Multi-Agent Reinforcement Learning for Robust and Scalable Failure Root-Cause Analysis

Christopher Aust*
Hasso-Plattner Institute
University of Potsdam
Germany
christopher.aust@student.hpi.uni-potsdam.de

Ulrike Bath
Hasso-Plattner Institute
University of Potsdam
Germany
ulrike.bath@student.hpi.uni-potsdam.de

Florence Boettger
Hasso-Plattner Institute
University of Potsdam
Germany
florence.boettger@student.hpi.uni-potsdam.de

Jonas Krah
Hasso-Plattner Institute
University of Potsdam
Germany
jonas.krah@student.hpi.uni-potsdam.de

Christian M. Adriano†
Hasso-Plattner Institute
University of Potsdam
Germany
christian.adriano@hpi.de

Holger Giese‡
Hasso-Plattner Institute
University of Potsdam
Germany
holger.giese@hpi.de

ABSTRACT

Diffuse events like pandemics, wars, or natural disasters inflict local changes that can quickly spread globally via human behavior contagion, e.g., the hoarding of toilet papers (Covid-19) and, more recently, sunflower oil (invasion of Ukraine). These events affect how people utilize various distributed systems that operate from medical care and personal finance to travel and e-commerce platforms. As these systems rely more extensively on machine learning prediction models, changes in their usage correspond to unplanned distribution shifts, also called out-distribution data (OOD), that can quickly degrade the prediction accuracy of these built-in models and cause havoc to the respective systems and users. We investigated this OOD phenomenon and their mitigation challenges in the realm of machine-learning-based self-adaptive systems. In particular, we focused on how to endow these systems with robustness capabilities under partially observable system states, for instance, how to detect failures whose root-causes are hidden because of uncertain patterns of propagation. Our approach combined two strategies: (1) vertical separation-of-concerns among monitoring, analysis, planning, and execution failure fix actions (via a two-layered multi-agent reinforcement learning architecture) and (2) horizontal divide-and-conquer of the large state-space (via clustering among autonomous agents a set of shops of a multi-tenant e-commerce platform). This way we aimed to isolate the effects of distribution shifts, while providing enough time for the system to counteract further losses of prediction accuracy. For that, we relied on transfer learning among agents and distributing training and execution of shops per agent clusters. We evaluated the approach via a series of controlled experiments on the effect of perturbations across levels of failure pattern complexities and the ratio of shops per agent. The results showed that we could achieve certain robustness guarantees with respect to (1) the convergence towards a minimal number of fixes per failure pattern (successful repair actions), (2) the convergence of the average regret (1 - predicted probability of correct action), and (3) the stability of the corresponding policies (constrained outliers, i.e., number of incorrect actions after point of convergence).

1 INTRODUCTION

Deep neural networks have revolutionized the field of reinforcement learning and achieved remarkable success in many areas [38, 50]. However, industry surveys report that from 55% to 72% of companies are not being able to deploy AI systems [24].

Current systems cannot adapt to more complex and evolving realities. Realities that play like strong adversaries against these AI systems. The inability to adapt is a problem of lack of robustness in these AI Systems [62].

Studies on Markov Decision Process and reinforcement learning [55] mostly rely on a consistent, barely changing environment which is mostly incompatible in practice. Thus, robust reinforcement learning became more and more important and research already achieved improvements in performance [29].

Instead of tackling the problem of robustness with a general distribution shift in data, we take a closer look at how data may behave and adjust the system accordingly. Models in practice learn from data that is directly linked to real-world conditions in such a way that drastic changes in behavior can lead to such distribution shifts, also called out-of-distribution data (OOD). Diffuse events like pandemics, wars, or natural disasters inflict local changes that can quickly spread globally via human behavior contagion [45], e.g., the sudden demand for toilet papers or face masks (Covid-19) which peaked first in Italy, followed by Spain, France, Canada, and the US. Different events might also trigger similar behavior, e.g., panic buying of sunflower oil more recently (invasion of Ukraine). These events affect how people utilize various distributed systems that operate from medical care and personal finance to travel and e-commerce platforms [17]. As these systems rely more extensively on machine learning prediction models, changes in their usage correspond to unplanned OOD that can quickly degrade the prediction

*Four first authors in alphabetical order. All authors contributed equally to this research.
†Research supervisor
‡Department head

accuracy of these built-in models and cause havoc to the respective systems and users.

We investigated this OOD phenomenon and their mitigation challenges in the realm of machine-learning-based self-adaptive systems. In particular, we focused on how to endow these systems with robustness capabilities under partially observable system states, for instance, how to detect failures whose root-causes are hidden because of uncertain patterns of propagation.

Our approach combines two strategies: (1) vertical separation-of-concerns among monitoring, analysis, planning, and execution failure fix actions by implementing a two-layered multi-agent reinforcement learning architecture and (2) horizontal divide-and-conquer of the large state-space via clustering among autonomous agents a set of shops of a multi-tenant e-commerce platform. This way, we aimed to isolate the effects of distribution shifts, while providing enough time for the system to counteract further losses of prediction accuracy.

Our contributions comprise (1) a prototypical method to successfully tackle real-world robustness challenges involving behaviorally influenced environments and (2) its scalability. The implications of our results are promising in a sense that society could be better supported by more robust systems. After all, the pandemic revealed how intertwined our lives with AI systems are by now. Changes in behavior influence how these systems work which in turn also influences our behavior. To allow the reproduction of our results, we made the procedures, models, and data publicly available to the community [1].

We structured the paper as follows. The next section provides an overview of the recent advancements in the field. In section 3 we explain the preliminaries that are the foundations of our approach including the example scenario. Our approach is then detailed in section 4, followed by the results and discussion of our system in section 5. In section 6, we review the threats to validity and give further extension possibilities in section 7. Finally, in section 8, we summarize our contributions and future work.

2 STATE OF THE ART

2.1 Reinforcement Learning

Reinforcement learning (RL) is a machine learning method where an agent independently learns a strategy to minimize the cumulative regret by choosing action according to their immediate and discounted cumulative rewards. This way, the agent learns independently in which situation which action is the best. Unlike supervised learning, RL does not require data up front. Instead, the database is formed through extensive trial-and-error runs within a simulation scenario [55]. The most popular state-of-the-art RL approaches will be summarized in the following.

The Deep Q-Learning (DQN) is a simple and very commonly used model. The network is trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights [39]. An approximate approach can be realized with the Proximal Policy Optimization (PPO) model, an algorithm with high data efficiency and a reliable performance while using only first-order optimization [46]. Based on the DQN, a Deep DQN (DDQN) [34] decouples the estimation and instead uses an actor-critic architecture. A DQN can tend to select overestimated values which is solved in an DDQN. In our project, we also decided to implement an actor-critic approach for more sample-efficiency. The chosen architecture will be explained in more detail in the following.

Advantage Actor-Critic. The Advantage Actor-Critic (A2C) is one of many algorithms based on the policy gradient theorem. The theorem implies learning a policy for selecting actions without the need for a value function. However, especially for the actor-critic, a value function can still be used in the learning process of the policy parameters [55]. The actor-critic includes two neural networks. The actor represents the policy that is used for determining an action based on a probability distribution given the current state of an environment. The critic's neural network, however, is the value function corresponding to the current state that is pushing the actor towards the optimal action selection. As described by Konda and Tsitsiklis [30] the actor-critic method is having a faster convergence compared to actor-only methods, as the variance is reduced.

2.2 Robust Reinforcement Learning

Even though of the breakthroughs and achievements in RL [39, 51], the adaptation to real-world problems still raises challenges. Real-world systems can suffer from latencies, noise, non-stationarity and distribution shifts over time. These usually fall short in a perfect simulated environment where a state is fully transparent to the agent. A summary of current challenges regarding real-world adaption can be found in [13]. Generalization in RL focuses on overcoming these problems by improving policies with the desired robustness, transfer, and adaptation properties. As generalization itself is a broad topic, improvements in one context could harm generalization in others. Kirk et al. [29] compare present solutions for single-agent settings, this also includes single-agent settings with multiple agents to increase the diversity of the environment. In this work, we focus on the following challenges of robust RL: partial observability, distribution shifts, and scalability.

Partial Observability. Most relevant real systems for RL are partially observable, e.g. malfunctioning sensors might pollute observations or external influences can result in noise. These problems can be formulated as *Partially Observable Markov Decision Process* where the agent's observation is detached from the state. Hausknecht et al. [23] use recurrent neural networks which enable the agent to track and recover hidden states, while Ghost et al. [21] use Bayesian RL techniques to compensate the missing information.

Distribution Shifts. The real-world environment is also in constant change. Hence, in Deep RL, a neural network is trained on a stream of data whose distribution shifts over time. Based on this premise, fine-tuning on new cases would disrupt previously acquired knowledge resulting in the infamous problem of Catastrophic Forgetting, where new tasks are likely to override weights learned for past tasks and, thus, decrease the model's performance for those tasks. Current approaches address this problem in various aspects. In [7], simple improvements to the replay buffer are shown which result in better robustness with only a slight increase in computational requirements. Other effective solutions are based on multi-agent approaches, e.g. [43] where different tasks are delegated between neural networks as well as transfer-learning solutions [12] where experience obtained in learning to perform one task may also improve the performance in a related but distinct task. Even

though this architecture has a much higher storing cost because of the higher number of models, it also confers robustness and scalability to the solution.

Scalability. RL algorithms are known to require orders of magnitude more data than pure deep learning models [4]. The distribution shift challenge worsens this problem because previous collected data becomes invalid for training, hence curtailing the data-efficiency provided by solutions like experience replay [16, 35], model-based RL [40, 41], and batch/pretrained RL [48, 54].

2.3 Multi-Agent Reinforcement Learning

In multi-agent reinforcement learning (MARL), multiple agents are interacting with a shared environment and are attempting to maximize their reward. As described by Gronauer and Diepold [12], the multi-agent setting is a broad area with several subfields. First, different training scenarios can arise because of a centralized or distributed composition. The same applies to the execution of the agents that can be centralized or decentralized. There are several taxonomies for a multi-agent setting that depends either on the information available for each agent or on the reward structure. The reward structure decides whether an agent is cooperative, competitive, or neither. Irrespective of reward structure, new challenges arise due to the growing complexity of handling multiple agents compared to single RL. Gronauer and Diepold [12] have summed up these challenges. The non-stationarity emerges due to different learning and changing agents. Their actions have to be coordinated and the reward for each agent's action has to be assigned correctly. Note that the agents could also be subject to partial observability that needs to be considered while learning. This leads to a growing computational effort, especially when the number of agents needs to be scaled.

This project achieved a more robust system by adopting a MARL with distributed learning on two levels: cooperative agents working within a partially observable state-space and coordinated by controller that is responsible for directing events (failures), sharing reward information (actual utility), and consolidating individual actions into a unified decision making process (ranking).

2.4 Root-Cause Analysis of Failure Propagation

Failure propagation is frequent phenomenon in software and hardware systems, but it is critical in systems with complex and stochastic dependencies, for instance, component-based architectures [59], cloud-based microservices [53], and cyber-physical system [36]. The complexity and stochasticity of the dependencies pose challenges for root-cause analysis (RCA) because the identification of the failing unit can involve a combinatorial set of possibilities. Because many of these systems are responsible for safety-critical functionalities, the problem goes beyond avoiding the cost of brute-force search, one should be able to repair the system as quickly as possible.

Various mechanisms were studied on how to expedite the identification of a failure, for instance, time series-based [25, 42] graph traversal-based [53, 60], causal inference-based [25, 37], and deep learning methods [15, 44].

Graph-based techniques are powerful, but pose challenges to scalability. Therefore, we decided to first investigate an approach that does not rely on any prior knowledge of the structure of the graph and also does not learn that structure. Instead, it only learns how to map a pattern of node combinations to a heat map of failing probabilities. Because the heat map covers the entire state space of nodes, the node with the highest probability is the candidate to be the root-cause of the failure. Each failure being represented by a finite trace (list of nodes, i.e., software components).

3 SCENARIO

This section describes our setting, initial assumptions and introduces some architectural orientations. In subsection 3.1, the ground phenomenon of propagating failures will be defined. Following in subsection 3.2 is the scenario with the problem statement which we want to solve with our system. Afterwards, unsupervised learning approaches will be presented in subsection 3.3 that are the basis for our implementation.

3.1 Phenomenon

Failures or malfunctioning components can happen in any system that is monitored and subject to unscheduled repairs. However, different systems can present unknown components' interdependencies which result in various failure propagating patterns. The so-called ripple effect occurs when an initial disturbance to a system propagates outward to disturb an increasingly larger portion of the system. Next we describe our set of assumptions about a system with failing components.

Failure Propagation Assumption. Any real failure should produce at least one visible event, either an observation that matches the failure location or an observation that results from a failure propagation. Thus, a system without failures does not need to be fixed and can be ignored. This implies that we did not handle the failure masking situations, in which multiple failures occlude on another. Insection 7 we will address a possible extension named "perfect failure masking".

Failure Type Assumptions. Failures are events that can either disable a component (cease to respond to requests) or cause the component to under-perform, e.g., increase response time in servicing requests from other components. Since we are focusing on a more robust architecture first, we ignore the different failure types and instead improve the performance of finding the failing component in general.

Failure Fixes Assumptions. Fixes might either involve restoring a component to a state before a failure event or to change the state of the component in order to improve its performance, for example, add instances to a component that is overloaded with request.

Overhead and Stability Phenomena. Note that one should not add more component instances than necessary because that would add an overhead cost that would reduce the overall utility. Therefore, there is a concern that actions to fix (add or remove instances) might lead to overshooting (too many instances). On the other hand, too few instances might cause undershooting, which can become a source of instability when there are large and rapid fluctuations in the number of requests. In these cases, it might be challenging to determine how many instances on average are necessary to cope with the varying number of requests.

3.2 *mRUBiS*

To replicate a complex and evolving reality, we adopted an e-commerce platform for multi-tenant online shops. This platform is a framework for simulating self-adaptation behavior like self-optimization and self-healing. All shops consist of 18 types of components in the default architecture setting. Each shop is autonomous in a sense that its components do not depend on external components and can have locally customized configurations. Each component holds information about its state and its current utility. The utility of a shop is the sum of the utilities of its components [56]. Failures can be injected into any component of the shop. As explained in 3.1, different failures types can be injected in *mRUBiS* and corresponding actions can be undertaken to fix them and increase the shop's utility. The overall state of *mRUBiS* corresponds to the current failure types of all the components and the sum of current utilities of all shops. This state is partially shared with the agents, which in case of failures, must select for each failing component the appropriate repair (fix) action to take from a list of available actions.

3.3 Unsupervised Learning Approaches

For the realization of a robust system, we have implemented the following approaches in self-adapting and self-managing.

Self-Adaptive Systems. A Self-Adaptive System is an approach to achieve robustness to changes that are only completely known at runtime. [32] Andersson et al. [3] are describing the behavior of a self-adaptive system as the shift of development and change activities during the development time to run-time and, thus, the reassigning of responsibility for these activities to the system. As a result, the development process is changing and cannot be decoupled from the running system anymore. This behavior can be implemented by an architecture-based adaption with a system using a feedback loop to realize this goal [20]. Andersson et al; [3] is describing this more precisely as a *Managing Subsystem* that implements the *Adaptation Logic*. This logic in turn manages a *Managed Subsystem* which implements the *Domain Logic*. This adaption logic can be realized by following the MAPE-K approach described in the following paragraph.

MAPE-K. The architecture was first described by Kephart and Chess to solve the problem of automatically managing complex systems [28]. The intention of this approach is to manage a system by following the administrator's goal without the need for external input. The self-management is achieved by the autonomic manager that is executing four consecutive functions by sharing a knowledge base. Sinreich is referring to this concept as an intelligent control loop and describing it as follows [52]: The first function of this loop *monitors* the managed element to gather information such as metrics. The *analysis* function is using that gathered information to make predictions for further action, thus helping the automatic manager to gain information about the IT environment. The *planning* is generating a specific action plan to achieve the defined goal and objectives. As the last step, this generated plan is *executed* in a controlled manner to apply updates. All those functions are accessing shared knowledge to fulfill their particular goal.

4 APPROACH

Our approach combines explorations of multiple ideas. In subsection 4.1 we describe how we adjusted *mRUBiS* with the resulting communication explained in subsection 4.2. Afterwards, we introduce an actor-critic agent in subsection 4.3, integrated in our robust multi-agent architecture in subsection 4.4.

4.1 *mRUBiS* Changes

We have optimized some *mRUBiS* processes as well as adjusted some settings to fit to our scenario. These will be explained below.

Observation. A state in *mRUBiS* is originally equivalent to its observation, meaning the agent has exact information about which components fail, and the available actions, and will choose an available fixing action. We decoupled the real state from the observation. As described in 3.1. In our scenario, the agent will not choose from a list of available actions for the failing component but will instead be presented with a complete observation of all the shops and must decide whether it applies some fix to a component or not.

Failure Propagation. Originally, a failure only affected the component it was injected into. However, it is more realistic that a failure in a component has consequences on connected components. None of the components exist in a vacuum, so it is only reasonable that they would affect one another. To implement this, we added failure propagation. We set a baseline probability of p for a failure to propagate. With a probability of p, the failure will propagate to a connected component, given by a Markov Chain. If the failure does not propagate, the propagation is finished. The propagated failure then has a probability of p to further propagate, repeating the process. Propagated failures impact utility as though they were normal failures, but in order to fix them, the root cause needs to be fixed. As such, agents need to learn the structure of the components to distinguish root causes from propagated failures. p can be set by the *Python* side and defaults to $p = 0.5$. Additionally, we also implemented options for setting the length of the failure propagation trace to a fixed length l, as well as defining the exact trace that is to be created.

Global State. Originally, *mRUBiS* calculated utility and other aspects of components only when necessary, without meaningfully saving them. As such, many of the same values had to be frequently recalculated, and it was difficult to change the program flow because many relevant pieces of data were only available during certain parts of it and then discarded. We changed this by creating one all-encompassing state dictionary, which includes the data for all shops, their components, and the components' attributes. We can then change only the aspects of this dictionary which need changing, and send the whole dictionary to the *Python* side. As a result, all data is available at all times in an easily accessible location, and it can be changed as necessary.

