Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master Thesis

Multi-detector Framework to Detect and Localize Failures in Microservice Applications

Author: Kai Zhang (VU ID: 12712469 UVA ID:12712469)

1st supervisor: Alexandru Iosup
daily supervisor: Sacheendra Talluri
2nd reader: Animesh Trivedi

A thesis submitted in fulfillment of the requirements for *the joint UvA-VU Master of Science degree in Computer Science*

October 4, 2021

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

It is popular for large software to follow a microservice architecture instead of a monolithic architecture. Each module of the original monolith becomes a dedicated service in a new system and the service communicate with each other over the network. This architecture change leads to a change in failure detection methods. In our work, we first discussed common failures that occur in microservice applications. We present the challenges posed by the failures and the microservice architecture to failure detection. Finally, we implement a failure detector which combines detectors for different failures into a single unified framework. The failure detector implementation includes two parts: static file parser and run-time failure detector. The parser detects common failure patterns and instruments the application for runtime failure detection. The failure detector is designed to handle the following failures at runtime: error swallow, error unhandled, logic failure, data corruption, vulnerable operations, and infinite loops. We evaluate the failure detector on a microservice research project, and obtain over 90% accurate rate, 6% memory overhead, 35% CPU overhead, and 5% request waiting overhead.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

The faster time to market, high scalability and ease of deployment advantages (18)(19) of the microservice architecture(20) have made it increasingly popular. This microservice architecture is here to stay in this digital society, providing services for users in different industries, governments, and academia. The report "Microservices Adoption in 2020"(21) from O'Reilly shows that more than three quarters (77 percent) of businesses have now adopted microservices, which includes numerous network service providers that we are familiar with, such as Netflix, Amazon, Uber, eBay and so on(22).

Because of the large number of people served by the applications that follow the microservice architecture, service failure will lead to time and economic losses to both service providers and users. People now expect services based on the microservice architecture to have high availability. In order to achieve this goal, an effective microservice platform failure detector is very necessary.

The failure detector is a tool used to detect any failure and help the developer find out the reason for failure of the computer system. It can be used to detect failed or partially functioning hardware. It can also be used to detect software service failures due to human error, logical problems, etc.

The failure detector developed in this article refers to the failure detector at the software level of the microservice architecture. Due to the rapid development of distributed computing and microservices, in the production environment, the basic hardware architecture has a dedicated team to build and maintain without programmers worrying about it. On the other hand, microservice architecture refers to a new software development architecture. Developers do not need to care about the underlying hardware, but instead focus more on the writing of microservice code. So this article will not focus on failure detectors other than software related failure detectors.

1. INTRODUCTION

In the microservice architecture, the services running independently in the container combined with the communication network linking different services give it the advantages of highly scalable and easy to deploy. However, because of this special structure, the performance and system isolation brought by the container have brought huge challenges to the failure detection in the microservice architecture. There are three obstacles to make a good microservice failure detector: how to detect software failure in different services and different containers, how to make different failure detectors in different containers work together, and how to trace the source of failure.

First, although we can now see countless failure detectors, the vast majority of failure detectors are developed for monolithic architectures, or large Internet companies have their own internal tools. Second, most developers use software system log records or failure messages displayed on the console to detect and analyze failure. Such a method may be misled in the failure diagnosis process due to the lack of relevant system information and the running status of related services, and finally waste a lot of time. Developers need an effective detector system to help them detect failures on the microservice architecture platform. This problem is particularly significant on microservice platforms, because different microservices are not interoperable from system information to runtime variable data, which brings great challenges to failure detection. The third and the final point is that in the actual deployment of microservices, different failures in the system may occur at the same time, and the existing failure detectors published in each paper mostly point to a certain type of failure. How to make different failure detectors or failure detection algorithms work together is rarely studied.

In our paper, we target to design a failure detector, whose name is MFD (Microservice Failure Detector), which is 1) Detect most programming related failures (hardware failures are not included) 2) Can systematically detect, trace, localize failure for the user and get related failure context 3) Easy to extend so that can easily update or add new mechanism in future 4) Do not depend on language special component, for example, Virtual Machine in Java, so that the detector can be implemented in all kinds of programming language 5) Can help the microservice software has high availability, which is one important QoS metric in cloud computing(23).

In this paper, we introduced and developed MFD (Microservice failure detector): It has many effective failure detection methods that migrate from the monolithic architecture or SOA architecture, and uses the easy-to-find software architecture to make different detection algorithms work together to find the software failures on the microservice architecture and record the relevant failure context. Besides, it has the following characteristics: 1)

Detect most programming related failures (hardware failures are not included) 2) Can systematically detect, trace, localize failure for the user and get related failure context 3) Easy to extend so that can easily update or add new mechanism in future 4)) Do not depend on language special component, for example, Virtual Machine in Java, so that the detector can be implemented in all kinds of programming language 5) Can help the microservice software has high availability, which is one important QoS metric in cloud computing(23).

In addition, we also introduced two failure simulation methods for the detector experimental evaluation to get the overhead and performance of our failure detector.

1.1 Problem Statement

In this section, we point out four key issues found in the microservice platform failure detector survey that can reveal new insights and requirements for failure detection tools and technologies.

First of all, in the current research, there is no such paper to summarize failures, classify failures, and analyze the failures reasons in the microservice architecture. For issues that have such an important impact, it is surprising that not only is there no article discusses potential failure from vulnerable components in the system aspect, but also there is no microservice failure summarize from actual application. This makes the failure detector users and new developers have no systematic understanding of what software failures live in the microservice architecture, and can only study assume the failures type and trigger location are familiar to monolithic architecture (3). This has led to developers cannot develop the failure detector suitable for the microservice architecture.

Secondly, we need a framework that is easy to expand and can be used to cooperate with various software failure detectors and software failure detection algorithms. In many papers, there are many excellent failure detectors, such as (5) (1)(24)(25) to detect data corruption in runtime, (2) static code analyzer to detect infinite loop without running the software, statistical correlation failure detector (6) (26) (7) (3) (8), (9), (10), (11) (12) (13) (14) by parsing logs. However, no matter how good a detector is, it has its own limitations. The users of the detector need to choose the right detector according to their needs. Besides, there will be many potential failures in a system at the same time, and we need different detectors to cooperate together. How to prevent the detectors from influencing each other and effectively share information with each other to better cooperate is needed for developers.

Third, we need to comprehensively evaluate the failure detector with common microservice software structure, service pressure, and all kinds of possible failure in the real world. At present, we cannot find a scientific and effective method to test and evaluate microservice failure detectors. The common method in the past was to add failures to the code artificially. In this case, the developer of the failure detector knows where and when the failure will occur. Else, find a well-known software system to test the version that knows the specific failure. However, the result is not convincing enough, because these failures types, locations, related context are known to the detector developer. The failures are not inserted randomly, so the detector user does not know if a detector can achieve the same result on other systems.

The fourth and final point is that an open-source failure detector that is easy to obtain and contribute is very important for failure detector developers and users. An open-source platform can not only reduce the developer's use cost, but also continuously provide suggestions for detectors or contribute better failure detection algorithms during use. At the same time, it also helps the communication between practitioners to help each other make wise decisions in the process of use.

1.2 Research Questions

In order to build an effective error detector for the microservice architecture and conduct a scientific and effective performance evaluation, we propose the following 4 research questions (RQs):

RQ1:What is the state-of-the-art in failure detection techniques?

Before we build detectors for the microservice architecture, we need to know the-state-of-art failure detection methods. We need to know the core algorithm and design idea of these detectors and then migrate them to the microservice architecture. Moreover, there's no such paper to summarize and briefly introduce popular failure detectors in recent years.

RQ2: What properties of a microservice architecture make building a failure detector difficult?

We first need to investigate what kinds of failures occur in applications that follow the microservice architecture. It is difficult as existing studies are either too broad to use in our paper, for example many papers even include hardware related and nature disaster related failures, or some software failure discussed in the paper is not related to the microservice architecture. At the same time, new components in the architecture bring novel failures which we need to gather from many articles.

We need to know what features of the microservice architecture prevent developers from locating and finding where the failure occurs. Answering this question is not easy, because different developers have different ideas. We need to find the answer from GitHub, authoritative article, and forums where microservice architecture developers are active.

RQ3: How to design a failure detector that addresses concerns specific to the microservice architecture?

The answer to this question is the core contribution of our paper. We target at to design an easy extendable, coding language independent, system independent (easy to migrate) failure detector framework for microservice architecture.

RQ4: How to evaluate our failure detector on microservice architectures?

Though there are many failure detector evaluation, however they are not based on the microservice architecture or many of them only evaluate a single kind of failure. In microservice architecture's failure detector evaluation it includes 5 core but difficult elements: Design failure evaluation experiment, choose and write a testing microservice software, simulate failure, detect failures and analyze results.

1.3 Approach

RQ1:To answer this research question, we conducted a literature survey in the area of failure detectors which were published in recent ten years. This will help to know state-of-art failure detectors' algorithm, suitable apply condition, trade-offs during the implementation. Moreover, how and why did these detectors evolved is also important. We need to put more focus on the failure detection algorithms that are widely applied in the industry to know the different algorithms' limitations, advantages and also users' core requirements.

RQ2: To answer this research question, we did two important things. First, we conducted a literature survey on failure classification paper from well-known publisher channel to extract software failure information. This helps us to get a comprehensive view on the way to know, classify and summarize the failure happens in real life. Secondly, we need to know what new failure may happen on the microservice architecture and what challenge we may face to design and implement the failure detector. We searched vulnerable components in microservice, new failures happens in microservice, obstacle on microservice failure detection from well-known tech website, GitHub and cloud computing related papers. This is because though the microservice is widely used, it is rather new and no peer-reviewed paper focused on the microservice's failure. What we can do is to gather information from the blog or the website has high influence on the microservice architecture or from the cloud computing failure-related paper that talked about the microservice architecture as special condition.

RQ3: To solve this research question, first, we use the requirements we discussed in RQ2 as a design principle. Then, we combined plug-in architecture advantages and microservice architecture's communication interface to build the failure detector's core framework. Finally, we select the relevant failure detectors' algorithms which we found in related work as our implementation's basic detectors. In this way, we build a prototype of a framework to combine different sub-detectors for detecting and localizing failures on the microservice platform.

RQ4: To answer this question, we did an experimental evaluation on it. The evaluation's test software is from a famous research group. We have two types of failure injectors. One is written by ourselves to inject logic error, the other one is 'strace' to inject interaction error like I/O and network communication. We simulate the workload in real life to get the overhead and performance of our failure detector.

1.4 Main contributions

In this thesis we provide the following Technical, Conceptual, and Experimental contributions, mapped to the research questions and chapters that it answers:

- (Conceptual, RQ1) Literature survey on the most popular and the state-of-art failures detection methods in academia and industry filed. (Chapter 2)
- (Conceptual, RQ2) System survey on the microservice architecture's failures and discussion on failure trigger reasons and failures findings. (Chapter 3)
- (Conceptual, RQ2) Analysis of the challenge and critical issues on microservice failure detector failure implementation. (Chapter 3)
- (Conceptual, RQ3) Design overview and requirement analysis of MFD, gives a sub-detectors combing framework prototype to reader (Chapter 4)
- (Technical, RQ3) Design, implement, deployment the failure detector framework and corresponding sub-failure-detectors. (Chapter 4)
- (Technical, RQ4) Find and rewrite a microservice research software to prepare for experimental evaluation. First, the software simulates the real architecture and components in real industry deployment. Moreover, add the service container monitor tools is also important. (Chapter 5)

- (Technical, RQ4) Implement a failure injector to inject failures in our evaluation software during the detector's evaluation.(Chapter 5)
- (Experimental, RQ4) Design and deployment of experiments for MFD on the high-performance computer.(Chapter 5)
- (Experimental, RQ4) Quantitative the overhead, precision, false alarm of the evaluation result. Demonstration on the advantages of MFD. Analysis limitation and corresponding reason. (Chapter 5)

2

Related Failure Detectors

This section will review failure detection-related works that can inspire MFD development. In this section, we will briefly outline the relevant failure detector technologies in the past five years. We classified failure detectors, the result can be seen in Table 2.1, and analyze the existing error detection technologies that may apply to the MFD Detector.

Detector Type	Target Failure	Precision	Remark
Watch dog	Partial Failure	91%	Need file parse time
Runtime Checksum/Duplicator	Data Corruption	99%	Need extra memory and may slow the functions
Static Code Analyser	Infinite Loop	95%	Hard on implementation
Statistical Correlation Detector	All Kinds of Failures	77%	
Machine Learning Detector	All Kinds of Failures	88%-92%	don't know effect after migration to other application

Table 2.1: Related failure & failure detectors. (Source: table constructed from published results. (5) (2) (1) (6) (7)(8)(9)(10)(11)(12)(13)(14).)

Viewing all kinds of detectors during the research is an impossible task, we set the criteria to filter error detectors as follows:

1. The failure detectors that use mainstream detecting algorithm of recent years in industry or academia. The detectors' paper should include detector design principle, detector design algorithm and detector experimental evaluation & analysis.

2. The detection granularity of the detector needs to be at least at the software's function level. This means the detectors that detect failures of nodes in clusters, the detector detects hardware failure, etc. will not be included in this section.

3. Tracing and localizing failure is a challenge in detector design. This section will also introduce related detectors that have failure tracing techniques or failure locating techniques through networks.

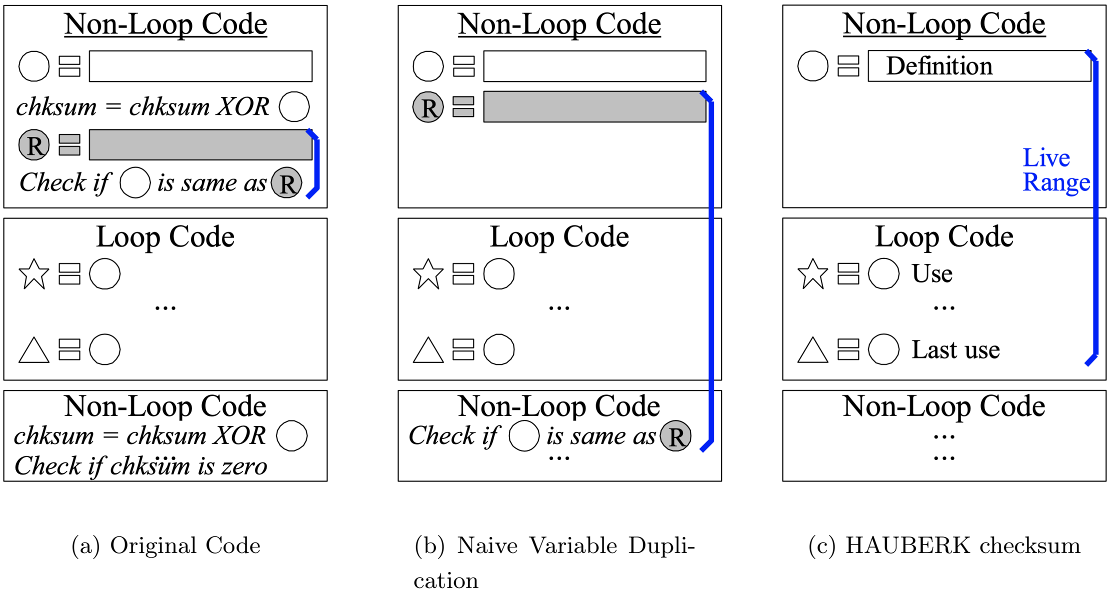
2.1 Watch dog - Runtime Partial Failure Detector

The implementation from Chang Lou et al.(5) called OmegaGen uses watchdog, which is a run-time selective detector. First, the watchdog searches the location of long-running blocks (like while loop) and vulnerable code blocks (like I/O, Network). The watchdog packs detected code blocks and run packed code parallel with the original code. It checks the result at a checkpoint, if there's a failure detected, it will generate a corresponding report for the developers.

There is a very important point was discussed in (5) that we need to pay attention to. The run-time detector interacts with local variables, or event I/O, networks during run-time can cause side effects or even inject failure to the original function. We need to make sure our failure detector has no side effects.

2.2 Checksum/Duplicator - Run-time Data Corruption Detector

The implementation from Keun Soo Yim et al. called HAUBERK (1) focuses on data corruption run-time full-range detector. It provides two high efficient methods. One is the naive variable duplication method, the other one is the checksum duplication method(HAUBERK). The comparison between original code, naive variable duplication, and HAUBERK checksum can be seen in Figure 2.1.



(a) Original Code

(b) Naive Variable Duplication

(c) HAUBERK checksum

Figure 2.1: Run-time data corruption detector. (Source: image adapted from (1))

The naive variable duplication method duplicates the variables after the variable declaration or assignment. The variable will be checked if they are the same before the next declaration, assignment, or the end of the function.

The checksum duplication method(HAUBERK) uses 4-bit checksums to verify the final variable status. The principle behind this method is familiar to the ECC memory(24)(25). First, set checksum by target variable after declaration. Secondly, duplicate the target variable after the variable. Thirdly, check if the duplicate variable is the same as the original variable to ensure the variable is not corrupt during checksum generating time. (The duplicated variable can be removed/deleted from memory after this stage). Later, update the checksum before the next declaration, assignment, or the end of the function. Finally, if the checksum is 0 then the variable is not corrupted, otherwise, it is corrupted.

2.3 Static Code Analyzer

The implementation called DScope from Ting Dai et al. (2) used this idea. It is a static code analysis detector to detect an infinite loop hang due to corruption. This detector targets detecting possible infinite loops raised by data corruptions in cloud services. We can get an example in Listing 2.1. In the example, if the variable `last` is corrupted then the while loop flag `fileComplete` is never set to be true, and causing infinite loop in the function completeFile. This kind of data corruption is even hard to be detected by previously introduced data corruption detector in section 2.2, cause the input data can be already corrupted.

Listing 2.1: Infinite loop example 1. (Source: code adapted from (2))

```
    // DFSOutputStream.java #HDFS-5438 (v0.23.0)
1665 private void completeFile(ExtendedBlock last )...{
       ...
1667  boolean fileComplete = false;
1668  while (!fileComplete) {
1669    fileComplete = dfsClient.namenode.complete(src, dfsClient.clientName, last);
         ...
      }
1689 }
```

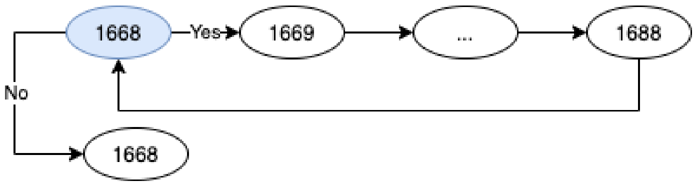


Figure 2.2: Loop path extraction of listing 2.1 (Source: image adapted from (2))

This method has three basic steps:

1. Extract Loop path to graph (like Figure 2.2)

2. Check if loop depends on vulnerable operations, like I/O.

3. Analysis loop stride and bound to check if all paths of the loop can reach the exit condition of a loop even if the variables that the exit flag depend on corrupt.

This method can detect, or in other words "predict" the infinite loop failure before the production stage. This can not only lower the cost of deployment stage bug detection, but also save time for service development. However, this one is the hardest to develop.

2.4 Machine Learning Detector

In recent years, AI/ML technology has shined in various fields, and the same is true in the field of general-purpose error detectors.

All the Machine Learning related detectors, no matter they monolithic architecture failure (8), (9), (10), (11), (12) (13) or from microservice architecture failure from (14), are all belong to supervised machine learning methods or supervised deep learning methods (e.g. logic regression, RNN, Random Forest, Support Vector Classifier).

Parsing logs and analyzing system metrics are their common ways to achieve the final target, and they have a common core framework

The framework includes two main parts, offline and online. The offline part start first, its basic steps are preparing data, identify events and event mining. Then the online part uses logs or metric data to predict failure. If the detector encounters a failure but fail to predict, it will update to the failure detection model.

