

Institute of Software Technology  
Reliable Software Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Bachelorarbeit

# Algorithms for Efficient Chaos Experiment Selection in Microservice Architectures

Niklas Kammhoff

<b>Course of Study:</b>	Softwaretechnik
<b>Examiner:</b>	Dr.-Ing. André van Hoorn
<b>Supervisor:</b>	<i>University of Stuttgart:</i> M.Sc. Thomas F. Düllmann Dr.-Ing. André van Hoorn  <i>Monash University:</i> Dr. Aldeida Aleti
<b>Commenced:</b>	January 24, 2019
<b>Completed:</b>	July 24, 2019
<b>CR-Classification:</b>	I.7.2



# Abstract

Chaos Engineering is an approach to resilience benchmarking for distributed architectures. Existing approaches for experiment selection include random fault injections and lineage driven fault injection, however, while quite exhaustive, they do lack in the area of efficiency. The ORCAS project introduces a new approach to resilience benchmarking for microservice architectures by leveraging knowledge about the architecture, as well as the simulation of experiments outside of the production environment. By prioritizing certain experiments, instead of choosing at random, the goal is to provide an efficient alternative to the existing approaches. As one of the main components of the ORCAS project, the decision engine is responsible for experiment selection based on the inputs of the architecture and previous experiment results. An outline for integrating the results into this framework is presented.

This thesis explores algorithms suitable for efficient experiment selection in the decision engine, and evaluates these algorithms. Algorithms presented are based on Bayesian networks, reinforcement learning, artificial neural networks, and multilevel feedback queues.

For evaluation, the algorithms are evaluated based on their efficiency and fault detection rate. For this, five versions of an example architecture have been used. The five architectures are grouped into three equivalence groups, based on their level of resilience. One candidate of each group is then used for evaluation.

The results show that reinforcement learning is a suitable approach for the problem of efficiently selecting fault injections, since it does not require a training phase. In any case, almost all algorithms improve on the results of random fault injection selection.



## Kurzfassung

Chaos Engineering beschreibt die Idee der Einführung von Fehlern in laufende Systeme. Das System wird dann, basierend auf einer vorher definierten Metrik, beobachtet. Diese Beobachtung zeigt, ob das System sich verhält, wie erwartet. Durch diese Einführung von künstlichen Fehlern ist es möglich Systeme auf ihre Widerstandsfähigkeit gegen unerwartete Ereignisse zu untersuchen.

Als Teil des ORCAS Projekts generiert die Orcas Decision Engine Chaos Experimente, welche dann sowohl in einem Simulator, als auch in einem realen System für Tests der Widerstandsfähigkeit verwendet werden sollen. Die Ergebnisse dieser Experimente werden dann in der Orcas Decision Engine verwendet, um die folgende Auswahl von Chaos Experimenten zu verbessern. Neben den Experimentergebnissen sind auch Informationen über die Microservice-Architektur, wie zum Beispiel Abhängigkeiten zwischen Operationen, bekannt.

Diese Arbeit präsentiert Algorithmen zur Wahl von Fehlern, die dann in Chaos Experimenten verwendet werden können. Diese Algorithmen kommen aus den Bereichen Bayes'sche Netze, Reinforcement Learning und Multilevel Feedback Queues.

In einer experimentellen Auswertung werden die Algorithmen mit einer zufälligen Wahl von Fehlern verglichen. Metriken zum Vergleich der Effizienz und Fehlererkennungsrate werden definiert.

Die Ergebnisse zeigen, dass Reinforcement Learning eine passende Herangehensweise für die Wahl von Fehlern ist. Im Vergleich zu Bayes'sche Netze ist hier keine Trainingphase notwendig. Fast alle Algorithmen zeigen deutliche Verbesserungen im Vergleich zu der zufälligen Wahl von Fehlern.



# Contents

---

1	Introduction	1
2	Foundations	7
2.1	Microservice Architectures . . . . .	7
2.2	Resilience Patterns and Antipatterns . . . . .	8
2.3	Fault Injections . . . . .	10
2.4	Chaos Engineering . . . . .	11
2.5	Test Suite Optimization . . . . .	13
2.6	ORCAS Project . . . . .	15
2.7	Algorithms . . . . .	20
2.8	Kolmogorov-Smirnov Test . . . . .	30
3	Related Work	31
3.1	Lineage Driven Fault Injection . . . . .	31
3.2	Reinforcement Learning for Test Case Prioritization and Selection in Continuous Integration . . . . .	33
4	Algorithms	35
4.1	Example Architecture . . . . .	36
4.2	Framework . . . . .	37
4.3	Bayesian Network . . . . .	42
4.4	Reinforcement Learning . . . . .	52
4.5	Multilevel Feedback Queue . . . . .	71
5	ORCAS Decision Engine	75
5.1	Chaos Experiments . . . . .	75
5.2	Integration with MiSim Simulator . . . . .	78
6	Evaluation	81
6.1	Experiment Settings . . . . .	81

6.2	Description of Results . . . . .	90
6.3	Discussion of Results . . . . .	103
6.4	Threats to Validity . . . . .	106
7	Conclusion	109
	Bibliography	113



# List of Figures

---

1.1	The ORCAS framework . . . . .	2
2.1	Overview of antipatterns and patterns . . . . .	9
2.2	States of a fault injection experiment . . . . .	11
2.3	ORCAS overview . . . . .	15
2.4	Agent-environment interaction in a MDP . . . . .	21
2.5	3-layer feed-forward network . . . . .	28
2.6	Multilevel Feedback Queue example . . . . .	29
2.7	KS-Test Comparison of Cumulative Fraction Plots . . . . .	30
3.1	Lineage Driven Fault Injection . . . . .	32
4.1	Architecture used as example in this chapter . . . . .	37
4.2	UML class diagram for strategy pattern . . . . .	38
4.3	UML class diagram of interface for algorithms . . . . .	39
4.4	Sequence diagram visualizing interaction between components . . . . .	41
4.5	Bayesian network graph . . . . .	44
4.6	Bayesian network graph . . . . .	49
4.7	Dependency graph for example architecture . . . . .	62
4.8	MLFQ for example architecture after initialization . . . . .	74
4.9	MLFQ for example architecture after multiple iterations . . . . .	74
6.1	Reference architecture . . . . .	82
6.2	<i>arch01</i> . . . . .	84
6.3	<i>arch02</i> . . . . .	84
6.4	<i>arch03</i> . . . . .	85
6.5	<i>arch04</i> . . . . .	85
6.6	<i>arch05</i> . . . . .	86
6.7	ER metric for <i>arch01</i> , line graph . . . . .	92
6.8	FDR metric for <i>arch01</i> , line graph . . . . .	93
6.9	FDR metric for <i>arch01</i> , box graph . . . . .	93

6.10	WFDR metric for <i>arch01</i> , box graph . . . . .	94
6.11	ER metric for <i>arch02</i> , line graph . . . . .	95
6.12	FDR metric for <i>arch02</i> , line graph . . . . .	96
6.13	FDR metric for <i>arch02</i> , box graph . . . . .	96
6.14	WFDR metric for <i>arch02</i> , box graph . . . . .	97
6.15	ER metric for <i>arch05</i> , line graph . . . . .	98
6.16	FDR metric for <i>arch05</i> , line graph . . . . .	99
6.17	FDR metric for <i>arch05</i> , box graph . . . . .	99
6.18	WFDR metric for <i>arch05</i> , box graph . . . . .	100
6.19	Average times for <i>get_experiment</i> . . . . .	101
6.20	Average times for <i>enter_result</i> . . . . .	102

# List of Tables

---

4.1	Probabilities for variables . . . . .	46
4.2	Observed results after 100 iterations . . . . .	46
4.3	Probabilities after training . . . . .	47
4.4	Variables for fault injections . . . . .	51
4.5	Probabilities after initialization . . . . .	51
4.6	Observed results after 100 iterations . . . . .	52
4.7	Probabilities after training . . . . .	52
4.8	Initial estimated rewards . . . . .	56
4.9	Estimated rewards after multiple iterations . . . . .	56
4.10	Initial estimated rewards for actions . . . . .	59
4.11	Estimated rewards for actions after multiple iterations . . . . .	59
4.12	States and actions for example architecture . . . . .	62
4.13	Feature extractions of fault injections . . . . .	67
4.14	Extracted features for fault injection . . . . .	71
4.15	Priority groups for features . . . . .	71
6.1	Example setup runs . . . . .	88
6.2	Parameters used for algorithms . . . . .	90
6.3	Reference names for algorithms . . . . .	91
6.4	Counter results for <i>arch01</i> . . . . .	94
6.5	Counter results for <i>arch02</i> . . . . .	97
6.6	Counter results for <i>arch05</i> . . . . .	100



# List of Algorithms

---

1	$\epsilon$ -greedy k-armed Bandit Algorithm . . . . .	24
2	Optimistic Initial Value Algorithm . . . . .	25
3	Q-Learning Algorithm . . . . .	26
4	Low Level Bayesian Network Algorithm . . . . .	45
5	High Level Bayesian Network Algorithm . . . . .	50
6	$\epsilon$ -greedy Bandit Algorithm . . . . .	55
7	Optimistic Initial Value Algorithm . . . . .	58
8	Q-Learning Algorithm . . . . .	63
9	Artificial Neural Network Algorithm . . . . .	66
10	Tableau Algorithm . . . . .	70
11	MLFQ Algorithm . . . . .	73



## Chapter 1

# Introduction

---

This work is a bachelor's thesis, part of a Bachelor of Science Software Engineering degree at the University of Stuttgart, and written at the Reliable Software Systems Group.

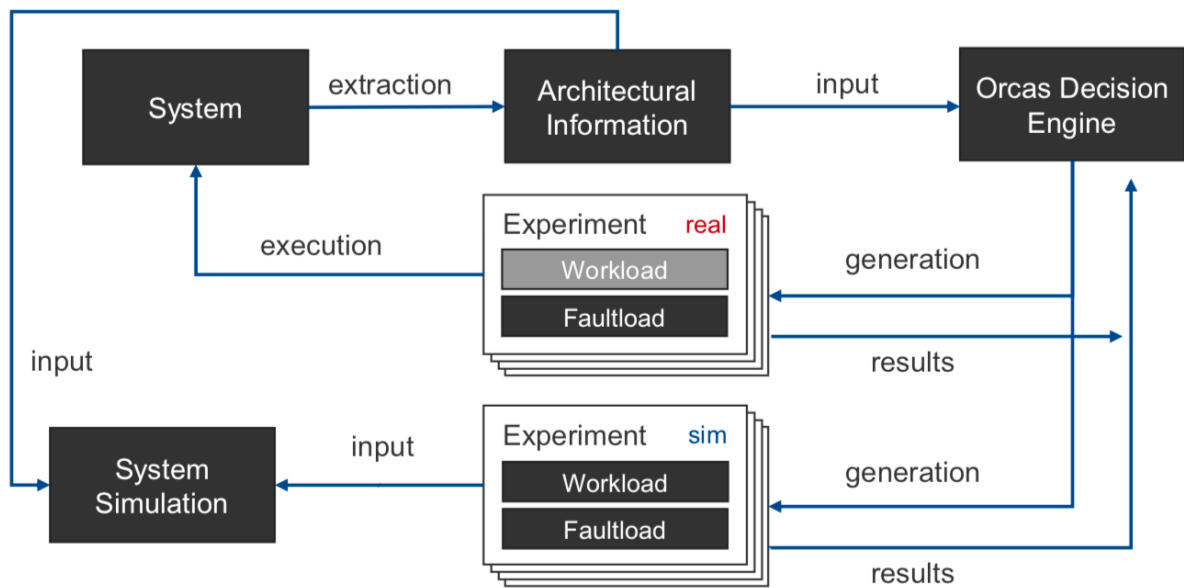
This chapter will motivate and define the problem, present the goals of this thesis, summarize the results, and provide the structure of this document.

## Motivation

Over the last years, monolithic architectures have been starting to lose in popularity. Instead, architectures made up of smaller, self-contained microservices are starting to gain in popularity [New15]. With this come a vast array of advantages.

Each microservice can use its own languages and technologies. Each microservice can be run by a different team. Each microservice can focus on one task, reducing the average complexity of the code in each microservice. Each microservice can communicate with those, and only those, microservices it needs to.

But, what if one of those microservices fails? And all the other microservices are waiting for a response of one microservice, flooding it with requests the second it comes back online? An entire system collapses, because one small, self-contained microservice failed. To prevent this, the *design for failure* principle is essential. Designing for failure in microservice architectures generally involves the usage of resilience patterns. Patterns like *circuit breaker* prevent other services waiting for a failed service, allowing them to take an alternative action, instead of being stuck. However, here comes the next problem. How does one verify that a system is resilient to faults?



**Figure 1.1:** The ORCAS framework [HADP18]

Netflix introduced chaos engineering [RHB+17], with the idea of shutting instances of random services, simulating failures. These so-called chaos experiments were performed on live production systems, as only there real traffic existed, and real circumstances provided real results on the resilience of the architecture.

However, injecting random faults is not very efficient, as each microservice is self-contained, implemented independent of other microservices, and as such each microservice has a varying degree of fault tolerance.

The ORCAS project [HADP18] challenges this status-quo of either manually defining chaos experiments, or using random fault injections, by working towards a more efficient approach for chaos engineering. The idea, visualized in Figure 1.1, is as follows:

- The architectural model, containing information about the different services of a microservice architecture, is extracted by analyzing traces.
- The architectural model is analyzed in the decision engine.
- The decision engine generates chaos experiments that are run against the live system and a simulator.
- The simulator simulates the microservice architecture, adding an alternative to running experiments on live systems. This has the advantage of any faults that are revealed not causing a problem for any user of the live system.



- 
- The decision engine receives feedback from the experiment results, using it to optimize future chaos experiment selections.

This thesis focusses on the decision engine component, exploring different algorithms and comparing them on their efficiency in regards to selecting fault injections, that are to be used in chaos experiments.

## Problem

Given is an architectural model of a microservice architecture, including information about the services of the architecture, like the presence or absence of the circuit breaker pattern for a particular service. Furthermore, fault injection results for a reference architecture are available for evaluation purposes. The tasks of this thesis are as follows:

- Find algorithms that can make use of the available information of a microservice architecture to select fault injections, optimizing the selection as results of prior fault injections are processed.
- Compare the algorithms in regards to their efficiency for finding faults. For this, metrics that measure the efficiency have to be defined.

## Goals

The goals of this thesis are as follows:

- Define a common interface and framework that can be used for the evaluation of algorithms that select fault injections.
- Research algorithms that can be used for the efficient selection of fault injections into a microservice architecture.
- Implement the researched algorithms, using the previously defined common interface.
- Define metrics used for evaluation of the algorithms. These metrics should cover both the efficiency, and general quality of results of the algorithms.
- Evaluate the algorithms in an experimental evaluation, using the provided reference architectural model and available fault injection data.

# Summary of Results

This thesis defines a framework for evaluating algorithms that select fault injections, that are to be used in chaos experiments. This framework utilizes the strategy pattern to define a common interface for algorithms. This interface is made up of the following methods:

- *init*: The initialization of the algorithm.
- *get\_experiment*: The selection of the fault injection.
- *enter\_result*: The processing of the result, received after the chaos experiment finished.
- *reset*: A reset function, only used for evaluation purposes.

The algorithms for selecting fault injections presented, and evaluated, are as follows:

- *Low Level Bayesian Network*: This algorithm utilizes a simple Bayesian network, made up of only the *faultRevealed* variable and the possible fault injections. It uses the observations of previous fault injections to compute probabilities for future selections.
- *High Level Bayesian Network*: The high level Bayesian network utilizes more variables, compared to the low level approach. Instead of using fault injections directly, it extracts variables describing the fault injections, and uses those to compute probabilities for variable sets. Variables used are: *faultRevealed* (the result), *circuitBreaker*, *dependencies*, and *faultType*.
- *$\epsilon$ -greedy Reinforcement Learning*: A simple reinforcement learning algorithm, that utilizes a  $\epsilon$  parameter to randomly select fault injections, exploring more options.
- *Optimistic Initial Value Reinforcement Learning*: A reinforcement learning algorithm that, instead of incrementally improving the estimates, initially all assumes all fault injections to be great. It then adjusts the estimates down, exploring by selecting the highest values.
- *Q-Learning Reinforcement Learning*: This reinforcement learning algorithm utilizes the nodes of a dependency graph to explore and reveal the most vulnerable paths a request can take through the microservice architecture.
- *Neural Network Classifier*: A perceptron classifier is used to compute probabilities for fault injections. The computed probabilities are then used for selecting one fault injection at a time.

- 
- *Tableau Reinforcement Learning*: Feature sets are extracted from fault injections. Feature sets are then prioritized, based on previous results. A feature set from the highest priority is then chosen, and a fault injection that matches the features is returned.
  - *Multilevel Feedback Queue*: Building on multilevel feedback queues in process scheduling, this algorithm sorts fault injections into multiple queues, with different priorities assigned to each queue. First a queue is selected, and then the fault injection in the front of the queue is returned, and reassigned to another queue, based on the result.

These algorithms select fault injections, either *abort*, or *delay*, into operations of services of a microservice architecture. The microservice architecture is provided through an extracted architectural model, as introduced by Beck [Bec18].

The experimental evaluation uses the algorithms on three example architectures, and compares them to a random selection of fault injections. For evaluation, the following metrics were used: *Efficiency Rate*, *Fault Detection Rate*, *Weighted Fault Detection Rate*, *times* for the *get\_experiment* and *enter\_result* operations, and *selection counters* for each fault injection.

The results indicate that the *Neural Network Classifier* generally performs as one of the best. Both  *$\epsilon$ -greedy Reinforcement Learning*, and *Optimistic Initial Value Reinforcement Learning* perform well, while the latter behaves almost deterministic. *Low Level* and *High Level* Bayesian networks deliver similar results, however training impacts their efficiency. *Multilevel Feedback Queue* performed average, while *Tableau Reinforcement Learning* performed close to *random* selection for the example architectures. *Q-Learning Reinforcement Learning* had a wide variance in results, performing bad for one architecture, and good for the rest.

Futhermore, an outline for integrating the results into the ORCAS framework is provided. For automating the chaos experiment execution into a live system, as well as the simulator, the *ChaosToolkit*<sup>1</sup> can be utilized. However, before chaos experiments can be generated and run automatically, a steady-state check would have to be generated through the architectural model, and a common approach for injecting faults into the live system is required. As of now, both of those tasks have to be performed manually.

Supplementary material, including the results of the experimental evaluation and code for generating them, is available under [https://zenodo.org/record/3265806#.XRtMgy\\_8LOQ](https://zenodo.org/record/3265806#.XRtMgy_8LOQ).

<sup>1</sup><http://chaostoolkit.org>

## Thesis Structure

This thesis is structured as follows:

1. **Introduction:** Introduce the problem and present the context. Describe the goals of the thesis.
2. **Foundations:** Present relevant foundations based on literature research. Topics presented will be foundations on microservice architectures, resilience patterns, fault injections, chaos engineering, test suite optimization, the related ORCAS project, and specific algorithms.
3. **Related Work:** Review related work on this subject, including lineage driven fault injection and a reinforcement learning approach for prioritizing test cases in continuous integration environments.
4. **Algorithms:** Given an overview of the framework used for this thesis, including the definition of the common interface that the implemented algorithms use. Go on to present the individual algorithms in a common scheme, followed by examples of their application.
5. **ORCAS Decision Engine:** This chapter focuses on the integration of this work into the ORCAS framework. This describes how the results of this thesis can be adapted to work in a real environment.
6. **Evaluation:** The experiment setting and research questions for the evaluation will be presented. After, the evaluation results will be described and discussed.
7. **Conclusion:** A concluding summary on the work and an outlook into possible future work that builds on this thesis will be provided.

## Chapter 2

# Foundations

---

This chapter presents foundations for the following parts of this thesis. Topics covered are the microservice architecture style, resilience patterns and antipatterns, fault injections, chaos engineering, test suite optimization, the ORCAS project, which the work of this thesis contributes to, and foundations on the algorithms presented as part of this thesis.

## 2.1 Microservice Architectures

To avoid some of the problems [VGC+15] of monolithic applications in the context of cloud computing, for example the added complexity of a shared codebase and increased management cost, microservice architectures have been gaining in popularity, finding adoption at companies like Amazon [Kra11] and Netflix [Mau15].

In comparison to the monolithic approach, the microservice architectural style divides an application into a smaller set of services, with each independently offering a subset of the services provided by the application. The various services then interact via messages [DGL+17].

The use of microservices allows the developers to build and test microservices in independent teams, while utilizing technologies and programming languages optimal for the particular use-case [VGC+15]. Furthermore, the task of deploying, scaling, and operating the microservice is also handled independent from other microservices, allowing the appropriate use of server types (high CPU, high memory, etc.) and scaling rules. Due to the independent deployment of the individual microservices, the pattern also allows for the use of continuous delivery strategies, since new deployments do not affect other microservices and they can continue their operation without any disruptions.

To summarize, the microservice architectural style allows companies to scale their applications efficiently in the cloud, reduce complexity, grow their developer base easily due to a separation of teams, and achieve agility.

While the microservice architectural style avoids the *single point of failure* problem, it does bring a new range of problems. Since the microservices can fail individually and generally communicate over a network, calls between them can fail at any time. In this context it is important to follow the *design for failure* principle [LF], as in both expecting and gracefully handling the failure of services.

Considering this, we now want to introduce the notion of resilience patterns and antipatterns.

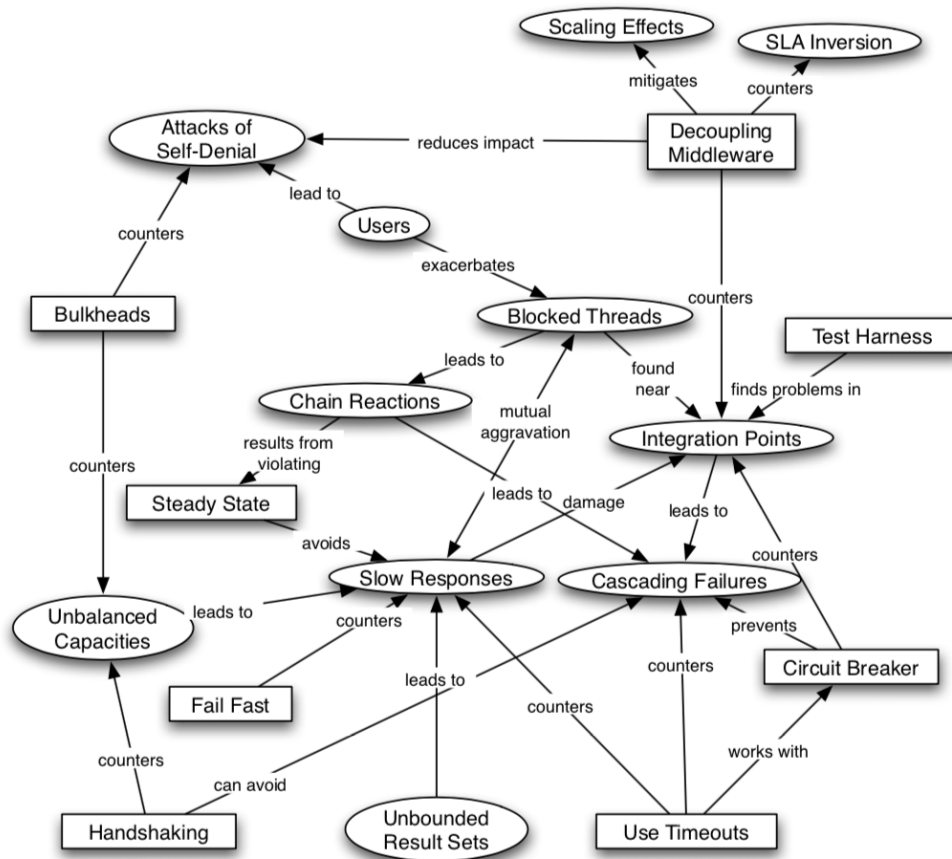
## 2.2 Resilience Patterns and Antipatterns

When inspecting system failures, one can often find common elements, a pattern of events, in the chain of failure. The so-called *Resilience Antipatterns* [Nyy07] (or Stability Antipatterns) deal with describing these recurring patterns. These antipatterns bring the system out of a stable state, into an instable state. Stability itself is defined as follows:

### **Definition 1**

**Stability:** *A resilient system keeps processing transactions, even when there are transient impulses, persistent stresses, or component failures disrupting normal processing [Nyy07].*

*Resilience Patterns* [Nyy07] (or Stability Patterns) counter resilience antipatterns on the design and architecture level, attempting to make the system resilient towards them. While it is impossible to prevent any failures from occurring, these patterns help with containing damages and maintaining partial functionality of a system. The goal of any system is to remain in a stable state, and resilience patterns are one way to prevent common occurrences of instability.



**Figure 2.1:** Overview of antipatterns and patterns [Nyy07]

Figure 2.1 shows an outtake of some of these resilience patterns and their corresponding antipatterns. Resilience patterns are displayed rectangular, while antipatterns are displayed as ovals.

### 2.2.1 Circuit Breaker

A common antipattern [Nyy07] is the scenario of *Cascading Failures*, where the failure of one or few components leads to the failure of different components. Especially for microservice architectures, with many services communicating with different services, a failure of one service, and as a result remote calls not finishing, can lead to outages of the entire system.

The usage of the *Circuit Breaker* pattern prevents this by opting for a previously defined fallback, in case of a failure. For doing so, a remote call is wrapped in a circuit breaker, e.g. through Hystrix [Net]. Hystrix then monitors the failure rate, and if a previously

defined threshold is exceeded, opens the *circuit* and immediately chooses the defined fallback function for the next execution of that particular remote call. The circuit breaker gives the requested service time to recover, before performing a single request as a *health check*. If successful, the *circuit* closes and the following invocations are executed as expected. If the *health check* fails, the *circuit* opens once again, and the fallback is used [Nyy07].

### 2.3 Fault Injections

Fault injections are a practical approach for achieving confidence in a system's robustness to unforeseen faulty conditions [NCM16]. In this context we need to introduce the terminology of *faults*, *errors*, and *failures* [ALRL04]:

#### **Definition 2**

***Fault:*** A fault is the cause of an incorrect system state.

#### **Definition 3**

***Error:*** An error is an incorrect system state, which has been caused by a fault.

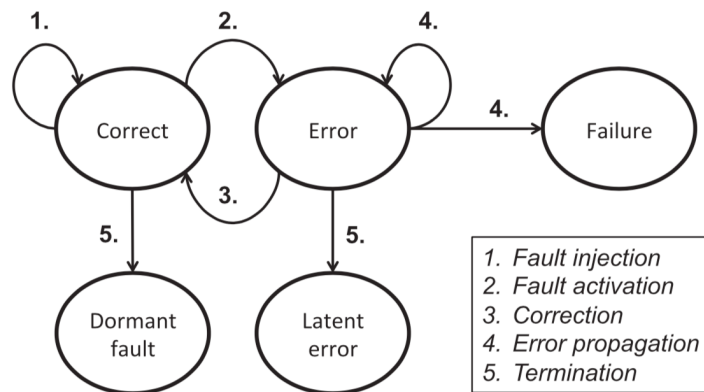
#### **Definition 4**

***Failure:*** A failure is the perception of an error by a user or external system.

For example: A software counts the days of sun in a year and returns the information on demand. However, when requesting the days of sun the output is zero, even though there have been sunny days. This is due to a bug in a subroutine, which incorrectly counts sunny days as cloudy, the *fault*. On the first day of sun the incorrect counter is incremented, causing the system state to be incorrect, the *error*. When the user then requests the amount of sunny days, the *failure* occurs: The result is not as expected.