Failure Injection. Failure Injection was initially implemented as a deterministic approach, where, for episode t, we take $t' = t \mod 10$, and select one of 10 fixed injections based on t'. This process was hard-coded around 10 shops, so it would only ever inject into those 10 shops. This worked if there were exactly 10 shops, but if you made any changes to that number, issues would arise: If you add more shops, then any shops after the 10th would never receive issues; if you remove shops, then there would be

injections into nonexistent shops. Since we wish to test scalability of our algorithms, it was necessary to expand the number of shops. As such, we created our own failure injection algorithm. Instead of being hard-coded, we first use a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ to determine the number of shops we will inject into. The mean μ and variance σ^2 of said distribution are defined from *Python*. We then randomly select a number of shops equal to the result of the Gaussian to inject into, and for each selected shop, we inject into a random component. Additionally, to keep this process deterministic (and therefore reproducible), we use the current episode t as the seed for any random operation. As such, the same episode number will always yield the same injections. We also initially tested a version of this where we used $t' = t \mod 10$ as the seed, to keep the modulus-10 nature of the original injection mechanism, but ultimately abandoned that idea, as the repetition was merely a limitation of the old mechanism, and not a feature that we wish to reproduce.

Multi-Agent Adaptions. As the *mRUBiS* framework originally only is thought to be used with a single agent, some adaptions had to be taken to accommodate the usage of multiple agents. Originally, it was guaranteed that the agent will always have a fix for the failing component, as it chooses from a list of available actions. With multiple agents, it can happen that one or more will not find the correct component to fix. Therefore an intermediary step was introduced that checks whether the chosen fixes are correct, and only if so, will send them to *mRUBiS* .

4.2 Communication

We use a socket to communicate between *mRUBiS* and the agents. Previously, *mRUBiS* and Python would send messages on a per-shop or a per-component basis, i.e., one communication would consist of only the relevant messages for a component. The reason for this was that the socket has a character limit of 8192 characters for *mRUBiS* sending messages to *Python*, and this splitting of communication was used to circumvent it.

However, this method was not the most effective. Since the full limit of 8192 characters was barely reached under this approach (many more messages than needed to be were sent). Additionally, in order to avoid this limit, they often only communicated the changes.

We adapted the existing communication by using a chunked communicator, as seen in Figure 1. If the message length is less than the limit of 8192, the message is sent normally. Otherwise, the message is split into $n := \left\lceil \frac{l}{8192} \right\rceil$ chunks, where l is the message length, with the first $n-1$ chunks being of length exactly 8192, and the last chunk containing the remaining message. First, *mRUBiS* sends n to the *Python* side, which also lets *Python* know that it will be receiving n chunks. Then, it sends one chunk after the other, before finally sending the information that it had finished. Then, *Python* concatenates the chunk it received back into the original message.

This method of communication allows us to be more efficient than previously, since we communicate the information with as few messages as possible. Additionally, by allowing us to send the entire state instead of only the changes, we can simplify a lot of the operations that would normally be used to modify the state.

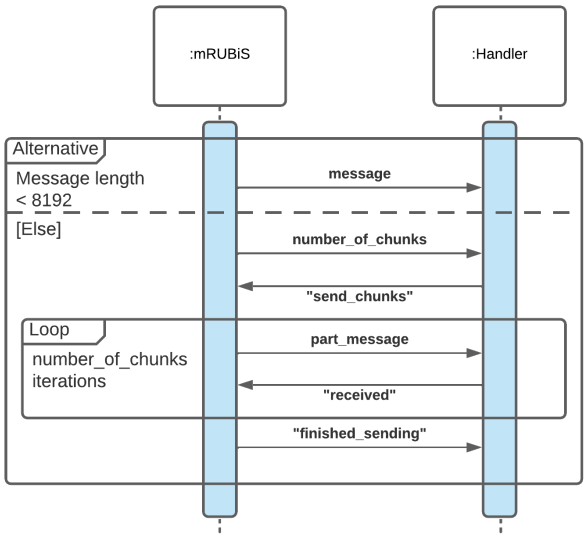


Figure 1: UML Sequence diagram of the new communication workflow

By creating one state that we only need to send back and forth once per run, we can dramatically simplify the communication process. Instead of many messages, we can simply have *mRUBiS* send the global state, and then for the agent handler to send back the chosen actions. One can see a somewhat simplified view of this procedure in Figure 2. As can be seen, during each run, only one message is sent back and forth, with confirmation messages in-between. The global state is then split into the parts relevant to each agent, sent to corresponding agent, the agents reply with their sub-actions, and the Rank Learner sorts these actions before passing them back to *mRUBiS* . This synchronized approach makes the communication process more easily understandable and intuitive, as well as allowing for easier changes to it.

4.3 Actor-Critic Agent

Several shops are assigned to an agent that will be fixed independently of each other. For each failing shop, the agent predicts the failing root-cause that needs to be fixed. The agent shown in Figure 3 is implemented by following the A2c approach, described in section 2.1, to use the benefits of a faster convergence. The actor outputs a probability distribution over all possible components. The critic's value function had thereby influence on the actor's training. The prediction of the failing component is achieved by using a $TD(0)$ approach as the network needs only one step to get immediate feedback from *mRUBiS* . The input for both networks is the current state of each component within one shop as a one-hot encoding. The actor's probability distribution is ranked within the shop saved as a memory until the shop is fixed.

In each step, the next most certain component is chosen. With this memory, we prevent the execution of the same fix multiple times. After selecting each shop's component, they are ranked

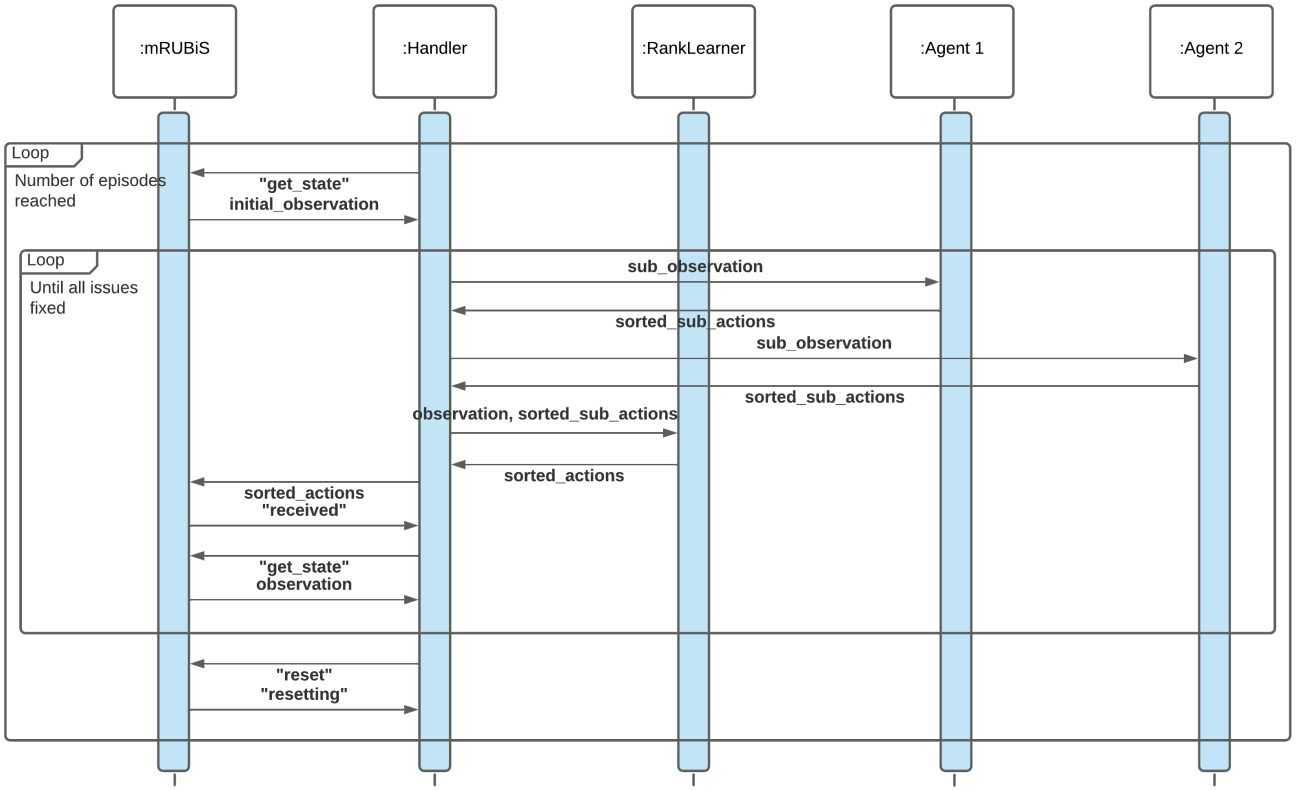


Figure 2: UML Sequence Diagram of the workflow between *mRUBiS* and *Python*

the shops by using the predicted utility of the most likely failing component of each shop that needs to be fixed by *mRUBiS* .

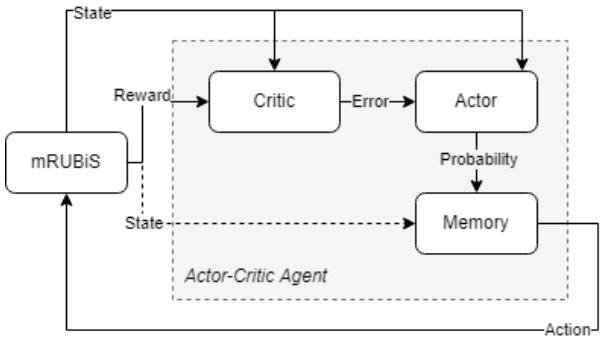


Figure 3: Actor-Critic Agent

4.4 Robust Multi-Agent Architecture

Hierarchical Learning Structure. With the divide-and-conquer principle, we implemented a coordinator class `MultiAgentController` (MAC) shown in Figure 5. The MAC organizes observations, actions, and rewards and forwards them to the agents according to a specific shop distribution. Only this coordinator is used to interact with the agents and acts as a mediator between the environment. We access the *mRUBiS* environment through a *Gym* interface which makes it easier to separate the agent setup and the *mRUBiS* environment.

Inside the MAC, a hierarchical setup is splitting up the learning process. Foremost, the single agent is trying to learn the prediction of the failing component using a basic Keras actor-critic implementation. As a next step, the rank learner is using the output of a ridge regression method that is predicting the utility for a component. All failing components are ranked by using the predicted utility for achieving the best possible utility gain. Instead of having a complex network that is predicting a sorted list of failing components and that can hardly scale and react to changes, we decided to have a hierarchical cooperative architecture. Thus, the complexity of our problem is reduced. Therefore, we split the learning into two steps: The learning of the actor-critic networks (L1) and the learning of the ridge regression model to rank the different probability distributions (L2). This architecture can handle a scalable number of networks that can react to changes of the underlying *mRUBiS* configuration.

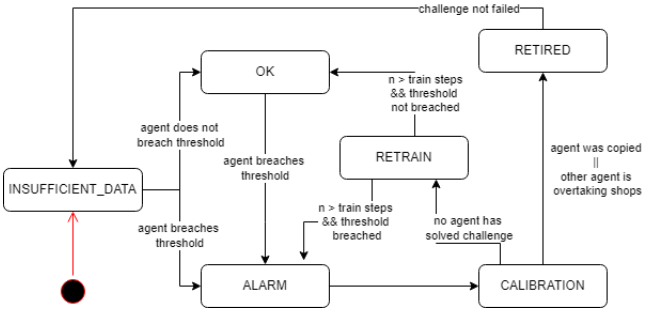


Figure 4: Robustness State Diagram

Robustness. The robustness is introduced by following the described MAPE-K approach that is following the state diagram shown in Figure 4. In our scenario, the robustness component constantly analyzes the history of each agent's loss of the predictions. As both networks of the actor and the critic are sharing some layers and the actor's network is used to choose the failing component, it is the actor's custom loss function that determines whether a prediction was good or not. The defined threshold for the loss is checked for the last n episodes. Only if m breaches are detected, the status for this agent will be changed accordingly. After the analysis step is done, the planning will prepare an execution plan for all agents in an alarm state. Here we do plausibility checks to reduce the number of execution plans needed to find a better performing agent. Only agents are considered that did not make the same last mistakes as the failing agent. The challenge is required to rank all healthy and retired agents. For this, the last observation that led to the wrong prediction is used to find the regret for each agent under the assumption the agent is doing a correct prediction. If no agent is either healthy or is passing the plausibility check, the failing agent is copied and retired to be available for future challenges. At the same time, the original failing agent retrains n episodes after it is analyzed again by the robustness component. As soon as the analysis is done and any execution plan is available, it is executed by temporary moving shops to another agent that is ranked the best in the list of execution plans. If this assisting agent is performing well, i.e. it is predicting most of the shops correctly, it will overtake the shops of the failing agent. Otherwise, the next execution plan is tested until no more plans are available. An agent with no more shops consequently retires to be available for future challenges of other agents, e.g. if a rollback is happening in the environment.

5 EVALUATION

To see how well an agent is performing, we are using the following metrics:

Tries represents the number of steps needed for an agent to fix a shop. If an agent has multiple shops, the averaged is taken. Since an agent chooses a component only once in each failure episode, the maximum number of tries is 18.

Regret is a more detailed metric to show how certain an agent when choosing the correct component. After each episode, the regret is calculated as $1-P(\text{failing component})$ from the probability distribution of the actor.

In our experiments, we focused on our goals regarding robustness, shown in subsection 5.1 and 5.2, and scalability in subsection 5.3.

5.1 MAPE-K Robustness

As explained in the introduction/scenario, we are interested to see whether we can improve the robustness of *mRUBiS* by applying an architecture-based adaption using the MAPE-K approach. To recreate the "wave effect", we have set up the following scenario in two phases:

(1) A failure graph, see represents how failures propagate between components. Two agents were pre-trained for 1000 episodes on different failure graphs, see Figure 6 graph 1 and 2 respectively. Figure 7a shows that both agents learn to reduce the regret and converge. The convergence is more clear when we consider the number of tries it takes to find the correct fix shown in Figure 7b. Both agents can find the correct fix generally in one attempt in less than 200 episodes of training.

(2) Now we assume an existing system with agents that can find the failing component that *mRUBiS* needs to fix. Different agents in this deployed system represent different failure graphs to be able to fix different states of *mRUBiS* . Following the "wave effect", a new rollout of a different failure graph, in this case fig b, will be simulated by *mRUBiS* for the whole system. *Agent 2* was already confronted with this change and, thus, is pre-trained with this failure graph, whereas the other was pre-trained with a slightly different failure graph. As one of these agents is confronted with a change in the system, the feedback loop should detect this change and start developing a plan to prevent a considerable loss in performance.

First, the previously described scenario was executed with and without the robustness feature as a baseline, see Figure 8. Therefore both pre-trained agents were run against *mRUBiS* using a second failure graph, whereas *Agent 1* was pre-trained with another failure graph. For the baseline, the result can be seen in Figure 8a. As can be seen in the figure, *Agent 2* that was previously trained on the same *mRUBiS* setup continues to learn where it stopped. Consequently, a convergence was not reached after 1000 episodes as it continues the next 1000 episodes. Contrary to that, the regret of *Agent 1* does not continue the previous curve, but starts again with the highest possible regret. The plot shows that *Agent 1* has retrained its policy to reduce the loss. It takes almost 200 episodes again to reach the same level that was reached after 1000 episodes of pre-training. This emphasizes that retraining an agent for a new failure graph is as costly as waiting for 200 episodes, but still performance out training from scratch.

Having this baseline, we executed the same setup with the robustness feature activated, see Figure 8b. With the robustness feature activated, shops are moved between agents. If another agent is performing better, the previously owning agent retires and is not producing any metrics. Thus, you see the average over all shops regardless of their corresponding agent instead of having a plot for the regret of both agents. The robustness function has successfully moved the shops from *Agent 1* to *Agent 2* which can be seen in the chart as there is no peak in the beginning.

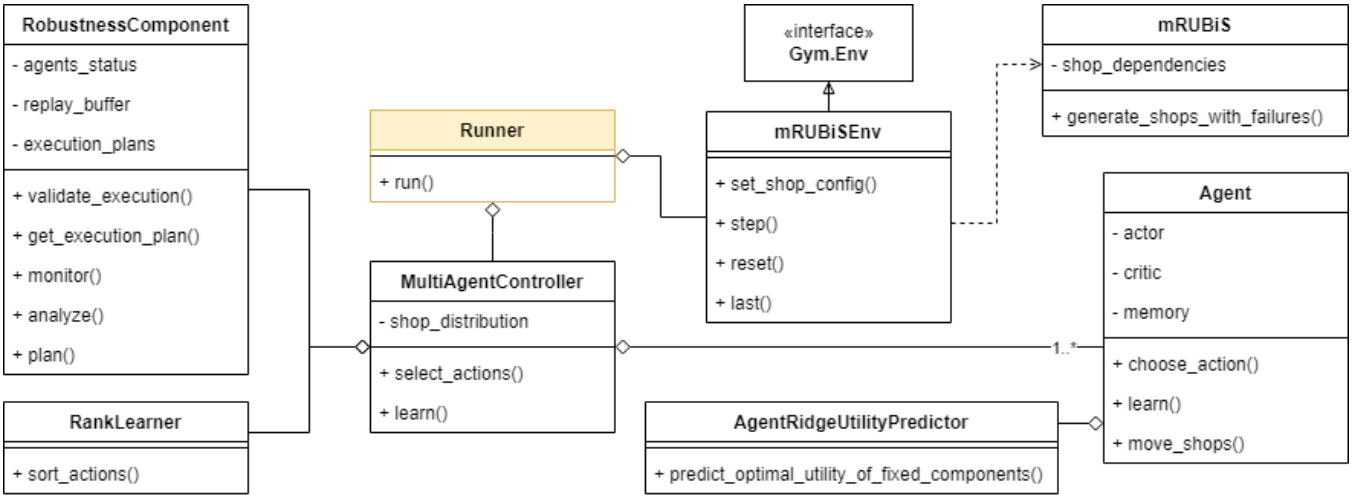


Figure 5: Robust Multi-Agent Architectural Diagram

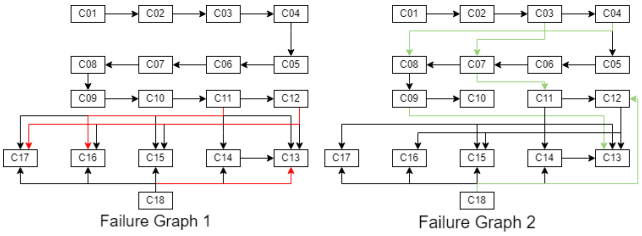


Figure 6: The failure graph displaying components' failure propagation. In the second graph, the red arrows were deleted and the green arrows were added.

The peak after 600 episodes might be due to a previously unknown path as the trace distribution is extensive. Since the robustness threshold were not breached longer than the specified interval, no action was taken. However on the contrary, since *Agent 1* does not perform better, *Agent 2* would have gone into the retrain state.