There are various advantages of this method. The most important one is a prediction! This cannot be done by any other detector. Moreover, this method has high precision, ranged from 88%(9) to 99%(11). Last but not least, it can detect all types of failures from software aspect to hardware aspect, from code line granularity to cluster granularity.

However, the coin has two sides, it also has many drawbacks. Initially, where, when, how to collect the log and the system metric? Secondly, how to cold start the model training, there are two methods. One is using historical data, the other one has injected failure and train the model. These two methods cannot 100% cover all kinds of failures, especially silent failure. The third problem is model portability. Due to it is based on log or system metrics, it is hard to know its precision when the system cluster scale to larger or smaller. Also, the log and system metric data vary from system to system, one trained model can have a very low effect on other platforms.

2.5 Statistical Correlation Detector

The majority research (6), (26), (7) of these methods are base on system metrics' correlation with failures. The most representative implementation is from the Z.-Y. Wang's team (3). The detector tries to detect all kinds of software related bugs in the microservice architecture system on function granularity. It includes three basic parts, service trajectory monitoring, service trajectory construction, and diagnostic.

The detection flow of this method is familiar to section 2.4. The detector collects system metric data and use PCA to get failure correlation value to detect failure.

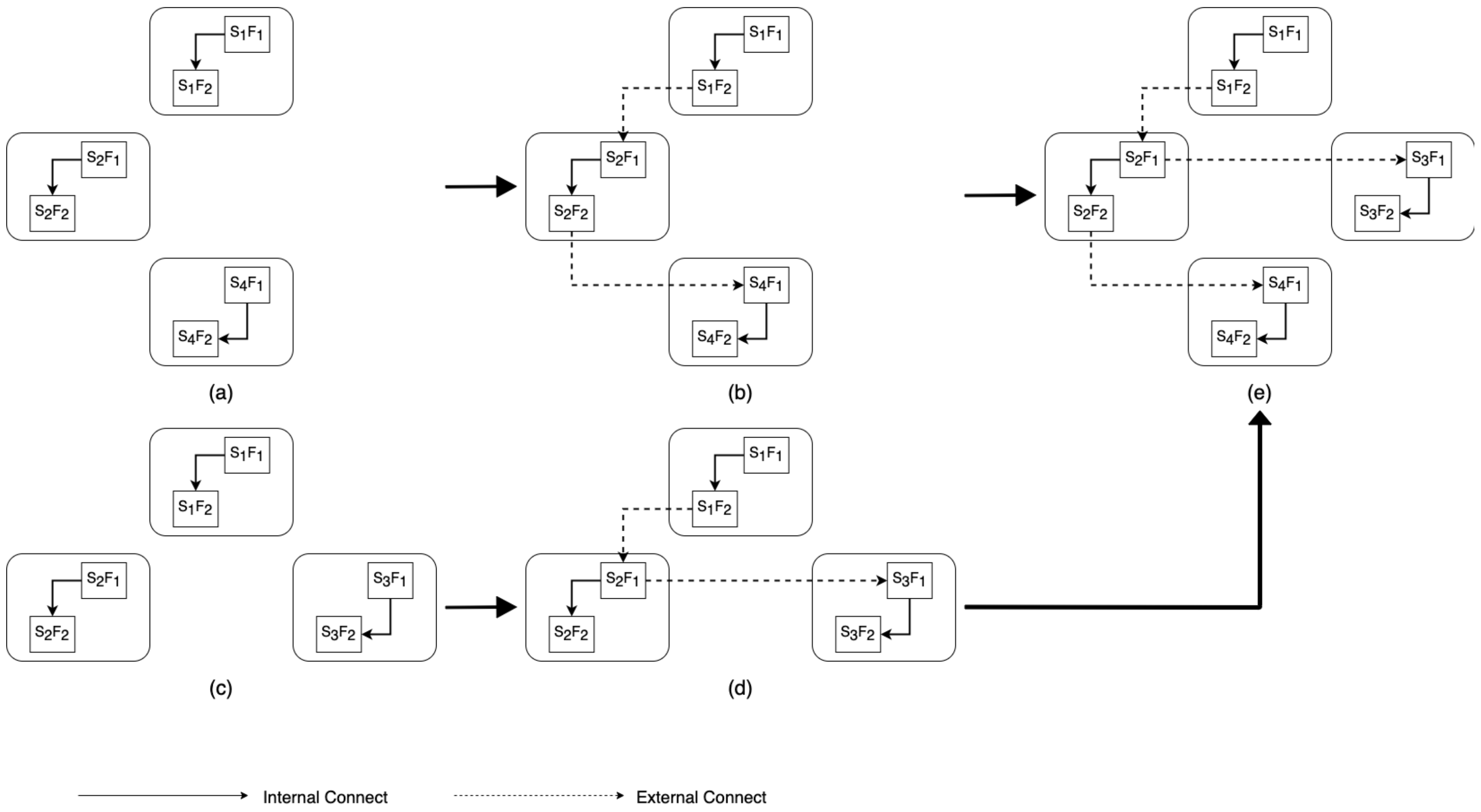


Figure 2.3: Services action tracing from Z.-Y. Wang's team. (Source: image taken from (3))

$$\begin{aligned} Message &= (requestUID, methodUID, callerUID, calleeList, info) \\ info &= (callType, serviceUID, order, startTime, endTime, duration) \end{aligned} \tag{2.1}$$

Figure 2.4: Message equation. (Source: equation adapted from (3))

The most special part of this method is the actions of its service tracing and microservice relation network establishing's strategy. We will use Figure 2.3 to combine with examples to help illustrate the action tracing method. In the examples, Request A's traverse path between services is S1 → S2 → S3 and Request B's traverse path between services is S1 → S2 → S4. In the internal of each service, it used old-fashioned methods. The detector injects Instrumentation into each service, the Instrumentation logs function executes elements and order then form the relation network of each service's internal code(Figure 2.3 (a)). When a function in service wants to call the external service function, the caller function packs its execution information in the format of Message Equation Figure 2.4 then sends it to the destination function. The caller message makes two subnetworks connect together(Figure 2.3 (b)). With the incoming request executing, the more different request the more services connections add into the network(Figure 2.3 (c), (d), (e)).

3

Requirements for the Failure Detector

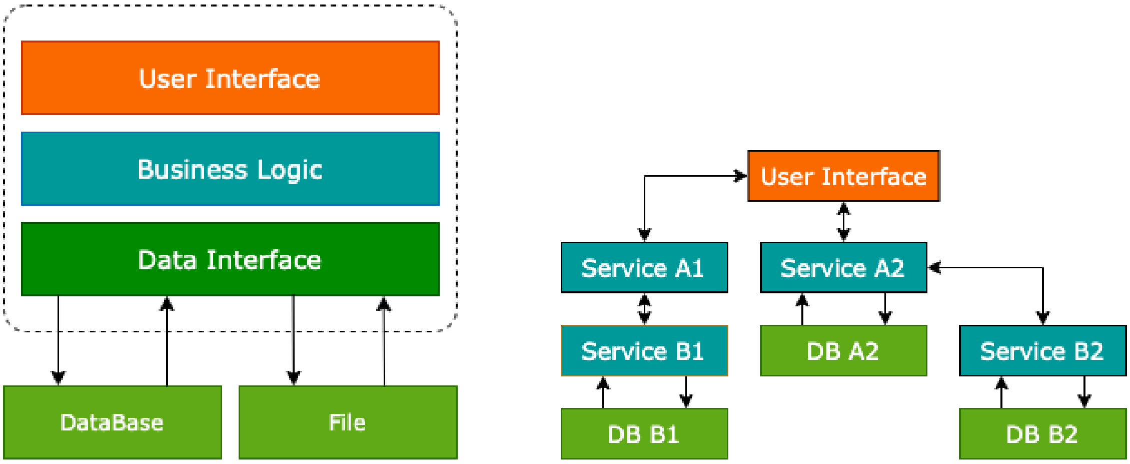
3.1 Challenge in Design a microservice Architecture Failure Detector

The software development architecture evolved from the monolithic architecture to the microservice architecture. Along with the numerous benefits (e.g. better scalability, greater agility and faster development cycles, module isolation), the microservices bring to software development, it also brings many challenges to system monitoring and failure detection. In this section, we will talk about the challenges that we may face in the failure detector development and deployment. The microservice architecture is defined by (27) as:

> *An approach to software and systems architecture that builds on the well-established concept of modularization but emphasizes technical boundaries. Each module—each microservice is implemented and operated as a small yet independent system, offering access to its internal logic and data through a well-defined network interface.(27)*

From the definition, we can transfer the definition text to a graph example compared with the monolithic architecture example as shown in Fig. 3.1. Based on the definition and graphs, we summarized 5 challenges in the microservice's failure detector research:

- **Failure Injection Location Changed**
  The monolithic architecture (MoA) has multiple software layers, the failure injector can inject failure into the layers on demand. In contrast, when it comes to microservices architecture (MiA), the software layer is no longer exists and replaced with



(a) Monolithic Architecture

(b) Microservices Architecture

Figure 3.1: Monolithic and microservices architecture. (Source: image adapted from (4))

a distributed microservices network. A failure injector only can inject failures in service. For example, if the tester wants to inject an I/O failure/congestion into a software. In MoA, the tester can use tools to inject a failure in the data layer. In MiA, the tester only can inject a failure in a service's data layer. For instance, the tester can inject a failure between service B2 and DB B2 but won't influence the I/O connection between service B1 and DB B1.

Moreover, all services in MiA communicate with each other by an interface, which does not exist in MoA. This internal network is also available to inject failures.

- **Failure Injection Method Complexity**
  In the MiA, different services can run with different statuses, like stop, starting, running, hang. In contrast, the MoA's components' status is the same. The failure injection plan changing causes the failure inject methods to change. Furthermore, the MiA services communicate with each other by an interface. Many new protocols, new mechanisms, and new hardware engage to support this target. Meanwhile, more outages can happen in these places and require new failure injectors/methods to simulate.

- **Cross-cutting concerns**
  The definition of microservices means that each module in the MiA can run as an independent system, which means that each service has/needs its configuration,

recorder, indicator monitor, and health checker. Testers must deal with many cross-cutting concerns challenges. For example, when the tester needs to check how much system resource does a request require? In MoA the testers only need to check the resource consumption between request and finish time. For the microservice architecture, the tester has to check each related services' start time, finish time, and corresponding resource consumption for the request, then summarize resource consumption to get a final result.

- **Distributed Detector**
  In the MoA and traditional cloud architecture, no matter if a detector is a statistical file failure detector(2), a statistical correlation failure detector (6), (26), (7), a runtime outage detector (1), (28), (5) or a machine learning (ML) failure detector (8), (9), (10), (11), (12), (13), (14), the detector has only one core detector to do troubleshoot. It is not easy to make a one core failure detector in MiA, even if in an external system metric/system log ML detector(14) also needs a revolution.) How to make a distributed failure detector or make multiple small detectors' in each service cooperate is a challenge.

- **Trace Complexity & Failure Localization Complexity**
  Due to the highly coupled components in MoA become loosely coupled services in MiA, the services use new methods to communicate, and it brings a challenge in failure tracing and locating methods. This is because that the failure propagation method of the microservice architecture is different from the traditional method. (For example, in Figure 3.1(b) service A1 depends on service B1. If there is an error in service B1, then A1 will also show an error. The developer should fix the error in service B1 instead of trying to repair service A1, but two error reports may mislead the developer.) We need to find new ways to locate where the error actually happened in the microservice framework.

3.2 Understading Failures targetede by MFD

The start of this section presents all kinds of failures in related structures and newly emerged failures in microservice architecture. Then this section discusses what kind of failures can be detected by the MFD, so that readers can see whether the MFD detector is suitable for their research topic or application. Besides, the scope of MFD is included. Finally, the failure findings part will discuss failure.

Failures live in Microservice Architecture

The software architecture is constantly evolving. Before we start to design an error detector for a new architecture, we need to understand the most common and important errors of its previous related architecture. After screening common failures and thinking about the characteristics of the new software architecture, we will know all potential failures on a new architecture. This not only allows the error detector developers to have a clear development direction and plan, but also allows the detector users to clearly know which kinds of error does the new detector can accurately indicate, which kinds of outage can the detector potential dig out, and which kinds of error cannot be detected.

We found that two papers have done detailed investigations and classifications about cloud computing failures. (16) analyzed unexpected outages from 32 famous cloud services in real life like WhatsApp, Ebay, Facebook, etc within 7 years from 2009 to 2015. (15) conducted a thorough study of development and deployment issues of six popular and important cloud systems (Hadoop MapReduce, HDFS, HBase, Cassandra, ZooKeeper, and Flume).

The cloud development and deployment errors described in the two reports are roughly the same. We merge the error classification results of the two reports in a new table. There are two special circumstances about the table we need to point out. First, since the natural disasters and power shortages mentioned in literature (16) are not in the scope of computer science, we have removed these two errors. Secondly, the human error in (16) is 100% coincident with other errors, and the cause of the error is relatively vague, so we removed it too.

Microservices Architecture new concepts bring new potential failures to the failure detection field. As discussed in section 3.1's Failure Injection Location Changed, failures can be injected into the communication interface. The most common failures in communication network are I/O failed, congestion and data corruption(17) and we added *communication interface load*, *communication interface corruption*, *communication interface closed* into previous two literatures' merged table and got Table 3.1.

	Main Outage	Sub Outage	Remark
Hardware	Stop (Fail)	Stop (Fail)	
	Corruption	Corruption	
	Limp Mode	Limp Mode	
Software	Error	Error Swallow	
		Error Unhanded	
	Hang	Infinite loop	
		Deadlock	Multi-threads
	Bugs	Data Corruption	
		Logic-Specific	
		Configuration	Deployment
		Data Races	Multi-threads
	Cross-Service Dependencies	Communication Interface Load	
		Communication Interface Corruption	
		Communication Interface Closed	
		External Traffic Load	Deployment
	Security Attacks	Security Attacks	Deployment

Target
Assist
Cannot

(a) Failure Root Causes

Aspect	Reliability
	Performance
	Availability
	Scalability
	Topology
Impacts	Failed Operations
	Performance Problems
	Component Downtime

High Priority

(b) Aspect & Impacts Failure

Table 3.1: Microservices architecture failures. (Source: table constructed from published results(15)(16)(17))

In the table, the failures are sorted into four main classifications. Root Causes (Hardware), Root Causes (Software), Aspects and Impacts. A failure root cause means a failure with a detailed phenomenon in the system, one root cause, one phenomenon. The *Aspect* failure links to Quality of Service, it can be influenced by many evaluation metrics. This means a root cause failure can influence many Aspects' failures, on the contrary, an aspect failure links to multiple root cause failures. in the system. The *Impact* failure is an implication of the root cause. Root causes influence or leads to the final result of Aspects and Impacts. During MFD plan stage, we choose what failure root causes do the detector needs to detect. In the MFD design stage, Aspects that present quality of service and impacts will also take into consideration.

Target Failure Explanation

Different error detectors have different goals and limitations, so does the MFD. In the following paragraphs, we will illustrate the following 4 points, and the overview can be seen in Table 3.1(a):

1. What kind of failure does the MFD aim at?
2. What kind of failure does it can assist on failure detection?
3. What kind of failure does it not include in current version?
4. What is the MFD's failure localization granularity?

As said in the introduction, we want to design the MFD as a detector that can systematically detect and localize software development failures before or at run-time. As a consequence, the MFD does not have a specific mechanism to check hardware conditions, but can help the users to detect related problems. For example, the MFD can give a warning when a called function relies on services that are not available, which means the MFD will also warn users when the dependent service are down due to hardware reasons. In this case, the MFD can help users to detect hardware-related problems. The MFD also has some limitations on the software side. The deadlock and data-race bugs are linked to Multi-threading. If we want to deal with related failures, the difficulty will increase exponentially. Thus, these two outages are not included in the MFD first version. In addition, the *Configuration*, *External Traffic Load* and the *Security Attacks* belong to deployment outages, which cannot fully controlled and fixed by services developer. These three outages do not include the MFD failure detector scope too. Nevertheless, the MFD runs can monitor and log all variables' data, environment information, configuration setting at the run-time, so it can assist the users to detect the related failure.

We target to detect failures on code-line granularity, and on variable granularity during some circumstances (like nil pointer error). We studied failure is with respect to a statement in a line deviating from the functionalities it is supposed to provide(e.g. data corruption on assignment statement, failed to get data from I/O), which is the function failure or the service failure root cause.

To sum up, the MFD targets at detecting Error Swallow Failure, Error Unhandled Failure, Infinite Loop, Data Corruption, Logic-specific failure, and Communication Related Failure on code-line granularity.

Failures Findings

In this section, we list the failure finding we found during failure-related research. This is important because detect a failure that happens in the system is not the target we purpose, but detect the failure, know the failure, and helps the developers to fix a failure is more important. To achieve the goal we need to know what properties do failures have. What common impacts trigger by failures. What confused developers most during failure diagnosis?

Finding 1. Cloud Services Availability First

From (15)'s research, users often grade systems based on clear performance and availability metrics, which are strongly related to QoS. In addition, (29) found that reducing availability can lead to huge losses, about $285 million have been lost yearly because of the cloud availability downgrade to 99.91%.

Finding 2. Failures Cause Long Service Downtime

From (16) we can know that all software root causes have a maximum downtime of more than 50 hours. The software bug's median downtime is around 6 hours, and its downtime is the highest among all kinds of failure. Moreover, 69% of failures are reported with downtime information.

Finding 3. Failure & Difficult to Diagnosis

As reported in (15), the median diagnosis time is 6 days and 5 hours. Even if locating where was the bug happen is not difficult for developers, for example, the run-time can tell the developer directly where did it stop, and the error handler can tell the developer where was the bug, it is still difficult to fix failures. A common cause is that a failure phenomenon can have multiple causes. For instance, the return of SQL query execution from a database is empty can because of invalid SQL configuration, invalid SQL query, SQL server is down, and so on. The other common reason is insufficient exposure of run-time information to developers, so the developer has to enable the logger and wait until the bug happens again.

Finding 4. Failure & Service Stuck

As (5) says, nearly half (48%) of the partial failures cause some functionality stuck.

Finding 5. Silent Failure

According to (5)'s finding, 15% of the partial failures are silent, for example, data corruption, inconsistency, wrong results, etc. They are usually difficult to discover if there's no specific mechanism during the run-time.

Finding 6. Specific Condition & Failure

According to (15)'s investigation, we know that 71% of failures are triggered by certain environmental conditions, configuration, inputs, or failures propagate from other processes. The related failures cannot be fully detected in the development and production testings. These failures require special mechanisms to detect at run-time. Furthermore, if a run-time detector uses a different configuration or inputs, the detector may fail to detect such failures.

4

Framework Design and Implementation

This chapter first provides an overview of the MFD design (section 4.1). It then illustrates the detailed designs of the AST (Abstract Syntax Tree) File Parser & Detector (section 4.2) and the core idea of Runtime Failure detector (section 4.3). In the final part, we will discuss the detail of this research's real implementation.

4.1 Design Principle

We discussed the failure detector final target clearly in section 3.2, and did research on the related detector which can inspire us in Chapter 2. In this section, we will first introduce our design principle, then we will illustrate the MFD framework overview.

As said in section 3.2, in the design stage the *Aspects* (QoS Related) and the *Impacts* will also take into consideration because they are part of microservice's important evaluation components. The principles help developers to establish a common standard for the implementation stage.

Principle 1: Ensure Microservices' High Availability

The MFD should have a very basic complete failure recovery mechanism.