One has to differentiate [NCM16] a *fault injection* and an *error injection*. The former deals with injecting code changes, *faults*, into the software system, while the latter injects software-, hardware-, or environment-related fault effects, *errors*. When injecting a fault one has to consider three aspects, (i) what to inject, (ii) where to inject, and (iii) when to inject.





**Figure 2.2:** States of a fault injection experiment [NCM16]

Figure 2.2 shows the different states a system can take after injecting a fault.

Assuming our system is working as intended when injecting the fault and it is activated we move into the *error* state. If our system is not resilient towards the fault and the error propagates until the system exhibits a *failure*. The error can also remain *latent* when not propagating. If the fault is not activated it remains *dormant*.

A fault injection system is typically made up of three components, the *load generator*, which generates load for the target system after the fault injection, the *injector*, which injects the fault into the target system, and a *monitor*, that allows the user to assert the outcome of a fault injection. A *controller* orchestrates the execution of fault injections, while storing results for analysis purposes [NCM16].

A fault injection itself consists of the workload, for the *load generator*, and faultload, which the *injector* implements into the system. While fault injections can consist of multiple faults at a time, injecting one fault at a time is typical [NCM16].

## 2.4 Chaos Engineering

Netflix published their approach to the reliability problem of microservice architectures, Chaos Engineering [RHB+17], in 2012.

### Definition 5

**Chaos Engineering:** *Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in its capability to withstand turbulent conditions in production. [RHB+17]*

Chaos Engineering is based on running chaos experiments, which Netflix describes in four steps [RHB+17]:

1. Define a 'steady state' as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real world events, like server crashes, severed network connections, etc.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group

Furthermore, Netflix specifies five principles that apply to the ideal usage of Chaos Engineering [RHB+17]:

- Build a Hypothesis around Steady State Behavior: Focus on the measurable output of a system.
- Vary Real-world Events: Prioritize events by potential impact or estimated frequency.
- Run Experiments in Production: Sampling real traffic is the only way to reliably capture the request path.
- Automate Experiments to Run Continuously: Automate both orchestration and analysis.
- Minimize Blast Radius: The fallout from experiments should be minimized and contained.

As described by step three *Introduce variables that reflect real world events* experiments rely on the injection of faults, like failures, latencies, or similar events. To define and observe changes to the steady state after injecting faults, metrics, and in turn monitoring for measuring these metrics, are needed.

This thesis will focus on the third step, the introduction of chaos into the system in the form of fault injections.

Chaos monkeys [Blo11] are the original go-to approach to chaos engineering, and is still the go-to for chaos automation.

### 2.4.1 Chaos Monkeys

Netflix first built the chaos monkey tool with the idea of unleashing a wild monkey into a data center, randomly shutting down instances and chewing through cables. With that idea turned into a reality, the chaos monkey tool [Blo11] allows random disabling of cloud instances through Spinnaker [Blo16]. Taking this idea one step further, the chaos gorilla [Blo11] allows random disabling of entire Amazon availability zone. As part of the Simian Army [htt16], further tools were available for maintaining a clean state of the instances (e.g. conformity monkey and janitor monkey [Blo11]). However, maintenance of the simian army tool ceased in 2016, and the most popular monkey types have been rolled into separate tools [htt16].

## 2.5 Test Suite Optimization

When talking about test suite optimization one has to differentiate between test suite reduction, and test case prioritization. Test suite reduction has the goal of reducing the size of a test suite to obtain a subset that yields equivalent coverage with respect to some criteria with respect to some criteria [RM+09][HGS93] Test case prioritization has a different goal of optimizing results of a test suite by prioritizing test cases [EMR02] and creating a sequence of test cases with the goal of revealing faults as early as possible.

The following sections will define the test suite reduction and test case prioritization problems.

### 2.5.1 Test Suite Reduction

The *Test Suite Reduction Problem* [HGS93] describes the problem of reducing the size of a test suite  $T$ , a set of test cases, as much as possible, so that it still satisfies a set of requirements defined to measure coverage of the program, e.g. statement coverage or branch coverage. The formal definition goes as follows:

**Definition 6**

*Given: Test suite,  $T$ , a set of test-case requirements,  $r_1, r_2, \dots, r_n$ , that must be satisfied to provide the best coverage of the program.*

*Problem: Find  $T' \subset T$  such that  $\forall T'', T'' \subset T, T''$  satisfies all  $r_i$ :  $|T'| \leq |T''|$ .*

### 2.5.2 Test Case Prioritization

The *Test Case Prioritization Problem* [RUCH01][EMR02] is the problem of prioritizing test cases in a test suite  $T$ . The problem looks to find a permutation of test cases, that returns the best possible results, based on a previously defined function value function  $f$ . The formal definition is:

**Definition 7**

*Given:* Test suite,  $T$ , a set of all permutations of  $T$ ,  $PT$ , a function from  $PT$  to real numbers,  $f$ .

*Problem:* Find  $T' \in PT$  such that  $\forall T'', T'' \in PT, T'' \neq T' : f(T') \geq f(T'')$ .

$f$ , or the test case prioritization, can have different goals for assigning a value to a permutation. Some of these goals can be:

- Rate of fault detection
- Rate of code coverage
- Rate of increase of confidence in reliability
- Rate of fault detection in specific code changes

Rothermel et al. [RUCH01] differentiate between two types of test case prioritization: *General test case prioritization*, and *version-specific test case prioritization*.

*General test case prioritization* has the goal of finding a permutation of test cases that will be useful over different versions of a software, e.g. regression testing after the release of a new version. The goal of the prioritized permutation is to be more successful *on average* over multiple versions, than the non-prioritized test suite.

*Version-specific test case prioritization* on the other hand attempts to optimize prioritization for a specific version, considering recent changes in the new version, without considering efficiency for other versions.

Elbaum, Malishevsky, and Rothermel [EMR02] and Rothermel et al. [RUCH01] present a vast array of approaches to test case prioritization. These include approaches based on statement coverage, probability of exposing faults, function coverage, and probability of fault existence.

Furthermore, Elbaum, Malishevsky, and Rothermel [EMR02] define a metric for evaluating the efficiency of a test suite's rate of fault detection, *Average Percentage of Faults Detected (APFD)*. For a test suite  $T$  of  $n$  test cases, and a set of  $m$  faults  $F$  that are

revealed by  $T$ , let  $TF_i$  be the first test case in ordering  $T'$  of  $T$  which reveals fault  $i$ . The APFD for the test suite  $T'$  is given by the equation:

$$(2.1) \text{ APFD} = 1 - \frac{TF_1 + TF_2 + \dots + TF_m}{nm} + \frac{1}{2n}$$

The APFD value then ranges between 0 and 100, with a higher value indicating a faster fault detection rate. APFD assumes that all test cases in a test suite are executed, eventually. To allow for only a subset of test cases to be executed, possibly leaving some faults undetected, [QCW07] introduces the adapted metric of Normalized APFD. For the subset of test cases  $TS_i$ , let  $r$  denote the amount of faults found in  $TS_i$ . NAPFD considers the percentage of faults found out of all  $m$  existing faults. If all faults have been detected, then NAPFD is equivalent to APFD [SGMM17b].

$$(2.2) \text{ NAPFD} = p - \text{APFD} \text{ with } p = \frac{r}{m}$$

## 2.6 ORCAS Project

To provide an alternative to random fault injections in the context of resilience benchmarking for microservice architectures, the ORCAS project [HADP18] proposes an option for more efficient resilience benchmarking by leveraging automatically extracted knowledge about the architecture, relationship between patterns, antipatterns, and fault injections, and simulation results to reduce the number of benchmarks executed in testing and production environments.

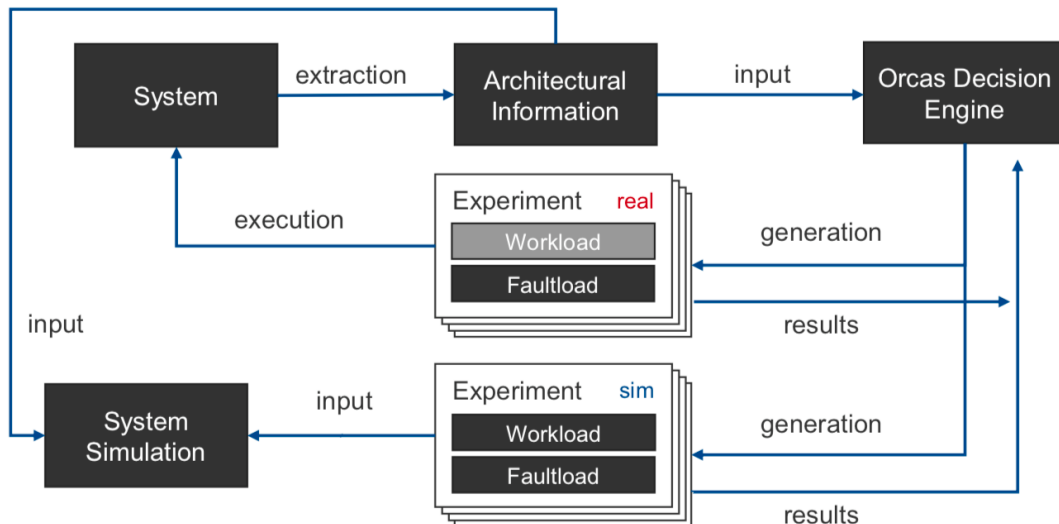


Figure 2.3: ORCAS overview [HADP18]

Figure 2.3 shows the basic workflow of the ORCAS project. Outside of the system, which is given, three components are required. In the following the purpose of the individual components will be explained.

### 2.6.1 Architecture Extractor

The Architecture Extractor [HADP18] is based on a modeling language for microservice architectures, as defined in [DH17]. It contains services and their relationships, as well as any used resilience patterns.

Beck [Bec18] presents a microservice architecture extraction approach utilizing the Jaeger tracer, which builds on OpenTracing. OpenTracing, while not a standard [opea], is hosted by the Cloud Native Foundation [opeb] and attempts to provide a more standardized API for instrumentation and distributed tracing. These APIs allow developers to add instrumentation, without being locked into a particular vendor or product. Available for most common languages [opec], various tracing systems build on the OpenTracing standard [oped]. This allows different components to use different tracers, which, as long as they are based on OpenTracing, are able to understand each other [Sig16]. Jaeger itself, one of the tracers supporting OpenTracing, too, is hosted by the Cloud Native Foundation [jae]. Jaeger's functionality can be used for distributed transaction monitoring, root cause analysis, distributed context propagation, service dependency analysis, and performance and latency optimiation [Bec18] [jae].

Beck's extracted architectural model consists of basic information about the microservice architecture, including information about services, operations, dependencies between operations, and the existence of circuit breaker patterns for operations. Listing 2.1 presents an example from the architecture extractor's output.

```
{
  "microservices": [
    {
      "name": "frontend",
      "instances": 2,
      "patterns": [],
      "capacity": 1000,
      "operations": [
        {
          "name": "getUsers",
          "demand": 100,
          "circuitBreaker": {
            "rollingWindow": 10,
            "requestVolumeThreshold": 20,
            "errorThresholdPercentage": 0.5,
            "timeout": 1,

```

```

    "sleepWindow": 5
  },
  "dependencies": [
    {
      "service": "users",
      "operation": "authentication",
      "probability": 1.0
    },
    {
      "service": "users",
      "operation": "users",
      "probability": 1.0
    }
  ]
},
...
]
}

```

**Listing 2.1:** Example model from extraction model [Bec]

However, not all of this information actually got extracted from the tracing information, some of it only consists of default values [Bec18]. Default values are set for:

- *probability* values in dependencies
- *circuitBreaker* parameters
- *demand* for operations
- *capacity* for services
- *patterns* for services

This leaves us with the following, more simplified model of generated architecture information:

```

{
  "microservices": [
    {
      "name": "frontend",
      "instances": 2,
      "operations": [
        {
          "name": "getUsers",
          "circuitBreaker": true,
          "dependencies": [
            {
              "service": "users",
              "operation": "authentication",
            },
          ],
        },
      ],
    },
  ],
}

```

```
        {
            "service": "users",
            "operation": "users",
        }
    ],
    ...
]
```

**Listing 2.2:** Example model from extraction model

The information provided by the simplified model is:

- The services of the microservice architecture
- The amount of instances a service has
- The operations a service has
- The presence or absence of the circuit breaker pattern for an operation
- Dependencies between operations

### 2.6.2 Decision Engine

The Decision Engine [HADP18] takes the architectural input and generates experiments (consisting of both workloads and faultloads) for both the system and simulator to execute. Based on the experiment results the model used for experiment generation shall be optimized. To generate experiments more efficiently knowledge about (anti)patterns and dependencies between services should be utilized. Currently a first prototype of a Bayesian network is used to build a probabilistic model for experiment determination. After generating the experiment it will be executed in either the simulator or a testing environment.

Approaching the task of comparing different approaches to choosing experiments is the main goal of this thesis.

### 2.6.3 Simulator

The simulator [HADP18] executes instances of the architectural model based on a given workload and faultload definition and returns errors, response times, and utilization of resources.



In [BGZ17] and [Bec18] the MiSim microservice resilience simulator [BGZ] was proposed and enhanced. The MiSim microservice resilience simulator simulates fault injections into a microservice architecture, with support for the popular circuit breaker pattern. The implementation of the circuit breaker is an adaptation of Hystrix's [Net] version of the circuit breaker, to be similar to the behavior of an actual software library [Bec18].

The MiSim simulator simulates microservice architectures based on the architecture model presented in Section 2.6.1. An experiment model allows defining of both workload and fault load. Listing 2.3 is an example for an experiment model.

```
{
  "simulation_meta_data": {
    "experiment_name": "ABCDE Experiment",
    "model_name": "Schema ABCDE",
    "duration": 50,
    "report": "",
    "datapoints": 50,
    "seed": 979
  },
  "request_generators": [
    {
      "microservice": "A",
      "operation": "a1",
      "interval": 0.25
    },
    {
      "microservice": "A",
      "operation": "a2",
      "interval": 0.25
    }
  ],
  "chaosmonkeys": [
    {
      "microservice": "E",
      "instances": 1,
      "time": 10
    }
  ]
}
```

**Listing 2.3:** Experiment model for MiSim [BGZ]

The experiment model [BGZ17] consists of the following parts:

**simulation\_meta\_data:** This array holds basic information about the experiment.

- *experiment\_meta\_data*: The name of the experiment.
- *model\_name*: The architecture model the simulator runs against.

- *duration*: The experiment duration.
- *report*: Whether a report should be generated, "" (blank) generates a detailed report, while "*minimalistic*" generates a minimal report, and "*none*" generates no report.
- *datapoints*: The number of datapoints for the charts in the report. "0" generates no charts, while "-1" adds a datapoint for each second of the simulation.
- *seed*: An integer value that seeds the probabilities of operations during the experiment. The value is deterministic, meaning that the same value for the seed will always return the same result of the simulation.

**request\_generators**: This array holds request generators that request the defined operation in the given interval, e.g. a load balancer. This could be seen as the workload of the fault injection.

- *microservice*: The name of the service
- *operation*: The name of the operation to be requested
- *interval*: The time interval for new requests, in seconds.

**chaosmonkeys**: This is the faultload of the fault injection. Multiple chaosmonkeys can be defined, as multiple faults can be injected.

- *microservice*: The service that is to be shut down
- *instances*: The amount of instances to shut down
- *time*: The timepoint (after the start of the simulation) that the instances of the service should be shut down

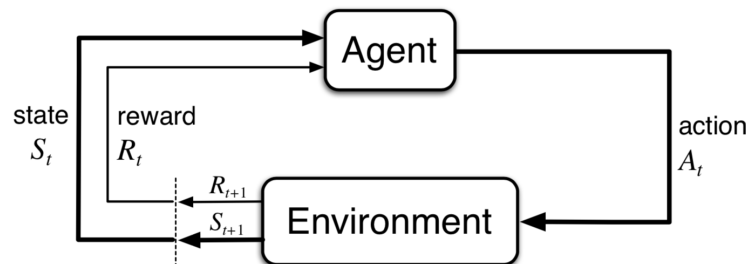
## 2.7 Algorithms

This section presents foundations related to the algorithms directly. Presented will be foundations for reinforcement learning, Bayesian networks, artificial neural networks, and multilevel feedback queues.

These topics were chosen, as they provide various levels of complexity in regards to solving the problem of selecting experiments. Multilevel feedback queues are quite a simple approach, Bayesian networks are a popular introduction to learning, also used as one of the most basic classifiers in natural language processing, and reinforcement learning presents the, theoretically, ideal approach to solving this problem, based on the problem definition of learning from experiences.

### 2.7.1 Reinforcement Learning

Without depending on preprocessed data, reinforcement learning (RL) learns from its own experience. There is no labeled data to train from, instead the basic idea is to utilize the information gain of learning in unknown areas by exploration [Kun00].



**Figure 2.4:** Agent-environment interaction [SB+98]

Figure 2.4 visualizes the basic interaction of RL. An environment has states, and RL causes actions to shift between states. Each of these actions is then rewarded with a numerical value, a reward for the so-called *agent*. Future rewards for the agent depend on the knowledge learned from prior actions.

Here the trade-off between exploration and exploitation becomes a main characteristic of RL. While the agent wants to maximize the rewards of future actions, which it does by picking the action that he knows to provide the highest estimated reward, it also wants to explore new actions that may provide even higher rewards. Furthermore, not every reward may be immediate, as multiple actions may be necessary to obtain a delayed reward. This principle of trial and error is a unique feature of RL [Kun00].

The mapping of states to actions that yield the highest reward, is the core problem of RL. A *policy*  $\pi$  of an agent describes the mapping of states to actions as a set of probabilities.  $\pi(a|s)$  describes the probability for selecting action  $a$  while in state  $s$  [Kun00].

Beyond the agent, environment, rewards, and policy, two further main subelements exist in a RL system: a *value function*, and, optionally, a *model of the system* [SB+98].

The *value function* specifies what is good in the long run, accounting for the aforementioned delayed rewards. While a reward is immediate, the value function considers the *long-term* view of all states and their rewards. While determining rewards is an easy task, estimating good values is a much harder task, as the sequences and observations taken from the agent have to be considered through its entire lifetime [SB+98].

The last component differentiates between *model-based* and *model-free* methods. The environment model is used for planning, allowing model-based methods to anticipate

future actions ahead of time. The opposite, the model-free approach, relies entirely on trial and error.

As RL is learning during runtime, it is considered to be an *online learning* method [SGMM17a].

### Task Types

If a goal, e.g. the successful parking of a car, can be achieved in a finite number of actions, in an *episode* of steps, with *terminal states* describing the end of an episode, we consider the task of achieving the goal a *episodic task* [SB+98]. If, however, there is no identifiable break, we consider the task a *continuing task* [SB+98]. For episodes we can consider the sum of all rewards of all actions in the episode to calculate our *expected return*, and compare different rewards with different outcomes [SB+98]. For continuing tasks, where the count of actions is infinite, a *discount rate*  $\gamma$  is necessary for calculating the *expected return* of an episode  $G_t$  [SB+98]. Instead of

$$(2.3) \quad G_t = R_{t+1} + R_{t+2} + \dots + R_{t+T}$$

where  $T$  is the total amount of steps in an episode, one has to use

$$(2.4) \quad G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

with  $0 \leq \gamma \leq 1$  as discount rate, limiting the value of future actions, compared to present actions.  $\gamma$  is a parameter, with  $\gamma = 0$  only considering immediate rewards, and  $\gamma < 1$  returning a finite value for finite reward sequences.

### Value Function

In RL the learner needs information about the quality of a state in order to find the optimal policy. The value function  $V_\pi$  computes the expected reward for a state, if the agent follow the policy  $\pi$  [SB+98].

$$(2.5) \quad V_\pi(s) = \mathbb{E}[G_t | S_t = s]$$

For actions,  $Q_\pi(a)$  denotes the *action-value function for policy  $\pi$* .

$$(2.6) \quad Q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a]$$

The value function  $V_\pi$  and  $Q_\pi$  are related, as  $V_\pi$  considers the best action for each state. It can then be seen as [Kun00]

$$(2.7) \quad V_\pi(s) = \sum_a (\pi(s, a) Q_\pi(s, a))$$

## k-armed Bandit Problem

The *k-armed Bandit Problem* is the problem of repeatedly being faced with  $k$  options. After choosing an option, the action, a numerical value is returned as reward. The goal is to maximize the total reward for a certain amount of actions.

Sutton, Barto, et al. [SB+98] describes the problem as a bandit sitting at a slot machine. The bandit is repeatedly faced with the same choice among  $k$  options. One single slot machine would be a 1-armed bandit problem, as only one lever, or option, is available. After choosing an option, the bandit is rewarded with a numerical value (the winnings). By learning about which of the  $k$  levers is the best to use, the bandit attempts to maximize the winnings [SB+98] by repeatedly choosing the best levers.

Noteable for the  $k$ -armed bandit problem is that the problem is stationary, the action always occurs from the same state and the state of the environment never changes. The *value* of an action then is the estimated reward, as there is no consideration for sequences, due to the lack of states. The selected action at position  $t$  is  $A_t$ , and the corresponding reward is  $R_t$ . The value of arbitrary action  $a$  is then the optimal estimated reward:

$$(2.8) \quad Q_*(a) = \mathbb{E}[R_t | A_t = a]$$

. If the values were known, the problem would be trivial, as we would always choose the highest valued action. Thus, we only work with estimates for the value, with  $Q_t(a)$  being the estimate for  $Q_*(a)$  at step  $t$ .

When it comes to choosing the next action, there is the *greedy* approach, selecting the action with the highest estimated value. This is the exploitative approach, whereas the expolorative approach selects an action that, based solely on the estimated values, is not ideal. When to exploit and when to explore is not trivial to figure out, however, exploring early would make sense, to maximize the estimated values for later exploitation.

One simple approach to calculating the estimated value  $Q_t(a)$  is as follows:

$$(2.9) \quad Q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot I_{A_i=a}}{\sum_{i=1}^{t-1} I_{A_i=a}}$$

Here  $I_{A_i=a}$  is 1 if the action  $a$  was selected at step  $i$ ,  $A_i = a$ , and 0 if  $A_i \neq a$ .

To prevent division by 0, a default value for the estimated reward  $Q_t(a)$  has to be defined, e.g. 0. To save on computational resources, the estimated value can also be updated incrementally. For one single action the new estimated value  $Q_{n+1}$  can be calculated by utilizing the old estimate  $Q_n$  in comparison to the reward  $R_n$  received:

$$(2.10) \quad Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$

The *greedy*, exploitative, selection is then defined as

$$(2.11) \quad A_t = \operatorname{argmax}_a Q_t(a)$$

where  $\operatorname{argmax}_a$  selects an action  $a$  that has the highest estimated reward value  $Q_t(a)$ .

To also consider explorative selections, the parameter  $\epsilon$  can be introduced. With probability  $\epsilon$  a random action is selected. Combined with *greedy* exploitative selection, this is called  $\epsilon$ -*greedy* selection.

Summing all of these up gives the following  $\epsilon$ -*greedy*  $k$ -armed bandit algorithm for selecting actions in a single state environment.

---

**Algorithm 1**  $\epsilon$ -greedy  $k$ -armed Bandit Algorithm [SB+98]

---

```
procedure  $\epsilon$ -GREEDY K-ARMED BANDIT( $k$ )  
  for  $a \leftarrow, n$  do  
     $Q(a) \leftarrow 0$   
     $N(a) \leftarrow 0$   
  end for  
  while true do  
     $A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$   
     $R \leftarrow \text{bandit}(A)$   
     $N(A) \leftarrow N(A) + 1$   
     $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$   
  end while  
end procedure
```

---

While the  $\epsilon$ -*greedy* approach starts every estimate  $Q_1(a)$  at 0, and corrects the estimate upwards by exploring random actions through the  $\epsilon$  parameter, another approach is to assign *optimistic initial values*, e.g.  $Q_1(a) = 5 \forall a$ , and then correct the estimate downwards, exploring by choosing the least corrected estimate [SB+98]. In pseudo code this would be:

**Algorithm 2** Optimistic Initial Value Algorithm [SB+98]

---

```

procedure OPTIMISTIC INITIAL VALUE( $o$ )           //  $o$  is optimistic initial value
  for  $a \leftarrow, n$  do
     $Q(a) \leftarrow o$ 
     $N(a) \leftarrow 0$ 
  end for
  while true do
     $A \leftarrow \operatorname{argmax}_a Q(a)$ 
     $R \leftarrow \text{bandit}(A)$ 
     $N(A) \leftarrow N(A) + 1$ 
     $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$ 
  end while
end procedure

```

---

Instead of assigning the same initial value to each action, one could also utilize initial knowledge about the estimated result of actions to set initial values.  $\epsilon$ -greedy would then allow for exploration after.

### 2.7.2 Temporal Difference Learning

Sutton, Barto, et al. [SB+98] list three classes for solving reinforcement learning tasks: *Dynamic Programming*, *Monte Carlo methods*, and *Temporal Difference Learning*.

*Dynamic Programming* divides a problem into subproblems, thus splitting the task into smaller steps. For this, a perfect model of the environment is required [Kun00].

*Monte Carlo methods* do not require a perfect model, and instead learn from sequences of state-action-reward samples. However, updates only occur after complete sequences, upon reaching a final state [Kun00].

*Temporal Difference Learning* (TD) combines the two approaches, not requiring a model, and updating after each step.

Sutton, Barto, et al. [SB+98] describes TD methods through a weather forecasting example, e.g. predicting the weather for Saturday. Predicting Saturday's weather on an earlier day by evaluating the actualy outcome is a supervised approach [SB+98]. Here, the change of weather over the course of time leading up to Saturday is ignored, only the weather at the time of prediction is compared to the weather on Saturday. TD methods consider all days from the time of prediction up to Saturday, and the prediction is adjusted based on the weather on the different days. Instead of a single update, TD methods compute  $T - 1$  updates for an episode of  $T$  time steps [SB+98][Kun00].

## Q-Learning

Q-Learning is an off-policy TD algorithm. Compared to the single state  $k$ -armed bandit algorithm, the environment can take on multiple states. Furthermore, the next state's value is considered when updating the estimated reward function, that is as follows:

$$(2.12) \quad Q(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

$\alpha$  is a parameter, the learning rate of the algorithm, with  $0 < \alpha \leq 1$ . A higher value for  $\alpha$  prioritizes newly learned, while a lower value prioritizes what is already known.

