorithms, such as policy-based reinforcement learning. In addition, we aim to address the current limitations of our strategies and will, for instance, also consider feature modifications during evolution.

Acknowledgments. We cordially thank Amir Molzam Sharifloo for constructive discussions during the conception of initial ideas, as well as Alexander Palm for his comments on earlier drafts. Our research received funding from the EU's Horizon 2020 R&I programme under grants 780351 (ENACT) and 871525 (FogProtect).

References

1. Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P.: Feature model differences. In: Proceedings of the 24th International Conference on Advanced Information Systems Engineering. pp. 629–645. CAiSE'12 (2012)
2. Arabnejad, H., Pahl, C., Jamshidi, P., Estrada, G.: A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In: 17th Intl Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017. pp. 64–73 (2017)
3. Barrett, E., Howley, E., Duggan, J.: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurrency and Computation: Practice and Experience 25(12), 1656–1674 (2013)

4. Bu, X., Rao, J., Xu, C.: Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. IEEE Trans. Parallel Distrib. Syst. 24(4), 681–690 (2013)
5. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Automated Software Engineering. 23(4), 687–733 (Dec 2016)
6. Caporuscio, M., D'Angelo, M., Grassi, V., Mirandola, R.: Reinforcement learning techniques for decentralized self-adaptive service assembly. In: 5th Eur. Conference on Service-Oriented and Cloud Computing, ESOCC'16. vol. 9846, pp. 53–68 (2016)
7. Chen, B., Peng, X., Yu, Y., Nuseibeh, B., Zhao, W.: Self-adaptation through incremental generative model transformations at runtime. In: 36th Intl Conf on Softw. Eng., ICSE '14. pp. 676–687 (2014)
8. Chen, T., Bahsoon, R.: Self-adaptive and online QoS modeling for cloud-based software services. IEEE Trans. Software Eng. 43(5), 453–475 (2017)
9. De Lemos, R., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Softw. Eng. for Self-Adaptive Systems II, LNCS, vol. 7475, pp. 1–32. Springer (2013)
10. D'Ippolito, N., Braberman, V.A., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: 36th Intl Conf. on Softw. Eng., ICSE '14. pp. 688–699 (2014)
11. Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., Truck, I.: Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In: 7th Intl Conf. on Autonomic and Autonomous Systems, ICAS'11. pp. 67–74 (2011)
12. Esfahani, N., Elkhodary, A., Malek, S.: A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems. IEEE Trans. Softw. Eng. 39(11), 1467–1493 (Nov 2013)
13. Filho, R.V.R., Porter, B.: Defining emergent software using continuous self-assembly, perception, and learning. TAAS 12(3), 16:1–16:25 (2017)
14. Hinchey, M., Park, S., Schmid, K.: Building dynamic software product lines. IEEE Computer 45(10), 22–26 (2012)
15. Jamshidi, P., Velez, M., Kästner, C., Siegmund, N., Kawthekar, P.: Transfer learning for improving model predictions in highly configurable software. In: 12th Intl Symposium on Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS '17. pp. 31–41 (2017)
16. Kinneer, C., Coker, Z., Wang, J., Garlan, D., Le Goues, C.: Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In: 13th Intl Symposium on Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS'18. pp. 40–50 (2018)
17. Mann, Z.Á.: Interplay of virtual machine selection and virtual machine placement. In: 5th European Conf. on Service-Oriented and Cloud Computing, ESOCC'16. vol. 9846, pp. 137–151 (2016)
18. Mann, Z.Á.: Resource optimization across the cloud stack. IEEE Transactions on Parallel and Distributed Systems 29(1), 169–182 (2018)
19. Metzger, A., Bayer, A., Doyle, D., Molzam Sharifloo, A., Pohl, K., Wessling, F.: Coordinated run-time adaptation of variability-intensive systems: An application in cloud computing. In: 1st Int'l Workshop on Variability and Complexity in Software Design, VACE'16 (2016)
20. Metzger, A., Di Nitto, E.: Addressing highly dynamic changes in service-oriented systems: Towards agile evolution and adaptation. In: Agile and Lean Service-Oriented Development: Foundations, Theory and Practice. pp. 33–46 (2012)

21. Metzger, A., Pohl, K.: Software product line engineering and variability management: Achievements and challenges. In: Future of Software Engineering, FOSE'14. pp. 70–84 (2014)
22. Moustafa, A., Zhang, M.: Learning efficient compositions for QoS-aware service provisioning. In: IEEE Intl Conf. on Web Services, ICWS'14. pp. 185–192 (2014)
23. Nachum, O., Norouzi, M., Xu, K., Schuurmans, D.: Bridging the gap between value and policy based reinforcement learning. In: Advances in Neural Inform. Proc. Systems 12 (NIPS 2017). pp. 2772–2782 (2017)
24. Palm, A., Metzger, A., Pohl, K.: Online reinforcement learning for self-adaptive information systems. In: Yu, E., Dustdar, S. (eds.) Int'l Conference on Advanced Information Systems Engineering, CAiSE'20 (2020)
25. Papazoglou, M.P.: The challenges of service evolution. In: 20th Int'l Conf. on Advanced Inform. Systems Eng., CAiSE'08. pp. 1–15 (2008)
26. Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R.Y., Chen, X., Asfour, T., Abbeel, P., Andrychowicz, M.: Parameter space noise for exploration. In: 6th Intl Conf. on Learning Representations, ICLR 2018. OpenReview.net (2018)
27. Ramirez, A.J., Cheng, B.H.C., McKinley, P.K., Beckmann, B.E.: Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In: 7th Intl Conf. on Autonomic Computing, ICAC'10. pp. 225–234 (2010)
28. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009)
29. Sharifloo, A.M., Metzger, A., Quinton, C., Baresi, L., Pohl, K.: Learning and evolution in dynamic software product lines. In: 11th Intl Symposium on Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS'16. pp. 158–164 (2016)
30. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence Models for Highly Configurable Systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 284–294. ESEC/FSE 2015, ACM, New York, NY, USA (2015)
31. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA, 2nd edn. (2018)
32. Taylor, M.E., Stone, P.: Transfer learning for reinforcement learning domains: A survey. J. Mach. Learn. Res. 10, 1633–1685 (2009)
33. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Computing 10(3), 287–299 (2007)
34. Thüm, T., Batory, D., Kastner, C.: Reasoning About Edits to Feature Models. In: 31st Intl Conf. on Softw. Eng., ICSE'09. pp. 254–264 (2009)
35. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85 (Jan 2014)
36. Thüm, T., Kästner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In: 15th Intl Conf. on Software Product Lines, SPLC'11. pp. 191–200 (2011)
37. Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., Michiels, S.: Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In: 15th Intl Symp. on Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS 2020. ACM (2020)
38. Wang, H., Gu, M., Yu, Q., Fei, H., Li, J., Tao, Y.: Large-scale and adaptive service composition using deep reinforcement learning. In: 15th Intl Conference on Service-Oriented Computing (ICSOC'17). pp. 383–391 (2017)

39. Xu, C., Rao, J., Bu, X.: URL: A unified reinforcement learning approach for autonomic cloud management. J. Parallel Distrib. Comput. 72(2), 95–105 (2012)
40. Zhao, T., Zhang, W., Zhao, H., Jin, Z.: A reinforcement learning-based framework for the generation and evolution of adaptation rules. In: Intl Conf. on Autonomic Computing, ICAC'17. pp. 103–112 (2017)