As Finding 1 (in section 3.2) shows, no matter in what software development architecture, high availability is an essential aspect. Due to different failures can have different phenomenons and impacts. The MFD must have suitable mechanisms to recover service from failure. The basic requirement of failure tolerance is the mechanism needs to recover the server's configurations and variables data to the status before the failure happens and is available to run for the next request. For partial failure, due to has high potential to make stuck (Finding 4 in section 3.2), the tolerance mechanism should stop the hang/slow phenomenon. For complete failure, the tolerance mechanism should recover service from failure.

Principle 2: Traceable and High Failure Location Accuracy

The MFD should detect failures and trace the failure propagation correctly.

The section 3.1 fully discussed that due to the module in Monolithic Architecture change to isolated service system in Microservice Architecture, tracing and failure propagation becomes a challenge. To lower down failure detection time and help developers diagnose failure quickly, the detector should localize failure Accurately, and it is an important property to evaluate the quality of the failure detector.

Principle 3: Log Runtime Content

The MFD should log variable history runtime information after the failure is detected.

From the Findings 3.2, 3.2 we know the failure condition like configuration, system metric, input variable, and variable history data is very important for failure diagnosis. The MFD detector should have a mechanism to log all variable history data no matter what kind of failure is detected.

For example, if line 5 in Listing 4.1 raises an exception, the detector then logs system metrics, inputs, and variable history data (Table 4.1).

Listing 4.1: History data example code.

```
1 func main() {
2 0  var r1 = &Random1{Integer: 1}
3 1  r1.func0()
4 2  r1.Integer += 1
5 3  r1.func1()
6 4  r1.Integer += 1
7 5  r1.func3()
8 }
```

Principle 4: Selectively Highlight Vulnerable actions

There are some vulnerable operations that can cause partial failures, especially the operation like network interaction (includes external request and communication between

Variable Name	Line Number	Value
r1	5	Random1Integer: 3
r1	4	Random1Integer: 3
r1	3	Random1Integer: 2
r1	2	Random1Integer: 2
r1	1	Random1Integer: 1
r1	0	Random1Integer: 1

Table 4.1: History data example data.

internal services.), I/O, or some functions can cause fatal errors because of invalid configuration, wrong input, or wrong logic. The MFD needs to localize them to avoid then cause complete failure or infinite waiting.

Principle 5: Low in coupling and High in cohesion

The MFD's program architecture should be low in coupling and high in cohesion to make sure the MFD is open for extension.

In MFD, there will be multiple small detectors to cooperate with each other to realize MFD's final target. The MFD needs to extend in the future to handle errors like Deadlock, Data races problems, or even cluster level problems. Furthermore, current sub-detectors need to evolve, redesign or remove during or after the first version development. Currently, most detectors are low cohesion detectors, and this is one of the reasons each of they can only one specific failure. Low in coupling and High in cohesion is open for extension and modification, which is suitable for the MFD's requirement.

Principle 6: Prevent Side Effects

The section 2.1 points out the runtime detector has the possibility to corrupt original data and has potential failure risks interacting with local I/O. We need to prevent those side effects.

Structure

In this section, we will give the MFD architecture design overview illustration. The MFD has two basic parts, the AST file parser part, and the runtime detector part.

The AST file parser (which will be explained in detail in (section 4.2) part's main functionality is to extract file information for the detector, insert runtime detectors' code, insert service action tracker and catch ignored variables. Besides, it also has basic logic familiar to Static Code Analyzer (section 2.3) to detect potential infinite loops before code running.

The core parser and the core updater are two main components composed AST file parser part. The core parser extracts the file code and passes extracted information to the updater. The updater injects the required runtime failure detector, variables into the code. One important thing that needs to point out is that the core updater is composed of multiple sub-updater on the developer's demand. The sub-updater can inject its own kind of run time detector to the code or do some file modification.

The runtime failure detector part (which will be explained in detail in (section 4.3) is the part to detect and recover services from failure during runtime. The detector deep copies runtime data and examines it. Once the failure happens, the detector logs the failure type, failure location, current, and related functions variables history data, then starts internal mechanism to ensure the service exit safely.

4.2 AST File Parser Implementation

This section describes AST File Parser's usage of each component in detail. Although it is just used to insert detectors into code, it has many components and programming ideas to make sure the file can be manipulated correctly and extensible.

First, we will discuss two main components with related module in section 4.2. Then this section explains the AST File Parser working approach in section 4.2. The end of this section goes deeper to some sub-updaters' usage and structure in section 4.2.

Main Components

As said in the section introduction, the AST file parser has two main components: the core parser and the core updater. In a real implementation, there are multiple sub-updaters that act as real file updating executors, and the core updater connects the core parser iteration flow and all sub-updaters actions. We will explain core parser, core updater, and multiple updaters one by one, then continue to introduce AST File Parser Flow.

The core parser

The core parser is based on Abstract Syntax Tree(AST), which is a way of representing the syntax of a programming language as a hierarchical tree-like structure(30). In Figure 4.1 we can see an example, the example contains `ast.File` node, `ast.FuncDecl` nodes, `ast.BlockStmt` nodes. Each higher-level class is a code packer for the lower level. A file can contain multiple functions. A function can contain multiple blocks. A block can contain sub-code blocks or multiple code lines. The code line is the smallest level of executable code and target granularity of the MFD (section 3.2). We set File Level (the highest level), Func Level, Block Level, Line Level (the lowest level) as the parser processing level. Meanwhile, we need to analyze variables' data during runtime to detect error swallow, error unhandled, and data corruption failure. We classify the variable and variable properties into element levels. The summary can be seen in Table 4.2.

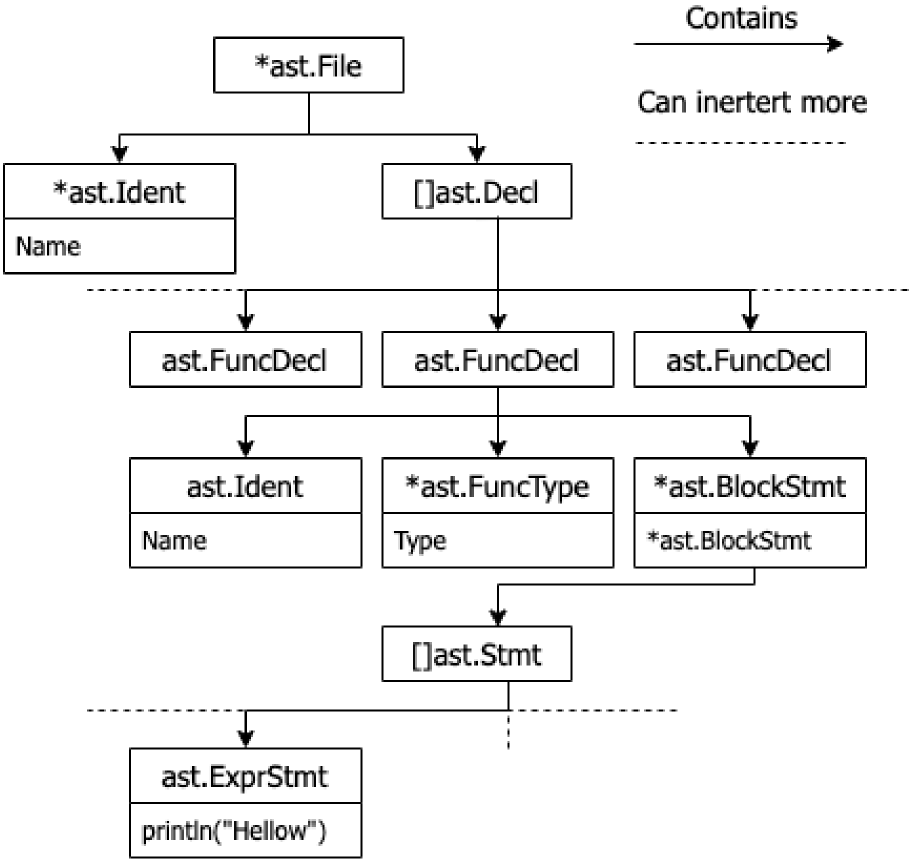


Figure 4.1: AST tree example.

We choose to use ASTs to parse and manipulate code files because it is safer than doing those operations directly on the code as text or on a list of tokens. Manipulating text is always dangerous because it shows the least amount of context. Moreover, the iteration of AST provides a common iteration flow for all sub-updaters to ensure the "Low in coupling and High in cohesion" design principle. The parser walks nodes by the Pre-Order Tree Traversal method(31), which visits the root node then left child, right child. When the parser reaches a node in a new processing level, the node will send to the level's updater

Level	Example
File Level	Valid Code File
Func Level	`func main(){ }`
Block Level	`if(){ }`, `while(){ }`, `for(){ }` ...
Line Level	`var a error`, `a.funcB()`, `a = b.funcC()` ...

Table 4.2: Processing level corresponding code example.

to manipulate code. The relation with updaters illustrates detailed in section 4.2.

The core updater

The core updater's main functionalities are providing common configuration (setting) storage for all sub-updaters and providing a common interface, which is used to communicate with the core parser, for all sub-updaters.

First, the core updater starts and initializes common configurations for all sub-updaters. Then the core updater registers itself into the core parser. Then, all required sub-updaters initialize and register into the core updater. After this step, as long as the sub-updater registers successfully in the core parser, the core parser sends a common configuration for all sub-updaters, which enables different sub-updater to communicate with each other. Finally, the different processing level's nodes can be updated by the corresponding sub-updater through the core updater's interface. The related operation is shown as shown in Figure 4.2's brown arrow line.

This design ensures Principle 5 Low in coupling and High in cohesion. Different sub-updater has related function and target, but different sub-updater for different processing levels can be easily added and removed.

The sub-updaters

The sub-updaters are the real executors to manipulate code files. They implement an interface defined by the core updater. The function of sub-updaters includes creating sub-updaters, updating sub-updaters configuration, extract information from sub-updaters, and modify code in the code parser's AST node iteration progress. All interface functions names and usages are shown in the following list (the parameter and return type of function is not shown in the list):

** The word 'level' in the list is the same meaning as the parser processing level explained in core parser (section 4.2).*

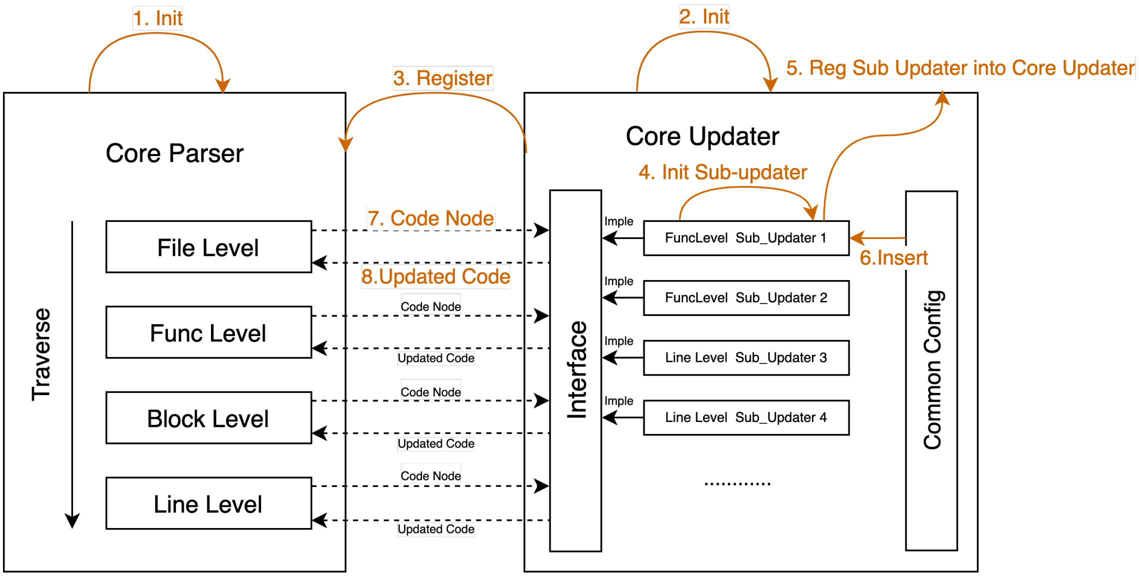


Figure 4.2: Relation and operation between core parser and core updater.

- `Initiate` function. It initializes the sub-updater with all sub-updater's common configuration shared by all sub-updaters and the sub-updater's initialization configuration value from the user's input value or default value when there's no input configuration.

- `GetDefaultConfigValue` function. This one returns the default own configuration value of the sub-updater. This function will be called when a sub-updater's setting is not in the user's input value.

- `SetLevelConfigByComments` function. This function update configuration by comment for the input AST node. If there's no corresponding comment configuration for the sub-updater the configuration of the current level will inherit a higher processing level's configuration (file-level will inherit sub-updater's initialization value).

- `RemoveLevelConfig` function. This function removes the lower level's configuration when the core parser iterates node from a lower level to a higher level (e.g. from line level to block-level).

- `Process` function. This is function is target at modifying, updating, inserting code into parsed AST node by incoming arguments and sub-updater's configuration.

- ExternalGetInfo Under some special circumstances, the core parser depends on executed sub-updater processed data. The required data can be extracted by this function from executed sub-updater.

In the first version of MFD failure detector, we designed seven sub-updaters for it. Three sub-updaters for failure detector: InfiniteLoopDetectorUpdater, DetPanicRecoverDetectorUpdater, ValidatorUpdater, one service trace action sub-updater: DetClientExtractor and three common variable sub-updater: RestoreUnderscoreUpdater, IgnoreIdentUpdater, and IgnoreStructUpdater. We will illustrate these sub-updaters in section 4.2 clearly.

Working Approach

Figure 4.3 shows the basic workflow of the AST File Parser, it has 3 main steps: Prepare for file analyzing, Initialize core component and Walk and Traverse.

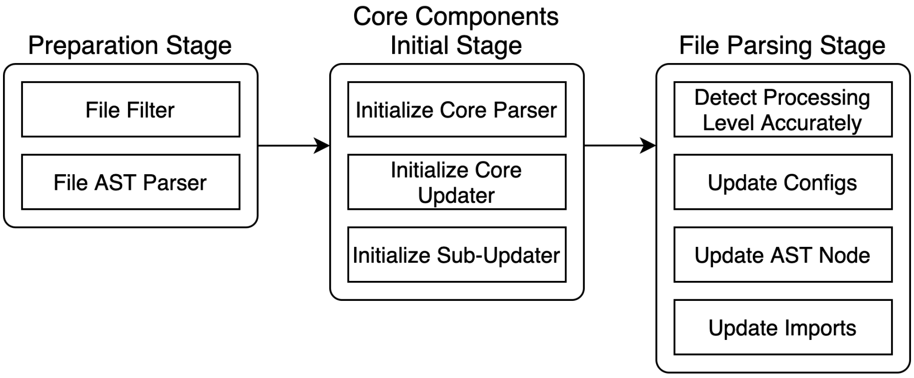


Figure 4.3: AST file parser working approach.

Preparation Stage

Filter and parse project files are the first step of preparation. In a project there are some mature modules and generated files (like GRPC communication protocol generated from proto file), those code blocks are fully tested and validated by many projects and programmers which means they are unlikely to have essential failure and many of them has sound failure recovery mechanism. Besides, they are not the core logic of developed microservices. In this case, we can skip and not examine these files to save project parse time and temporal & spatial resources during runtime. After the file passes the file filter and if it is a code file, the corresponding programming language's parser will extract the code file to the AST tree, the extracted data and original file information stores in memory and pass to the core parser initialization when all project files are examined. The flow is shown in Figure 4.4's Parse File Stage.

Then we need to initialize the core parser, the core updater, and required sub-updaters one by one, the graphical illustration can be seen in Figure 4.4's Core Component Initialization Stage. The core parser is initialized with parsed AST file nodes, related file information, and each processing level updater's configuration. Then the system initializes a new core updater with the user's input configuration. After this, there's a list of required detector sub-updaters, which are listed in section 4.2, create and register into the core updater. Once all sub-updater creates successfully, the core updater inserts a common configuration that shares configurations for all sub-updaters and can be used to communicate with other sub-updaters. Finally, the core updater registers into the core parser and the preparation stage is finished.

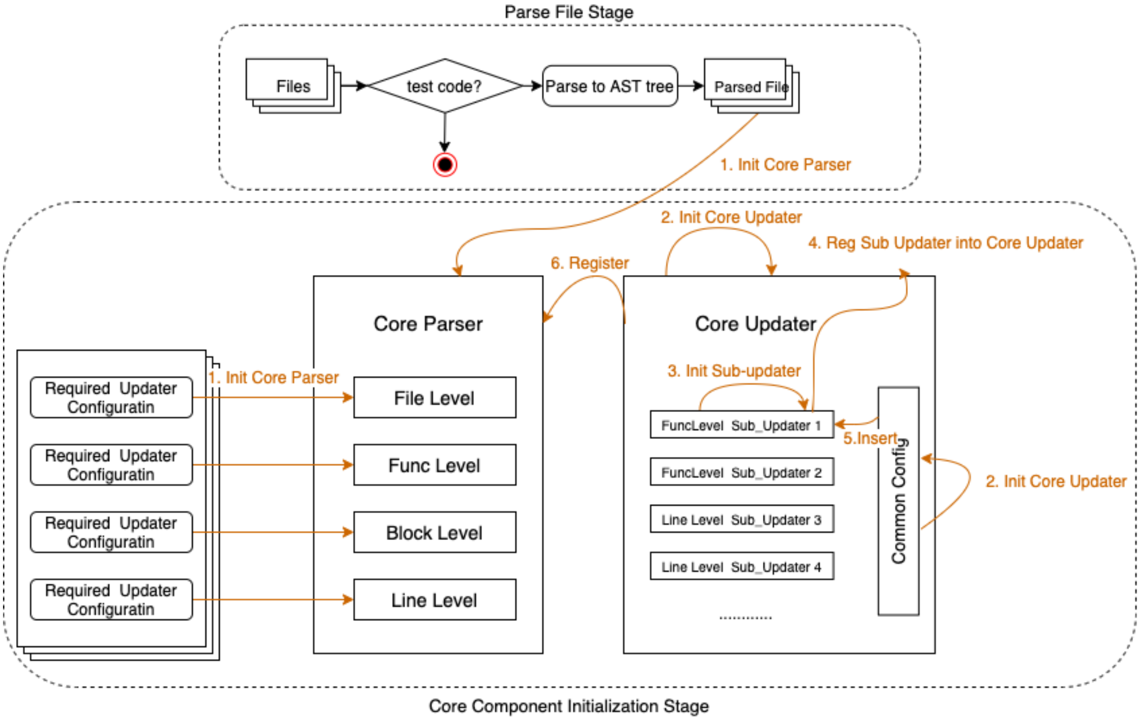


Figure 4.4: Preparation stage.

Traverse AST Tree

Walk and Update In this stage, the core parser has a set of AST file nodes that need to process. The core parser iterates each AST file node, in the iteration, the core parser uses a tree walker to traverse the AST file node in Pre-Order and interact with sub-updater to manipulate files. As said in section 4.2, the core parser has four processing levels: File Level, Func Level, Block Level, and Line Level. The walker walk starts from File Level, and won't go to children of Line Level. There's a walker traversing example in Figure 4.5's Core Parser Iteration part. When the walker reaches a processing level, the core parser must use the core updater's `UpdateConfig` with the current walker cursor's node-related information to renew updaters' configuration. Then the core parser can interact with sub-updaters to modify code on demand. In the MFD, the File processing level does not have a sub-updater. The Func Level uses `DetClientExtractor` to draw relation map among services and functions. The Block level uses `InfiniteLoopDetectorUpdater` sub-updater to inject runtime infinite loop detectors for `for` loop and `while` loop code blocks and this sub-updater uses static file analysis method to check potential infinite loop, which can warn the user before code running. The line-level has `IgnoreIdentUpdater`, `IgnoreStructUpdater`, `DetPanicRecoverDetectorUpdater`, `ValidatorUpdater`, `RestoreUnderscoreUpdater`, to inject runtime validator and runtime panic failure detector. If there's any detector injects into the code file, the detector injects detector declaration and import into the code file. The figure illustration is shown in Figure 4.5.