At first,  $Q$  is initialized as an arbitrary value for all state-action pairs  $(s, a)$ . Then, actions are chosen from  $Q$  through a policy, e.g.  $\epsilon$ -greedy. The reward of the action is received, and  $Q(s, a)$  is updated. The state changes to the new state. Repeat this until the end of an episode is reached, i.e. a terminal state is reached [SB+98].

---

**Algorithm 3** Q-Learning Algorithm [SB+98]

---

**procedure** QLEARNING( $\alpha, \epsilon, \gamma$ )

  Initialize  $Q(s, a) \forall s \in S, a \in A(s)$ , arbitrarily, except  $Q(\text{terminal}, -) = 0$

**repeat**

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.  $\epsilon$ -greedy)

    Take action  $A$ , receive  $R, S'$

$Q(S, a) = (1 - \alpha) \cdot Q(S, A) + \alpha[R + \gamma \cdot \max_a Q(S', a) - Q(S, A)]$   $S = S'$

**until**  $S$  is terminal

**end procedure**

---

If the discount factor  $\gamma$  is less than 1, non-episodic tasks are possible too.

### 2.7.3 Bayesian Networks

The Bayesian Network approach to modeling conditional dependencies has been gaining in popularity over the last decade, with applications in machine learning, text mining, natural language processing, weather forecasting, and more adopting it [Ben08].

A Bayesian network (BN)  $B$  can be defined as an annotated acyclical graph that represents a joint probability distribution over a set of random variables  $V$ .  $B$  is then defined as  $B = \langle G, P \rangle$  with  $G$  being a directed acyclical graph, encoding independence assumptions. In  $G$  a node  $X_i$  is independent from all non-descendant variables, i.e. if  $X_i$  is dependent on a variable, that variable is a descendant of  $X_i$ . Nodes  $X_i$  are *unconditional* if they do not have any parents, else they are *conditional*.



$P$  is a set of parameters for the network. Elements of  $P$  are the conditional dependencies of the variable nodes  $X_i$ , with  $P_B(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P_B(X_i|\pi_i)$  where  $\pi_i$  is the set of parents of node  $X_i$  [Ben08].

A BN provides the basis for Bayesian Inference algorithms. These use the Bayes' theorem in equation 2.13 to update variables.

$$(2.13) \quad P(X_2|X_1) = \frac{P(X_1|X_2) \cdot P(X_1)}{P(X_2)}$$

In the BN  $X_1$  is a child node of  $X_2$ . The goal is calculating the *posterior probability*  $P(X_2|X_1)$ , based on the given parameter  $P_B(X_1|X_2)$  and independent observations  $P(X_1)$  (*marginal probability*) and  $P(X_2)$  (*prior probability*) [NJ09].

## 2.7.4 Artificial Neural Networks

Artificial Neural Networks (ANNs) are being used to solve a variety of problems, including pattern recognition, prediction, optimization, associative memory, and control [JMM96]. ANN are an approach to overcoming some of the shortcomings of the von Neumann model, some examples of these being [JMM96]:

- massive parallelism
- learning ability
- generalization ability
- adaptivity
- fault tolerance

Taking inspiration from the principles found in biological neural networks, that allow humans to solve complex perceptual problems (like spotting individuals in a crowd), an artificial neural network can be visualized as a weighted directed graph. Artificial neurons make up the nodes, while edges are neuron outputs and inputs. Based on the presence of cycles ANNs can be categorized as either *feed-forward*, when there exists no cycle in the graph, or *recurrent*, if there is a cycle. Feed-forward networks are *static*, returning one output for one input, while recurrent networks return a sequence of outputs for the same input, making them *dynamic* [JMM96].

The learning process of ANNs happens by updating the weights of edges between neurons, optimizing the performance for tasks. While not always the case, at least part of this generally happens as part of a supervised learning phase, meaning the network is provided with labeled training patterns during a training phase. Supervised

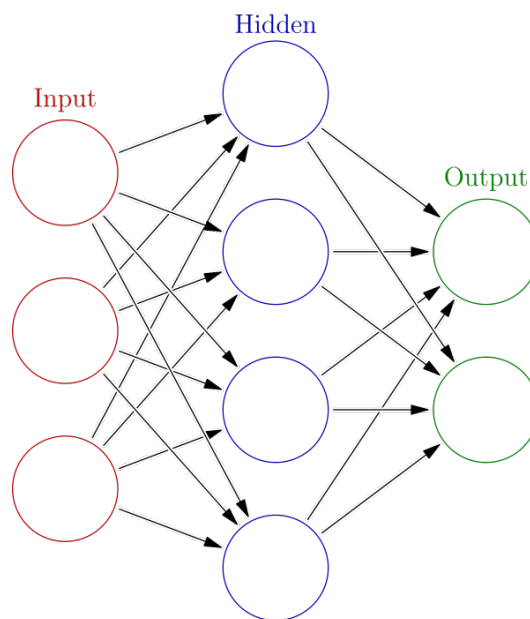
learning is an alternative to unsupervised learning, meaning there is no human input, like clustering.

During supervised learning the ANN compares the output  $y$  with the desired output  $d$ , provided by the labeled data. If  $y \neq d$ , the ANN has to error-correct and modify its connection weights. Here the perceptron-learning rule can be used. It can be imagined as a hyperplane that separates points in an  $n$ -dimensional space, staying the same when  $y = d$ , and shifting when  $y \neq d$  [JMM96].

One of the most popular approaches to neural networks is a *multilayer perceptron*. A back-propagating approach to learning minimizes the squared-error cost function

$$(2.14) \quad E = \frac{1}{2} \sum_{i=1}^p \|y^{(i)} - d^{(i)}\|^2$$

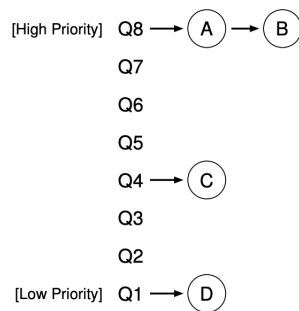
by back-tracking the error starting at the output layer, and adjusting weights accordingly. Figure 2.5 shows an example feed-forward network, made up of one input, one output, and one hidden layer. The hidden layers allow optimizing boundaries between pattern classes, as each layer adds hyperplanes from perceptrons. These hyperplanes can then be formed into a hyperregion from following hidden layers, allowing a clearer separation [JMM96].



**Figure 2.5:** 3-layer feed-forward network [Glo]

### 2.7.5 Multilevel Feedback Queues

The Multilevel Queue (MLQ) was first developed by Corbato et al. [CMD62] for scheduling of processes, using different queues for different process types, e.g. a separate queue for system processes, interactive processes, and batch processes. Each queue has a priority assigned to it, as well as an own scheduling algorithm. Selecting the next process then first selects a queue based on the priority, followed by selecting a process from the queue based on the queue's scheduling algorithm. Popular scheduling algorithms are Round Robin (RR) or First In First Out (FIFO). FIFO scheduling schedules processes based on the time of arrival. RR scheduling also goes based on arrival order, however limits the execution time of a process, returning a process to the end of the queue if it didn't finish.



**Figure 2.6:** Multilevel Feedback Queue example [AA15]

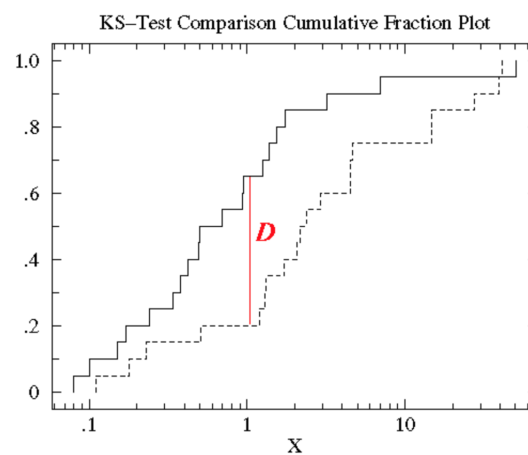
The Multilevel Feedback Queue (MLFQ) [AA15] advance on the MLQ approach by adding the concept of processes moving between queues, changing their priority as they do so. For example, [AA15] proposes the following rules for moving processes between queues:

- All queues use RR scheduling
- Take processes from the queue with the highest priority, containing waiting processes
- All processes start in the highest priority queue
- If a process does not finish in the assigned time slot, it moves to the next lowest priority
- If a process finished in the assigned time slot, it remains at the same priority
- Periodically move all waiting processes to the highest priority queue

## 2.8 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (KS) test [Mas51] is used to test the "goodness of fit" between the distribution of a set of sample values and a theoretical distribution, a hypothesis. While generally used to test if a given distribution is the normal distribution, testing for normality, it can also be used to compare two independent samples.

The KS test works by computing the cumulative fraction function for both distributions. For each value  $x$ , the cumulative fraction function computes what percentage of values in the distribution is strictly smaller than  $x$  [Kir96].



**Figure 2.7:** KS-Test Comparison of Cumulative Fraction Plots [Kir96]

The KS test then computes the vertical maximum deviation between the two cumulative fraction functions as  $D$ . If the KS statistic  $D$  is high, the significance  $p$ -value is low. If  $D$  is low, the distributions are similar, and the significance  $p$ -value is higher.

## Chapter 3

# Related Work

---

This chapter explores work related to this thesis. First, the Lineage Driven Fault Injection approach, highlighted by Netflix in [RHB+17], will be presented. Then, a reinforcement learning approach to scheduling test cases in a continuous integration environment will be summarized.

### 3.1 Lineage Driven Fault Injection

While a random injection of faults is a simple and general way to go about building experiments, it is also inefficient by exploring redundant scenarios, and not abusing any knowledge about potential weaknesses of an architecture [AAS+16].

As an alternative to random injections, the Lineage-Driven Fault Injection (LDFI) approach is based on two key insights [ARH15]:

- *Fault-tolerance is redundancy*: A system is fault-tolerant if it provides enough alternative ways to obtaining an expected outcome that is it resilient to some pre-defined set of fault conditions
- *Start with successful outcomes and reason backwards*: Go from effects to causes, instead of searching the space of all possible executions.

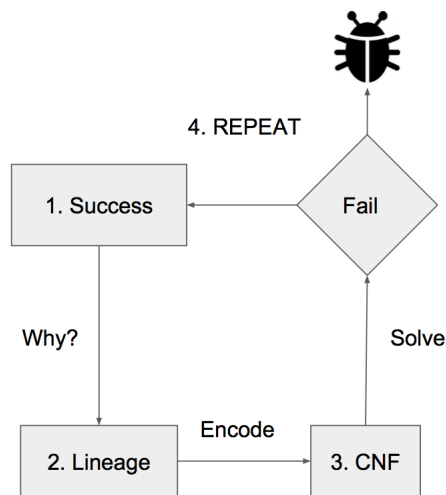
Data lineage exploits both of these insights, by building a lineage graph starting at the correct outcome and going back through all possible causes, revealing the redundant paths.

The goal of LDFI is to find the sets of faults that will reveal a fault. To do so, it represents fault sets in a conjunctive normal form (CNF, conjunctions of disjunctions). The CNF is initialized by tracing one successful request through the system. After constructing the

initial CNF, each of the fault sets is injected into the system. After injecting a fault set from the CNF, two outcomes are possible [AAS+16]:

1. System fails and exposes a fault-tolerance weakness
2. System succeeds and indicates alternative strategy to bring the expected outcome

If the latter is the case the old lineage graph (CNF) is merged with the new lineage graph (the new execution trace of the successful request) and a new CNF is constructed. This heavily relies on the alternative path that was taken being known through some kind of request tracing.



**Figure 3.1:** Lineage Driven Fault Injection [AAS+16]

Figure 3.1 shows this iterative process of finding faults through LDFI. First, a successful request path is followed through the system. This path is then modeled as a CNF of fault sets. Utilizing these fault sets will either (a), return the correct result, revealing a new path, or (b), reveal a fault. If (a) occurs, and no fault is revealed, the path taken can, again, be transformed into a CNF for fault injections.

However, as there is an infinite amount of possible inputs, expert knowledge is necessary to reduce those to a realistic set. Also, due to the very exhaustive approach of following every possible path, a lot of experiments with a low probability of revealing faults are executed, which can prove to be costly. In case of a non-deterministic system, LDFI might never terminate.

Nonetheless, for an initial case study on fault-tolerant protocols, LDFI showed significant improvements over random fault injections for uncovering specific faults [ARH15].

## 3.2 Reinforcement Learning for Test Case Prioritization and Selection in Continuous Integration

Spieker et al. [SGMM17b] presented two reinforcement learning approaches to prioritizing and selecting test cases in continuous integration environments [SGMM17b].

The first is a tableau approach. Here, each test case is considered as a state, with a set amount of actions available. The action in this case is an integer within a set range that allows for later sorting, and thus prioritizing. The exploration works through the previously introduced  $\epsilon$ -greedy exploration, selecting random actions with probability  $\epsilon$ . The  $Q$  for a state-action pair is then updated using the known

$$(3.1) \quad Q(s, a) = Q(s, a) + \frac{1}{n} * (R - Q(s, a))$$

with  $n$  being the amount of the state-action  $(s, a)$  pair has been selected in total, and  $R$  being the reward received [SGMM17b]. The tableau agent takes exploration rate  $\epsilon$  and the number of actions *action\_size* as parameters [SGMM17b].

The second approach is relying on an artificial neural network (ANN) function approximator to calculate probabilities for a test case to show a fault. Here, the states, again, are test cases. However, the action here is not an integer chosen from a range. Instead, it is a probability computed by the ANN. For exploration, the ANN adds a random value drawn from a Gaussian distribution, which is then added to the suggested action. The higher the exploration rate, the higher the deviation from the policy's suggested action, and in turn the higher the degree of exploration. For learning purposes, the ANN agent keeps a memory of experiences. Each experience is made up of the test case, probability, action, and reward received. When the memory is full, older experiences get replaced with newer experiences. When training, the ANN takes a sample of the memory and uses backpropagation with stochastic gradient descent to learn [SGMM17b]. The ANN takes a hidden node parameter, experience memory size, and experience batch sample size [SGMM17b].

For both agents the test cases are preprocessed into feature tuples. The features used are:

- *duration* of the test case, grouping into equivalence classes 0 and 1
- *last execution* of the test case, grouping into equivalence classes 0 and 1
- *last results*, using the results of the last four executions

For the reward function, three functions were tested [SGMM17b]:

### 3 Related Work

---

- *Failure Count Reward*: Counting the failures found during the execution of a test sequence and returning the same reward for all test cases.
- *Test Case Failure Reward*: Reward individual test cases with their result, i.e. 1 if they failed, and 0 if they didn't reveal a fault.
- *Time-ranked Reward*: Each failed test case is rewarded the total failure count, while passed test cases are rewarded the total failure count minus the failed test cases that followed the passed test case. This also considers the position of individual test cases in the test sequence.

After prioritizing test cases with either the tableau or ANN agent, a test sequence of test cases is built. The goal of the test sequence is to find as many faults as possible within half of the time it would take to execute all test cases [SGMM17b].

For experiment evaluation purposes the NAPFD metric presented in 2.5.2 is used. Industrial data sets of test suites with results provide the data for comparison of the tableau agent, ANN agent, and a random prioritization agent [SGMM17b].



## Chapter 4

# Algorithms

---

This chapter will present a framework for implementing the algorithms, and the algorithms, based on the previously presented research, adapted for the selection of chaos experiments.

The algorithms originate from three different areas:

1. *Bayesian Networks*: Bayesian networks, and Bayes' theorem (equation 2.13), are statistical models that suit themselves for the problem of selection chaos experiments by providing a statistical model to directly use and update iteratively. While a training phase is required for Bayesian networks, they later allow to predict the likelihood of certain causes, e.g., details about a microservice, obtained from the architectural model, that lead to a symptom, i.e., a fault being revealed. Two approaches based on Bayesian inference are presented, with one being based on circuit breaker presence alone, and one considering circuit breakers in addition to dependencies between operations. This approach was originally mentioned by van Hoorn et al. [HADP18].
2. *Reinforcement Learning*: Ideally presenting itself for this use case, by not requiring a training phase and learning from experience, reinforcement learning is a, theoretically, perfect choice for the problem of efficient chaos experiment selection. Spieker et al. [SGMM17b] present a reinforcement learning approach to creating test sequences that are time limited, tackling a similar to problem. Presented here are both single-state approaches that do not consider architectural details outside of the chaos experiment definitions, and multi-state approaches that do consider architectural information, like circuit breaker existence and dependencies between operations.
3. *Multilevel Feedback Queue*: Taking a more simplistic approach, multilevel feedback queues are supposed to efficiently schedule chaos experiments to optimize fault detection rate of the chaos experiments selected. This was chosen due to prior

familiarity with process scheduling, and multilevel feedback queues providing a much simpler approach that can be applied to all systems, while requiring very little extra effort.

The common goal of all algorithms is the selection of one fault injection, to be used in a chaos experiment, at a time, and improving the selection over time by processing the result received from executing the chaos experiment and learning for future selections.

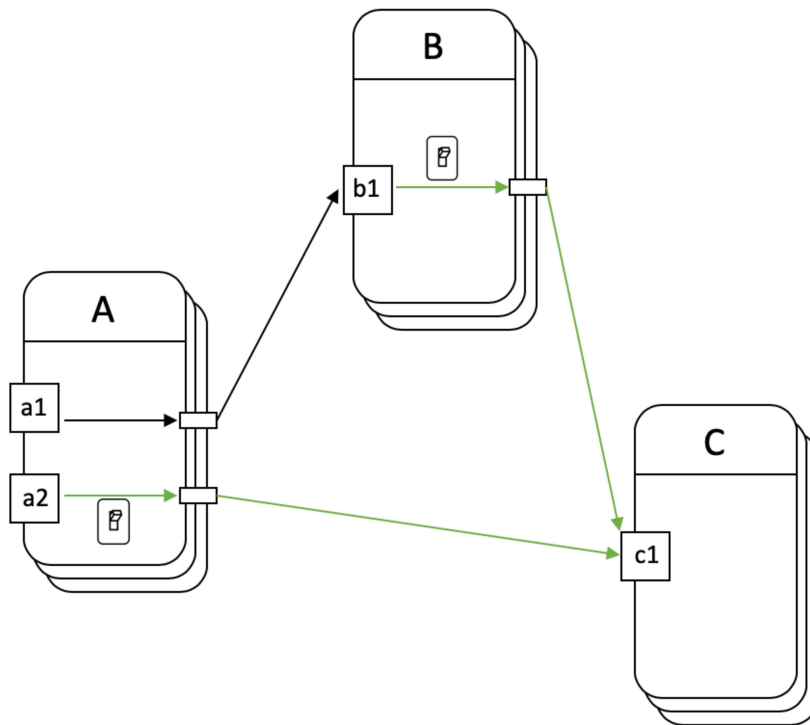
Architectural information available to all algorithms consists of the architecture model described in 2.6.1: the services, instances of services, operations, dependencies between operations, and circuit breaker implementations in operations.

The remainder of this chapter is structured as follows:

- Section 4.1 introduces an example architecture that will be used for describing the algorithms.
- Section 4.2 introduces the framework, including the common interface for implementing the algorithms.
- Section 4.3 presents the Bayesian network based algorithms.
- Section 4.4 presents the reinforcement learning based algorithms.
- Section 4.5 presents the Multilevel Feedback Queue algorithm.

### 4.1 Example Architecture

For the purpose of describing the algorithms, the following example architecture will be used throughout this chapter:



**Figure 4.1:** Architecture used as example in this chapter

This architecture is made up of the three services  $A, B, C$  with operations  $a1, a2, b1, c1$ . A circuit breaker is implemented for the  $a2$  and  $b1$  operations. This circuit breaker applies to all calls that the  $a2$  and  $b1$  operations make. If any of the calls return errors or time out too many times, the circuit will open. For example, if  $c2$  returns too many errors or times out too many times, the circuit breakers in  $a2$  and  $b1$  will open. Dependencies between operations are as follows, where  $(x \rightarrow y)$  implies  $x$  depending on  $y$ :  $(a1 \rightarrow b1), (a2 \rightarrow c1), (b1 \rightarrow c1)$ .

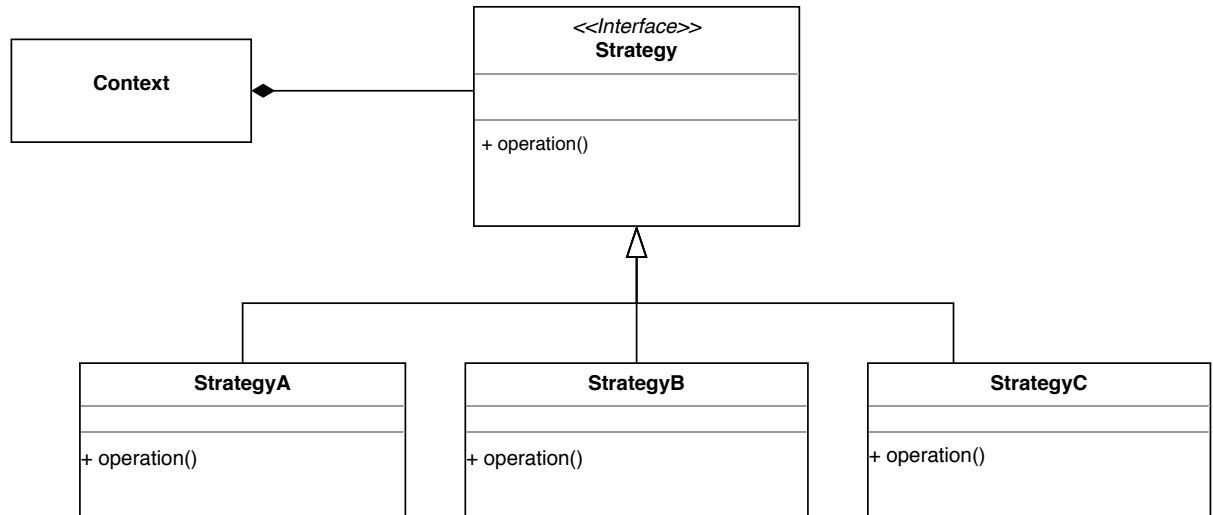
The circuit breaker implementations are visualized by the switch icon, while the remote calls that are affected by the circuit breaker are colored green.

## 4.2 Framework

The framework for implementing and evaluating the different algorithms has to allow for easy replacement of algorithms, adjusting of parameters, retrieving of fault injection data, and presentation of results.

### 4.2.1 Interface

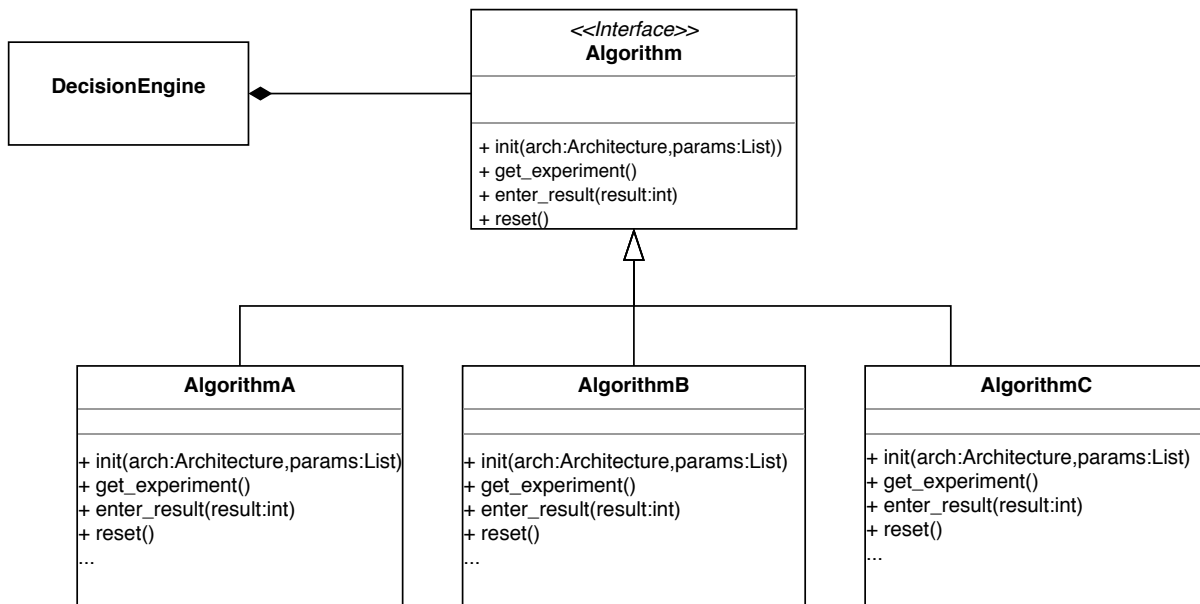
To support easy replacement of algorithms, the strategy pattern [Gam95] is followed.



**Figure 4.2:** UML class diagram for strategy pattern

The strategy pattern, as visualized in Figure 4.2, allows the direct replacement of objects by defining a common interface that all algorithm classes implement. When evaluating the algorithms, this will allow for easy replacement during runtime, as all objects of the strategy share the same signature. All of the experiment execution and visualization can remain the same, as all strategy implementations will return the same results through the same calls.