The results show that a feedback-loop is reducing the loss drastically and can make the system robust to changes. However, we assume that changes propagate in waves due to the behavioral contagion effect. Therefore at least one agent has encountered this change before it affects other agents and, thus, went into retraining. Nevertheless, this approach is a stepping stone for further research regarding robustness in real-world systems. The suggested MAPE-K implementation is one possible solution that increases the robustness especially for the behavioral contagion effect. Further adaptions and changes can be made depending on the requirements of the system applied by the suggested design principles from Brun et al. [6].

Improvements and next possible steps will be discussed in section 7.

5.2 Failure Propagation Robustness

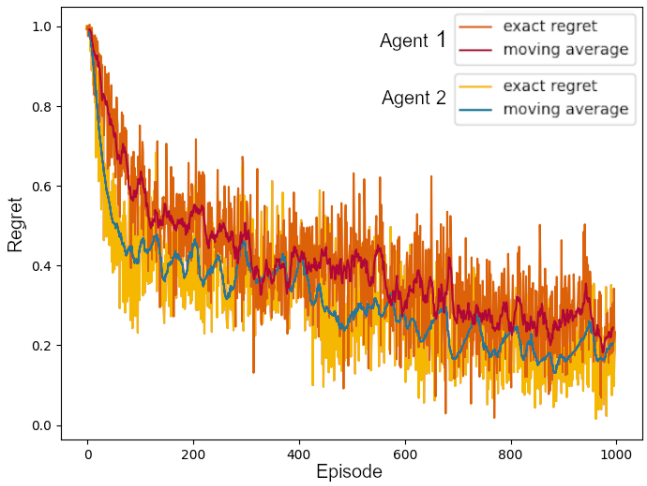
Experimental Design. To further investigate the robustness of the learning, we want to research how the agents perform with different levels of failure propagation at the moment of training and afterwards.

Therefore in a first step we compare the convergence curves of agents that are trained on failure traces with a specific length. Because for this experiment the number of shops that an agent has is irrelevant, it was chosen to work with a single shop for each agent. The agents were trained for a period of 2000 episodes. The failure traces we looked at varied from a single failing component to a trace that includes 5 components. In each of those traces only the first component is the actual failing component the agent needs to find. The maximum trace length was kept at 5 because further increases would lead to less variance in the generated traces due to the limited number of components and dependencies in the shop architecture. This could lead to the agents overfitting on the few long traces.

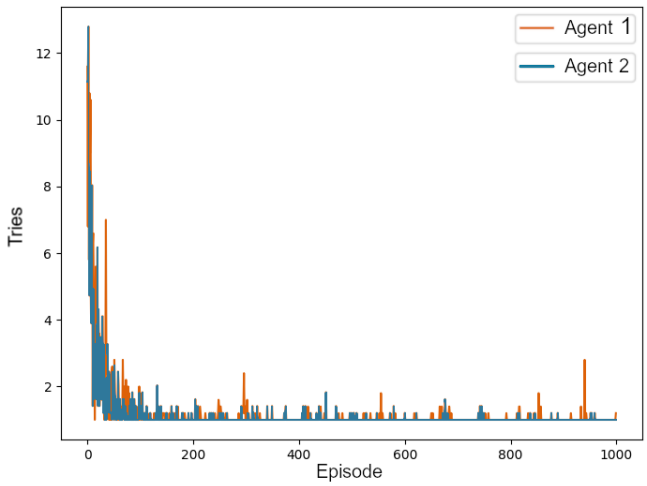
In a next experiment, it is evaluated how robust those trained agents are to changes of the environment.These changes come in the form of failure traces with lengths, that the agent hasn't seen before. We also add a baseline agent that chooses components from the failure trace with an equal probability for comparison.

Results. For the first experiment, Figure 9a shows the number of tries needed for agents trained on different trace sizes to find the correct failing component. It is fitted to an exponential curve for better visibility. It can be seen that after around 400 episodes, all agents have converged and only need a single try to find the correct component.

In Figure 9c the moving average of the regret for the same agents is shown. It can be observed, that even after 400 episodes, the training still results in improvement of the regret. To more easily see when this happens, the regret data was fitted to a generalized logistic function as described in the Scalability section. Looking at

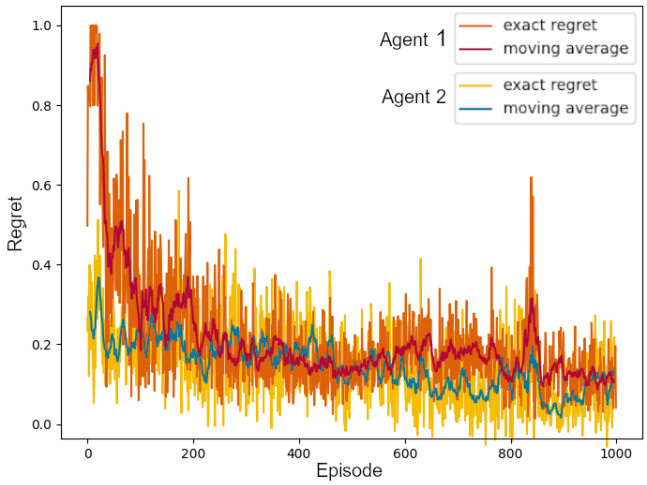


(a) Agents' regret to respective failure graph with 10 shops each.

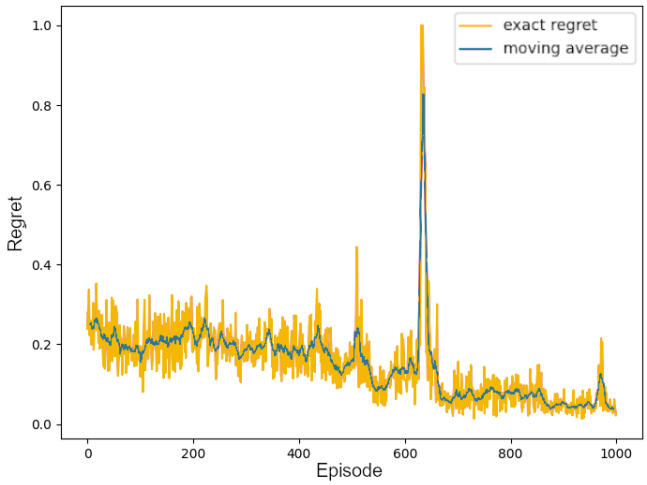


(b) Raw tries data for both agents with 10 shops each.

Figure 7: Phase 1: Train two agents for 1000 episodes with different failure graph configurations.



(a) Robustness deactivated (Baseline): Regret of *Agent 1* and *Agent 2* with 5 shops each



(b) Robustness activated: Regret of all 10 shops

Figure 8: Phase 2: Evaluate pre-trained agents with failure graph 1 with our robustness feature.

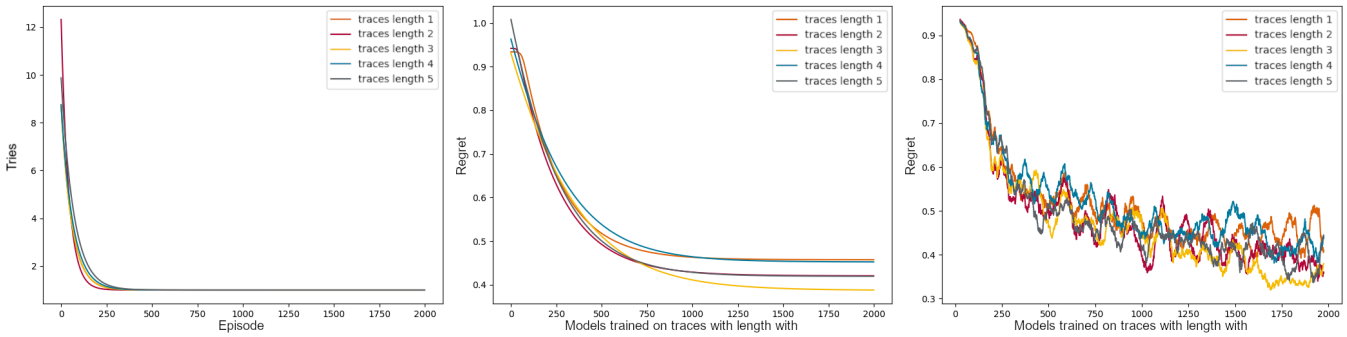
Figure 9b we can observe that the convergence of the regret starts around 1000 episodes.

In the second experiment, we compare the average regrets for 100 failure trace injections for the different agents that we previously trained. In the first two tables (Table 1 and 2) we use the trained models that performed best on their respective traces. We choose them by finding the minimum of the moving averages in Figure 9c. All those are in the late phase of the training from the 1600 to the 1900 episodes.

We observe in Table 1 that the average regret continuously improves and approaches the regret of the model that was trained on traces with length 5. In Table 2 we can observe that all tests result in a average regret of around 0.6. For the next two tables(Table 3 and 4) the previous experiments were repeated using the models at a point of time when the regret convergence starts, at around 1000 episodes.

In Figure 10a and Figure 10b we can see the average tries an agent needs to find the failing component when we inject traces of length 3 and 5 respectively. We can also see the results of the baseline agent. It can be observed in both cases, that the agents are close w.r.t. the length of traces tested on, perform the best. The baseline agent performs worst in both scenarios.

Discussion. First we have a look at the convergence of the different agents concerning the regret. We observe in Figure 9b that the agents start to converge around epoch 1000 to values from 0.4 to

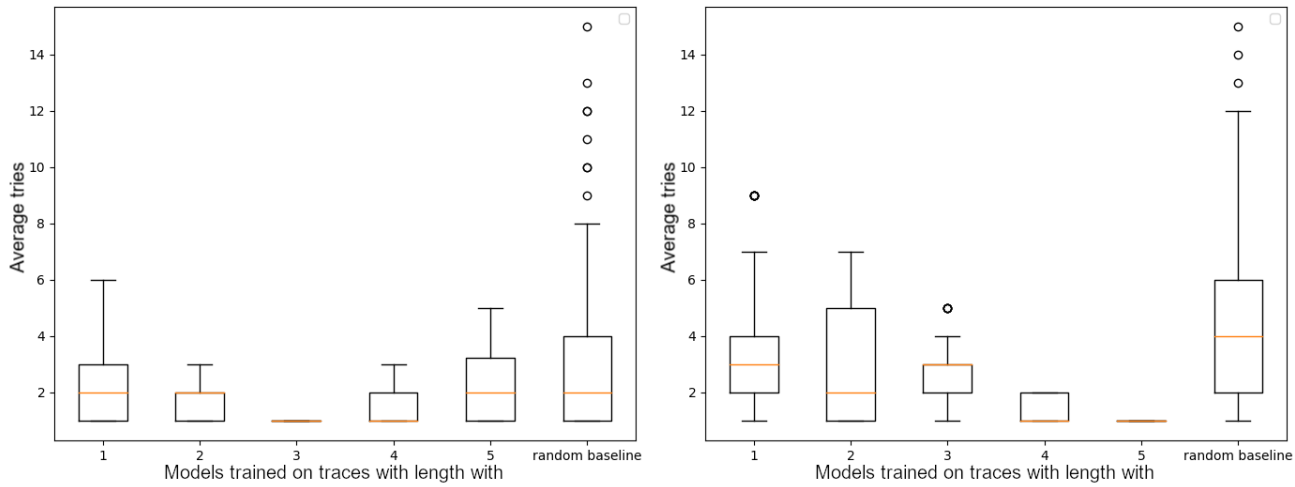


(a) Fitted number of tries for agents trained on different trace lengths

(b) Fitted regret functions

(c) Moving average over 50 episodes

Figure 9: Experiment 1: Comparison of convergence curves of agents trained on failure traces with different lengths.



(a) Test on random traces with length 3.

(b) Test on random traces with length 5.

Figure 10: Experiment 2: Average tries an agent needs to find the failing component when we inject traces of length 3 and 5 respectively.

Trained on Traces with Length	Tested on Traces with Length	Average Regret	Percentage Change
1	5	0.7999	-
2	5	0.7950	0.6
3	5	0.7882	1.5
4	5	0.6650	16.9
5	5	0.4009	49.9

Table 1: Average regret obtained from tests on traces length 5 using best model

Trained on Traces with Length	Tested on Traces with Length	Average Regret
1	2	0.6656
2	3	0.5459
3	4	0.6479
4	5	0.6650

Table 2: Average regret obtained from tests on longer traces using best model

0.5. We also see that values we converge to improve from agent 1 to agent 3. This is expected, as we suppose that with longer traces, the agent is able to learn more about the the architecture of the shop and the dependencies between its components. But this trend does not continue, as agents 4 and 5 perform worse. Therefore we have to say this assumption does not hold.

Now we want to analyse the region of regret values that we converge to. Interestingly we never get a better than 0.4. Even in the case where the trace is of length 1, meaning the agent just has to pick the one failing component that exists. This can most likely be

Trained on Traces with Length	Tested on Traces with Length	Average Regret	Percentage Change
1	5	0.7937	-
2	5	0.7214	9.1
3	5	0.7814	1.5
4	5	0.6422	19.1
5	5	0.4442	44.0

Table 3: Average regret obtained from tests on traces length 5 using model at convergence

Trained on Traces with Length	Tested on Traces with Length	Average Regret
1	2	0.6912
2	3	0.6412
3	4	0.7045
4	5	0.6422

Table 4: Average regret obtained from tests on longer traces using model at convergence

explained by having a look to when to number of tries converges to 1. As said before, this happens around episode 400 (Figure 9a), meaning after this episode, the agent almost never gets a negative reward again, because negative rewards are only given when the agent needs more than one try to pick the correct component. Thus after episode 400, the agent will only get reinforced with the probability distribution that it already has. Since we do not currently have a more precise way to penalize the probability distribution more accordingly, by e.g. giving higher rewards for a lower regret and vice versa, we will start to converge not too far from this episode. Having a more precise reward and penalization system should probably improve the regret value we converge to and could be an interesting further experiment.

Next we look at the experiments in Table 1. Here we test different pre-trained agents on traces with length 5. We would have assumed to have a diminishing returns when it comes to regret value improvements by training on continuously larger traces. This would be, because the agents that were trained on longer traces, should have a clearer picture of the dependencies in the architecture and thus a better regret value. Here on the other hand, we don't see diminishing returns, but almost equal results for the first 3 agents. Big improvements come only with the agent that was trained on 4 traces.

This observation is also confirmed when we look at Table 2. In this experiment we train and test on trace lengths with a gap of one component. We see that overall the results lie in the same range. Since the relative difference in the size of traces decreases, one would assume that the regret would improve consequently.

Therefore we must follow that the agents fail to learn the architecture of the shops and their dependencies. By having the model learn a separate representation of the architecture, we could mitigate this, but with the cost of having a more complex model.

In Table 3 the same experiment as above was repeated using models that didn't fully converge yet. The idea behind this, was that the models might still generalize better or be not too fitted to the specific trace sizes they were trained on. This turned out to be true, as all results are slightly better than in Table 1 (except for the one of trace length 5, which should obviously be worse). In Table 4 the same pattern as before can be observed, the results lie all in same range of values. Here on the other hand, the results all seem a bit worse then before. This can be explained by the proximity to the trace lengths that was tested on. Since we test close to the trace length that we trained on, the better fitted models seem to perform a bit better and beat the generalisation.

A part of this is also observable in Figure 10a. Here we see a boxplot that shows the amount of tries needed to find the failing component of tries of length 3 for different agents. We see that agents with lengths 2 and 4 had the best results, while agents 1 and 5 perform worse. A similar pattern can be observed in Figure 10b. Therefore it can be concluded that proximity to the trace length tested on seems to important.

But if we come back to the question of how robust these agents are to changes, we see that even agents that were that were trained and tested on totally different trace sizes, still perform decent. While not perfect, the results were always better than the random baseline. Thus we can argue that these are agents have a certain degree of robustness to those changes.

5.3 Scalability

Experimental Design. We have introduced a multi-agent setup into our architecture to improve its scalability, specifically regarding setups with a larger number of shops. However, when using a multi-agent setup, there are multiple ways to distribute the shops among the agents. In the following experiments, we analyze and compare different combinations of agents and shops in order to determine optimal setups.

To achieve this, we use two measures of quality: The first such measure is the average amount of tries among all agents that it takes for an episode to finish, which occurs when all issues are fixed. We use tries as the measure, as we measure the number of attempts it takes before all issues are fixed. The second measure we use is the regret. In each iteration, each agent assigns a certain probability to choose each issue. The regret is defined as the probability that an agent chooses an issue that is not the root cause. Let $p_i, 1 \leq i \leq 18$ be the probability for each of the 18 components that they are chosen to be fixed. Let component j be the root cause of the failure. The regret Δ, then, is defined as

$$\Delta := 1 - p_j.$$

These measures each have their own advantages and challenges: The tries have an easily determined baseline that they should be converging to, that being one try, since that is the optimal outcome. Additionally, they are the most direct measure of quality: The number of tries measures how long it takes to solve the issues, which is the number that we wish to minimize. Regret, however, does not have such an easy answer, and the values that regret converges to are a lot less consistent. On the other hand, the measure of tries obscures some information: As soon as the correct component is

chosen frequently, the outcome will be one try. However, this ignores whether that component was chosen by a slim margin over another, or whether it had a wide margin over the next component. The regret, then, can easily show these intricacies.

As the data is naturally noisy, we will make use a goodness of fit function to express the basic shape of each curve while minimizing the effect of noise. For the tries, we use an exponential function with a negative base, with t being the episode number and y being the expected number of trials:

$$y = a \cdot e^{-bt} + 1.$$

We use one as the asymptote due to the assumption that the tries will converge towards one. As will be seen from the data, an exponential function is a good fit to model the tries, as it expresses a steep initial descent followed by a slower approach towards an asymptote.

An exponential function cannot be used to model the regret. Instead, we use a generalized logistic function, with y being the expected regret and t the episode number:

$$y = a + \frac{k - a}{\left(1 + e^{-b(x-m)}\right)^{\frac{1}{v}}}.$$

Where a and k are the asymptotes, m is the time at which the asymptotes are switched, b is the growth or decay rate, and v influences how steeply growth or decay occurs on which asymptote. We use a logistic function to model the regret since in most cases, it first decreases slowly, then decreases at a faster rate, before then converging towards an asymptote. For setups that perform very well, the decrease begins at a fast rate, but even these can be fitted towards a logistic function, since the upper asymptote can be set to be outside of the data range.

To fit the methods to the data, we used the `scipy.optimize.curve_fit` function within *Python*. This function performs the task of finding the parameters that best approximate the data using a least squares approach.

Results. We will first consider the case where the number of total shops is constant and we distribute a variable number of agents among those shops. As an example, for 10 total shops, we can either have 1 agent with 10 shops each, 2 agents with 5 shops each, 5 agents with 2 shops each, or 10 agents with 1 shop each.