Detect Code Line Granularity File, Func, Block Level has specific classes (`ast.File`, `ast.FuncDecl`, `ast.BlockStmt`) as indicators to know the walker's cursor reaches a new processing level. But how does the parser know that it reached the line level? A code line is the smallest granularity that contains executable code. It has three basic statement types in any programming language: declare statement, expression statement, and assignment statement, the examples are shown in Table 4.3. When the core parser walker encounter `ast.DeclStmt`, `ast.ExprStmt` and `ast.AssiStmt` the walker knows it reaches the code line level. Due to the line processor is the lowest processing level, so the walker won't go to the child node as shown in Figure 4.5, how to deal with element level will discuss in the next part.

Type	Example
Expression Statement	`a.b.c()`
Declare Statement	`var a = fmt.Sprint("")`, `var a int` ...
Assignment Statement	`a := b.c.d()`...

Table 4.3: Code statement example

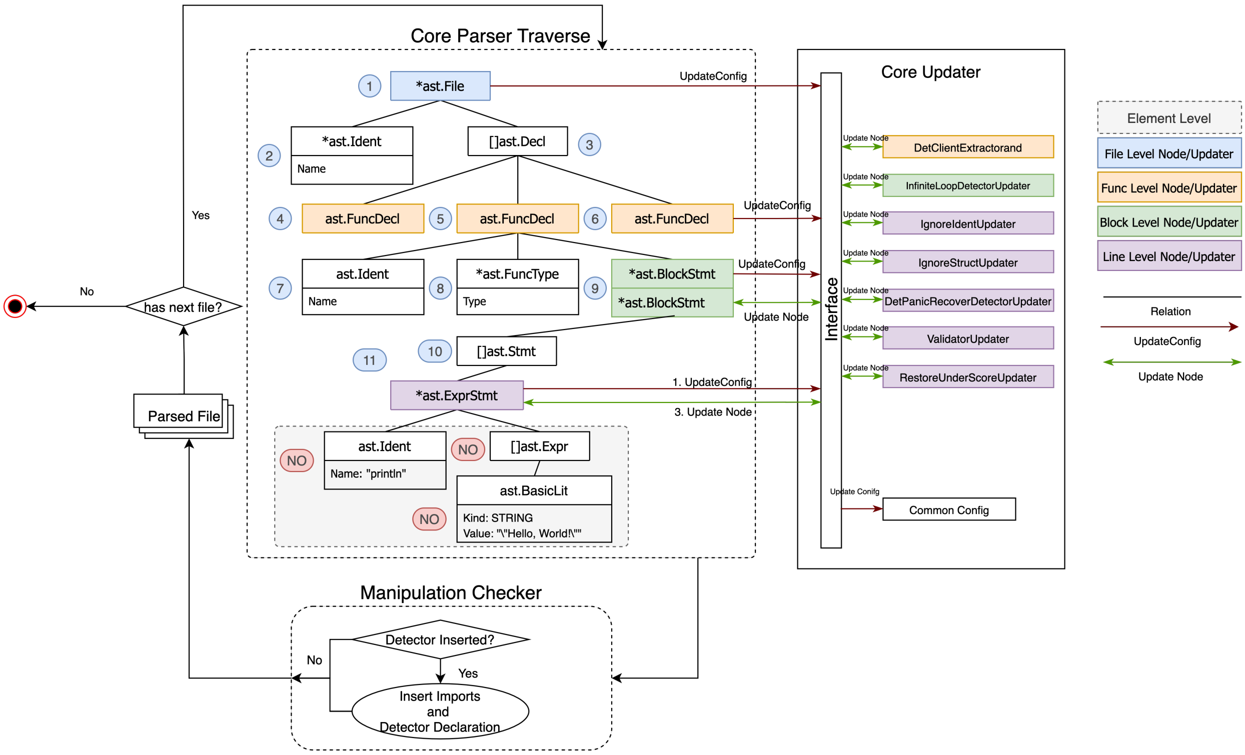


Figure 4.5: Walk and traverse.

Element Level Extractor No matter which kind of runtime failure detector needs to interact with variables and properties in original code. The detector-related sub-updaters need a module to extract element level information before the sub-updaters send the AST node to sub-updater to manipulate code file, we call the module as *Element Level Extractor*. This means the `Process` operation to manipulate has three subcomponents shown in Figure 4.6: extract elements, send extracted elements and AST node to an updater. The updater returns the processed AST node.

Insert Runtime Detector Code Stage

First, we analyze the input of the extractor. The input of element extractor has four types, declare statement, expression statement, and assignment statement from code line and expression from loop exit condition. The common part of the input is that all input types are composed by expression (`ast.Expr` in Golang). The difference is that expression statement and assignment statement has two sides split by token, the split expressions on the left side are right side execution return result. So the basic logic of extractor is the

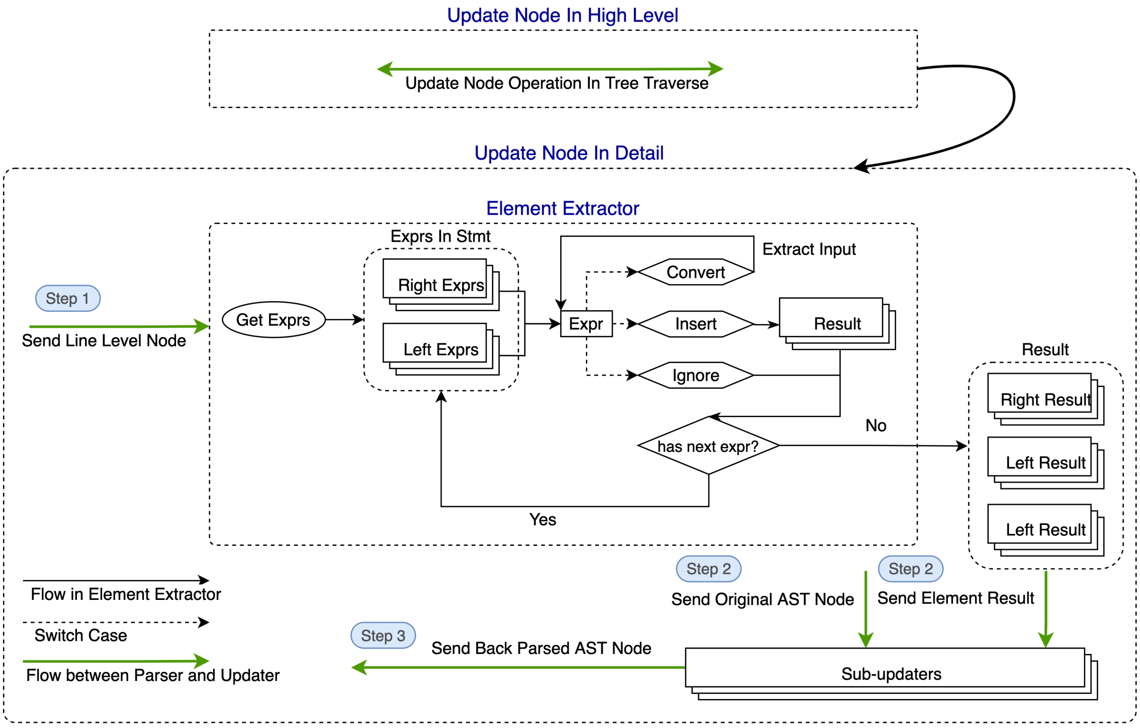


Figure 4.6: AST parser element extractor.

extractor iterate and processes all expressions in input, extract expression to get variables and return. The element extractor has three returns, shown in the following list:

1. Left side traverses return. It contains results after processing the left side of the line statement split by token if the statement is that expression statement and assignment statement, or the return is empty for other statement types.

2. Right side traverses return. It contains results after processing the right side of the line statement split by token if the statement is that expression statement and assignment statement, or the return is the result after processing the whole statement for other statement types.

3. All combined return. It combines the previous two previous return results.

The result format also needs to take into consideration. The runtime detectors need to interact with variables and properties, so the core elements in the result are all variables and properties in the code line. Can we simply put them into a set or array and return the result? The answer is no because it has the potential to make the detector raise complete failure. For instance, in List 4.2, the detector want to check if the property `s.Mongo.Collection` corrupted. If the property `s.Mongo` is a pointer and points to nil due to data corruption or wrong program logic, the detector will raise a complete error. We need to return all its related parent property and variables in the return to help the detector validate a property's parent properties and variables. In this case, we use a tree to store the processed result, the tree uses the hierarchy method to store variables and property in lines, an example is shown in Figure 4.7.

Listing 4.2: Potential failure example.

```go
// Modified part of github.com/kzmain/hotel-reservation-platform/services/user/server.go
...
func (s *Server) mgoGetUser(user *user.Request) error {
   var err error
   s.Mongo.Collection, s.Mongo.Database := s.Mongo.DB("user-db").C("user"), s.Mongo.DB("user-db")
   Detector.Detect(s.Mongo.Collection) // Example Detector
   err = s.Mongo.Collection.Find(bson.M{"_id": user.Username}).One(&user)
    ...
}
...
```

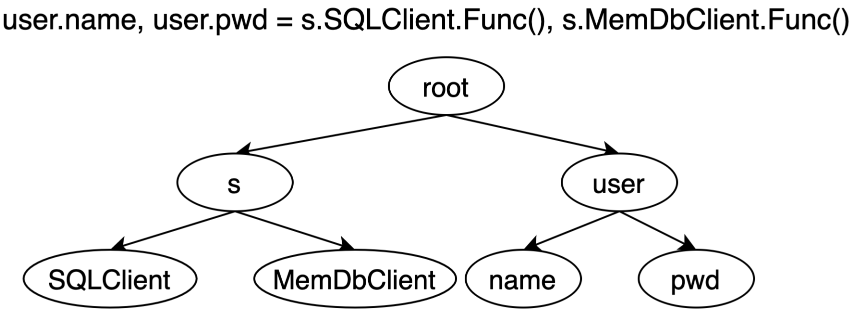

user.name, user.pwd = s.SQLClient.Func(), s.MemDbClient.Func()
root
s
user
SQLClient
MemDbClient
name
pwd

Figure 4.7: AST parser element extractor result tree.

This extractor's final part introduces how does the extractor get variables or properties from code expression. The element extractor iterate and processes all expressions from the input. The extractor doesn't always store expression directly in result trees, because not all expressions contents are variable or property. The extractor ignores some expressions types. As the Table 4.4 shows, the type like Function Type, Slice Type, Map Type ... are the expression types to represent the variable's type, which does not contain the variable, so the extractor ignores these expressions. The extractor converts some expression types to standard variable/property expressions. Call Function Type, Pointer Dereference Type are Pointer Reference Type, Binary belong to the convert condition. This is because the code expression contains variable expression and combined with token, the extractor needs to remove these token then store the converted expression in the result tree. For variable expression and property expression, the extractor directly stores them into the result trees.

Expression Type	Example	AST Class in Golang	Type
Variable	`varA`	`ast.Ident`	insert
Property	`varA.PropertyB`	`ast.SelectorExpr`	insert
Call Function	`varA.PropertyB.FuncC()`	`ast.CallExpr`	convert
Pointer (Dereference)	`var a = *b` (*b is ast.StarExpr)	`ast.StarExpr`	convert
Pointer (Reference)	`&a`	`ast.UnaryExpr`	convert
Binary Expression	`a == b`	`ast.BinaryExpr`	convert
Function Type	`a = func(){}` (func(){} is ast.FuncLit)	`ast.FuncLit`	ignore
Slice (array) Type	`[]int{}`	`ast.ArrayType`	ignore
Map Type	`map[KEY_TYPE]VALUE_TYPE`	`ast.MapType`	ignore

Table 4.4: All expression kinds table.

There is a special condition for the variables' extraction. It is variable ignorant in an assignment statement. Sometimes the call expression on the right side of the assignment token can return one or multiple instances and the developer doesn't need them to participate in the remaining logic, so the developer uses an Underscore to ignore them (e.g. `rst1, _ = a.returnTwoInstance()`). This is a condition that the programmer forgets to write an error handler but uses an Underscore to ignore the error. When the extractor encounter a variable whose name is "_", the extractor renames the underscore variable with random variable name start with "astUnderSocre" prefix (e.g. `rst1, astUnderSocreB = a.returnTwoInstance()`).

Insert Detector Import Declaration and Related Module

The detector-related sub-updaters only inject runtime detectors call expression statements to check failure in code. While the declaration of the detector and related module is not injected. In this condition, if the code file has a detector, the code compiler won't understand the code meaning.

After previous AST tree traversing, the AST File Parser examines if the sub-updaters inject the runtime detector call expression. If the parser gets the answer "Yes", the parser will examine what import does the code file lack because runtime detectors' required modules can have overlap with other code. Finally, the parser inserts needed imports and detector declaration.

MFD's Sub-updaters

In this section, we will illustrate each sub-updater usage in detail. We list 7 sub-updaters in section 4.2, and also know the sub-updaters modify code file in the core parser's file AST traverse process. But why do we need these updaters, what does each sub-updater do, and final output examples are vague, and we will explain these vague points.

Action Tracer & Vulnerable Action Protection sub-updater

Trace request action among services and localize failure is a challenge (as said in Chapter 3.1), in our implementation we used a static code analyzer to draw a network of internal services and also links to external services. The step is the same as shown in Figure 2.3, the difference is the function link in that method uses Java Instrumentation to extract function information and function links, this method uses a static file parser.

We also use a mechanism familiar to section 2.5 (Statistical Correlation Detector in Related work) to trace request action during runtime, which needs to insert related code by sub-updater. We discuss related sub-updater in the following paragraph.

ClientExtractorVulnerableOptProtector One important thing that needs to bear in mind is that internal/external service communication, I/O, etc are vulnerable operations. We need to use the timeout method or other mechanism to recover vulnerable operations when it is partial failure status.

Protect Vulnerable Functions The common partial failure can cause an infinite loop, stuck, so the idea is to set time out and error handlers to protect Vulnerable functions. `ClientExtractorVulnerableOptProtector` sub-updater uses rule matching method to detect I/O client, services client declaration and vulnerable operation execution. The rule matching method can be regex pattern matching, AST node structure matching, user tagging, or even use the Machine Learning method to match. Once the sub-updater find an

I/O client, a service client, or any other vulnerable operation client, the sub-updater will add a timeout setting to it if the client can set such attribute (example in List 4.3). If there's a vulnerable operation and the operation's client cannot set a timeout, this sub-updater will pack the function and implement the timeout for the function. (example in List 4.4, timeout 1 second). If the function does not return after the timeout, the packed function will raise `panic()` error to indicate failures.

Listing 4.3: Rate microservice example.

```
// Original Code Example
session, err := mgo.Dial(s.MongoAddr)
s.Mongo = session

// Converted Code Example
session, err := mgo.Dial(s.MongoAddr)
session.SetSocketTimeout(five)
session.SetSyncTimeout(five)
s.Mongo = session
```

Listing 4.4: Rate microservice example.s

```
func VulnerableFunc() Result {
...
}

// Original Code Example
var example Result = VulnerableFunc()

// Converted Code Example
var example Result
tmpVar := make(chan Result, 1)
go func() {
   tmpVar <- VulnerableFunc()
}()
select {
   case res := <-tmpVar:
      example = res
   case <-time.After(1 * time.Second):
      panic("VulnerableFunc() Timeout")
}
```

Draw Service Map DetClientExtractor is an updater to draw a service communication map. The microservice's definition says internal services communicate with each other by an interface. So it is easy to use the rule matching method to:

- Get service name if the code file is a service's code file.
- Get what function does current service have and how does internal function link to each other.
- Get what other services link to the service.
- Get connection between the service function with an external service function.

With so much information it would be easy to draw connections among functions and functions, services between services. We will illustrate it with code in the demo project, shown in Listing 4.5, and Figure 4.8. The example service in code Listing 4.5 uses gRpc as a communication interface.

First, we need to check if the code file belongs to a service. The gRpc service communication interface register pattern is `[SERVICE_NAME].Register[SERVICE_NAME]Server(,)`, which can indicate this is a service and the service's name. In the example, List 4.5 Line 51, the service's communication interface is established by `rate.RegisterRateServer(srv, s)`. The code means this is a service code and the service name is 'rate'.

Then during the iteration, we can get the function name at Func processing granularity, and we can log a list of all service internal functions (Figure 4.8 (b)). Furthermore, we need to classify what functions are internal functions and what functions communicate with other services.

Later in AST nodes iteration, we can get relations between services and store them in a tree. The internal function link establish is easy to understand, they can use pattern `RECEIVER_NAME.CALL_INTERNAL_FUNC()` to match , for example code in the List 4.5 Line 97, Line 154, Line 157, Line 159, Line 198 (Figure 4.8(c)). The link to other service's function has a special pattern: the communication argument. Service communication in Microservices Architecture uses a special argument to: 1) Share security principals and credentials. 2) Trace Local and distributed information. In gRpc the argument is `context.Context`. It can use `RECEIVER_NAME.OTHER_SERVICE_CLIENT_NAME.OTHER_SERVICE_FUNC(CONTE` `...)` to match, for example Line 85, we can know the function `GetRoomRatePlansByRoomIdsAndDatesAndR` connect to profile service client's `GetRoomProfileByRoomId` function (Figure 4.8(c)). The link to external/special service is a little complex, it requires the MFD user to set the matching pattern. For example, `s.MySQL.FUNCTION()` means current processing function links to MySQL storage service, `COLLECTION.Find()` means current processing function links to Mongo storage service,`RECEIVER_NAME.Memcache.Get()` means current processing function links to Memory Cache storage service. for example code in the List 4.5 Line 135, Line 171, Line 181 (Figure 4.8(c)'s green part).

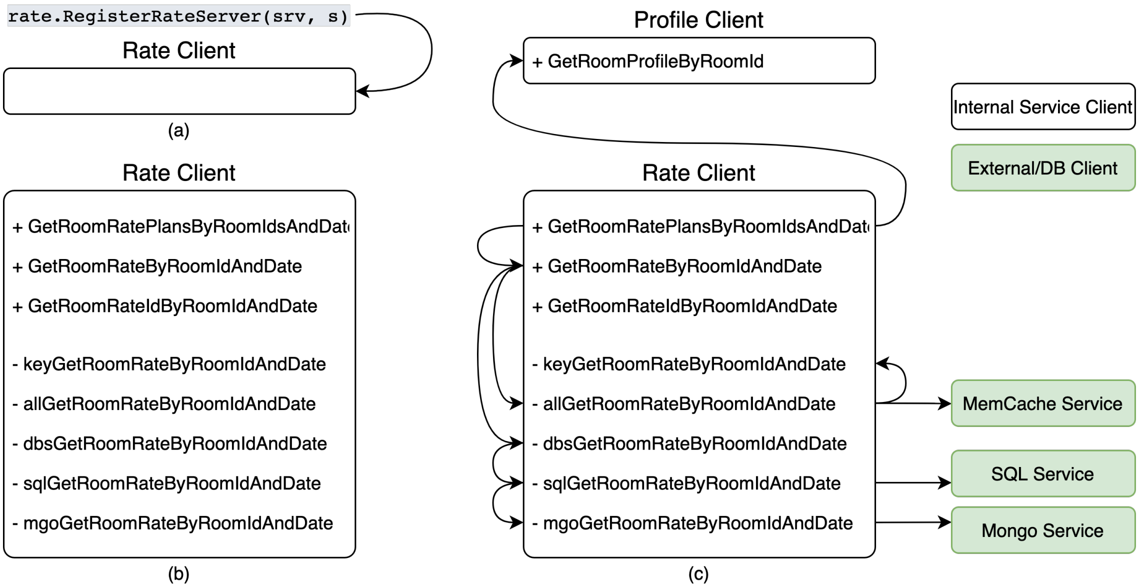

rate.RegisterRateServer(srv, s)
Rate Client
(a)
Profile Client
+ GetRoomProfileByRoomId
Internal Service Client
External/DB Client
Rate Client
+ GetRoomRatePlansByRoomIdsAndDat
+ GetRoomRateByRoomIdAndDate
+ GetRoomRateIdByRoomIdAndDate
- keyGetRoomRateByRoomIdAndDate
- allGetRoomRateByRoomIdAndDate
- dbsGetRoomRateByRoomIdAndDate
- sqlGetRoomRateByRoomIdAndDate
- mgoGetRoomRateByRoomIdAndDate
(b)
Rate Client
+ GetRoomRatePlansByRoomIdsAndDat
+ GetRoomRateByRoomIdAndDate
+ GetRoomRateIdByRoomIdAndDate
- keyGetRoomRateByRoomIdAndDate
- allGetRoomRateByRoomIdAndDate
- dbsGetRoomRateByRoomIdAndDate
- sqlGetRoomRateByRoomIdAndDate
- mgoGetRoomRateByRoomIdAndDate
(c)
MemCache Service
SQL Service
Mongo Service

Figure 4.8: AST parser's client extractor sub-updater.