Applied to the implementation of multiple algorithms, the following adaption of the strategy pattern can be used:



**Figure 4.3:** UML class diagram of interface for algorithms

The interface, as shown in Figure 4.3, is made up of the following main operations, that the individual algorithms will implement:

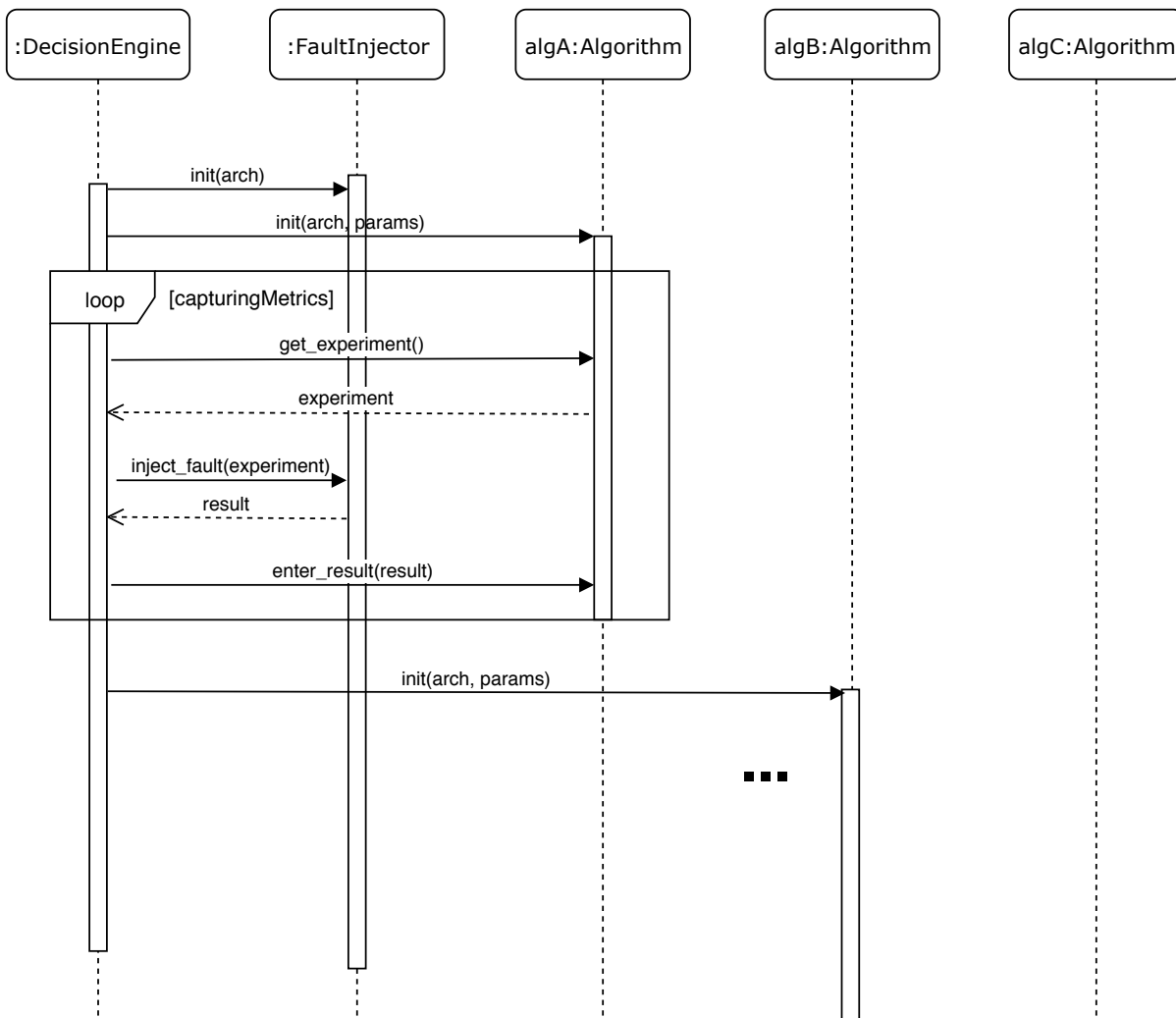
- *init(arch : Architecture, params : List)* This function initializes the model of the algorithm with all necessary architectural information. *arch* is an instance of the architecture, previously described in 2.6.1, that is used. *params* is a list of parameters, individual for each algorithm. The individual parameters of the algorithms will be described in the algorithm descriptions. This is the constructor of the class.
- *get\_experiment()* This function returns the experiment that the algorithm selected. This function does not take any parameters. This function returns the fault injection to be used in the experiment, e.g. *b1-abort*. The algorithms are only choosing fault injections, not constructing a full chaos experiment, which would include a steady state and more.
- *enter\_result(result : int)* This function updates the model with the result from the last fault injection. Multiple integration points get queried for their status, the *int* value *result* passed to the function is the sum of integration points that show a fault.
- *reset()* For evaluation purposes, the algorithms can be reset to their default state.

### 4.2.2 Fault Injections

Two types of fault injections will be used, *delay* and *abort*. *delay* inserts a delay to a request, while *abort* immediately returns an error code.

Figure 4.4 shows a sequence diagram that visualizes the function calls during the evaluation in this framework. First, all of the algorithms will be initialized. Then the algorithms will be used to generate experiments. When generating experiments, first *get\_experiment* will be called, which returns the fault injection that is to be injected. Then, through *enter\_result*, the result of the fault injection, obtained from the fault injector, will be returned to the algorithm.

The algorithms are only choosing the fault injections, the chaos experiment design will be described in Chapter 5.



**Figure 4.4:** Sequence diagram visualizing interaction between components

### 4.2.3 Language

All of the work that is part of this thesis is implemented in Python3. Python was chosen due to its great support and availability of tools for data science and machine learning. Additionally, Python offers support for object oriented programming, which allows the usage and implementation of the strategy pattern [Gam95].

### 4.3 Bayesian Network

The information about a microservice architecture, combined with results of previous experiment results, can be used to build a Bayesian network. In the following, two approaches for defining of variables in the Bayesian network will be presented. First, a low level approach, that only includes information about the fault injection, will be shown. Second, a high level approach will add architectural information to the variables.

Both algorithms use their variables in combination with Bayes' theorem presented in equation 2.13 to predict the probability of revealing a fault.

#### 4.3.1 Low Level Learning

For the low level algorithm two types of variables are used, the fault injection (*operation - faultType*) and the *fr* variable. The *fr* variable is the result of the chaos experiment, indicating whether a fault has been revealed or not. If a fault has been revealed,  $fr = 1$ . Else,  $fr = 0$ .

The algorithm uses data from prior observations to predict the probability of a fault injection revealing a fault to predict the probability of the fault injection, e.g. *c1-abort*, revealing a fault.

$$\begin{aligned}
 (4.1) \quad P(fr = 1|c1 - abort) &= \frac{P(c1 - abort|fr = 1) \cdot P(fr)}{P(c1 - abort)} \\
 &= \frac{P(c1 - abort|fr = 1) \cdot P(fr)}{P(c1 - abort|fr = 1) \cdot P(fr) + P(c1 - abort|fr = 0) \cdot P(fr = 0)}
 \end{aligned}$$

This equation describes the Bayes' theorem 2.13, applied to the variables *fr*, the observation whether a fault has been found after a chaos experiment has been executed, and the variables *operation - faultType*, the fault injections in the chaos experiment.

Here:

$$(4.2) \quad P(c1 - abort|fr = 1) = \frac{n(c1 - abort \wedge fr = 1)}{n(c1 - abort)}$$

$$(4.3) \quad P(fr = 1) = \frac{n(fr = 1)}{n(all)}$$



$$(4.4) P(c1 - abort|fr = 0) = \frac{n(c1 - abort) - n(c1 - abort \wedge fr = 1)}{n(c1 - abort)}$$

$$(4.5) P(fr = 0) = \frac{n(total) - n(fr = 1)}{n(total)}$$

$n(..)$  stands for the amount of observations for the respective parameters.  $n(total)$  stands for all observations, i.e.  $n(fr = 1)$  and  $n(fr = 0)$ .

The algorithm then selects a fault injection, using the computed probabilities  $P(fr = 1|faultInjection)$ . After the fault was injected and the result was received, the probabilities are updated.

Bayesian algorithms generally require a training phase, prior to being used to generate results. Since labeled training data is generally not available for this problem, the algorithm initializes with uniformly distributed probabilities for each fault injection. Until a certain threshold of iterations has been reached, the algorithm continues to use these initial probabilities, choosing at random, while observing the results. After the threshold has been reached, the algorithm will switch to the trained probabilities.

The algorithm takes the following parameter:

- *training\_size*: This parameter defines at what threshold of iterations the trained results should be used for selection, instead of choosing randomly from a uniform distribution.

In the following, the common functions from the interface will be described.

*init*

- Initialize the parameters, including setting the training mode to on.
- Extract fault injections from architecture model.
- Initialize observation tracking for all fault injections.
- Initialize probabilities for fault injections, using a uniform distribution.

*get\_experiment*

- Choose a fault injection, using the probabilities
- Increment the iterations counter.
- If the training limit has been reached, turn off the training mode.

*enter\_result*

- Add the result for the last fault injection to the observations.
- If the algorithm is not in training mode, recalculate the probability for all fault injections. After, normalize all probabilities so they sum up to 1.

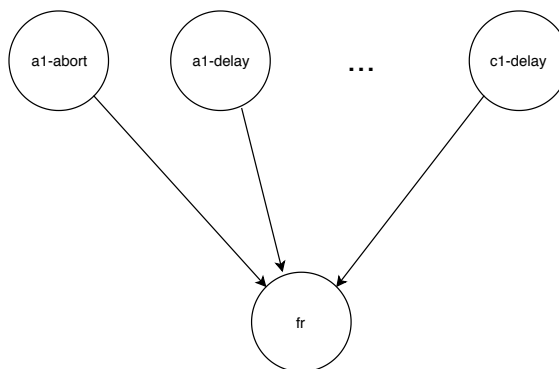
Algorithm 4 presents the algorithm.

Bayesian networks, also used as classifiers in natural language processing, generally do require a lot of data for training, before returning good results. It is unclear, how efficient this approach will be.

While this low level algorithm only considers prior observations, ignoring knowledge about the architecture, a high level algorithm will be introduced in Section ??.

Example

The Bayesian network used for this low level algorithm is represented in Figure 4.5.



**Figure 4.5:** Bayesian network graph

**Algorithm 4** Low Level Bayesian Network Algorithm

---

```

1: procedure INIT(arch, training_size)
2:   faults  $\leftarrow$  get_faults(arch)
3:   counters  $\leftarrow$  []
4:   probs  $\leftarrow$  []
5:   faults  $\leftarrow$  []
6:   revealed  $\leftarrow$  0
7:   injected  $\leftarrow$  0
8:   train_mode  $\leftarrow$  True
9:   training  $\leftarrow$  training_size
10:  iter  $\leftarrow$  0
11:  for fault in faults do
12:    counters.append([0, 0]) // initialize revealed, total to 0,0
13:    probs.append( $\frac{1}{\text{len}(\text{faults})}$ )
14:  end for
15: end procedure
16: procedure GET_EXPERIMENT
17:  index  $\leftarrow$  choice(range(len(faults)), probs) // get the index of the operation,
  using the probabilities
18:  iter  $\leftarrow$  iter + 1
19:  if iter > training then
20:    train_mode  $\leftarrow$  False
21:  end if
22:  return faults[index]
23: end procedure
24: procedure ENTER_RESULT(result)
25:  if result > 0 then
26:    counters[index][0]  $\leftarrow$  counters[index][0] + 1
27:    revealed  $\leftarrow$  revealed + 1
28:  end if
29:  counters[index][1]  $\leftarrow$  counters[index][1] + 1
30:  injected  $\leftarrow$  injected + 1
31:  if train_mode == False then
32:    for i, len(faults) do
33:      probs[i]  $\leftarrow$  update_probs()
34:    end for
35:  end if
36: end procedure

```

---

First, all probabilities will be initialized uniformly distributed. For the example architecture visualized in Figure 4.1, Table 4.1 shows the variables of each fault injection, while Table 4.5 shows the initial probability table for those variables.

$$\begin{aligned} p(fr = 1|a1 - abort) &= 0.125 \\ p(fr = 1|a1 - delay) &= 0.125 \\ p(fr = 1|a2 - abort) &= 0.125 \\ p(fr = 1|a2 - delay) &= 0.125 \\ p(fr = 1|b1 - abort) &= 0.125 \\ p(fr = 1|b1 - delay) &= 0.125 \\ p(fr = 1|c1 - abort) &= 0.125 \\ p(fr = 1|c1 - delay) &= 0.125 \end{aligned}$$

**Table 4.1:** Probabilities for variables

Then, training will be performed for the first *training\_size* generated experiments. After each generated experiment, the result will be tracked. Table 4.2 lists example results after training of 100 experiments. These results are not real and are only used for descriptive purposes.

<b>Fault Injection</b>	<b>Faults Revealed</b>	<b>Total</b>
a1-abort	13	15
a1-delay	13	14
a2-abort	15	17
a2-delay	9	12
b1-abort	8	10
b1-delay	9	11
c1-abort	1	13
c1-delay	2	8

**Table 4.2:** Observed results after 100 iterations

When training is finished, the algorithm will use learned probabilities based on the observed results. Table 4.3 lists the probabilities that will be used after training, as well as the (rounded) normalized probabilities that sum up to 1 total, the actual selection probability.

Variables	Computed	Normalized
$P(fr = 1 a1 - abort)$	0.938	0.153
$P(fr = 1 a1 - delay)$	0.968	0.158
$P(fr = 1 a2 - abort)$	0.946	0.154
$P(fr = 1 a2 - delay)$	0.875	0.142
$P(fr = 1 b1 - abort)$	0.903	0.147
$P(fr = 1 b1 - delay)$	0.913	0.149
$P(fr = 1 c1 - abort)$	0.162	0.026
$P(fr = 1 c1 - delay)$	0.4375	0.071

**Table 4.3:** Probabilities after training

### 4.3.2 High Level Learning

Building onto the previously presented low level algorithm, the high level algorithm omits the fault injection variable. Instead, it uses information about the architectural model to group fault injections into equivalence classes, and then selecting a fault injection that matches the variables selected.

The variables used are, again,  $fr$ , whether a fault has been revealed, and the architectural information variables  $cb$ ,  $dep$ , as well as the fault type variable  $ft$ .

For predicting the probability of fault injections to reveal a fault, the algorithm now predicts the probability of different parameter combinations of the variables. The  $cb$  variable is a numerical value, specifying the amount of incoming dependencies that are protected by a circuit breaker. The  $dep$  variable is the amount of incoming dependencies. The  $ft$  variable is the type of the fault, either *abort* or *delay*.

The probability of a fault being revealed is then predicted for all parameter settings:

(4.6)

$$\begin{aligned}
 P(fr = 1|cb = c \wedge dep = d \wedge ft = f) &= \frac{P(cb = c \wedge dep = d \wedge ft = f|fr = 1) \cdot P(fr = 1)}{P(cb = c \wedge dep = d \wedge ft = f)} \\
 &= \frac{P(cb = c \wedge dep = d \wedge ft = f|fr = 1)P(fr = 1)}{P(cb = c \wedge dep = d \wedge ft = f|fr = 1)P(fr = 1) + P(cb = c \wedge dep = d \wedge ft = f|fr = 0)P(fr = 0)}
 \end{aligned}$$

Here:

$$(4.7) \quad P(cb = c \wedge dep = d \wedge ft = f|fr = 1) = \frac{n(cb = c \wedge dep = d \wedge ft = f \wedge fr = 1)}{n(cb = c \wedge dep = d \wedge ft = f)}$$

$$(4.8) \quad P(fr = 1) = \frac{n(fr = 1)}{n(total)}$$

$$(4.9) \quad \begin{aligned} & P(cb = c \wedge dep = d \wedge ft = f | fr = 0) \\ &= \frac{n(cb = c \wedge dep = d \wedge ft = f) - n(cb = c \wedge dep = d \wedge ft = f \wedge fr = 1)}{n(cb = c \wedge dep = d \wedge ft = f)} \end{aligned}$$

$$(4.10) \quad P(fr = 0) = \frac{n(total) - n(fr = 1)}{n(total)}$$

Again,  $n(\dots)$  stands for the number of instances that match the parameters.

After computing all probabilities, a parameter combination is selected. Then, a fault injection, fulfilling the parameter combination, is selected and returned from all candidates, using a uniform distribution.

A training phase is also required. The same threshold approach to training will be taken, as presented for the low level algorithm.

The algorithm takes the following parameter:

- *training\_size*: This parameter defines at what threshold of iterations the trained results should be used for selection, instead of choosing random from a uniform distribution.

The following will describe the common functions from the interface.

*init*

- Initialize the parameters, including setting the training mode to on.
- Extract the variables from the operations, using the provided architectural model.
- Initialize the observation counters for the variable sets.
- Initialize the probabilities for the variable sets, using a uniform distribution.

*get\_experiment*

- Choose a variable set, using the probabilities.
- Choose a fault injection that matches the variable set.
- Increment the iteration counter.
- If the iteration counter has reached the training limit, turn the training mode off.

*enter\_result*

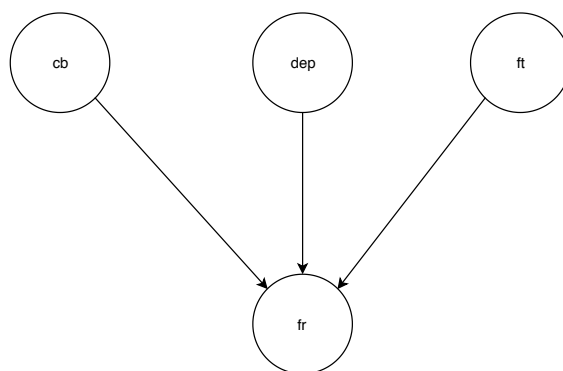
- Add the result to the observations of the last chosen variable set.
- If the training mode is off, adjust the probabilities of all variable sets. After, normalize all probabilities to sum up to 1.

Algorithm 5 presents the algorithm.

This high level algorithm is based on the assumption that the *cb*, *dep*, and *ft* variables have an impact on the probability that a fault could be revealed. Again, training is most likely necessary before returned results are good.

Example

The Bayesian network of variables used for this low level algorithm is represented in Figure 4.6.



**Figure 4.6:** Bayesian network graph

**Algorithm 5** High Level Bayesian Network Algorithm

---

```
1: procedure INIT(arch, training_size)
2:   variables  $\leftarrow$  get_variables(arch) // get the variable sets
3:   counters  $\leftarrow$  []
4:   probs  $\leftarrow$  []
5:   faults  $\leftarrow$  []
6:   revealed  $\leftarrow$  0
7:   injected  $\leftarrow$  0
8:   train_mode  $\leftarrow$  True
9:   training  $\leftarrow$  training_size
10:  iter  $\leftarrow$  0
11:  for feature in features do
12:    counters.append([0, 0]) // initialize revealed, total to 0,0
13:    probs.append( $\frac{1}{\text{len}(\text{faults})}$ )
14:  end for
15: end procedure
16: procedure GET_EXPERIMENT
17:  index  $\leftarrow$  choice(range(len(variables)), probs) // get the index of the operation,
  using the probabilities
18:  iter  $\leftarrow$  iter + 1
19:  if iter > training then
20:    train_mode  $\leftarrow$  False
21:  end if
22:  fault  $\leftarrow$  choose_fault(variables[index])
23:  return fault
24: end procedure
25: procedure ENTER_RESULT(result)
26:  if result > 0 then
27:    counters[index][0]  $\leftarrow$  counters[index][0] + 1
28:    revealed  $\leftarrow$  revealed + 1
29:  end if
30:  counters[index][1]  $\leftarrow$  counters[index][1] + 1
31:  injected  $\leftarrow$  injected + 1
32:  if not train_mode then
33:    for i, len(variables) do
34:      probs[i]  $\leftarrow$  update_probs()
35:    end for
36:  end if
37: end procedure
```

---



First, all probabilities will be initialized uniformly distributed. For the example architecture visualized in Figure 4.1, Table 4.4 shows the variables of each fault injection, while 4.5 shows the initial probability table for those variables.

Fault Injection	Variables
a1-abort	cb = 0, dep = 0, ft = abort
a1-delay	cb = 0, dep = 0, ft = delay
a2-abort	cb = 0, dep = 0, ft = abort
a2-delay	cb = 0, dep = 0, ft = delay
b1-abort	cb = 0, dep = 1, ft = abort
b1-delay	cb = 0, dep = 1, ft = delay
c1-abort	cb = 2, dep = 2, ft = abort
c1-delay	cb = 2, dep = 2, ft = delay

**Table 4.4:** Variables for fault injections

Variables	Probability
cb = 0, dep = 0, ft = abort	0.166
cb = 0, dep = 0, ft = delay	0.166
cb = 0, dep = 1, ft = abort	0.166
cb = 0, dep = 1, ft = delay	0.166
cb = 2, dep = 2, ft = abort	0.166
cb = 2, dep = 2, ft = delay	0.166

**Table 4.5:** Probabilities after initialization

Then, training will be performed for the first *training\_size* generated experiments. After each generated experiment, the result will be tracked. Table 4.6 lists example results after training of 100 experiments. These results are not real and are only used for descriptive purposes.

Variables	Faults Revealed	Total
cb = 0, dep = 0, ft = abort	14	16
cb = 0, dep = 0, ft = delay	15	17
cb = 0, dep = 1, ft = abort	15	16
cb = 0, dep = 1, ft = delay	17	18
cb = 2, dep = 2, ft = abort	2	18
cb = 2, dep = 2, ft = delay	1	15

**Table 4.6:** Observed results after 100 iterations

When training is finished, the algorithm will use learned probabilities based on the observed results. Table 4.7 lists the probabilities computed after training, as well as the normalized probabilities, summing up to 1, that are used for selection.

Variables	Computed	Normalized
cb = 0, dep = 0, ft = abort	0.926	0.227
cb = 0, dep = 0, ft = delay	0.93	0.228
cb = 0, dep = 1, ft = abort	0.964	0.236
cb = 0, dep = 1, ft = delay	0.968	0.237
cb = 2, dep = 2, ft = abort	0.181	0.045
cb = 2, dep = 2, ft = delay	0.113	0.028

**Table 4.7:** Probabilities after training

## 4.4 Reinforcement Learning

Reinforcement Learning (RL) has been introduced in Section 2.7.1. A notable feature of RL algorithms is the learning process, as training with labeled data is not necessary. The algorithms only learn from the experiences, making this an ideal approach for application in unknown systems, that do not have vast amounts of data from previous chaos experiments available.

In RL the *agent* chooses *actions* based on a *policy*. These actions are then executed in the environment, which returns a *reward*. When it comes to choosing actions, the main problem of RL, one has to balance between strictly following the knowledge from previous experiences, *exploiting* the action with the highest estimated reward, and *exploring* new actions to possibly gain better actions for future exploitation.

RL can also consider multiple states, with policies considering states for action selections, or be stateless. The following will present algorithms for experiment selection, with both purely data-driven stateless approaches, as well as approaches that consider architectural information for experiment selection. Additionally, a combination of RL and an artificial neural network classifier will be presented.

#### 4.4.1 Reward Functions

An essential part of RL is the existence of reward values for actions. For the application in chaos experiment selection, two reward functions will be proposed.

(1) The first reward function is the most simplistic binary view of experiment results as rewards. For an action  $a$ , the fault injection into the system, the reward function returns 0 if no fault can be detected in the system, or 1 if a fault can be detected at any of the integration points.

$$(4.11) \quad R_1(a) = \begin{cases} 0 & \text{no fault detected} \\ 1 & \text{fault detected at any integration point} \end{cases}$$

(2) The second reward function considers the impact of the fault injection, taking the reward beyond a binary level. For an action  $a$ , the fault injection into the system, the reward function returns  $j$ , with  $j$  being the count of integration points that show a fault. So, if no fault could be found, the reward function returns 0. If one integration point shows a fault, the reward function returns 1. If two integration points show a fault, the reward function returns 2.

$$(4.12) \quad R_2(a) = \begin{cases} 0 & \text{no fault detected} \\ j & \text{fault detected at } j \text{ integration points} \end{cases}$$

For the example architecture in Figure 4.1, integration points could be the operations  $a_1$  and  $a_2$ . Four cases are now possible:

- $a_1$  and  $a_2$  show no fault: (1) returns 0. (2) returns 0.
- $a_1$  shows a fault,  $a_2$  shows no fault: (1) returns 1. (2) returns 1.
- $a_1$  shows not fault,  $a_2$  shows a fault: (1) returns 1. (2) returns 1.
- $a_1$  and  $a_2$  show faults: (1) returns 1. (2) returns 2.

### 4.4.2 $\epsilon$ -greedy

The  $\epsilon$ -greedy algorithm is adopted from [SB+98], as described in Section 2.7.1.

The  $\epsilon$ -greedy algorithm is stateless, meaning the probability of choosing actions does not vary based on the state of the environment.

The algorithm uses the following parameters:

- $\epsilon$ : The rate of exploration, used as chance for selecting a random action.
- $\gamma$ : The discount rate, used to reduce the  $\epsilon$  rate of exploration as more about the system is known.
- $\epsilon_{min}$ : The minimum value for  $\epsilon$ . When  $\epsilon$  reaches  $\epsilon_{min}$ , the discount rate no longer reduces it any further.

The following will describe the common interface functions implemented by this algorithm.

*init*

- Initialize the algorithm parameters.
- Define the actions. For this algorithm, the actions are the fault injections, extracted from the architectural model.
- Initialize the policy. Set the estimated reward (quality)  $Q$  and counter  $N$  to 0 for all actions.

*get\_experiment*

- Choose an action either based on the highest estimated reward  $Q$  in the policy, with probability  $1 - \epsilon$ , or choose a random action for exploration, with probability  $\epsilon$ .

*enter\_result*

- Update the policy. Increment the counter  $N$ , and update the estimated reward  $Q$ . The update function for  $Q$  is

$$Q(a) = Q(a) + \frac{1}{N(a)} \cdot (result - Q(a))$$

Initial rewards have a higher priority, since the counter  $N(a)$  will be lower.

- Adjust the exploration rate, using the  $\gamma$  and  $min\_epsilon$  parameters.

$$\epsilon = (\epsilon - min\_epsilon) \cdot \gamma + min\_epsilon$$

Algorithm 6 presents the algorithm.

---

**Algorithm 6**  $\epsilon$ -greedy Bandit Algorithm

---

```

1: procedure INIT(arch,  $\epsilon$ ,  $\gamma$ , min_ $\epsilon$ )
2:   faults  $\leftarrow$  get_faults(arch)
3:   for fault in faults do
4:      $Q[fault] = 0$ 
5:      $N[fault] = 0$ 
6:   end for
7: end procedure
8: procedure GET_EXPERIMENT
9:   if random()  $< \epsilon$  then                                // Explore, choosing random action
10:    action  $\leftarrow$  random( $Q$ )
11:  else                                                       // Exploit, using known estimates
12:    action  $\leftarrow$  max( $Q$ )
13:  end if
14:  return action
15: end procedure
16: procedure ENTER_RESULT(result)
17:    $N[action] \leftarrow N[action] + 1$ 
18:    $Q[action] \leftarrow Q[action] + \frac{1}{N[action]} \cdot (result - Q[action])$ 
19:    $\epsilon \leftarrow (\epsilon - min\_epsilon) \cdot \gamma + min\_epsilon$ 
20: end procedure

```

---

The proposed  $\epsilon$ -greedy algorithm presents an approach to balancing the exploration and exploitation rate of the agent. While some fault injections may be preferred, the  $\epsilon$  parameter allows all to always be considered. This prevents any single part of the system from remaining untested.

### Example

Using the example presented in Figure 4.1, the  $\epsilon$ -greedy algorithm initializes with the following estimated rewards for actions, as shown in Table 4.8.

Action	Estimated Reward $Q(a)$
a1-abort	0
a1-delay	0
a2-abort	0
a2-delay	0
b1-abort	0
b1-delay	0
c1-abort	0
c1-delay	0

**Table 4.8:** Initial estimated rewards

When exploiting, not using the  $\epsilon$  exploration rate, a random action will be taken, since all are the same. However, after a couple of iterations, the estimated reward values will be changing. Once estimated reward values are different, as Table 4.9 shows, the actions with the highest estimated reward will be chosen with probability  $1 - \epsilon$ , and a random action will be chosen with probability  $\epsilon$ .