We will consider the plot for the raw amount of tries first:

As can be seen in Figure 11, as you add more agents—and with that, fewer shops per agent—it takes longer for the tries to converge towards 1 try. This can be seen more clearly in the fitted exponential curves:

One can see a clear hierarchy in Figure 12, where setups with fewer agents perform better than those with more agents.

We will analyze the quality of these curves via two metrics: Firstly, we will consider the point of convergence. Since these plots are asymptotic towards 1 try, they will never exactly reach 1, and their derivative will never reach 0. We can, however, set an ε and find out when the derivative reaches ε. We know that

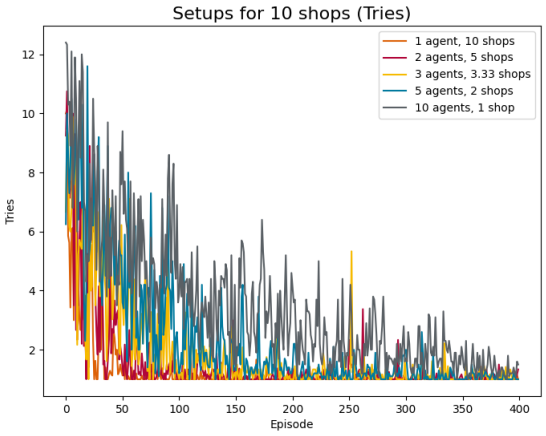


Figure 11: Raw tries data for Setups for 10 shops

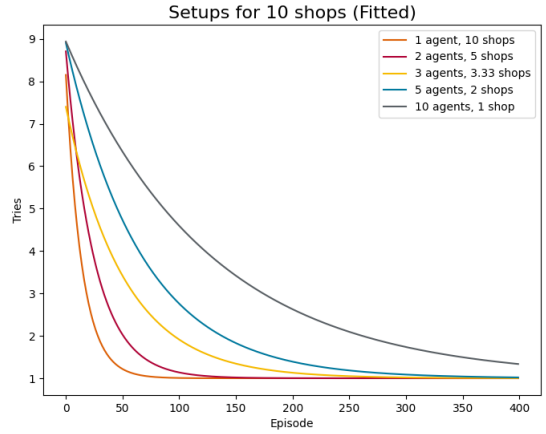


Figure 12: Fitted tries data for Setups for 10 shops

$$\varepsilon = \frac{d}{dt} a \cdot e^{-bt} + 1$$
$$\varepsilon = -ab \cdot e^{-bt}$$
$$t = \frac{\ln\left(\frac{-ab}{\varepsilon}\right)}{b}$$

With that, we can now calculate the point of convergence for a given ϵ as the error term that we allow. The results can be seen in Table 5.

Once we have a set point of convergence, we can calculate outliers. This is done as follows: We start from the convergence point defined by ε and define another threshold ε_2. An outlier is then defined as any episode after the convergence point where the raw data point is greater than $1 + \varepsilon_2$. We then calculate the relative frequency of outliers, that being the amount of outlier data points

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 10 shops	23.028	**56.003**	**88.979**	**121.955**	**154.931**
2 agents, 5 shops	28.116	85.258	142.401	199.543	256.686
3 agents, 3.33 shops	11.306	129.490	247.674	365.858	484.042
5 agents, 2 shops	11.491	164.627	317.763	470.899	624.034
10 agents, 1 shop	**-58.501**	232.048	522.598	813.147	1103.696

Table 5: Convergence points for Setups for 10 shops. The earliest convergence for each ε is bolded.

post-convergence divided by the total number of episodes post-convergence. This way, we can account for different convergence points. With that, we have two dimensions to consider in Table 6: The value of ε, shown in the rows, and the value of ε_2, shown in the columns. For any given $(\varepsilon, \varepsilon_2)$ pair, we can calculate the lowest relative outlier frequency among all setups and supply the ID of the setup which yielded that number, in rising order starting from 0. The order corresponds to the order in which they are listed on other tables, such as Table 5.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.223 (#0)	0.175 (#0)	0.154 (#0)	0.144 (#0)	0.127 (#0)
0.1	0.223 (#0)	0.175 (#0)	0.154 (#0)	0.144 (#0)	0.127 (#0)
0.2	0.101 (#0)	0.058 (#0)	0.045 (#0)	0.043 (#0)	0.041 (#0)
0.3	0.077 (#0)	0.041 (#0)	0.032 (#0)	0.029 (#0)	0.029 (#0)
0.4	0.059 (#0)	0.023 (#0)	0.023 (#0)	0.022 (#0)	0.020 (#0)
0.5	0.040 (#0)	0.009 (#0)	0.010 (#0)	0.000 (#2)	0.008 (#0)

Table 6: Fewest relative outliers for Setups for 10 shops

We will now consider the regret. The raw regret can be seen on Figure 14.

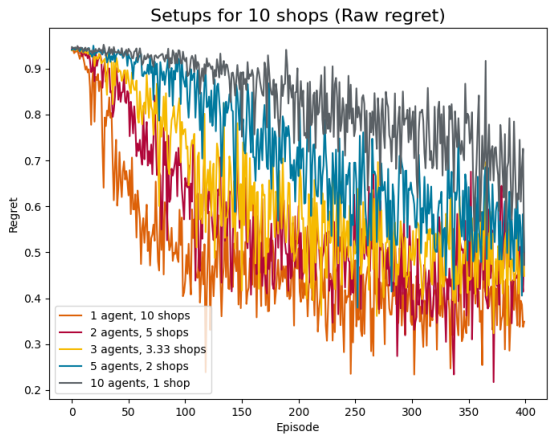


Figure 13: Raw regret data for Setups for 10 shops

The fitted general logistic functions for the data can be seen in Figure 14. We can once more see how the 1 agent setup uniformly performs above the others. Additionally, we can see that the curves for the 1 agent and 2 agents setups resemble an exponential function, whereas the other plots have a more classical logistics function shape.

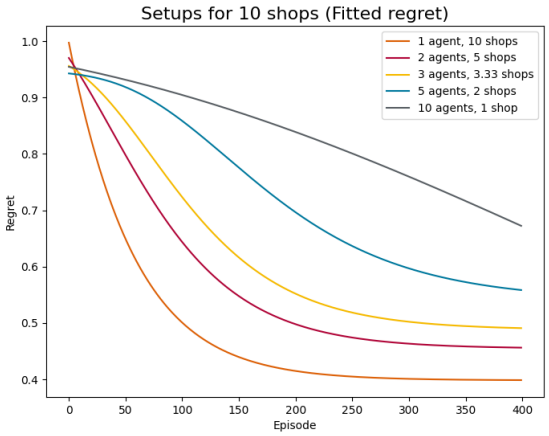


Figure 14: Fitted regret data for Setups for 10 shops

Due to the fact that the general logistics function is monotonic, we can inverse it. This allows us to find the episode t where a certain regret y is reached:

$$t = \frac{-\ln\left(\left(\frac{a-k}{a-y}\right)^v - 1\right)}{b} + m$$

Notably, this only returns a valid t if $\min(a,k) < y < \max(a,k)$, as a and k are the asymptotes. With this formula, we can now calculate the expected episode for when a setup first reaches a given regret. We can see the results of this in Table 7, which confirms that the 1 agent setup performs the best. We display a — in the table if the regret value is outside of the asymptotes or if said regret is not reached within the alotted episodes.

	0.4	0.5	0.6	0.7	0.8
1 agent, 10 shops	**322.69**	**100.37**	**62.26**	**39.57**	**23.23**
2 agents, 5 shops	—	196.96	119.19	79.62	49.43
3 agents, 3.33 shops	—	310.82	160.11	109.38	71.46
5 agents, 2 shops	—	—	295.17	197.55	135.73
10 agents, 1 shop	—	—	—	368.50	251.06

Table 7: Regret analysis for Setups for 10 shops. The earliest setup to reach the given threshold is in bold.

We will now consider the same analysis for 20 total shops. For brevity's sake, we will exclude the raw data plots from now on.

In Figure 15, we can once again see a clear hierarchy of fewer agents performing better. We can now perform convergence and outlier analysis on this data:

Now we will look at the regret analysis. We will consider the fitted regret in Figure 16. We can once again see that as you add more agents, the regret decreases more slowly.

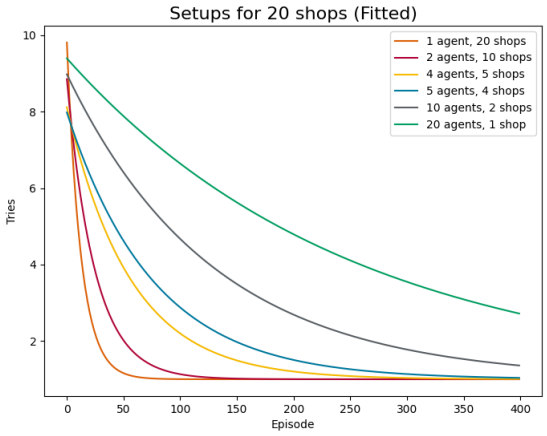


Figure 15: Fitted tries data for Setups for 20 shops

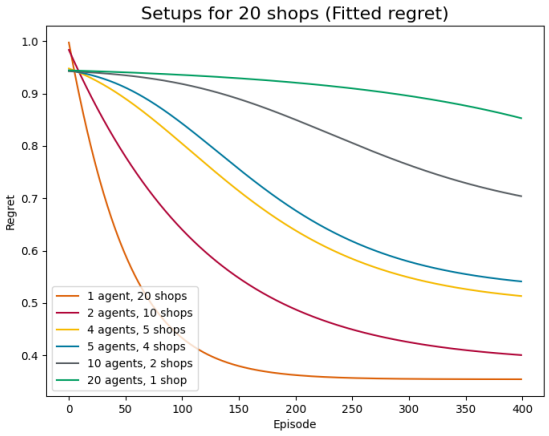


Figure 16: Fitted regret data for Setups for 20 shops

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 20 shops	24.360	**53.037**	**81.714**	**110.391**	**139.068**
2 agents, 10 shops	28.523	84.865	141.207	197.548	253.890
4 agents, 5 shops	13.307	142.538	271.769	401.000	530.232
5 agents, 4 shops	-6.630	168.688	344.006	519.324	694.642
10 agents, 2 shops	-61.823	234.879	531.582	828.284	1124.987
20 agents, 1 shop	**-276.384**	303.110	882.603	1462.097	2041.591

Table 8: Convergence points for Setups for 20 shops. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.253 (#0)	0.220 (#0)	0.186 (#0)	0.173 (#0)	0.162 (#0)
0.1	0.253 (#0)	0.220 (#0)	0.186 (#0)	0.173 (#0)	0.162 (#0)
0.2	0.128 (#0)	0.090 (#0)	0.066 (#0)	0.062 (#0)	0.065 (#0)
0.3	0.091 (#0)	0.058 (#0)	0.035 (#0)	0.035 (#0)	0.038 (#0)
0.4	0.069 (#0)	0.043 (#0)	0.022 (#0)	0.021 (#0)	0.023 (#0)
0.5	0.051 (#0)	0.032 (#1)	0.016 (#0)	0.014 (#0)	0.015 (#0)

Table 9: Fewest relative outliers for Setups for 20 shops

	0.4	0.5	0.6	0.7	0.8
1 agent, 20 shops	**123.95**	**72.35**	**48.26**	**31.94**	**19.33**
2 agents, 10 shops	—	187.68	119.08	76.00	44.04
4 agents, 5 shops	—	—	233.43	158.25	102.26
5 agents, 4 shops	—	—	270.19	184.08	124.89
10 agents, 2 shops	—	—	—	—	256.23
20 agents, 1 shop	—	—	—	—	—

Table 10: Regret analysis for Setups for 20 shops. The earliest setup to reach the given threshold is bolded.

Lastly, we will look at the data for 80 shops, a considerable increase in scale.

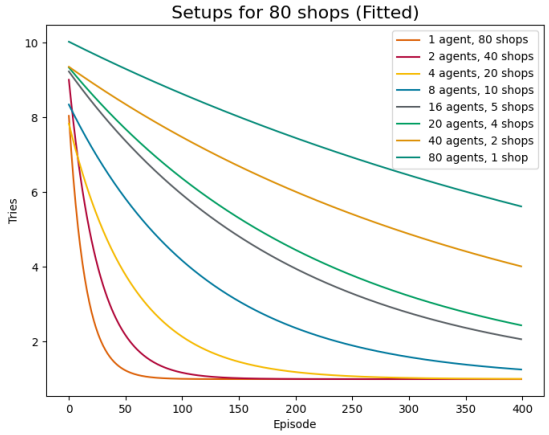


Figure 17: Fitted tries data for Setups for 80 shops

Figure 17 once more displays the hierarchy where a lower number of agents corresponds to more rapidly decreasing tries.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 80 shops	22.989	**56.678**	**90.366**	**124.054**	**157.743**
2 agents, 40 shops	29.249	89.202	149.154	209.106	269.058
4 agents, 20 shops	11.353	139.681	268.009	396.337	524.665
8 agents, 10 shops	-57.494	216.590	490.675	764.759	1038.844
16 agents, 5 shops	-168.855	280.913	730.680	1180.447	1630.214
20 agents, 4 shops	-227.733	295.350	818.434	1341.517	1864.600
40 agents, 2 shops	-605.190	296.664	1198.518	2100.372	3002.226
80 agents, 1 shop	**-1124.380**	247.425	1619.230	2991.035	4362.840

Table 11: Convergence points for Setups for 80 shops. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.255 (#0)	0.207 (#0)	0.168 (#0)	0.160 (#0)	0.145 (#0)
0.1	0.255 (#0)	0.207 (#0)	0.168 (#0)	0.160 (#0)	0.145 (#0)
0.2	0.125 (#0)	0.076 (#0)	0.049 (#0)	0.047 (#0)	0.045 (#0)
0.3	0.082 (#0)	0.041 (#0)	0.019 (#0)	0.018 (#0)	0.017 (#0)
0.4	0.053 (#0)	0.020 (#0)	0.013 (#0)	0.011 (#0)	0.008 (#0)
0.5	0.048 (#0)	0.015 (#0)	0.013 (#0)	0.011 (#0)	0.008 (#0)

Table 12: Fewest relative outliers for Setups for 80 shops

We will now consider the fitted regret for 80 shops. As can be seen from Figure 18, we can recognize the same hierarchy of performance here. Notably, the 80 agents and 40 agents setups makes barely any progress regarding regret.

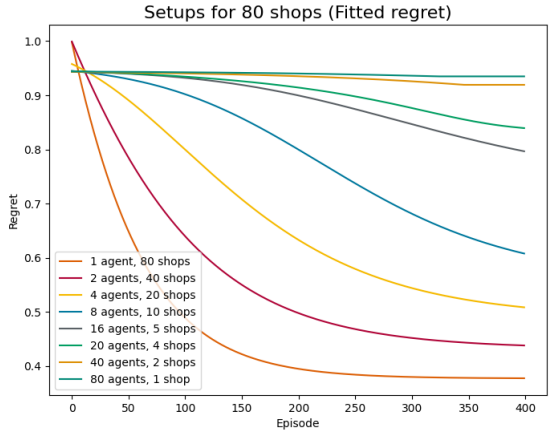


Figure 18: Fitted regret data for Setups for 80 shops

	0.4	0.5	0.6	0.7	0.8
1 agent, 80 shops	**185.35**	**94.56**	**61.22**	**39.84**	**23.82**
2 agents, 40 shops	—	196.68	118.84	76.34	45.76
4 agents, 20 shops	—	—	228.10	154.50	99.67
8 agents, 10 shops	—	—	—	281.98	199.62
16 agents, 5 shops	—	—	—	—	390.80
20 agents, 4 shops	—	—	—	—	—
40 agents, 2 shops	—	—	—	—	—
80 agents, 1 shop	—	—	—	—	—

Table 13: Regret analysis for Setups for 80 shops. The earliest setup to reach the given threshold is bolded.

Now that we have compared setups with the same number of total shops, we will compare setups with the same number of shops per agent. This way, we can analyze how adding more agents and shops at the same rate affects performance.

When looking at the outliers in Table 15, one will see that there is no data for $\varepsilon \geq -10^{-3}$. That is because for these values, none of

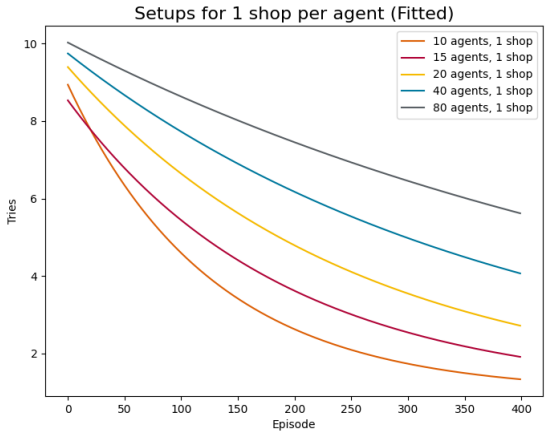


Figure 19: Fitted tries data for Setups for 1 shop per agent

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
10 agents, 1 shop	-58.501	**232.048**	**522.598**	**813.147**	**1103.696**
15 agents, 1 shop	-174.347	261.424	697.195	1132.967	1568.738
20 agents, 1 shop	-276.384	303.110	882.603	1462.097	2041.591
40 agents, 1 shop	-560.863	316.479	1193.820	2071.162	2948.503
80 agents, 1 shop	**-1124.380**	247.425	1619.230	2991.035	4362.840

Table 14: Convergence points for Setups for 1 shop per agent. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.988 (#0)	0.970 (#0)	—	—	—
0.1	0.958 (#0)	0.904 (#0)	—	—	—
0.2	0.912 (#0)	0.796 (#0)	—	—	—
0.3	0.882 (#0)	0.731 (#0)	—	—	—
0.4	0.843 (#0)	0.647 (#0)	—	—	—
0.5	0.802 (#0)	0.581 (#0)	—	—	—

Table 15: Fewest relative outliers for Setups for 1 shop per agent

	0.4	0.5	0.6	0.7	0.8
10 agents, 1 shop	—	—	—	**368.50**	**251.06**
15 agents, 1 shop	—	—	—	—	382.85
20 agents, 1 shop	—	—	—	—	—
40 agents, 1 shop	—	—	—	—	—
80 agents, 1 shop	—	—	—	—	—

Table 16: Regret analysis for Setups for 1 shop per agent. The earliest setup to reach the given threshold is bolded.

the setups reach that convergence point within the time frame we have data for.

We will now perform this same analysis for setups with 2 shops per agent.

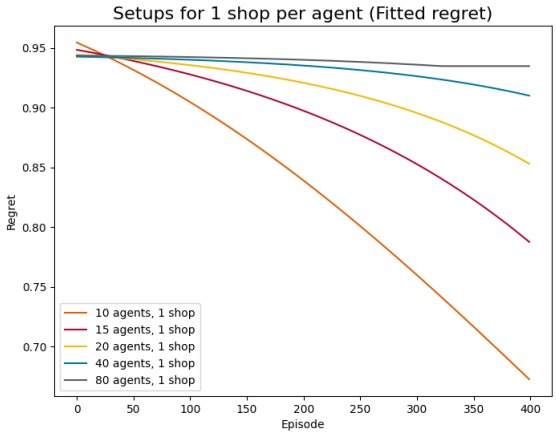


Figure 20: Fitted regret data for Setups for 1 shop per agent

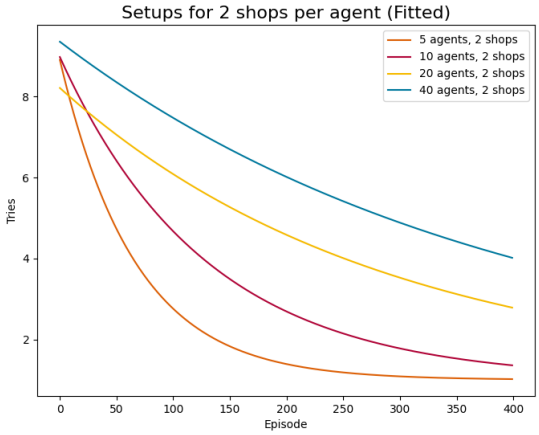


Figure 21: Fitted tries data for Setups for 2 shops per agent

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
5 agents, 2 shops	11.491	**164.627**	**317.763**	**470.899**	**624.034**
10 agents, 2 shops	-61.823	234.879	531.582	828.284	1124.987
20 agents, 2 shops	-393.683	264.500	922.682	1580.865	2239.047
40 agents, 2 shops	**-605.190**	296.664	1198.518	2100.372	3002.226

Table 17: Convergence points for Setups for 2 shops per agent. The earliest convergence for each ε is bolded.

Now that we have analyzed setups with the same number shops per agent, we will finally compare setups with the same number of agents. As such, we will consider how performance changes if new shops are equally distributed among agents. We will first consider setups with 1 agent and then setups with 2 agents.

Discussion. Let us first consider the examples with a constant number of shops. As we have already seen, it universally leads to better

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.768 (#0)	0.638 (#0)	0.402 (#0)	—	—
0.1	0.673 (#0)	0.485 (#0)	0.256 (#0)	—	—
0.2	0.549 (#0)	0.298 (#0)	0.122 (#0)	—	—
0.3	0.497 (#0)	0.230 (#0)	0.037 (#0)	—	—
0.4	0.441 (#0)	0.174 (#0)	0.024 (#0)	—	—
0.5	0.405 (#0)	0.136 (#0)	0.012 (#0)	—	—

Table 18: Fewest relative outliers for Setups for 2 shops per agent

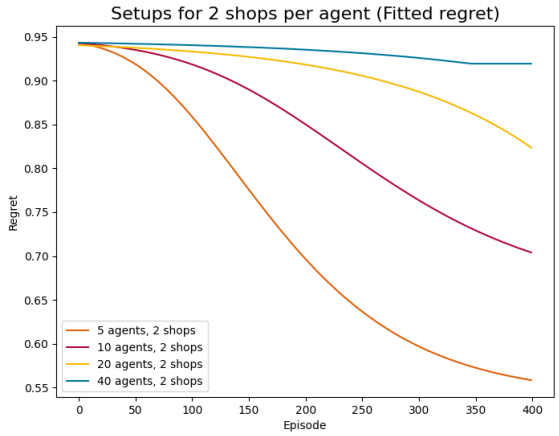


Figure 22: Fitted regret data for Setups for 2 shops per agent

	0.4	0.5	0.6	0.7	0.8
5 agents, 2 shops	—	—	**295.17**	**197.55**	**135.73**
10 agents, 2 shops	—	—	—	—	256.23
20 agents, 2 shops	—	—	—	—	—
40 agents, 2 shops	—	—	—	—	—

Table 19: Regret analysis for Setups for 2 shops per agent. The earliest setup to reach the given threshold is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 10 shops	23.028	56.003	88.979	121.955	154.931
1 agent, 15 shops	**22.685**	**48.087**	**73.488**	**98.889**	**124.291**
1 agent, 20 shops	24.360	53.037	81.714	110.391	139.068
1 agent, 40 shops	24.957	55.149	85.341	115.533	145.724
1 agent, 80 shops	22.989	56.678	90.366	124.054	157.743

Table 20: Convergence points for Setups for 1 agent. The earliest convergence for each ε is bolded.

results if fewer total agents are used. This could be explained as follows: When we have more agents, and with that, fewer shops per agent, the data that the agents see becomes more sparse. As a result, they will learn more slowly, since it will take them a longer amount of time to receive the same amount of data.

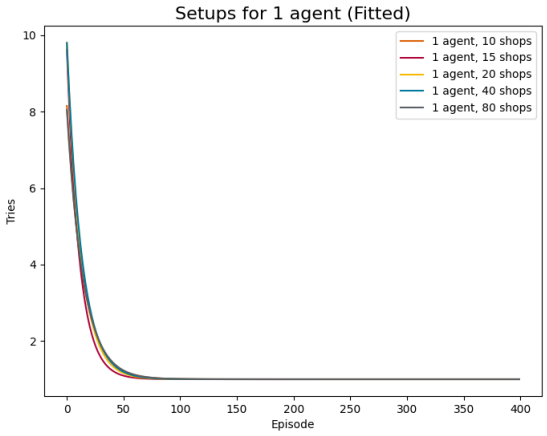


Figure 23: Fitted tries data for Setups for 1 agent

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.223 (#0)	0.175 (#0)	0.154 (#0)	0.144 (#0)	0.127 (#0)
0.1	0.223 (#0)	0.175 (#0)	0.154 (#0)	0.144 (#0)	0.127 (#0)
0.2	0.101 (#0)	0.058 (#0)	0.045 (#0)	0.043 (#0)	0.041 (#0)
0.3	0.077 (#0)	0.041 (#0)	0.019 (#4)	0.018 (#4)	0.017 (#4)
0.4	0.053 (#4)	0.020 (#4)	0.013 (#4)	0.011 (#4)	0.008 (#4)
0.5	0.040 (#0)	0.009 (#0)	0.010 (#0)	0.007 (#3)	0.007 (#1)

Table 21: Fewest relative outliers for Setups for 1 agent

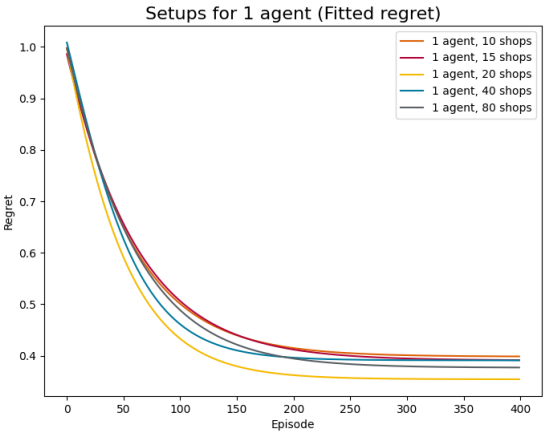


Figure 24: Fitted regret data for Setups for 1 agent

One notable case where more agents perform better is for the convergence, specifically for $\varepsilon = -10^{-1}$, as can be seen in Table 5, Table 8, and Table 11. The reason for this is as follows: Since more agents generally converge more slowly, this means that even starting out, they already have a small derivative. However, on the other hand, it takes them a lot longer to reach a derivative of $\varepsilon = -10^{-2}$.

	0.4	0.5	0.6	0.7	0.8
1 agent, 10 shops	322.69	100.37	62.26	39.57	23.23
1 agent, 15 shops	248.63	103.27	64.41	40.73	23.50
1 agent, 20 shops	**123.95**	**72.35**	**48.26**	**31.94**	**19.33**
1 agent, 40 shops	179.13	82.35	55.22	37.67	23.94
1 agent, 80 shops	185.35	94.56	61.22	39.84	23.82

Table 22: Regret analysis for Setups for 1 agent. The earliest setup to reach the given threshold is bolded.

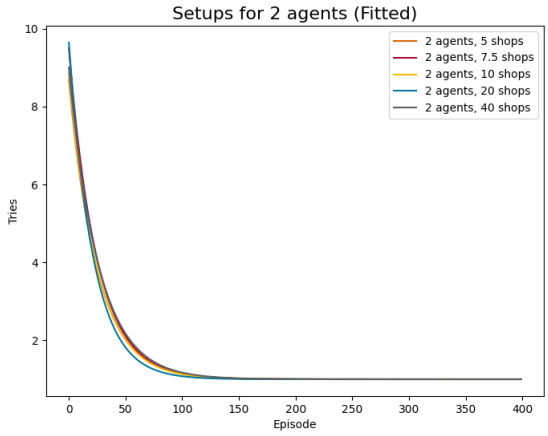


Figure 25: Fitted tries data for Setups for 2 agents

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
2 agents, 5 shops	**28.116**	85.258	142.401	199.543	256.686
2 agents, 7.5 shops	30.485	87.068	143.652	200.235	256.818
2 agents, 10 shops	28.523	84.865	141.207	197.548	253.890
2 agents, 20 shops	29.801	**78.607**	**127.413**	**176.220**	**225.026**
2 agents, 40 shops	29.249	89.202	149.154	209.106	269.058

Table 23: Convergence points for Setups for 2 agents. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.423 (#2)	0.349 (#2)	0.291 (#2)	0.238 (#2)	0.233 (#2)
0.1	0.377 (#2)	0.302 (#2)	0.248 (#2)	0.198 (#2)	0.192 (#2)
0.2	0.272 (#2)	0.187 (#2)	0.140 (#2)	0.114 (#2)	0.103 (#2)
0.3	0.210 (#2)	0.117 (#2)	0.081 (#2)	0.074 (#2)	0.055 (#2)
0.4	0.151 (#0)	0.073 (#2)	0.043 (#0)	0.040 (#0)	0.021 (#0)
0.5	0.111 (#0)	0.032 (#2)	0.019 (#0)	0.020 (#0)	0.014 (#0)

Table 24: Fewest relative outliers for Setups for 2 agents

As such, we can deduce that a derivative of $\varepsilon = -10^{-1}$ is too large to meaningfully define convergence at our current scale, especially since for all other ε the data is conclusive in that 1 agent setups perform best.

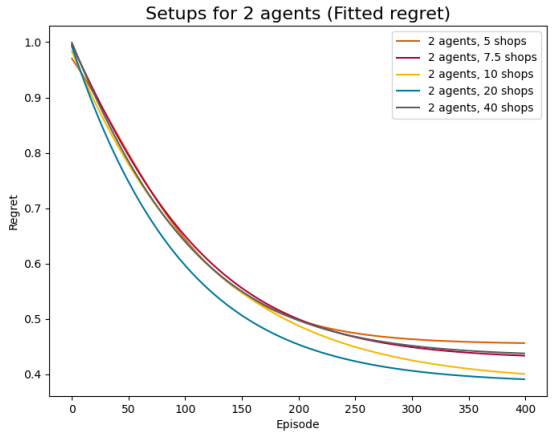


Figure 26: Fitted regret data for Setups for 2 agents

	0.4	0.5	0.6	0.7	0.8
2 agents, 5 shops	—	196.96	119.19	79.62	49.43
2 agents, 7.5 shops	—	199.28	123.38	80.35	48.62
2 agents, 10 shops	—	187.68	119.08	76.00	44.04
2 agents, 20 shops	**328.21**	**154.67**	**98.45**	**63.22**	**37.17**
2 agents, 40 shops	—	196.68	118.84	76.34	45.76

Table 25: Regret analysis for Setups for 2 agents. The earliest setup to reach the given threshold is bolded.

Let us now consider the data on outliers. While the 1 agent setups usually perform best, there are some exceptions. In Table 6, we see the 3 agents setup perform best for $\varepsilon = -10^{-4}, \varepsilon_2 = 0.5$. The reason for this is as follows: The 3 agents setup reaches an outlier frequency of 0 thanks to the fact that for $\varepsilon = -10^{-4}$, it converges at 365.86, meaning that there are only 34 values to consider outliers for. As such, there are fewer general opportunities for failure, especially since $\varepsilon_2 = 0.5$ means that there is quite a lot of leeway for what constitutes an outlier.

We can see from Figure 12 and Figure 14 that the regret data provides a lot more granularity than the tries data. Especially when considering the 1 agent and 2 agents setups, the tries converge rather quickly, whereas it takes more episodes for the regret to converge. This can be explained by the fact that for the tries, it does not matter how large the regret is, as long as the probability for the right component to be chosen is larger than any other component's. On the other hand, the regret still leaves a lot of room to grow after that point, leading to more certainty.

For 20 shops, we can see 2 agents reaching the lowest frequency of outliers for $\varepsilon = -10^{-2}, \varepsilon_2 = 0.5$ in Table 9. Notably, this is by a very slim margin: The 2 agents setup has a relative frequency of 0.03175, the 1 agent setup has one of 0.03179. As such, while the 1 agent setup is better overall, it loses some ground to the 2 agents setup, which comes close to the 1 agent setup in a few other circumstances as well.

For 80 shops, we can see that 1 agent outperforms all other setups regarding outliers in Table 12. This further cements that for 80 shops, adding more agents results in even harsher penalties than for fewer total shops.

Two notable aspects can be observed throughout each outlier chart: Firstly, as ε_2 increases, the frequency of outliers shrinks. This can easily be explained by the fact that a higher threshold for what constitutes an outlier leads to fewer outliers. The other interesting aspect is that a higher ε also leads to a lower frequency of outliers. This is likely because as the episodes increase, the general relative frequency of outliers also decreases, since the agents are still learning.

Let us now consider the data for a fixed number of shops per agent. We can very clearly see that adding more total agents decreases performance in Figure 19, Figure 20, Figure 21, and Figure 22. This fits without earlier observation that the critical element for performance is the number of relative data that an agent receives. As we add more shops and agents, the relative frequency of data that a given agent receives decreases—for 10 agents with 1 shop each, each agent receives $\frac{1}{10}$ of the data, whereas for 80 agents with 1 shop each, each agent receives only $\frac{1}{80}$ of the data.

It is also notable for the 1 shop per agent setups in Table 15 that for $\varepsilon \geq 10^{-3}$, there is no data for any of the outliers, since even the setup that converges the fastest for $\varepsilon = 10^{-3}$, that being 10 agents, 1 shop at $t = 522.598$, converges at a point for which we do not have raw data. As such, outliers cannot be determined for these values. Even for the convergence points for which we do have data, the results here are rather poor compared to the 1 agent, 10 shops setup, for instance. While the 10 agents setup has the fewest relative outliers for $\varepsilon = -10^{-2}, \varepsilon_2 = 0.5$ at 0.581, the 1 agent setup reaches a relative outlier frequency of 0.009 at that point.

Similar observations can be made for 2 shops per agent, although notably the minimum outlier frequency is much improved here thanks to the addition of the 5 agents, 2 shops setup, which manages to have a better data-per-agent ratio than any setup for 1 shop per agent at $\frac{1}{5}$. Other than that, the data seems to clearly cement the conclusion that adding more agents at the same rate you add shops worsens performance significantly.

Lastly, we will look at the data points where the number of shops is constant. This is notable for the reason that each of these setups have the same number of data-per-agent ratio: For 1 agent setups, the agent receives all of the data, whereas for 2 agents, each agent receives $\frac{1}{2}$ of the data. As such, the hypothesis would be that these setups should perform rather similarly.

Let us consider the 1 agent setups first. We can see in Table 20 that the 15 shops setup converges first for each of our measures, although we can also see that the distance that the other setups have is a lot smaller than what we have been seeing in previous tables. The outliers in Table 21 are where we see some interesting data: For $\varepsilon_2 \leq 0.2$, the 10 shops setup has the fewest outliers. However, for $\varepsilon_2 = 0.3, \varepsilon \geq -10^{-3}$ we can see that the 80 shops setup performs best. This can likely be explained by the fact that, in light of the similarities between the setups, the fact that 80 shops converges last also means that its data points overall incorporate more episodes of learning. The fact that this only happens for lower ε_2 thresholds seems to underline that, showing that while the 80 shops setup

seems to often go over smaller thresholds, it performs better when the threshold is set larger.

For $\varepsilon_2 = 0.4$, we can see a continuation of this fact: Now, the 80 shops setup is showing the fewest frequency of outliers overall. For $\varepsilon_2 = 0.5$, however, we can see yet another shift: For $\varepsilon \leq -10^{-3}$, we once again see the 10 shops setup performing the best, with 40 shops performing best for $\varepsilon = -10^{-4}$ and 15 shops performing best for $\varepsilon = -10^{-5}$. It seems then that many of the outliers in the 80 shops setup fell between 1.4 and 1.5, such that they allowed that setup to perform well for $\varepsilon_2 = 0.4$, but perform worse for $\varepsilon_2 = 0.5$.

Looking at the regret for the 1 agent setups in Table 22, we can very clearly see that the 20 agents setup is performing the best overall, beating out each of the other setups by a fair margin. This implies that 20 shops per agent is the best-performing ratio when trying to optimize regret, with diminishing returns as you add more shops.

Let us now consider the 2 agent setups. We can see in Table 23 that 5 shops per agent converges best for $\varepsilon = -10^{-1}$, whereas 20 shops per agent performs best for other values of ε. This is an interesting data point in that it marks the first time, barring the earlier exceptions for very poorly-performing setups at $\varepsilon = -10^{-1}$, that one setup does not converge first by all metrics. Although notably, the 20 shops setup only converges less than 2 episodes after the 5 shops setup.

When looking at outliers in Table 24, we can see that the 10 shops setup is performing best for $\varepsilon_2 \leq 0.3$, with the 5 shops setup performing best for higher thresholds. This implies that the 5 shops setup has most of its outliers under 1.3 tries.

What is overall notable regarding the tries data is the lack of a true best setup. While 20 shops converges first, 10 shops features the best outlier data. Overall, combined with the tries data from the 1 agent setups, it can be deduced that 10 shops per agent performs best as far as outliers are concerned, whereas 15-20 shops perform best regarding fast convergence.

When looking at the regret data in Table 25, one can see that 20 shops still performs far better than any other setup. Although one can also see that 10 shops appears to be catching up near the end, which could imply that 10 shops may be better in the long run—after all, it is the second setup to approach $y = 0.5$.

As such, we can overall deduce the following: For best performance, we wish to minimize the amount of total agents, since that way, each agent receives the most relative data. When we compare setups with the same amount of total agents, it appears that overall, the sweet spot appears to be 10 to 20 shops per agent, depending on whether you favor fast convergence, fewer outliers, or fast-shrinking regret.

5.4 Architecture Comparison

Experimental Design. As explained in subsection 4.1, we have altered the failure injection to use a Gaussian distribution to determine the number of failures, as opposed to the old architecture, which used a purely precoded set of injections chosen by $t' = t \mod 10$. We created a new architecture that is wholly random, using t as the seed, and one that is random within mod 10, that is, it uses t' as the seed. We will now analyze, using the same methods as outlines in subsection 5.3, how these changes altered the performance.

As a baseline, we will use a basic setup for 10 shops consisting of 3 agents, with two of them handling 3 shops and one of them handling 4 shops.

Results. First, we will look at the tries data for each of the architectures:

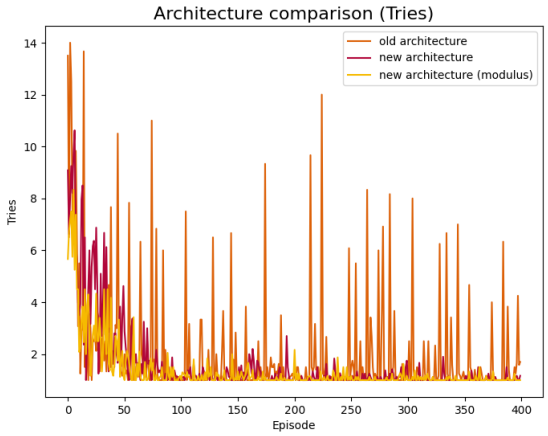


Figure 27: Raw tries data for Setups for Architecture comparison

We will now fit exponential curves to this data and analyze it:

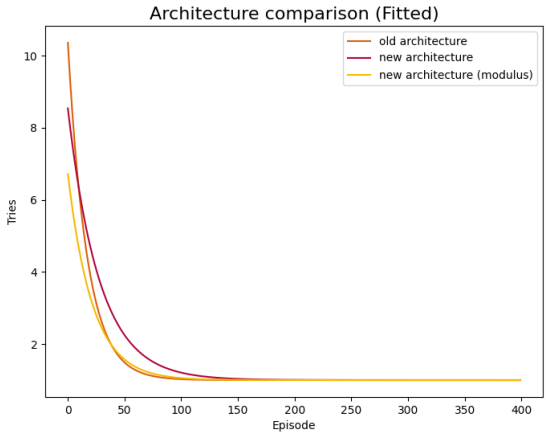


Figure 28: Fitted tries data for Setups for Architecture comparison

Additionally, we will consider the regret and analyze it accordingly after fitting it to a generalized logistic function.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
old architecture	28.976	**68.035**	**107.093**	**146.152**	**185.210**
new architecture	27.739	91.798	155.858	219.917	283.976
new architecture (mod)	**21.023**	71.043	121.064	171.084	221.104

Table 26: Convergence points for Architecture comparison. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.378 (#2)	0.314 (#2)	0.277 (#2)	0.232 (#2)	0.224 (#1)
0.1	0.352 (#2)	0.287 (#2)	0.252 (#2)	0.211 (#2)	0.213 (#2)
0.2	0.204 (#2)	0.131 (#2)	0.108 (#2)	0.083 (#2)	0.079 (#2)
0.3	0.172 (#2)	0.095 (#2)	0.076 (#2)	0.053 (#2)	0.056 (#2)
0.4	0.130 (#2)	0.058 (#2)	0.040 (#2)	0.026 (#2)	0.028 (#2)
0.5	0.106 (#2)	0.030 (#2)	0.022 (#2)	0.013 (#2)	0.011 (#2)

Table 27: Fewest relative outliers for Architecture comparison

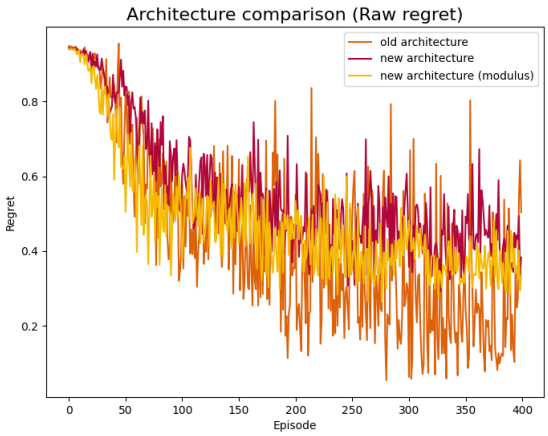


Figure 29: Raw regret data for Setups for Architecture comparison

	0.4	0.5	0.6	0.7	0.8
old architecture	**170.93**	119.34	84.14	57.11	34.98
new architecture	—	179.18	110.13	72.29	43.87
new architecture (mod)	226.37	**107.14**	**67.75**	**43.15**	**25.08**

Table 28: Regret analysis for Architecture comparison. The earliest setup to reach the given threshold is bolded.

Discussion. We can see from the raw tries data in Figure 27 that the old architecture data is extremely noisy, even after converging. While we can see from Table 27 that the new architecture with modulus has the fewest outliers, the old architecture consistently has the most outliers. That is likely because some of the pre-defined injections that it uses present an extreme spike in complexity and

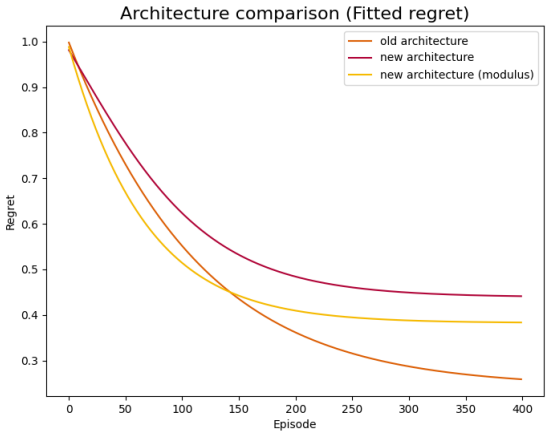


Figure 30: Fitted regret data for Setups for Architecture comparison

difficulty compared to the others. The Gaussian distribution, on the other hand, leads to more evenly balanced complexity, so there are fewer spikes.

When looking at the fitted tries curves in Figure 28, it can be seen that the new architecture performs worse when fitted, with the new architecture with mod performing about as well as the old architecture, although it notably performs better in the early episodes.

Looking at the convergence points in Table 26, we can see that while the new architecture with mod reaches early convergence more quickly, the old architecture converges quicker for more strict ε. This can be explained by the new architecture being steeper and therefore converging sooner.

We will now consider the regret. As we can see from Figure 29, the old architecture's curve is still rather noisy, though not as much so as the one for the tries curve.

The fitted curves in Figure 30 further show that the new architecture without mod performs worse than the one with mod. That is entirely understandable and even intentional, as the repeating injections of the modulus version implies that it can be learned more easily compared to random injections. We can also once again see that the new architecture with modulus initially outperforms the old architecture before being outpaced by it. This could be explained by the difficult injections of the old architecture being even more of a roadblock early on, which the new architecture circumvents.

The analysis in Table 28 further cements this idea, by showing that the new architecture with modulo performs better than the old architecture initially, before being overtaken by the old architecture for $y = 0.4$. Notably, the new architecture without modulo does not even reach $y = 0.4$.

As such, it can overall be said that implementing the new architecture does constitute a performance loss, although it notably vastly improves the outlier frequency. Additionally, what we are

striving for in this case is not best performance, but realistic performance, and the hard-coded injections of the old architecture were not conducive to that.

6 THREATS TO VALIDITY

External validity discusses the situations for which the research assumptions and outcomes might not generalize to a different but relevant setting [58]. One generalizability threat is how representative are differences across failure traces with different root-causes. We mitigated this threat by generating failure traces by randomly selecting components to inject failures. This way, we minimized the risk of favoring a set of components. Note however, that the failure traces are restricted by the architecture, which limits the size of the traces from any given component. Therefore, we cannot claim generalization architectures that have very distinct graphs from the *mRUBiS* .

Internal validity is the most common validity concern [49] and it evaluates if the evidences of our interventions in the experiment were the necessary causes for the observed effects. One validity situation is that any increase in the length of traces during training and testing would have an impact on average regret, similarly the increase in the rate of shops per agent. To mitigate the risk of confounding, we ran controlled experiments that fixed the potential confounder in order to identify the effect of a certain factors, for instance, increasing the size of traces during training or the number of shops while keeping number of agents fixed. We also investigated various levels of control, for instance, testing traces of length 3 and 5 or number of agents 1 and 2. One possible control that we would like to do in the future is the number of unique traces in each training set.

Construct validity concerns the situations for which the operational indicators do not measure the actual concepts (constructs). This might happen through bias in the definitions and methods [57] applied to the constructs. The metrics for detecting convergence, regret, and outliers are the most sensitive to subjective bias. To mitigate that, we combined smoothness techniques (namely sliding window average and goodness to fit) and derived formulations that, given various levels of threshold, allowed to unambiguously compute converge points, outliers, and minimum regret.

Conclusion validity concerns the violations in the assumptions of the statistical methods that we adopted. In our experiment, the most relevant assumption is normality of the data, as we took the average as representative summary statistic. As we can see by the boxplots, the regret data does not follow a Gaussian shape. However, we mitigated this violation by not relying solely on the average regret, but also the convergence in the tries. Because the convergence results of tries showed to be less conservative than the regret, we might have some confidence that the latter is soft lower bound approximation of the overall convergence of the various configurations investigated.

7 ARCHITECTURAL EXTENSIONS

Multi-Armed Bandits Approach. Stochastic Multi-armed Bandits (MAB) [31, 55] aim to discover the arm with the highest reward. As our goal is to find the correct component to fix, we could model the explore & exploit problem as a Stochastic MAB. However, as the ranking of components is subject to mistakes, we would also like to have the components ranked as close as possible to their true priorities. That would allow to go over the component list in an optimal sequence (in case that the component selected is not the one that should be fixed).

Therefore, we would need to optimize both for cumulative reward (choose the top arm) and the correct ranking of candidate fixes. For that, we could apply combinatorial multi-armed bandits (CMAB) [9, 18, 27] whose objective is to maximize the direct sum of rewards obtained from a combination of arms. CMAB would also enable to incorporate fairness constraint [26, 33, 47] as means to prioritize components that have low utility, but high failure rate. Another extension is to allow actions (component fix) to vary by failure type. For that we could apply contextual MAB [2, 5, 11], which could also be made combinatorial [61] and fair [10].

Once we have modeled the component choice as a MAB, we could use an Upper Confidence Bound (CB) algorithm to balance exploration and exploitation. However, for the case that we model a CMAB, we might wish to use the Stochastically Dominant Confidence Bound (SDCB) [8] instead, as it performs better in the CMAB environment.

Neural Network-Based Detection of Underperformance. Underperformance can happen for two reasons: (1) the probability of success for a fix was too far from the actual or (2) the predicted utility of the corresponding component was too far from the actual one. The latter information is promptly available from *mRUBiS* , whereas the former not. Assuming that each agent has learned a policy by training an actor-critic agent, another network can learn whether a given agent policy is performing well or not and recommend a set of measures. For this, we would need to build a loss function that can distinguish whether the problem is in the utility prediction or in the fix success probability prediction. An implementation could be placed at the level of the `RankLearner`, which already fulfills a coordination role of all agents and the total utility. A possible approach to learn this could be the MAB introduced in the previous paragraph.

Asynchronous Failure Injection. Currently, *mRUBiS* forces all agents to operate within the same cycle of failure injection and new failures can only be injected into a shop after all existing failures have been fixed. While this is not a concern in a single agent architecture, with multi-agents this prevents the system from benefiting from agents that have higher failure fixing capabilities, i.e., can fix failures quicker. Because these high-capability agents have to wait for the other agents to produce their fixes, the entire multi-agent system has to operate at the maximum speed of the slowest agent. To overcome this limitation, the communication pattern used by *mRUBiS* has to be relaxed to inject issues not only once every episode, but also to communicate in a multi-threaded manner. Moreover, this limitation also constrains the `MultiAgentController` to communicate with all available agents sequentially instead of in parallel or on-demand, for instance, in blackboard or publisher-subscriber architectural style.

Data Generative Process for Utility. The utility of a component and shop is stochastic, which means that there is a probability distribution that represents those utility values. This distribution is

produced by a data generation process (DGP). Although this process is unknown to the agents, the DGP can be approximated by collecting data and fitting a predictive model. In order to be able to simulate the DGP, one has to select models that allow to be executed in "reverse", i.e., given the outcomes, provide the inputs. Bayesian models are capable of that [19].

Non-stationary utility. Currently, the utility produced by *mRUBiS* is stochastic but stationary. In the future, we might want to relax this assumption of stationarity, as the system could, for example, be overloaded or a shift in the usage pattern could occur. Such a change of the utility is currently not detectable because the implementation uses a synthetic static reward created by using the difference between two steps. For this, the question emerges what the phenomenon of non-stationary is and how it can be identified to mitigate this phenomenon. One approach is to model the non-stationary process by reflecting it as RL hyperparameters, like in Max-Entropy based RL approaches [14, 22].

Multi-Failure Phenomenon. Currently, only one failure per shop can happen, thus only the root-cause of the failure trace needs to be fixed. However, this assumption could be relaxed in the future by allowing multiple components actually to fail - hence, needing to be fixed as well instead of only propagating a failure. To enable this phenomenon, the question arises of how a failing component can be distinguished within a trace and, consequently, be mitigated. The identification of all failing components could be solved by using a combination of symbolic and probabilistic rules that determine multi-graph-style dependencies.

Perfect Failure Masking Phenomenon. The current assumption is that if there is an observation that indicates a failure in a shop, the shop might present a failure. Consequently, we assume that the absence of evidence is evidence of absence. For relaxing this assumption, we need to investigate the latent failures that do not have any indication of a failure in a given shop. These latent failures might happen because of the phenomenon of perfect failure masking, where one failure occludes the other. Therefore, we have to study how this phenomenon can be produced, detected, and mitigated. Currently, the agent is not able to learn any temporal information which could be introduced by using a recurrent neural network. Moreover, the action space has to be increased to include an action that is not doing anything in order to enable the relaxed assumption and request the agent to solve this problem.

8 CONCLUSION AND FUTURE WORK

We developed and evaluated an approach based on the MAPE-K feedback loop to improve the robustness of a multi-agent setup that was trained using the actor-critic algorithm to help *mRUBiS* fix its components. We focused on the behavioral contagion effect as it is the root cause for a rolling update of a real-world scenario. As described in subsection 5.1 we were able to prove that our approach can reduce the loss drastically. This is a stepping stone for robustness in real-world systems.

Our future work will incorporate the extensions introduced in section 7 that can be clustered into (1) the relaxation of assumptions to move *mRUBiS* even closer to a real-world system and (2) the improvement of the existing system.

Regarding (1) we have suggested a couple of changes for the current *mRUBiS* setup. We would like to relax the assumption of synchronous failure injections that is preventing *mRUBiS* from performing faster. Moreover, we want to enable *mRUBiS* to have a non-stationary utility as a shift in the usage pattern could occur for example. Another extension includes multiple failures instead of just one root cause to mimic a real-world system that has several failing components. Regarding multiple failures, we could also allow latent failures that are promoting a perfect failure masking phenomenon that the system is currently not capable of. For cluster (2), we do not have to change *mRUBiS* but the multi-agent architecture. Currently, the system can only detect underperformance for the probability of successfully fixing a component by applying MAPE-K. However, the system could also learn whether the predicted utility is underperforming. The combination of both goals could be solved by applying a MAB approach that is trying to predict the correct ranking of fixes. Another approach supporting the learning process, especially for the uncertainty in utility is a Bayesian model that could be used to develop a predictive model and help the current architecture to be more robust.

We have investigated how the distribution of multiple agents among the shops influences resulting tries and regret in subsection 5.3. In pursuit of this goal, we have performed a multitude of experiments on different such distributions and analyzed the resulting data by fitting it to exponential functions for the tries and generalized logistic functions for the regret. Using these fits, we then were able to approximate the time of convergence, the relative number of outliers post-convergence and the quality of the regret.

We have concluded that, for a given number of shops, it is usually better to keep the number of agents at a minimum. Similarly, if you have a given number of shops per agents, you wish to minimize the total number of shops. When you have a fix number of agents, the optimal number of shops depends on whether you wish to minimize tries or regret. For tries, 10 to 15 shops per agent yields optimal results, whereas 20 shops per agent leads to optimal regret.

Future work will incorporate even more experiments, pushing the scalability further by using more total shops or longer experiments. Additionally, we will include the effect of the future work outlined in section 7 on the data, and investigate how it affects our results.

A APPENDIX

To build the graphs in a reproducible way, we developed the simulation to use a seed when producing the traces. The agents' networks are currently initialized randomly. Nevertheless, the described behavior is reproducible as the networks are converging. To achieve the convergence, we have tested different network configurations for which some did reach the expectations sooner or later. For the results shown in this report, we used the model settings listed in Table 29. The corresponding actor-network is shown in Figure 31 and the critic network in Figure 32. All values were based on intuition and experience and could be different for more complex problems regarding the extensions presented in section 7. Therefore, we could add more layers and dimensions for increasing complexity. The learning rate could be reduced for more complex traces or decreased for simpler ones. As it does not seem that we reached overfitting in

our scenario, we did not add any dropout or other regularization mechanisms.

	Actor	Critic
Number of Layers	4	4
Loss Function	Custom Loss	Mean Squared Error
Optimizer	Adam	Adam
Learning Rate	0.001	0.0005
Other Activations	Relu	Relu
Last Layer Activation	Softmax	Linear

Table 29: Keras Model Settings

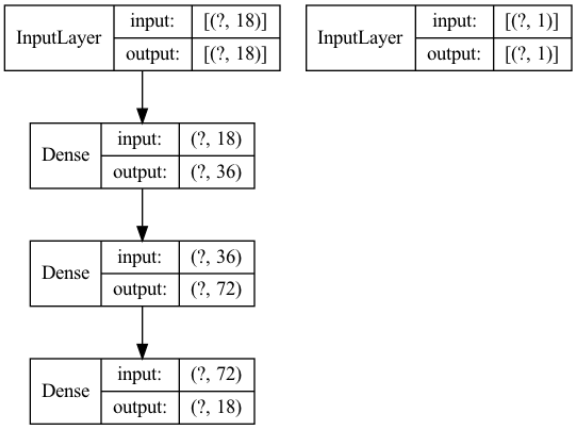


Figure 31: Actor Network

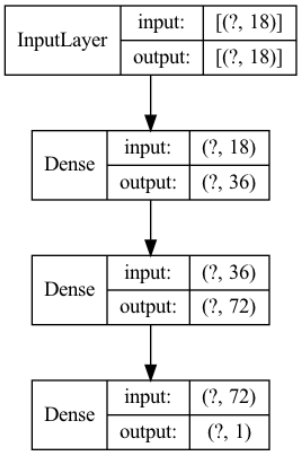


Figure 32: Critic Network

A.1 Additional plots and tables

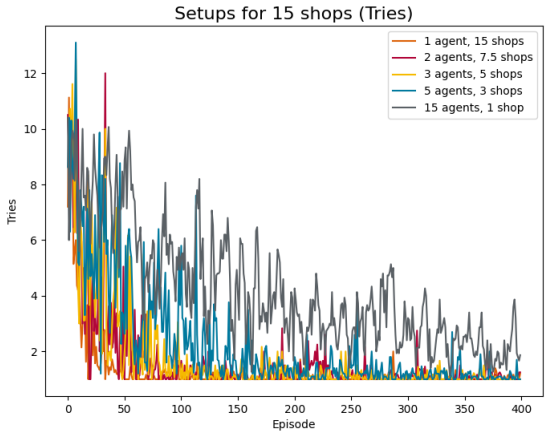


Figure 33: Raw tries data for Setups for 15 shops

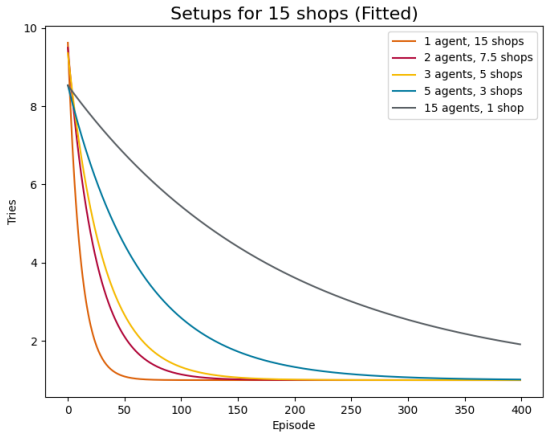


Figure 34: Fitted tries data for Setups for 15 shops

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 15 shops	22.685	**48.087**	**73.488**	**98.889**	**124.291**
2 agents, 7.5 shops	30.485	87.068	143.652	200.235	256.818
3 agents, 5 shops	30.741	102.159	173.576	244.994	316.412
5 agents, 3 shops	10.256	158.157	306.057	453.958	601.859
15 agents, 1 shop	**-174.347**	261.424	697.195	1132.967	1568.738

Table 30: Convergence points for Setups for 15 shops. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.271 (#0)	0.219 (#0)	0.212 (#0)	0.189 (#0)	0.182 (#0)
0.1	0.271 (#0)	0.219 (#0)	0.212 (#0)	0.189 (#0)	0.182 (#0)
0.2	0.146 (#0)	0.097 (#0)	0.092 (#0)	0.076 (#0)	0.076 (#0)
0.3	0.093 (#0)	0.046 (#0)	0.043 (#0)	0.030 (#0)	0.029 (#0)
0.4	0.058 (#0)	0.023 (#0)	0.018 (#0)	0.013 (#0)	0.011 (#0)
0.5	0.042 (#0)	0.017 (#0)	0.015 (#0)	0.010 (#0)	0.007 (#0)

Table 31: Fewest relative outliers for Setups for 15 shops

	0.4	0.5	0.6	0.7	0.8
1 agent, 15 shops	**248.63**	**103.27**	**64.41**	**40.73**	**23.50**
2 agents, 7.5 shops	—	199.28	123.38	80.35	48.62
3 agents, 5 shops	—	257.74	161.10	109.71	69.69
5 agents, 3 shops	—	—	279.73	186.62	125.38
15 agents, 1 shop	—	—	—	—	382.85

Table 32: Regret analysis for Setups for 15 shops. The earliest setup to reach the given threshold is bolded.

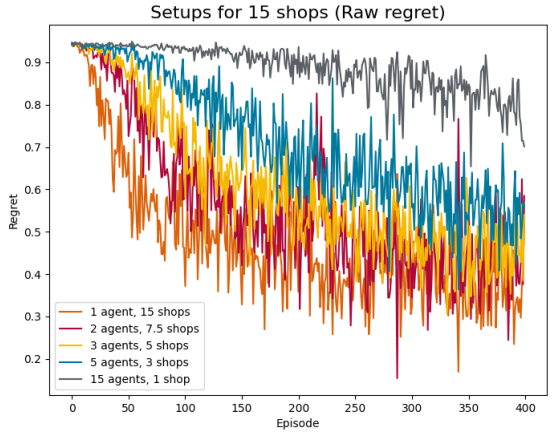


Figure 35: Raw regret data for Setups for 15 shops

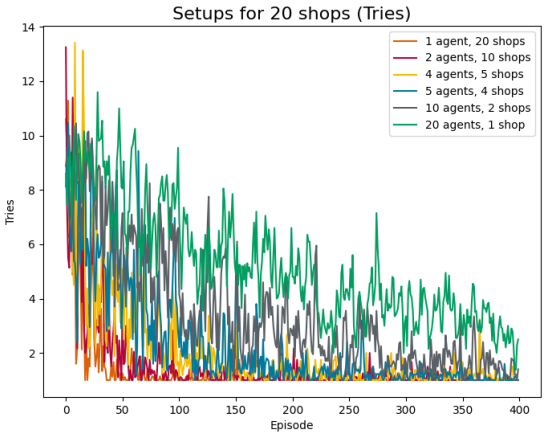


Figure 37: Raw tries data for Setups for 20 shops

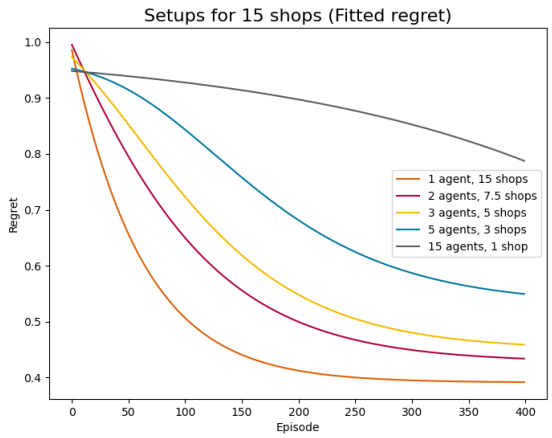


Figure 36: Fitted regret data for Setups for 15 shops

Figure 38: Raw regret data for Setups for 20 shops

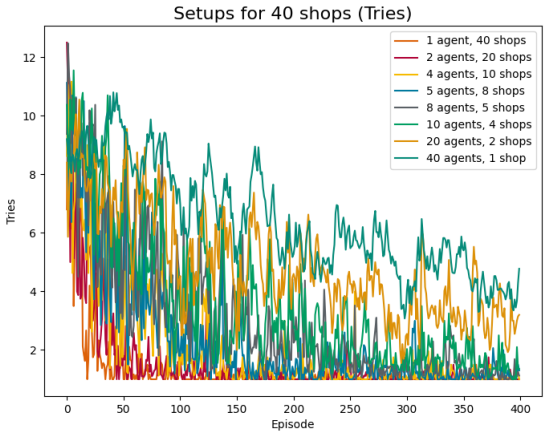


Figure 39: Raw tries data for Setups for 40 shops

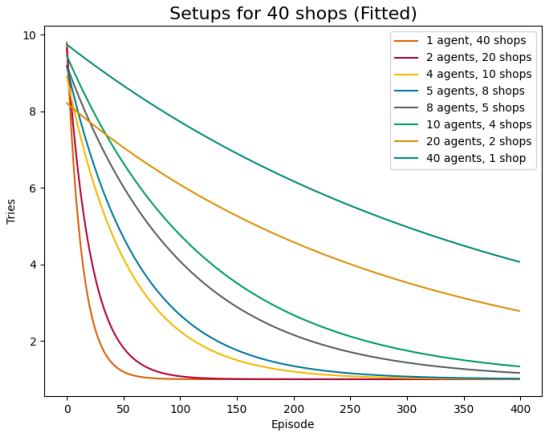


Figure 40: Fitted tries data for Setups for 40 shops

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
1 agent, 40 shops	24.957	**55.149**	**85.341**	**115.533**	**145.724**
2 agents, 20 shops	29.801	78.607	127.413	176.220	225.026
4 agents, 10 shops	20.467	145.392	270.316	395.240	520.164
5 agents, 8 shops	16.226	161.646	307.067	452.487	597.907
8 agents, 5 shops	-22.724	212.711	448.145	683.580	919.014
10 agents, 4 shops	-47.101	237.648	522.398	807.147	1091.896
20 agents, 2 shops	-393.683	264.500	922.682	1580.865	2239.047
40 agents, 1 shop	**-560.863**	316.479	1193.820	2071.162	2948.503

Table 33: Convergence points for Setups for 40 shops. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.272 (#0)	0.230 (#0)	0.194 (#0)	0.176 (#0)	0.165 (#0)
0.1	0.272 (#0)	0.230 (#0)	0.194 (#0)	0.176 (#0)	0.165 (#0)
0.2	0.147 (#0)	0.096 (#0)	0.070 (#0)	0.056 (#0)	0.043 (#0)
0.3	0.123 (#0)	0.073 (#0)	0.048 (#0)	0.000 (#2)	0.028 (#0)
0.4	0.091 (#0)	0.055 (#0)	0.035 (#0)	0.000 (#2)	0.016 (#0)
0.5	0.059 (#0)	0.029 (#0)	0.016 (#0)	0.000 (#2)	0.008 (#0)

Table 34: Fewest relative outliers for Setups for 40 shops

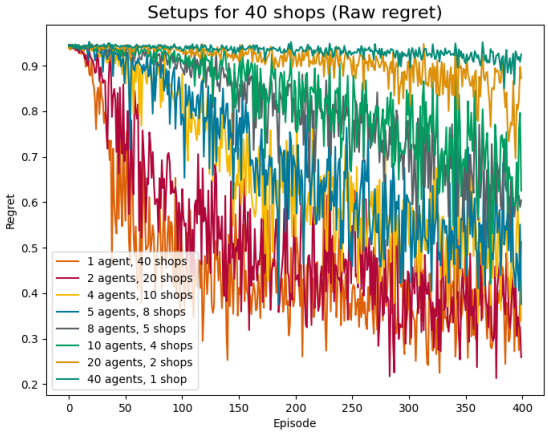


Figure 41: Raw regret data for Setups for 40 shops

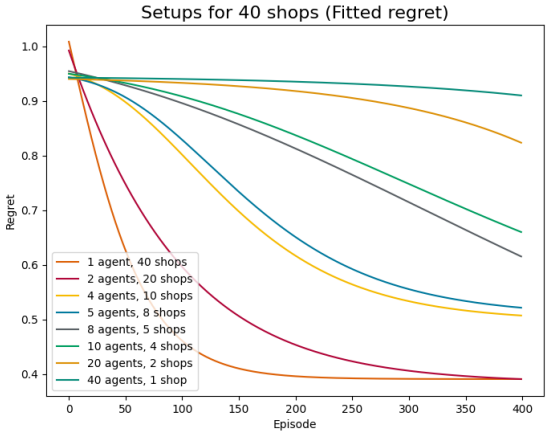


Figure 42: Fitted regret data for Setups for 40 shops

	0.4	0.5	0.6	0.7	0.8
1 agent, 40 shops	**179.13**	**82.35**	**55.22**	**37.67**	**23.94**
2 agents, 20 shops	328.21	154.67	98.45	63.22	37.17
4 agents, 10 shops	—	—	213.42	148.77	100.72
5 agents, 8 shops	—	—	242.11	168.80	115.30
8 agents, 5 shops	—	—	—	314.48	213.79
10 agents, 4 shops	—	—	—	352.92	243.14
20 agents, 2 shops	—	—	—	—	—
40 agents, 1 shop	—	—	—	—	—

Table 35: Regret analysis for Setups for 40 shops. The earliest setup to reach the given threshold is bolded.

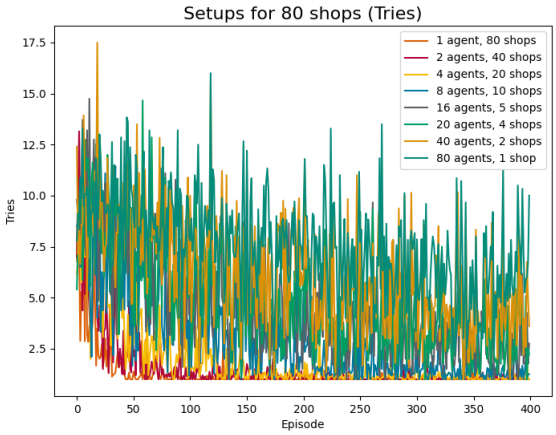


Figure 43: Raw tries data for Setups for 80 shops

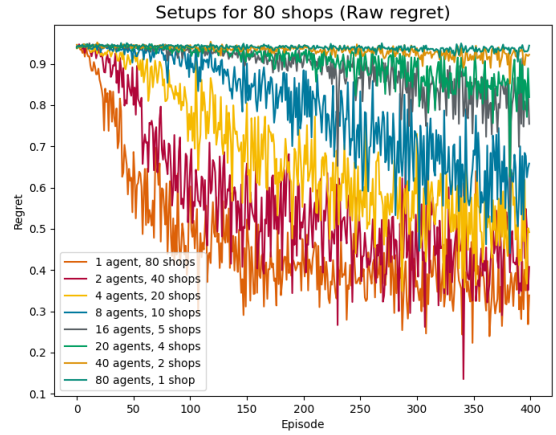


Figure 44: Raw regret data for Setups for 80 shops

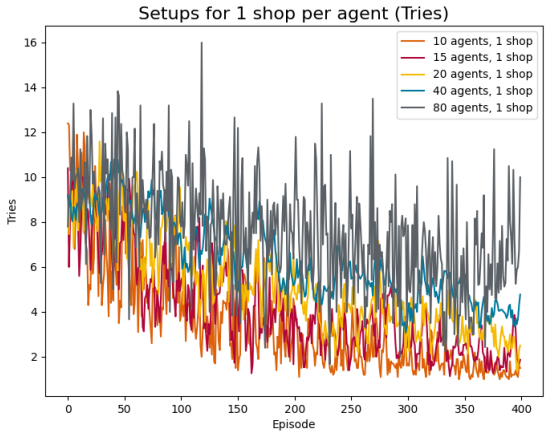


Figure 45: Raw tries data for Setups for 1 shop per agent

Figure 46: Raw regret data for Setups for 1 shop per agent

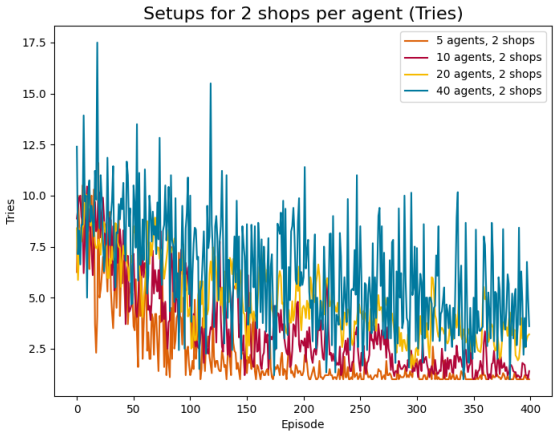


Figure 47: Raw tries data for Setups for 2 shops per agent

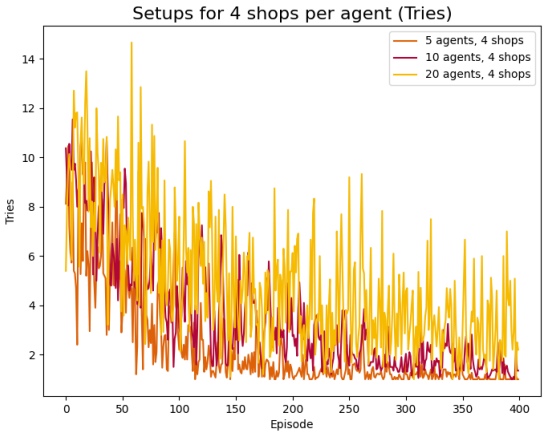


Figure 49: Raw tries data for Setups for 4 shops per agent

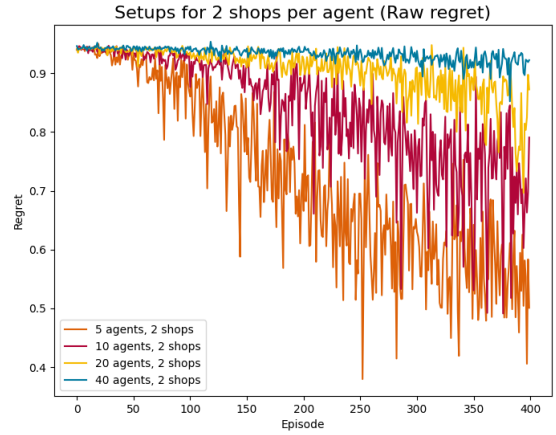


Figure 48: Raw regret data for Setups for 2 shops per agent

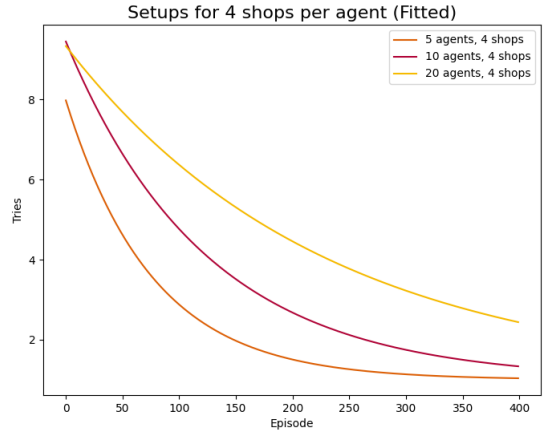


Figure 50: Fitted tries data for Setups for 4 shops per agent

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
5 agents, 4 shops	-6.630	**168.688**	**344.006**	**519.324**	**694.642**
10 agents, 4 shops	-47.101	237.648	522.398	807.147	1091.896
20 agents, 4 shops	**-227.733**	295.350	818.434	1341.517	1864.600

Table 36: Convergence points for Setups for 4 shops per agent. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.820 (#0)	0.697 (#0)	0.345 (#0)	—	—
0.1	0.720 (#0)	0.537 (#0)	0.255 (#0)	—	—
0.2	0.613 (#0)	0.368 (#0)	0.164 (#0)	—	—
0.3	0.547 (#0)	0.286 (#0)	0.091 (#0)	—	—
0.4	0.470 (#0)	0.199 (#0)	0.073 (#0)	—	—
0.5	0.440 (#0)	0.156 (#0)	0.055 (#0)	—	—

Table 37: Fewest relative outliers for Setups for 4 shops per agent

	0.4	0.5	0.6	0.7	0.8
5 agents, 4 shops	—	—	**270.19**	**184.08**	**124.89**
10 agents, 4 shops	—	—	—	352.92	243.14
20 agents, 4 shops	—	—	—	—	—

Table 38: Regret analysis for Setups for 4 shops per agent. The earliest setup to reach the given threshold is bolded.

Figure 51: Raw regret data for Setups for 4 shops per agent

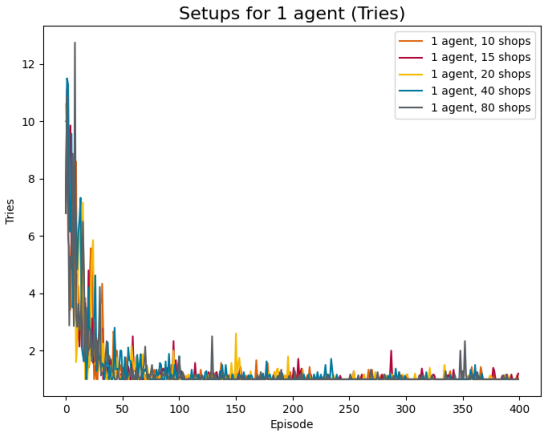


Figure 53: Raw tries data for Setups for 1 agent

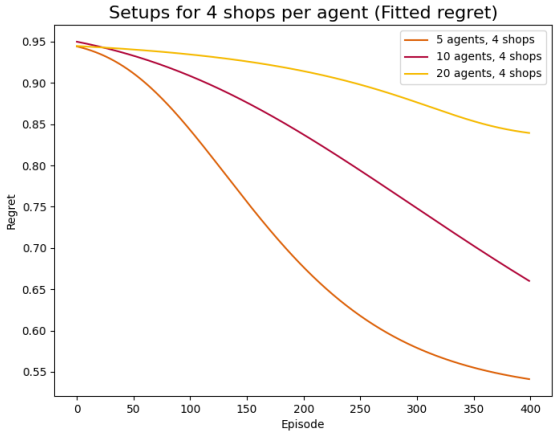


Figure 52: Fitted regret data for Setups for 4 shops per agent

Figure 54: Raw regret data for Setups for 1 agent

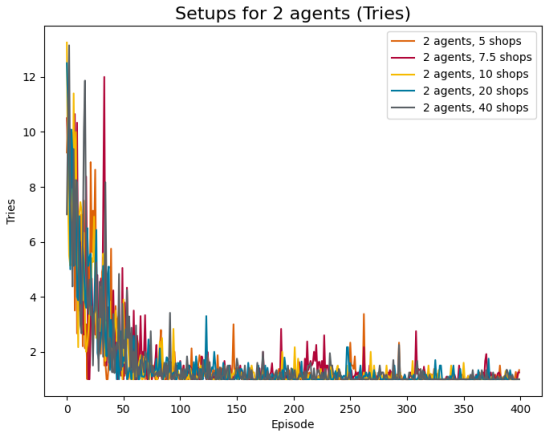


Figure 55: Raw tries data for Setups for 2 agents

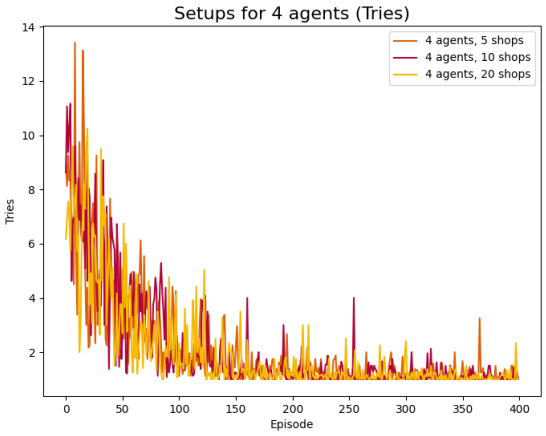


Figure 57: Raw tries data for Setups for 4 agents

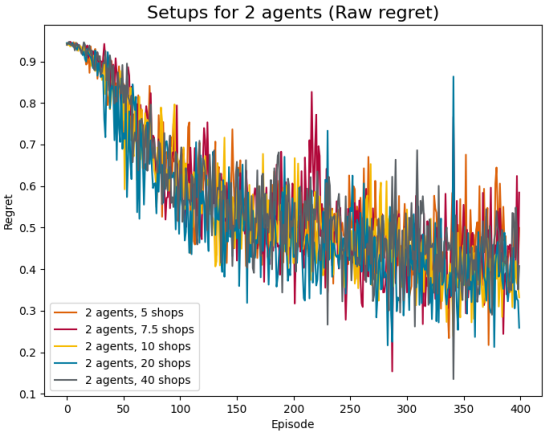


Figure 56: Raw regret data for Setups for 2 agents

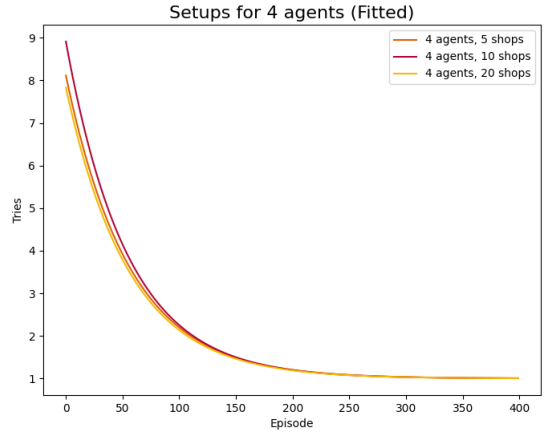


Figure 58: Fitted tries data for Setups for 4 agents

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
4 agents, 5 shops	13.307	142.538	271.769	401.000	530.232
4 agents, 10 shops	20.467	145.392	270.316	**395.240**	**520.164**
4 agents, 20 shops	**11.353**	**139.681**	**268.009**	396.337	524.665

Table 39: Convergence points for Setups for 4 agents. The earliest convergence for each ε is bolded.

	-10^{-1}	-10^{-2}	-10^{-3}	-10^{-4}	-10^{-5}
0.0	0.688 (#2)	0.562 (#2)	0.458 (#2)	0.333 (#2)	—
0.1	0.620 (#1)	0.449 (#1)	0.366 (#2)	0.333 (#2)	—
0.2	0.508 (#2)	0.315 (#2)	0.237 (#2)	0.333 (#2)	—
0.3	0.417 (#1)	0.205 (#1)	0.130 (#2)	0.000 (#1)	—
0.4	0.363 (#2)	0.142 (#2)	0.069 (#2)	0.000 (#1)	—
0.5	0.306 (#1)	0.087 (#1)	0.038 (#2)	0.000 (#1)	—

Table 40: Fewest relative outliers for Setups for 4 agents

Figure 59: Raw regret data for Setups for 4 agents

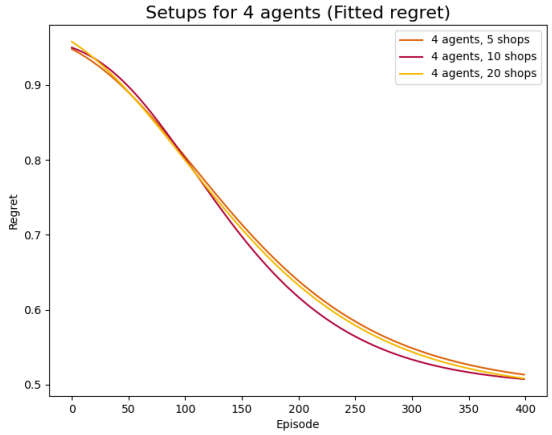


Figure 60: Fitted regret data for Setups for 4 agents

	0.4	0.5	0.6	0.7	0.8
4 agents, 5 shops	—	—	233.43	158.25	102.26
4 agents, 10 shops	—	—	**213.42**	**148.77**	100.72
4 agents, 20 shops	—	—	228.10	154.50	**99.67**

Table 41: Regret analysis for Setups for 4 agents. The earliest setup to reach the given threshold is bolded.

REFERENCES

[1] 2022. GitHub Robust MARL. https://github.com/hpi-sam/robust-marl4sas. Accessed: 2022-03-31.

[2] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. 2014. Taming the monster: A fast and simple algorithm for contextual bandits. In *International Conference on Machine Learning*. PMLR, 1638–1646.

[3] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. 2013. *Software Engineering Processes for Self-Adaptive Systems*. 51–75. https://doi.org/10.1007/978-3-642-35813-5_3

[4] Matthew Botvinick, Sam Ritter, Jane X Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. 2019. Reinforcement learning, fast and slow. *Trends in cognitive sciences* 23, 5 (2019), 408–422.

[5] Djallel Bouneffouf, Irina Rish, and Charu Aggarwal. 2020. Survey on applications of multi-armed and contextual bandits. In *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–8.

[6] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antónia Lopes, and Michael Smit. 2013. A Design Space for Self-Adaptive Systems. (01 2013). https://doi.org/10.1007/978-3-642-35813-5_2

[7] Pietro Buzzega, Matteo Boschini, Angelo Porrello, and Simone Calderara. 2020. Rethinking Experience Replay: a Bag of Tricks for Continual Learning. https://doi.org/10.48550/ARXIV.2010.05595

[8] Wei Chen, Wei Hu, Fu Li, Jian Li, Yu Liu, and Pinyan Lu. 2016. Combinatorial Multi-Armed Bandit with General Reward Functions. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/aa169b49b583a2b5af89203c2b78c67c-Paper.pdf

[9] Wei Chen, Yajun Wang, Yang Yuan, and Qinshi Wang. 2016. Combinatorial multi-armed bandit and its extension to probabilistically triggered arms. *The Journal of Machine Learning Research* 17, 1 (2016), 1746–1778.

[10] Yifang Chen, Alex Cuellar, Haipeng Luo, Jignesh Modi, Heramb Nemlekar, and Stefanos Nikolaidis. 2020. Fair contextual multi-armed bandits: Theory and experiments. In *Conference on Uncertainty in Artificial Intelligence*. PMLR, 181–190.

[11] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. 2011. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 208–214.

[12] Sven Gronauer Klaus Diepold. 2021. *Multi-agent deep reinforcement learning: a survey*. Artificial Intelligence Review. https://doi.org/10.1007/s10462-021-09996-w

[13] Gabriel Dulac-Arnold, Nir Levine, Daniel J. Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. 2020. An empirical investigation of the challenges of real-world reinforcement learning. https://doi.org/10.48550/ARXIV.2003.11881

[14] Benjamin Eysenbach and Sergey Levine. 2021. Maximum entropy rl (provably) solves some robust rl problems. *arXiv preprint arXiv:2103.06257* (2021).

[15] Yong Feng, Jinglong Chen, Jingsong Xie, Tianci Zhang, Haixin Lv, and Tongyang Pan. 2021. Meta-Learning as a Promising Approach for Few-Shot Cross-Domain Fault Diagnosis: Algorithms, Applications, and Prospects. *Knowledge-Based Systems* (2021), 107646.

[16] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. 2017. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*. PMLR, 1146–1155.

[17] Michal Folwarczny, Nils Larsen, Tobias Otterbring, Agata Gasiorowska, and Valdimar Sigurdsson. 2022. Viral Viruses and Modified Mobility: Cyberspace Disease Salience Predicts Human Movement Patterns. https://doi.org/10.31234/osf.io/xtks7

[18] Yi Gai, Bhaskar Krishnamachari, and Rahul Jain. 2010. Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation. In *2010 IEEE Symposium on New Frontiers in Dynamic Spectrum (DySPAN)*. IEEE, 1–9.

[19] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. 2013. *Bayesian data analysis*. Chapman and Hall/CRC.

[20] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *CoRR* abs/2103.04112 (2021). arXiv:2103.04112 https://arxiv.org/abs/2103.04112

[21] Dibya Ghosh, Jad Rahme, Aviral Kumar, Amy Zhang, Ryan P. Adams, and Sergey Levine. 2021. Why Generalization in RL is Difficult: Epistemic POMDPs and Implicit Partial Observability. https://doi.org/10.48550/ARXIV.2107.06277

[22] Seungyul Han and Youngchul Sung. 2021. A Max-Min Entropy Framework for Reinforcement Learning. *Advances in Neural Information Processing Systems* 34 (2021).

[23] Matthew Hausknecht and Peter Stone. 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. https://doi.org/10.48550/ARXIV.1507.06527

[24] Research Institute. 2006. *The AI-powered enterprise: Unlocking the potential of AI at scale.* Technical Report. Capgemini.

[25] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. 2019. ExplainIt! – A Declarative Root-Cause Analysis Engine for Time Series Data (Extended Version). *Proceedings of the 2019 International Conference on Management of Data* (June 2019), 333–348. https://doi.org/10.1145/3299869.3314048 arXiv:1903.08132

[26] Matthew Joseph, Michael Kearns, Jamie H Morgenstern, and Aaron Roth. 2016. Fairness in learning: Classic and contextual bandits. *Advances in neural information processing systems* 29 (2016).

[27] Julian Katz-Samuels, Lalit Jain, Kevin G Jamieson, et al. 2020. An empirical process approach to the union bound: Practical algorithms for combinatorial and linear bandits. *Advances in Neural Information Processing Systems* 33 (2020), 10371–10382.

[28] Jeffrey Kephart and D.M. Chess. 2003. The Vision Of Autonomic Computing. *Computer* 36 (02 2003), 41 – 50. https://doi.org/10.1109/MC.2003.1160055

[29] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. 2021. A Survey of Generalisation in Deep Reinforcement Learning. https://doi.org/10.48550/ARXIV.2111.09794

[30] Vijay Konda and John Tsitsiklis. 1999. Actor-critic algorithms. *Advances in neural information processing systems* 12 (1999).

[31] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit algorithms.* Cambridge University Press.

[32] 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. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software engineering for self-adaptive systems II.* Springer, 1–32.

[33] Fengjiao Li, Jia Liu, and Bo Ji. 2019. Combinatorial sleeping bandits with fairness constraints. *IEEE Transactions on Network Science and Engineering* 7, 3 (2019), 1799–1813.

[34] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2019. Continuous Control with Deep Reinforcement Learning. *arXiv:1509.02971 [cs, stat]* (July 2019). arXiv:1509.02971 [cs, stat]

[35] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8, 3 (1992), 293–321.

[36] Shuai Lin, Limin Jia, Hengrun Zhang, and Pengzhu Zhang. 2021. Network approach to modelling and analysing failure propagation in high-speed train systems. *International Journal of Systems Science: Operations & Logistics* (2021), 1–17.

[37] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. 2020. Localizing Failure Root Causes in a Microservice through Causality Inference. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS).* IEEE, 1–10.

[38] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. 2021. A graph placement methodology for fast chip design. *Nature* 594 (06 2021), 207–212. https://doi.org/10.1038/s41586-021-03544-w

[39] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]* (Dec. 2013). arXiv:1312.5602 [cs]

[40] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. 2020. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712* (2020).

[41] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. 2018. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems* 31 (2018).

[42] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, Deeper, Easier: Crowdsourcing Diagnosis of Microservice Kernel Failure from User Space. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 646–657.

[43] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive Neural Networks. https://doi.org/10.48550/ARXIV.1606.04671

[44] Dominik Scheinert, Alexander Acker, Lauritz Thamsen, Morgan K. Geldenhuys, and Odej Kao. 2021. Learning Dependencies in Distributed Cloud Applications to Identify and Localize Anomalies. *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)* (May 2021), 7–12. https://doi.org/10.1109/CloudIntelligence52565.2021.00011 arXiv:2103.05245

[45] Sebastian Schmidt, Christoph Benke, and Christiane A. Pané-Farré. 2021. Purchasing under threat: Changes in shopping patterns during the COVID-19 pandemic. https://doi.org/10.1371/journal.pone.0253231

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]* (Aug. 2017). arXiv:1707.06347 [cs]

[47] Candice Schumann, Zhi Lang, Nicholas Mattei, and John P Dickerson. 2019. Group fairness in bandit arm selection. *arXiv preprint arXiv:1912.03802* (2019).

[48] Max Schwarzer, Nitarshan Rajkumar, Michael Noukhovitch, Ankesh Anand, Laurent Charlin, R Devon Hjelm, Philip Bachman, and Aaron C Courville. 2021. Pretraining representations for data-efficient reinforcement learning. *Advances in Neural Information Processing Systems* 34 (2021).

[49] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on internal and external validity in empirical software engineering. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 9–19.

[50] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. https://doi.org/10.48550/ARXIV.1712.01815

[51] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[52] David Sinreich. 2006. An architectural blueprint for autonomic computing.

[53] Jacopo Soldani, Giuseppe Montesano, and Antonio Brogi. 2021. What Went Wrong? Explaining Cascading Failures in Microservice-Based Applications. In *Symposium and Summer School on Service-Oriented Computing.* Springer, 133–153.

[54] Adam Stooke, Kimin Lee, Pieter Abbeel, and Michael Laskin. 2021. Decoupling representation learning from reinforcement learning. In *International Conference on Machine Learning.* PMLR, 9870–9879.

[55] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[56] Thomas Vogel. 2018. mRUBiS: An exemplar for model-based architectural self-healing and self-optimization. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems.* 101–107.

[57] Roel J Wieringa. 2014. *Design science methodology for information systems and software engineering.* Springer.

[58] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Bjorn Regnell, and Anders Wessl'en. 2012. *Experimentation in software engineering.* Springer.

[59] Chuanli Wu, Zixiang Wang, Siwei Zhou, Dongdong Zhao, Jing Tian, and Jianwen Xiang. 2021. Reliability Analysis of Systems Subject to Imperfect Fault Coverage Considering Failure Propagation and Component Relevancy. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE, 210–217.

[60] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium.* IEEE, 1–9.

[61] Chien-Sheng Yang, Ramtin Pedarsani, and A Salman Avestimehr. 2021. Edge computing in the dark: Leveraging contextual-combinatorial bandit and coded computing. *IEEE/ACM Transactions on Networking* 29, 3 (2021), 1022–1031.

[62] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P. Xing, Laurent El Ghaoui, and Michael I. Jordan. 2019. Theoretically Principled Trade-off between Robustness and Accuracy. https://doi.org/10.48550/ARXIV.1901.08573