After all service code files are processed, this updater connects all service relation tree to form the service connect network (like related work in Figure 2.3).

Listing 4.5: Rate microservice example.

```
1  // Cite from
       github.com/kzmain/hotel-reservation-platform/services/rate/server.go
   ...
50 func (s *Server) RegGrpcServer(srv *grpc.Server) {
51    rate.RegisterRateServer(srv, s)
52 }
53
54 func (s *Server) InitClients() error {
55    profileClient, err := s.Server.InitProfileClient()
56    s.profileClient = profileClient
      ...
61 }
62
63 func (s *Server) GetRoomRatePlansByRoomIdsAndDatesAndRoomNumber(ctx
       context.Context, ...) (..., ...) {
        ...
        roomProfile, err := s.profileClient.GetRoomProfileByRoomId(ctx,...)
        ...
        dailyRate, err = s.GetRoomRateByRoomIdAndDate(ctx, ...)
        ...
}

func (s *Server) GetRoomRateByRoomIdAndDate(ctx context.Context, ...) (..., ...){
    ...
}

func (s *Server) keyGetRoomRateByRoomIdAndDate(roomId string, roomDate string) string {
    ...
}

func (s *Server) allGetRoomRateByRoomIdAndDate(...) error{
        ...
        item, err = s.Memcache.Get(..., ...)
        ...
}

func (s *Server) dbsGetRoomRateByRoomIdAndDate(...) error{
        ...
        err = s.sqlGetRoomRateByRoomIdAndDate(dateRate)
        ...
        err = s.mgoGetRoomRateByRoomIdAndDate(dateRate)
        ...
        err = s.sqlGetRoomRateByRoomIdAndDate(dateRate)
        ...
}

func (s *Server) sqlGetRoomRateByRoomIdAndDate(...) error {
    ...
    err := s.MySQL.Get(dateRate, query)
    ...
}

func (s *Server) mgoGetRoomRateByRoomIdAndDate(...) error {
    ...
    c := s.Mongo.DB("rate-db").C("rate-spec")
    count, err := c.Find(...).Count()
    ...
}

func (s *Server) GetRoomRateIdByRoomIdAndDate(..., ...) (..., ...){
   ...
   err := s.allGetRoomRateByRoomIdAndDate(&dailyRate)
   ...
}
```

ServiceInternalMessageUpdater This updater only processes the service function implements a microservice communication interface to help trace actions of incoming requests. In the previous sub-updater's illustration, service communication uses a special argument to 1) Share security principles and credentials. 2) Trace Local and distributed information. In gRpc the argument is `context.Context`. During runtime, we can add requestID, check data corruption request variable into context to trace action, and log start time at the start of function and end time before the return statement. The example parsed code can be seen in the List. 4.6.

Listing 4.6: Update conxtex example.

```
func (s *Server) GetRoomRateByRoomIdAndDate(ctx context.Context, req ...)
    (..., error){
   ctx, s.TracerTimer = trace.S(ctx, "clientName","CalledFunctionName")
   defer s.TracerTimer.E()
}
```

This is familiar to section 2.5 Microservice Statistical Correlation Detector in Related Work. The two methodologies have one main difference. This method draws the service communication network before runtime, the section 2.5 uses Java Instrumentation during runtime to draw the network during runtime. In our implementation, we use a static code analyzer combined with a runtime tracer because this mechanism base on a logic that can easily migrate to different programming languages.

Common variable sub-updater

The common variable sub-updater does variable filtering, variable renaming, new variable declaration after the element extractor returns result trees. It is a basic sub-updater for any variable-related sub-updater. This means when an AST node wants to process a node, it passes line-level nodes to the element extractor and gets element results, as shown in

Figure 4.6, then all results will pass to these three common sub-updaters before sending to the detector manipulation sub-updater to filter the variable that needs to process, or add new variable declaration.

We need these three updaters because the detector or cooperating module cannot take some special type, variable with a special name, or need to insert some variable in implementation.

IgnoreStructUpdater This ignores the variables with type declare in the sub-updater's configuration. For example, if the user doesn't want to trace the variable data corruption of type, `int` the core updater will ignore related variables during the process. The MFD has this is because some enumerated types need to ignore in failure detector/injector implementation.

IgnoreIdentUpdater This ignores the variables and properties with names declare in the sub-updater's configuration. For example, if the user thinks "user.pwd" is a save property and adds it into the sub-updater's configuration, the updater will ignore "user.pwd" during the process. The MFD has this is because some configuration variables need to ignore in failure detector/injector implementation.

RestoreUnderscoreUpdater In section 4.2's Element Level Extractor discussion, the extractor convert Underscore variable to variable with random name start with "astUnderSocre" prefix.

This sub-updater inserts the random variable declarations' statement before the random variable code line if the other code sub-updaters want to keep the random variable name. If other sub-updaters do not want to keep the random variable name, this sub-updater will convert the variable's name back to underscore. All examples are shown in Listing 4.7. The MFD needs this one to simulate failure ignore and failure swallow in failure injection.

Listing 4.7: Restore underscore updater.

```
// Code before Element Level Extractor
rst1, _ = a.returnTwoInstance()

// Code After Element Level Extractor
rst1, astUnderSocreB = a.returnTwoInstance()

// RestoreUnderscoreUpdater convert variable back to Underscore
rst1, _ = a.returnTwoInstance()
```

```

// RestoreUnderscoreUpdater keep new random variable
var astUnderSocreB interface{}
rst1, astUnderSocreB = a.returnTwoInstance()
```

Runtime failure detector related sub-updater

The sub-updater of this section inserts runtime failure detector code and does static code analysis on the input AST node. For runtime detector, we only give code insert demonstration in this section, the mechanism will introduce in section 4.3.

InfiniteLoopDetectorUpdater This sub-updater inserts a runtime Infinite Loop Detector. The runtime detector's code has two parts. This insertion first part is an infinite loop detector's register, which indicates it has a loop here. The register uses continue condition's elements, code line position, random ID, action tracer, and timeout bound as parameters. The insertion second part is the detector to check if there has an infinite loop in runtime and recover the function from the infinite loop. The code example shows in the List. 4.8.

Listing 4.8: Insert infinite loop updater

```
// Original Code Block
for (_sDate.Before(_eDate)){

}

//Code with infinite loop detector
Detector.InfiniteLoopPrepare(token.Position{}, s.TracerTimer,
    "timeBound0.600000", _sDate, _eDate, "special_id")
for (_sDate.Before(_eDate)) {
    defer Detector.Recover()
   Detector.InfiniteLoopDetect("special_id")
}
```

Moreover, this sub-updater uses a mechanism familiar to section 2.3 Static Code Analyzer to check if the code has an infinite loop. The mechanism checks if the exit condition is always true (like boolean true, $1 > 0$). If the answer is true, the updater will inform the MFD user, else the sub-updater gets variables of loop exit condition. The sub-updater checks if the variable in the exit condition is updated in its loop code block. It's important to note that the mechanism does not analyze internal logic, related function or complex exit expression, so the mechanism can only detect a small part of infinite loops. So the static code analysis part only can detect always true, and the exit condition is not updated in the loop's condition. However, if the exit condition is updated by a special function without the exit condition's variable, the analyzer will give a false alarm.

DetPanicRecoverDetectorUpdater This sub-updater inserts runtime Panic Detector. The detector catches complete failure during runtime to detect logic-specific failure, the detector uses code position, tracer, and vulnerable indicator as parameters. The code example shows in List. 4.9.

Listing 4.9: Insert panic failure detector.

```
// Original Code Block
var rst = caller.VunFunc()

//Code with Panic Failure Detector
defer Detector.PandicDetector("Vulnerable VunFunc()", token.Position{},
    s.TracerTimer)
var rst = caller.VunFunc()
```

ValidatorUpdater This sub-updater inserts the *Validator Detector* into the code file. The detector has twos parts, the first part is before the code line to check if the code line's variables/properties have data corruption failure and nil pointer error. The second part is after the code line to check if the code line has error handle failure. The code example shows in the List. 4.10. The validator uses the if statement block to check the variable/property's value then check children's properties value, which looks like multiple trees.

Listing 4.10: Insert validator detector.

```
// Original Code Block
resUser, inPwd := &user.Request{Username: req.Username},
    s.getSHA256(req.Password)

//Code with variable validator
if !Detector.VDetect("req", token.Position{}, PreMode, req, s.tracer){
   if !Detector.VDetect("req.Username", token.Position{}, PreMode,
       req.Username){
   }
   if !Detector.VDetect("req.Password", token.Position{}, PreMode,
       req.Password){
    }
}
resUser, inPwd := &user.Request{Username: req.Username},
     s.getSHA256(req.Password)
if !Detector.VDetect("req", token.Position{}, PostMode, req, s.tracer){
    if !Detector.VDetect("req.Username", token.Position{}, PostMode,
         req.Username){
    }
    if !Detector.VDetect("req.Password", token.Position{}, PostMode,
         req.Password){
    }
}
```

4.3 Runtime Failure Detector Implementation

The runtime failure detector detects failures, logs failures, related context, traces failure, and recover service from failure. To ensure section 4.1 Low in coupling and High in cohesion, the failure detector also uses core detector cooperate many sub-detector to detect failure. The core detector stores the common setting for all sub-detectors and defines the interface of sub-detectors. The sub-detectors detect different kinds of failure, and they are called from the core detector during runtime. The structure example is shown in Figure 4.9. This structure is familiar to updater & sub-updater in section 4.2 AST File Parser. In future evolving, a developer can easily add or remove a sub-detector from the core detector. In the following sections we will discuss all detectors' usage and workflow, in Table 4.5 you can get an overview of all runtime detectors.

Detector Name	Illustration
Service Internal Message Tracer	Trace Action, Check Data Corruption in service communication
InfiniteLoopDetector	Detect Infinite Loop
Validator	Detect Swallow Error, Unhandeled Error, Data Corruption, Logic Error
PanicErrorDetector	Detect Complete Failure, Handle Failure Info From Other Detector

Table 4.5: Runtime failure detectors.

Side Effects For All Detectors

Because of the detector's code runs together with the original services code, mechanisms are required to avoid detector introduce failure into service. The first one is variable duplication, all input arguments are duplicated at the entrance of the detector. Secondly,

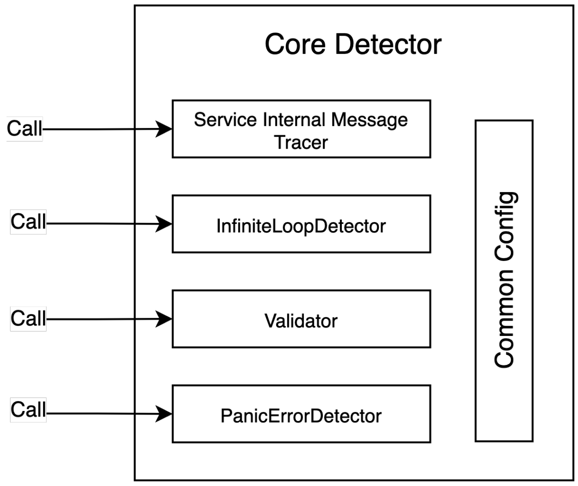


Figure 4.9: Core detectors and sub-detectors

all detectors have a recovery mechanism to ensure the detector's failure won't influence the service's running.

Core Detector

The core updater has a map to store all sub-detectors who implements the sub-detector interface, the runtime can call specific sub-detector by type and input argument.

The core updater has a "recovered" boolean attribute to indicate if the service is recovered from complete failure, and a "reason" string attribute to store the detected failure reason.

Service Internal Message Tracer & Communication Data Corruption Detector

The usage of this detector is to detect data corruption in service communication and to trace service actions of the incoming requests. This detector only works in the service function with communication interface. The graph illustration of this detector can be found in Figure 4.10.

The section 4.2 Runtime failure detector related sub-updater's ServiceInternalMessageUpdater insert a logic to duplicate request data into `context.Context` before the server send request.

If the detector has a validator variable in context, The detector will compare the incoming request variable and validate the validator variable in the service communication context, if the value is not the same the core updater sets the failure reason attribute as

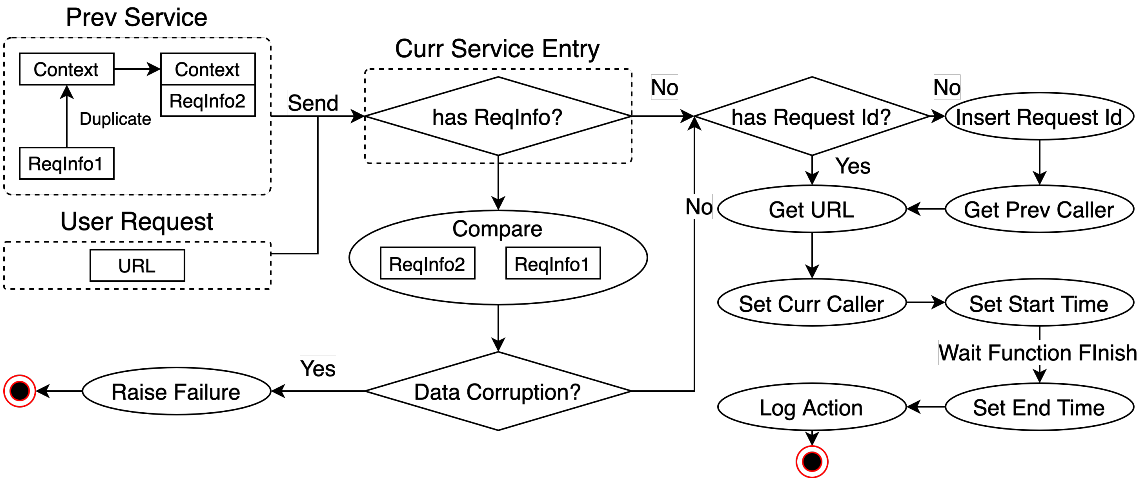


Figure 4.10: Internal message tracer & communication data corruption detector.

"ServiceDataCorruption" and raise except failure. The failure will be handled by section 4.3 Panic Failure Detector.

After data validation, the detector will set a request ID, if the incoming context does not have one. Then the detector gets the previous caller name, logs incoming time, incoming request-id, incoming request URL, current function name so that the detector can trace the action of the incoming request and localize failure. The time will be logged again after the service function return.

InfiniteLoopDetector

The usage of this detector is to detect infinite loop in runtime. Infinite loop is a common partial failure in code.

The loop detector does preparation before enter the loop code block. It starts a new timer to check the loop running duration. In the loop block, the loop detector checks the loop execution time every time at the start of the new execution circle. Once the execution duration reaches timeout bound (determined by input, default timeout is 0.2 seconds), the core updater sets the failure reason as "InifniteLoop" and raises complete failure. The failure will be handled by section 4.3.

Validator

The validator detector has two parts, "Pre" checker before a code line execution and "Post" checker after a code line execution.

The "Pre" part is to detect data corruption failure and register variables into a validator. The Figure 4.11 shows this part's workflow. If the input is an error, the check finish, else the detector goes to the next step. If the input variable/property cannot be found in detector registration, the detector will register the input into the detector (deep copy and store). If the input is in the detector, the detector will check if their value is the same. If the detector does detect failure, the core detector will set failure reason property to "InternalDataCorruptionError" and raise fatal failure.

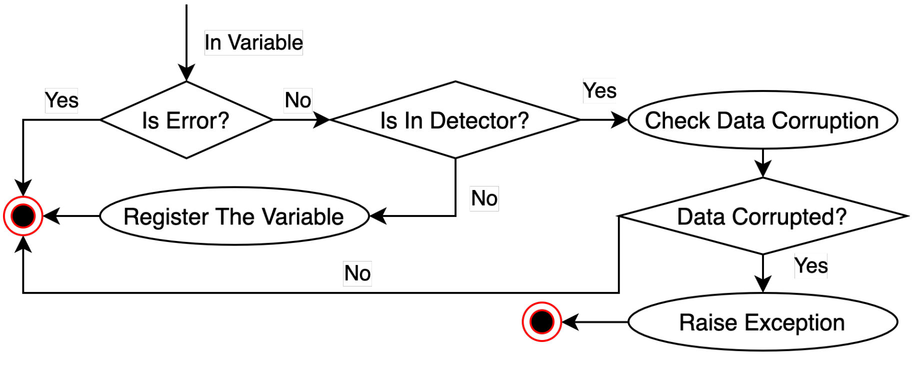


Figure 4.11: Validator pre part.

The "Post" part is to prepare for data corruption examine, deal with detect error swallow and error unhanded problem, which can be triggered by a communication interface overloaded, communication interface closed and vulnerable operation partial failure.

The Figure 4.12 shows workflow of this part. Post validator registers object instances and pointers (except for error type) in code line after execution. If the pointer is nil and not an error type, we regard the error happens in execution, because the function failed to set the value to the pointer. The detector will set the core updater "reason" property as "ExternalExecutionFailure" and raise complete failure. If the input is a non-empty error and the variable name starts with "astUnderSocre", which was "_" and updated by AST Parser's RestoreUnderscoreUpdater, the core updater will change the "reason" property as "VulnerableOptNotHandled" and raise complete failure.

DetPanicRecoverDetector

The "DetPanicRecoverDetector" is injected before every code line, it handles complete failure by logic problem and complete failure raised by other detectors. It logs, error code line, error reason, runtime variable value after the failure happens, and also the request-id & variable context to trace action failure. The sub-detector uses defer to log the previous

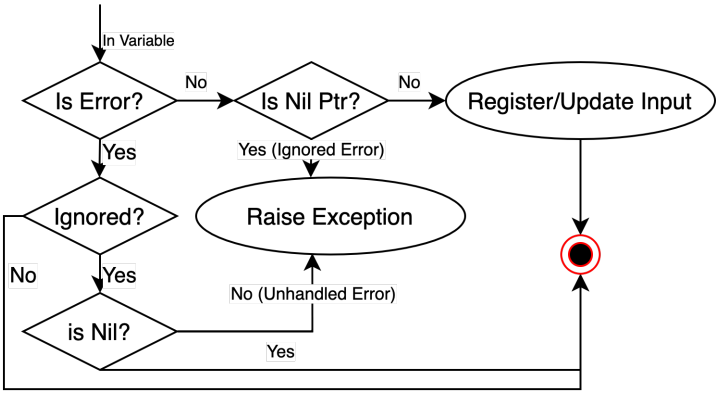


Figure 4.12: Validator post part.

context and use "recovered" as a log flag. When a complete failure is triggered, the defer function runs from the failure location to the start of a function. If the sub-detector detects the fatal failure, the recovered property of the core detector is set to true and logs failure reasons. When a deferred detector function recovered flag is true, the detector logs required history variable data.