Action	Estimated Reward $Q(a)$
a1-abort	0.5
a1-delay	0.25
a2-abort	0.66
a2-delay	0.25
b1-abort	0.5
b1-delay	0.25
c1-abort	0
c1-delay	0

**Table 4.9:** Estimated rewards after multiple iterations

### 4.4.3 Optimistic Initial Value

The proposed  $\epsilon$ -greedy approach initializes all estimated rewards at 0, and incrementally adjusts the estimates based on experiences. Assuming that our reward function generally

returns a low reward  $r \in 0, 1, 2$ . By setting the initial value of the estimated reward to a very optimistic value, not achievable through just rewards, the rewards will then incrementally adjust the estimated reward value downwards. Here, there is no need for an  $\epsilon$  parameter. The balance between exploration and exploitation can be achieved by always picking the action with the highest estimated reward. Since all estimated reward values will be adjusted downwards, the worse actions will be adjusted downwards at a higher pace than the good actions, since the  $\delta$  between the current estimated reward, and the received reward, is larger than for good actions. This also allows for omitting of any  $\epsilon$  adjusting, as eventually the exploration rate will decrease, the closer the estimated rewards get to their true value.

Not using the exploration rate  $\epsilon$ , the optimistic initial value only requires one parameter:

- *initial\_q*: The optimistic initial value for estimated reward for all actions. *o* should be optimistic, meaning that it should be significantly higher than the reward actually expected for any action.

The following will describe the common interface functions, as implemented by this algorithm.

*init*

- Initialize the algorithm parameters.
- Define the actions. The actions are fault injections, extracted from the architectural model.
- Initialize the policy. The estimated reward values  $Q$  are initialized to the parameter *initial\_q*. The counter  $N$  is initialized to 0.

*get\_experiment*

- Choose the action based on the highest estimated reward.

*enter\_result*

- Update the policy, incrementing the counter  $N$  by 1, and adjusting the estimated reward  $Q$ . The update function is

$$Q(a) = Q(a) + \frac{1}{N(a) + 1} \cdot (result - Q(a))$$

Algorithm 7 presents the algorithm.

---

**Algorithm 7** Optimistic Initial Value Algorithm

---

```
1: procedure INIT(arch, initial_q)
2:   faults  $\leftarrow$  get_faults(arch)
3:   for fault in faults do
4:      $Q[\textit{fault}] = \textit{initial\_q}$ 
5:      $N[\textit{fault}] = 0$ 
6:   end for
7: end procedure
8: procedure GET_EXPERIMENT
9:   action  $\leftarrow$  max( $Q$ )
10:  return action
11: end procedure
12: procedure ENTER_RESULT(result)
13:    $N[\textit{action}] \leftarrow N[\textit{action}] + 1$ 
14:    $Q[\textit{action}] \leftarrow Q[\textit{action}] + \frac{1}{N[\textit{action}] + 1} \cdot (\textit{result} - Q[\textit{action}])$ 
15: end procedure
```

---

Again, the optimist initial value algorithm provides a simple approach to experiment selection. The hope is that the agent naturally finds the optimal actions at a faster rate, than  $\epsilon$  balancing of exploration and exploitation may achieve. However, optimistic initial value selection still comes with the inherent advantages of reinforcement learning: Always consider all options. This allows no part of the system to remain untested, as all fault injections are considered.

### Example

In comparison to the previous  $\epsilon$ -greedy algorithm, the estimated reward values will now not be initialized to 0. Let the parameter *initial\_q* be 5. Then, all actions will be initialized with an estimated reward of 5, as shown in Table 4.10.



Action	Estimated Reward $Q(a)$
a1-abort	5
a1-delay	5
a2-abort	5
a2-delay	5
b1-abort	5
b1-delay	5
c1-abort	5
c1-delay	5

**Table 4.10:** Initial estimated rewards for actions

However, this time there is no exploration rate. Instead, the action with the highest estimated reward will always be chosen. This causes the results of different runs to be similar, as there is no random variance in selecting experiments. If the rewards for actions are always the same, the algorithm will return the same results for different runs, the only difference being the random selection when multiple actions have the same estimated reward. Table 4.11 shows possible values of the estimated rewards getting lower, instead of higher, after a few iterations.

Action	Estimated Reward $Q(a)$
a1-abort	1.5
a1-delay	3
a2-abort	3
a2-delay	2
b1-abort	2
b1-delay	3
c1-abort	2.5
c1-delay	2.5

**Table 4.11:** Estimated rewards for actions after multiple iterations

#### 4.4.4 Q-Learning

Q-Learning introduces the notion of multiple states and transitions between states to the algorithms presented. Instead of the single state approaches taken with  $\epsilon$ -greedy and optimistic initial value, Q-Learning now attempts to find the best  $Q$ , considering states and actions that trigger transitions between states.

Usually states can be more obvious descriptions of the environment, e.g., the current figure layout of a chess board. However, we do not have this kind of direct simple state representation for our problem. To represent states with knowledge about the architecture, the following representation is used:

Our states are nodes in the dependency graph of the operations. Outgoing transitions, or actions of a state, to other states are the incoming dependencies of an operation in the dependency graph. If an operation  $c$  has incoming dependencies from  $a$  and  $b$ , the state  $c$  will have the actions  $a$  and  $b$ . When choosing action  $a$ , the state will transition to the next state,  $a$ . Upon choosing an action, a random fault, either *abort* or *delay*, will be injected into the current state, and the state will change to the chosen state.

The end of a sequence is reached if a terminal state has been reached. A terminal state is a state that does not have any actions (an operation without incoming dependencies). Then, a random state will be chosen to continue.

The idea behind this modelling of states and actions is to take real paths through the system, injecting faults along the way. Q-Learning should then allow the algorithm to find the most vulnerable paths.

The proposed Q-Learning approach has the following parameters:

- $\epsilon$ , as previously introduced, chooses random actions from the available actions with probability  $\epsilon$ . Contrary to the  $\epsilon$  in  *$\epsilon$ -greedy* however, there is no decay present.
- $\gamma$ , the discount rate, impacts the weighting of future values. When calculating the new  $Q$  value of a state-action pair, both the received reward, and the highest possible state-action value of the new state are considered.  $\gamma$  has an impact on weighing the the highest possible state-action value of the new state. A higher value for  $\gamma$  increases the impact of the new state, while a low  $\gamma$  prioritizes the reward received.
- $learning\_rate \in (0, 1]$  is used to weigh the previous  $Q$  value for a state-action pair in comparison to the received reward in combination with the highest possible state-action value of the new state. A low value for  $learning\_rate$  prioritizes prior knowledge, while a higher value for  $learning\_rate$  prioritizes newly learned results.
- $initial\_q$  is the initial  $Q$  value for state-action pairs.

The following will describe the common interface functions, as implemented by this algorithm.

*init*

- Initialize the algorithm parameters.
- Initialize the states into a dictionary. States, operations, are keys, and have  $Q$  and  $N$  dictionaries as value. These  $Q$  and  $N$  dictionaries track the estimated reward and counter for the actions, incoming dependencies.
- Initialize the  $Q$  and  $N$  values for all state-action pairs. The  $Q$  value is initially set to the parameter *initial\_q* for all state-action pairs, while  $N$  starts at 0.
- Choose a random first state from the eligible states (states that have actions), to start a new episode.

*get\_experiment*

- When starting a new episode, return the current state with a random fault type from *abort* and *delay* as fault injection.
- Else: Select a new state from the current state's actions. With probability  $1 - \epsilon$  choose the next state with the highest estimated reward, with probability  $\epsilon$  choose a random next state from the available actions.
- Choose a random fault type from *abort* and *delay*.
- Return the next state, appended by the randomly selected fault type, as fault injection.

*enter\_result*

- If a new episode started, skip processing the result, as no action was taken to get there.
- Else: Receive the reward value  $R$ .
- Update the  $Q$  and  $N$  values for the state-action (state-nextstate) pair, using the information about the new state. The update function is

$$\begin{aligned}
 &Q(\text{state}, \text{next\_state}) \\
 &= (1 - \text{learning\_rate}) \cdot Q(\text{state}, \text{next\_state}) \\
 &\quad + \text{learning\_rate} \cdot (\text{result} + \gamma \cdot \max_a Q(\text{next\_state}) - Q(\text{state}, \text{next\_state}))
 \end{aligned}$$

Here,  $\max_a Q(\text{next\_state}, a)$  takes the maximum estimated reward of all actions in the next state. This considers how good the next step(s) will be. If the next state does not have any actions, i.e. it is terminal, use 0.

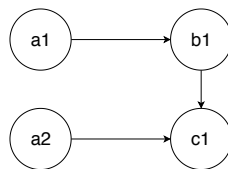
- If the next state, the selected action, is a terminal state, i.e. it doesn't have any actions, choose a new random state. This will start a new episode.
- Change the current state to the next state.

Algorithm 8 presents the algorithm.

Q-Learning, contrary to the previously introduced reinforcement learning approaches, utilizes information about the architecture and separates the system into multiple states. The hope is for Q-Learning to be able to find the paths through the architecture that reveal the most faults by backpropagating through dependencies. The choice to go backwards, choosing incoming dependencies as actions, instead of going forward through outgoing dependencies, as LDFI 3.1 does, is based on the assumptions that fault injections into services with incoming dependencies have a higher impact than fault injections into services without incoming fault injections, and finding faults further back may reveal faults in multiple integration points.

### Example

For the example architecture in Figure 4.1, the following dependency graph in Figure 4.7 would be used.



**Figure 4.7:** Dependency graph for example architecture

From this dependency graph, the following states and actions will be extracted:

State	Actions
a1	-
a2	-
b1	a1
c1	b1,a2

**Table 4.12:** States and actions for example architecture

**Algorithm 8** Q-Learning Algorithm

---

```

1: procedure INIT(arch,  $\epsilon$ ,  $\gamma$ , learning_rate, initial_q)
2:   states  $\leftarrow$  init_states(arch)
3:   for state in states do
4:     for action in state.actions do
5:        $Q[s, a] \leftarrow$  initial_q
6:        $N[s, a] \leftarrow 0$ 
7:     end for
8:   end for
9:   new_episode  $\leftarrow$  True
10:  state  $\leftarrow$  random(states)
11: end procedure
12: procedure GET_EXPERIMENT
13:  type  $\leftarrow$  random(['abort', 'delay'])
14:  if new_episode == True then
15:    action = state
16:    return state + ' -' + type
17:  end if
18:  if random() <  $\epsilon$  then                                     // Explore, choosing random action
19:    action  $\leftarrow$  random(state.actions)
20:  else                                                         // Exploit, using known estimates
21:    action  $\leftarrow$  max(state.actions, Q)
22:  end if
23:  return action + ' -' + type
24: end procedure
25: procedure ENTER_RESULT(result)
26:  if new_episode == True then                               // if in new episode, don't process the result,
  since no actual action was taken to get there
27:    new_episode  $\leftarrow$  False
28:    if state is terminal then                               // start new episode if state is terminal
29:      state  $\leftarrow$  random(states)
30:      new_episode  $\leftarrow$  True
31:    end if
32:    return
33:  end if
34:   $N[\textit{state}, \textit{action}] \leftarrow N[\textit{state}, \textit{action}] + 1$ 
35:  update  $Q[\textit{state}, \textit{action}]$ 
36:  state  $\leftarrow$  action
37:  if state is terminal then
38:    state  $\leftarrow$  random(states)
39:    new_episode  $\leftarrow$  True
40:  end if
41: end procedure

```

---

Let the current state be  $c1$ .  $c1$  then has the actions  $b1$  and  $a2$  with estimated rewards of 0.5 and 0.25. With a probability of  $\epsilon$ , either  $b1$  or  $a2$  will be chosen. Else,  $b1$  will be chosen, as it has the highest estimated reward. For this example,  $b1$  will be chosen.

After choosing the action, a random fault type from  $abort, delay$  will be selected, e.g.  $abort$ . Then  $b1$ - $abort$  will be returned as fault injection.

Let the reward be 1, i.e. a fault has been revealed. The estimated reward value for the state-action pair  $c1, b1$  will then be updated, and the new state will be set to  $b1$ . Since  $b1$  is a valid state, i.e. it has actions, the iteration finishes at this point. If  $b1$  was not valid, i.e. it didn't have any actions, a valid state would be chosen at random.

### 4.4.5 Artificial Neural Network

This approach is heavily based on [SGMM17b], altered for the problem of selecting fault injections.

The core component of this algorithm is an artificial neural network multi-layer perceptron classifier. The input to the classifier are features about fault injections into operations that were previously extracted from the architectural model. All features considered are:

- $cb$ : The amount of dependencies that are protected by a circuit breaker for this operation.
- $dep$ : The amount of incoming dependencies of the operation.
- $hist$ : The results of the last 4 fault injections of this type into this operation.
- $inst$ : The instances of the service that the operation belongs to.
- $fault\_type$ : The type of the fault injection, either 1 for  $abort$  or 0 for  $delay$ .

Using these features as input the classifier returns a percentage value. This percentage value is generated for the features of all possible fault injections, and the resulting probabilities are used for selecting a fault injection.

Training is required for this approach, potentially making this only suitable for large architectures with time to train. Similar to the previously introduced training of the Bayesian algorithms, training occurs during the initial selections.

While in training mode, training happens on every fifth iteration by sampling a batch of memories. Memories are experiences: fault injections and their corresponding reward. When the memory capacity has been reached, older experiences will start getting replaced by new experiences.

The algorithm has the following parameter:

- *hidden\_size*: The amount of hidden layers in the multi-layer perceptron neural network. Hidden layers were explained in 2.7.4.

The following will describe the common interface functions, as implemented by this algorithm.

*init*

- Initialize the algorithm variables.
- Initialize an empty memory.
- Initialize the multi-layer perceptron classifier, using the *hidden\_size* parameter.
- Generate all fault injections from the provided architectural model.

*get\_experiment*

- Do this for all fault injections:
  - If this is the first selection (model has not been fitted yet): Choose a random probability percentage.
  - Else: Pass the extracted features into the classifier, receive the probability percentage.
- Normalize the probabilities of the fault injections, so they sum up to 1.
- Use the normalized probabilities to choose a fault injection.
- Remember the fault injection selected for future learning.

*enter\_result*

- Add the (fault injection, reward) pair to the memory.
- Learn from a sample batch of experiences from the memory on every fifth (and the first) iteration.

Algorithm 9 presents the algorithm.

This neural network based approach is the first reinforcement learning approach to utilize all the available architectural data.

**Algorithm 9** Artificial Neural Network Algorithm

---

```
1: procedure INIT(arch, hidden_size)
2:   iterations  $\leftarrow$  0
3:   model  $\leftarrow$  PerceptronClassifier(hidden_size)
4:   model_fit  $\leftarrow$  False
5:   experiences  $\leftarrow$  []
6:   features  $\leftarrow$  get_features(arch)           // get feature sets for the fault injections
7: end procedure
8: procedure GET_EXPERIMENT
9:   probs  $\leftarrow$  []
10:  for feature in features do
11:    if model_fit == True then
12:      probs.append(model.predict_proba(feature))
13:    else
14:      probs.append(random())
15:    end if
16:  end for
17:  total  $\leftarrow$  sum(probs)
18:  for prob in probs do                               // normalize probabilities
19:    prob  $\leftarrow$   $\frac{prob}{total}$ 
20:  end for
21:  feature  $\leftarrow$  random(features, probs)
22:  fault  $\leftarrow$  find_fault(feature)
23:  return fault
24: end procedure
25: procedure ENTER_RESULT(result)
26:   iterations  $\leftarrow$  iterations + 1
27:   experiences.append(feature, result)
28:   update_history(feature, result)
29:   if iterations%5 == 0 then
30:     train()
31:   end if
32: end procedure
33: procedure TRAIN
34:   model.train(experience)
35:   model_fit  $\leftarrow$  True
36: end procedure
```

---



### Example

For the example architecture in Figure 4.1, the following fault injections and features would be used by the artificial neural network algorithm:

a1-abort	$cb = 0, dep = 0, hist = 0, 1, 1, 0, inst = 1, fault\_type = 1$
a1-delay	$cb = 0, dep = 0, hist = 0, 1, 0, 0, inst = 1, fault\_type = 0$
a2-abort	$cb = 0, dep = 0, hist = 1, 1, 1, 0, inst = 1, fault\_type = 1$
a2-delay	$cb = 0, dep = 0, hist = 0, 0, 0, 1, inst = 1, fault\_type = 0$
b1-abort	$cb = 0, dep = 1, hist = 1, 1, 1, 0, inst = 1, fault\_type = 1$
b1-delay	$cb = 0, dep = 1, hist = 0, 0, 1, 0, inst = 1, fault\_type = 0$
c1-abort	$cb = 2, dep = 2, hist = 0, 1, 0, 0, inst = 1, fault\_type = 1$
c1-delay	$cb = 2, dep = 2, hist = 0, 1, 0, 1, inst = 1, fault\_type = 0$

**Table 4.13:** Feature extractions of fault injections

When choosing a fault injection, the artificial neural network will be used to compute priorities for all feature sets. These priorities will then be normalized and used to select fault injections.

After selecting a feature set, a matching fault injection will be chosen based on the selected feature set. Table 4.13 shows the extraction of the features.

When receiving the reward, the artificial neural network will be retrained on every fifth iteration, based on the reward received for feature sets selected.

#### 4.4.6 Tableau

The tableau algorithm, also adapted from [SGMM17b], takes a similar approach to the previously presented neural network based algorithm in Section 4.4.5. Originally presented for prioritizing test cases to use in test sequences, the same prioritization idea is adopted for selecting a fault injection. Again, features are extracted from fault injections and used, grouping multiple fault injections based on features.

Applied to the reinforcement learning concept of states and actions, states are features sets and actions are priority groups, with the amount of groups being a parameter. When selecting a fault injection, this algorithm categorizes all features sets into priority groups, choosing the priorities by using reinforcement learning's estimated reward with  $\epsilon$ -greedy exploration. Then, a feature set is taken at random from the highest priority group, and a matching fault injection is selected, at random if multiple match.

The features used are the following:

- *cb*: The amount of incoming dependencies that are protected by a circuit breaker pattern.
- *dep*: The amount of incoming dependencies of an operation.
- *ft*: The fault type of a fault injection.

The algorithm has the following parameters:

- $\epsilon$ : The exploration rate, used for choosing random actions with probability  $\epsilon$ .
- $\gamma$ : The discount rate, used for adjusting of the  $\epsilon$  exploration rate. As the algorithm continues to learn, it will explore less and less.
- *min\_* $\epsilon$ : The minimum value of  $\epsilon$ . When reached, the discount rate  $\gamma$  will no longer reduce  $\epsilon$  any further.
- *initial\_* $q$ : The optimistic initial estimated reward to be assigned for all action value pairs  $Q(s, a)$  during initialization.
- *action\_**size*: The range of options to choose from. An action is an integer number, assigning the state (fault injection) to a priority.

The following will describe the common interface functions, as implemented by this algorithm.

*init*

- Initialize the algorithm parameters.
- Generate all fault injections from the provided architectural model.
- Iterate over the fault injections and retrieve the feature sets, states.
- Iterate over the states, and initialize the  $Q$  and  $N$  arrays for each, with the amount of actions being the *action\_**size* parameter. Initialize  $Q$  to *initial\_* $q$  and  $N$  to 0.

*get\_experiment*

- Iterate over all states, choosing an action (priority) for each. Choose the action with the highest estimated reward with probability  $1 - \epsilon$ . Choose a random action with probability  $\epsilon$ .
- Sort the states into their priority groups.
- Randomly choose a state from the highest non-empty priority group.
- Choose a fault injection that matches the state, feature set. If multiple match, choose one at random.

*enter\_result*

- Update the  $Q$  and  $N$  values of the state-action pair selected, using the update function:

$$Q(s, a) = Q(s, a) + \frac{1}{N(s, a) + 1} \cdot (result - Q(s, a))$$

Algorithm 10 presents the algorithm.

The presented tableau algorithm is similar to the previously introduced *optimistic initial value* algorithm. However, instead of choosing actions as fault injections directly, priority numbers are used to prioritize fault injections.

Example

Table 4.14 shows the extracted features for the fault injections into the example architecture in Figure 4.1.

**Algorithm 10** Tableau Algorithm

---

```
1: procedure INIT(arch,  $\epsilon$ ,  $\gamma$ , initial_q, min_ $\epsilon$ , action_size)
2:   features  $\leftarrow$  get_features(arch)           // get feature sets for the fault injections
3:   for feature in features do
4:     for a in range(action_size) do
5:        $Q[\textit{feature}, a] \leftarrow \textit{initial\_q}$ 
6:        $N[\textit{feature}, a] \leftarrow 0$ 
7:     end for
8:   end for
9: end procedure
10: procedure GET_EXPERIMENT
11:   lists  $\leftarrow$  init_lists(action_size)
12:   for feature in features do
13:     if random()  $< \epsilon$  then
14:       priority = random(action_size)           // choose random action
15:     else
16:       priority = argmax_a( $Q[\textit{feature}, a]$ )   // get the action with the highest
estimate
17:     end if
18:     lists[priority].append(feature)
19:   end for
20:   for i, action_size - 1 do
21:     if len(lists[i])  $\neq 0$  then
22:       list = lists[i]
23:       action = i
24:       break
25:     end if
26:   end for
27:   feature  $\leftarrow$  random(list)
28:   fault  $\leftarrow$  find_fault(feature)
29:   return fault
30: end procedure
31: procedure ENTER_RESULT(result)
32:    $N[\textit{feature}] \leftarrow N[\textit{feature}] + 1$ 
33:    $Q[\textit{feature}, \textit{action}] \leftarrow Q[\textit{feature}, \textit{action}] + \frac{1}{N[\textit{feature}, \textit{action}] + 1} \cdot (\textit{result} -$ 
 $Q[\textit{feature}, \textit{action}])$ 
34:    $\epsilon \leftarrow (\epsilon - \textit{min\_}\epsilon) \cdot \gamma + \textit{min\_}\epsilon$ 
35: end procedure
```

---

Fault Injection	Features
a1-abort	$cb = 0, dep = 0, ft = abort$
a1-delay	$cb = 0, dep = 0, ft = delay$
a2-abort	$cb = 0, dep = 0, ft = abort$
a2-delay	$cb = 0, dep = 0, ft = delay$
b1-abort	$cb = 0, dep = 1, ft = abort$
b1-delay	$cb = 0, dep = 1, ft = delay$
c1-abort	$cb = 2, dep = 2, ft = abort$
c1-delay	$cb = 2, dep = 2, ft = delay$

**Table 4.14:** Extracted features for fault injection

Let  $action\_size$  be 2. Then, each of the states, or feature sets, will have two actions. Each action is a priority group to group the feature set into. When selecting a fault injection, the algorithm will iterate over all states, feature sets, and select actions, priority groups, based on the estimated reward and exploration rate  $\epsilon$  for each state-action, feature set-priority, pair. After, one feature set from the highest non-empty priority group will be chosen, and a matching fault injection will be returned.

Group	Feature Sets
0	$(0, 1, abort), (0, 0, abort)$
1	$(0, 0, delay), (0, 1, delay), (2, 0, abort), (2, 0, delay)$

**Table 4.15:** Priority groups for features

If the fault injection  $b1 - abort$  is returned, and the priority group for the feature set matching was 0, the estimated reward value for the action 0 of state  $b1 - abort$  will be updated with the result.

## 4.5 Multilevel Feedback Queue

Originally used for process scheduling, using CPU time occupation for priority determination, the multilevel feedback queue allows for scheduling based on prior results. This can be adapted to chaos experiment scheduling by using fault injection results to change the priority of future fault injections.

The following rules are proposed for fault injection selection, using MLFQ:

- All fault injections start in the highest priority queue.
- All queues are assigned a probability of selection. For queues  $q_p$  with priorities  $p \in 1, 2, 3, 4, 5, \dots$  the probability of selection halves for each priority level,  $p(q_i) = 2 * p(q_{i+1})$ , i.e.  $p(q_1) = 2 \cdot p(q_2) = 2 \cdot (2 \cdot p(q_3)) = 2 \cdot (2 \cdot (2 \cdot p(q_r))) = \dots$
- All queues use FIFO scheduling.
- If a fault injection reveals a fault, it moves up to the next highest priority queue, unless it already is in the highest queue.
- If a fault injection doesn't reveal a fault, it moves to the next lower priority queue, unless it already is in the lowest queue.

The algorithm takes the following parameters:

- *num\_queues*: The amount of priority queues to use.
- *iterations\_reset*: The point at which to reset the queues, sorting all fault injections into the highest priority queue.

The following will describe the common interface functions, as implemented by this algorithm.

*init*

- Initialize the queues and assign probabilities based on  $p(q_i) = 2 * p(q_{i+1})$ .
- Add all fault injections to the first queue.

*get\_experiment*

- Select a queue based on the previously assigned priorities.
- Take the fault injection at the front of the queue, following FIFO scheduling.
- Remember the fault injection for re-scheduling.
- Increment the iteration counter.

*enter\_result*

- Add the fault injection to the queue, based on the previously defined rules. If the fault injection revealed a fault, move to the next higher queue. If the fault injection did not reveal a fault, lower the priority and move to the next lower queue.
- If the iteration limit has been reached, reset the queues.

Algorithm 11 presents the algorithm.

---

**Algorithm 11** MLFQ Algorithm

---

```

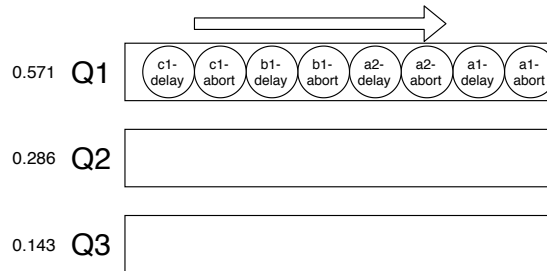
1: procedure INIT(arch, num_queues, iterations_reset)
2:   queues  $\leftarrow$  init_queues(num_queues, arch)
3:   iterations  $\leftarrow$  0
4: end procedure
5: procedure GET_EXPERIMENT
6:   iterations  $\leftarrow$  iterations + 1
7:   for i, num_queues - 1 do
8:     if len(queues[i]) > 0 then
9:       action = queues[i].get()
10:      queue = i
11:      return action
12:    end if
13:  end for
14: end procedure
15: procedure ENTER_RESULT(result)
16:   if result > 0 then
17:     if queue > 0 then
18:       queues[queue - 1].put(action)
19:     else
20:       queues[queue].put(action)           // already in highest queue
21:     end if
22:   else
23:     if queue < num_queues - 1 then
24:       queues[queue + 1].put(action)
25:     else
26:       queues[queue].put(action)         // already in lowest queue
27:     end if
28:   end if
29: end procedure

```

---

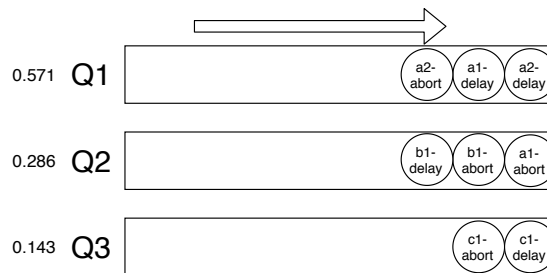
## Example

At first, all fault injections are put into the highest level queue, Figure 4.8 visualizes this for the example architecture in Figure 4.1 with the parameter  $num\_queue = 3$ . Then,



**Figure 4.8:** MLFQ for example architecture after initialization

using the probabilities of the queues, a queue is selected and the front fault injection in the queue is returned as next fault injection. When receiving the result, the fault injection is then returned into the appropriate queue.