5

Experimental Evaluation

We evaluated the MFD to know: Does the approach work well in microservice architecture? Can The detector detect and trace our target common failures in the real world? Does the MFD have false alarms? What extra-temporal and spatial does the MFD cost?

To get the answer we run controlled variable experiments on a popular open-sourced microservice research program to evaluate the efficacy of the MFD failure detector. We mimic anomalies in the evaluation program and diagnose the failures using our MFD failure detector.

In this section we will introduce section 5.1 Target system Information (Hardware Information and Metric We Collected), section 5.2 Simulation Application (Simulate Application in Real World), section 5.3 Failure Injection (How does failures inject), section 5.4 to check what Hardware and Time overhead does the MFD costs, section 5.5 to show evaluation results and discusses related reasons.

5.1 Target system Information

We perform our experiments on Vultr.com's dedicated server, with 320 GB SSD, 6 CPU, 16384 MB Memory, and 5000 GB Bandwidth. The evaluation-related runtime environment is The Docker Engine is v20.10.7., the Golang version is go1.16.3 darwin/amd64, and Ubuntu 20.10 x64.

All Running information is monitored by prometheus(32), grafana(33), cadvisor(34) and logged by promscale(35) in Postgres SQL database. They run different services at the same time with the simulation application. At every one second, they collect 88 metrics from every docker container, the most important two metrics are CPU usage, memory usage.

5.2 Simulation Application

We need to choose a suitable project to test our failure detector. Then we need to set some criteria to select a suitable demonstration project:

- Still maintained in the year 2021
- Available to get source code
- Available to deploy on the development machine and the testing machine
- Project set up by research group or some paper did research on the project
- The project is written by one programming language. (Though our detector bases on logic and can be extended to any language, however it needs much time to write different language version. We need to ensure it has high accuracy and efficiency in one language first)

We search some projects and analysis them, the result is shown in Table 5.1.

Name	Language	Type	Others
Train Ticket(36)	Multi-Language	From Research Group	High System Requirement
Sock Shop(37)	Java	From Commercial Company	Cannot Get All Source Code
Hotel Reservation(38)	Golang	From Research Group	Can start and well maintained

Table 5.1: High star microservice test platform.

In our evaluation, we choose the hotel reservation project from (38). The project's last update is on July 2021, and it is being set up by the SAIL group at Cornell University. The microservice is developed in Golang with the gRpc communication interface. However, it still has some shortages. For example, it has a data inconsistency problem, a service's database not only stores the data related to the service but also stores data related to other services. In this situation, one data update/change does not sync to all duplication, which causes a data inconsistency problem. At the same time, the data size is too small to make tests. We rewrite some service functions and wrote a MongoDB & SQLDB data seeder to reduce the data duplication and ensure referential integrity. Moreover, all the data is stored by MongoDB in the original version, we also add MySQL database into the simulation program which is closer to a real development environment. The framework can run in MongoDb mode, SQL mode, or Mix mode. The service network can be seen in Figure 5.1. The modified code is on GitHub (39). The core service code line is 11028 SLOCs.

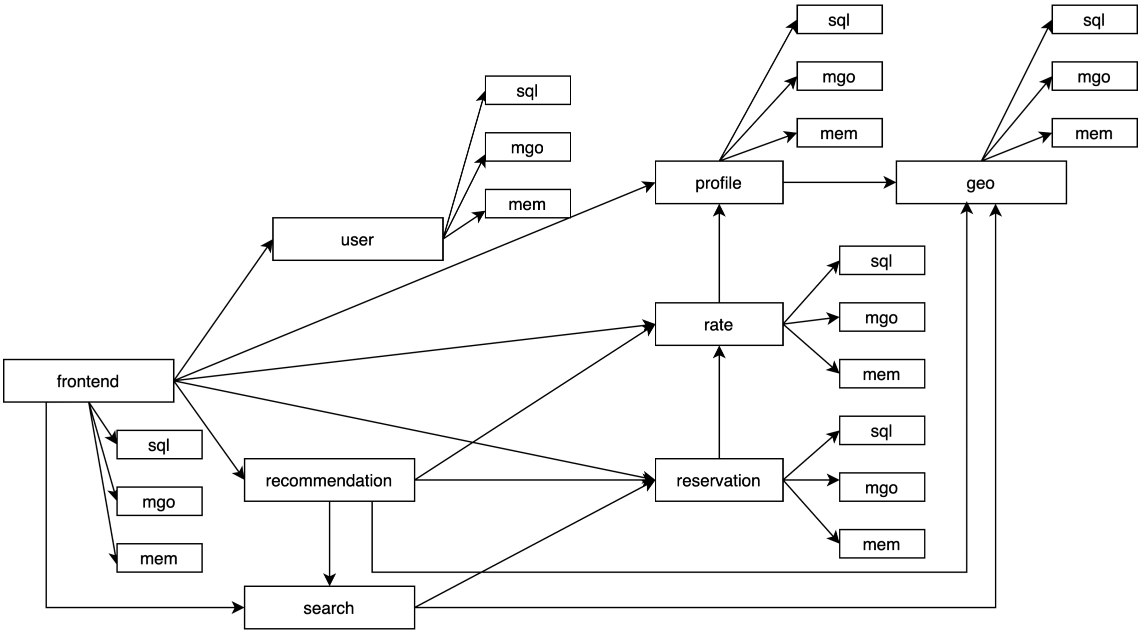


Figure 5.1: Evaluation service network.

5.3 Failure Injection

Failure Injection Technique

The failure injection includes two separate parts, one is internal failure injection, the other one is external failure injection. The failure injection methods are shown in Table 5.2.

	Main Outage	Sub Outage	Injector
Software	Error	Error Swallow	By code modification
		Error Unhanded	By code modification
	Hang	Infinite loop	By runtime injector
	Bugs	Data Corruption	By runtime injector
		Logic-Specific	By runtime injector
	Cross-Service Dependencies	Communication Corruption	By runtime injector
		Communication Interface Interface Overloaded	By `strace` system injector
		Communication Interface Closed	By `strace` system injector

Table 5.2: Failure and inject methods.

The internal failure injection, which means happens inside a service, includes simulating error ignorant, simulating error swallow, simulating data corruption, simulating logic error, and simulating infinite loop. These We use failure injection sub-updaters combine with section 4.2 AST File Parser to inject related failure. In error swallow mode, the failure injector removes return error assignment to the normal call statement, like shown in List. 5.1. In error ignore mode, the failure injector changes the error variable name in the assignment statement to underscore, like shown in List. 5.1.

Listing 5.1: Insert error unhanded/swallow error.

```
// Simulate time out
var err error = varA.VunFunc()

// Error ignore mode
_ := varA.VunFunc()

// Error swallow mode
varA.VunFunc()
```

The data corruption and logic error use one failure injector, the injector corrupts the object to a default value to simulate data corruption and changes the pointer to nil pointer to simulate the logic error. Furthermore, this injector has data restore operation, which changes variable data back to the correct value to avoid side effects in the continued test. The infinite loop detector packs the loop exit condition into a function. In the runtime, before the code enters the loop block, the packed function will define the continued condition is always true or use the exit condition of the original code.

The external failure injection, which means happens between two services, includes simulating network congestion, network interruption. We use `strace`(40) to simulate function timeout and network conjunction. The injected command is shown in Listing 5.2

Listing 5.2: External error failure injection.

```
// Simulate time out
strace -f -e trace=network -e fault=network:when=1+20:error=ETIMEDOUT -p 1
    -tt -o ./fault.log

// Simulate network conjunction
strace -f -e trace=network -e fault=network:when=1+20:error=EIO -p 1 -tt -o
    ./fault.log
```

Failure Injection Plan

To evaluate the MFD failure detector, we need to have different experiment groups to do a comparison. In our experiment, we have two basic influences we need to take into consideration, service requests' workload and code versions.

The First one is the workload send to our detector. We need to simulate a real situation and think about a suitable workload in the evaluation. This is because the service cannot work in an ideal world that users send requests one after another. Suitable workload enables tester can check real request delay (for example some request can send to the service very early, but due to network congestion in the request can be returned later than other requests), real hardware overhead and how does service perform. In our plan, the workload was to send 100 requests/second and last for 10 seconds to the evaluation software. However, when we tried to conduct the test, we found `wrk` is not available to do that due to some request response is time is long. We change the plan to 50 requests/second and last for 20 seconds instead.

Then from our code version aspect, as we said in section 5.3 we need to simulate error swallow and error unhandled situation. In the evaluation project we chose, all potential errors are handled. Moreover, we cannot say we remove which error handler to simulate error swallow/unhandled because it is not trustworthy. So we remove all error handlers, or swallow all errors (except for the error controls work logic/flow) and do failure injection on new versions of code. The test plan we can check from Table 5.3.

Group Name =>	Original Version	Detector Version	Error Unhandled Version	Error Swallow Version
With Failure Detector	No	Yes	Yes	Yes
Internal Failure Injector	No	No	Yes	Yes
External Failure Injector	10 Overload 10 Closed	10 Overload 10 Closed	10 Overload 10 Closed	10 Overload 10 Closed
Request Frequency	50 Req/s	50 Req/s	50 Req/s	50 Req/s
Work Load Last For	20s	20s	20s	20s
Group Repeat Time	50	50	50	50

Table 5.3: Real situation experiment plan.

In the simulation application, there are six different requests shows in Listing 5.3. We will run them in experiments on the frequency, as the experiment plan table show. Each of the requests will send to the simulation software 1000 times in 20 seconds, then repeats 50 times. So each request will run 50K times in the experiment. The relation between request and service is shown in Table 5.4. To simulate the workload we use `wrk2`'s command which is shown in Listing 5.4. The command means the tool `wrk2` runs a benchmark for 20 seconds, using 2 threads, keeping 10 HTTP connections open, and a constant throughput of 50 requests per second.

Name	Related Service
Request 1	fnt_service, geo_service, rsv_service, pfl_service, rte_service
Request 2	fnt_service, usr_service
Request 3	fnt_service, rec_service, pfl_service, geo_service
Request 4	fnt_service, rec_service, pfl_service, geo_service
Request 5	fnt_service, rec_service, pfl_service, geo_service
Request 6	fnt_service, rsv_service, usr_service

Table 5.4: Request and service map.

Listing 5.3: Example requests in evaluation

```
Request 1 :
"http://localhost:5000/hotels?inDate=2015-08-09&outDate=2015-08-10&lat=37.7867&lon=-122.4112&r
Request 2 :
"http://localhost:5000/user?username=user0&password=password0"
Request 3 :
"http://localhost:5000/recommendations?require=dis&lat=37.7867&lon=-122.4112"
Request 4 :
"http://localhost:5000/recommendations?require=rate&lat=37.7867&lon=-122.4112"
Request 5 :
"http://localhost:5000/recommendations?require=price&lat=37.7867&lon=-122.4112"
Request 6 :
"http://localhost:5000/reservation?username=user0&password=password0&inDate=2015-04-09&outDate
```

Listing 5.4: Work load request command.

```
wrk -t2 -c10 -d20s -R50 --latency URL
```

For internal failure, the failure injector inserts one failure after the core injector is called 150 times, so the failure inject location and failure inject time are random in the evaluation. For the external plan, the `strace` injects a failure on step 100.

The failure detection success standard is important to classify correct detection and failure detection. The failure pass standard is the detector can detect the failure happen, can figure out failure type accurately, can trace the failure, can log related context, the service is not down or does not need to restart.

5.4 Performance and Overhead

In this section, we conduct experiments on static file parse speed, throughout overhead, memory & CPU extra usage. The summary is shown in Table 5.5.

Property	Result
Parsing Speed	1.28 seconds
Vulnerable Function Detection Coverage	97% (Missing 2)
CPU Overhead	36.26%
Memory Overhead	6.24%
Request Waiting Overhead	2.97% - 6.19%

Table 5.5: Performance and overhead overview.

Parsing Speed In our experiment there are 11028 SLOCs need to compile, and takes 1.28 seconds on average to generate detectors and injectors (Figure 5.2), which means around 10K SLOCs/second. The parse time changes according to the platform hardware abilities.

Vulnerable Function Detection The code parser used 5 special rules to detect vulnerable functions, and time out for infinite loop and vulnerable operation is set for 0.2 seconds. In the file processing stage, the MFD located 66 vulnerable operations, 10 for MySQL, 24 for MongoDb, 10 for Memory Database, 22 for internal service clients communication and IO. In the examination, The MFD covers 97% venerable operation (missed two). This may cause silent failure or partial failure during the runtime. The file parser encountered a problem when the parser tries to pack vulnerable operations in the timeout block instead of just setting the timeout attribute on the vulnerable operation. In the timeout implementation, the MFD needs to know the function's return type. In the evaluation project, there are four vulnerable operations that use assignment declaration, for example: `row := s.MySQL.QueryRow(query)`. In this condition, we have to declare the return type first and parse the project again.

CPU & Memory Overhead We measured CPU overhead for each services with 6 different requests, shows in Listing 5.3. The detailed result is shown in Figure5.3 5.4 5.5 5.6, due to Request 3 - 5 uses same services and functions in action trace we combined the results together. The Figure5.3 shows Request 1 different services' CPU usage and memory consumption. The Figure5.4 shows Request 2 related services' CPU usage and memory consumption. The Figure5.5 shows Request 3 - 5 different services' CPU usage

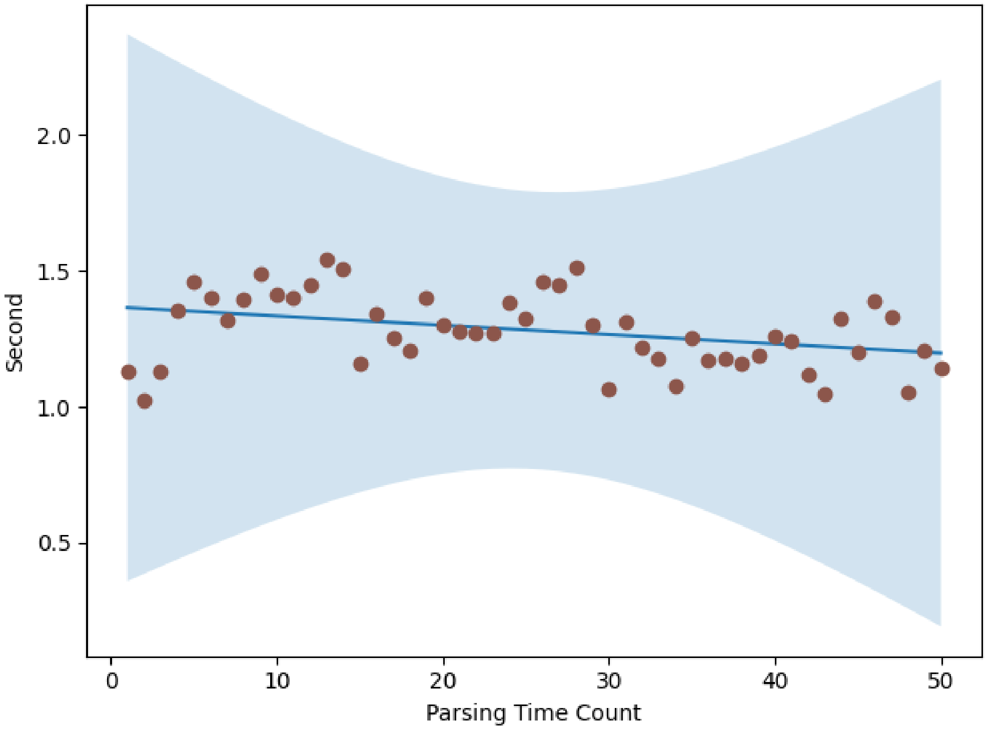


Figure 5.2: File parsing speed.

*x *axis*'s unit is times.

*y *axis*'s unit is seconds.

and memory consumption. The Figure5.6 shows Request 6 related services' CPU usage and memory consumption. Moreover, average summary is shown in Table 5.6.

From the table we can see the consumption of CPU usage surge a lot, most of them consume 30% more than original code, compares to (1)'s watchdog solution, which consumes only around 5% more CPU usage our work still has much work to do. This is because the MFD examines every line before and after statement execution, even a simple assignment like `var a = 1`, but watchdog puts the most focus on vulnerable operations and code blocks. Furthermore, fnt_service CPU usage surges more than any other service, because the service's calculation and statements need to execute are the most. This means the CPU usage raise percentage is linked to original service CPU usage. The more calculation of original service, the more CPU usage the MFD consumes.

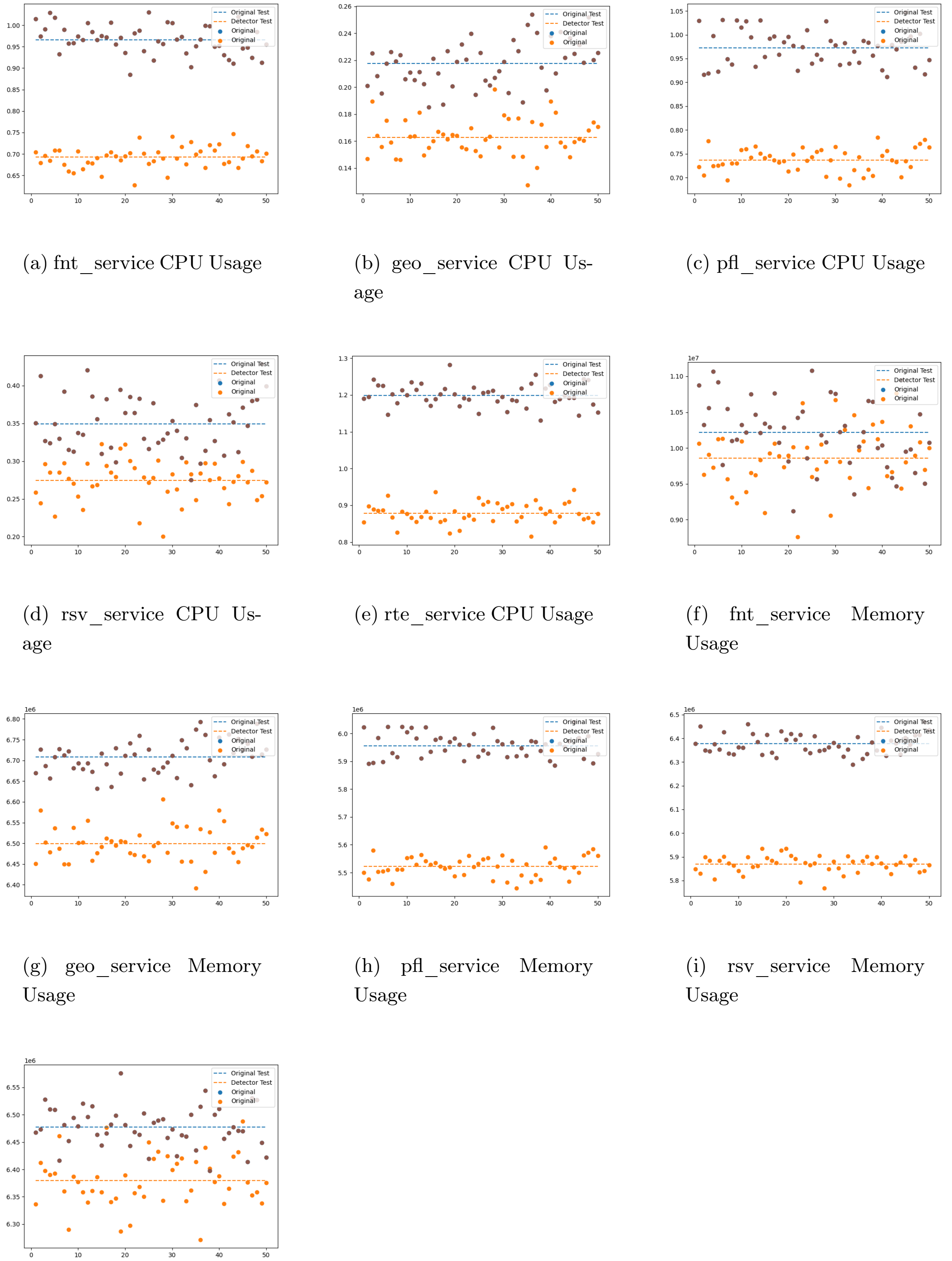
The MFD's memory consumption raise is around 6% more, which is not high compares to (1)'s watchdog around 4.2% more memory consumption rising. The fnt_service raises around 9% most among all services, no matter in which request. This is due to in implementation fnt_service stores all returned results from other services, and the MFD spends more space to store data even some data is no longer needed.

Request	Service Name	CPU Ori	CPU Det	Change	Mem Ori	Mem Det	Change
Request 1	fnt_service	0.696	0.966	38.71%	9.86	10.22	3.65%
	geo_service	0.163	0.218	33.64%	6.50	6.71	3.21%
	pfl_service	0.731	0.975	33.53%	5.52	5.96	7.86%
	rsv_service	0.274	0.349	27.41%	5.87	6.38	8.65%
	rte_service	0.878	1.199	36.52%	6.38	6.48	1.54%
Request 2	fnt_service	0.433	0.634	46.47%	8.22	9.02	9.83%
	usr_service	0.389	0.535	37.58%	7.42	8.33	12.27%
Request 3 - 5	fnt_service	0.050	0.075	50.32%	7.01	7.33	4.63%
	geo_service	0.252	0.341	34.95%	7.03	7.24	3.08%
	pfl_service	0.258	0.343	32.98%	6.07	6.54	7.83%
	rec_service	0.113	0.147	30.21%	6.89	7.50	8.75%
Request 6	fnt_service	0.310	0.443	42.89%	9.16	9.96	8.68%
	usr_service	0.228	0.302	32.58%	8.25	8.51	3.14%
	rsv_service	0.316	0.415	31.21%	8.87	9.25	4.27%
Average		0.364	0.496	36.26%	7.36	7.82	6.24%

Table 5.6: All request's CPU & memory usage.

**Ori* represents for original code without detector.

**Det* represents for code with detector



(a) fnt_service CPU Usage

(b) geo_service CPU Usage

(c) pfl_service CPU Usage

(d) rsv_service CPU Usage

(e) rte_service CPU Usage

(f) fnt_service Memory Usage

(g) geo_service Memory Usage

(h) pfl_service Memory Usage

(i) rsv_service Memory Usage

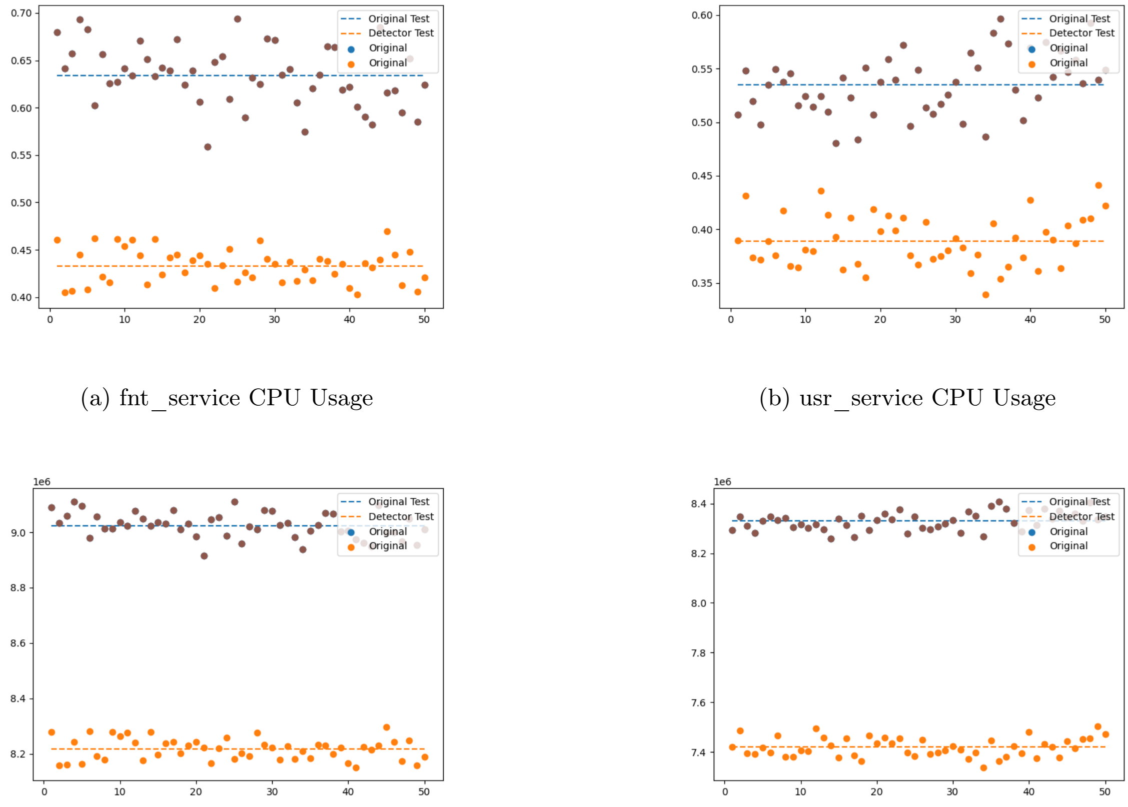
(j) rte_service Memory Usage

Figure 5.3: Request 1 search request CPU & memory usage.

**x axis*'s unit is times.

**y axis*'s unit for CPU Usage is Percentage divide by 100.

**y axis*'s unit for Memory Usage is Bytes.



(a) fnt_service CPU Usage

(b) usr_service CPU Usage

(c) fnt_service Memory Usage

(d) usr_service Memory Usage

Figure 5.4: Request 2 user request CPU & memory usage.

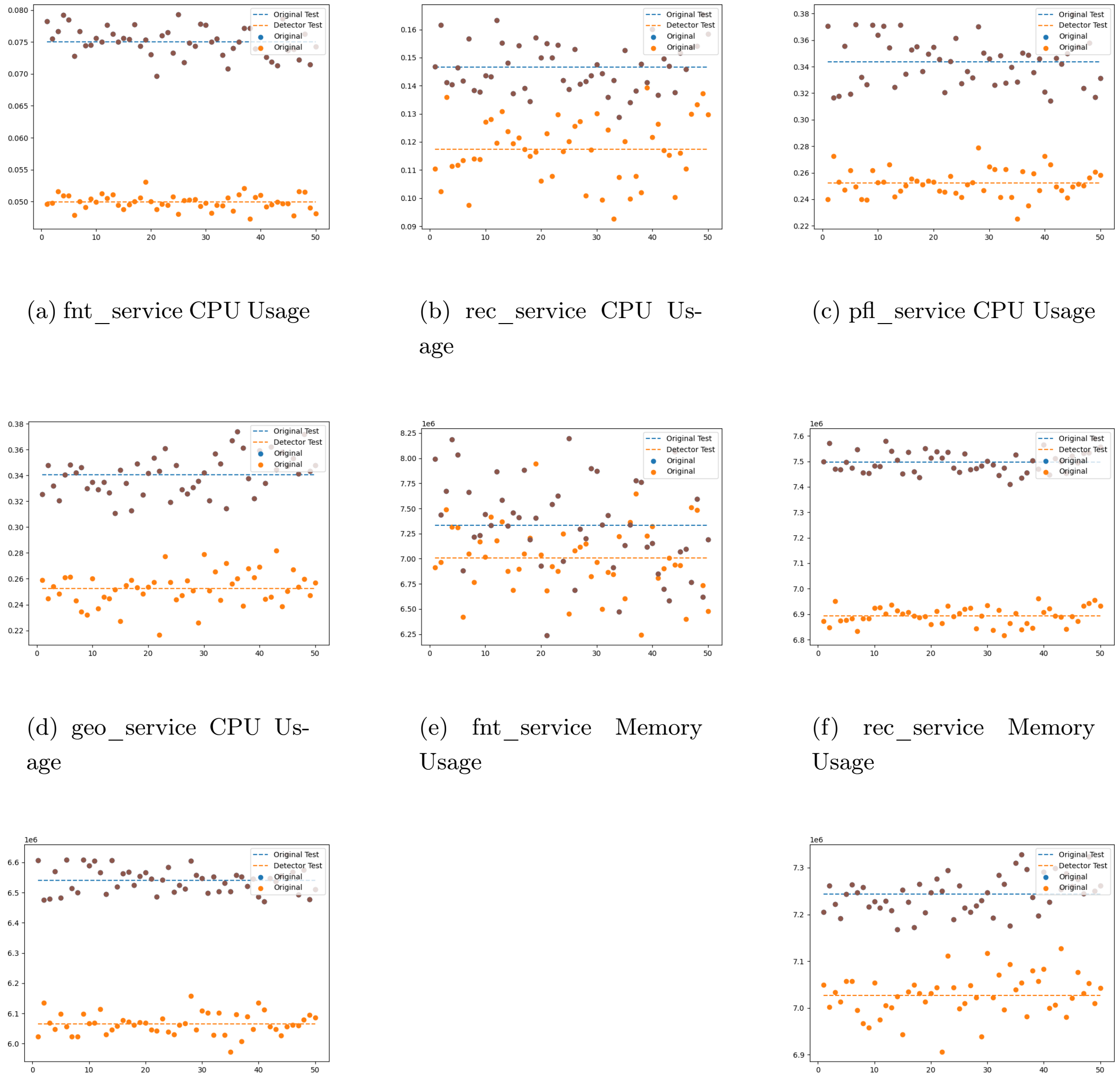
**x axis*'s unit is times.

**y axis*'s unit for CPU Usage is Percentage divide by 100.

**y axis*'s unit for Memory Usage is Bytes.

Request Waiting Time More calculation leads to more waiting time for users. We need to know how does the detector influence end-user. We summarized workload report into Table 5.8 from wrk2's result, and uses all data to draw the Figure5.7. The request waiting time shown in the table is the request's 50-time group test' average value. The Figure5.7 shows original code version and code with detectors version requests' 50%, 75%, 100%'s response time distribution and comparison. The blue part represents for original code and the cyan part represents for code with detector. Compare to hardware overhead, the waiting time increasing is more bearable. The minimum increase is 3% and the maximum waiting time increase is 6%.

From the table, we can see if a request related service has more calculation, the more



(a) fnt_service CPU Usage

(b) rec_service CPU Usage

(c) pfl_service CPU Usage

(d) geo_service CPU Usage

(e) fnt_service Memory Usage

(f) rec_service Memory Usage

(g) pfl_service Memory Usage

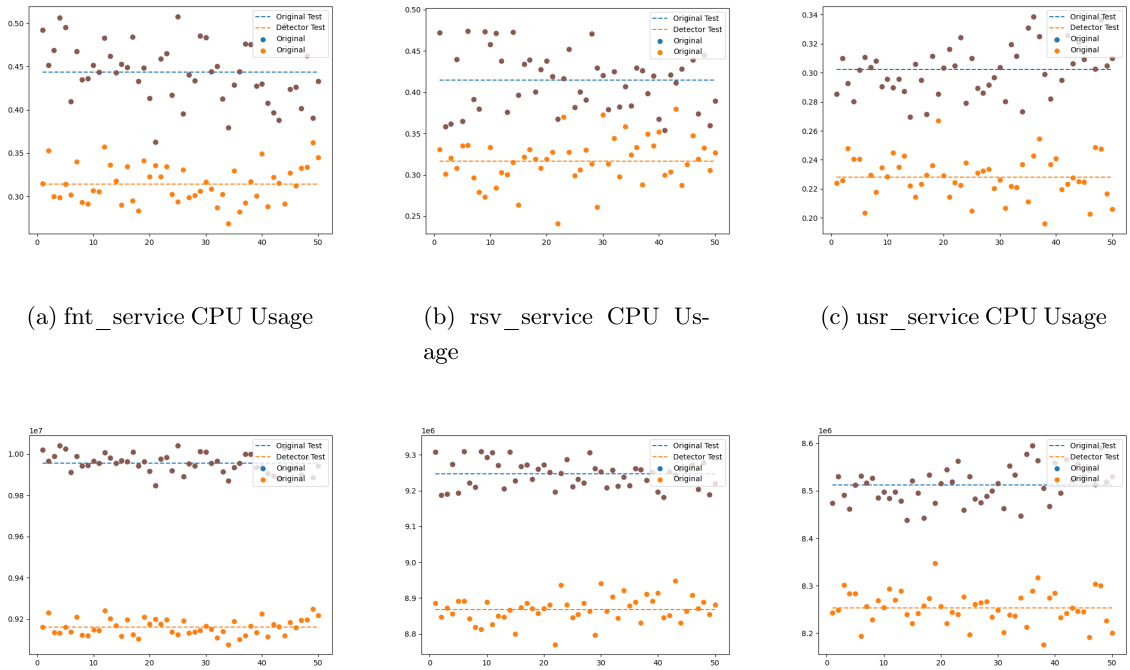
(h) geo_service Memory Usage

Figure 5.5: Request 3 - 5 recommendation request CPU & memory usage.

**x axis*'s unit is times.

**y axis*'s unit for CPU Usage is Percentage divide by 100.

**y axis*'s unit for Memory Usage is Bytes.



(a) fnt_service CPU Usage

(b) rsv_service CPU Usage

(c) usr_service CPU Usage

(d) fnt_service Memory Usage

(e) rsv_service Memory Usage

(f) usr_service Memory Usage

Figure 5.6: Request 6 CPU & memory usage.

**x axis*'s unit is times.

**y axis*'s unit for CPU Usage is Percentage divide by 100.

**y axis*'s unit for Memory Usage is Bytes.

waiting time percentage it increases. Moreover, no matter in which group, the 100%'s waiting for time's increasing is more than other response percentages (50% and75%). This means a stuck request needs to wait more time to get the resource to continue the calculation.

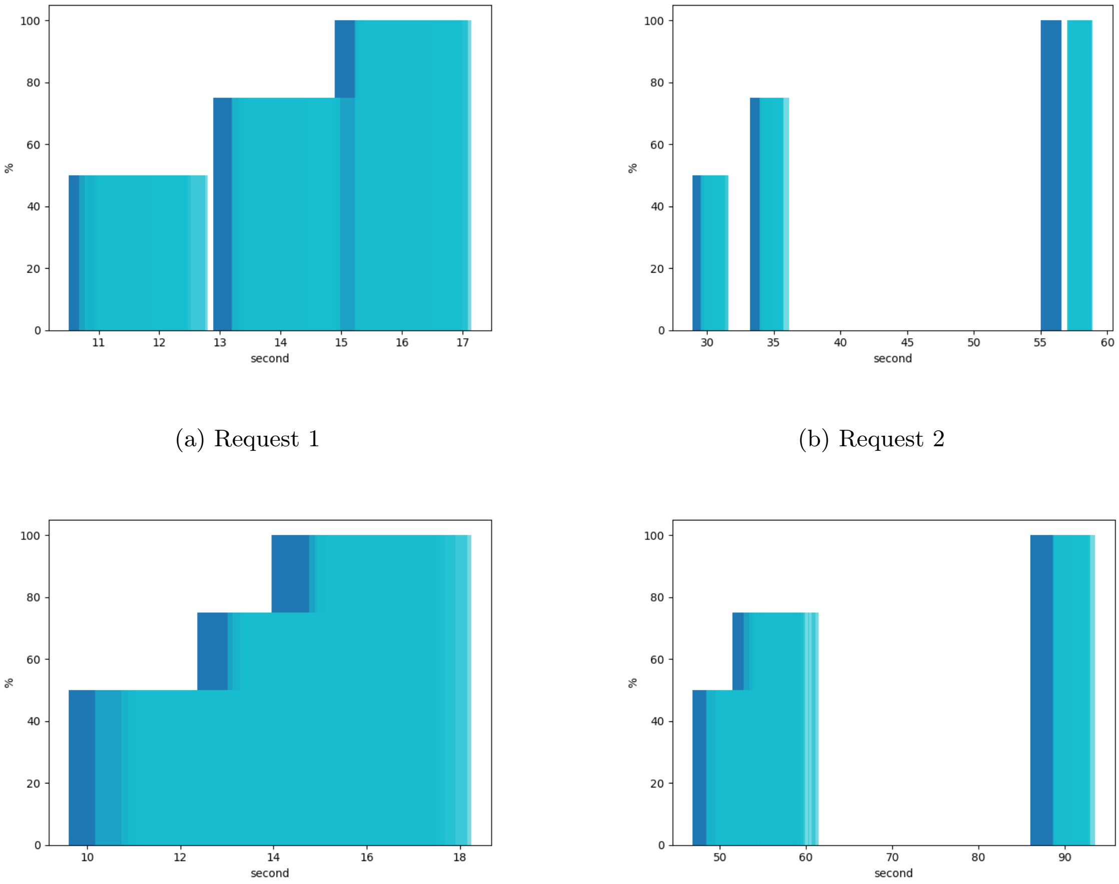
Table 5.7: Add caption

	Request 1	Request 2	Request 3 - 5	Request 6
50% Ori	11.421s	29.811ms	12.949s	51.524ms
50% Det	11.867s	30.69ms	13.467s	53.053ms
50% Raise	3.91%	2.95%	4.00%	2.97%
75% Ori	13.345s	33.77ms	13.312s	53.469ms
75% Det	13.912s	34.814ms	13.894s	55.103ms
75% Raise	4.25%	3.09%	4.37%	3.06%
100% Ori	15.966s	56.056ms	15.436s	89.8121ms
100% Det	16.867s	58.141ms	16.392s	94.3449ms
100% Raise	5.64%	3.72%	6.19%	5.05%

Table 5.8: Network response time.

**Ori* represents for original code without detector.

**Det* represents for code with detector



(a) Request 1

(b) Request 2

(c) Request 3 - 5

(d) Request 6

Figure 5.7: Network response result.

**x axis*'s unit is second.

**y axis*'s unit is percentage.

5.5 Runtime Failure Detection

In our experiment, there are three versions of source code, original code with own handler version, error unhandled version, error swallow version. In each version we send 6 requests 50K times, the failure injection and failure detection detail is shown in Table. 5.9, 5.10, due to Request 3 - 5 uses same services and functions in action trace we combined the results together. In this section, we split the error detection results into two parts because Infinite Loop and External failure are both related to the vulnerable operation and both use timeout to detect failure.

Group	Service	ZeroValue		NilPtr		Precision
		Inserted	Detected	Inserted	Detected	
Request 1 Original	fnt_service	37	36	21	18	93.10%
	geo_service	18	17	12	10	90.00%
	pfl_service	37	35	27	24	92.19%
	rsv_service	26	24	35	30	88.52%
	rte_service	27	26	23	10	91.20%
Request 1 Error Unhandled	fnt_service	36	35	18	15	92.59%
	geo_service	23	21	19	17	90.48%
	pfl_service	32	32	11	9	95.35%
	rsv_service	21	21	33	27	88.89%
	rte_service	29	28	19	16	91.67%
Request Error Swallow	fnt_service	35	34	17	12	88.46%
	geo_service	19	19	13	8	84.38%
	pfl_service	31	29	21	18	90.38%
	rsv_service	24	24	31	26	90.91%
	rte_service	27	26	28	24	90.91%
Request 2 Original	fnt_service	97	93	24	20	93.39%
	usr_service	83	80	91	84	94.25%
Request 2 Error Unhandled	fnt_service	111	104	27	23	92.03%
	usr_service	73	70	103	92	92.05%
Request 2 Error Swallow	fnt_service	94	89	29	21	89.43%
	usr_service	78	72	96	82	88.51%
Request 3 - 5 Original	fnt_service	103	100	78	71	94.48%
	geo_service	85	79	89	76	89.08%
	pfl_service	103	98	111	101	92.99%
	rec_service	79	73	82	72	90.06%
Request 3 - 5 Error Unhandled	fnt_service	100	95	77	68	92.09%
	geo_service	87	81	83	67	87.06%
	pfl_service	105	102	103	94	94.23%
	rec_service	81	78	88	73	89.35%
Request 3 - 5 Error Swallow	fnt_service	101	96	78	64	89.39%
	geo_service	84	81	87	78	92.98%
	pfl_service	107	102	86	72	90.16%
	rec_service	76	73	90	74	88.55%
Request 6 Original	fnt_service	76	71	82	75	92.41%
	usr_service	83	78	91	77	89.08%
	rsv_service	64	60	72	68	94.12%
Request 6 Error Unhandled	fnt_service	77	75	85	75	92.59%
	usr_service	87	85	66	54	90.85%
	rsv_service	62	57	78	72	92.14%
Request 6 Error Swallow	fnt_service	82	79	87	74	90.53%
	usr_service	84	81	64	53	90.54%
	rsv_service	66	61	89	82	92.26%
Total		2750	2620	2464	2135	91.20%

Table 5.9: Zero value and nil pointer failure injection and detection.

NilPtr & Zero Value Condition

Accuracy The nil pointer and zero value failure injection are to simulate data corruption, logic specific problem (one logic of code forgets to initial pointer property), and also evaluate the failure detector under error unhandled, error swallow condition. From the Table 5.9 we can see our detector gets around 91% accuracy under this failure condition. The unhandled version's accuracy is almost the same as the original version with handlers. This is because the detector use mechanism that forces the detector to handle ignored failure. The swallow version is lower than others, which means there's some silent failure happened in the system. Furthermore, Nil Pointer's failure detection accuracy is lower than normal variable object data corruption.

So why does the failure not detected? We trace back by injectors logged to file, it has three main reasons, corruption before return, error flag not clear and a silent failure continue (this one will discuss in False Alarm Paragraph)

We use an example to explain corruption before return. For example in Listing 5.5. After all, code finished in `funcB`, the failure injector injects data corruption, and there's no detector between the injector and return statement. Then the result returns to `funcA`. The detector after `funcA` only examines unhandled error or registers previous code line's variable, so in this condition, the detector regards this as a correct variable and updates the variable's data.

Listing 5.5: Work load request command.

```
func funcA(){
    funcB(&in)
    Detector.Register(in)
}

func funcB(in *RtType) RtType{
    ...
    Injector.InjectZeroValue(result)
    return result
}
```

Error flag not clear is also a common one that the detector ignores injected error. for example in Listing 5.6, the code request rows from database. Then scan the data to local variables' properties. If data in `row` corrupted, the detector will regard it as correct. At the same time, there's no error return to the scan function. Another situation is the developer

Group	Service	External Inject			Infinite Loop		
		Insert	Detected	Precision	Insert	Detected	Precision
Request 1 Original	fnt_service	7	7	100%			
	geo_service	9	9	100%			
	pfl_service	8	8	100%			
	rsv_service	11	11	100%	12	11	91.67%
	rte_service	13	13	100%			
Request 1 Error Unhandled	fnt_service	16	16	100%			
	geo_service	6	6	100%			
	pfl_service	9	9	100%			
	rsv_service	12	12	100%	11	10	90.91%
	rte_service	7	7	100%			
Request 1 Error Swallow	fnt_service	14	14	100%			
	geo_service	8	8	100%			
	pfl_service	4	4	100%			
	rsv_service	17	17	100%	11	10	90.91%
	rte_service	7	7	100%			
Request 2 Original	fnt_service	22	22	100%			
	usr_service	28	28	100%			
Request 2 Error Unhandled	fnt_service	23	23	100%			
	usr_service	27	27	100%			
Request 2 Error Swallow	fnt_service	24	24	100%			
	usr_service	26	26	100%			
Request 3 - 5 Original	fnt_service	63	63	100%			
	geo_service	71	71	100%			
	pfl_service	58	58	100%			
	rec_service	78	78	100%	13	13	100%
Request 3 - 5 Error Unhandled	fnt_service	63	63	100%			
	geo_service	72	72	100%			
	pfl_service	67	67	100%			
	rec_service	68	68	100%	14	14	100%
Request 3 - 5 Error Swallow	fnt_service	69	69	100%			
	geo_service	61	61	100%			
	pfl_service	67	67	100%	13	13	100%
	rec_service	63	63	100%			
Request 6 Original	fnt_service	17	17	100%			
	usr_service	18	18	100%			
	rsv_service	15	15	100%	14	13	92.85%
Request 6 Error Unhandled	fnt_service	13	13	100%			
	usr_service	16	16	100%			
	rsv_service	21	21	100%	14	14	100%
Request 6 Error Swallow	fnt_service	19	19	100%			
	usr_service	17	17	100%	15	14	93.33%
	rsv_service	14	14	100%			
Total		1248	1248	1	117	112	95.73%

Table 5.10: External network failure and infinite loop failure injection and detection.

uses non-error types, such as string, to indicate a potential error in the called function and the detector does not detect this situation.

Listing 5.6: Work load request command.

```
row := s.MySQL.QueryRow(fmt.Sprintf(sql.GetUsernamePasswordQuery,
    user.Username))
row.Scan(&resFound, &user.Username, &user.Password)
```

False Alarm From the Table 5.11 we classified can know false alarm is around 4.5%. The most common false alarm is false continue. A detector of a function detects a failure, then raises failure and returns an empty variable. The return does not contain any error, and an exception is recovered by the previous detector. The action goes back to the caller function, and the caller function regards the false return as a correct return. Then continue to execute the remaining lines until encounter a logic problem. If the failure does not encounter any logic problem, it will become a silent failure and return a false response to the end-user.

	Request 1	Request 2	Request 3 - 5	Request 6
False Alarm Ratio	4.67%	4.12%	4.71%	4.81%

Table 5.11: False alarm ratio.

Time Efficiency and Failure Localization Accuracy The time efficiency summarized in Table 5.12. We can know that the MFD's failure detection time efficiency is almost the same for the services operate on the same machine and docker configuration. However, the service needs more system resources (Request 1) needs a little more time to detect failure.

The Localization is around 78% because false continue happens in the system, it is due to false continue and undetected property's data corruption.

	Request 1	Request 2	Request 3 - 5	Request 6
Average Response Time (ms)	26.02	21.72	23.01	21.23
Failure Total Detect Time (Times)	680	830	1968	1277
Failure Localization Correct Time (Times)	492	653	1504	961
Failure Localization Correct Percentage	72.35%	78.67%	76.42%	75.25%

Table 5.12: Time efficiency and failure localization accuracy.

Infinite Loop Detector & Venerable Operation Detector

As shown in Table 5.10, these two detector are the most accurate detectors, both get nearly 100% accuracy. However, they have their own limitation that need to improve.

False Alarm Or Long timeout Timeout is an efficient way to break long-running or non-stop code blocks. However, how to set a suitable timeout may can use another paper to fully discuss that. In our evaluation, we set timeout as 0.2 seconds. Due to the strict timeout, many false alarm raises under current work load and timeout bound. The infinite loop false alarm ratio is shown in Table 5.13.

	Request 1	Request 2	Request 3 - 5	Request 6
False Alarm Ratio	22.91%	N/A	12.43%	13.47%

Table 5.13: Infinite loop false alarm ratio.

Why the false alarm is so high? We trace back to the runtime context and found 3 main reasons: some work needs long time calculation, some data cannot be got quickly from the database due to resource conflict, some iteration data is big, and the program needs time to process that. The code in Request 1 includes a for-loop in a for-loop. Besides, one loop includes an external I/O which slows down one loop circle's speed and raises false alarms frequently. Setting a small timeout can grade down system reliability, a long timeout will influence the detector's sensitivity.

Modification Manually Under circumstance venerable operation does not have timeout property. The parser will pack the venerable operation in a timeout block, however, the parser cannot get the external function's return type to inject a timeout code block if the programmer did not declare the return variable type in advance. Under this circumstance, the tester has to insert the type manually after reading the code file.

Side Effect

To avoid side effects, we duplicate the input variable before the failure detector's examination and the detector has its recovery for venerable operation. At the same time, we have a recovery function for service to ensure service reliability. The basic idea of the recover function is, the code tries to go to the next line. If there's an error, the detector will help the service step back to go to the previous "correct" configuration. During the test, this idea works most time, but make one service locked twice a time.

The recovery function injects failure to the Request 6's lock. For example in Listing 5.7. The lock should unlock the lock of service. However, when the failure happens, the `defer s.lock.Unlock()` unlocks the service lock, then the recovery function sets the lock's status to a deadlock.

Listing 5.7: Work load request command.

```
func (s *Server) MakeReservation(ctx context.Context, req *reservation.Request) (*reservation.Result, error) {
    ...
    defer Detector.Recover(s.lock)
   defer s.lock.Unlock()
   if err != nil {
      return res, err
   }
    ...
}
```

So in continuous development, we not only need to focus on vulnerable functions but also "dangerous variables" to ensure they exit vulnerable correctly.

6

Conclusions And Future Work

In this part, we will discuss what we have done, limitations, and future work. The software architecture is becoming more loosely coupled and complex. This leads to tracing failure and detecting a new emerging partial failure in Microservice Architecture Software (especially new failures in service communication). We first studied failure classification in cloud systems by scholarly paper and new types of failure from microservice-related authoritative websites (41). These all need a new mechanism to face new emerging difficulties. In our work, we introduced a new tracing method to trace and locate failures both before and in runtime. Moreover, we designed loosely coupled failure detectors and injectors to detect failures in microservice architecture software. At the same time, the detector includes a simple failure recovery mechanism to ensure the services' availability.

6.1 Conclusion

RQ1:What is the state-of-the-art in failure detection techniques?

We introduced 5 state-of-art failure detection algorithms after we conducted a literature survey. The first one is watch-dog, it inserts the detectors around vulnerable operations to detects partial failure. The second one uses variable duplication or variable checksum to detect data corruption. This methodology requires more memory space and calculation ability than any other resources. The third one is static code analyzer, it can detect infinite loop by analysis code without running. The fourth one (Machine Learning Detector) and final one (Statistical Correlation Detector) are similar to each other. They use math models or equations to predict failures. But the algorithm in Statistical Correlation Detector introduces a method to trace actions in SOA or microservice architecture. It uses the microservice's communication interface to action information among services.

RQ2: What properties of a microservice architecture make building a failure detector difficult?

In our findings shows, tracing & locating actions in the microservice architecture, solving cross-cutting concerns, designing a distributed failure detector are the call challenge in building a failure detector. Besides, new failures in microservice architecture's communication network and components is also a challenge.

RQ3: How to design a failure detector that addresses concerns specific to the microservice architecture?

In our implementation, our ideas are: 1) For portability, inject the code of the runtime failure detector into the original code. The detector's algorithm should not include any mechanism that is not available to implement in other coding language. 2) For scalability, the failure detector calls the standard interface when the program is running and uses different sub-detectors to check or verify failure. 3) For the high availability of microservice architecture, once a failure is found, report the failure, and use a simple program recovery mechanism to restore the program to the state before the failure occurred. 4) For error traceability, use microservice architecture's communication network to track the variable context when the error occurs and get context before the error occurs.

So in our implementation, there are two parts, File Parser and Runtime Failure Detector. They have a unified commonality that is high scalability, so they all use plug-in framework to ensure that sub-parsers or sub-detectors can be quickly added or removed. In File Parser, we use abstract tree parser to ensure that the code block and program structure of the added detector are correct. In Runtime Failure Detector we include: data corruption detector, partial failure detector, complete failure detector, infinite loop detector, failure tracker and failure recovery component. They run together on the core detector framework, and use share memory to communicate failure information.

RQ4: How to evaluate our failure detector on microservice architectures?

First of all, we need to select an evaluation platform. We have selected a microservice architectures research platform from Cornell University and, in order to simulate the network structure in the industry, we modified it and deployed it on a high-performance computer.

Secondly, we need to simulate failures on the evaluation platform. In order to simulate internal failures, we use abstract tree file parser to insert the failure injectors' code to simulate data corruption, partial failure, complete failure, infinite loop failure, logic problem, etc. Moreover, modify or delete the part of the code that returns the error to simulate the failure handling problem. In order to simulate the external environment to cause failure in the program, we use the `strace` to inject errors.

Finally, we conducted experimental evaluation. Take 6 different requests as six different groups, and each request runs 10,000 times at a speed of 50Req/sec to measure the overhead, accuracy and false accuracy rates respectively.

When we come to table about result. We get vulnerable operation coverage 97%, CPU Overhead 36.26%, Memory consumption overhead 6.24%, Request Waiting Overhead 2.97% - 6.19%. Moreover, We get around 90% failure detection accuracy, around 75% failure line-level failure localization accuracy, and 0.021 secs average failure detection time.

6.2 Future Work

In our current work, it still has limitations and more work in the future. First, the static file parser cannot detect all vulnerable operations when the MFD user is not familiar with the whole project and familiar with the related programming language, due to the detection pattern is input by the user. While in real life, a Microservice Architecture program is written by multiple languages and groups of programmer with different programming habits, all these influences' failure detection accuracy.

Secondly, the detector in the microservice communication is one-way communication, this makes the failure detect user does not know if the failure happens due to a network problem, or related service status problem or logic problem. Also, missing communication causes false alarms and false continue in the evaluation.

Thirdly, there are still has partial silent failures that cannot be found by the current mechanism, for example, data corrupted when a statement executes or a function's related function failed, but the function did not receive error information due to related function is recovered. Then the statement continues falsely. Finally, in our implementation, the detector does failure detection on every code line, which consumes time and uses massive hardware resources. A method to detect vulnerable operation and monitor dangerous variable can make the system works more efficiently.

References

[1] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. **Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU**. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 287–300, 2011. iii, v, 3, 9, 10, 11, 19, 60

[2] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. **DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems**. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 313–325, New York, NY, USA, 2018. Association for Computing Machinery. iii, v, 3, 9, 11, 12, 19

[3] Z.-Y Wang, T. Wang, W.-B Zhang, N.-J Chen, and C. Zuo. **Fault Diagnosis for Microservices with Execution Trace Monitoring**. *Ruan Jian Xue Bao/Journal of Software*, **28**:1435–1454, 06 2017. iii, 3, 13, 14, 15

[4] **Microservices vs. Monolithic Architectures**, Mar 2019. iii, 18

[5] Chang Lou, Peng Huang, and Scott Smith. **Understanding, Detecting and Localizing Partial Failures in Large System Software**. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 559–574. USENIX Association, 2020. v, 3, 9, 10, 19, 24

[6] Yan Yu and Haopeng Chen. **An Approach to Failure Prediction in Cluster by Self-updating Cause-and-Effect Graph**. In Dilma Da Silva, Qingyang Wang, and Liang-Jie Zhang, editors, *Cloud Computing – CLOUD 2019*, pages 114–129, Cham, 2019. Springer International Publishing. v, 3, 9, 13, 19

[7] Ana Gainaru andFranck Cappello andMarc Snir and William Kramer. **Fault prediction under the microscope: A closer look into HPC systems**. 2012. v, 3, 9, 13, 19

[8] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. **Anomaly Detection from System Tracing Data Using Multimodal Deep Learning**. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 179–186, 2019. v, 3, 9, 13, 19

[9] Zilong Zhao, Sophie Cerf, Robert Birke, Bogdan Robu, Sara Bouchenak, Sonia Ben Mokhtar, and Lydia Y Chen. **Robust Anomaly Detection on Unreliable Data**. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 630–637, 2019. v, 3, 9, 13, 19

[10] Kejiang Ye, Yangyang Liu, Guoyao Xu, and Cheng-Zhong Xu. **Fault Injection and Detection for Artificial Intelligence Applications in Container-Based Clouds**. In Min Luo and Liang-Jie Zhang, editors, *Cloud Computing – CLOUD 2018*, pages 112–127, Cham, 2018. Springer International Publishing. v, 3, 9, 13, 19

[11] A. Netti, Z. Kiziltan, Ö. Babaoglu, A. Sîrbu, Andrea Bartolini, and Andrea Borghesi. **Online Fault Classification in HPC Systems through Machine Learning**. In *Euro-Par*, 2019. v, 3, 9, 13, 19

[12] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. **Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning**. *IEEE Transactions on Parallel and Distributed Systems*, **30**(4):883–896, 2019. v, 3, 9, 13, 19

[13] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. **An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack**. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 124–131, 2019. v, 3, 9, 13, 19

[14] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. **Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs**. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 683–694, New York, NY, USA, 2019. Association for Computing Machinery. v, 3, 9, 13, 19

[15] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. **What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems**. page 1–14, 2014. v, 20, 21, 23, 24

[16] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. **Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages**. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery. v, 20, 21, 23

[17] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. **LossRadar: Fast Detection of Lost Packets in Data Center Networks**. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery. v, 20, 21

[18] Nane Kratzke and Peter-Christian Quint. **Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study**. *J. Syst. Softw.*, **126**:1–16, 2017. 1

[19] **What are microservices?** 1

[20] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. **Contextual Understanding of Microservice Architecture: Current and Future Directions**. *SIGAPP Appl. Comput. Rev.*, **17**(4):29–45, January 2018. 1

[21] Steve Swoyer Mike Loukides. **Microservices adoption in 2020**, Jul 2020. 1

[22] Kaya Ismail. **7 tech giants embracing microservices**, Aug 2018. 1

[23] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. **Quality-of-service in cloud computing: modeling techniques and their applications**. *Journal of Internet Services and Applications*, **5**(1):11, Sep 2014. 2, 3

[24] **Error Detection in Computer Networks**, Oct 2019. 3, 11

[25] **Error detection and correction**, Apr 2021. 3, 11

[26] Qing Xie. Qian Zhu, Teresa Tung. **Automatic Fault Diagnosis in Cloud Infrastructure**. 2013. 3, 13, 19

[27] Pooyan Jamshidi, C. Pahl, N. Mendonça, James Lewis, and Stefan Tilkov. **Microservices: The Journey So Far and Challenges Ahead**. *IEEE Softw.*, **35**:24–35, 2018. 17

[28] Peipei Wang, D. Dean, and Xiaohui Gu. **Understanding Real World Data Corruptions in Cloud Systems**. *2015 IEEE International Conference on Cloud Engineering*, pages 116–125, 2015. 19

[29] Mohammadreza Mesbahi, Amir Rahmani, and Mehdi Hosseinzadeh. **Reliability and high availability in cloud computing environments: a reference roadmap**. *Human-centric Computing and Information Sciences*, **8**, 12 2018. 23

[30] **Abstract syntax tree**, Apr 2021. 28

[31] **Tree traversal**, Aug 2021. 29

[32] Prometheus. **Prometheus - monitoring system; time series database**. 53

[33] **Grafana: The open OBSERVABILITY PLATFORM**. 53

[34] Google. **Google/Cadvisor: Analyzes resource usage and performance characteristics of running containers.** 53

[35] Timescale. **Timescale/Promscale: An OPEN-SOURCE analytical platform for Prometheus metrics**. 53

[36] FudanSELab. **FudanSELab/train-ticket: Train ticket - a Benchmark microservice system**, Aug 2021. 54

[37] Microservices-Demo. **Microservices-Demo/Microservices-Demo: Deployment scripts; config for sock shop**. 54

[38] Delimitrou. **Delimitrou/Deathstarbench: Open-source benchmark suite for cloud microservices**. 54

[39] KZMAIN. **Hotel Reservation Platform; Data Seeder; AFD Failure Detector/Injector**. 55

[40] FOSDEMTALKS. **Can strace make you fail? Strace syscall fault injection**, Mar 2018. 56

[41] **Microservices vs monolithic architecture**. 75