**Figure 4.9:** MLFQ for example architecture after multiple iterations

For example, Figure 4.9 shows the state of the queues after a few iterations. The next queue selected to take an element from is the second queue, the front element is *b1-abort*. After returning *b1-abort*, a result of 1 is received, a fault has been revealed. Now, *b1-abort* is sorted into the next higher queue, the first queue. If the result was 0, *b1-abort* would have been put into the third queue instead.



## Chapter 5

# ORCAS Decision Engine

---

The decision engine is responsible for chaos experiment selection in the ORCAS framework. As part of that it receives an architecture model from the architecture extractor, and provides the input for the simulator.

Outside of selecting chaos experiments, the chaos experiments also have to be run.

## 5.1 Chaos Experiments

Chapter 4 presented various approaches to selecting fault injections. However, a chaos experiment requires more than the injection of a fault. As introduced in Section 2.4, a chaos experiment is made up of two main parts:

- (1) A steady-state that is a measurable output of a system, and indicates a healthy system.
- (2) An action that introduces chaos to the system.

While (2) has been addressed with the selection of a fault injection, (1) is still outstanding. As most systems differ, it is currently not possible to introduce a general steady-state. As a result, the steady-state has to be user defined, for now.

Examples of good actions for testing the steady-state of a system include the play button for a video streaming service, or the order function of an online store. Both of those are the essential functions of the respective product, business metrics that are directly related to the service the product provides. Using either of them triggers a whole chain of events, making them ideal candidates, as a lot of the system can be reached within one call.

### 5.1.1 Running Chaos Experiments

For automating chaos experiments, two components are necessary.

(1) an automator for chaos experiments, orchestrating and automating the steady-state hypothesis check prior and post experiment, as well as running the action that introduces chaos to the system.

(2) a fault injection tool, that the chaos experiment tool can call to inject faults.

For (1), ChaosToolkit<sup>1</sup> provides an open platform for automating chaos experiments, with extensions available for many popular, and related, platforms, including various clouds and related tools like Instana and Istio. Netflix has also announced a chaos automation platform, ChAP, which will use Netflix's Failure Injection Tool (FIT) to inject faults [BBHR16]. Gremlin<sup>2</sup>, originally used for injecting faults, also allows for scheduling of fault injections, as well as integration into continuous delivery pipelines [Laf17]. Being open to the public and integrating with most chaos related tools, ChaosToolkit appears to be the best option for automating chaos experiments generated by the decision engine.

For (2) there are multiple options. The previously mentioned FIT adds latency or error codes to requests by adding HTTP headers to requests [RHB+17]. Netflix recommends Istio<sup>3</sup> as an open alternative to their proprietary FIT tool. Istio itself is a service mesh tool, adding sidecars to services and taking over the network communication functionality. By handling communication, Istio can add delays or errors to requests. Rosenthal et al. [RHB+17] also lists LinkedIn's Simoorg, Pumba, Chaos Lemur, Chaos Lambda, Blockade, Chaos-http-proxy, Monkey-ops, Chaos Dingo, and Tugbot as tools that can be used for introducing chaos on various levels [HADP18].

#### ChaosToolkit

As the scope of this thesis does not extend to a full implementation of the decision engine, only a draft for converting chaos experiments to the model used by ChaosToolkit will be provided.

The ChaosToolkit tool relies on a chaos experiment file in a JSON format. For this use case, the following template for this experiment can be used:

<sup>1</sup><https://chaostoolkit.org>

<sup>2</sup><https://www.gremlin.com>

<sup>3</sup><https://istio.io>

```

{
  "title": "chaosExperiment",
  "version": "1.0.0",
  "description": "Generated experiment",
  "steady-state-hypothesis": {
    "title": "ssh",
    "probes": [
      {
        "type": "probe",
        "tolerance": "0",
        "name": "health-check",
        "provider": {
          "type": "python",
          "module": "health_check",
          "func": "get_health"
        }
      }
    ]
  },
  "method": [
    {
      "type": "action",
      "name": "inject-fault",
      "provider": {
        "type": "python",
        "module": "fault_injector",
        "func": "inject_fault",
        "arguments": {
          "service": "C",
          "operation": "c2",
          "action": "delay"
        }
      }
    }
  ]
}

```

This chaos experiment model is made up of three main components:

- *metadata*: The metadata, i.e. the title, version, description, are required fields that describe the experiment
- *steady-state-hypothesis*: The steady state hypothesis describes how the systems health should be checked. Here, multiple probes can be defined. Each probe contains a provider, either a Python function, HTTP call, or process call, that is to be called as part of this probe. For the use case of ORCAS, ideally the framework would contain a module for generating a steady state hypothesis based on the given architectural model, e.g. identifying key components that could be queried.

- *method*: The method is the fault injection, again including a provider that is either a Python function, HTTP call, or process call. For ORCAS, this would depend on how which fault injection tool would be used, e.g. Istio or Gremlin.

An additional *rollback* component could define actions to reset the system, disabling any active fault injections.

After passing the chaos experiment model to the ChaosToolkit, a *journal.json* file describes the experiment results. This file can then be parsed to retrieve the experiment results, that are to be passed back to the decision engine.

## 5.2 Integration with MiSim Simulator

The MiSim Simulator, developed by Beck, Günthör, and Zorn [BGZ17], simulates the injection of aborts into services, and supports the circuit breaker pattern. It does not support the injection of faults into single operations. As the result of the algorithms presented in Chapter 4 is the injection of either an abort or delay fault into an operation, a direct conversion for an experiment into the simulator is not possible. Instead, an abort can be injected into the service of the operation, indirectly injecting it into the operation. However, other operations may be affected too. Injecting a delay is currently not possible.

Section 2.6.3 described the experiment model of the simulator. The *chaosmonkey* component could currently be generated by taking the service of a fault injection, and shutting down all instances. The *request\_generators* component could be populated with operations that depend on the operation that a fault injection would be injected to. Taking the example in Figure 4.1, the following experiment model could be defined for the fault injection *c1-abort*, with *a2* and *b1* depending on *c1*.

```
{
  "simulation_meta_data": {
    "experiment_name": "ABC Experiment",
    "model_name": "Schema ABC",
    "duration": 50,
    "report": "",
    "datapoints": 50,
    "seed": 979
  },
  "request_generators": [
    {
      "microservice": "B",
      "operation": "b1",
      "interval": 0.25
    }
  ],
}
```

```
{
  "microservice": "A",
  "operation": "a2",
  "interval": 0.25
}
],
"chaosmonkeys": [
  {
    "microservice": "C",
    "instances": 1,
    "time": 10
  }
]
}
```

However, since neither operations, nor delays, are currently supported by the simulator, a full model conversion at this point does not make sense.



## Chapter 6

# Evaluation

---

This chapter will cover the evaluation of the algorithms previously introduced in Chapter 4.

During this evaluation, the following research questions will be reviewed:

- RQ1: Which algorithm has the highest fault detection rate??
- RQ2: Is there a noticeable difference between algorithms that use architectural information, and those that do not?
- RQ3: How efficient are the algorithms?

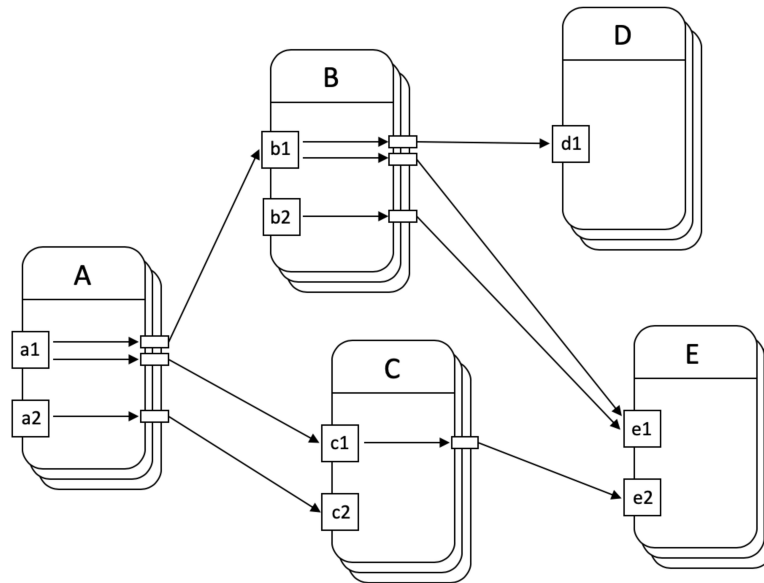
First, the experiment setting and metrics to be measured will be defined. Then, the measured metrics will be presented. Finally, the results will be discussed, and the threats to validity will be addressed.

## 6.1 Experiment Settings

This section will describe the experiment settings. The framework introduced in Chapter 4 will be used.

### 6.1.1 Architectures

The ORCAS project contains example fault injection data for the reference architecture visualized in Figure 6.1.



**Figure 6.1:** Reference architecture [Bec18]

For the reference architecture in Figure 6.1, fault injections have been run and data has been collected. This example setup <sup>1</sup> uses Hystrix <sup>2</sup> for circuit breaker pattern implementations, and Istio <sup>3</sup> for fault injections. The fault injections *abort*, returning a 500 error code, and *delay*, injecting an artificial 2 second delay, were used.

Results of fault injections were measured by sending requests to the operations of the service *a*, *a1* and *a2* respectively, performing a check on the steady-state. The operations *a1* and *a2* are considered as integration points.

As *a1* and *a2* depend on the operations of the other services, a fault in one of the latter operations will be revealed when querying either *a1* or *a2*. The exception for this is the operation *b2*. As a result, fault injections into *b2* will be omitted, as no data for those are available. Furthermore, fault injections into *a1* and *a2* will be omitted, as they are used as integration points for measuring of the steady-state of the system.

Hystrix was used to implement circuit breakers in the following operations: *a1*, *a2*, *b1*, *c1*. Toggling the circuit breaker for different experiment runs was supposed to allow for 16 different architectures, with the circuit breaker status varying for each

<sup>1</sup><https://github.com/orcas-elite/example-setups>

<sup>2</sup><https://github.com/netflix/hystrix>

<sup>3</sup><https://istio.io>



architecture. Example circuit breaker states are:  $a1false-a2false-b1false-c1false$ , i.e. no circuit breaker at all,  $a1true-a2false-b1false-c1false$ , i.e. a circuit breaker implemented for the  $a1$  operation, or  $a1true-a2true-b1true-c1true$ , i.e. a circuit breaker implemented for the operations  $a1$ ,  $a2$ , and  $b1$ , however not for  $c1$ .

However, because the way circuit breakers work, not all 16 architectures can be considered for evaluation, since some will return equal results. A circuit breaker changes to the open state, if a percentage of requests time out, or returns errors. Since all operations depend, directly or in-directly, on either  $a1$ , or  $a2$ , a single circuit breaker in  $a1$  will show the same results as circuit breakers in  $a1$  and any operation that depends  $a1$  (in-directly) depends on. This is because any error that would open a circuit breaker, will also open the circuit breaker in  $a1$ .

As an example: If operations  $a1$  and  $b1$  have a circuit breaker implemented, and  $a1$  has a dependency on  $b1$  that will always be used, the circuit breaker in  $b1$  will show no difference in results, since any error found in  $b1$  would also be found in  $a1$ , opening the circuit breaker in  $a1$ .

Since circuit breakers in all operations result in a healthy behavior at all times, assuming the circuit breakers work as expected. Similar, circuit breakers for no operations results in a fault every time, since the system is not resilient at all. As a result, those architectures will be omitted, too.

As a result, only the following architectures will be considered:

- *arch01*: The circuit breaker in the integration point  $a1$  will be activated, and apply for calls to  $b1$ ,  $c1$ ,  $d1$ ,  $e1$ , and  $e2$ .
- *arch02*: The circuit breaker in the integration point  $a2$  will be activated, and apply for calls to  $c2$ .
- *arch03*: The circuit breaker in  $b1$  will be activated, and apply for calls to  $d1$  and  $e1$ .
- *arch04*: The circuit breaker in  $c1$  will be activated, and apply for calls to  $e2$ .
- *arch05*: The circuit breaker in  $b1$  and  $c1$  will be activated, and apply for calls to  $d1$ ,  $e1$ , and  $e2$ .

The circuit breaker implementations are visualized by the switch icon, while the remote calls that are affected by the circuit breaker are colored green.

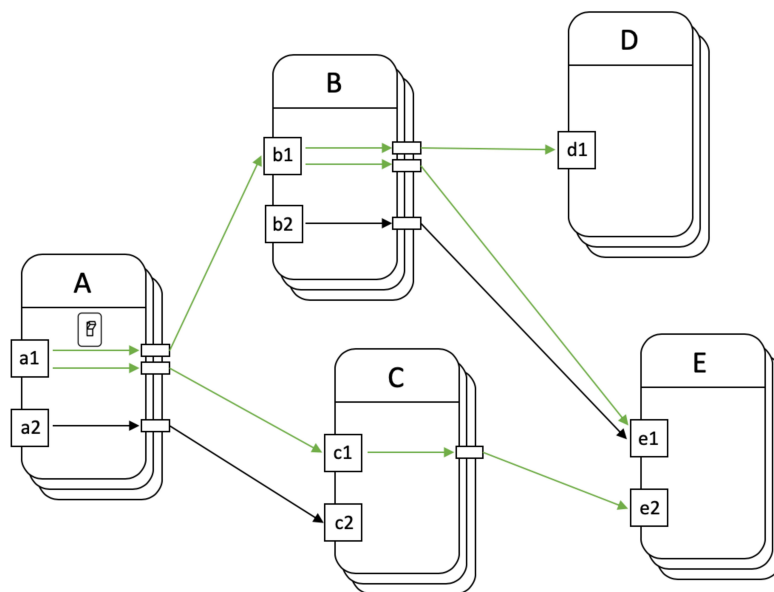


Figure 6.2: *arch01*

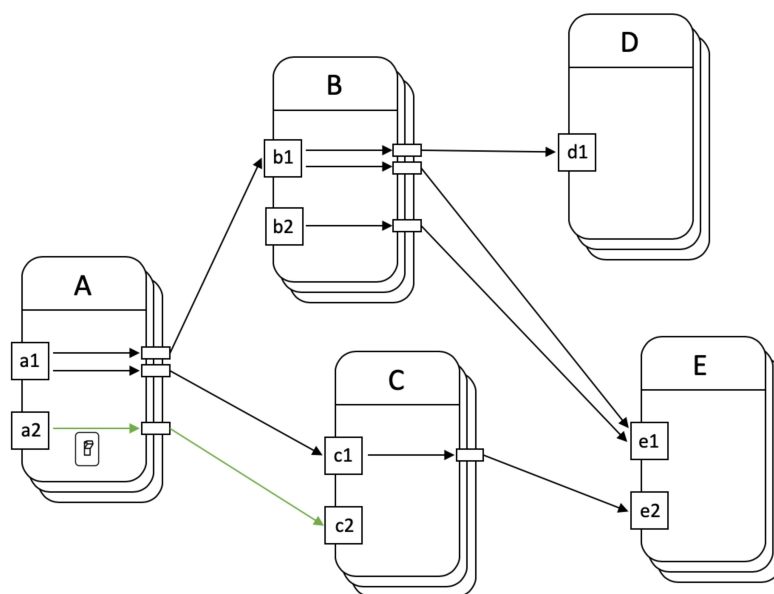


Figure 6.3: *arch02*

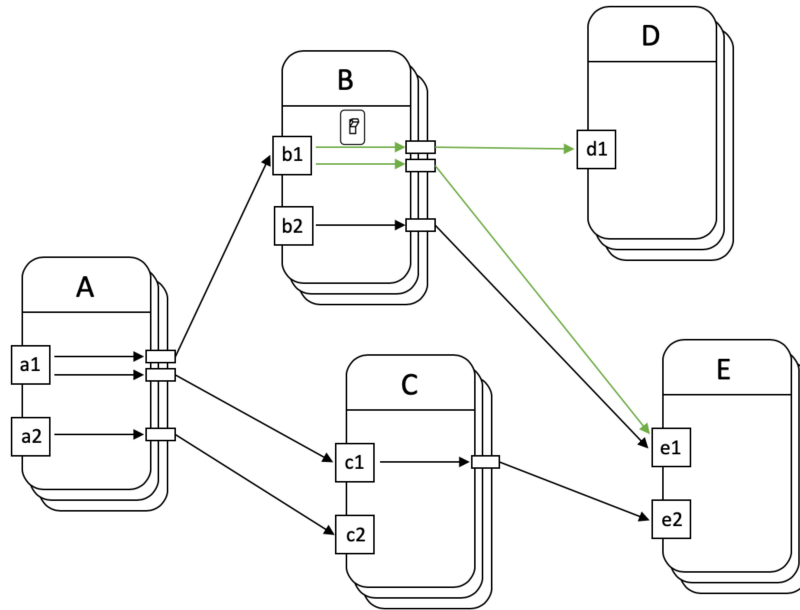


Figure 6.4: *arch03*

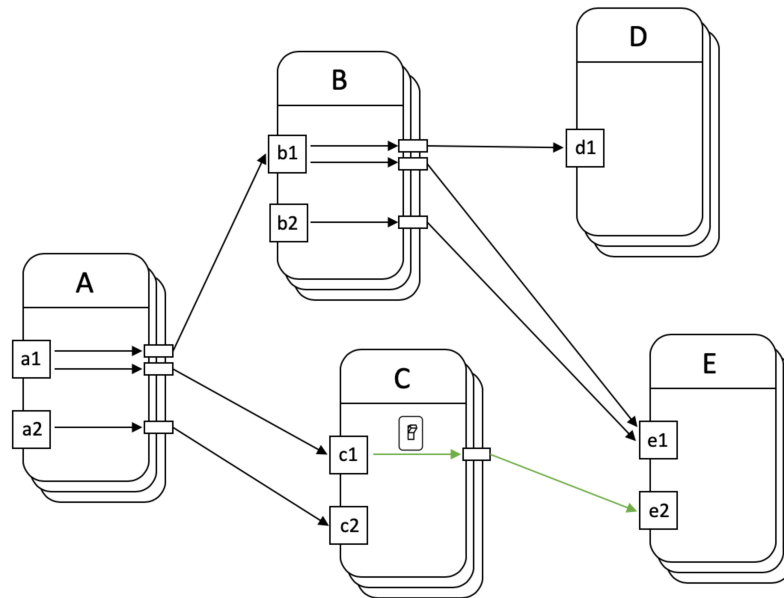


Figure 6.5: *arch04*

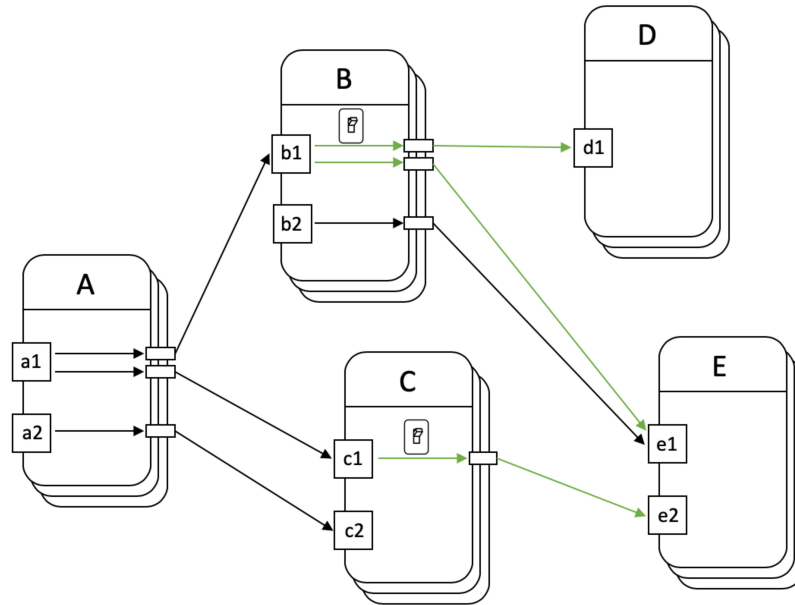


Figure 6.6: arch05

### 6.1.2 Chaos Experiment

A chaos experiment is defined through a steady-state and an action that introduces chaos into a system. The steady-state hypothesis is that the steady-state remains healthy, even after introducing chaos into the system.

The chaos experiments used in this evaluation will be as follows:

- *Steady-State:* The steady-state will be measured through the results of requests sent to the integration points *a1* and *a2*. The system is healthy if *a1* and *a2* both return a 200 HTTP status code and reply within 0.06s.
- *Chaos Action:* The chaos introducing is performed through the fault injection that the algorithms generate.

### 6.1.3 Metrics

During evaluation, there are two items of interest about each of the algorithms:

- (1) How efficient is it in regards to selecting fault injections that reveal faults?
- (2) How good is the average fault detection rate?

Section 2.5.2 presented the *APFD* and *NAPFD* metrics, used for the evaluation of test case prioritization. However, both of those metrics are oriented towards executing either, in the case of *APFD*, all test cases of a test suite, or, in the case of *NAPFD*, a subset of the test cases in a test suite. However, as we are only selecting one fault injection, or test case, at a time, neither of these seemed to be working for this evaluation.

Instead, for (1), the following efficiency metric will be used:

- *Efficiency Rate (ER)*: The efficiency rate  $ER$  is defined in equation 6.1, with  $n$  being the total amount of fault injections required to reach a threshold of faults revealed  $p$ . A lower value, closer to 1, is better.

$$(6.1) \quad ER = \frac{n}{p}$$

The idea behind this metric is that more efficient algorithms will have a lower value for  $ER$ , as they reveal faults at a faster rate. During the evaluation, different values for  $p$  will be compared. The metric is average experiments required to reveal a fault.

For (2), the following two metrics will be used:

- *Fault Detetction Rate (FDR)*: The  $FDR$  metric runs  $n$  fault injections, and counts the faults revealed when querying both integration points  $a1$  and  $a2$ . The value lies between 0 and 1, where a higher value is better. Here,  $r_i$  is the result of the  $i$ th fault injection.

$$(6.2) \quad FDR = \frac{\sum_{i=1}^n r_i}{n}$$

$$(6.3) \quad r_i = \begin{cases} 1, & \text{if } a1 \text{ or } a2 \text{ return an error or timeout} \\ 0, & \text{else} \end{cases}$$

- *Weighted Fault Detetction Rate (WFDR)*: The  $WFDR$  metric builds on  $FDR$ . However, it also considers the impact on the system and user in general. When two integration points signal an error, the impact will be significantly higher, than when only one integration point signals an error. Building on this assumption, the  $WFDR$  metric differentiates between both integration points signaling an error, only one integration point signaling an error, and no integration point signaling an error for the  $i$ th result of a fault injection,  $r_i$ . Again,  $n$  fault injections are run. The value lies between 0 and 2, where a higher value is better.

$$(6.4) \quad WFDR = \frac{\sum_{i=1}^n r_i}{n}$$

$$(6.5) \ r_i = \begin{cases} 2, & \text{if } a1 \text{ and } a2 \text{ return an error or timeout} \\ 1, & \text{if either } a1 \text{ or } a2 \text{ return an error or timeout} \\ 0, & \text{else} \end{cases}$$

Additionally, both the average time for the operations *get\_experiment* and *result* will be tracked.

Finally, the *counters* of fault injections chosen will be tracked for each of the algorithms.

#### 6.1.4 Mocking Data

As mentioned in Section 6.1.1, example data for fault injections for the architectures used in the evaluation is available. This data will be used to mock fault injections into the real system, foregoing the effort and time of running injections into a live system.

Date	Data points
2018-09-16	706
2018-09-19	25484
2018-09-28	25525
2018-10-09	25516
2018-10-21	25525
2018-11-04	25530
2018-11-18	25524
2018-12-02	25506

**Table 6.1:** Example setup runs <sup>4</sup>

The example setup was run multiple times, with the amount of data varying. Kolmogorov-Smirnov 2.8 were ran to verify that there are no significant differences between the data sets, with the intent of merging them into one large data set. However, due to the vast amount of data, the Kolmogorov-Smirnov tests did not return any significant results (with p-values being way under a significance value of 0.05 for almost identical kernel density estimate plots). Instead, the run with the largest data set was selected for evaluation purposes. As listed in Table 6.1, that data set is from the run on 2018-11-04. The Table 6.1 consists of the columns *Date* and *Data points*. The prior column is the

date of the example setup run, while the latter is the amount of data points for the fault injection *b1-abort* for no active circuit breaker pattern.

For each fault injection the integration points *a1* and *a2* were queried. The data for each fault injection is in the following form:

```
2018-11-04T10:35:57.481103, 500, Internal Server Error, http://chaos-kube:31380/a1,
0.024900000,
{"timestamp":"2018-11-04T10:35:57.476+0000",status:500,"error":"Internal
Server Error","message":"502 Bad Gateway","path":"/a1"}"
2018-11-04T10:35:58.472416, 200, OK, http://chaos-kube:31380/a2, 0.019084000,
Operation a/a2 executed successfully.
```

**Listing 6.1:** Data points for *b1-abort* fault injection

Each data point, with examples showcased in listing 6.1, is made up of the following parts:

- *timestamp*: The time of capturing the data point.
- *HTTP status code*: The status code received as response for the request sent to the respective integration point, either 200, OK, or 500, fault
- *HTTP status code message*: The response message, describing the HTTP status code
- *request target*: The target for the request
- *response time*: The time it took to receive a response from the system
- *system response*: The system's response, either indicating a successful execution, or describing the server error. When an error occurs, it contains the internal error message.

This data can be cleaned, separated, and processed.

For cleaning, first, the *timestamp* column can be removed, as chronological order is not of importance. Second, the *system response* can be removed, as it doesn't give any additional information. Then, the *HTTP status code message* can be removed, as it only describes the *HTTP status code*.

For separating, the data can be separated for each of the integration points *a1* and *a2*, based on the *request target*.

For processing, the data can then be analyzed on whether or not a fault has occurred. Faults occur either if a 500 error code is returned, or if a timeout occurs. As mentioned in Section 6.1.1,  $0.06s$  is assumed to be the limit for regular requests. This threshold is used for the timeouts, with any request taking over  $0.06s$  being classified as revealing a fault.

As a result, the data consists of only one binary value, whether a fault has been revealed or not.

Of this data, a sample of size 5000 is taken for each fault injection. When retrieving mocked results, 1 data point from those 5000 is then taken.

### 6.1.5 Parameters

After individually testing different parameters for the algorithms, the following parameters were used for this evaluation. The naming follows the reference names introduced in Table 6.3:

<i>bayes – low</i>	<ul style="list-style-type: none"> <li>• <i>training_size</i> = 200</li> </ul>
<i>bayes – high</i>	<ul style="list-style-type: none"> <li>• <i>training_size</i> = 200</li> </ul>
<i>bandit – eps</i>	<ul style="list-style-type: none"> <li>• <math>\epsilon = 0.5</math></li> <li>• <math>\gamma = 0.99</math></li> <li>• <i>min_epsilon</i> = 0.1</li> </ul>
<i>bandit – opt</i>	<ul style="list-style-type: none"> <li>• <i>initial_q</i> = 5</li> </ul>
<i>qlearning</i>	<ul style="list-style-type: none"> <li>• <math>\epsilon = 0.2</math></li> <li>• <math>\gamma = 0.99</math></li> <li>• <i>learning_rate</i> = 0.1</li> <li>• <i>initial_q</i> = 5</li> </ul>
<i>ann</i>	<ul style="list-style-type: none"> <li>• <i>hidden_size</i> = 12</li> </ul>
<i>tableau</i>	<ul style="list-style-type: none"> <li>• <math>\epsilon = 0.4</math></li> <li>• <math>\gamma = 0.99</math></li> <li>• <i>initial_q</i> = 5</li> <li>• <i>min_epsilon</i> = 0.1</li> <li>• <i>action_size</i> = 5</li> </ul>
<i>mlfq</i>	<ul style="list-style-type: none"> <li>• <i>queue_count</i> = 5</li> <li>• <i>iterations_reset</i> = 200</li> </ul>

**Table 6.2:** Parameters used for algorithms

## 6.2 Description of Results

The architectures *arch01* to *arch05* can be grouped into three groups, based on how many operations are protected by the circuit breaker pattern:



- High resilience: *arch01* 6.2
- Medium resilience: *arch05* 6.6
- Low resilience: *arch02* 6.3, *arch03* 6.4, *arch04* 6.5

To limit the amount of data presented here, only one candidate for each group will be presented. These candidates are: *arch01* (high), *arch05* (medium), and *arch02* (low).

For each architecture, first the ER, FDR, and WFDR metrics, as well the as the fault injection counters will be presented. Then, the time metrics for the *get\_experiment* and *enter\_result* operations, with data used from the *arch02* architecture experiment, will be presented.

To increase confidence in the results, the experiments for each metric and architecture were performed a total of 20 times. The mean values of the results generated by the individual runs was then taken.

For the ER metric, each algorithm generated fault injections, until a total of 150 faults have been revealed. For the FDR and WFDR metrics, each algorithm generated 400 fault injections. These values were chosen after experimenting with higher values, but not finding higher values to give additional information.

For reinforcement learning algorithms, both reward functions introduced in Section 4.4.1, were tested. However, the results show no significant difference, so only the results using reward function (1) will be presented.

The algorithms will be referenced as follows:

Algorithm	Reference Name
RL: $\epsilon$ -greedy 4.4.2	<i>bandit-eps</i>
RL: Optimistic Initial Value 4.4.3	<i>bandit-opt</i>
Bayesian Network (High Level) 4.3.2	<i>bayes-high</i>
Bayesian Network (Low Level) 4.3.1	<i>bayes-low</i>
Multilevel Feedback Queue 4.5	<i>mlfq</i>
RL: Tableau 4.4.6	<i>tableau</i>
RL: Artificial Neural Network 4.4.5	<i>ann</i>
RL: Q-Learning 4.4.4	<i>qlearning</i>

**Table 6.3:** Reference names for algorithms

6.2.1 *arch01*

Architecture *arch01* is visualized in Figure 6.2. *arch01* is considered to be a highly resilient architecture, based on the circuit breaker pattern implementations.

The ER (Figure 6.7) and FDR (figures 6.8, 6.9) metrics show *qlearning* to be performing the worst, delivering worse results than *random*, with *tableau* coming in second worst, however still ahead of *random*. Both *qlearning* and *tableau* show little to no learning.

All other algorithms show clear learning, with the *bandit-opt* overall performing best, while *ann* performs best initially.

The WFDR metric in Figure 6.10 shows identical results to the FDR metric, with *ann* having a significantly larger variance over the runs, while *bandit-opt* behaves almost deterministic.

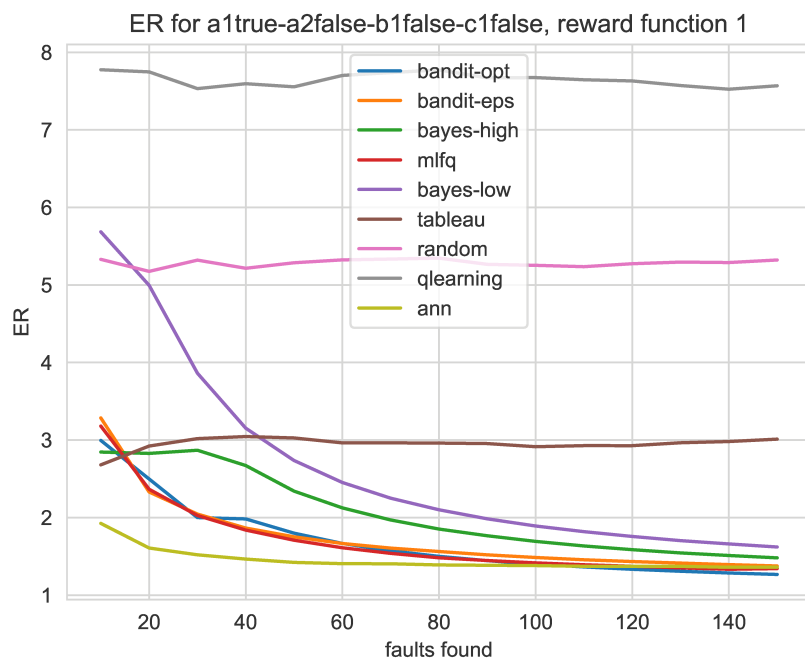


Figure 6.7: ER metric for *arch01*, line graph

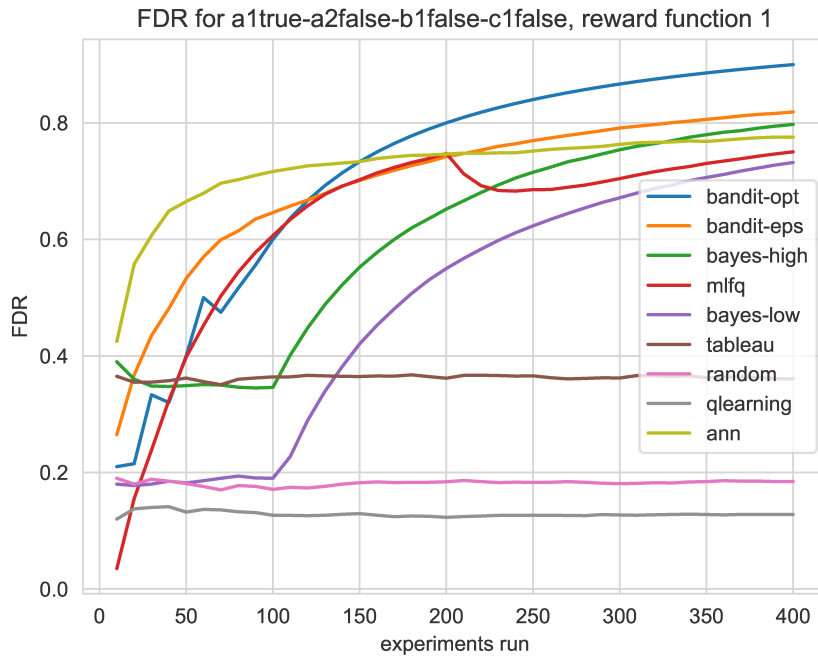


Figure 6.8: FDR metric for *arch01*, line graph

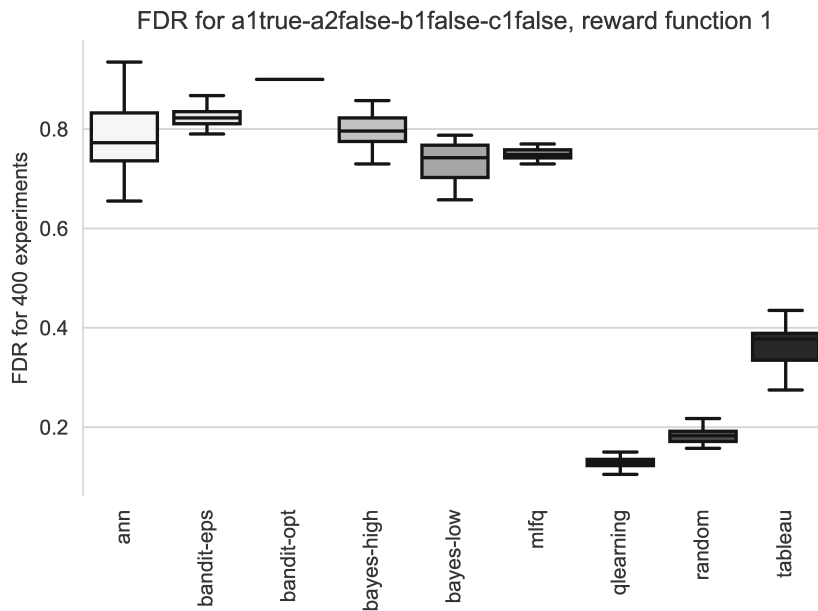
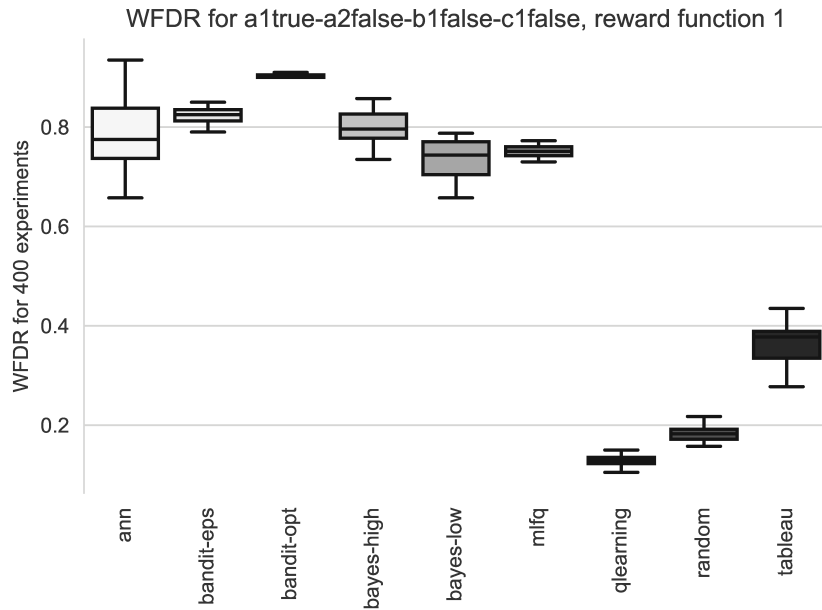


Figure 6.9: FDR metric for *arch01*, box graph

Figure 6.10: WFDR metric for *arch01*, box graph

fault-injection	ann	bandit-eps	bandit-opt	bayes-high	bayes-low	mlfq	qlearning	random	tableau
b1-abort	17.45	6.85	3	8.5	7.5	9.5	65.75	30.2	32.65
b1-delay	11.75	6.1	3.15	12.9	12.6	9.45	65.9	33.2	32.7
c1-abort	18.3	6.3	3	7.75	7.6	7.25	43.95	32.8	30.75
c1-delay	15.25	6.1	3.2	11.75	11.2	7.35	42.25	31.95	32.15
c2-abort	145.7	302.75	178.8	152	145.2	149.55	22.9	31.1	64.6
c2-delay	152.65	16.95	178.4	158.1	145.35	148.55	22.6	31.5	67.6
d1-abort	3.89	12.35	3	3.85	7.5	12.85	21.4	33.75	22.25
d1-delay	4.8	6.8	3.2	6.85	12.8	10.85	19.7	31.6	19.15
e1-abort	5.05	5.3	3	5.85	6.7	8.5	20.55	32.8	23.25
e1-delay	5.05	6.85	3.15	8.2	11.4	8.55	20.8	35.15	20
e2-abort	5.47	5.75	3	4.65	7.15	7.7	22.3	32.55	21
e2-delay	3.67	5.9	3.1	7.6	13	7.9	19.9	31.4	21.9

Table 6.4: Counter results for *arch01*

### 6.2.2 *arch02*

Architecture *arch02* is visualized in Figure 6.3. *arch02* is considered to be an architecture with low resilience, based on the circuit breaker implementations.

The ER metric (Figure 6.11) and FDR metric (figures 6.12, 6.13) show that *qlearning* shows significant improvements over *arch01*, while *tableau* is at the bottom, with a high variance. *bandit-opt* shows the overall best results, while *ann* and *bandit-eps* initially perform slightly better. Contrary to *arch01*, *ann* here has a much lower variance.

Again, the WFDR metric in Figure 6.14 shows no significant difference to the FDR metric.

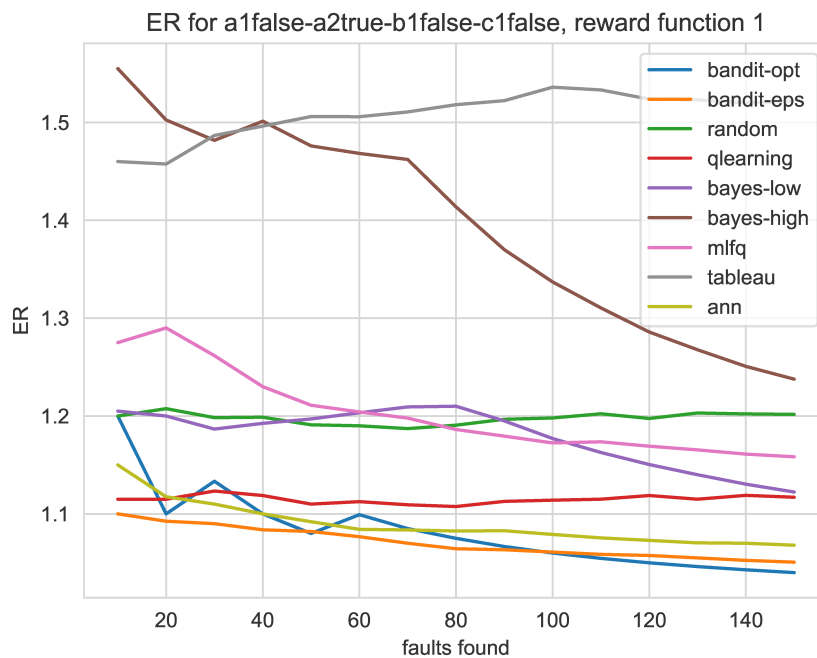


Figure 6.11: ER metric for *arch02*, line graph

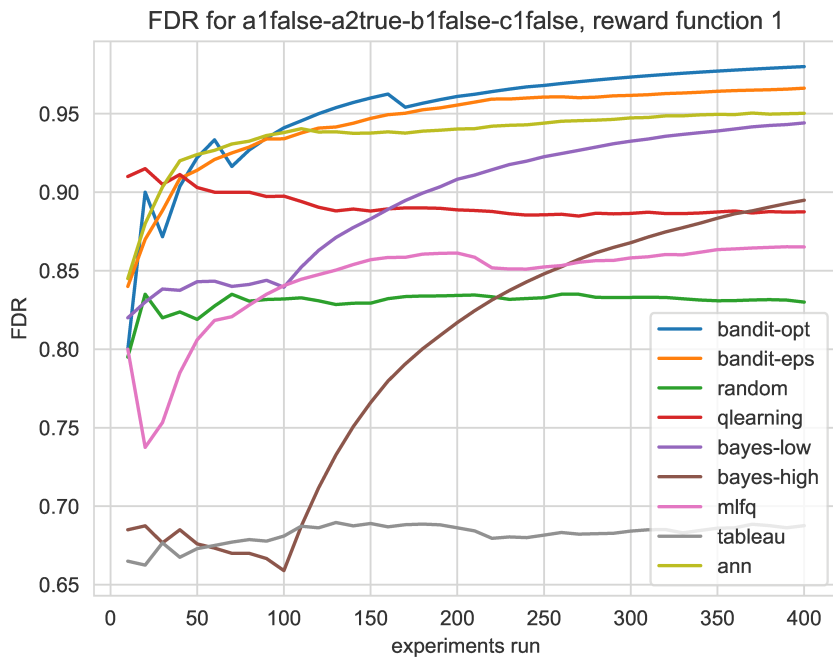


Figure 6.12: FDR metric for *arch02*, line graph

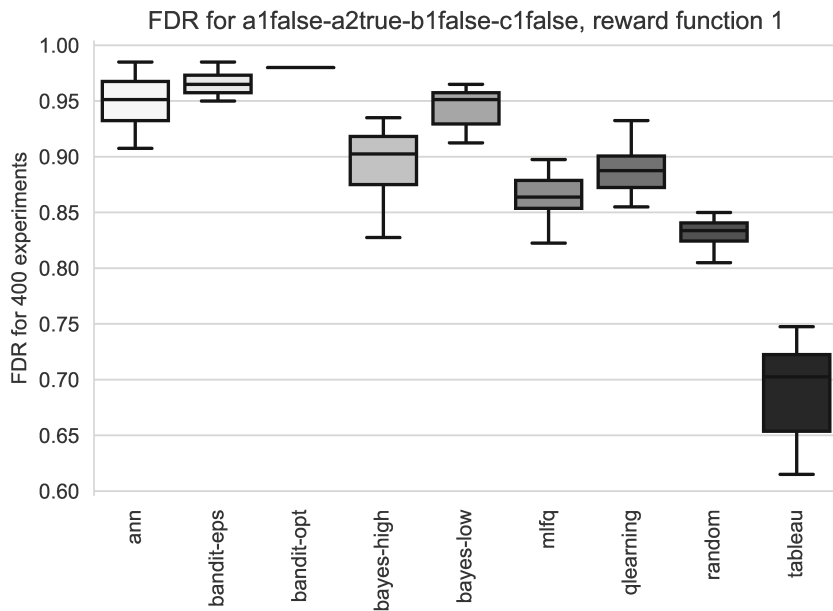
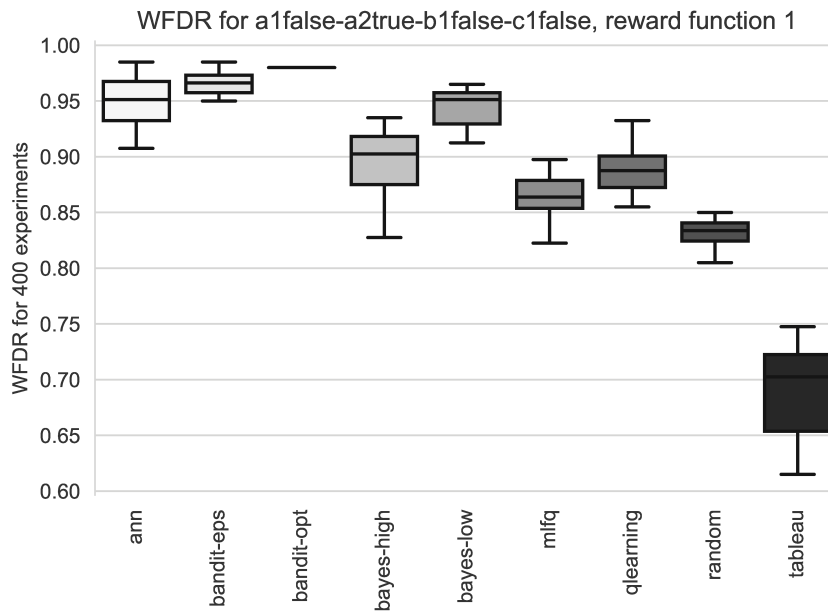


Figure 6.13: FDR metric for *arch02*, box graph

Figure 6.14: WFDR metric for *arch02*, box graph

fault-injection	ann	bandit-eps	bandit-opt	bayes-high	bayes-low	mlfq	qlearning	random	tableau
b1-abort	35.4	6.5	38	45.6	34.05	34.1	62	34.4	30.3
b1-delay	39.75	6.75	38	41.35	36.8	33.8	68.2	31.95	35.7
c1-abort	34.95	5.9	38	42.7	37.1	32.9	44.25	33	33.2
c1-delay	36.55	6.6	38	43.75	34.8	32.7	43.6	31.75	31.7
c2-abort	4.75	5.15	3.05	15.9	7.45	27.05	21.8	30	65.05
c2-delay	6.25	5.55	3.4	23.95	13.95	25.25	20.3	34.3	62.85
d1-abort	37.45	316.25	38.95	27.95	36.1	34.55	19.55	30.7	21.4
d1-delay	37.95	11	38.6	26.2	38.3	34.3	23.25	30.6	21.2
e1-abort	39.35	5.85	38	29.6	35.95	33.6	20.85	32.25	21.95
e1-delay	38.65	5.9	38	30	37.85	33.4	21.9	32.75	18.8
e2-abort	39.85	6.9	38	31.9	39	33.25	21.55	34.15	24.15
e2-delay	37.1	6.25	38	29.1	36.65	33.1	20.75	32.15	21.7

Table 6.5: Counter results for *arch02*

6.2.3 *arch05*

Architecture *arch05* is visualized in Figure 6.6. *arch05* is considered to have medium resilience, based on the circuit breaker pattern implementations.

The ER (Figure 6.15) and FDR (figures 6.16, 6.17), again, show *qlearning* to be better than for *arch01*. Here, *ann* and *bandit-opt* perform best, with *ann* initially performing better. Similar to the previous architectures, *bandit-opt* shows a zick-zack pattern, occasionally getting worse, before rapidly improving again. *mlfq* also shows a clear drop.

*tableau* and *random* perform almost identical for *arch05*.

Again, the WFDR metric in Figure 6.18 is almost identical to the FDR metric.

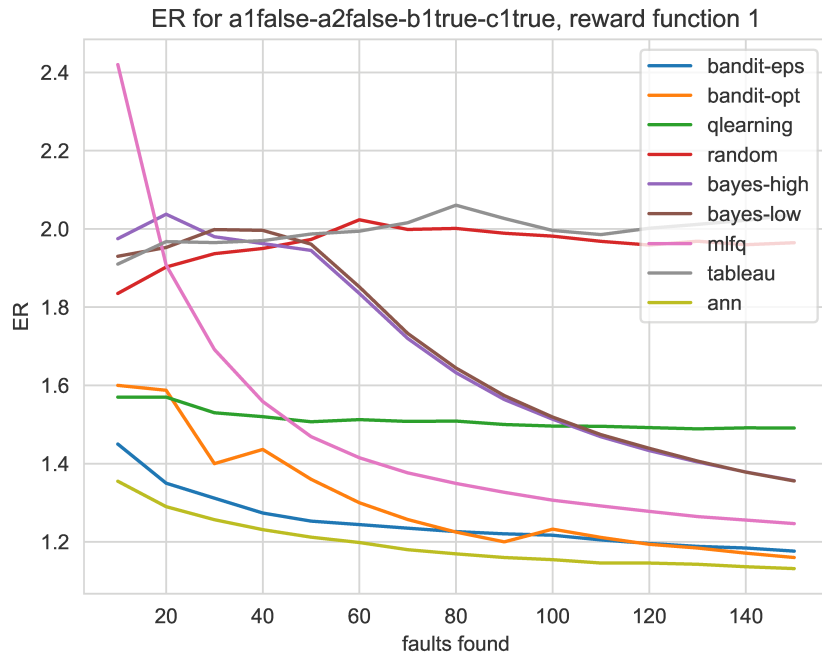


Figure 6.15: ER metric for *arch05*, line graph



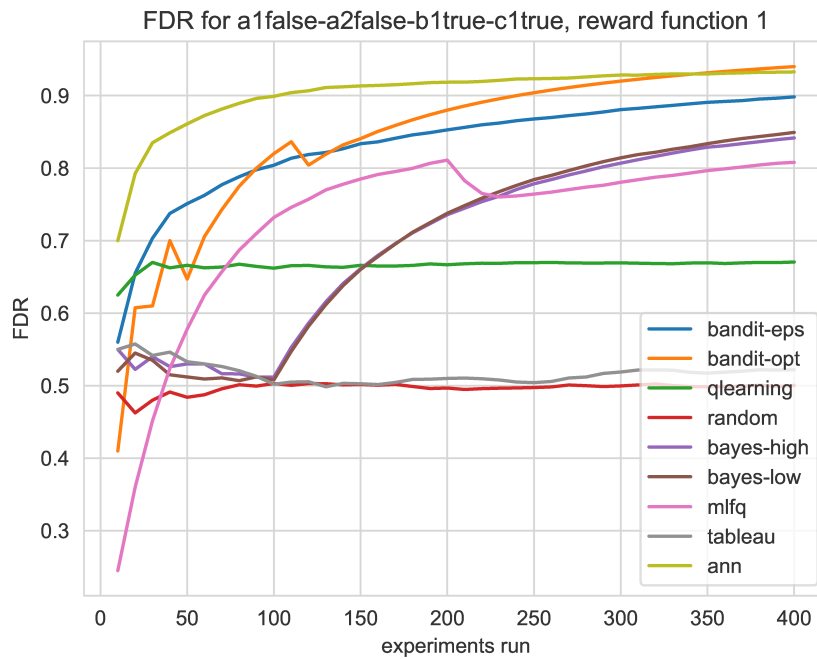


Figure 6.16: FDR metric for *arch05*, line graph

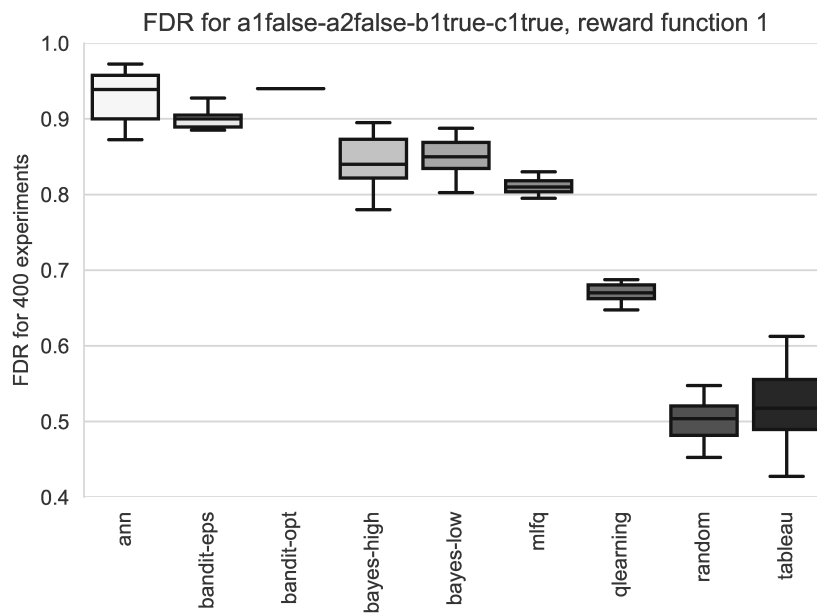
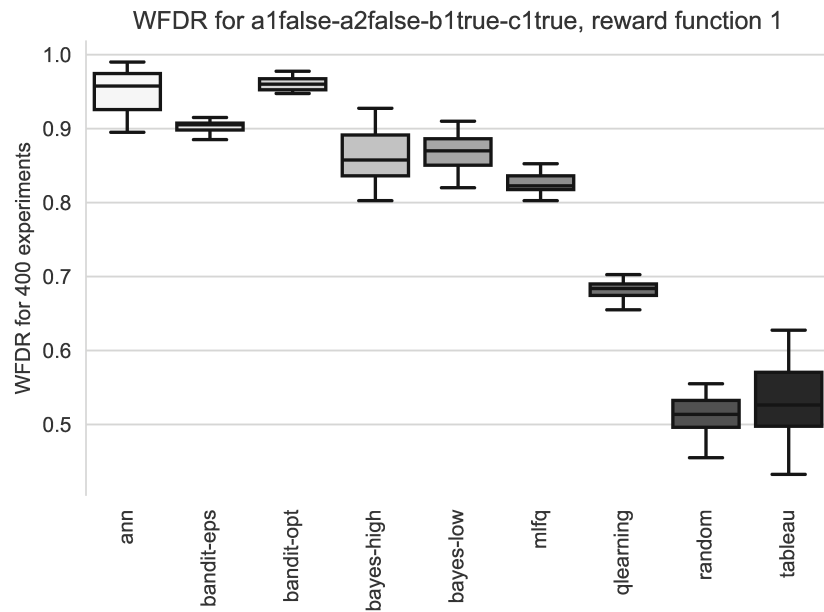


Figure 6.17: FDR metric for *arch05*, box graph

Figure 6.18: WFDR metric for *arch05*, box graph

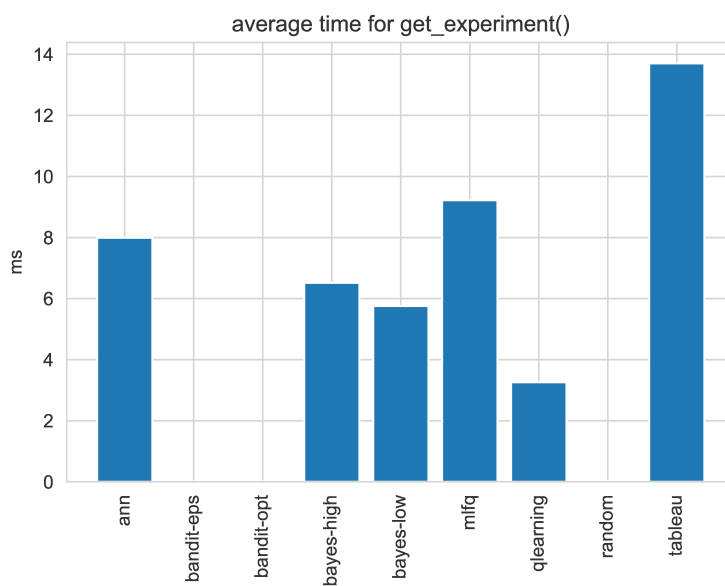
fault-injection	ann	bandit-eps	bandit-opt	bayes-high	bayes-low	mlfq	qlearning	random	tableau
b1-abort	60.85	310.26	62	52.65	54.6	53.55	66.45	33.35	32.05
b1-delay	59.8	31.15	61.95	55.3	56.1	53.2	64.85	32.65	34.8
c1-abort	59.3	6.2	61	55.95	56.3	52	42.35	34.1	32.45
c1-delay	62.85	7.25	61	52.5	54.3	51.9	44.45	33.1	32.2
c2-abort	61.3	6.65	61.8	51.25	54.45	52.7	21.1	32.9	30.45
c2-delay	62.55	6.95	61.35	53.25	56.85	52.35	21	33.55	32.8
d1-abort	2.84	7.7	3	8.8	7.3	14.85	21.05	31.7	36.2
d1-delay	4.75	6.25	3.45	13.9	11.75	12.8	20.85	31.7	29.2
e1-abort	2.65	5.9	3	7.65	7	11.65	22.45	32.35	34.95
e1-delay	3.95	5	3.2	14.15	12.25	11.65	20.3	31.4	30
e2-abort	2.68	4.8	3	7.65	7.8	10.6	21.5	30.7	32.4
e2-delay	4.85	5.45	3.25	14.95	9.3	10.75	21.65	30.5	30.5

Table 6.6: Counter results for *arch05*

### 6.2.4 Times

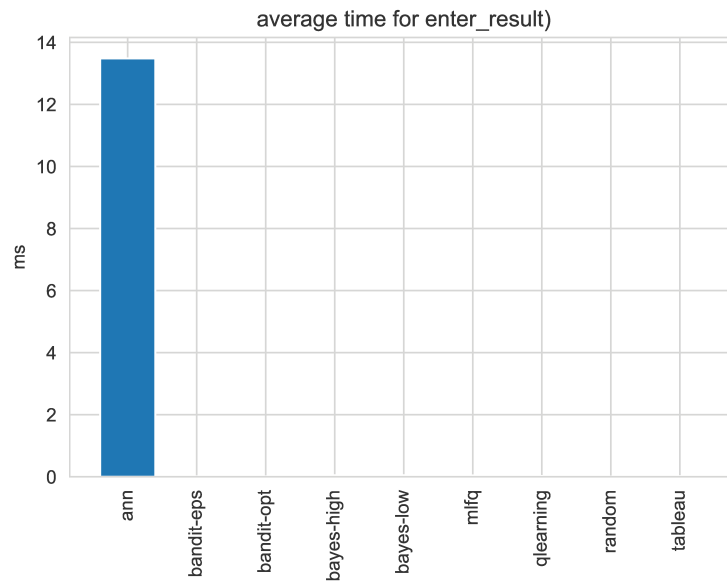
For the times metrics, 400 experiments were generated by each algorithm. The time for both the *get\_experiment* and *enter\_result* operations were tracked for architecture *arch02*. The average time for each will be presented.

Figure 6.19 shows that *bandit-eps* and *bandit-opt* take almost no time to generate experiments. *qlearning* requires a bit longer, while *mlfq* and the Bayesian algorithms *bayes-low* and *bayes-high* require more time. The *ann* algorithm is only overtaken by the *tableau* algorithm in regards to the time it takes to return a fault injection.



**Figure 6.19:** Average times for *get\_experiment*

Figure 6.20 shows that almost all algorithms require close to no time to process the result. Only *ann* requires significantly more time for processing the result.



**Figure 6.20:** Average times for *enter\_result*

### 6.2.5 Summary

The following observations can be made from the results:

- *qlearning* and *tableau* perform consistent, showing no clear improvement from learning.
- *qlearning* performs very different, based on the architecture used.
- *tableau* consistently performs bad.
- *ann* consistently performs good, initially being, or close to being, the best. However, it is higher variance in results, than other algorithms (except *tableau*).
- *mlfq* consistently performs average, better than *tableau*, but worse than the *bayes*, *ann*, and *bandit* algorithms.
- The *bayes* algorithms show clear learning.
- *bandit-opt* consistently performs best, when looking at the overall results. It shows clear signs of improvement, with rapid fall-offs inbetween.
- *bandit-eps* consistently performs average.
- *bandit-eps* focusses on single fault injections, as seen in the selection counters.

- *ann*, *bandit-opt*, *bayes-high*, *bayes-low*, and *mlfq* prioritize operations, but evenly distribute between fault injection types, as seen in the selection counters.
- *ann* is the only algorithm that requires time for processing the fault injection result.

## 6.3 Discussion of Results

This section will discuss the results of the individual algorithms. Additionally, the presented results will be reviewed in regards to the research questions.

The results for the *FDR* and *WFDR* metrics were identical for this evaluation. This is due to the fact that no operation can be reached by both the integration points *a1* and *a2*. Instead, each operation can be reached either *a1*, or *a2*, exclusively. However, this metric may provide useful for other architectures, since it does provide insight into the health of the system as a whole.

### 6.3.1 *ann*

*ann* is one of the algorithms that considers the architectural model of the architecture. It consistently performs good, showing high efficiency in the early stages, while consistently delivering good results for all architectures reviewed. However, *ann* was the only algorithm that required significant time for processing the fault injection result.

Based on the results, *ann* is one of the algorithms with the highest potential for future work.

### 6.3.2 *bandit-eps*

*bandit-eps* does not consider architectural information. However, it still consistently performs good, close to *bandit-opt* and *ann*. A downside of *bandit-eps* is the behavior of focussing on a single fault injection, as seen in tables 6.4, 6.5, and 6.6.

### 6.3.3 bandit-opt

*bandit-opt* does not consider architectural information. Furthermore, *bandit-opt* does not use random exploration. Instead, as the box plots in figures 6.9, 6.13, and 6.17 show, it behaves deterministic. This also explains the very good results into the later stages, as the algorithm does not waste any time on exploration. Instead, it exploits the knowledge of the best fault injections. The exploration occurs on the occasional drop-offs.

While delivering good results for the metrics, the inconsistency make *bandit-opt* not the ideal choice.

### 6.3.4 bayes-high

Except for *arch01*, *bayes-high* performs slightly worse than *bayes-low*. The difference in *bayes-low* can be explained by the mapping of feature sets to fault injections. The only vulnerable operation in *arch01*, *c2*, has a lower selection probability, when randomly selecting from feature sets, instead of fault injections directly. That makes the initial random selection phase during the training perform better for *bayes-high*, compared to the training of *bayes-low*. However, after, they behave similar, even for *arch01*.

Overall, *bayes-high* makes more sense for the future application, assuming that more features become available. However, currently, *bayes-low* outperforms *bayes-high*.

### 6.3.5 bayes-low

Without considering any architectural information, and only relying on fault injection results, *bayes-low* performs similar to the *bandit* algorithms. However, due to the initial training phase, the results are more delayed, making it less ideal, in regards to efficiency. Additionally, the time to get experiments was also higher than for *bandit* algorithms.

### 6.3.6 mlfq

*mlfq* comes in right in the middle of all algorithms. Due to the reset of the queues, the efficiency will occasionally reset. Since *mlfq* does not consider architectural information, and performs worse, than *bandit* algorithms and *bayes-low*, it should not be considered any further.

### 6.3.7 *qlearning*

The initial idea of *qlearning* was to use the dependencies between operations, finding the most vulnerable path that requests could potentially take. However, it shows very little learning, making it less efficient than other algorithms.

In *arch01*, *qlearning* performed significantly worse than for *arch02* and *arch05*. This is due to the only vulnerable operation in *arch01*, *c2*, having no outgoing dependencies. The only possible path to *c2* is then caused by the random selection of the starting state of a new episode. This could potentially be optimized, by using the value of states to alter this random selection.

### 6.3.8 *tableau*

The *tableau* algorithms attempted to prioritize fault injections, as adopted from the test case prioritization in [SGMM17b]. However, the results were overall bad, and *tableau* should not be considered further, unless more than one fault injections could be injected at a time. If a sequence of fault injections would be used, *tableau* may prove better, as more fault injections from the highest priority would be used and rewarded.

### 6.3.9 Research Questions

This section will review the research questions, and give answers based on the previously presented experiment results.

RQ1: Which algorithm has the highest fault detection rate?

*bandit-opt* and *ann* consistently deliver the best results. However, *bandit-opt* deliver very deterministic results, while *ann* has a wider variance. Generally, a wider variance may be preferred for live systems, as it indicates a higher rate of exploration.

RQ2: Is there a noticeable difference between algorithms that use architectural information, and those that do not?

The *ann* algorithm initially performs best for *arch01* and *arch05*. For *arch02*, it still initially performs good. However, *ann* also has the highest variance in overall performance. *bayes-low* and *bayes-high* perform similar, depending on the architecture. Generally

speaking, more architectural information may be required to see a clear improvement, over using only results.

RQ3: How efficient are the algorithms?

There is no significant difference between algorithms that use architectural information, and those that only rely on fault injections results. *ann* performs best for those that use architectural information, while *bandit-opt* and *bandit-eps* perform best for those that do not consider architectural information. The *bayes* algorithms initially perform less efficient, since a training phase is required.

## 6.4 Threats to Validity

This section will discuss possible threats to the validity of the previously presented results of the experimental evaluation.

### 6.4.1 Conclusion Validity

The threats to conclusion validity concern the relationship between treatment and outcome of an experiment [WRH+12].

Possible threats for this are the incorrect samples for the fault injection data. Since the Kolmogorov-Smirnov test did not return a *p-value* over the significance threshold 0.05 for all fault injection data sets of the different dates in the sample data, only data from one of those dates has been used for the experimental evaluation. Since the system did not change between fault injections, drawing a sample from all data points for one fault injection should not have threaten the validity of the outcome. Furthermore, during the experimental evaluation about 500 fault injections were performed. These 500 fault injections each took 1 sample from a total of 5000 data points each. This should allow for the probability of taking the same sample multiple times to be low.

Since the experiment was repeated 20 times, the results can be believed not to be at random. The line plots use the average of all experiment runs, while the box plots show the variance between the different runs.



### 6.4.2 Internal Validity

Threats to internal validity are considered as the influences that can affect the independent variable with respect to causality.

A potential threat is the selection of parameters for the algorithms. Especially the *training\_size* parameter for the Bayesian network algorithms can have a high impact on the results.

### 6.4.3 Construct Validity

The threats to construct validity are concerned with the relation between the experiment and the theory behind the experiment [WRH+12].

A possible threat is the *inadequate preoperational explication of constructs* [WRH+12], i.e. an in-sufficient clarifications on the research questions. Since RQ1 and RQ3 can be directly compared, using the respective metrics FDR and ER, this should not be a problem. For RQ2, multiple metrics have been considered. However, since multiple aspects have to be considered, the result to this research question may depend on the personal interpretation of *noticeable*.

Another threat may be the definition of when a fault is revealed. While the 500 error code is a clear indication of a fault, the 0.06s timeout threshold may threaten the results, especially if another system was used. The threshold was set, based on the response time of requests into the system in a healthy state.

### 6.4.4 External Validity

The threats to external validity are concerned with the ability of generalizing the results to industrial practice [WRH+12].

Since only one main architecture, with different circuit breaker implementations, was used for the experimental evaluation, the results may not be applicable to other architectures. For example, the *qlearning* algorithm may show better or worse results for larger architectures. The same applies to the other algorithms.



## Chapter 7

# Conclusion

---

This thesis explored different algorithms for fault injection selections, to be used in chaos experiments, as part of the decision engine in the ORCAS framework.

First, the foundations for the related topics were presented. For the algorithms themselves, Bayesian networks, reinforcement learning, artificial neural networks, and multi-level feedback queues were researched.

Then, related works to efficient fault injections and test case prioritization with reinforcement learning were presented.

Next, the algorithms were presented. These are:

- **Low Level Bayesian:** A low level Bayesian network based approach, that does not consider architectural information.
- **High Level Bayesian:** A high level Bayesian network approach, that does consider architectural information.
- **$\epsilon$ -greedy Reinforcement Learning:** A single state reinforcement learning approach, that does not consider architectural information.
- **Optimistic Initial Value Reinforcement Learning:** An adaptation of  $\epsilon$ -greedy, which omits random exploration. This algorithm does not consider architectural information.
- **Neural Network based Reinforcement Learning:** This algorithm combined a perceptron classifier with reinforcement learning principles. Architectural information of operations was used as features, that the perceptron classifier used to compute probabilities of revealing faults for each fault injection.
- **Tableau Reinforcement Learning:** Architectural information about operations was extracted into feature sets, which were prioritized into priority groups.

- **Multilevel Feedback Queues:** Adapted from process scheduling, fault injections were sorted into different priority queues, based on their previous results. This algorithm does not consider architectural information.

Chapter 5 then presented the context and possible integration into the ORCAS framework, and future automation of chaos experiments.

The final chapter evaluated the algorithms based on Efficiency Rate (ER), Fault Detection Rate (FDR), Weighted Fault Detection Rate (WFDR), and Time metrics.

While the ER, FDR, and WFDR metrics showed the neural network based reinforcement learning algorithm to have the overall best results, the  $\epsilon$ -greedy and optimum initial value algorithms also performed good. Contrary to the neural network based algorithm, the latter two do not require architectural information. Furthermore, the neural network based approach is the only approach to require time for processing the fault injection results. However, applied to the case of executing chaos experiments, that may take multiple seconds, this time is negligible.

Looking forward, reinforcement learning seems like the ideal approach for solving this problem. The neural network based algorithm introduced an approach to utilizing the available architectural information, while the  $\epsilon$ -greedy and optimistic initial value algorithms present simple approaches, that still generated good results.

However, this evaluation was performed on a relatively small architecture. To get more realistic results, experiments on larger architectures should be performed. This thesis also did not consider all options for possible algorithms. The hope is that the result of this work can be used as a starting point in the future.

## Future Work

There are three unresolved areas, that could be explored in future work:

- **Automation and integration:** Chapter 5 proposed possible approaches to automating chaos experiments, and integrating the algorithm results with the MiSim simulator. However, first a steady-state hypothesis has to be generated, ideally based on the architecture. For an integration with the simulator, the simulator should support fault injections into single operations, instead of services.
- **Architectural model:** The architectural model is currently very minimal, and, outside of the circuit breaker pattern, does not include any additional resilience information. This model could be improved, by adding support for more pattern detection.

- 
- Experimental evaluation: The experimental evaluation of this thesis was performed on a relatively small reference architecture. Performing the experiments on larger architectures, possibly even live environments, might deliver more accurate results.



## Appendix

# Bibliography

---

- [AA15] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau. “Operating systems: Three easy pieces.” In: vol. 1. Arpaci-Dusseau Books, 2015. Chap. The Multi-Level Feedback Queue (cit. on p. 29).
- [AAS+16] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, L. Hochstein. “Automating failure testing research at internet scale.” In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM. 2016, pp. 17–28 (cit. on pp. 31, 32).
- [ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing.” In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (Jan. 2004), pp. 11–33 (cit. on p. 10).
- [ARH15] P. Alvaro, J. Rosen, J. M. Hellerstein. “Lineage-driven fault injection.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 331–346 (cit. on pp. 31, 32).
- [BBHR16] A. Blohowiak, A. Basiri, L. Hochstein, C. Rosenthal. “A platform for automating chaos experiments.” In: *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2016, pp. 5–8 (cit. on p. 76).
- [Bec] S. Beck. *github.com/orcas-elite/arch-extract*. Accessed on: 2019-04-04. URL: [https://github.com/orcas-elite/arch-extract/blob/master/arch-extract/architecture\\_model.json](https://github.com/orcas-elite/arch-extract/blob/master/arch-extract/architecture_model.json) (cit. on p. 17).
- [Bec18] S. Beck. “Simulation Based Evaluation Of Resilience Antipatterns In Microservice Architectures.” Bachelor’s Thesis. University of Stuttgart, Aug. 2018 (cit. on pp. 5, 16, 17, 19, 82).
- [Ben08] I. Ben-Gal. “Bayesian Networks.” In: *Encyclopedia of Statistics in Quality and Reliability*. American Cancer Society, 2008. ISBN: 9780470061572 (cit. on pp. 26, 27).

- [BGZ] S. Beck, J. Günthör, C. Zorn. *MiSim Microservice Resilience Simulator*. URL: <https://github.com/orcas-elite/resilience-simulator> (cit. on p. 19).
- [BGZ17] S. Beck, J. Günthör, C. Zorn. “Simulation-based Resilience Prediction of Microservice Architectures.” Fachstudie. University of Stuttgart, Dec. 2017 (cit. on pp. 19, 78).
- [Blo11] N. T. Blog. *The Netflix Simian Army*. <https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>. Accessed: 2019-01-13. 2011 (cit. on pp. 12, 13).
- [Blo16] N. T. Blog. *Netflix Chaos Monkeys Upgraded*. <https://medium.com/netflix-techblog/netflix-chaos-monkey-upgraded-1d679429be5d>. Accessed: 2019-01-13. 2016 (cit. on p. 13).
- [CMD62] F. J. Corbató, M. Merwin-Daggett, R. C. Daley. “An experimental time-sharing system.” In: *Proceedings of the AFIPS Fall Joint Computer Conference*. 1962, pp. 335–344 (cit. on p. 29).
- [DGL+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, L. Safina. *Microservices: yesterday, today, and tomorrow*. Apr. 2017 (cit. on p. 7).
- [DH17] T. F. Düllmann, A. van Hoorn. “Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches.” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE ’17 Companion. New York, NY, USA: ACM, 2017, pp. 171–172 (cit. on p. 16).
- [EMR02] S. Elbaum, A. G. Malishevsky, G. Rothermel. “Test case prioritization: A family of empirical studies.” In: *IEEE transactions on software engineering* 28.2 (2002), pp. 159–182 (cit. on pp. 13, 14).
- [Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995 (cit. on pp. 38, 41).
- [Glo] Glosser.ca. *Colored neural network*. Accessed on: 2019-04-17. URL: [https://en.wikipedia.org/wiki/File:Colored\\_neural\\_network.svg](https://en.wikipedia.org/wiki/File:Colored_neural_network.svg) (cit. on p. 28).
- [HADP18] A. van Hoorn, A. Aleti, T. F. Düllmann, T. Pitakrat. “ORCAS: Efficient Resilience Benchmarking of Microservice Architectures.” In: *29th IEEE International Symposium on Software Reliability Engineering*. 2018 (cit. on pp. 2, 15, 16, 18, 35, 76).
- [HGS93] M. J. Harrold, R. Gupta, M. L. Soffa. “A methodology for controlling the size of a test suite.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pp. 270–285 (cit. on p. 13).



- [htt16] <https://github.com/Netflix/NetflixFramework>. *Netflix/SimianArmy*. [https://github.com/Netflix/SimianArmy](https://github.com/Netflix/NetflixFramework). Accessed: 2019-01-13. 2016 (cit. on p. 13).
- [jae] [jaegertracing.io](http://jaegertracing.io). *Jaeger*. Accessed on: 2019-04-04. URL: <https://www.jaegertracing.io> (cit. on p. 16).
- [JMM96] A. K. Jain, J. Mao, K. Mohiuddin. "Artificial neural networks: A tutorial." In: *Computer* 3 (1996), pp. 31–44 (cit. on pp. 27, 28).
- [Kir96] T. Kirkman. *Kolmogorov-Smirnov Test*. Accessed on: 29-05-2019. 1996. URL: <http://www.physics.csbsju.edu/stats/KS-test.html> (cit. on p. 30).
- [Kra11] S. Kramer. *The Biggest Thing Amazon Got Right: The Platform*. Oct. 2011 (cit. on p. 7).
- [Kun00] F. Kunz. "An introduction to temporal difference learning." In: *Seminar on Autonomous Learning Systems*. 2000 (cit. on pp. 21, 22, 25).
- [Laf17] M. Lafeldt. *A Primer on Automating Chaos*. Accessed on: 29-05-2019. Sept. 2017. URL: <https://www.gremlin.com/community/tutorials/a-primer-on-automating-chaos/> (cit. on p. 76).
- [LF] J. Lewis, M. Fowler. *Microservices—a definition of this new architectural term*. <https://martinfowler.com/articles/microservices.html>. Accessed: 2019-01-13 (cit. on p. 8).
- [Mas51] F. J. Massey Jr. "The Kolmogorov-Smirnov test for goodness of fit." In: *Journal of the American statistical Association* 46.253 (1951), pp. 68–78 (cit. on p. 30).
- [Mau15] T. Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*. Feb. 2015. URL: <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/> (cit. on p. 7).
- [NCM16] R. Natella, D. Cotroneo, H. S. Madeira. "Assessing Dependability with Software Fault Injection: A Survey." In: *ACM Comput. Surv.* 48.3 (Feb. 2016), 44:1–44:55 (cit. on pp. 10, 11).
- [Net] Netflix. *Hystrix*. Accessed on: 2019-04-08. URL: <https://github.com/netflix/hystrix> (cit. on pp. 9, 19).
- [New15] S. Newman. *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc.", 2015 (cit. on p. 1).
- [NJ09] T. D. Nielsen, F. V. Jensen. *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009 (cit. on p. 27).
- [Nyy07] M. T. Nygard. *Release It! Design and Deploy Production-Ready Software*. Apr. 2007, pp. 20–144 (cit. on pp. 8–10).

## Bibliography

---

- [opea] opentracing.io. *OpenTracing*. Accessed on: 2019-04-04. URL: <https://opentracing.io/docs/overview/what-is-tracing/> (cit. on p. 16).
- [opeb] opentracing.io. *OpenTracing*. Accessed on: 2019-04-04 (cit. on p. 16).
- [opec] opentracing.io. *OpenTracing*. Accessed on: 2019-04-04. URL: <https://opentracing.io/docs/supported-languages/> (cit. on p. 16).
- [oped] opentracing.io. *OpenTracing*. Accessed on: 2019-04-04. URL: <https://opentracing.io/docs/supported-tracers/> (cit. on p. 16).
- [QCW07] X. Qu, M. B. Cohen, K. M. Woolf. “Combinatorial interaction regression testing: A study of test case generation and prioritization.” In: *2007 IEEE International Conference on Software Maintenance*. IEEE. 2007, pp. 255–264 (cit. on p. 15).
- [RHB+17] C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones, A. Basiri. *Chaos Engineering*. O’Reilly Media, 2017 (cit. on pp. 2, 11, 12, 31, 76).
- [RM+09] I. ur Rehman, S. U. R. Malik, et al. “The impact of test case reduction and prioritization on software testing effectiveness.” In: *2009 International Conference on Emerging Technologies*. IEEE. 2009, pp. 416–421 (cit. on p. 13).
- [RUCH01] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. “Prioritizing test cases for regression testing.” In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948 (cit. on p. 14).
- [SB+98] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998 (cit. on pp. 21–26, 54).
- [SGMM17a] H. Spieker, A. Gotlieb, D. Marijan, M. Mossige. “Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration.” In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 12–22 (cit. on p. 22).
- [SGMM17b] H. Spieker, A. Gotlieb, D. Marijan, M. Mossige. “Reinforcement learning for automatic test case prioritization and selection in continuous integration.” In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. 2017, pp. 12–22 (cit. on pp. 15, 33–35, 64, 67, 105).
- [Sig16] B. Sigelman. *Towards Turnkey Distributed Tracing*. Accessed on: 2019-04-04. 2016. URL: <https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736> (cit. on p. 16).

- [VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.” In: *2015 10th Computing Colombian Conference*. IEEE, 2015 (cit. on p. 7).
- [WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 106, 107).

All links were last followed on July 07, 2019.



## **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature