Non-Intrusive Causal Dependency Model-based Performance Anomaly Detection and Localization in Cloud Applications

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Senbo Fu

B.S., Information Science and Technology, Sun Yat-sen University, China
M.S., Electrical and Computer Engineering, Carnegie Mellon University, USA

Carnegie Mellon University
Pittsburgh, PA

December, 2019

Acknowledgements

It requires a lot of efforts and I am sincerely grateful for many people who have helped me a lot along the long Ph.D. journey. First, I would like to express my sincere gratitude to my advisor Prof. Hyong Kim for his continuous guidance and support on my Ph.D. work. He had taught me a lot of precious lessons such as how to organize thoughts and work systematically. Thank him for his patience and tolerance when I make mistakes. It is an unforgettable and valuable experience to be a Ph.D. student with him.

I would like to express my sincere gratitude to another advisor Prof. Rui Prior for his continuous support and encouragement. Whenever I need help in research or life, he always offers the most generous support as he can. I would not be able to finish my Ph.D. without his help in the most difficult moments. I will always remember the moments he helped me to apply for project funding during the last two years in Portugal. It is an amazing and unforgettable experience to work with him.

I would also like to thank my committee members: Prof. Anthony Rowe, Prof. Aswin C. Sankaranarayanan, and Prof. Paulo Carvalho for their comments and suggestions on my thesis and defense.

I would like to acknowledge the 5-year financial support by FCT (Fundação para a Ciência e a Tecnologia) through the Carnegie Mellon Portugal Program under Grant (SFRH/BD/51150/2010). I also appreciate the funding provided by Instituto de Telecomunicações (IT Porto) project NanoSTIMA A/BIM/2018-B00400, project UID/EEA/50008/2013, and project UID/EEA/50008/2019.

I would like to express gratitude to my excellent friends and colleagues: Chen Wang, Ke Wang, Shuang Su, Tiago Carvalho, Jiyeon Kim, John Payne, and Andrew Fox from theONE research group, Shaolong Liu, Zhuo Chen, Xuanle Ren, Jiyuan Zhang, Changting Xu and many other people for their friendship, kind discussion, productive comments, and help in daily work and life. Special thanks go to Shaun Stevens, Mark Blanco, Zhongli Zhang for their help in reviewing my thesis and improving its readability. I also owe my sincere thanks to all the colleagues from Instituto de Telecomunicações, Faculdade de Ciências da Universidade do Porto. Their presence makes my life in Porto easier and more interesting. I would also like to thank all staff in both CMU and Portugal side: Samantha Goldstein, Nathan Snizaski, Alexanda Ferreira, Sara Brandao, Ana Lopes for their help in the smooth transfer between Pittsburgh and Porto.

Finally, I would like to thank my grandparents, parents, and other family members for their endless love and support. I would like to thank my best friend and most beautiful girlfriend Ms. Qian Shi for her love, support, and accompany. This journey gives me more confidence and strength to take any challenge in the future.

Abstract

Infrastructure-as-a-Service (IaaS) Cloud is a popular platform for providing virtual computing and storage resource to millions of users all over the world. It allows many Cloud users to deploy their applications in a simple and cost-effective way. Cloud applications are usually deployed in virtual components such as virtual machines and containers. Each virtual component provides a specific function and they together provide services to customers. Due to the complex and dynamic nature, Cloud applications are prone to performance anomalies. Performance anomalies degrade the quality of experience for the users and may cause loss of revenue for service providers. Performance anomalies could propagate from one component to another through their interactions. A faulty component could cause abnormal behaviors in many other components. When there is an anomaly, it is important to detect it and locate the faulty component as quickly as possible. Virtual components owned by Cloud tenants do not provide visibility nor access to Cloud providers. Existing IaaS Cloud infrastructures usually monitor resource consumption and activity of each component. However, the resource utilization metrics do not reflect the actual service performance.

We propose decentralized methods for anomaly detection and localization in non-intrusive fashion. We detect performance anomalies and localize the faulty component in Cloud applications without any information about inner workings of virtual components. Our systems do not own these virtual components and treat them as black boxes. We monitor network traffic from each virtual component and its interaction with other components. The interaction behavior is not affected by fault propagation if all component involved in the local interaction are normal. This discovery helps us quickly filter out normal components. We classify these interactions into three different dependency primitives. We show that these dependency primitives help achieve better anomaly detection and localization in Cloud applications.

We propose DMADL (Dependency Model-based Anomaly Detection and Localization) to estimate the mean response time of each component using the arrival and departure pattern of data packets. DMADL achieves anomaly localization through the dependency model and component impact analysis. We also propose DMFDL (Dependency Model-based Flow ratio analysis for Anomaly Detection and Localization) and DMCDL (Dependency Model-based flow Correlation analysis for Anomaly Detection and Localiza-

tion) to model the relationship that the response flow always follow the request flow within an acceptable time limit for each component service. This relationship is true for varying workload conditions at any component of Cloud applications as long as it runs in normal operation.

We evaluate our methods in realistic deployment scenarios using the CloudSuite web search application, the Olio web application, and the MediaWiki application. The results show that DMADL achieves accurate response time estimation at each component. DMADL has around 95% precision and 5% false negative rate in both anomaly detection and localization under varying workload scenarios. DMFDL and DMCDL have, on average, 87% precision and 5% false negative rate in anomaly detection and localization. Compared to the anomaly detection methods based on resource utilization metrics, DMFDL and DMCDL achieve on average 18% higher precision, and 17% fewer false negatives. In anomaly localization, DMFDL and DMCDL achieve around 15% higher precision and 10% fewer false negatives than FChain, another black-box component-level fault localization method. FChain relies on the chronological changing order of components without considering the dependency model. We also evaluate DMADL, DMFDL, and DMCDL with extensive chronic faults. We show that our methods detect anomaly within 5 minutes for extensive chronic faults.

Contents

Contents vii

List of Tables xii

List of Figures xiii

1 Introduction 1

1.1 Challenges . . . 2

1.2 Contributions . . . 3

1.2.1 Dependency Model . . . 4

1.2.2 DMADL Method . . . 4

1.2.3 DMFDL Method . . . 5

1.2.4 DMCDL Method . . . 6

1.2.5 Non-Intrusiveness and Low Overhead . . . 6

1.2.6 Extensive Chronic Fault Evaluation . . . 7

1.3 Thesis Organization . . . 7

2 Related Works 8

2.1 Anomaly Detection . . . 8

2.1.1 Response Time-based Anomaly Detection . . . 8

2.1.2 Queuing Model-based Anomaly Detection . . . 9

2.1.3 Machine Learning Model-based Anomaly Detection . . . 10

2.1.4 Correlation-based Anomaly Detection . . . 12

2.2 Anomaly Localization . . . 13

2.2.1 Trace-based Anomaly Localization . . . 13

2.2.2 Application Component Dependency Analysis . . . 15

2.2.3 Machine Learning-based Anomaly Localization . . . 17

3 Fault Models **19**

3.1 Fault Model . . . 19

3.1.1 Resource Bottleneck Fault . . . 19

3.1.2 Common Software Fault . . . 20

3.1.3 Extensive Chronic Fault . . . 20

3.2 Fault Propagation . . . 22

4 Experimental Setup **25**

4.1 Application Benchmarks . . . 25

4.1.1 MediaWiki Application . . . 25

4.1.2 Olio Web Application . . . 26

4.1.3 CloudSuite Web Search Benchmark . . . 27

4.2 Workload Configuration . . . 27

4.3 Evaluation Criterion . . . 28

4.3.1 Anomaly Detection . . . 28

4.3.2 Anomaly Localization . . . 28

4.4 Experimental Setup for Comparative Study . . . 29

4.4.1 Anomaly Detection . . . 29

4.4.2 Anomaly Localization . . . 29

5 Dependency Model **31**

5.1 Dependency Model . . . 32

5.1.1 Composite Dependency . . . 32

5.1.2 Concurrent Dependency . . . 34

5.1.3 Distributed Dependency . . . 34

5.2 Dependency Extraction Analysis . . . 35

5.2.1 Distributed Dependency Extraction . . . 36

5.2.2 Composite or Concurrent Dependency Extraction . . . 37

5.3 Distributed View of Dependency . . . 38

5.3.1 Dependency Extraction Algorithm . . . 38

5.4 Experimental Evaluation . . . 39

5.4.1 MediaWiki Application . . . 40

5.4.2 Olio Web Application . . . 41
5.4.3 CloudSuite Web Search Application . . . 41
5.4.4 Distributed Dependency Extraction: Impact of Sampling Interval . . . 42
5.4.5 Composite and Concurrent Dependency Extraction: Impact of the Sampling Interval 43
5.5 Discussion . . . 44

6 DMADL: Dependency Model-based Response Time Analysis for Anomaly Detection and Localization 46

6.1 Mean Response Time Estimation . . . 46
6.1.1 Monitored Metrics . . . 47
6.1.2 Accuracy Analysis . . . 47
6.1.3 Theoretical Overhead Analysis . . . 49
6.2 Anomaly Detection . . . 50
6.3 Anomaly Localization . . . 50
6.3.1 No Dependent Component . . . 51
6.3.2 Single Dependent Component . . . 52
6.3.3 Multiple Dependent Components . . . 52
6.3.4 Subsystem Impact Analysis . . . 53
6.3.5 Component Impact Analysis . . . 54
6.4 Experimental Evaluation . . . 55
6.4.1 Mean Response Time Estimation . . . 55
6.4.2 Resource Fault Detection . . . 57
6.4.3 Common Software Fault Detection . . . 57
6.4.4 Anomaly Detection Performance . . . 60
6.4.5 Anomaly Localization Case Study . . . 63
6.4.6 Anomaly Localization Performance . . . 67
6.4.7 Overhead Analysis . . . 68
6.5 Summary . . . 70

7 DMFDL: Dependency Model-based Flow Ratio Analysis for Anomaly Detection and Localization 72

7.1 DMFDL overview . . . 73
7.2 Traffic Flow Monitor . . . 74
7.3 Anomaly Detection . . . 75

7.3.1 Flow Ratio Model . . . 75
7.3.2 Detection Algorithm . . . 78
7.4 Anomaly Localization . . . 79
7.4.1 Local Interaction Behavior Characterization . . . 79
7.4.2 No Dependent Component . . . 81
7.4.3 Single Dependent Component . . . 81
7.4.4 Distributed Dependency Primitive . . . 81
7.4.5 Concurrent Dependency Primitive . . . 82
7.4.6 Composite Dependency Primitive . . . 83
7.5 Experimental Evaluation . . . 84
7.5.1 Flow Ratio Stability Analysis . . . 85
7.5.2 Resource Fault Detection . . . 88
7.5.3 Common Software Fault Detection . . . 90
7.5.4 Anomaly Detection Performance . . . 92
7.5.5 Anomaly Localization Case Study . . . 93
7.5.6 Anomaly Localization Performance . . . 98
7.5.7 Overhead Analysis . . . 99
7.6 Summary . . . 100

8 DMCDL: Dependency Model-based Flow Correlation Analysis for Anomaly Detection and Localization 101

8.1 DMCDL overview . . . 102
8.2 Traffic Flow Monitor . . . 102
8.3 Anomaly Detection . . . 103
8.3.1 Traffic Flow Correlation . . . 103
8.3.2 Detection Algorithm . . . 105
8.4 Anomaly Localization . . . 107
8.4.1 Local Interaction Behavior Characterization . . . 107
8.4.2 No Dependent Component . . . 108
8.4.3 Single Dependent Component . . . 108
8.4.4 Distributed Dependency Primitive . . . 109
8.4.5 Composite Dependency Primitive . . . 110
8.4.6 Concurrent Dependency Primitive . . . 110

8.5 Experimental Evaluation . . . 111
8.5.1 Sampling Interval Selection . . . 111
8.5.2 Correlation Window Selection . . . 114
8.5.3 Resource Fault Detection . . . 116
8.5.4 Common Software Fault Detection . . . 119
8.5.5 Anomaly Detection Performance . . . 122
8.5.6 Anomaly Localization Case Study . . . 123
8.5.7 Anomaly Localization Performance . . . 129
8.5.8 Overhead Analysis . . . 130
8.6 Summary . . . 130

9 Performance Comparison **131**
9.1 Case Study of Extensive Chronic Faults . . . 131
9.1.1 Apache HTTP Server . . . 131
9.1.2 Nginx Application Server . . . 138
9.1.3 MySQL Database Server . . . 142
9.2 Overall Anomaly Detection Performance . . . 145
9.3 Overall Anomaly Localization Performance . . . 146
9.4 Discussion . . . 147

10 Conclusion **148**

Bibliography **151**

List of Tables

3.1 Fault Models, Fault Scenarios, Fault Injection Techniques and References 20
3.2 Software performance bugs occurred in different application platforms 21
3.3 Extensive chronic faults in different application platforms 23

5.1 Variable notations for response time characterization 33
5.2 Collected system metrics for dependency extraction 36

6.1 Metrics collected by DMADL for anomaly detection 47

List of Figures

2.1 The fault localization process of FChain 18

3.1 An example of anomaly propagation in multi-tier applications 24

4.1 The setup of 3-tier MediaWiki benchmark application 26
4.2 The setup of 3-tier Olio web application 26
4.3 The SolrCloud setup of CloudSuite web search application 27

5.1 An example of a multi-tier application 31
5.2 The composite dependency primitive 33
5.3 The concurrent dependency primitive 34
5.4 The distributed dependency primitive 35
5.5 A centralized view of dependency graph for an application 38
5.6 A distributed view of dependency at each component of the application 38
5.7 Distributed dependency primitive analysis at the load balancer in MediaWiki application . . . 40
5.8 The dependency primitive analysis in MediaWiki application 41
5.9 The relationship of request and response flow at the load balancer in Olio application 42
5.10 The lag correlation for the dependency primitive analysis in SolrCloud application 43
5.11 The impact of sampling interval on the distributed dependency primitive extraction 43
5.12 The impact of sampling interval on the distributed dependency primitive extraction 44
5.13 The impact of sampling interval on composite dependency extraction 44

6.1 The response time estimation scheme of DMADL using data packets in each TCP flow 48
6.2 The case with a single dependent component 52
6.3 The case with multiple dependent components 53
6.4 The DMADL estimation of mean response time and mean request rate in CloudSuite web search application 56

6.5 The DMADL estimation of mean response time and mean request rate in Olio web application 56
6.6 The DMADL estimation of mean response time and mean request rate in MediaWiki application 56
6.7 The DMADL detection result of resource faults in CloudSuite web search application 58
6.8 The DMADL detection result of resource faults in Olio web application 58
6.9 The DMADL detection result of resource faults in MediaWiki application 58
6.10 The DMADL detection result of performance bugs in CloudSuite web search application 59
6.11 The DMADL detection result of performance bugs in Olio web application 59
6.12 The DMADL detection result of performance bugs in MediaWiki application 60
6.13 The detection performance of DMADL and other methods in CloudSuite web search application. 61
6.14 The detection performance of DMADL and other methods in Olio web application 61
6.15 The detection performance of DMADL and other methods in MediaWiki application 61
6.16 The detection latency of DMADL and other methods in different Cloud applications. 62
6.17 The component impact analysis for anomaly localization in CloudSuite web search application 63
6.18 The component impact analysis for anomaly localization in Olio web application 64
6.19 The component impact analysis for anomaly localization in MediaWiki application 66
6.20 The localization performance of DMADL and other methods in CloudSuite web search application 68
6.21 The localization performance of DMADL and other methods in Olio web application 68
6.22 The localization performance of DMADL and other methods in MediaWiki application 69
6.23 The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in CloudSuite web search application. 69
6.24 The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in Olio web serving application. 69
6.25 The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in MediaWiki application. 70

7.1 The operation diagram of DMFDL for anomaly detection and localization 73
7.2 The traffic flow monitor for a component service A 74
7.3 The stability of flow ratio with different sampling intervals under varying workload intensity in CloudSuite web search application 85
7.4 The stability of flow ratio with different sampling intervals under varying workload intensity in Olio web application 86

7.5 The stability of flow ratio with different sampling intervals under varying workload intensity in MediaWiki application 87
7.6 The flow ratio for resource fault detection in CloudSuite web search application 88
7.7 The flow ratio for resource fault detection in Olio web application 89
7.8 The flow ratio for resource fault detection in MediaWiki application 89
7.9 The flow ratio for performance bug detection in CloudSuite web search application 90
7.10 The flow ratio for performance bug detection in Olio web application 91
7.11 The flow ratio for performance bug detection in MediaWiki application 91
7.12 The detection performance of DMFDL and other methods in CloudSuite web search application. 92
7.13 The detection performance of DMFDL and other methods in Olio web application 93
7.14 The detection performance of DMFDL and other methods in MediaWiki application 93
7.15 The detection latency of DMFDL and other methods in different Cloud applications 94
7.16 The request flow ratio for anomaly localization in CloudSuite web search application 94
7.17 The request/response flow ratio for anomaly localization in Olio web application 95
7.18 The request/response flow ratio for anomaly localization in MediaWiki application 97
7.19 The localization performance of DMFDL and other methods in CloudSuite web search application 99
7.20 The localization performance of DMFDL and other methods in Olio web application 99
7.21 The localization performance of DMFDL and other methods in MediaWiki application 100

8.1 The operation diagram of DMCDL for anomaly detection and localization 102
8.2 The stability of flow correlation with different sampling intervals under varying workload intensity in CloudSuite web search application 112
8.3 The stability of flow correlation with different sampling intervals under varying workload intensity in Olio web application 113
8.4 The stability of flow correlation with different sampling intervals under varying workload intensity in MediaWiki application 114
8.5 The stability of flow correlation with different correlation windows under varying workload intensity in CloudSuite web search application 115
8.6 The stability of flow correlation with different correlation windows under varying workload intensity in Olio web application 116
8.7 The stability of flow correlation with different correlation windows under varying workload intensity in MediaWiki application 117
8.8 The flow correlation for resource fault detection in CloudSuite web search application 117

8.9 The flow correlation for resource fault detection in Olio web application 118
8.10 The flow correlation for resource fault detection in MediaWiki application 118
8.11 The flow correlation for performance bug detection in CloudSuite web search application . . . 119
8.12 The flow correlation for performance bug detection in Olio web application 120
8.13 The flow correlation for performance bug detection in MediaWiki application 121
8.14 The detection performance of DMCDL and other methods in CloudSuite web search application 122
8.15 The detection performance of DMCDL and other methods in Olio web application 123
8.16 The detection performance of DMCDL and other methods in MediaWiki application 123
8.17 The detection latency of DMCDL and other methods in different Cloud applications 124
8.18 The request flow correlation for anomaly localization in CloudSuite web search application . . 124
8.19 The request/response flow correlation for anomaly localization in Olio web application 126
8.20 The request/response flow correlation for anomaly localization in MediaWiki application . . . 127
8.21 The localization performance of DMCDL and other methods CloudSuite web search application 129
8.22 The localization performance of DMCDL and other methods in Olio web application 129
8.23 The localization performance of DMCDL and other methods in MediaWiki application 130

9.1 The software detail for extensive chronic fault detection 132
9.2 Chronic memory fault detection in Apache HTTP server 133
9.3 Chronic concurrent fault detection in Apache HTTP server: child process hang failure 134
9.4 The detection latency at different process hang rates. 135
9.5 Chronic concurrency fault detection in Apache HTTP server: CPU usage spikes 136
9.6 Chronic semantic fault detection in Apache HTTP server: truncated response fault 137
9.7 Chronic semantic fault detection in Apache HTTP server: gradually increasing queue latency . 138
9.8 Chronic memory fault detection in Nginx HTTP server 139
9.9 Chronic concurrency fault detection in Nginx HTTP server: worker process failure 140
9.10 Chronic semantic fault detection in Nginx HTTP server: HTTP 206 partial content fault 142
9.11 Chronic memory fault detection in MySQL database server 143
9.12 Chronic concurrency fault detection in MySQL database server: CPU usage spikes 144
9.13 Chronic concurrency fault detection in MySQL database server: poor service performance . . . 145
9.14 The performance of different methods for anomaly detection in Cloud applications 146
9.15 The performance of different methods for anomaly localization in Cloud applications 147

Chapter 1

Introduction

Infrastructure-as-a-Service (IaaS) is a popular platform for providing virtual computing and storage resources to Cloud users. It allows many users to share a common physical infrastructure in a cost-effective way. Cloud applications are usually deployed in virtual components. These virtual components could be virtual machines or containers. Users manage their own virtual components and applications. The virtual components owned by Cloud users do not provide visibility or access to Cloud providers. Cloud providers must monitor the state of virtual components without access to their inner services. Due to their complex and dynamic nature, Cloud applications are prone to performance anomalies. Performance anomalies degrade the quality of experience to users and cause loss of revenue for service providers. When a single component has an anomaly, it propagates to other normal components through inter-component interaction. As a result, other components also show abnormal behavior. It is important to detect and locate the faulty component quickly before the faulty component causes a service failure.

Existing IaaS Cloud infrastructures include Amazon EC2, Microsoft Azure, and Google Cloud Platform. Amazon EC2 CloudWatch is a typical monitoring framework for virtual machines (VM). CloudWatch monitors the resource consumption and activity of each VM at a 5-minute polling interval. The monitored metrics include CPU utilization, disk reads/writes, network in and out, volume read/write, volume total read/write time, volume idle length and volume queue length. If Cloud users want to view more metrics of their VMs at 1-minute intervals, they have to pay an additional fee. CloudWatch also provides a set of system status checks for: 1) Loss of network connectivity and loss of system power and 2) Notification to VM owners of exhausted memory and corrupted file systems. 5-minute polling intervals are too long for performance-critical applications. Many things could go wrong in 5 minutes or even in 1 minute. The resource utilization metrics do not reflect the service performance properly. The service response time is more important to users than resource utilization metrics.

We propose non-intrusive dependency model-based methods for anomaly detection and localization in IaaS Cloud applications. Multi-tier Cloud applications run together on multiple components. These components have complex interactions and unpredictable behavior for different applications VDEP [43] introduces dependency primitives to characterize the execution sequence when a component calls multiple other components to process arrived requests. The basic dependency primitives together capture the complex behavior of distributed applications. The interactions of components in Cloud applications can be described by using three dependency primitives: Composite dependency, Concurrent dependency, and Distributed dependency. We identifies the dependency primitives using non-intrusive analysis and apply these dependency primitives in understanding the interaction of each component. Our system has agents distributed on physical hosts. The distributed agents run data monitoring tasks, interaction behavior analysis, anomaly detection, and localization tasks independently and autonomously. It does not require any knowledge from the operating system of virtual components or underlying application domains. Our agents do not send data to a central location for processing. The operation of each agent is simple. It is easy to deploy these agents on a large number of physical hosts. Our agents monitor network traffic outside virtual components and model the service performance of each component through analysis of traffic flows. The agents analyze the interaction behavior and use the performance model for anomaly detection and localization in Cloud applications.

1.1 Challenges

The scale, complexity, and dynamics of IaaS Cloud makes the interaction behavior characterization, anomaly detection, and anomaly localization challenging.

1. An IaaS Cloud data center has hundreds of thousands if not millions of physical servers. Each physical server could host tens of virtual components. Management of all these virtual components requires automatic and careful operation. It is unrealistic to have human operators detect anomalies and locate faulty components manually. An automatic tool for detection and localization is necessary for large-scale virtualized data centers.
2. There are a large number of system metrics available, such as different kinds of system resource usage and various performance counters in the operating system. It is difficult to determine the importance of each metric with regards to its relationship to the service performance. It is critical to determine the metrics which best reflect the service performance.
3. Cloud applications consist of multiple components interacting with each other to serve end users. It is important to understand their interaction behavior. The service performance of a component

depends on that of other components. Performance anomalies could propagate from one component to other components through their interactions. When an anomaly is detected at a component, it requires an effective method to determine whether the anomaly is caused by the local component or other components.

4. Cloud providers have limited access and visibility to the services inside virtual components. Intrusive monitoring is hard without the agreement and cooperation from users. The anomaly detection and localization must be done using traffic flows monitored outside virtual components.

1.2 Contributions

To address the aforementioned challenges, we propose non-intrusive and decentralized methods for anomaly detection and localization of faulty components in Cloud applications. We incorporate the interaction behavior amongst components and service performance model to further enhance accuracy in a non-intrusive way. Here are our main contributions:

1. Our work is the first anomaly detection and localization framework that incorporates dependency primitives for characterizing the performance of each component in normal operation. Existing works study the dependency path, but they do not consider the execution sequence in anomaly detection and localization. The execution sequence of components is important in multi-tier applications. It provides insights on how strong the behavior of a component is correlated with the behavior of other components.

2. The local interaction behavior helps our methods to quickly filter out normal components in localization of faulty components. The interaction behavior is not impacted by the fault propagation if all involved components in the local dependency model are normal. Other methods based on resource utilization do not utilize the interaction behaviors among application components. The fault propagation affects the resource usage of all components in the application. With the local dependency primitive at each component, our methods have a clear view of how the component interacts with other components. We show that a fault could propagate vertically amongst components in the same tier even though they do not interact with each other directly. Without understanding of the dependency primitive, it is very difficult to locate the true faulty component by checking individual components separately.

3. Our methods are non-intrusive. We use nothing more than the TCP header of packets. It is easy to deploy our agents on the IaaS Cloud platform and agents operate without a complex configuration.

The agents run on physical hosts and they do not need coordination from virtual components. Our methods are based on the fact that the response flow follows the request flow. We not only estimate the service response time, but also model the service performance using the relationship between request flow and response flow of components. Our methods achieve high precision and recall in detection of performance anomalies and localization of faulty components in realistic deployment of Cloud applications with a low overhead.

4. We evaluate our non-intrusive methods with extensive chronic software faults. Prior works for chronic software fault detection are based on analysis of software source code. It requires extensive efforts to do software debugging. Our methods show that it is possible to detect chronic software faults using traffic flows outside virtual components without the knowledge of the specific application software.

1.2.1 Dependency Model

The dependency model characterizes how a component interacts with its dependent components. We count the number of request data packets and response data packets at each component to infer the causal execution order. We do not need to parse application level protocols to extract request and response events for dependency model construction compared to VDEP. We validate our method for dependency model extraction with realistic deployment scenarios using the Olio multi-tier application, the CloudSuite web search application (SolrCloud setup), and the MediaWiki multi-tier application. Our method identifies the interaction behavior among components correctly in deployed applications. We avoid isolated analysis of each component by accounting for the dependency model in anomaly detection and localization. We show that the dependency model improves the performance in anomaly detection and localization compared to other methods that do not consider the dependency model.

1.2.2 DMADL Method

DMADL is a method that uses the estimated mean response time at each component with the dependency model for anomaly detection and localization. DMADL approximates the response time of each request as the elapsed time from the moment when he first request data packet arrives at the component to the moment the last response data packet leaves the component. The response time of a component represents the total amount of time spent by the component and its subsystem for processing requests. While existing methods rely on either analysis of server logs or inspection of packet payload, DMADL agents capture only data packets and analyze the TCP headers of each packet. We compare the estimated end-to-end

response time with the logged response time at both client side and server side. The results show that the estimated mean response time can accurately reflect the service performance at each component. We evaluate DMADL using both common fault models and performance bugs. The results show that DMADL achieves almost 95% detection precision and 5% false positives and false negatives for anomaly detection in deployed Cloud applications. We compare DMADL with existing resource utilization-based anomaly detection methods. DMADL achieves 32.3% higher detection precision and 21.9% fewer false negatives than resource utilization-based methods. DMADL also has a smaller detection latency, i.e., less than 5 seconds. DMADL detects anomalies over 20 seconds earlier than correlation methods.

DMADL further derives the subsystem response time of each component using the dependency model and the estimated response time of each dependent component. DMADL analyzes the impact of each component in real time and pinpoints the component with the highest impact on the service performance as faulty. We evaluate the effectiveness of DMADL through extensive experiments with different kinds of faults in multiple applications. We compare DMADL with another component-level localization method, FChain [42]. The results show that DMADL achieves 31.1% higher localization precision and 15.8% fewer false negatives than FChain. DMADL incurs less than 1% CPU cost and negligible memory cost for each virtual component under varying workload intensity. DMADL does not interfere with the performance of running services on virtual components or physical hosts.

1.2.3 DMFDL Method

DMFDL builds a flow ratio model and combines it with the dependency model for anomaly detection and localization. The flow ratio model uses an adaptive ratio between the number of outgoing response data packets and the number of request data packets to model the service performance in normal operation. It shows that the flow ratio is stable under varying workload intensity at each component of Cloud applications. A DMFDL agent detects an anomaly if the flow ratio in a decision window deviates too far from the flow ratio in normal operation or if the local dependency model changes. We compare DMFDL with resource utilization-based anomaly detection methods. DMFDL achieves over 22.6% higher detection precision and 17.4% fewer false negatives than resource utilization-based methods. DMFDL detects anomalies over 12 seconds earlier than methods that use the correlation between resource utilization.

DMFDL uses the local dependency model and the flow ratio model for anomaly localization. DMFDL agents use the ratio between the request flow to dependent components and the request flow to the local component to characterize the interaction behavior between the local component and its dependent components. DMFDL agents pinpoint components whose request flow ratios change the most or the earliest

as faulty under different dependency primitives. DMFDL achieves 21.5% higher localization precision and 11.2% fewer false negatives than FChain. DMFDL incurs less than 1% CPU cost and negligible memory cost for each virtual component under varying workload intensity. DMFDL does not interfere with the performance of services running on virtual components or physical hosts.

1.2.4 DMCDL Method

DMCDL applies the correlation analysis between the input request flow and the output response flow for modeling the service performance at each component of Cloud applications. In normal operation, the output response flow changes with the input request flow within an acceptable time limit. The flow correlation decreases if there is an anomaly in the component or its subsystem. We evaluate the stability of the flow correlation under varying workload intensities and use the flow correlation to characterize the service performance in normal operation. We compare DMCDL with resource utilization-based anomaly detection methods. DMCDL achieves around 20.1% higher detection precision and 17.0% fewer false negatives than resource utilization-based methods. DMCDL detects anomalies with larger latency than DMADL and DMFDL. DMCDL still detects anomalies with 5-second smaller detection latency than methods that use the correlation between resource utilization.

DMCDL uses the correlation between the request flow to a local component and the request flow to its dependent components to characterize the interaction between the local component and its dependent components. DMCDL agents pinpoint a dependent component as faulty if the request flow correlation of the dependent component decreases the earliest or the most under different dependency primitives. DMCDL uses the dependency model and the request flow correlation for anomaly localization in Cloud applications. We compare DMCDL with FChain, another non-intrusive component-level localization method. The results show that DMCDL achieves around 19.9% higher precision and 9.2% fewer false negatives in anomaly localization than FChain.

1.2.5 Non-Intrusiveness and Low Overhead

By restricting the task of traffic monitoring, dependency model extraction, anomaly detection, and localization to local components, the load on each agent does not increase with the size of the application nor the size of the data center. The load only depends on the workload intensity. Each agent operates autonomously, and it does not need coordination or communication from other agents during operation. Each agent does localization analysis only after it detects anomalies. Centralized localization methods have to analyze the behavior of all components when it detects anomalies in applications. Compared to

these methods, our methods reduce unnecessary operation and have more accurate results. Our agents do not require any domain knowledge and they take each component as a black box using passive monitoring techniques. It has low overhead, and it does not interfere with the performance of application services deployed inside virtual components.

1.2.6 Extensive Chronic Fault Evaluation

We evaluate our methods DMADL, DMFDL, and DMCDL in detecting sampled faults from three different classes of chronic software faults as used in [45]. Although no individual technique could detect all chronic faults, multiple techniques used together could. For those chronic software faults that do not show obvious symptoms from their occurrence, our methods could still detect these faults at an early stage.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, we discuss related works in anomaly detection and localization. Chapter 3 describes the fault model and fault propagation in Cloud applications. The experimental setup and evaluation criterion are described in Chapter 4. We analyze different dependency primitives and derive the relationship between subsystem response time and the mean response time of each dependent component in Chapter 5. We present DMADL for anomaly detection and localization in Chapter 6. We introduce DMFDL and evaluate its performance in Chapter 7. In Chapter 8, we introduce DMCDL and show its performance in anomaly detection and localization. We use our methods (DMADL, DMFDL, DMCDL) for detection of extensive chronic faults and show the comparative analysis of their performance in anomaly detection and localization in Chapter 9. We conclude the thesis in Chapter 10.

Chapter 2

Related Works

In this chapter, we review existing research works on performance monitoring and anomaly detection, application dependency, and anomaly localization by different methods and show our difference compared to their work.

2.1 Anomaly Detection

The goal of anomaly detection is to identify when the application performance is deviating from the specified or expected performance requirement. There are two different classes of anomaly detection methods: metric-based methods and model-based methods.

2.1.1 Response Time-based Anomaly Detection

The response time is an important performance metric in understanding the performance of applications. From users' perspective, the response time measures how responsive the application is while interacting with users. This class of approaches estimate the end-to-end response time as one of the most important metrics for performance anomaly detection.

There are many works in the literature using server-side response time estimation for anomaly detection [2, 5, 9, 11, 22, 44]. EtE [2] relies on request-response reconstruction and web page reconstruction to estimate the overall end-to-end service latency from the analysis of network traces. EtE requires the analysis of TCP segment payloads to extract all HTTP requests and corresponding responses. It reconstructs page access statistics by grouping requests and responses of each web page from knowledge base and statistical analysis. EtE may compete with running applications intensively for CPU and I/O bandwidth. It is better to place EtE as an independent network appliance between clients and application servers.

Olshefski et al. proposed ksniffer [5] and RLM [9] to passively infer application-level response times for HTTP transactions. ksniffer needs a kernel module to be installed on the server system, and RLM runs as a network appliance between clients and application systems. However, both tools require to parse TCP payloads and analyze HTTP headers, making them unsuitable for encrypted traffic such as HTTPS. As HTTP headers can be embedded anywhere in the TCP payload, they have to analyze all payload content and may incur high overhead. sMonitor [11] estimates the response time for HTTP and HTTPS through packet size analysis. All these works require intrusion and visibility into operating systems and application layers.

Adudump [22] continuously collects and processes TCP and IP headers in a streaming fashion to build client-server interaction model from the sequence and acknowledgement numbers. It further extracts the server-side response time and compares the response time distribution with historical distributions for detecting anomalies.

PBAD [44] estimates the upper bound of the server-side response time from packet counts and uses the estimated response time for anomaly detection. The adudump is the closest approach to ours for estimating the response time using only TCP headers. However, DMADL uses only data packets that contain TCP payload, and it does not need to reconstruct the TCP dialogue from the SEQ and ACK numbers. We estimate the mean response time for anomaly detection in real time. Other classes of anomaly detection approaches rely on the application or system logs for anomaly detection [23, 24]. Deep packet inspection techniques have scalability issues in terms of computational and storage capacity.

2.1.2 Queuing Model-based Anomaly Detection

Queuing model approaches model the application system as a set of connected queues. They take components such as the inter-arrival process, the service process, the number of servers and customers into consideration.

Urgaonkar et al. [6] uses a network of queues to model each tier of multi-tier applications. Their analytic model mostly considers the session-based workload, concurrency limits, and caching effects at each tier. They estimate response time by the mean value analysis. Their model assumes the arrival rates and residence time at each tier are provided by the logging mechanisms. They have to estimate the visit ratio and the service time at each tier. They also have to estimate the think time of each client.

Zhang et al. [16] applies a regression-based approximation of the CPU demand of client transactions on a given hardware. Then we use this approximation in an analytic model of a simple network of queues, each queue representing a tier, and show the approximation's effectiveness for modeling diverse

workloads with a changing transaction mix over time. Also, the unit of work is transactions, not web requests. They do not consider the inherent performance variability in a cloud environment and the load on the servers in terms of used memory and disk I/O.

Bi et al. [25] proposes a network of queuing models of multi-tier applications. First they need to determine the capacity of multiple VMs for each tier in terms of the request rates they can handle, then they compute the number of VMs required at each tier to satisfy the requirement of customer response time. Their approach requires monitoring system resource usage metrics (such as CPU, memory, network, and disk), and request rates at each single server of each tier. In their systems, per-tier service times are assumed to be drawn from a known fixed distribution. They consider that the per-tier utilization of VM for virtualized multi-tier application is the utilization metric correspond to the utilization of the busiest resource (e.g. CPU, disk, or network) for the tier. However, in their approach there is a very obvious problem that the service rate of the multi-tier application is the sum of service rates at each tier. It is not true because there is a dependency relationship among different tiers. The server rate at each tier could affect the service rate at other tiers, and simple summing the service rates at different tiers could cause deviated performance estimation.

2.1.3 Machine Learning Model-based Anomaly Detection

This class of approaches model the system performance using machine learning models. They either build the performance index as the regression model of various system resource usage metrics. They train the performance model during normal operation, and trigger anomaly alert when the system operation deviates from the trained model.

Cherkasova et al. [20] models the application resource consumption (mainly CPU consumption) by a set of different application transactions. They find out the CPU consumption for each transaction type by solving a regression model. If the observed CPU consumption cannot explained by the transaction model, they trigger anomaly alert or application performance change. They further profile the application latency and combine it with the transaction model to find out the service time for each type of transaction as the performance signature. After detecting performance anomalies, they compare the new performance signature against the old one to distinguish the transaction which caused the model change. This approach implies extracting details from all transactions, such as overall latency, outbound calls, and the average latency of outbound calls. Another drawback is that the CPU consumption model may not work well in modern applications, which contain many different transactions of complex dynamic contents. Lastly, it requires a large amount of training data and online adaptation to model the CPU consumption for each

transaction model, which may not be accurate enough when the training sample is too small and cannot scale to other servers.

UBL [31] uses Self-Organizing Maps (SOM) on resource utilization metrics to capture normal system behaviors and predict unknown anomalies. UBL first collects resource utilization of VMs (e.g., CPU usage, memory allocation, network I/O, disk I/O) in normal operation to learn the normal behavior of the system. UBL uses SOMs to model system behaviors in two different phases: learning and mapping. During the learning phase, the SOM uses a competitive learning process to adjust the weight vectors of different neurons. For each normal sample, the SOM updates the closest neuron and its neighborhood neurons. During application run-time, it maps each measurement vector to a neuron in the map and determines whether the sample is normal or abnormal from the distance between the neuron and its neighborhood neurons. There are several parameters to be configured in order to use SOM properly for anomaly detection, such as the map size, neuron vector initialization, and threshold for detection decision. Improper configuration would result in bad detection performance of the SOM. It also imposes a large overhead as the size of SOM increases in order to accurately model the dynamic system. UBL's performance degrades when the system has a completely previously-unseen workload.

Neural networks are used to analyze and learn the normal usage behavior of Cloud customers and detect anomalies [36]. The focus of the work is to detect security attacks, not performance anomalies.

Zhang *et al.* [53] presents an improved incremental self-organizing Map (IISOM) model for anomaly detection of virtual machines. They collect around 1000 run-time performance measurements to train the self-organizing maps for anomaly detection. They use a Weighted Euclidean Distance (WED) algorithm to speed up the training process and improve the model quality. However, their approach has similar problems to UBL [31]. The parameters of map size of SOM, learning rate, and initial weight value of neurons still have to be predefined by empirical estimation. It may incur high computational complexity and iterative regression for empirical estimation.

Sauvanaud *et al.* [55] describes an anomaly detection system (ADS) designed to detect errors related to the erroneous behavior of the service, and SLA violations in cloud services. It uses a system monitoring entity to collect metrics from both hypervisor counters and virtual machine OS counters, and detect anomalies using supervised machine learning models. The machine learning algorithms include random forests, neural networks, nearest neighbors, naive bayes, support vector machine (SVM), and gradient boosting algorithm.

2.1.4 Correlation-based Anomaly Detection

The correlation-based approaches first extract metrics that have high correlation with each other and then uses their correlations to characterize the application performance. Their detect performance anomalies when the corresponding correlations drop below a certain threshold.

PeerWatch [28] builds a canonical correlation analysis (CCA) model to extract correlated behaviors of system metrics between multiple application instances, and uses the model to cross check the status of those instances. It requires multiple instances in Cloud applications to exhibit similar workload pattern in terms of resource utilization pattern. This is a limitation in dynamic Clouds, where different instances show completely different pattern in resource usage. Correlation analysis across different instances may produce false positives, as different components could be totally different in their resource usage since different application functions require different resources.

Magalhaes *et al.* [26] monitors client transactions and system metrics to associate the response time with the number of processed transaction mixes. The Pearson correlation coefficient is used to describe the relationship between the amount of concurrent users and collected system parameters. A low correlation coefficient triggers the anomaly alert.

LFD [32] detects anomalies based on the hypothesis that the processing of a request alternates between the communication phase and the computation phase. It hypothesizes that the alternating activity induces correlated behavior between user-space CPU utilization with other system resources' consumption when the system is normal. They trigger an anomaly alert when the correlation between user-space CPU utilization and other system resource consumption decreases. LFD has to be implemented in each VM individually to explore the correlation between the user-space CPU utilization and other system metrics, and thus incurs an overhead that may affect the application performance.

CloudPD [37] combines resource utilization models with linear pairwise correlation models as an invariant of application behavior to detect anomalies. It requires both system-level and application-level metrics, such as service latency and throughput. CloudPD has to learn stable correlations among a large set of metrics. This limits its scalability in large Clouds due to numerous metrics in a data center with a very large number of VMs and applications.

PAD [41] proposes a combined threshold analysis and correlation-based approach for anomaly detection and root cause identification. PAD first collects data from different performance counters and compares counter values or their statistical properties (i.e., mean, median, or quantile) with predefined thresholds to identify violated performance counters. PAD uses correlation (Pearson and Spearman) coefficients to find out those performance counters that are responsible for the anomaly.

LADT [47] is a lightweight anomaly detection tool based on LFD. LADT correlates host and aggregated virtual machine (VM-level) metrics in Cloud data centers. LADT is based on the hypothesis that metrics from physical hosts and virtual machines (VMs) are strongly correlated in an anomaly-free system. LADT detects anomalies via correlation analysis between host-level and VM-level system metrics. As we know, the host resource utilization almost equals the aggregated resource utilization of all virtual machines. This approach works if there is anomaly with the physical host software, but it causes false negatives when the VMs have performance anomalies.

Wang *et al.* [48] uses an online incremental clustering method to recognize access behavior patterns, and uses the canonical correlation analysis (CCA) to model the correlation between workloads and metrics related to application performance or resource utilization in a specific access behavior pattern. It detects anomalies by discovering an abrupt change of correlation coefficients in an exponentially weighted moving average (EWMA) control chart, and locates suspicious metrics using a feature selection method. This approach assumes prior knowledge about the underlying applications and the workload. It requires knowledge of the exact number of components (servlets) in the application, and the details about different types of user requests.

EbAT [27] analyzes distributions of different metrics collected from virtual machines and physical hosts. They use entropy to capture the degree of dispersal or concentration of the distributions. They aggregate several raw metrics across the data center to form an entropy distribution. Finally, they detect anomalies via wavelet analysis and visual spike detection in general and at each level of the hierarchy.

2.2 Anomaly Localization

In multi-component applications, a fault at any component could propagate to the whole system such that many components behave abnormally. When a performance anomaly is detected, it is important to determine which component is the source of the anomaly. Works towards anomaly localization are divided into three different classes: localization based on system traces, using application dependency graphs, and using machine learning techniques.

2.2.1 Trace-based Anomaly Localization

Tracing requests' execution path is typical class of approach for locating faults and diagnosing problems. It usually requires either tagging requests and logging requests at each components (application-level). These approaches could give a clean picture of system operation. But it requires both domain knowledge

and too much human configuration or instrumentation to work well. It is hardly to scale this approach to heterogeneous applications in IaaS Cloud today.

Pinpoint [1] is a tool used for accurate detection and localization of faulty components in large distributed systems. The authors modify the J2EE middleware to capture the paths and track the success or failure status of requests traveling through a distributed system by tagging J2EE calls with a request ID. Pinpoint further infers components that are causing faults with client traces and request success/failure logs. It uses a data clustering algorithm to find out components that are highly correlated with failures of requests. One limitation of Pinpoint is that the middleware instrumentation requires visibility of VMs and applications, which is not reasonable in IaaS Cloud. A more obvious limitation is that it applies only to services based on the J2EE platform.

Magpie [3] specifically associates the traced messages with each incoming request by tagging a unique identifier and associating resource usage throughout the system with that identifier. It implies a much more sophisticated tracing infrastructure for debugging of low-level system problems. Project5 [4] and WAP5 [10] are aimed at debugging wide-area distributed applications by exposing causal communication sequences within applications and the delays of components that imply bottlenecks. They solve these problems by gathering traces, particularly overlapping traces from multiple sniffers, and reconciling them into a single trace for analysis.

PerfCompass [50] is a fault debugging tool based on the premise that it is important to distinguish between faults with a global impact and faults with a local impact, since the diagnosis and recovery steps for faults with a global impact or local impact are quite different. PerfCompass can use this information to suggest the root cause as either an external fault (e.g., environment-based) or an internal fault (e.g., software bug). PerfCompass records system calls from monitored application using lightweight kernel-level system call tracing tools. It performs fault localization and inference using four steps. First, it segments the large raw system call traces into execution units, which are groups of closely related system calls. Next, these execution units are processed to extract a set of fine-grained fault features (e.g., which threads are affected by the fault, or how quickly the fault manifests in different threads). Third, these fine-grained fault features are used to differentiate between faults with a global impact and faults with a local impact and identify the root cause as external or internal. Lastly, PerfCompass identifies the top affected system calls and provides ranks for those affected system calls, which allows developers to identify affected subsystems (e.g., network, I/O, CPU) and further diagnose the fault. Note that PerfCompass requires instrumenting the kernel running on each VM with a kernel tracing tool to monitor the system calls generated by each application running on that VM.

Roots [51] is a system for detecting anomalies and identifying their root cause in web applications

deployed in Platform-as-a-Service (PaaS) clouds. Roots tracks events within the PaaS Cloud related to each specific request and records the latency for each request for anomaly detection. Roots requires metadata injection and platform-level instrumentation. It requires tagging each request at the front-end web server for each application. It then intercepts and records events as the application code invokes various service implementations of the PaaS cloud using the tagged ID. Roots also records the latency of each application call to the internal cloud service implementation. In order to perform data collection, Roots must be able to introspect the entire platform software stack. Roots truly provides the accurate request execution path, but it requires too much human efforts and supports only a specific set of applications. In this work, we aim at detecting and localizing performance anomalies. Roots can help in detail performance diagnosis to find out the root cause after we pinpoint the root cause component.

Mdhaffar *et al.* [52] presents a cross-layer reactive monitoring approach for Cloud computing environments. The work monitors performance metrics across 4 different layers: physical host layer, virtual machine layer, middleware platform layer, and application layer. It uses a complex event processing (CEP) methodology to detect and repair performance-related problems. The correlation coefficients between metrics across different layers is used to reduce the number of monitored parameters and derive the interactions and root cause of performance-related problems.

These trace-based approaches precisely record the execution path information and locate the abnormal component through instrumenting either the application or middleware platform. They are very helpful to debug distributed applications, but the overhead brought by these tools is significant, hindering the possibility of deploying them in IaaS Clouds. Additionally, deploying these tools requires the administrators to understand heterogeneous middleware platform or application source code running inside VMs. Compared to these tools, we aim at developing an approach that has low overhead and does not need any specific knowledge of the underlying platform or applications running on VMs. It operates outside VMs for anomaly detection and localization.

2.2.2 Application Component Dependency Analysis

Deriving an application dependency graph is important for understanding the whole topology of multi-component applications. The dependency graph is useful to understand how performance anomalies or faults propagate from the source component to the whole application, and can be used to determine fault propagation paths and identify the root causes of anomalies.

The Sherlock [15] system is aimed at giving IT administrators the tools they need to localize performance problems and hard failures that affect end-users. Sherlock detects the existence of faults and

performance problems by monitoring the response time of services and localizes the problem to the most suspicious component. It needs to create an observation node for each client reporting service response time, making the service meta-node a parent of the observation node.

Constellation [18] uses a black-box approach to learn explicit models of dependencies using little more than the timings of packet transmission and reception. Constellation is aimed at discovering network-wide service dependencies and can help in diagnosing network-wide problems. The dependency building algorithm used by Constellation can also run in a distributed fashion. Orion [19] discovers machine dependencies using packet headers and timing information in network traffic based on a novel insight of delay spike based analysis.

CauseInfer [40] focuses on the dependency directions between every pair of communicating components in client-server Cloud applications. CauseInfer leverages the fact that packets sent by a server follows the change in packets sent by a client with some delay. CauseInfer uses the lag correlation of the sending traffic between two services to distinguish the dependency direction. For example. if a service A depends on another service B, the traffic sent by A follows the change in the traffic sent by B. We consider multiple pair-wise dependency relationship to characterize the interaction behavior between the local component and multiple dependent components.

VEDP [43] discovers dependency models that incorporate complex application behavior/interaction patterns in addition to communicative and causal relationships. VDEP introduces three dependency primitives as basis to represent the basic behaviors of component dependencies and they together capture the complex behavior of distributed Cloud applications. It derives response time characteristics for each dependency primitive and uses response time characteristics to validate the accuracy of dependency extraction. However, VDEP requires inspection into the packet payload to extract request and response events. It requires understanding of application protocols and decryption of encrypted application protocols such as HTTPS. These operations involve specific inspection operations for each specific service protocol. In real data center applications, the dependency graph can change frequently due to component failure and replacement, or component function change. We have agents build only local dependency which characterizes only the interaction between the local component and its dependent components. Whenever a component within the same application changes, only those components that involve direct interaction with the changed component needs to update its dependency graph. Thus, it is more robust to individual component change. In a centralized system, the process of building dependency graph has to repeat for all components.

2.2.3 Machine Learning-based Anomaly Localization

NetMedic [21] periodically captures a large set of variables ranging from system level to application level for problem diagnosis. The captured metrics include system resource utilization, exchanged traffic, application-level process activity such as inter-process communication, the fraction of failed requests, and the process response time. Then it models the communication behavior of components in the network as a dependency graph. After computing the abnormality of each component based on their deviation from normal history, NetMedic represents the component impact as weighted edges in the dependency graph. Finally It formulates detailed diagnosis as an graph inference problem which finds out the most likely cause with highest impact on the network.

PAL [30] pinpoints the faulty components in distributed applications by extracting anomaly propagation patterns. PAL first detects critical change points at different components, and then sorts all critical change points in chronological order to derive the propagation pattern. PAL assumes that the system-level metric changes caused by normal workload fluctuations are less significant than those caused by the performance bugs and that the anomaly onset time instants at faulty components are earlier than those of other components. PAL relies on either an external monitoring tool to keep track of the application SLO status. These assumptions may produce false positive as the workload for Cloud applications is unpredictable. Different types of workload consumes different resources intensively. The change point detection algorithm may not be able to differentiate anomaly change points from normal change points. In a distributed application, a component fault may affect the whole system instantly. A false change point leads to the false positive and false negative for anomaly localization.

FChain [38] uses a combination of predictability-based abnormal change point selection and dependency information for fault localization. FChain monitors low-level system metrics for each VM and learns normal fluctuation patterns for each metric. When a performance anomaly is detected, FChain first selects abnormal change points and sorts the starting time of abnormal changes at different components to identify the propagation path. It pinpoints the faulty component as the source component that shows the earliest fault manifestation. Further, it filters out spurious abnormal change propagation paths using inter-component dependencies. For example, Figure 2.1 shows how FChain pinpoints faulty components after it detects abnormal changes at the two application servers and the database. The application server 1 starts to exhibit abnormal change at time $t_0 = 200s$, and it is earlier than the application server 2 ($t_1 = 205s$) and the database ($t_2 = 210s$). FChain first infers that the abnormal changes start at application server 1 and it propagates following the path from the application server 1 to the application server 2, and finally to the database. Second, FChain finds that there is no dependency between the application

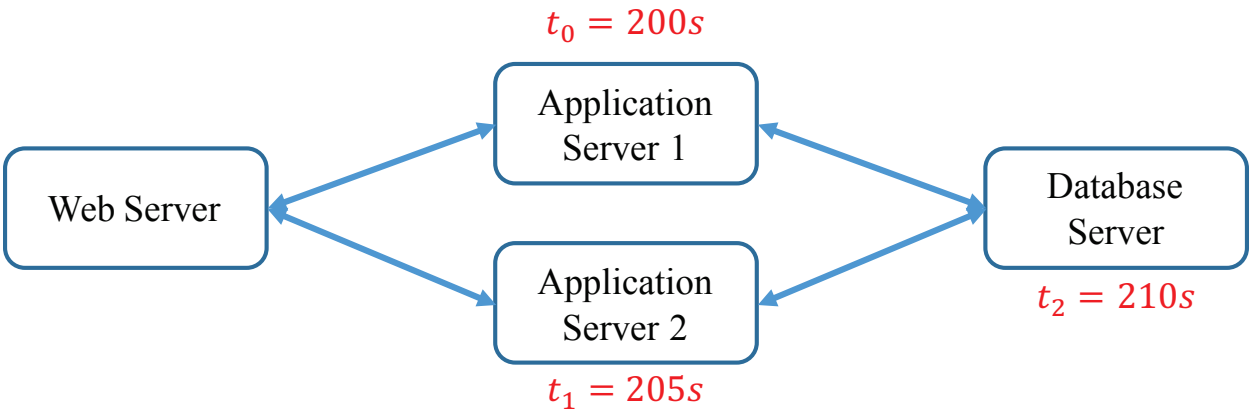


Figure 2.1: The fault localization process of FChain

server 1 and the application server 2, and infers that the propagation path from the application server 1 to the application server 2 does not exist. Therefore, FChain pinpoints the application server 2 as faulty as well. Finally, FChain pinpoints the two application servers as faulty components. It works well while the difference between the changing points of different components is observable (tens of seconds). In our applications, we observe that the anomaly propagates from the faulty components to normal components almost within 1 second. It is difficult to pinpoint the faulty components based on the chronological order of change points of different components. Another problem is that FChain requires a sufficient amount and diversity of training data to represent normal system states under different workload scenarios and all possible transitions between different states.

Zhang *et al.* [56] proposed and implemented a system called Deepview for virtual hard disk (VHD) failure localization in IaaS Cloud where virtual machines use remote storage service. Deepview first builds a global view of the system computing cluster, storage cluster, and network devices. It simplifies the global system view by aggregating possible paths between each pair of computing and storage cluster. They start from the bottom and go up for each cluster and aggregate the network devices by tiers, and then use shortest path routing to find the lowest overlap between each cluster pair. Deepview pinpoints the problem to network tiers after the simplification. Deepview further uses the Lasso regression with L1-norm constraint to estimate the failure probability for each component. Finally, Deepview uses hypothesis testing on the failure probability to decide whether to blame the corresponding component or not. Compared to Deepview, our approach focuses on virtual components and targets performance anomalies rather than disk failures.

Chapter 3

Fault Models

We evaluate our proposed systems with an extensive and comprehensive set of faults found in computing systems. We first introduce the fault models that we use in all of our experiments. We also analyze how a fault propagates among components in Cloud applications.

3.1 Fault Model

We introduce the faults in three categories. The first category of faults is about resource bottlenecks. These faults are mostly caused by improper resource allocation or bursting workload on IaaS Cloud. The second category of faults deal with software bugs. These faults cause immediate performance degradation, service hang, or even service crash problems. The third category of faults are extensive chronic faults. These faults do not have an immediate impact on the service performance and they usually hide in the system for a long time compared to software faults in the second class. We mainly focus on the faults that manifest as unacceptable response time. We do not consider faults that directly cause the virtual or physical infrastructure to crash. Crash faults (e.g., virtual machine crash, physical host crash) can be easily detected by a simple heartbeat mechanism.

3.1.1 Resource Bottleneck Fault

We first introduce common resource bottlenecks in Cloud applications as used in previous work in Table 3.1. We also show how we inject these faults in our virtual components.

Table 3.1: Fault Models, Fault Scenarios, Fault Injection Techniques and References

Fault Models	Fault Scenarios	Fault Injection Techniques	Reference
CPU-fault	• Application bug: infinite loops • CPU bottleneck by unexpected high workload, or resource contention in software	• Inject program to run infinite loops • Unexpected workload to cause CPU bottleneck	[12, 17, 28, 26, 27, 32, 33, 34, 38, 48, 50]
Memory-fault	• Memory leakage: continual memory allocation, no freeing • Memory contention: no available memory	• Run another memory-intensive program • Unexpected workload to cause CPU bottleneck	[12, 17, 28, 26, 32, 33, 34, 38, 48, 50]
Network-fault	• Congested network	• Packets are dropped with a random probability • Inject additional delay to packets	[17, 28, 32, 48, 50]
Disk-fault	• Disk I/O contention	• Run disk I/O intensive process in background	[12, 17, 28, 26, 27, 32, 33, 34, 38, 48, 47, 50]

3.1.2 Common Software Fault

There are numerous cases of service degradation due to software performance bugs. We search for software performance bugs reported by real world users in the bug repository. We focus on software performance bugs that cause service slowdown or hang. We reproduce those bugs as given in the bug reports to recreate the problem in our system. For those bugs that do not provide bug replication procedure, we emulate them with similar effect on the system. Table 3.2 gives the list of software performance bugs.

3.1.3 Extensive Chronic Fault

Chronic software faults occur in common software systems. Chronic software fault could hide in the system for a long time before it causes serious performance degradation. Randomly chosen samples of software faults from bug reports may not represent chronic faults properly. It is better to classify software faults into different classes and then sample a set of faults from each class.

The chronic software faults have three root causes according to Li et. al [13].

Table 3.2: Software performance bugs occurred in different application platforms

System	Fault ID	Fault Description
SolrCloud	SOLR-138	A burst of “expensive” queries can cause contention within lucene, greatly reducing effective throughput and causing more and more queries to stack up.
SolrCloud	SOLR-5935	SolrCloud hangs when querying and indexing is run at the same time during performance tests.
SolrCloud	SOLR-5216	A long time of stress-testing of document updates to SolrCloud can cause a distributed deadlock. The Solr instances start to see each other as down, flooding the Solr logs with "Connection Refused" exceptions
SolrCloud	SOLR-4940	The cluster crashes after executing a long and large query on the index set.
Apache	HTTPD-48905	On busy websites, some processes hang out and no longer process user requests.
Apache	HTTPD-57628	Apache processes randomly crash when serving request to download large file.
Apache	Configuration	Low number of connection pool threads causes the server to reject client connections.
MySQL	MySQL-54332	A MySQL deadlock bug that occurs when each one of two connections locks one table and tries to lock the other.
MySQL	MySQL-40968	MySQL server hangs when the server is under heavy load.
MySQL	MySQL-87164	MySQL queries running much slower in version 5.7 versus 5.6.
PHP	PHP-62418	The php-fpm master process crashes randomly on lightly loaded server.
Nginx	Nginx-62418	THe Nginx worker process crashes under heavy load.

1. Memory faults caused by improper handling of memory objects;
2. Concurrency faults that occur in multi-threading or multi-process environments;
3. Semantic faults which are caused by inconsistency with the design requirements or the programmer's intention. Faults that are not classified as memory or concurrency faults are considered as semantic faults [13].

We use these chronic software faults as they contain the most common and critical types of faults and they are the design focus of bug detection tools [7, 13]. For memory faults, we search for bugs that cause slow memory leakage. For concurrency faults, we search the bug reports using keywords such as "race", "lock", "concurrency", and "synchronization". We identify chronic faults as those faults that do not cause complete system failure and/or can persist for a long time before causing performance degradation. Concurrency faults usually cannot be reproduced easily, especially if they are subtle chronic faults. Hence, we analyze the effect of these faults and emulate these faults with similar effect as those chronic concurrency bugs. We have selected Apache HTTP Server [1], Nginx [2], and MySQL [3] as main software components to evaluate chronic software faults. Apache and Nginx are the two most popular web servers in market share. MySQL is the most popular database in market share. Table 3.3 shows the target software components, fault effect, and related bugs in evaluation of our methods. All selected bugs are found in Apache bugzilla [4], Nginx bugzilla [5], and MySQL bugzilla [6]

Most semantic faults in Apache and Nginx result in malformed responses to clients. Examples of these faults in Apache include corrupted header fields or a truncated body in the response. The chronic faults in Nginx cause it to send corrupted or malformed response content to clients. These chronic faults are usually preceded by data corruption in the system.

3.2 Fault Propagation

When a component becomes faulty, the fault propagates to other components through inter-component interaction. Understanding the fault propagation path helps to localize the faulty components from multiple components that exhibit abnormal behavior. Here we describe fault propagation from two different perspectives: dependency path and TCP path.

[1] Apache HTTP Server: `https://httpd.apache.org/`.
[2] Nginx HTTP Server: `https://nginx.org/en/`
[3] MySQL Database Server: `https://www.mysql.com/`
[4] Apache Bugzilla: `https://bz.apache.org/bugzilla/buglist.cgi?quicksearch=httpd`
[5] Nginx Bugzilla: `https://trac.nginx.org/nginx/report`
[6] MySQL Bugzilla: `https://bugs.mysql.com/`

Table 3.3: Extensive chronic faults in different application platforms

Software Component	**Fault Effect**	**Bugs**
Apache HTTP Server	slow memory leak	Apache #25667, #35404, #43223, #44975, #53435, #55296, #56271, #497077, #849272, #33899, #44026, #44783, #61222
Apache HTTP Server	child process failure	Apache #10266, #47370, #50702, #59798, #60071, #98979, #119128, #641968
Apache HTTP Server	high CPU usage	Apache #5225, #37680, #52858, #57544, #57800, #117832
Apache HTTP Server	truncated response fault	Apache #27292, #50481, #56176, #57476, #61147, #908583, #1569081
Nginx Application Server	slow memory leak	Nginx #568 #871, #996,#1482, #1509, #1587
Nginx Application Server	worker process failure	Nginx #912, #822, #192
Nginx Application Server	partial content error	Nginx #683, #1014, #1304, #549, #1357, #1550
MySQL Database Server	slow memory leak	MySQL #56924, #66740, #72885, #83047, #86082, #87501, #852477, #68287, #68514, #68980, #77403
MySQL Database Server	high CPU usage	MySQL #34312, #65778, #76402, #87637
MySQL Database Server	slow database queries	MySQL #15815, #36525, #37633, #67252, #71130, #80989, #86215, #88834

Dependency Path Propagation

First, fault propagation follows the application dependency path: from downstream components to upstream components. A faulty component could affect those components that depend on it. As the faulty component takes longer to process the request, the response time of its upstream components increases. As shown in Figure 3.1, when the component *AP2* is faulty, it also causes the response time of component *LB* to increase. As a result, *LB* exhibits abnormal behavior.

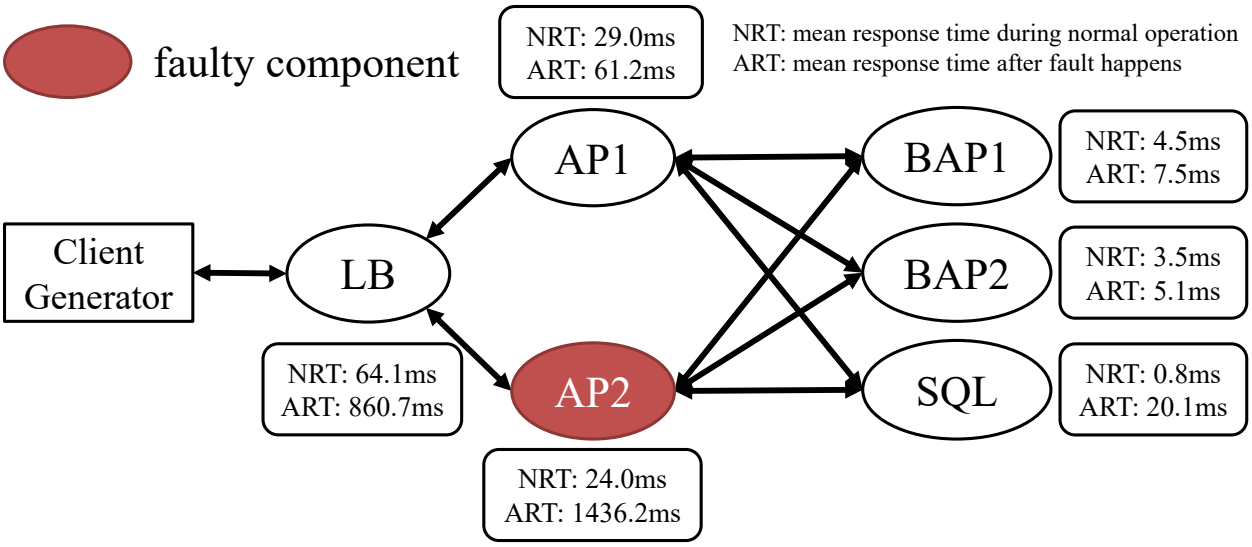


Figure 3.1: An example of anomaly propagation in multi-tier applications

TCP Propagation Path

The TCP congestion control impacts two interacting components. TCP propagation happens due to either of two reasons [42]: (1) The TCP sender does not send fast enough and causes the receiver to be read-blocked; (2) The TCP receiver does not receive or process data fast enough and causes the sender to be write-blocked. When a component is faulty, it impacts not only its upstream components, but also its dependent components through the TCP propagation path.

- The faulty component cannot process requests coming from its upstream components fast enough. Its upstream components are write-blocked as they try to send requests to the faulty component. The waiting time at the upstream components increases.
- The faulty component cannot read responses coming from its dependent components fast enough. Its dependent components are write-blocked as they try to return responses to the faulty component. The waiting time at the dependent components increase.

Chapter 4

Experimental Setup

In this chapter, we show the setup of application benchmarks in our data center. We also introduce evaluation criterion for the performance of our methods in anomaly detection and localization.

4.1 Application Benchmarks

We implement our anomaly detection and localization approaches on top of the KVM platform in our production data center. We conduct extensive experiments using 3 different application benchmarks: (1) MediaWiki [1]; (2) Olio web application [35]; (3) CloudSuite web search benchmark [49] with Apache SolrCloud setup [2].

4.1.1 MediaWiki Application

MediaWiki is a free, open-source web serving application. It is composed of an front web application server (Apache 2.2.15 + PHP 7.0.30) and a backend database server (MySQL 5.5.60). Requests are generated from the Wikipedia request trace between September 2007 and January 2008. We setup the MediaWiki application to characterize more complex modern applications by including two backend computation servers and deploy MediaWiki application as shown in Figure 4.1. Each node is configured with a 2-vCPU, and 4GB memory. Whenever the application servers need to query the database, they call the two backend servers to do a random-dimensional matrix multiplication. After the computation servers return the result, the application servers then query the database. The modified MediaWiki benchmark includes three basic dependency primitives. The two application servers a distributed dependency primitive. The

[1]Mediawiki: `https://www.mediawiki.org`.
[2]SolrCloud: `https://lucene.apache.org/solr/guide/7_4/solrcloud.html`

two computation servers (*bap1*, *bap2*), and the database server together form a composite dependency primitive.

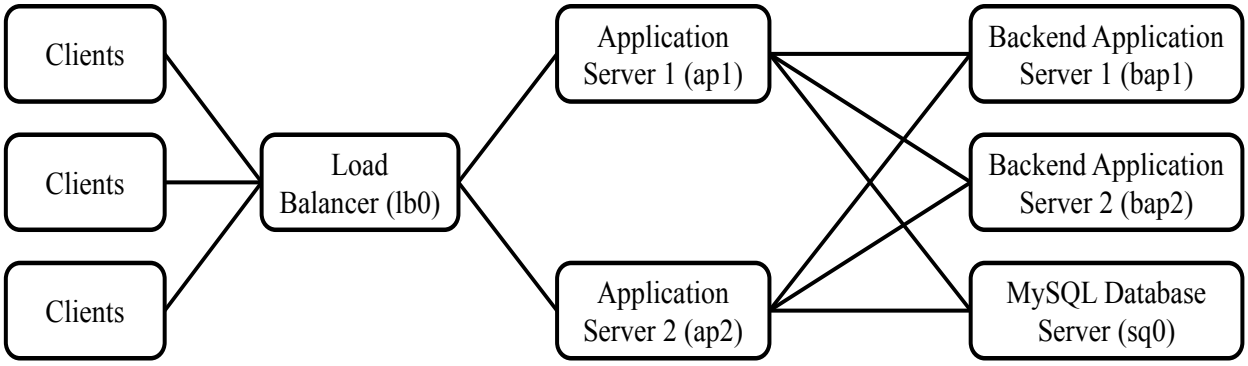


Figure 4.1: The setup of 3-tier MediaWiki benchmark application

4.1.2 Olio Web Application

The Olio web application [35] consists of an application server and a database server. We scale this setup to have more nodes serving user requests. First, we setup an Apache load balancer in front of two application servers. We configure the load balancer to support stickiness on top of cookies. Whenever a request arrives at the load balancer, the load balancer chooses one of the application server to process the request. The two application servers share a common MySQL database. We run the Rain [29] workload generator to produce dynamic workload to Olio web application. The topology setup of Olio application is shown in Figure 4.2.

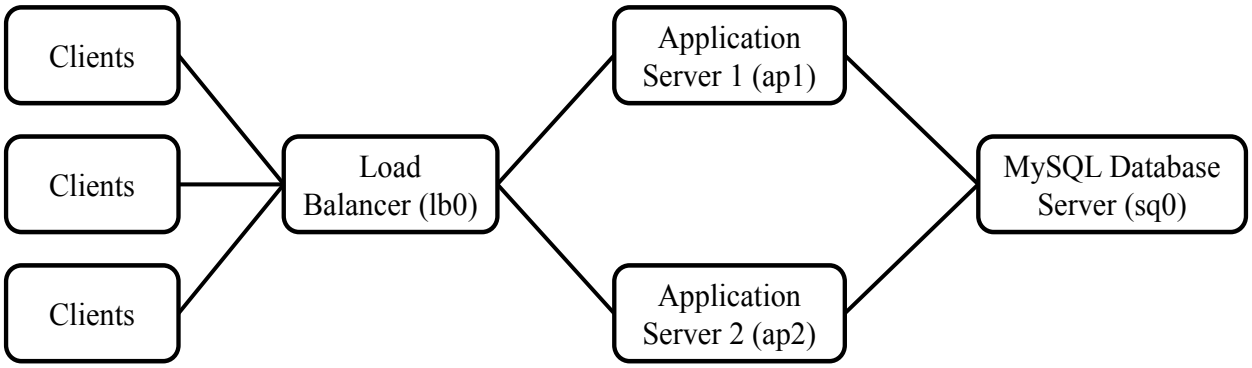


Figure 4.2: The setup of 3-tier Olio web application

4.1.3 CloudSuite Web Search Benchmark

The original CloudSuite web search benchmark [49] contains a single Solr indexing server serving 12GB data. However, a single node setup cannot scale and the response time could be very high under high workload. We deploy the CloudSuite web search application in a SolrCloud consisting of 8 virtual machines. We split the CloudSuite web search index data into 8 different shards (i.e., each shard has about 1.5GB index data) and deploy each shard to a different virtual machine.

With SolrCloud setup, users could select any node in the SolrCloud to send search requests. When a Solr node receives a search request in SolrCloud, the request is further routed to a node containing a shard that is part of the collection being searched. The reception node acts as an aggregator and it creates internal requests to each shard in the collection. It coordinates the responses and issues any subsequent internal requests as needed. The reception node aggregates results returned from other nodes and constructs the final response for the request. We show the SolrCloud setup for the CloudSuite web search application as shown in Figure 4.3.

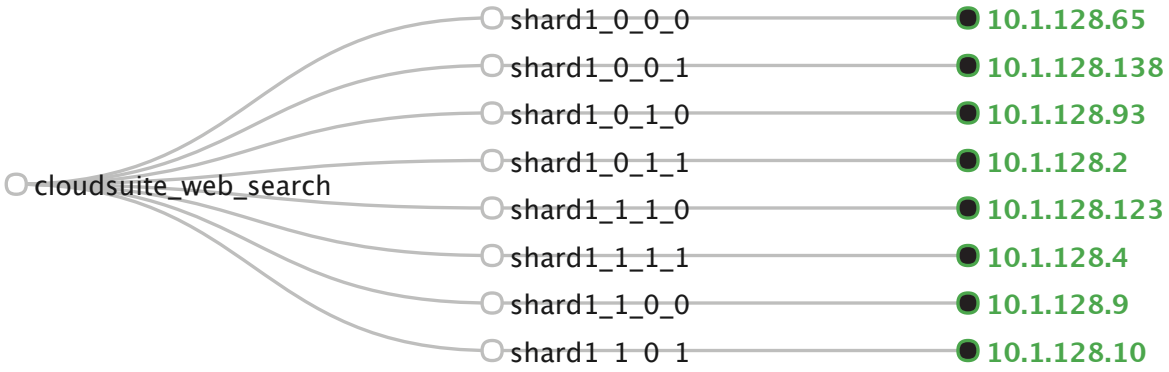


Figure 4.3: The SolrCloud setup of CloudSuite web search application

4.2 Workload Configuration

We evaluate our methods under different workload scenarios. We adjust the number of clients and keep the application CPU usage from 15% to 75% in normal operation. We do not consider cases where the system has less than 15% CPU usage or more than 75% CPU usage. When the system has less than 15%, the workload is too low. The randomness of individual requests incurs noise in the evaluation result. When the system has more than 75% CPU usage, the system tends to have resource contention among requests even in normal operation. We avoid more than 75% CPU usage as it obfuscates the system behavior in normal operation. The web page requests are generated using exponentially distributed think

times with mean of 7 seconds between receiving a response and issuing the next page request in a client session as used in [69]. Each experiment runs for 20 minutes, and the load becomes stable enough after first 3 minutes.

For resource bottleneck faults, we inject CPU, memory, network, and disk resource bottleneck faults in each experiment. Each fault lasts for 60 seconds, and the interval between fault injections is 120 seconds for the system to come back to normal operation. We repeat each experiment with the same configuration for 5 times to avoid the impact of random noise in dynamic system. For common software faults, we inject software bugs in different components of the application.

In evaluating the performance of our methods in anomaly detection, we conduct 4 sets of experiments for each kind of resource bottleneck faults, and software faults. Each set consists of experiments under 15 different workload scenarios. In evaluating the performance of our methods in anomaly localization, 4 faults are injected to different components in each experiment.

4.3 Evaluation Criterion

In this part, we introduce the criterion for evaluating the performance of our methods in anomaly detection and localization of faulty components in Cloud applications.

4.3.1 Anomaly Detection

We use the standard precision and recall metrics to evaluate the performance of our detection methods. Let D_{tp}, D_{fn} and D_{fp} denote the number of true positives (correct detection of an anomaly), false negatives (missing an anomaly), and false positives (detecting an anomaly under normal operation), respectively. The precision and recall metrics are calculated as follows,

$$Precision = \frac{D_{tp}}{D_{tp} + D_{fp}}, Recall = \frac{D_{tp}}{D_{tp} + D_{fn}}$$

We also use the detection latency to characterize how fast our method reacts to the anomaly. It is the elapsed time from when the system shows performance anomaly t_{actual} to when an anomaly alert is triggered $t_{detection}$.

$$Latency = t_{detection} - t_{actual}$$

4.3.2 Anomaly Localization

Given the number of injected faults L_I, the number of triggered anomaly alerts and localization results L_D, the number of true localization results L_T, the number of false localization decisions L_F, and the number

of missed faulty dependent components L_M. We use the following metrics to evaluate the performance of our methods.

1. Precision: $\frac{L_T}{L_D}$. It measures the percentage of correct localization decisions. The false positive rate is given by $\frac{L_F}{L_D}$. It measures the percentage of false localization results.
2. Recall: $1 - \frac{L_T}{L_I}$. It measures the completeness of the method in anomaly localization. The false negative rate is given by $\frac{L_T}{L_D}$. It measures the percentage of missed localization results.

A perfect anomaly localization scheme should achieve 100% localization precision and recall.

4.4 Experimental Setup for Comparative Study

In order to compare our non-intrusive methods with other methods for anomaly detection and localization, we also implement other methods. We describe the setup here.

4.4.1 Anomaly Detection

We compare our non-intrusive methods with resource utilization-based methods. They either rely on thresholding resource usage or discover the correlation among the usage of different resources. Amazon CloudWatch puts thresholds on the resource usage of virtual machines. Whenever the utilization of a resource exceeds the predefined threshold, it triggers an alert to the owner. Other methods correlate the usage of different resources. We choose one of the most representative methods LFD [**?**] as a comparison method for anomaly detection. LFD uses the highest correlation between user-space CPU utilization and other resource utilization metrics to characterize service performance. LFD has agents monitor the resource utilization metrics inside virtual components. The monitored resource utilization metrics include the user-space CPU usage, kernel-space CPU usage, memory utilization, memory page in, memory page out, number of incoming packets and outgoing packets in the network interface, and disk write of virtual machines. We implement LFD algorithm and uses it to detect anomalies at each component.

4.4.2 Anomaly Localization

We compare our non-intrusive methods with another black-box fault localization scheme FChain [38]. FChain has shown a superior performance compared to other methods. FChain is a non-intrusive method that monitors the resource usage outside virtual machines. We compare our methods with FChain for anomaly localization. FChain uses a predictability-based abnormal change point selection scheme to identify the onset time of the abnormal behaviors at different components. We implement FChain algorithm

and use it to pick the earliest abnormal change start time among 8 different metrics as the component's abnormal change start time. The 8 different metrics include the CPU usage, memory utilization, memory page in, memory page out, input network packets, output network packets, disk read, and disk write of virtual machines.

Chapter 5

Dependency Model

Cloud applications are usually deployed in multiple virtual machines (VM). Each VM provides a specific service or function to other VMs within the application. Figure 5.1 shows an example of a multi-tier application. When clients generate a request, the request first arrives at the web server *A*. The web server *A* then sends a request to the application server *B*, which, in turn, calls the database server *C*. Multi-tier applications can be regarded as being composed of nested components. A component's subsystem is the set of other components which the component has to call in order to process the received request. In Figure 5.1, database server *C* does not contain any subsystem. The subsystem of application server *B* is database server *C*. The subsystem of web server *A* consists of both application server *B* and database server *C*. A component may send requests to its subsystem in order to complete the request.

When a component has to wait for the response from its subsystem before local processing, the response time of a request is usually the sum of the time on the component and the time on its subsystem when the execution of local component and its subsystem does not run in parallel.

When a component does local processing in parallel while waiting for a response from its subsystem, the response time would be less than the sum of the time spent on the local component and the time spent on its subsystem.

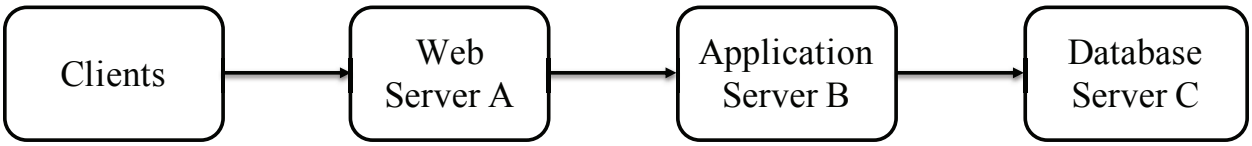


Figure 5.1: An example of a multi-tier application

When a component *A* calls component *B* and has to wait for *B*'s result, we consider *B* as the dependent component of *A*, and *A* as the upstream component of *B*. A component's dependent components are its

subsystem components that have direct interaction with the local component.

A component in multi-tier applications may have no dependent component, a single dependent component, or more than 2 dependent components, We discuss how to determine the subsystem response time of a component given the number of dependent components.

1. When a component has no any dependent component, its subsystem response time is always 0.

2. When a component has only one dependent component, its subsystem response time is the same as that of the dependent component if the communication latency is negligible.

3. When a component has multiple dependent components, the subsystem response time is more complex. VDEP [43] introduces three dependency primitives, and these dependency primitives form a basis to characterize the interaction behavior between the local component and its dependent components. The dependency primitives determine how to compute the subsystem response time given the response time of each dependent component.

5.1 Dependency Model

The dependency model consists of three basic dependency primitives: composite, concurrent, and distributed. It characterizes the complex interaction between a component and its dependent components (at least two dependent components). In this section, we illustrate different dependency primitives and show how the subsystem response time is determined from the response time of each dependent component.

For convenience, we use a simple case where a local component A has two dependent components, B and C to illustrate the difference between different dependency primitives. Table 5.1 gives a list of variables and notations used for mean subsystem response time derivation.

5.1.1 Composite Dependency

A composite dependency primitive describes the interaction behavior at a component where requests are sequentially processed by its dependent components. For example, a request to web server A may involve user authentication in server B before passing transaction requests to the transaction server C as shown in figure 5.2. A request arrived at A causes a various number of requests to different dependent components. The subsystem response time of a request is the sum of time spent on dependent components.

Table 5.1: Variable notations for response time characterization

Variables	Notation
ΔT	the length of the sampling interval for measuring the mean response time
$\overline{R_A}(t)$	the mean response time of component A during the interval $(t-\Delta T, t)$
$\overline{S_A}(t)$	the mean subsystem response time of component A during the interval $(t-\Delta T, t)$
$\overline{P_A}(t)$	the mean service time of component A during the interval $(t-\Delta T, t)$
$n_A(t)$	the number of requests arrived at component A during the interval $(t-\Delta T, t)$
b_i	the number of requests to component B caused by ith request at component A
c_j	the number of requests to component C caused by jth request at component A
$n_{AB}(t)$	the number of requests from component A to B during the interval $(t-\Delta T, t)$
$\overline{R_{AB}}(t)$	the mean response time of component B perceived by agent of A in interval $(t-\Delta T, t)$
R_{Bj}	the response time of jth request at component B during the interval $(t-\Delta T, t)$
$n_{AC}(t)$	the number of requests from component A to C during the interval $(t-\Delta T, t)$
$\overline{R_{AC}}(t)$	the mean response time of component C perceived by agent of A in interval $(t-\Delta T, t)$
R_{Ck}	the response time of kth request at component C during the interval $(t-\Delta T, t)$

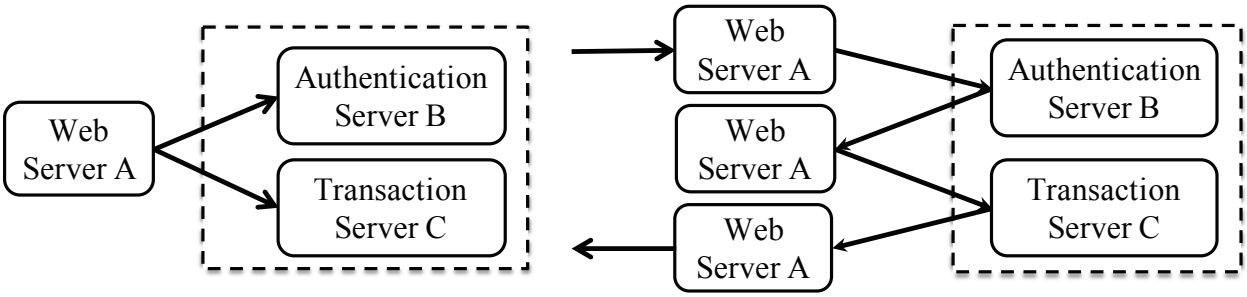


Figure 5.2: The composite dependency primitive

$$
\begin{aligned}
\overline{S_A}(t) &= \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \left(\left(\sum_{j=1}^{b_i} R_{B_j} + \sum_{k=1}^{c_i} R_{C_k} \right) \right) \\
&= \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \left(\sum_{j=1}^{b_i} R_{B_j} + \sum_{k=1}^{c_i} R_{C_k} \right) \\
&= \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \sum_{j=1}^{b_i} R_{B_j} + \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \sum_{k=1}^{c_i} R_{C_k} \\
&= \frac{n_{AB}(t)}{n_A(t)} \frac{1}{n_{AB}(t)} \sum_{i=1}^{n_A(t)} \sum_{j=1}^{b_i} R_{B_j} + \frac{n_{AC}(t)}{n_A(t)} \frac{1}{n_{AC}(t)} \sum_{i=1}^{n_A(t)} \sum_{k=1}^{c_i} R_{C_k} \\
&= \frac{n_{AB}(t)}{n_A(t)} * \overline{R_B}(t) + \frac{n_{AC}(t)}{n_A(t)} * \overline{R_C}(t)
\end{aligned}
$$

VDEP [43] has a mutual exclusion definition of composite dependency primitive: when B and C form a composite dependency primitive, A has to wait for B to finish its execution before sending requests to C. In this work, we adjust more flexible definition of composite primitive since component execution are

becoming more parallel in Cloud applications for better performance, as analyzed in Mystery [39]. We allow an overlap (but not concurrency) between the execution of B and C. On the other hand, we use a non-intrusive method and it is hard to tell whether B has finished execution or not before C's execution. As long as there exists a causal order between B and C, we consider that they form a composite dependency primitive. In this case, the subsystem response time is smaller than the sum of response time of composite dependent components.

5.1.2 Concurrent Dependency

A concurrent dependency primitive describes the interaction behavior at a component where requests are concurrently processed by its dependent components. For example, generating a web page in web server A involves concurrent requests to the database server B and the content server C as shown in figure 5.3. The subsystem response time of a request with the concurrent dependency primitive is the maximum time taken by each dependent component to process the request.

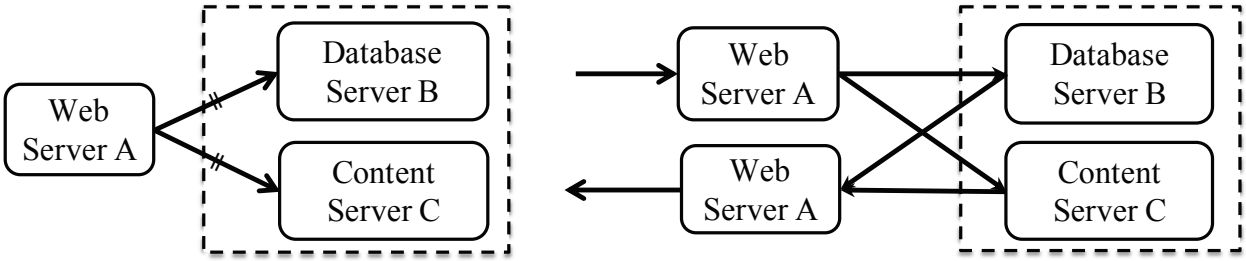


Figure 5.3: The concurrent dependency primitive

$$
\begin{aligned}
\overline{S_A}(t) &= \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \left(\max \left(\sum_{j=1}^{b_i} R_{B_j}, \sum_{k=1}^{c_i} R_{C_k} \right) \right) \\
&= \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \max \left(\sum_{j=1}^{b_i} R_{B_j}, \sum_{k=1}^{c_i} R_{C_k} \right) \\
&= \max \left(\frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \sum_{j=1}^{b_i} R_{B_j}, \frac{1}{n_A(t)} \sum_{i=1}^{n_A(t)} \sum_{k=1}^{c_i} R_{C_k} \right) \\
&= \max \left(\frac{n_{AB}(t)}{n_A(t)} \frac{1}{n_{AB}(t)} \sum_{i=1}^{n_A(t)} \sum_{j=1}^{b_i} R_{B_j}, \frac{n_{AC}(t)}{n_A(t)} \frac{1}{n_{AC}(t)} \sum_{i=1}^{n_A(t)} \sum_{k=1}^{c_i} R_{C_k} \right) \\
&= \max \left(\frac{n_{AB}(t)}{n_A(t)} * \overline{R_B}(t), \frac{n_{AC}(t)}{n_A(t)} * \overline{R_C}(t) \right)
\end{aligned}
$$

5.1.3 Distributed Dependency

A distributed dependency describes the interaction behavior at a component where each request chooses one of its dependent components to process the request. When a request arrives at the load balancer A, A

distributes the request to either B or C as shown in Figure 5.4. As the number of possible execution paths is finite, The subsystem response time of a request is the probabilistic summation of the time spent on all dependent components.

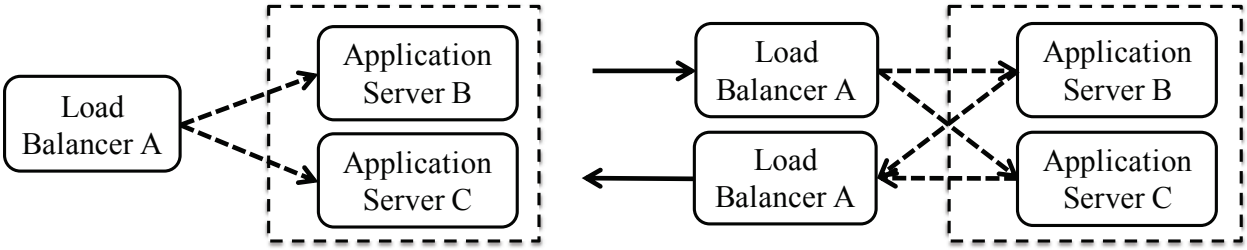


Figure 5.4: The distributed dependency primitive

$$\begin{aligned}
\overline{S_A}(t) &= \frac{1}{n_A(t)} \left(\sum_{i=1}^{n_{AB}(t)} R_{B_i} + \sum_{j=1}^{n_{AC}(t)} R_{C_j} \right) \\
&= \frac{1}{n_A(t)} \sum_{i=1}^{n_{AB}(t)} R_{B_i} + \frac{1}{n_A(t)} \sum_{j=1}^{n_{AC}(t)} R_{C_j} \\
&= \frac{1}{n_A(t)} \sum_{i=1}^{n_{AB}(t)} R_{B_i} + \frac{1}{n_A(t)} \sum_{j=1}^{n_{AC}(t)} R_{C_j} \\
&= \frac{n_{AB}(t)}{n_A(t)} \frac{1}{n_{AB}(t)} \sum_{i=1}^{n_A(t)} R_{B_i} + \frac{n_{AC}(t)}{n_A(t)} \frac{1}{n_{AC}(t)} \sum_{j=1}^{n_{AC}(t)} R_{C_j} \\
&= \frac{n_{AB}(t)}{n_A(t)} * \overline{R_B}(t) + \frac{n_{AC}(t)}{n_A(t)} * \overline{R_C}(t)
\end{aligned}$$

For components where local processing does not run in parallel with subsystem execution. The local service time can be derived given the mean response time and the mean subsystem response time.

$$\overline{P_A}(t) = \overline{R_A}(t) - \overline{S_A}(t)$$

But this does not hold when the local processing and subsystem overlaps.

5.2 Dependency Extraction Analysis

The dependency model is useful for characterizing the interaction behavior between a component and its dependent components. We use a black box method to extract the dependency primitive. The method requires to monitor the following metrics for each component using a monitoring interval ΔT_d.

- *Input request flow*: the number of incoming request data packets arrived at the component.
- *Output request flow*: the number of outgoing request data packets from the local component to each dependent components.

Table 5.2: Collected system metrics for dependency extraction

Variables	Notation
ΔT_d	the length of the sampling interval for dependency extraction
$X_A(t)$	the number of request data packets arrived at component A in the interval $(t - \Delta T_d, t)$
$X_{AB}(t)$	the number of request data packets from component A to B in the interval $(t - \Delta T_d, t)$
$X_{AC}(t)$	the number of request data packets from component A to C in the interval $(t - \Delta T_d, t)$

The variables and notations used for dependency model extraction are listed in Table 5.2.

5.2.1 Distributed Dependency Extraction

For distributed dependency, requests arrived at the local component are distributed to its dependent components for further execution. In real applications, the load balancer is the most common component that has distributed dependency primitive. In a load balancer, the number of arrived requests approximately equals the sum of the number of requests from the load balancer to its dependent components. This observation is also true in terms of the number of data packets: the number of request data packets arrived at the load balancer approximately equals the sum of the number of request data packets from the local component to its dependent components.

For components that do not have exactly the same behavior as the load balancer, it requires more general method to find out the distributed dependency primitive condition. But the relationship between the number of request data packets arrived at the component and the sum of the number of request data packets from the local component to its dependent components is stable and approximately constant C.

$$\frac{X_{AB}(t) + X_{AC}(t)}{X_A(t)} \approx C$$

In this work, we focus on the distributed dependency extraction for a component that has similar behavior as a common load balancer.

Sampling Interval

The sampling interval ΔT_d is important for the extraction of the distributed dependency. If ΔT_d is too short, it may happen that requests arrived at the local component within $(t - \Delta T_d, t)$ cannot be distributed to dependent components within the same sampling interval. In this case, $X_A(t)$ will differ significantly from $X_{AB}(t) + X_{AC}(t)$. The sampling interval should be at least as large as the maximum time for the local component to send a request to any of its dependent components. We define the relative error between the local input request flow and the sum of output request flow to all dependent components.

$$RE_X(t) = \frac{X_{AB}(t) + X_{AC}(t) - X_A(t)}{X_A(t)} RE_Y(t) = \frac{Y_{AB}(t) + Y_{AC}(t) - Y_A(t)}{Y_A(t)}$$

We show the impact of sampling interval on distributed dependency extraction in our experiments.

5.2.2 Composite or Concurrent Dependency Extraction

For non-distributed dependency, requests arrived at a local component cause requests to the dependent components in either sequential or concurrent way based on the implemented application logic. We analyze the difference between the composite and concurrent dependency primitives.

- In composite dependency as shown in Figure 5.2, A sends requests sequentially to B and C. There is a noticeable delay between the outgoing request flow to B and the outgoing request flow to C.
- In concurrent dependency as shown in Figure 5.3, A sends requests to both B and C concurrently. There is no delay between the outgoing request flow to B and the outgoing request flow to C.

We use the lag correlation between the number of request data packets from A to dependent components B and C to distinguish a composite dependency from a concurrent dependency. The lag correlation between time series $X_{AB}(t)$ and $X_{AC}(t)$ of length L is defined as

$$\rho_{B,C}(k) = \frac{\sum_0^{L-1} \left(X_{AB}(t) - \overline{X_{AB}(t)}\right)\left(X_{AC}(t+k) - \overline{X_{AC}(t)}\right)}{\sqrt{\sum_0^{L-1} \left(X_{AB}(t) - \overline{X_{AB}(t)}\right)^2}\sqrt{\sum_0^{L-1} \left(X_{AC}(t+k) - \overline{X_{AC}(t)}\right)^2}}$$

We pick $maxK$ as the lag value that achieves the maximum correlation between the request flows to two dependent components.

$$maxK = \arg\max_k \rho(k)$$

If $maxK = 0$, B and C execute concurrently. Otherwise, B and C execute sequentially. If $maxK > 0$, B executes before C. If $maxK < 0$, C executes before B.

Sampling Interval

The sampling interval ΔT_d is crucial for differentiating the composite dependency primitive from the concurrent dependency primitive. If ΔT_d is too large, it may happen that the outgoing request flow from A to its composite dependent components B and C, are observed within the same sampling interval. In this case, we may falsely identify the composite dependency primitive as the concurrent dependency primitive. The sampling interval should be smaller than the minimum delay between the outgoing request flows from the local component to its dependent components. We show the importance of the sampling interval in order to correctly identify the composite dependency extraction in our experiments.

5.3 Distributed View of Dependency

Previous methods rely on the dependency graph for the entire application in order to locate the faulty component. In this work, we do not need to build the dependency graph for the whole application. We employ a distributed method where agents distributed on physical hosts monitor local virtual machines and their dependent components. We show a global view of dependency for the whole application in Figure 5.5 and a distributed view of dependency at each agent in Figure 5.6.

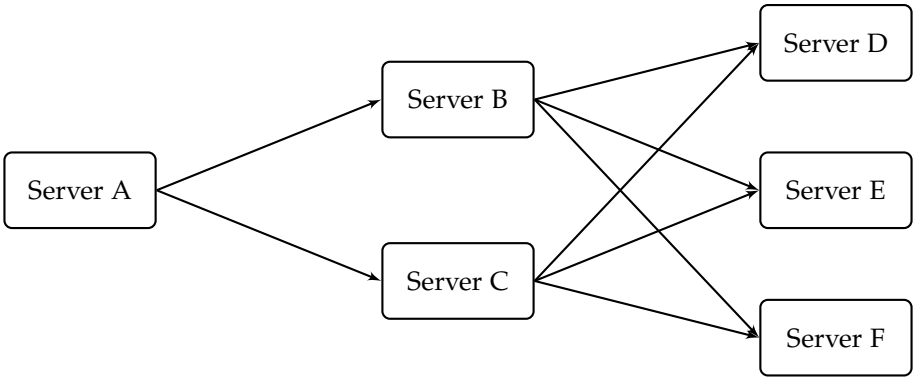
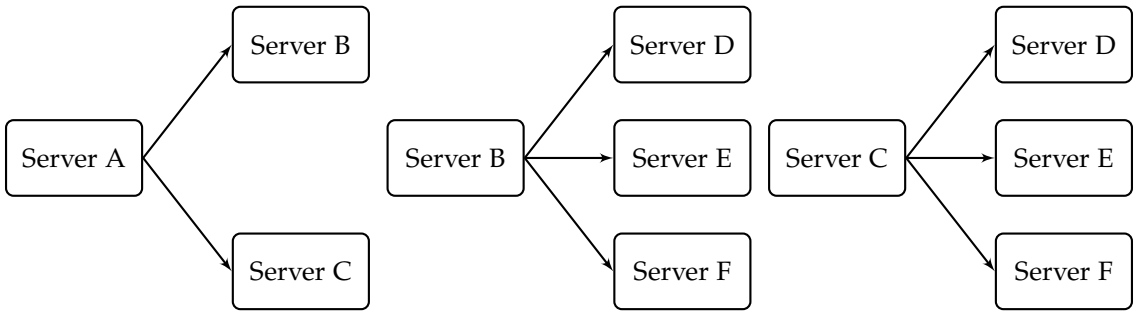


Figure 5.5: A centralized view of dependency graph for an application



(a) The local dependency view at A (b) The local dependency view at B (c) The local dependency view at C

Figure 5.6: A distributed view of dependency at each component of the application

5.3.1 Dependency Extraction Algorithm

Distributed dependency graph requires us to find out the dependency primitive locally. The global dependency graph is constructed with a combination of local partial dependency. We construct the dependency primitive for each component using Algorithm 1. The dependency extraction algorithm is applied for each virtual component (VM or container) for extracting local dependency from individual component's perspective. By combining the extracted dependency primitive from individual components, a global dependency graph can be built for the whole application. It also ensures that our dependency primitive

Algorithm 1: Dependency extraction

input : Given a local component A and its m dependent components $S = \{A_{d1}, A_{d2}, \cdots, A_{dm}\}$;

- A proper sampling interval ΔT_d
- The number of incoming request data packets at component A: $X_A(t)$
- The number of request data packets from component A to each of its dependent components: $\{X_{A_1}(t), X_{A_2}(t), \cdots, X_{A_m}(t)\}$

output: The execution sequence of component A's dependent components

if $X_A(t) \approx X_{A_1}(t) + X_{A_2}(t) + \cdots + X_{A_m}(t)$ **then**
  A's dependent components form a distributed primitive;
else
  foreach *i from* 1 *to* m **do**
    foreach *j from* 1 *to* m **do**
      foreach *k from* 1 *to* $\frac{L}{2}$ **do**
        Compute the correlation between $X_{A_i}(t)$ and $X_{A_j}(t)$ at a lag k:;

$$\rho_{i,j}(k) = \frac{\sum_0^{L-1} \left(X_{A_i}(t) - \overline{X_{A_i}(t)}\right)\left(X_{A_j}(t+k) - \overline{X_{A_j}(t)}\right)}{\sqrt{\sum_0^{L-1} \left(X_{A_i}(t) - \overline{X_{A_i}(t)}\right)^2}\sqrt{\sum_0^{L-1} \left(X_{A_j}(t+k) - \overline{X_{A_j}(t)}\right)^2}}$$

      end
      Find the lag that achieves the maximum correlation: $maxK = \arg\max_k \rho_{i,j}(k)$;
      if $maxK = 0$ **then**
        A's dependent components A_i and A_j form a concurrent dependency primitive.
      else
        A's dependent components A_i and A_j form a composite dependency primitive;
        if $maxK > 0$ **then**
          A_i executes before A_j
        else
          A_i executes after A_j
        end
      end
    end
  end
end

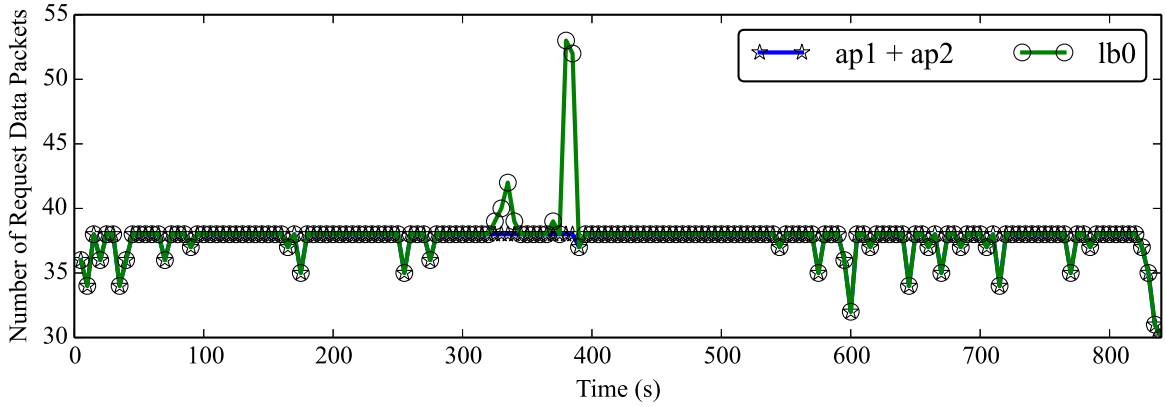
algorithm is scalable to large-scale applications as each component runs independently with limited set of dependent components.

5.4 Experimental Evaluation

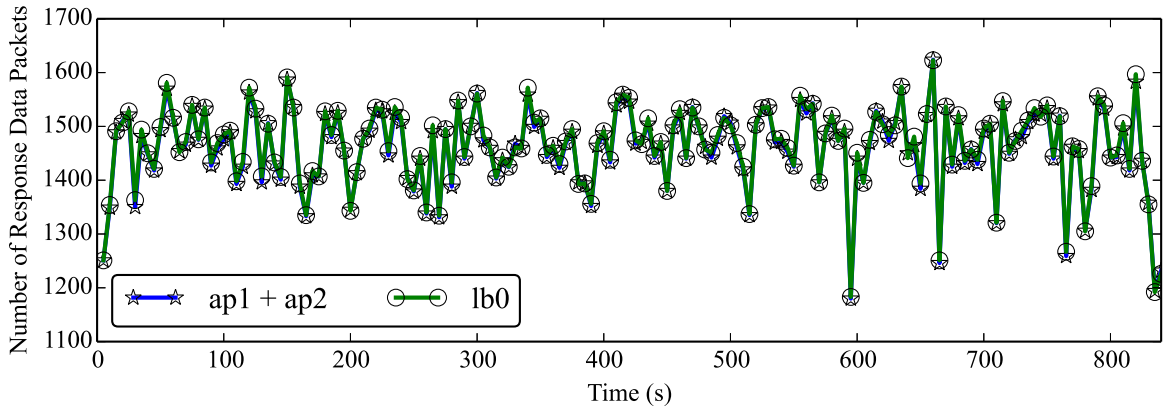
In this section, we use the black-box dependency extraction algorithm for dependency extraction in deployed applications in our cluster. The detailed experimental setup is described in Section 4.1.

5.4.1 MediaWiki Application

In the MediaWiki application, the dependency analysis is performed at the load balancer (*lb*), application server 1 (*ap1*), and application server 2 (*ap2*). Figure 5.7 show the relationship of the request flow and the response flow at the load balancer. The sum of the number of incoming request data packets going to the two application servers approximately equals the number of requests arriving at the load balancer. The sum of the number of outgoing response data packets returned by two application servers approximately equals the number of outgoing response data packets from the load balancer.



(a) The number of request data packets at the load balancer



(b) The number of response data packets at the load balancer

Figure 5.7: Distributed dependency primitive analysis at the load balancer in MediaWiki application

The number of request data packets arrived at the application server does not equal the sum of the number of request data packets from the application server to its dependent components. The distributed dependency primitive is not true for both application servers. We use the lag correlation analysis to extract the execution sequence of the computation servers (*ap1*, *ap2*) and database *sq0*. With the lag value of 1, the correlation between two computation servers reaches the maximum as shown in Figure 5.8. It accurately

models that the two computation servers execute sequentially and *bap1* executes before *bap2*. In Figure 5.8, the correlation reaches the maximum between the computation server *bap1* and database *sq0* with a lag value of 2. It accurately models the fact that the database executes after *bap1* and *bap2*. The lag correlation result is similar for component *ap2*.

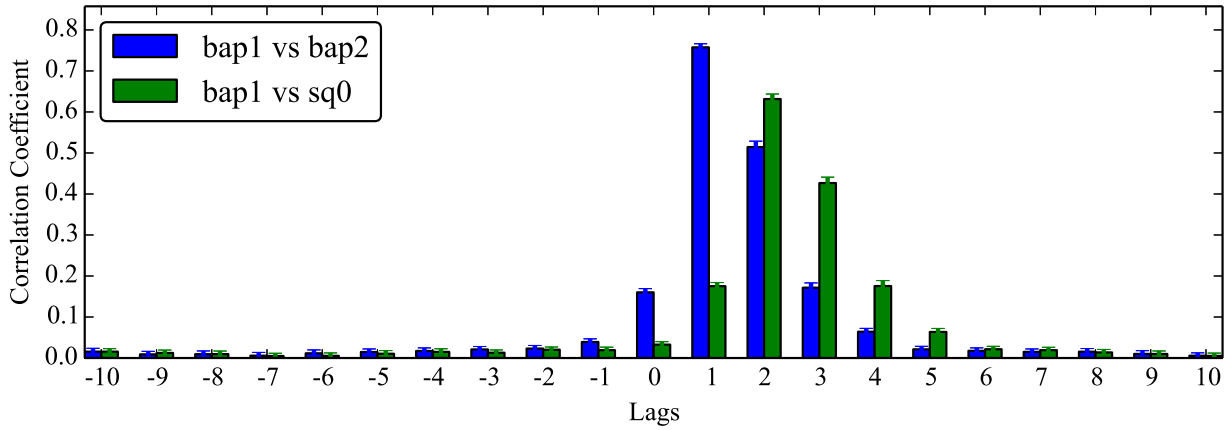


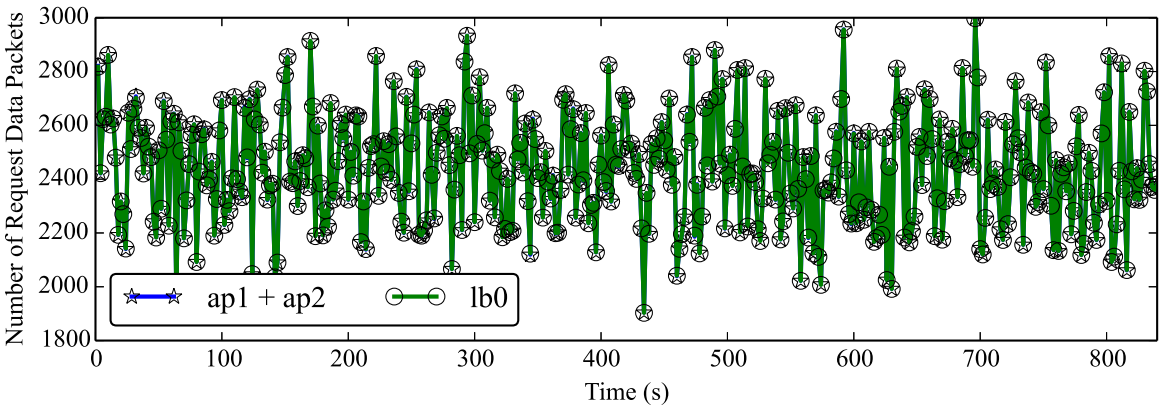
Figure 5.8: The dependency primitive analysis in MediaWiki application

5.4.2 Olio Web Application

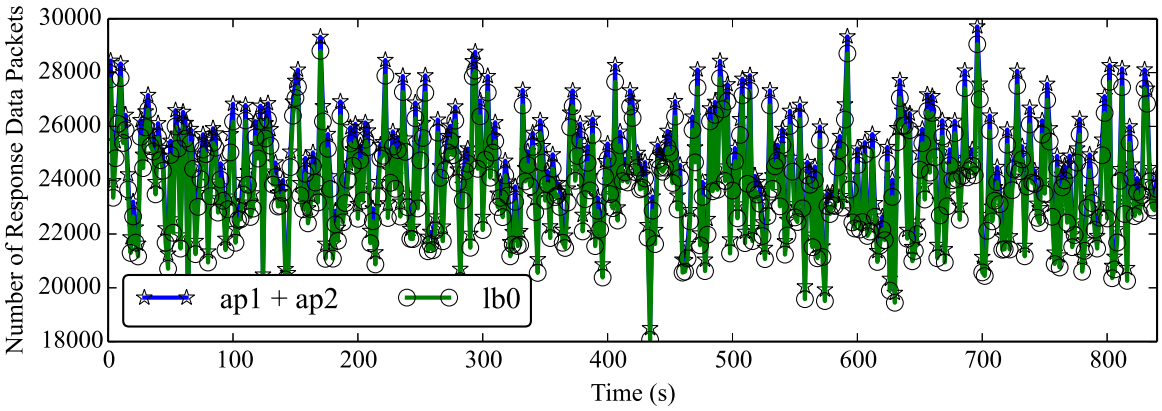
In the Olio application, the load balancer is configured to support session stickiness. When a request is directed to a backend server, then all following requests from the same user should be proxied to the same backend server. The sum of the number of request data packets from *lb0* to all its dependent components (*ap1* and *ap2*), and the number of request data packets arrived at *lb0* are shown in Figure 5.9a. The sum of the number of request data packets from *lb0* to its dependent components (i.e., *ap1*+*ap2*) approximately equals the number of request data packets arrived at *lb0*. Figure 5.9a shows that the sum of the number of response data packets from all its dependent components (i.e., *ap1*+*ap2*) approximately equals the number of response data packets departed from *lb0*. It validates the fact that the two application servers (*ap1* and *ap2*) form a distributed dependency primitive with the load balancer *lb0* in the Olio application.

5.4.3 CloudSuite Web Search Application

In SolrCloud application, we select a reception node for receiving user requests. For convenience, we label the reception node as *solr0*, and other nodes sequentially as *solr1* to *solr7*. To evaluate the dependency primitive at *solr0*, the number of request data packets from *solr0* to each of its dependent components is collected in every 5 milliseconds. We perform lag correlation analysis between "solr1" and all other dependent components. The result is shown in Figure 5.10, and the correlation achieves the maximum



(a) The relationship between request flows in the distributed dependency primitive



(b) The relationship between response flows in the distributed dependency primitive

Figure 5.9: The relationship of request and response flow at the load balancer in Olio application

with a lag value of 0. It means the latency among requests from *solr0* to its dependent components are always within 5ms. Since the mean response time of dependent components are larger than 5ms, all dependent components (*solr1*, *solr2*, *solr3*, *solr4*, *solr5*, *solr6*, *solr7*) form a concurrent dependency primitive with *solr0* in SolrCloud. It validates that the SolrCloud dependency primitive is concurrent as the reception node concurrently sends requests to other nodes and waits for their responses.

5.4.4 Distributed Dependency Extraction: Impact of Sampling Interval

When a local component and its dependent components form a distributed dependency primitive, it is observed that the number of request data packets arrived at the local component is almost equal to the number of request data packets issued from the local component to its dependent components. This is true if the sampling interval we used to collect the number of packets is large enough. In a dynamic system, it takes some time for a request to go to its dependent components after arriving at the local

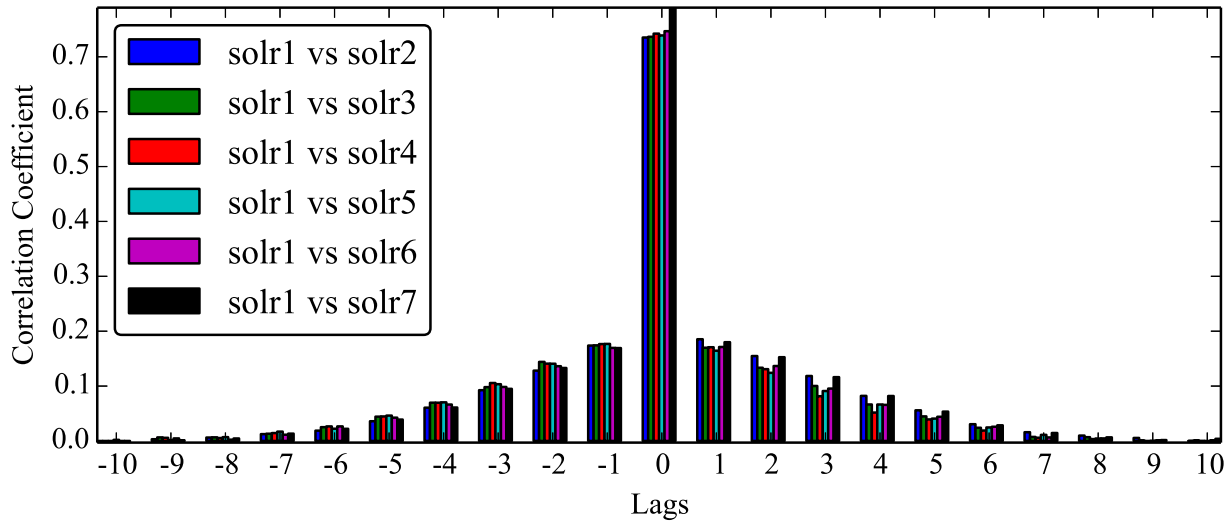


Figure 5.10: The lag correlation for the dependency primitive analysis in SolrCloud application

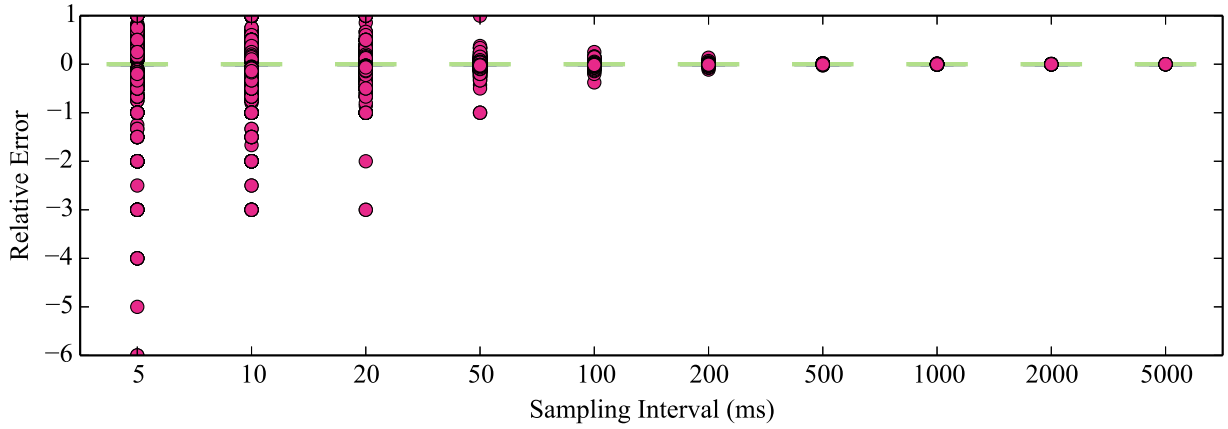


Figure 5.11: The impact of sampling interval on the distributed dependency primitive extraction

component. We may not always observe they fall into the same interval if the sampling interval is too small. We show the relative error between the input request flow to *lb0* and the sum of output request flow from *lb0* to all dependent components (*ap1* and *ap2*) using different sampling intervals in Figure 5.11. Figure 5.12 shows the mean relative error with 95% confidence interval. From the boxplot of the relative error, we learn that the distributed analysis is more accurate as we use a larger sampling interval. When the sampling interval is larger than the local response time, the relative error is almost negligible.

5.4.5 Composite and Concurrent Dependency Extraction: Impact of the Sampling Interval

As we know, the lag correlation analysis is sensitive to the size of the sampling interval. We show how the *maxK* changes as the sampling interval increases in Figure 5.13 for the composite dependency. The

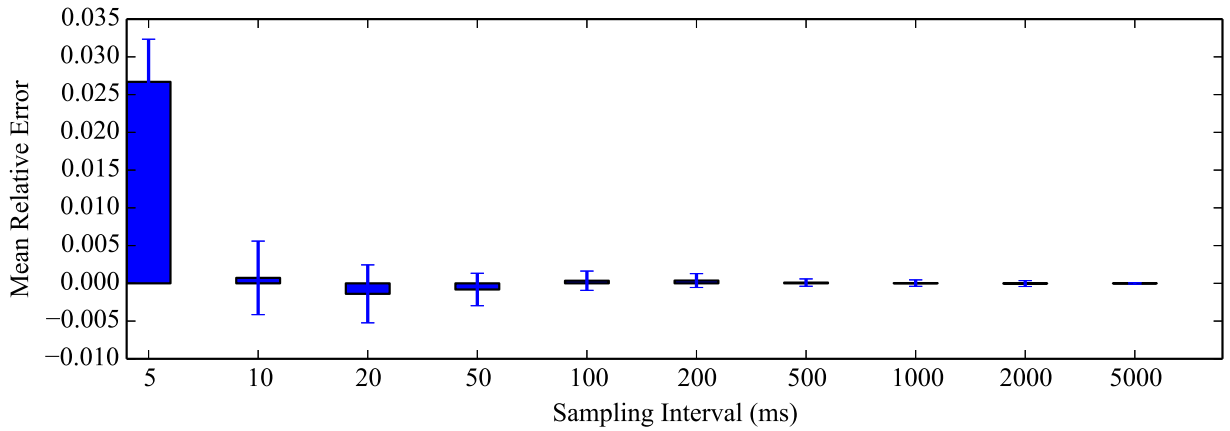


Figure 5.12: The impact of sampling interval on the distributed dependency primitive extraction

composite dependency extraction is sensitive to the sampling interval. As the sampling interval gets larger, the *maxK* gets smaller. The sequential execution among all dependent components of *ap1* if the sampling interval is larger than 50ms. In general, it is easy to extract the composite dependency primitive correctly with a sampling interval of 5ms. The impact of the sampling interval for dependency primitive analysis is similar as *ap2*.

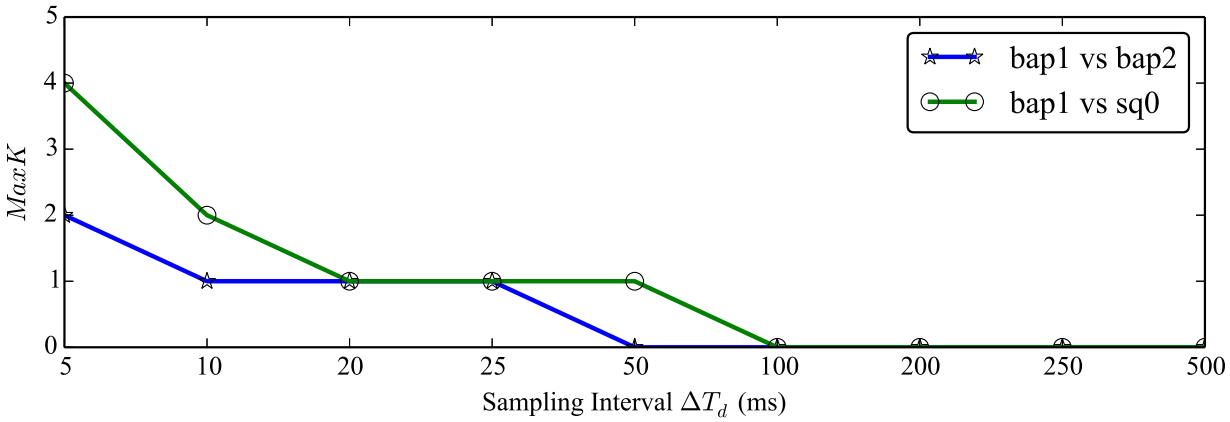


Figure 5.13: The impact of sampling interval on composite dependency extraction

5.5 Discussion

In this chapter, we derive the mean response time characteristics in different dependency primitive and evaluate our black-box dependency extraction method through practical applications. Experimental results show that our method could correctly identify different dependency primitives.

The sampling interval is an critical parameter for dependency extraction. To extract the distributed

dependency correctly, the sampling interval is better to be larger, If the a component does not have the distributed dependency primitive, a sampling interval small enough is important to identify whether the dependency primitive is composite or concurrent.

Chapter 6

DMADL: Dependency Model-based Response Time Analysis for Anomaly Detection and Localization

In this chapter, we propose DMADL, a dependency model-based anomaly detection and localization system for IaaS Cloud applications. DMADL has agents distributed on physical hosts. The agents do not require any knowledge from the underlying operating system or application domain in virtual components. DMADL monitors network traffic and estimates mean response time through the arrival and departure pattern of data packets. Any abnormal changes in the dependency model or the estimated response time trigger anomaly alerts. DMADL determines the impact of individual dependent component in the local response time of each component. When a component is faulty, its response time should dominate the response time of any component that has the faulty component as its subsystem component. DMADL pinpoints the faulty component by analyzing the impact of each component along the dependency path in a distributed fashion.

6.1 Mean Response Time Estimation

The response time of a component is measured from the time when a request arrives at the component to the time when the corresponding response leaves the component. Unexpected long response times degrade users' quality of experience. It is challenging to accurately monitor the response time from outside of a component without intrusive operation.

There are two main methods for obtaining the response time: deep packet inspection or server log analysis. The deep inspection of packets' payload requires either understanding of application protocols

Table 6.1: Metrics collected by DMADL for anomaly detection

Variables	Notation
$R(t)$	the mean response time of the service
$RQ(t)$	the number of requests arrived at the service
$RP(t)$	the number of responses departed from the service
$X(t)$	the number of request data packets arrived at the service
$Y(t)$	the number of response data packets departed from the service
$D(t)$	the number of request timeouts for the service

or decryption of encrypted application protocols such as HTTPS. The analysis of server logs requires intrusive access to the service inside virtual components and extra configuration for logging the response time. DMADL assumes that the communication between each pair of components are through requests and responses in Cloud applications. DMADL assumes serial operation without pipelining operations within each TCP connection. DMADL estimates the response time of a request as the elapsed time from when the first request data packet arrives at the service to when the last response data packet leaves the service. At each component, DMADL estimates the mean response time within each monitoring interval $(t - \Delta T, t)$ as a real-time indicator of the service performance including both the local component and its subsystem.

6.1.1 Monitored Metrics

In each monitoring interval $(t - \Delta T, t)$, DMADL agents distributed on physical hosts collect the metrics in Table 6.1 for each local service and its dependent services.

Figure 6.1 shows an example of how DMADL agents estimate the mean response time using data packets in a single TCP flow. The first request data packet arrives at A's local service at time t_0 during the monitoring interval $(t - \Delta T, t)$. Within the same monitoring interval $(t - \Delta T, t)$, the DMADL agent obtains the response time as the time difference between the timestamp of the last outgoing data packet t_1 and the timestamp of the first incoming data packet t_0: $t_1 - t_0$. In the next monitoring interval $(t, t + \Delta T)$, the DMADL agent observes more outgoing data packets still for the same request within the TCP flow. It updates the response time for the request arrived at time t_0 as $t_2 - t_0$. DMADL estimates the mean response time in real time using Algorithm 2.

6.1.2 Accuracy Analysis

When the last observed outgoing data packet within a monitoring interval is not the actual last data packet of a response, the estimated response time is smaller than the real response time. For example, the response time within interval $(t - \Delta T, t)$ is $t_1 - t_0$ in Figure 6.1. It is smaller than the real response time

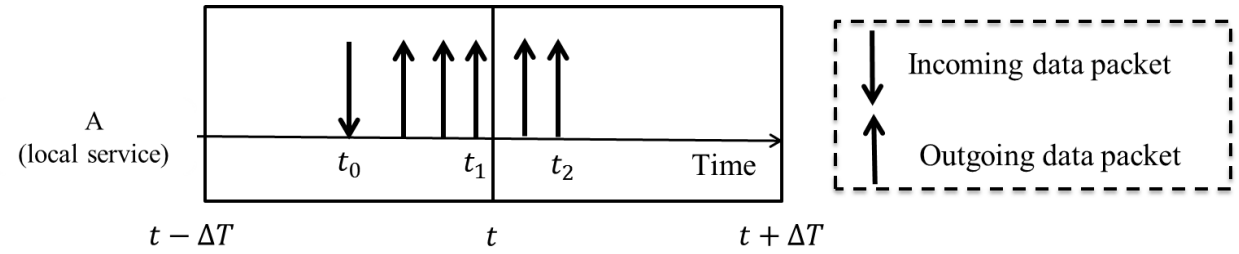


Figure 6.1: The response time estimation scheme of DMADL using data packets in each TCP flow

Algorithm 2: DMADL mean response time estimation

input : Stream of request data packets arrived at service A, and response data packets departed from service A in a monitoring interval $(t - \Delta T, t)$, the flow table $\mathcal{F}_{\mathcal{A}}$.

output: Estimated Metrics: mean response time $R_A(t)$, the number of requests $RQ_A(t)$, the number of responses $RP_A(t)$, the number of request data packets $X_A(t)$, the number of response data packets $Y_A(t)$.

Initialization $R_A(t) = 0, RQ_A(t) = 0, RP_A(t) = 0, X_A(t) = 0, Y_A(t) = 0, D_A(t) = 0$;
for *each data packet pkt arriving at A or departing from A in* $(t - \Delta T, t)$ **do**
  if *pkt is arriving at service A* **then**
    if $flow(pkt) \notin \mathcal{F}_{\mathcal{A}}$ **then**
      $X_A(t) + +$;
      Insert $flow(pkt)$ into $\mathcal{F}_{\mathcal{A}}$;
      $flow(pkt).t_{req} = pkt.ts$;
      $RQ_A(t) = RQ_A(t) + 1$;
      $flow(pkt).flag = 0$;
    end
  else
    if $flow(pkt) \in \mathcal{F}_{\mathcal{A}}$ **then**
      $Y_A(t) + +$;
      $flow(pkt).t_{rep} = pkt.ts$;
      if $flow(pkt).flag == 0$ **then**
        $flow(pkt).flag = 1$;
        $R_A(t) = R_A(t) + (pkt.ts - flow(pkt).t_{req})$;
        $RP_A(t) = RP_A(t) + 1$;
      else
        $R_A(t) = R_A(t) + (pkt.ts - flow(pkt).t_{rep})$;
      end
    end
  end
end
$R_A(t) = \frac{R_A(t)}{RP_A(t)}$;

$t_2 - t_0$. The mean response time estimated by DMADL agents is accurate when the first data packet and the last data packet of the same response are always observed in the same monitoring interval.

In normal operation, the response time RT would be always within SLA: $RT < SLA$. We denote the delay between the first data packet and the last data packet of a response as $\tau < RT < SLA$. The first data

packet of a response could occur at any time t_f during the monitoring interval $(t - \Delta T, t)$, the indicator function of whether we can observe both the first data packet and the last data packet of a response within the same interval is given as follows.

$$p = \begin{cases} 1, & t_f \in (t - \Delta T, t - \tau) \\ 0, & t_f \in (t - \tau, t) \end{cases}$$

Suppose the arrival time t_f of the first response data packet in the interval $(t - \Delta T, t)$ follows a distribution $f(q_t)$. τ follows a distribution $g(\tau_t)$ with a cumulative distribution function (CDF) $G(\tau_t)$. In a normal operation, $\tau_{max} < SLA$, and in abnormal operation τ_{max} increases and even violates the SLA (i.e. $\tau_{max} > SLA$). The probability that the first data packet and the last data packet of responses are observed within the same monitoring interval $(t - \Delta T, t)$ is,

$$\begin{aligned} p_{same}(t) &= \int_0^{\Delta T} P(q_t = t - x) * P(\tau_t \leq x) \\ &= \int_0^{\Delta T} f(t - x) * G(x) dx \end{aligned}$$

We assume the arrival time of the first response data packet within a monitoring interval is distributed according to the uniform distribution.

$$\begin{aligned} f(q_t) &= \frac{1}{\Delta T} \\ p_{same}(t) &= \frac{1}{\Delta T} \int_0^{\Delta T} G(x) dx > \frac{\Delta T - \tau_{\max}}{\Delta T} \end{aligned}$$

The probability $p_{same}(t)$ is exactly the area covered by the area covered by $G(x)$ within ΔT. Its area decreases when the component has performance anomalies. As we pick the monitoring interval $\Delta T \gg \tau_{max}$, the probability $p_{same}(t) > \frac{\Delta T - \tau_{max}}{\Delta T}$. Under this condition, the estimated mean response time by DMADL agents is accurate. In our system, we pick $\Delta T = 10 * \tau_{max}$. The estimated response times for more than 90% of responses are accurate.

6.1.3 Theoretical Overhead Analysis

We analyze the memory overhead of each DMADL agent in estimation of the mean response time. For each TCP flow, its data structure takes 288 bits. If there are $100,000$ TCP flows for a component service, the required memory size is about 3MB.

As the number of stored TCP flows increases in run time, DMADL agents periodically clean the expired TCP flows to avoid unnecessary memory consumption. A TCP flow record is cleaned from the memory if the inactive time of that flow exceeds a timeout threshold T_{out}. The timeout value T_{out} represents a critical limit above which the response time becomes unacceptable. If a request still does not receive its

response after T_{out} time, the agent counts the request as a timeout request. If a DMADL agent captures many timeout requests, the estimated mean response time is no longer accurate as the response times of many requests are not considered in the estimation of mean response time. In this case, users' experience degrades significantly. The DMADL agent triggers an anomaly alert immediately after it detects many timeout requests.

6.2 Anomaly Detection

DMADL agents use the local mean response time and the number of timeout requests at each component for anomaly detection. The performance of a component is suspicious if the mean response time exceeds an acceptable threshold. This threshold dynamically adapts to different workload scenarios to reflect the dynamic characteristics of Cloud services. A static threshold would fail to capture dynamic behaviors. We train the system and obtain the response time X_t during normal operation. The component's normal response time distribution is characterized with its mean μ_t and standard deviation σ_t. DMADL uses the exponentially weighted moving average to update the distribution:

$$\begin{aligned} \mu_t &= \alpha\mu_{t-1} + (1-\alpha)X_t \\ \sigma_t &= \alpha\sigma_{t-1} + (1-\alpha)\max\{|X_t - \mu_t|, \sigma_{t-1}\} \end{aligned}$$

Here, α is the weight put on the historical response time. The max function is used as a watermark for the maximum fluctuation in the historical response time.

If the current response time stays within $\lambda * \sigma_t$ from the mean μ_t, i.e., $X_t \in \{0, \mu_t + \lambda\sigma_t\}$, then the response time X_t is normal. Otherwise, the response time is considered as suspicious. The response time of a dynamic system usually fluctuates even in normal operation. DMADL agents use a window of latest W samples to make decisions. A DMADL agent triggers an anomaly alert if the number of suspicious data points in the decision window exceeds a tolerable threshold. Otherwise, the DMADL agent determines that it caused by normal system fluctuations.

6.3 Anomaly Localization

DMADL uses the dependency model and corresponding response time estimator for anomaly localization. When a DMADL agent detects an anomaly at a component, it first checks whether the local component has dependent components. The anomaly can be caused by a fault in the component itself or its subsystem components. DMADL has different localization schemes for different dependency primitives.

Algorithm 3: DMADL anomaly detection

input : Given a local component A and its m dependent components $S = \{A_{d1}, A_{d2}, \cdots, A_{dm}\}$
output: Detection result for the local component A

It selects a sampling interval ΔT_A to monitor the number of incoming request data packets and outgoing response data packets. The sampling interval ΔT_A should approximate the maximum acceptable response time at the local component.;
It monitors the number of request data packets arrived at component A: $X_A(t)$, and the number of response data packets departed from component A: $Y_A(t)$.;
It uses a window of latest W_A samples for detecting anomalies;
if $\sum_{t=1}^{W_A} D_A(t) > 0$ **then**
    $state(t) = -1$;
    Anomaly Alert: A does not return responses within the timeout value;
else if $\sum_{t=1}^{W_A} X_A(t) > 0$ *and* $\sum_{t=1}^{W_A} Y_A(t) == 0$ *and* $\sum_{t=1}^{W_A-1} X_A(t) > 0$ **then**
    $state(t) = -1$;
    Anomaly Alert: A does not return responses for requests;
else
    if $\sum_{t=1}^{W_A} R_A(t) >= \mu_A(t) + k_A\sigma_A(t)$ **then**
        $suspicious(t) = -1$;
    else
        $suspicious(t) = 1$;
    end
    if $\sum_{t=1}^{W_A} suspicious(t) <= TH_A$ **then**
        $state(t) = -1$;
        Anomaly Alert: A's response time is abnormal;
    else
        $state(t) = 1$;
    end
end

6.3.1 No Dependent Component

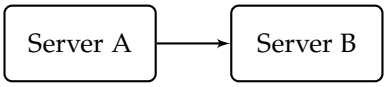
If a local component does not have any subsystem component, the local response time can be impacted by itself or its upstream components due to TCP propagation. As shown in Figure 3.1, the agent for component SQL detects larger response time but SQL is still normal. The agent for SQL cannot make the decision since the local anomaly can be caused by upstream component. The DMADL agent has different functions under different scenarios.

1. If the anomaly alert is triggered by abnormal response time, the agent does not perform localization analysis for a local component without any dependent component. The state of local component is given by DMADL agents of its upstream components.

2. If the anomaly alert is triggered by many detected timeout requests, many requests do not have their responses returned within the acceptable time limit. The DMADL agent decides that the local component is faulty and reports the result.

6.3.2 Single Dependent Component

When the local component has only one dependent component, the DMADL agent decides the source of anomaly from the mean response time of the local component and its subsystem. Figure 6.2 gives a case where a local component A has a single dependent component B. With different workload, a request



Variable	Notation
The number of requests at A	$n_A(t)$
The number of requests from A to B	$n_{AB}(t)$
The mean response time of A	$R_A(t)$
The mean response time of B monitored at A	$R_{AB}(t)$

Figure 6.2: The case with a single dependent component

arrived at A may cause a different number of requests to B: a request may only access A but not cause any request to B; a request may cause a few requests to B as it requires only a small part of service B; and a request may cause many requests to B as it relies heavily on B. On average, a request arrived at the local component A causes $\frac{n_{AB}(t)}{n_A(t)}$ requests to its dependent component B with mean response time $R_{AB}(t)$. Therefore, each request at the local component A spends $\frac{n_{AB}(t)}{n_A(t)} * R_{AB(t)}$ time on the dependent component B and B's subsystem. The mean response time of A's subsystem $S_A(t)$ is the average time spent at A's subsystem.

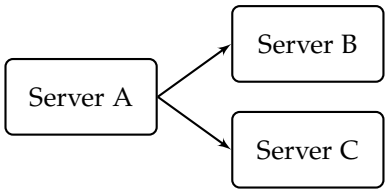
$$S_A(t) = \frac{n_{AB}(t)}{n_A(t)} * R_{AB}(t)$$

The DMADL agent first checks the state of the dependent component using following steps.

- If the agent captures many timeout requests or no outgoing responses for incoming requests in recent period, the agent directly determines that the dependent component as faulty.
- If step 1 is not satisfied, the agent checks the mean response time of the dependent component. If the mean and the subsystem impact (e.g., $\frac{S_A(t)}{R_A(t)}$) increases significantly in recent period, the agent determines that the dependent component as faulty. If the subsystem impact decreases, the agent determines that the dependent component is normal.

6.3.3 Multiple Dependent Components

If the local component has more than one dependent component, the DMADL agent uses the local dependency model to first determine the mean subsystem response time. We use a simple example of a local component A with two dependent components (B and C) in Figure 6.3 to show how to determine subsystem response time with different dependency primitives.



Variable	Notation
The number of requests at A	$n_A(t)$
The number of requests from A to B	$n_{AB}(t)$
The number of requests from A to C	$n_{AC}(t)$
The mean response time of A	$R_A(t)$
The mean response time of B monitored at A	$R_{AB}(t)$
The mean response time of C monitored at A	$R_{AC}(t)$

Figure 6.3: The case with multiple dependent components

Composite Dependency

If dependent components form a composite dependency primitive, the time spent at the subsystem is the sum of the time spent at its dependent components.

$$S_A(t) = \left(\frac{n_{AB}(t)}{n_A(t)} * R_{AB}(t) + \frac{n_{AC}(t)}{n_A(t)} * R_{AC}(t)\right)$$

Concurrent Dependency

If dependent components form a concurrent dependency primitive, the subsystem response time is the maximum time spent at individual dependent components.

$$S_A(t) = \max\left(\frac{n_{AB}(t)}{n_A(t)} * R_{AB}(t),\ \frac{n_{AC}(t)}{n_A(t)} * R_{AC}(t)\right)$$

Distributed Dependency

If dependent components form a distributed dependency primitive, the subsystem response time is the probabilistic sum of the time spent at its dependent components.

$$S_A(t) = \left(\frac{n_{AB}(t)}{n_A(t)} * R_{AB}(t) + \frac{n_{AC}(t)}{n_A(t)} * R_{AC}(t)\right)$$

After deriving the mean subsystem response time of the local component A, the DMADL agent introduces component impact analysis for anomaly localization.

6.3.4 Subsystem Impact Analysis

The subsystem impact is defined as the ratio of subsystem response time over the local mean response time. For example, the subsystem impact of A is defined as:

$$I_{S_A}(t) = \frac{S_A(t)}{R_A(t)}$$

When an anomaly is triggered and the DMADL detects a larger subsystem impact compared to that in normal operation, the DMADL agent determines that the anomaly is caused by the subsystem. The DMADL agent further finds out which dependent components are causing the local anomaly if there are multiple dependent components.

6.3.5 Component Impact Analysis

The DMADL agent determines the impact of dependent components using the local dependency model and uses the component impact to determine the source of anomaly.

Distributed and Composite Dependency Primitive

In distributed dependency primitive and composite dependency primitive, the impact of a dependent component is the ratio of its mean response time over the subsystem response time multiplied by the number of dependent components. The dependent component impact of B computed by the DMADL agent of the local component A during a monitoring interval $(t - \Delta T, t)$ is

$$I_{AB}(t) = \frac{R_{AB}(t)}{S_A(t)} * N_A$$

Similarly, the dependent component impact of C computed by the agent of the local component A during a monitoring interval $(t - \Delta T, t)$ is

$$I_{AC}(t) = \frac{R_{AC}(t)}{S_A(t)} * N_A$$

Concurrent Dependency Primitive

In concurrent dependency primitive, the impact of a dependent component is the ratio of mean response time over the sum of response time of dependent components multiplied by the number of dependent components.

$$I_{AB}(t) = \frac{R_{AB}(t)}{R_{AB}(t) + R_{AC}(t)}$$

For each dependent component, the agent repeats the following steps to pinpoint whether the dependent component is faulty or not.

1. If the agent captures many timeout requests at a dependent component in recent period, then it determines that the dependent component is faulty.
2. If step 1 is not satisfied, DMADL starts to check the mean response time of subsystem and dependent components. If the subsystem impact decreases, the DMADL agent determines that the subsystem is normal. If the subsystem impact increases compared to that in normal operation, the agent determines that the subsystem is faulty. After that, the agent determines that the dependent component whose impact increases as faulty.

If all dependent components are normal and the agent captures many timeout requests at the local component, the agent determines that the local component is faulty. Otherwise, the agent cannot determine the

state of the local component. Instead, its state would be reported by DMADL agents that are monitoring its upstream components.

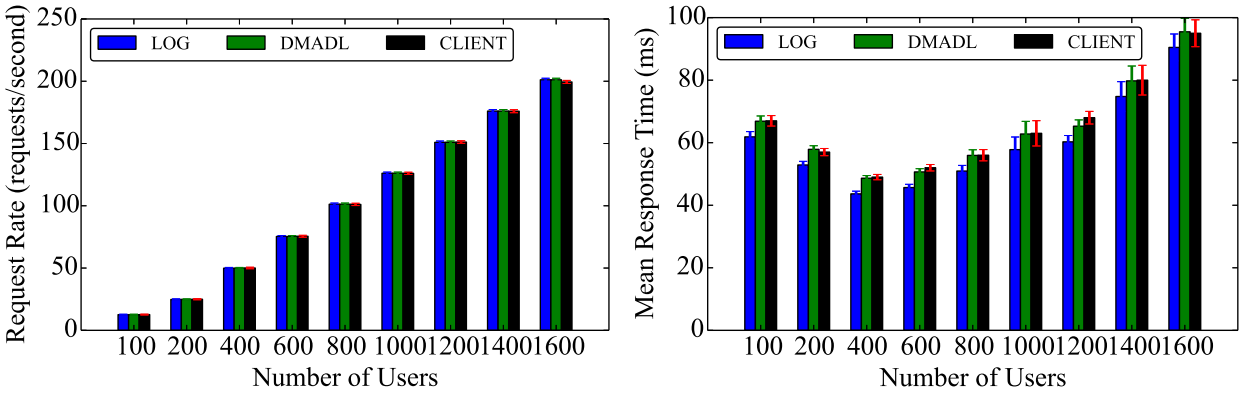
Each DMADL agent reports the localization result of its local component and corresponding dependent components. DMADL has a central administrator which summarizes the localization results of all agents and achieve anomaly localization in Cloud applications.

6.4 Experimental Evaluation

We first evaluate the accuracy of the approximated mean response time compared to the real response time obtained from the server logs and client logs under varying workloads. We then show the performance of anomaly detection using the estimated mean response time. After that, we evaluate DMADL in localization of faulty components with faults injected to different components in Cloud applications. Finally, we show the overhead of DMADL.

6.4.1 Mean Response Time Estimation

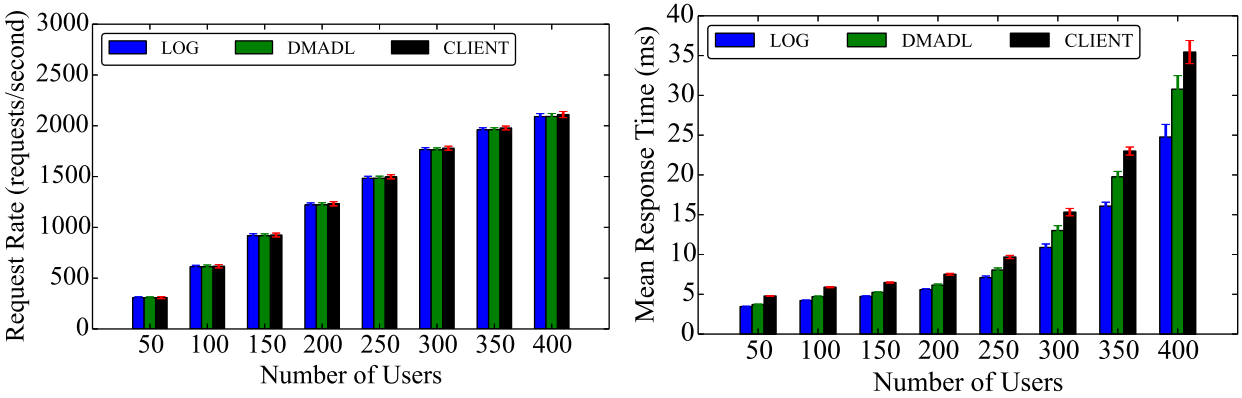
We evaluate DMADL in estimation of the mean response time and mean request rate with 95% confidence interval measured under varying workload intensity in different Cloud applications. To check its validity, we also measure the mean response time and mean request rate in client-side logs, and server logs. All variables are measured using a 1-second sampling interval. We show the result at the entry point of each application: CloudSuite web search application in Figure 6.4, Olio web application in Figure 6.4, and MediaWiki application in Figure 6.4. The mean response time approximated by DMADL agents usually falls between the client-side response time and the server-logged response time. The server-logged mean response time is usually smaller than the mean response time estimated by DMADL agents. DMADL agents estimate the response time of each request as the elapsed time from when the first data packet of the request arrives at the network interface to when the last data packet of its response departs from the network interface. The server logs the starting time of a request when it arrives at the application layer and the application header is parsed by the component service. The server logs a response when the response is passed from the application layer to bottom layers. The server-logged response time of a component includes the response time of its subsystem and the processing time of itself in the application layer, it does not include the waiting time and processing time below the application layer at the local component. DMADL has agents estimate the mean response time at each component of multi-component applications. The estimated mean response time characterizes the performance of the component and its subsystem. It is further used for anomaly detection and localization in Cloud applications.



(a) The request rate under varying workload

(b) The mean response time under varying workload

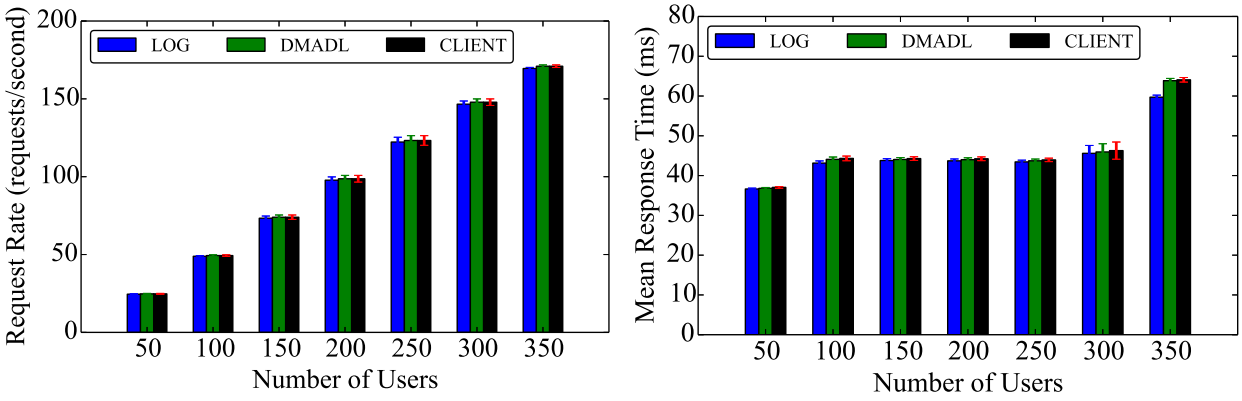
Figure 6.4: The DMADL estimation of mean response time and mean request rate in CloudSuite web search application



(a) The request rate under varying workload

(b) The mean response time under varying workload

Figure 6.5: The DMADL estimation of mean response time and mean request rate in Olio web application



(a) The request rate under varying workload

(b) The mean response time under varying workload

Figure 6.6: The DMADL estimation of mean response time and mean request rate in MediaWiki application

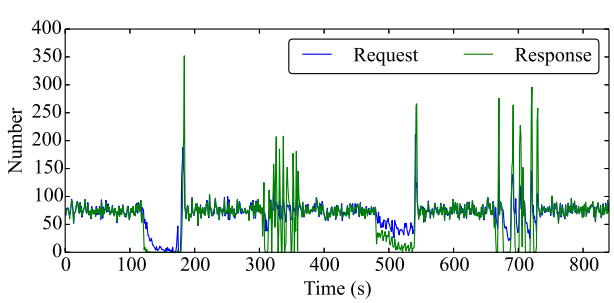
6.4.2 Resource Fault Detection

We show how DMADL detects different resource faults in different Cloud applications. We use a 1-second sampling interval to estimate the mean response time and count the number of timeout requests (e.g., the request is not responded within 5 seconds). Figure 6.7 shows the estimated mean response time and the number of timeout requests at the component *solr0* in CloudSuite web search application. Figure 6.8 shows the estimated mean response time and the number of request timeouts at the component *lb0* in Olio web application. Figure 6.9 shows the estimated mean response time and the number of request timeouts at the component *lb0* in MediaWiki application. All sampled metrics give a clear view about the effectiveness of our anomaly detection method.

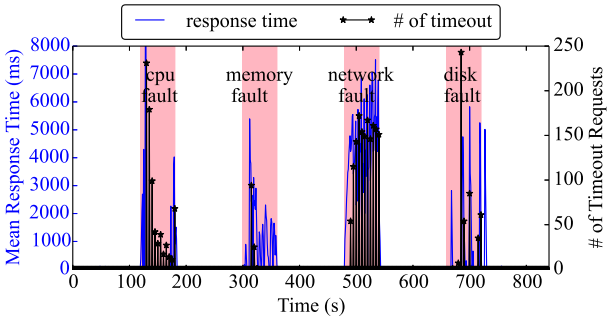
1. CPU fault. It takes much longer time to process each arriving request. The mean response time increases from several milliseconds to several seconds. Some requests are even not responded within the timeout limit. The estimated response time increases and there are a large number of timeout requests.
2. Memory fault. During the memory fault, each request requiring accessing the memory suffers much longer delay. The faulty processes keep spinning on "malloc()/free()" functions on the memory. When processes free the memory resource, the service gets enough memory to process requests and return responses. When processes request for memory allocation, the service gets stuck due to memory bottleneck. As a result, the mean response time suffers frequent fluctuation, as well as the number of requests and responses per second at the local component during the memory fault.
3. Network fault. During the network fault, each outgoing packet is delayed at the local component for much longer time. The estimated response time increases and DMADL agents also capture many timeout requests.
4. Disk fault. During the disk fault, requests that need access to disk data suffers much longer delay. There are no responses returned from the faulty component. The mean response time reaches several seconds, and some requests cannot have their responses within timeout limit.

6.4.3 Common Software Fault Detection

We further reproduce different performance bugs and inject them into experiments. The SOLR-5935 bug is injected to component *solr0* during 120-180s, and to component *solr2* during 480-540s. The SOLR-5216 bug is injected to component *solr0* during 300-360s, and to component *solr2* during 760-820s. Figure 6.10a shows the estimated number of requests and responses at component *solr0*. Figure 6.10b shows the

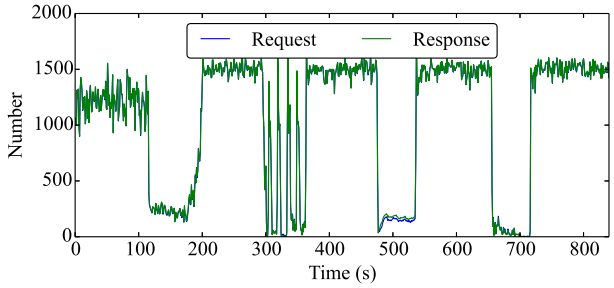


(a) The number of requests and responses at *solr0*.

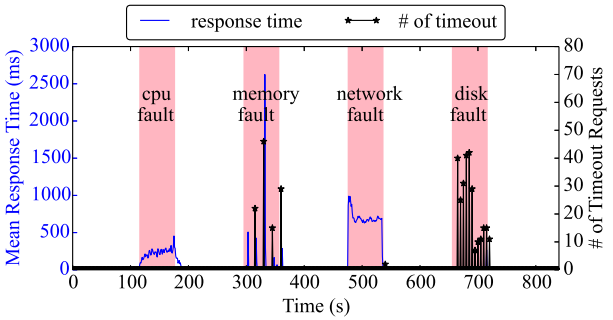


(b) The mean response time and timeout requests at *solr0*

Figure 6.7: The DMADL detection result of resource faults in CloudSuite web search application

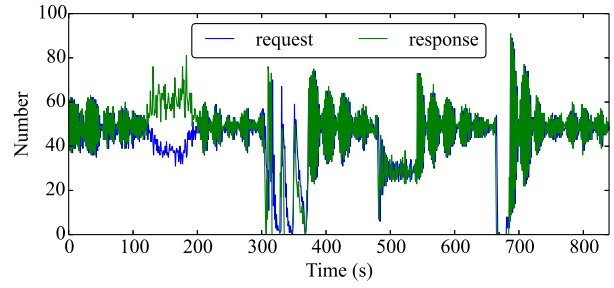


(a) The number of requests/responses at *lb0*.

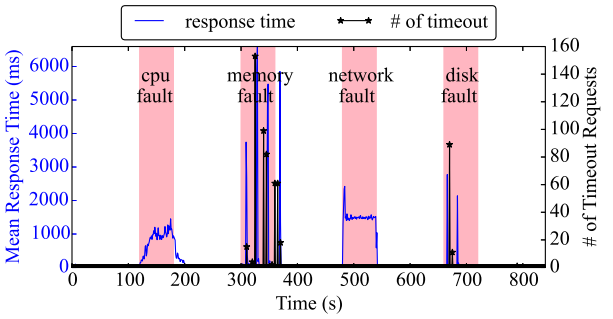


(b) The mean response time and timeout requests at *lb0*.

Figure 6.8: The DMADL detection result of resource faults in Olio web application



(a) The number of requests/responses at *lb0*.



(b) Mean response time and timeout requests at *lb0*.

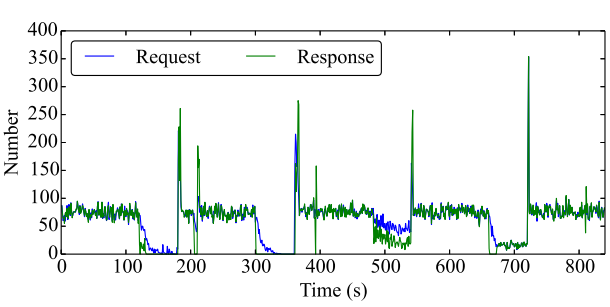
Figure 6.9: The DMADL detection result of resource faults in MediaWiki application

estimated mean response time and the number of timeout requests at the component *solr0* in CloudSuite web search application.

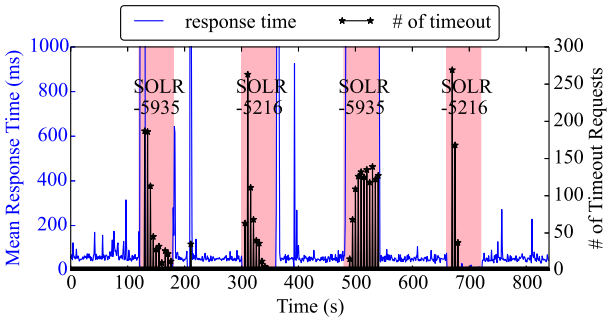
1. Deadlock bug: the SOLR-5935 bug at component *solr0* causes local solr threads to fall into deadlock. The component cannot return responses for incoming requests. Requests have to wait at the component until the deadlock is released. The SOLR-5935 bug at the dependent component *solr2* does not necessarily cause the local service to crash. It takes longer time to return responses for incoming

requests. The number of outgoing responses per second is much lower than the number of incoming requests per second at component *solr0*.

2. Service hang bug: the SOLR-5216 bug at component *solr0* causes the solr process to hang. The component cannot return responses for incoming requests. Requests have to wait in the component until the deadlock is released. The SOLR-5216 bug at the dependent component *solr2* does not necessarily cause the local service to crash. As all dependent components form a concurrent dependency primitive, the component *solr2* cannot return responses. The local component waits for requests until the timeout limit is reached. Finally, the component *solr0* returns incomplete responses back to clients.



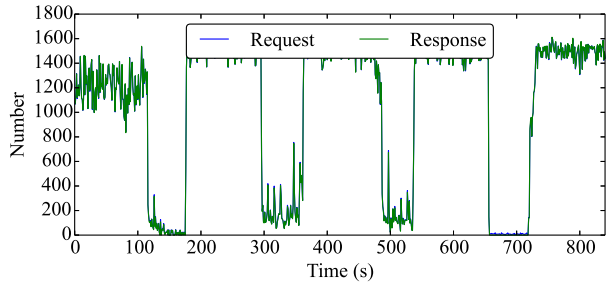
(a) The number of requests/responses collected at *solr0*.



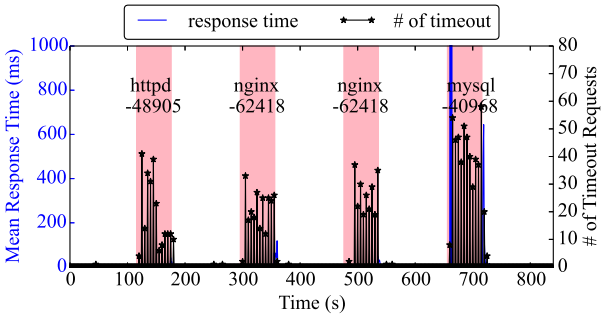
(b) The mean response time and timeout requests at *solr0*.

Figure 6.10: The DMADL detection result of performance bugs in CloudSuite web search application

Figure 6.11 shows the detection result of performance bugs in Olio application.



(a) The number of requests/responses at component *lb0*



(b) The mean response time and timeout requests at *lb0*

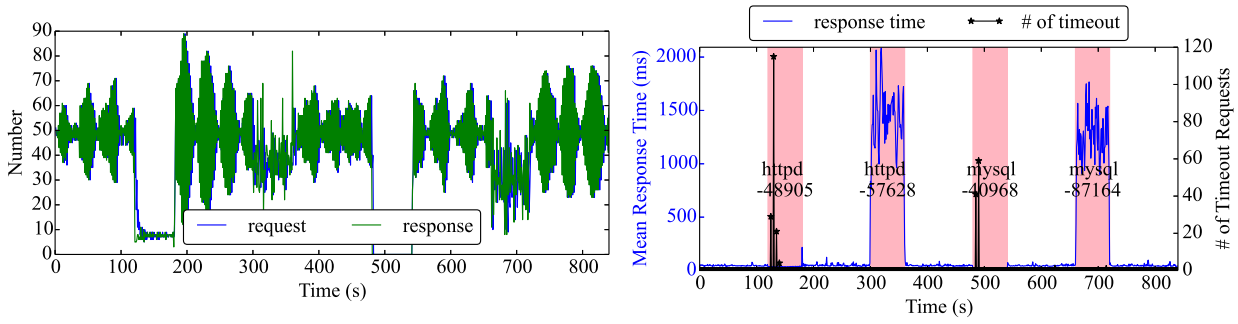
Figure 6.11: The DMADL detection result of performance bugs in Olio web application

1. Bug HTTPD-48905: the bug injected into the Apache load balancer *lb0* causes most child processes to hang. No responses would be returned from hanged processes. Many requests are considered as timeout requests. The remaining normal child processes still process requests correctly and the response time is still normal. DMADL triggers anomaly alert after detecting a large number of timeout requests.

2. Bug NGINX-62418: the bug injected into both Nginx application servers *ap1* causes worker processes to crash. The component would not be able to return responses for incoming requests. All upstream components cannot have any responses. DMADL detects a lot of timeout requests and triggers alert at component *lb0*.

3. Bug MySQL-40968: the bug at the database component *sq0* causes the MySQL process to hang without any error message. The component would not be able to return responses for incoming requests. Requests have to wait in the component until the MySQL process recovers. The MySQL process would not be able to return responses for arrived requests. The response time increases and DMADL detects a lot of timeout requests.

Figure 6.12 shows the detection result of performance bugs in MediaWiki application.



(a) The number of requests/responses at *lb0*

(b) The mean response time and timeout requests at *lb0*

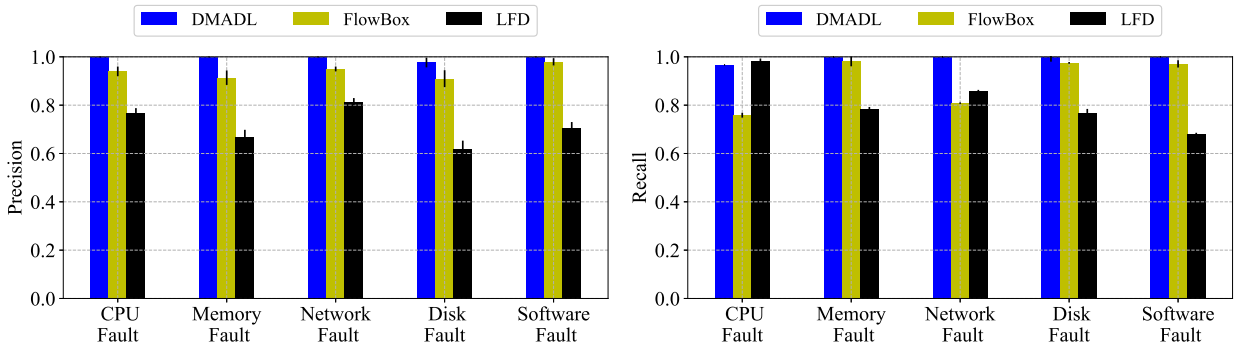
Figure 6.12: The DMADL detection result of performance bugs in MediaWiki application

1. Bug HTTPD-57628: the bug causes unexpected larger response time. DMADL triggers anomaly alert as the response time goes from 10ms to more than 1 second.

2. Bug MySQL-87614: the bug at the database component sq0 causes the MySQL process to suffer much larger latency in processing MySQL queries. It causes larger response time at the load balancer as the load balancer contains the database as a subsystem component.

6.4.4 Anomaly Detection Performance

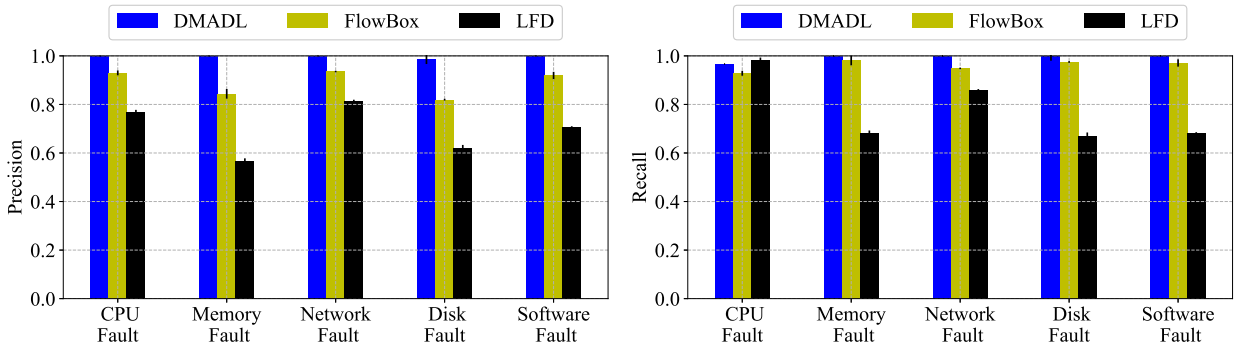
We show the performance of DMADL for anomaly detection in different Cloud applications. We perform a large number of experiments under varying workload intensities. During the experiment, we inject different kinds of faults into different components. We compare DMADL with two other black-box anomaly detection methods LFD [32], and FlowBox [45].



(a) The detection precision

(b) The detection recall

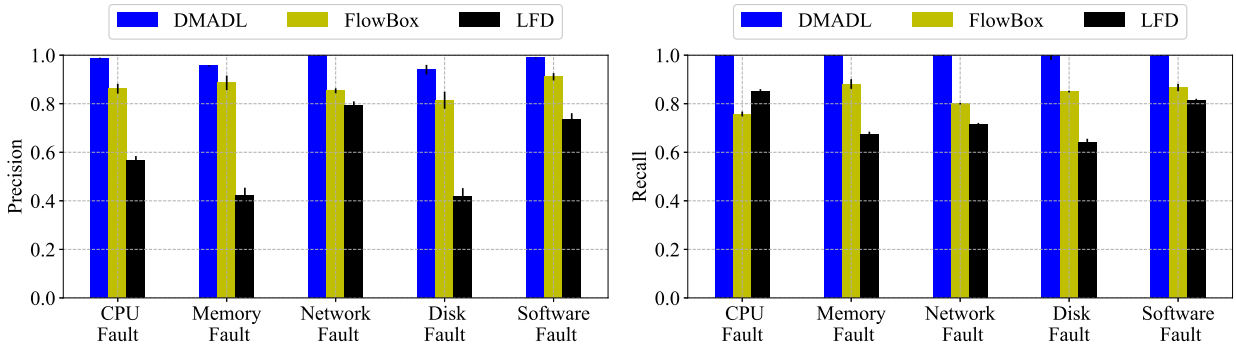
Figure 6.13: The detection performance of DMADL and other methods in CloudSuite web search application.



(a) The detection precision

(b) The detection recall

Figure 6.14: The detection performance of DMADL and other methods in Olio web application

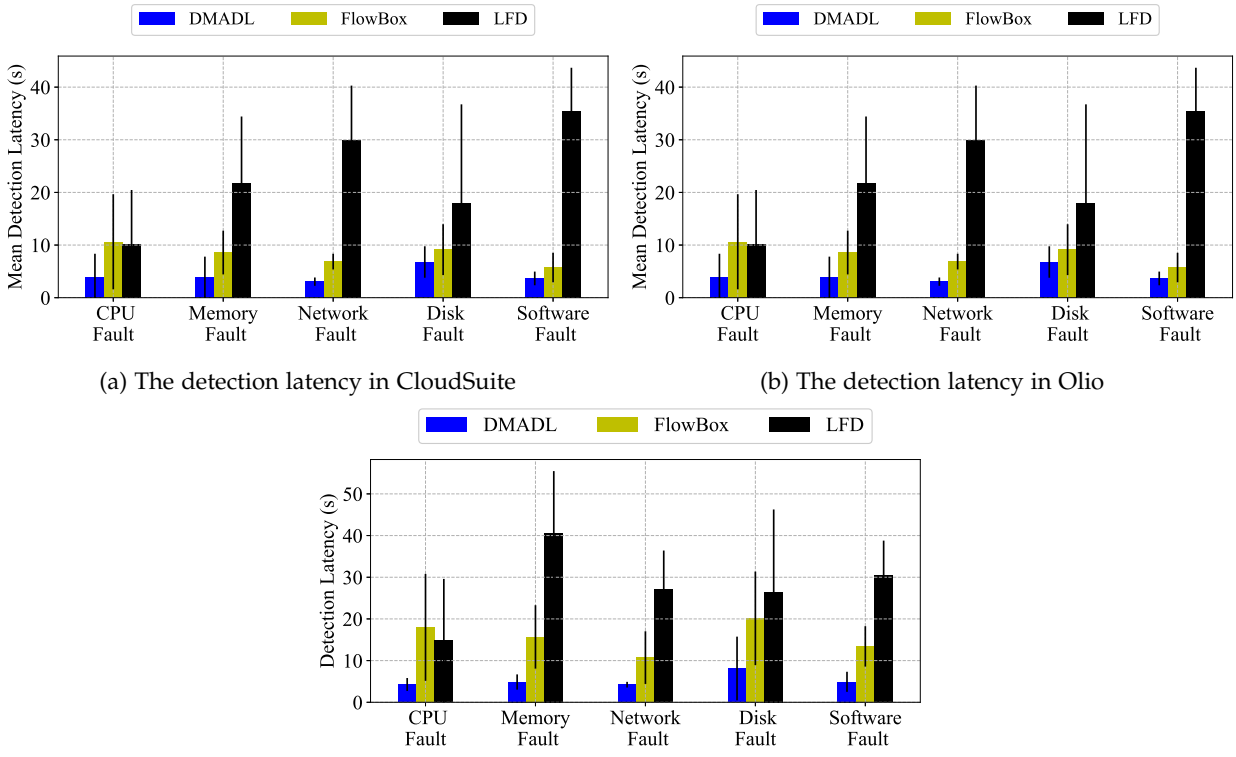


(a) The detection precision

(b) The detection recall

Figure 6.15: The detection performance of DMADL and other methods in MediaWiki application

We observe that both DMADL and FlowBox achieves high detection precision and low false positive for different faults. DMADL also achieves lowest false negative for different faults. FlowBox has high false negative while detecting CPU and network faults. We check into those cases and found that FlowBox misses many CPU and network faults when the workload intensity is low. When the workload intensity is very low, the limited capacity is enough to process requests. Another reason is that a lower workload intensity causes the common flow ratio to be more unpredictable. But with the adjusted flow ratio, DMADL still detects it well. LFD has the worst detection performance for all different faults. The main reason is that it selects the highest correlation between the user-space CPU utilization and other resource utilization as the metric to characterize the service performance. It is prone to high false positive and high false negative. There are two main reasons. One is that if the server does not use too much user-space CPU utilization, the correlation is not meaningful. The other reason is that some resource utilization metrics are still highly correlated with the user-space CPU utilization even when the server has different faults.



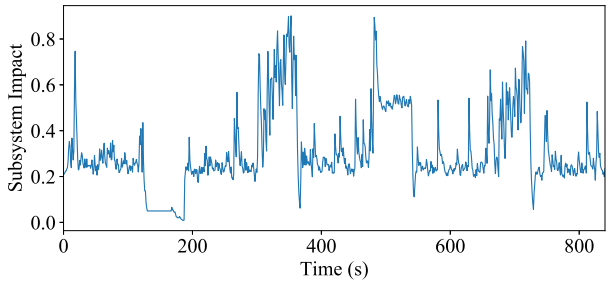
(a) The detection latency in CloudSuite

(b) The detection latency in Olio

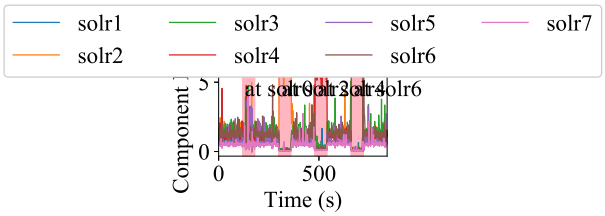
(c) The detection latency in MediaWiki

Figure 6.16: The detection latency of DMADL and other methods in different Cloud applications.

Figure 6.16 shows the mean detection latency for different kinds of faults in the CloudSuite web search application. The error bar denotes the standard deviation of the detection latency. DMADL and FlowBox achieves consistently small detection latency. LFD has much longer delay at detecting memory, network



(a) The subsystem impact



(b) The dependent component impact

Figure 6.17: The component impact analysis for anomaly localization in CloudSuite web search application

and software faults. It is because LFD uses a long-time window for computing the correlation. The fault usually causes LFD correlation to decrease gradually and the sliding window has to move long enough to trigger anomaly alert.

6.4.5 Anomaly Localization Case Study

In this section, we evaluate DMADL for anomaly localization in different Cloud applications when performance anomalies are injected at different components.

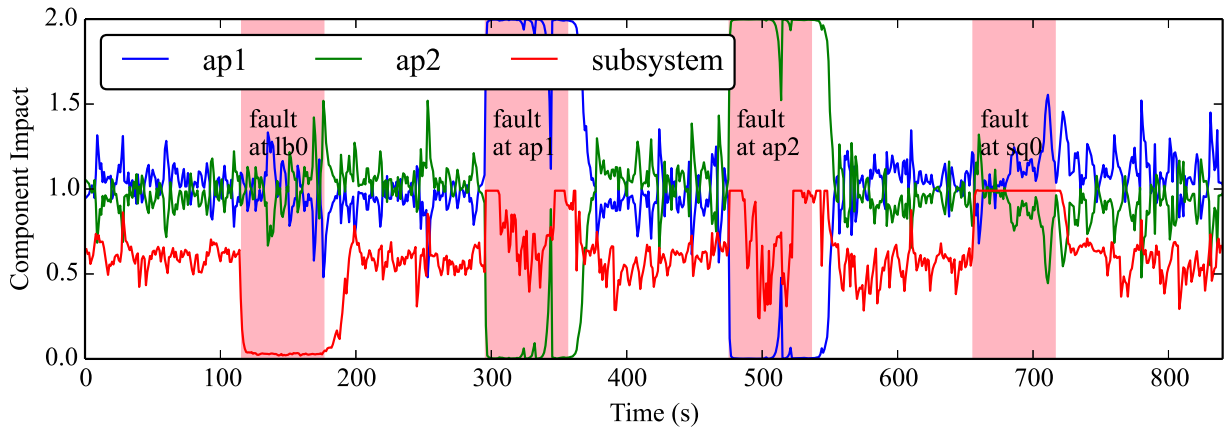
CloudSuite Web Search Application

Figure 6.17 shows the component impact analysis at component *solr0* when faults are injected to *solr0* during $120 - 180$s, *solr2* ($300 - 360$s), *solr4* ($480 - 540$s), and *solr6* ($660 - 720$s). When the fault is injected to *solr0*, the local response time increases mostly because the service time of *solr0* increases. The subsystem has smaller impact on the local response time, and the DMADL agent determines that the subsystem of *solr0* is normal. When the fault is injected to *solr2*, the response time of *solr2* increases. The subsystem response time of *solr0* also increases, and the subsystem impact of *solr0* increases. The DMADL agent of *solr0* determines that the anomaly is caused by the subsystem. The impact of *solr2* has much higher impact than other dependent components. The DMADL agent of *solr0* determines that the local anomaly is caused by *solr2*. The similar analysis applies when faults are injected to other dependent components.

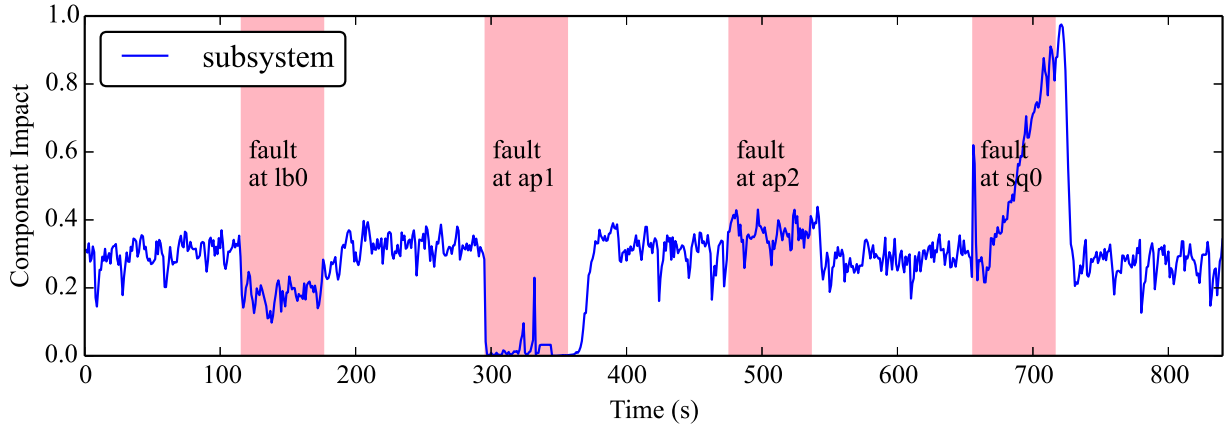
Olio Web Application

Figure 6.18 shows the component impact analysis at different components of Olio web application when faults are injected to *lb0* ($120 - 180$s), *ap1* ($300 - 360$s), *ap2* ($480 - 540$s), and *sq0* ($660 - 720$s) respectively.

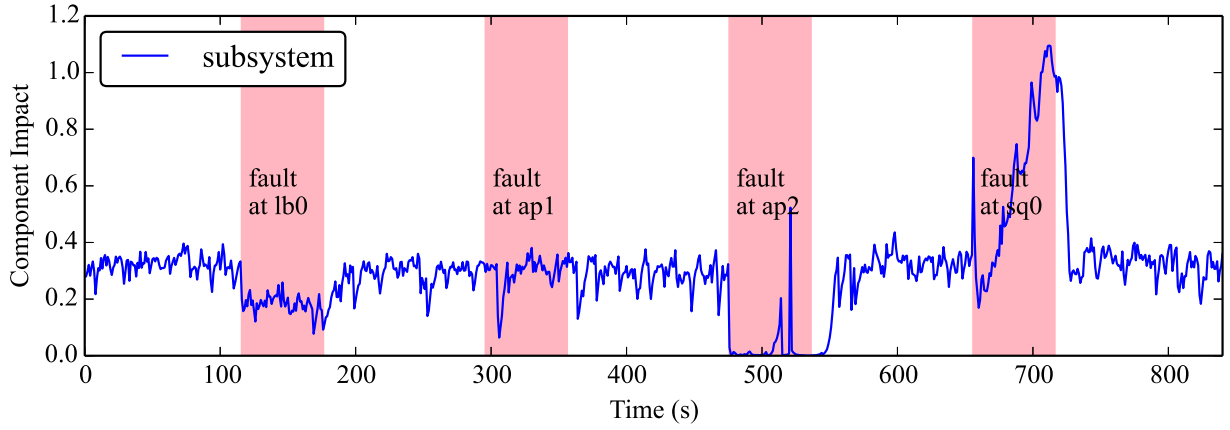
When the fault is injected to component *lb0*, the local service time of *lb0* increases. The local response time increases, but it is mostly caused by local component. The subsystem impact of *lb0* decreases. The



(a) Component impact analysis by DMADL agent of *lb0*



(b) Component impact analysis by DMADL agent of *ap1*



(c) Component impact analysis by DMADL agent of *ap2*

Figure 6.18: The component impact analysis for anomaly localization in Olio web application

fault at *lb0* may affect the response time of its dependent components. But the impact is similar on all dependent components. The impact of its dependent components (*ap1, ap2*) almost do not change

compared to normal operation. The DMADL agent of *lb0* determines that the local component is faulty.

When the fault is injected to component *ap1*, the response time of *ap1* increases and the subsystem response time of *lb0* increases significantly. The subsystem impact increases compared to normal operation. The DMADL agent of *lb0* determines that its subsystem is faulty. The response time of *ap2* is still normal. The impact of the dependent component *ap1* increases, but the impact of *ap1* decreases. The DMADL agent determines that the anomaly at *lb0* is caused by the dependent component *ap1*. The similar analysis also applies when the fault is injected to component *ap2*.

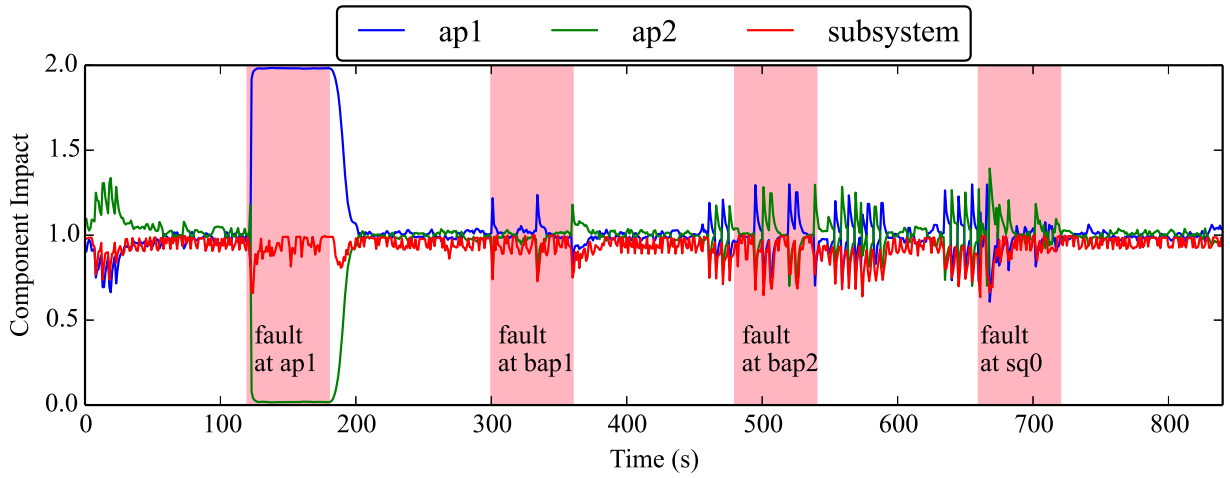
When the fault is injected to component *sq0*, the response time of *sq0* increases and it increases the response time of its upstream components (*ap1, ap2, lb0*). At component *lb0*, the local response time increases mostly because of its subsystem. The response time of both *ap1* and *ap2* increases similarly, and their component impacts do not show significant changes compared to normal operation. The subsystem impact of *lb0* increases and the DMADL agent of *lb0* determines that its subsystem as faulty. The component *ap1* and *ap2* have a single dependent component *sq0*, the subsystem impact increases. The DMADL agents of both *ap1* and *ap2* pinpoint sq0 as faulty. Combining these results, DMADL determines that *sq0* is faulty.

MediaWiki Application

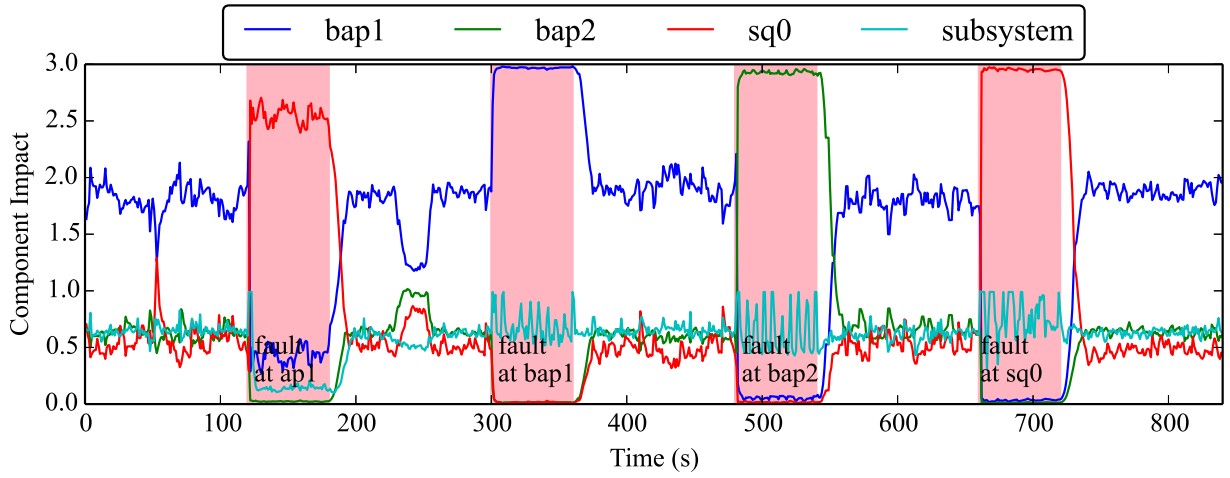
Figure 6.19 shows the component impact analysis at different components when faults are injected to component *ap1* (120-180s), *bap1* (300-360s), *bap2* (480-540s), and *sq0* (660-720s).

When component *ap1* is faulty, DMADL agents for component *lb0*, *ap1*, and *sq0* detect anomalies. The agent of *lb0* finds that the impact of *ap1* increases a lot, but the impact of *ap2* decreases. The subsystem impact is always 1.0. DMADL agent of *lb0* pinpoints its dependent component *ap1* as faulty. The agent of *ap1* checks the component impact of all dependent components. The subsystem impact decreases, it means the local response time increases not because of subsystem. Although, the component*sq0* has higher impact, DMADL still concludes that the subsystem is normal. DMADL of *ap1* does not perform localization for itself, and it does not report any component. DMADL agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

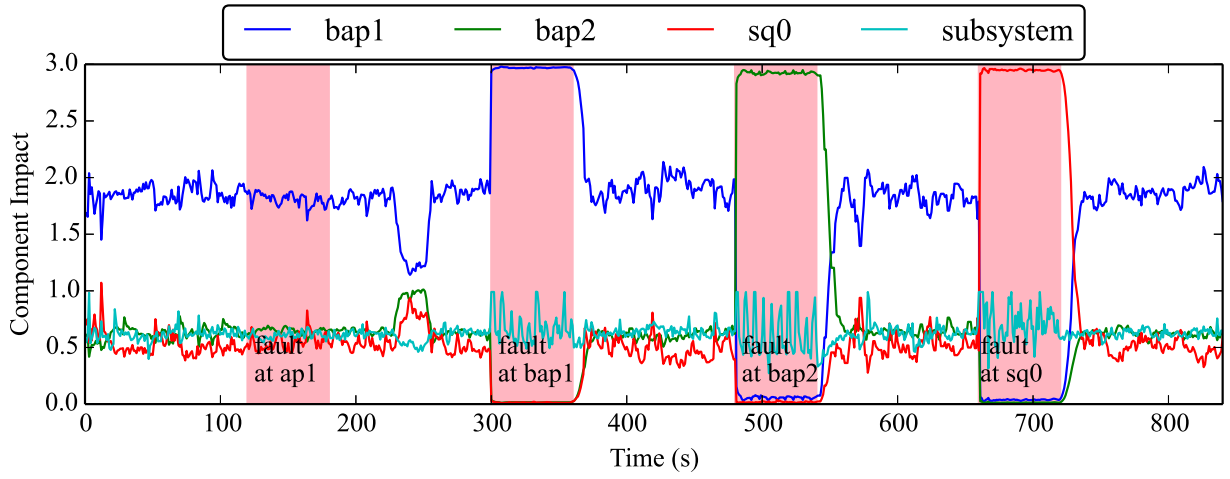
When component *bap1* is faulty, DMADL agents of component *lb0*, *ap1*, *ap2*, and *bap1* detect anomalies. The agent of *lb0* observes that the response time of both dependent components increase. The subsystem impact is always 1.0. The local response time increases due to the subsystem. DMADL agent of *lb0* pinpoints the subsystem as faulty. DMADL agent of *ap1* finds that the component impact of *bap1* increases and the subsystem impact does not decrease, It shows that the response time of *ap1* increases due to *bap1*. DMADL agent of *ap1* pinpoints the dependent component *bap1* as faulty. Similarly, DMADL agent of *ap2*



(a) The component impact analysis at component *lb0*



(b) The component impact analysis at component *ap1*



(c) The component impact analysis at component *ap2*

Figure 6.19: The component impact analysis for anomaly localization in MediaWiki application

also pinpoints *bap1* as faulty. DMADL agent of *bap1* does not perform localization as *bap1* does not have any dependent component.

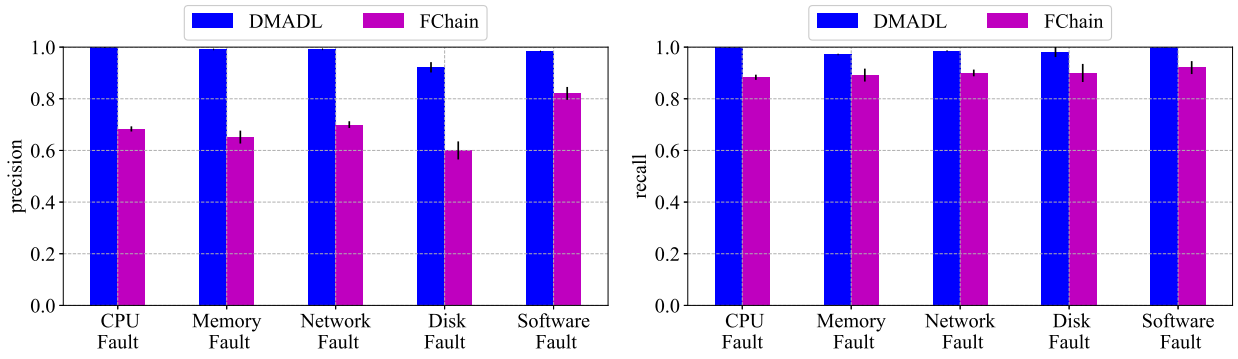
When component *bap2* is faulty, DMADL agents of component *lb0*, *ap1*, *ap2*, and *bap2* detect anomalies. The agent of *lb0* observes that the response time of both dependent components increase. The subsystem impact is always 1.0. The local response time increases due to the subsystem. DMADL agent of *lb0* pinpoints the subsystem as faulty. DMADL agent of *ap1* finds that the component impact of *bap2* increases and the subsystem impact does not decrease, It shows that the response time of *ap1* increases due to *bap2*. DMADL agent of *ap1* pinpoints the dependent component *bap2* as faulty. Similarly, DMADL agent of *ap2* also pinpoints *bap2* as faulty. DMADL agent of *bap2* does not perform localization as *bap1* does not have any dependent component.

When component *sq0* is faulty, DMADL agents of component *lb0*, *ap1*, *ap2*, and *sq0* detect anomalies. The agent of *lb0* observes that the response time of both dependent components increase. The subsystem impact is always 1.0. The local response time increases due to the subsystem. DMADL agent of *lb0* pinpoints the subsystem as faulty. DMADL agent of *ap1* finds that the component impact of *sq0* increases and the subsystem impact does not decrease, It shows that the response time of *ap1* increases due to *sq0*. DMADL agent of *ap1* pinpoints the dependent component *sq0* as faulty. Similarly, DMADL agent of *ap2* also pinpoints *sq0* as faulty. DMADL agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

6.4.6 Anomaly Localization Performance

Figure 6.20 shows the localization performance comparison of DMADL and FChain in CloudSuite web search application. DMADL achieves much higher precision and fewer false positives than FChain for all different faults. We further check the localization result given by FChain and find that there are around 6 different components showing the same change start time. It fails to determine the faulty component after applying FChain in anomaly localization. All dependent components have the same change point. For example, when a CPU fault happens at *solr2*, the abnormal change start time of the faulty component is the onset time of the CPU fault. The fault further affects the interaction between the index component *solr0* and other dependent components. Other dependent components show abnormal change in the number of incoming or outgoing packets. DMADL has fewer false negatives than FChain for all different faults. The main reason is that a component fault propagates very quickly due to high throughput communication when it interacts with other components. The fault at component *solr0* propagates to its dependent components. FChain falsely pinpoints its dependent components as faulty, but it misses the true faulty

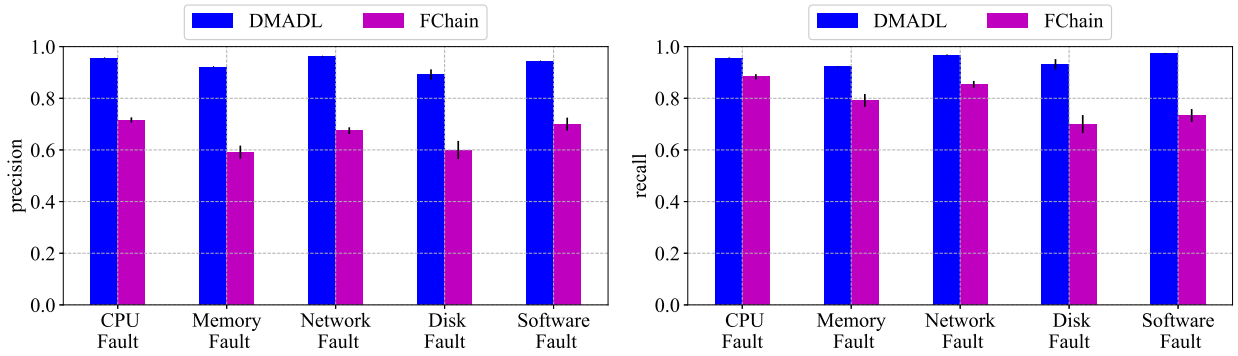
component *solr0*. Figure 6.21 and Figure 6.22 show the performance of DMADL and FChain in Olio web application and MediaWiki application, respectively. DMADL achieves much higher precision than FChain for all different faults. We further check the localization result of FChain and find that there are more than one component showing the same fault manifestation time. FChain fails to determine the faulty component in Cloud applications.



(a) The localization precision

(b) The localization recall

Figure 6.20: The localization performance of DMADL and other methods in CloudSuite web search application



(a) The localization precision

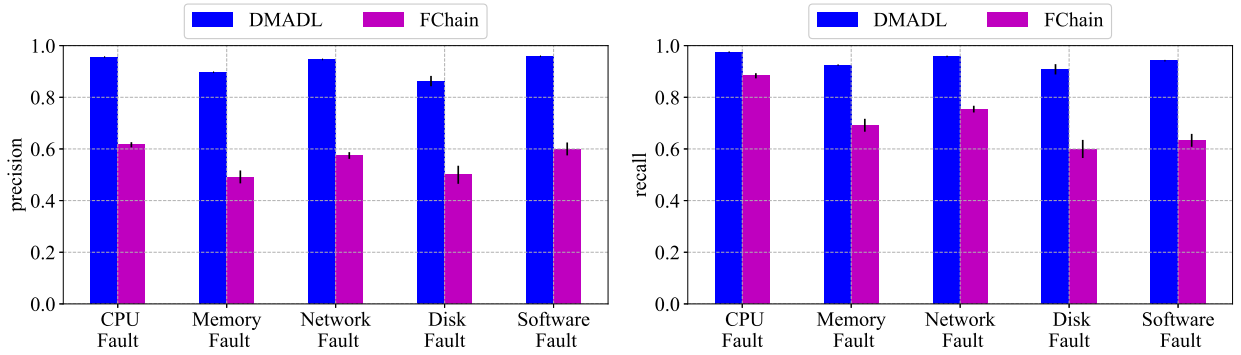
(b) The localization recall

Figure 6.21: The localization performance of DMADL and other methods in Olio web application

6.4.7 Overhead Analysis

The overhead of DMADL agent depends on the number of processed data packets per second. We show the CPU overhead of DMADL agents and the number of processed data packets at different components in Cloud applications.

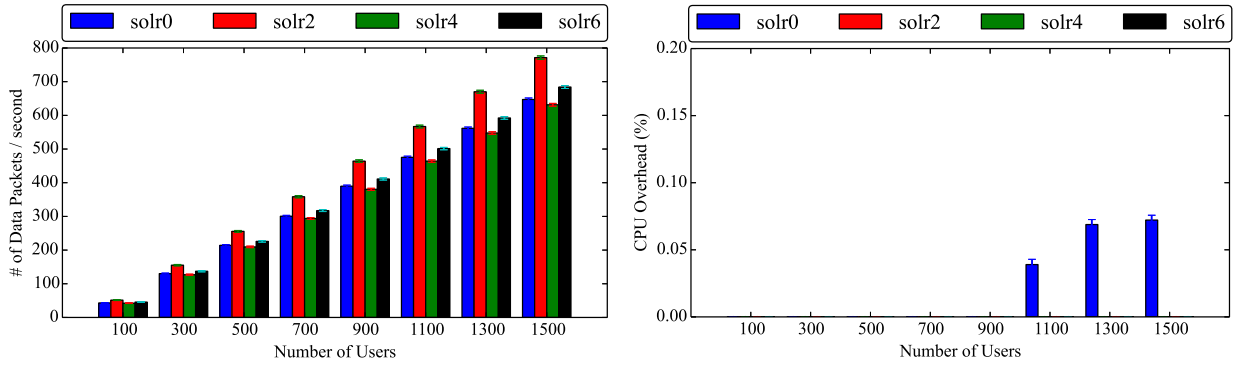
In CloudSuite web search application, the number of clients increases from 100 to 1600. DMADL uses less than 0.6% CPU while processing up to 9000 data packets per second under varying workload intensity



(a) The localization precision

(b) The localization recall

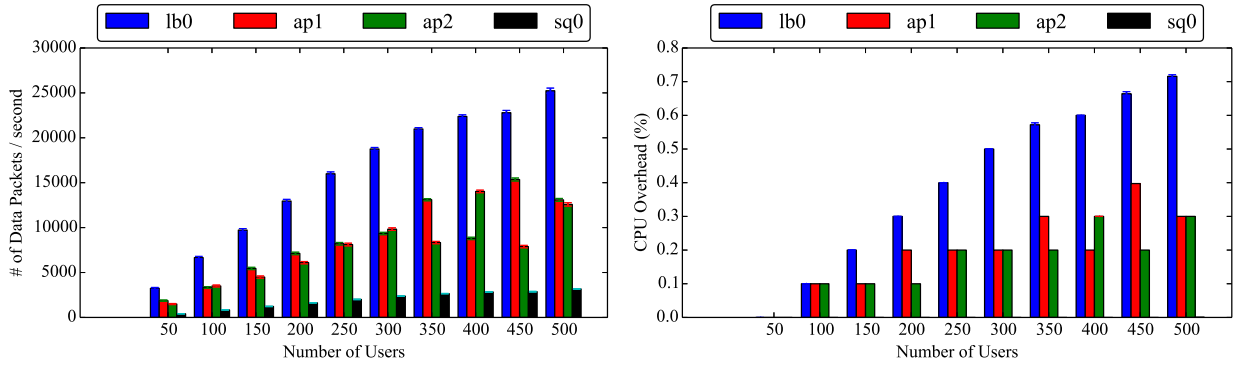
Figure 6.22: The localization performance of DMADL and other methods in MediaWiki application



(a) The number of processed data packets

(b) CPU overhead of DMADL agents

Figure 6.23: The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in CloudSuite web search application.

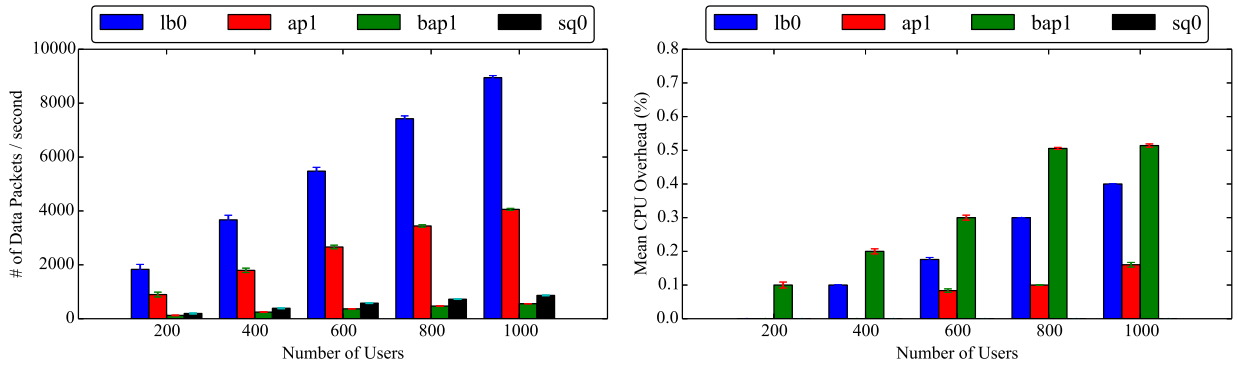


(a) The number of processed data packets

(b) CPU overhead of DMADL agents

Figure 6.24: The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in Olio web serving application.

as shown in Figure 6.23. We also measure the memory overhead of DMADL using "ps" tool in Linux. It always shows 0%. Since, the "ps" tool measures memory cost at a unit of 0.1%. DMADL incurs less than



(a) The number of processed data packets

(b) CPU overhead of DMADL agents

Figure 6.25: The CPU overhead and the number of processed data packets per second by DMADL under varying workload intensity in MediaWiki application.

0.1% memory under varying workload intensity.

In Olio application, the number of clients increases from 50 to 500. The component *lb0* has the highest number of processed data packets per second, but the DMADL agent still has only 0.7% CPU overhead for processing 25000 data packets per second. On average, DMADL costs less than 0.5% CPU resource while processing up to 9,000 data packets per second under varying workload as shown in Figure 6.24. We also measure the memory overhead of DMADL and DMADL incurs less than 0.1% memory under varying workload.

In MediaWiki web search application, the number of clients increases from 100 to 1000. DMADL costs less than 0.5% CPU resource while processing up to 900 data packets per second under varying workload at component *lb0*. DMADL agents of component (*ap1, ap2, sq0*) do not cause negligible CPU overhead. We also measure the memory overhead of DMADL and DMADL incurs less than 0.1% memory cost under varying workload. With less than 1% CPU overhead and negligible memory cost, DMADL could be deployed for anomaly detection and localization in large-scale Cloud applications.

6.5 Summary

DMADL combines the dependency model and response time estimation for anomaly detection, and localization in IaaS Cloud applications. The response time estimated by DMADL depends on the interaction pattern. Now HTTP/2 has been supported by almost all major browsers and web servers. We discuss what the estimated service response time by DMADL means in the interaction between end-users and front-end web servers over HTTP/2.

HTTP2 server push function allows servers to send files before they are requested. It is common that a request for a web page requires other resources such as images, java scripts, style sheets, and so on.

In previous HTTP protocols, the server sends the requested HTML back to clients, and waits for the browser to parse the HTML and issue subsequent requests for associated resources before it can send those resources. Now in HTTP/2, as the server knows the client will eventually request specific resources for a web page, it can send corresponding resources together with the HTML response back to the client in advance. The function allows websites to save page loading time by skipping network round-trips. In this case, the response time estimated by DMADL is the response time for the entire web page including all requested resources.

HTTP/2 multiplexing allows users to fire multiple requests in parallel in a single connection. The browser could fire off requests to get the HTML, CSS, images and corresponding resources together on the same connection without waiting for responses before sending out next requests. The server could respond these requests in any order to prevent "head of line" blocking problem. Similarly, all multiplexed requests for an entire web page are considered as one request, and corresponding responses are considered as one response by DMADL. The response time estimated by DMADL is the response time for the entire web page. If that is how most users use HTTP/2, DMADL can still be used for anomaly detection.

Chapter 7

DMFDL: Dependency Model-based Flow Ratio Analysis for Anomaly Detection and Localization

In this chapter, we present DMFDL, a dependency model-based traffic flow ratio analysis for performance anomaly detection and localization in IaaS Cloud. DMFDL is based on a simple relationship of data flow in any given component of Cloud applications: the number of requests should be almost equal to the number of responses within a given time interval in normal operations. A reasonable time interval corresponds to an acceptable service response time. We define this simple relationship as flow conservation. The flow refers to the transmitted messages from the start of request to the end of response in both directions. Regardless of the complexity of the application structure or unforeseen component behavior, the flow conservation model always holds during normal operations. Even with highly dynamic workload, the flow conservation still holds when a component runs in normal operation. When the number of requests changes, the number of corresponding responses changes accordingly during normal operations. The flow conservation relationship no longer holds at a component when itself or its subsystem has performance anomalies. For example, when the response time increases or the server cannot return any response, many requests cannot have corresponding responses within an acceptable time limit. The number of responses is much less than the number of requests in the same interval. It is usually hard to identify requests and responses from packet streams. DMFDL uses the flow ratio to model the flow conservation relationship between the number of request data packets and the number of response data packets. We show that it is feasible to detect performance anomalies by using the flow ratio to model the flow conservation relationship. DMFDL triggers anomaly alert when the flow ratio violates a dynamic threshold from the

normal profile. DMFDL adapts the flow ratio to consider the past variation and combines the adapted flow ratio with the dependency model to achieve more accurate detection and localization. DMFDL has agents distributed on physical hosts. The agents do not require domain knowledge from operating systems of virtual components or underlying applications. DMFDL agents operate outside virtual components in real time and each agent performs anomaly detection and localization locally.

7.1 DMFDL overview

DMFDL agents consist of 5 major functions: flow monitoring, dependency model extraction, flow ratio model, anomaly detection, and anomaly localization. The operating flow of DMADL agents is shown in Figure 7.1. The flow monitoring function works with packet streams outside virtual components and counts the number of data packets in traffic flows. The dependency model extraction function uses the request flow and the response flow to identify dependency primitives formed by a local component and its dependent components. The flow ratio profiling function models the flow conservation relationship between the request flow and the response flow using an adaptive flow ratio. The anomaly detection function uses the deviation of flow ratios and changes of the local interaction behavior to detect anomalies. The anomaly localization function analyzes the changes in flow ratios of multiple components and uses the pattern of changes in different dependency primitives for identifying the faulty component locally.

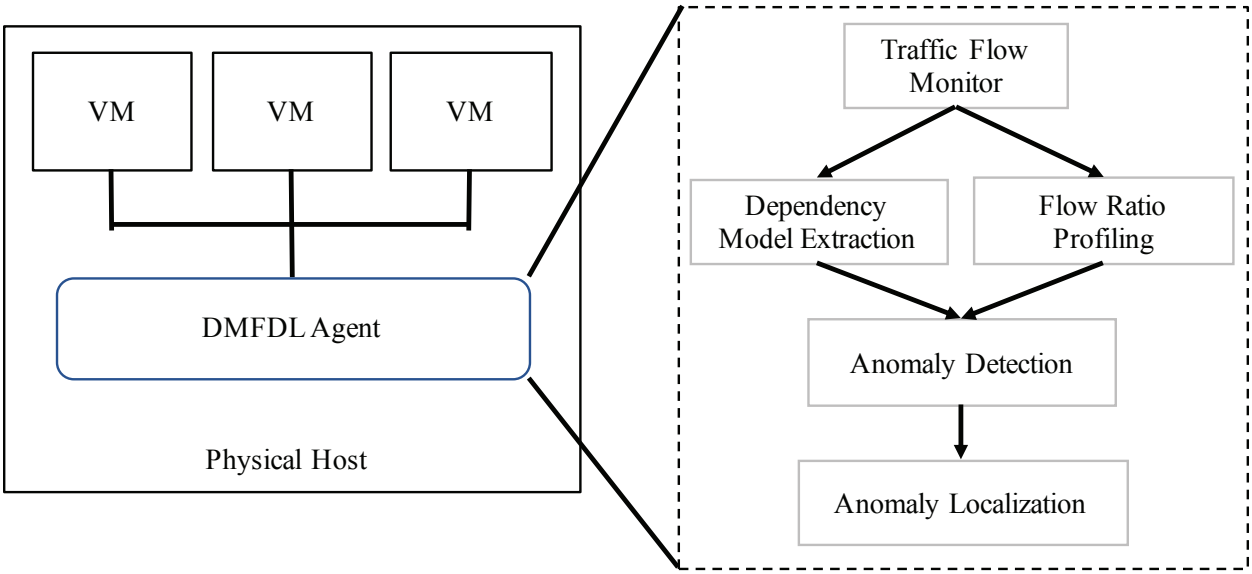


Figure 7.1: The operation diagram of DMFDL for anomaly detection and localization

7.2 Traffic Flow Monitor

DMFDL agents take each component as a black box and monitor network traffic flow outside the component. DMFDL considers only TCP traffic flow as it constitutes the majority of the network traffic in Cloud applications. We filter out control packets that do not contribute to the communication flow of requests and responses. Packets that do not contain TCP payload are treated as control packets, such as pure SYN, FIN, ACK, and RST. Data packets containing TCP payload carries useful data for requests and responses. For each component service, DMFDL agent monitors the following metrics in the interval $(t - \Delta T_f, t)$ (ΔT_f is the sampling interval):

- *Input request flow*: the number of request data packets arrived at the local component.
- *Output response flow*: the number of response data packets departing from the local component.

In Cloud applications, a component usually has multiple dependent components. If the local component contains any dependent component(s), the DMFDL agent also monitors the data flow between each dependent component and the local component.

- *Output request flow*: the number of outgoing request data packets from the local component to the dependent component.
- *Input response flow*: The number of incoming response data packets from the dependent component to the local component.

Figure 7.2 shows the network traffic flow for a local component A. In a interval $(t - \Delta T_f, t)$, we denote the input request flow as $X_A(t)$ and the output response flow as $Y_A(t)$. For its dependent component B, the output request flow is denoted as $X_{AB}(t)$ and the input response flow is denoted as $Y_{AB}(t)$. DMFDL further analyzes the relationship between the input request flow and the output response flow for anomaly detection at local component and its subsystem.

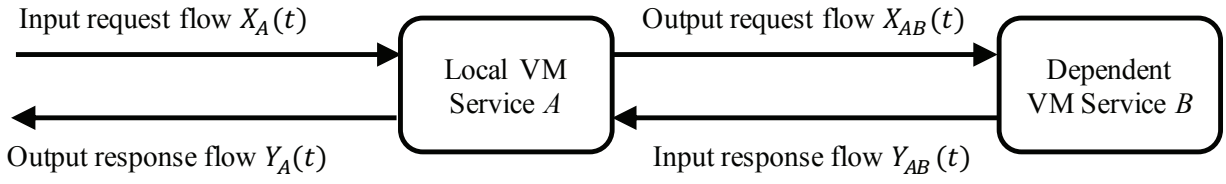


Figure 7.2: The traffic flow monitor for a component service A

7.3 Anomaly Detection

In this section, we first present the flow ratio model to characterize the service performance. Then we show how to use the flow ratio for anomaly detection.

7.3.1 Flow Ratio Model

Given any component in normal operation, request data packets are followed by corresponding response data packets within an acceptable time limit, which corresponds to the maximum service target as defined in service level agreement (SLA). Ideally, we can inspect payload to find out the exact response time and check the correctness of packet contents. However, the inspection of packet content is challenging in high speed channels and packets cannot be easily parsed in encrypted applications.

If we pick the sampling interval $\Delta T_f = SLA$, we observe that request data packets and corresponding response data packets fall into the same interval. When there is an anomaly at the local component or its subsystem, the local response time increases and exceeds the SLA target. the incoming request data packets and corresponding outgoing response data packets cannot be fall into the same monitoring interval. FlowBox [45] used the flow ratio to characterize the relationship, which is defined as the output response flow over the input request flow within a sampling interval. The flow ratio $R(t)$ is defined as the number of outgoing response data packets $Y(t)$ over the number of input data packets $X(t)$ in a sampling interval $(t - \Delta T_f, t)$.

$$R(t) = \frac{Y(t)}{X(t)}$$

When a local component and its subsystem components have performance anomaly, it takes much longer to respond to user requests or it cannot respond to requests properly. The flow ratio is supposed to decrease as incoming request data packets are not responded with a sampling interval. DMFDL shows the importance of the sampling interval in order to use the flow ratio for anomaly detection. FlowBox shows that the flow ratio varies significantly even in normal operation. The variability existing in the flow ratio prevents us from capturing the relationship between the input request flow and the output response flow in normal operation. If the sampling interval is too small, it may produce high false positive in anomaly detection. A large sampling interval obfuscates the difference in flow ratios between normal and abnormal operations. It misses real performance anomalies and causes false negative in anomaly detection. We select a proper sampling interval to balance the false positive and false negative.

In complex Cloud applications, it is not trivial to determine a proper sampling interval for each component service. First, different types of requests and responses contain different number of data packets. Given a component service A, there are N_A different types of requests. We denote the number of data

packets for a request of type i as x_i, and the number of data packets for its response as y_i. Within an observation interval $(t - \Delta T_f, t)$, the number of requests of type i is $\lambda_i(t)$. The flow ratio during each interval can be represented as

$$R(t) = \frac{Y(t)}{X(t)} = \frac{Z(t) + \sum_{i=1}^{N_A} \lambda_i(t) * y_i}{W(t) + \sum_{i=1}^{N_A} \lambda_i(t) * x_i}$$

In above equation, $Z(t)$ is the number of response data packets for requests observed before the current sampling interval; $W(t)$ is the number of request data packets that are not responded in the current interval yet; $\lambda_i(t)$ is a parameter characterized by the workload behavior. It is difficult to capture the dynamics of the flow ratio for each service. From [14], we learn that the flow ratio converges as $\sum_{i=1}^{N_A} \lambda_i(t)$ is larger, where $\sum_{i=1}^{N_A} \lambda_i(t)$ is the number of requests arrived in the current sampling interval that are responded within the same interval. The $Z(t)$ and $W(t)$ are negligible in normal and stable state as most requests are returned within the same interval: $\sum_{i=1}^{N_A} \lambda_i(t) * y_i \gg Z(t)$ and $\sum_{i=1}^{N_A} \lambda_i(t) * x_i \gg W(t)$. The flow ratio in normal operation can be simplified as:

$$R_N(t) = \frac{Y(t)}{X(t)} = \frac{Z(t) + \sum_{i=1}^{N_A} \lambda_i(t) * y_i}{W(t) + \sum_{i=1}^{N_A} \lambda_i(t) * x_i} \approx \frac{\sum_{i=1}^{N_A} \lambda_i(t) * y_i}{\sum_{i=1}^{N_A} \lambda_i(t) * x_i}$$

In abnormal operation, most requests cannot have its responses returned within the same interval (e.g., $\sum_{i=1}^{N_A} \lambda_i(t) \approx 0$). The flow ratio in an abnormal sampling interval is simplified as

$$R_A(t) = \frac{Y(t)}{X(t)} = \frac{Z(t) + \sum_{i=1}^{N_A} \lambda_i(t) * y_i}{W(t) + \sum_{i=1}^{N_A} \lambda_i(t) * x_i} \approx \frac{Z(t)}{W(t)}$$

As the workload is higher, there are more samples in a sampling interval. As a result, the flow ratio capture the normal service behavior more accurately. A low workload usually causes less samples in the sampling interval for computing the flow ratio. The variability of requests and responses cannot model the service behavior in normal operation. The flow ratio depends on the following factors:

- When a service is more complex, the variability of flow ratio across time is larger.
- When the workload mix is more dynamic, the variability of flow ratio across time is larger.
- When the workload intensity increases, the variability of the flow ratio is smaller.

However, the flow ratio proposed in FlowBox is time-ignorant. The flow ratio describes the relationship between requests and responses in a sampling interval in normal operation. But in abnormal operation, responses are no longer for requests in the same interval. For example, some responses in the current interval correspond to those requests in previous time intervals. We cannot predict how the flow ratio changes in abnormal operation. It may be either lower or higher than the flow ratio in normal operation,

Algorithm 4: DMFDL sampling interval selection

input : For a virtual component service A, the time series of input request flow $X_A(t)$ and output response flow $Y_A(t)$ during normal operation. A list of optional sampling intervals $\Delta T_i (i = 1, 2, \cdots)$

output: The proper sampling interval ΔT_f.

for *the sampling interval* ΔT_i **do**
  Compute the flow ratios using the sampling interval ΔT_i during normal operation: $R(t)$;
  Compute the mean of the flow ratio μ_i and the standard deviation σ_i;
  Compute the relative deviation;

$$K_i = \frac{\sigma_i}{\mu_i}$$

end
for $i = 2$ *to* n **do**
  if $\|K_i - K_{i-1} \leq th$ **then**
    break;
  end
end
The proper sampling interval is: $\Delta T_f = \Delta T_i$;

and it may be similar to the flow ratio in normal operation. It depends on $Z(t)$ and $W(t)$. The flow ratio may be against our expectation that the ratio between the number of response data packets over the number of request data packets should be lower as requests take longer to be processed in abnormal operation. It is because $W(t)$ is not negligible in abnormal operation. If we know the value of $W(t)$, we could remove it from $Y(t)$ and the resulting flow ratio would be accordant with our expectation.

In each sampling interval, some response data packets are not for those requests in the current window. It is difficult to predict or control the workload of each component in Cloud applications. The possible solution is to properly increase the window size under lower workload to have more samples in the window. The flow ratio could more accurately characterize the normal service behavior. If the workload intensity for a component is high enough, we choose the ideal sampling interval size as the maximum acceptable response time for the underlying service.

It is not trivial to specify the service level target (SLA) for each component. DMFDL gives another algorithm 4 to automatically determine the proper sampling interval for the flow ratio model. To obtain the proper sampling interval, DMFDL chooses among a set of optional sampling intervals, and selects the proper sampling interval as the interval that achieves stable flow ratio in normal operation.

Training stage

DMFDL trains the system under normal operation for a certain period to obtain the normal profile $R_N(t)$ for the flow ratio of each component. Based on the distribution of the flow ratio in the training stage,

DMFDL determines two thresholds for the flow ratio in normal operation: a lower bound R_{lb} and an upper bound R_{ub}. If the flow ratio in a sampling interval is either below the lower threshold or above the upper threshold, DMFDL consider the flow ratio in the interval as suspicious.

When the service is normal, the flow ratio in most sampling intervals falls within the normal range: $R_{lb} \leq R(t) \leq R_{ub}$. In each interval $(t - \Delta T_f, t)$, DMFDL first checks the latest flow ratio $R(t)$.

If $R(t) > R_{ub}$, it means that there are more response data packets than request data packets. The reason is that many response data packets are for those requests in previous intervals before the current interval. It infer that the service may be abnormal before this interval.

If $R(t) < R_{lb}$, it means there are less response data packets than request data packets in the current window. There are two possible causes:

- The service is still normal, but a lot of requests arrive at the end of the current interval and most of their response data packets are not responded within the same interval. But the response data packets would be responded within the next interval.
- The service is abnormal, requests take much longer to be processed. As a result, most of their response data packets cannot be responded within the same interval. The response time is much larger than the SLA.

To differentiate between these two cases, DMFDL further uses the input request flow and the output response flow in the current interval to make a decision. DMFDL starts to aggregate input request flow and the output response flow from the suspicious interval until the flow ratio becomes no longer suspicious.

- If the suspicious flow ratio is caused by the first cause, many requests are not responded within the current interval. But they would be responded within the next interval, The flow ratio becomes normal again in the next interval.
- If the suspicious flow ratio is caused by anomaly, the response time is much larger than the sampling interval. The requests in the current interval would not be observed within the upcoming interval as well. As a result, the flow ratio may still be suspicious in several upcoming intervals. As a result, DMFDL observes a contiguous sequence of suspicious flow ratios until the service is normal again.

7.3.2 Detection Algorithm

In a dynamic system, it is not unusual to have occasional upward or downward spikes in the flow ratio even when the service is normal. As a result, the flow ratio can fall outside the normal profile sometimes. It is easy to produce some false positives if the detection algorithm is based on the flow ratio within a

single interval. However, a sustained observation of suspicious flow ratios are unusual and the service probably has performance anomaly. Rather than putting threshold directly on the aggregate flow ratio, DMFDL triggers anomaly alert based on the behavior of flow ratios within a recent decision window. In real time, DMFDL uses a moving window of W_d flow ratios to capture the pattern of the flow ratio. In DMFDL, the ratio deviation $S_d(t)$ is defined as the sum of the deviations in the decision window.

$$S_d(t) = \sum_{i=0}^{W_d-1} \left[R(t-i) - \overline{R_N}\right]$$

DMFDL compares the flow ratio deviation in current decision window with profiled flow ratio deviation in normal operation. DMFDL decides whether the system state is suspicious if the ratio deviation exceeds the profiled normal threshold.

DMFDL uses a window of latest W states and combines it with the dependency model to trigger anomaly alert: (1) The number of suspicious interval exceeds a tolerable threshold; (2) There are dependent components no longer has data flow communication with the local component. The procedure of DMFDL anomaly detection is described in Algorithm 5.

7.4 Anomaly Localization

DMFDL performs component-level anomaly localization in a distributed way. Each DMFDL agent analyzes the possible source of anomaly locally in the local component and its dependent components. Each DMFDL agent reports the local result to a central administrator, and the DMFDL administrator combines all local results together and determines the faulty component.

7.4.1 Local Interaction Behavior Characterization

Due to the fault propagation, the flow ratio of a local component decreases due to three possible causes: (1) the upstream components; (2) the local component itself; or (3) the local component's subsystem. If the local component has dependent components and the local anomaly is caused by the upstream components of the local component, the interaction between the local component and its dependent components is still normal. If the local anomaly is caused by the local component or its subsystem components, the interaction between the local component and its dependent components are affected due to anomaly propagation. DMFDL agents use the interaction between the local component and its dependent components to determine whether the local anomaly is caused by the upstream components, the local component, or its subsystem components.

Algorithm 5: DMFDL anomaly detection

input: $X_A(t), Y_A(t), W, DW, LB_A(t), UB_A(t), CX_A(t), CY_A(t), \mu_A(t)$
Initialization: $CX_A(t) = 0, CY_A(t) = 0, R(t) = \mu_A(t)$;
for *each timestamp t* **do**
  $\begin{cases} CX_A(t) = CX_A(t) + X_A(t) \\ CY_A(t) = CY_A(t) + Y_A(t) \end{cases}$;
  if $CX(t) == 0$ **then**
    $R_A(t) = R_A(t-1)$;
  else
    Compute the flow ratios: $R_A(t) = \frac{CY(t)}{CX(t)}$;
    if $R_A(t) >= LB_A(t)$ **then**
      $\begin{cases} CX_A(t) = 0 \\ CY_A(t) = 0 \end{cases}$;
    end
  end
  if $R_A(t) >= UB_A(t)$ **then**
    $R_A(t) = \frac{R_A(t-1)+R_A(t)}{2}$;
  end
  if $\sum_{i=t-W}^{t} (R_A(i) - \mu_A(t)) \geq F_A(t)$ **then**
    $S(t) = 1$;
  end
  if $\sum_{i=t-W}^{t} (R_A(i) - \mu_A(t)) \geq F_A(t)$ **then**
    $S_A(t) = 1$;
  end
  if $\sum_{i=t-DW}^{t} (S_A(t)) \geq DW/2$ **then**
    Trigger an anomaly alert for service A;
  end
end
;

DMFDL agents characterizes the local interaction behavior between the local component and each dependent component using the request flow ratio and the response flow ratio. For a dependent component B, the request flow ratio $R(X_A, X_AB, t)$ is defined as the ratio between the output request flow $X_AB(t)$ and the local input request flow $X_A(t)$. The response flow ratio $R(Y_A, Y_AB, t)$ is defined as the ratio between the local output response flow $Y_A(t)$ and the input response flow $Y_AB(t)$.

$$R(X_A, X_{AB}, t) = \frac{X_{AB}(t)}{X_A(t)}, R(Y_A, Y_{AB}, t) = \frac{Y_A(t)}{Y_A(t)},$$

DMFDL agents use the request flow ratio and the response flow ratio for anomaly localization at each component of Cloud applications.

7.4.2 No Dependent Component

When the local component has no dependent component, the anomaly is caused by either the fault at the local component or the fault at its upstream tier. DMFDL cannot report any localization result due to lack of information about upstream components. In an extreme case where the DMFDL agent observes only input request flow but without output response flow, the flow ratio decreases to 0 and the DMFDL agent reports that the local component is faulty.

7.4.3 Single Dependent Component

When the local component has a single dependent component, DMFDL analyzes different causes and their difference for localization analysis.

1. If the anomaly is caused by upstream components, it does not affect the interaction between the local component and the dependent component. The request flow ratio and the response flow ratio do not change.
2. If the anomaly is caused by the local component, the local input request flow decreases due to TCP propagation and the local output response flow also decreases as the local response time increases. The dependent component still processes requests normally and returns responses to the local component. But those responses wait longer at the local component. The response flow ratio usually increases compared to normal operation.
3. If the anomaly is caused by the dependent component or its subsystem components, the local component is write-blocked when it tries to send requests to the dependent component. The request flow ratio decreases. After receiving responses from the dependent component, the local component returns those responses to upstream components normally. The response flow ratio is still normal.

From above analysis, when both the request flow ratio and the response flow ratio have big changes compared to normal operation, the DMFDL agent determines the local component as faulty. If the request flow ratio and the response flow ratio almost do not change, the DMFDL agent determines that the local component and its dependent component as normal. If the request flow ratio changes and the response flow ratio almost does not change, the DMFDL agent determines that the dependent component as faulty.

7.4.4 Distributed Dependency Primitive

In distributed dependency setup, dependent components are usually independent from each other. The local component still works properly if some dependent components are still normal. For a local compo-

nent with a distributed set of dependent components, the sum of output request flow to all dependent components should almost equal the local input request flow. The sum of input response flow should almost equal the local output response flow.

1. If the anomaly is caused by its upstream components, the interactions between the local component and all dependent components are still normal. The request flow ratios and response flow ratios of all dependent components almost do not change.
2. If the anomaly is caused by the local component, the local fault has similar impact on the output request flow to all dependent components. The requests that should be directed to its dependent components wait longer at the local component. The responses that should be returned from the local component wait longer at the local component. The request flow ratios and response flow ratios of all dependent components should change similarly.
3. If the anomaly is caused by a dependent component or its subsystem components, requests that should be sent to the faulty dependent component wait longer at the local component. The requests that are destined to other normal dependent components are still distributed normally. Therefore, less requests arrived at the local component are directed to the faulty component. The request flow ratio of the faulty dependent component decreases, and the request flow ratios of normal dependent components increase. Most responses returned from the local component are coming from normal dependent components. The response flow ratio of the faulty dependent component increases compared to that in normal operation. The response flow ratios of other normal dependent components decrease.

From above analysis, when the request flow ratio of a dependent component decreases and its response flow ratio increases, the DMFDL agent determines the dependent component as faulty. When the request flow ratios and response flow ratios of all dependent components change similarly compared to those in normal operation, the DMFDL agent determines the local component as faulty.

7.4.5 Concurrent Dependency Primitive

In concurrent dependency primitive, the local component sends requests to all dependent components concurrently. A faulty dependent component affects the interaction between the local component and all dependent components. DMFDL analyzes the effect of three different causes for the local component with a concurrent set of dependent components.

1. If the anomaly is caused by its upstream components, the interactions between the local component and all dependent components are still normal. The request flow ratios and response flow ratios almost do not change.

2. If the anomaly is caused by the local component, requests arrived at the local component take longer to reach its dependent components. The request flow ratios of all dependent components change similarly. It also takes longer time for the local component to return responses back to the upstream components after receiving responses from all dependent components. The response flow ratios of all dependent components change similarly as the impact on all dependent components are similar.

3. If the anomaly is caused by a faulty dependent component in the concurrent set, the local component is write-blocked when it tries to send requests to the faulty dependent component due to the TCP propagation. The local waiting time increases, and the request flow ratio of the faulty dependent component changes the most. The request flow ratios of other normal dependent components also change, but not as much as the faulty dependent component. When the local component receives responses from normal dependent components, it still waits for responses from the faulty dependent component. The response flow ratio of normal dependent components changes much more than the response flow ratios of the faulty dependent component.

From above analysis, when the request flow ratios and the response flow ratios of all dependent components change with similar scale compared to those in normal operation, the DMFDL agent determines the local component as faulty. If the request flow ratios and the response flow ratios of all dependent components almost do not change, the DMFDL agent determines that the local component and its subsystem components are normal. If the request flow ratio of a dependent component changes much more than those of other dependent components, and the response flow ratio of changes less than those of other dependent components, the DMFDL agent determines the dependent component as faulty.

7.4.6 Composite Dependency Primitive

In composite dependency primitive, the local component sends requests to all dependent components sequentially. A faulty dependent component usually delays the interaction between the local component and those dependent components that execute after the faulty component.

1. If the anomaly is caused by its upstream components, the interactions between the local component and all dependent components are still normal. The request flow ratios and response flow ratios of all dependent components almost do not change.

2. If the anomaly is caused by the local component, requests take longer to reach its dependent components. The request flow ratios of all its dependent components change with similar scale. It also takes longer time for the local component to return responses after receiving responses from all dependent components. The response flow ratios of all dependent components change similarly.

3. If the anomaly is caused by a dependent component or its subsystem components in the composite set. The local component is write-blocked when it tries to send requests to the faulty dependent component due to the TCP propagation. The requests that should be sent to the faulty dependent component wait longer time at the local component. The request flow ratio of the faulty dependent component changes more than that of dependent components that execute before the faulty dependent component. As the faulty dependent component takes longer to process requests, it also takes longer time for the local component to call dependent components that execute after the faulty component. The request flow ratios of dependent components that execute after the faulty component change more than that of the faulty component. The delay between the input response flow from those dependent components that execute before the faulty component and the local output response flow changes. The response flow ratios of dependent components that execute before the faulty dependent component change more than that of the faulty dependent component.

From above analysis, if the request flow ratios and the response flow ratios of all dependent components change with similar scale compared to those in normal operation, the DMFDL agent determines the local component as faulty. If the request flow ratios and the response flow ratios almost do not change, the DMFDL agent determines the local component and its subsystem components as normal. If the request flow ratio of a dependent component changes much more than that of dependent components that execute before it, and the response flow ratios of dependent components execute before it changes much more than those of dependent components that execute after it, the DMFDL agent determines the dependent component and its subsystem components as faulty.

Each DMFDL agent reports only the localization result of the local component and its dependent components. Finally, DMFDL has a central administrator summarize localization results of all agents and determines the faulty component in Cloud applications.

7.5 Experimental Evaluation

the performance of DMFDL using modern Cloud applications. We focus on 2 aspects: (I) How does the flow ratio changes under varying workload scenarios in different Cloud applications? (II) How effective is

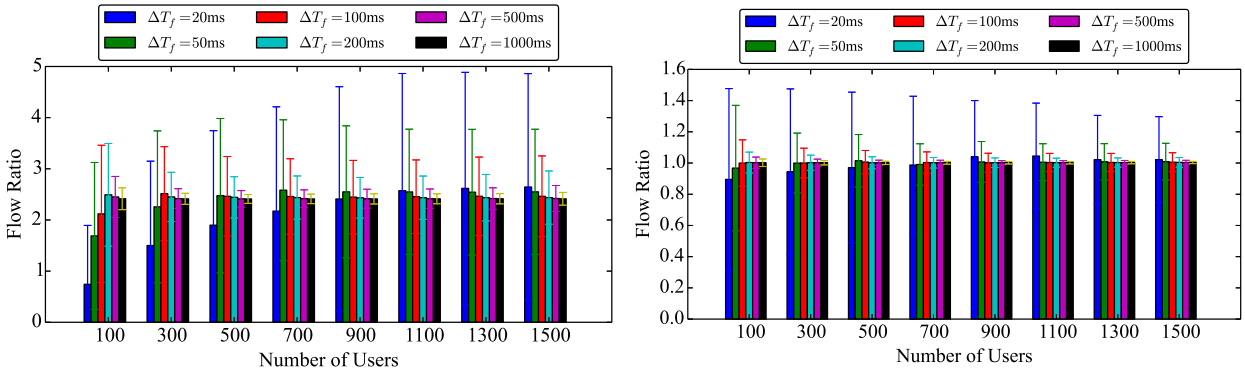
DMFDL in detecting various kinds of performance anomalies in different Cloud applications? (III) How effective is DMFDL in finding out the root cause of performance anomalies?

7.5.1 Flow Ratio Stability Analysis

The sampling interval is important to use the flow ratio for modeling the service performance normal operation. We use varying sampling intervals to compute the flow ratio in different Cloud applications.

CloudSuite Web Search Application

Figure 7.3 shows the flow ratio of different components with different sampling intervals under varying workload intensity in CloudSuite web search application. The flow ratio becomes more stable as the sam-



(a) The flow ratio at component *solr0*

(b) The flow ratio at component *solr6*

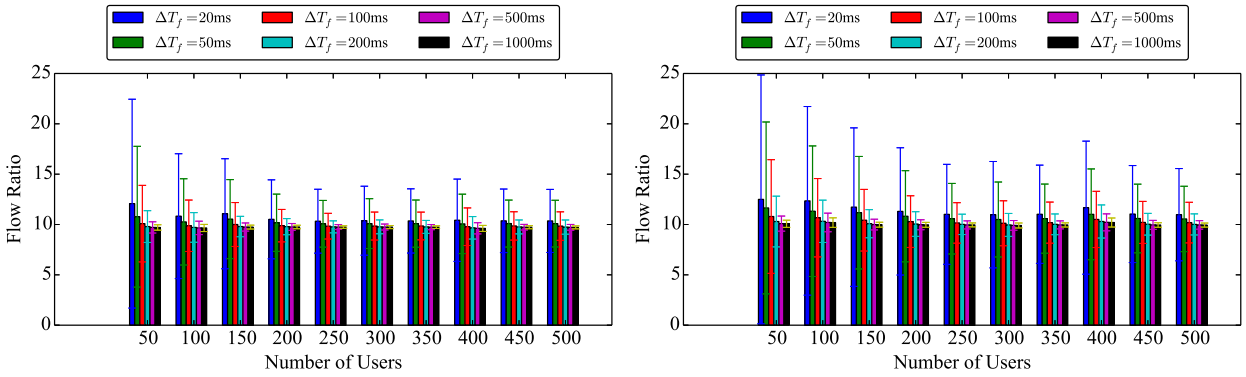
Figure 7.3: The stability of flow ratio with different sampling intervals under varying workload intensity in CloudSuite web search application

pling interval increases. The flow ratio of all components suffers very large fluctuation as the sampling interval $\Delta T_f \leq 200$ms as the standard deviation is almost as large as the mean flow ratio. But the mean flow ratio is almost stable when the sampling interval $\Delta T_f \geq 500$ms across varying workload intensity. Comparatively, the sampling interval has a larger impact on the flow ratio of the index component *solr0* than the flow ratio of its dependent component *solr6*. It is because the response time of the index component has much larger variance. The flow ratio of component *solr6* is stable enough as the sampling interval increases $\Delta T_f \geq 100$ms. It is because the response time of component *solr6* is usually smaller than 100ms. The result of other dependent components of *solr0* is similar to *solr6*.

Olio Web Application

The stability of the flow ratio is important for performance anomaly detection and localization under different workload scenarios. Figure 7.4 shows the flow ratio of different components in the Olio web

application with different sampling intervals under varying workload intensity. The flow ratio becomes



(a) The flow ratio at component *lb0*

(b) The flow ratio at component *ap1*

(c) The flow ratio at component *ap2*

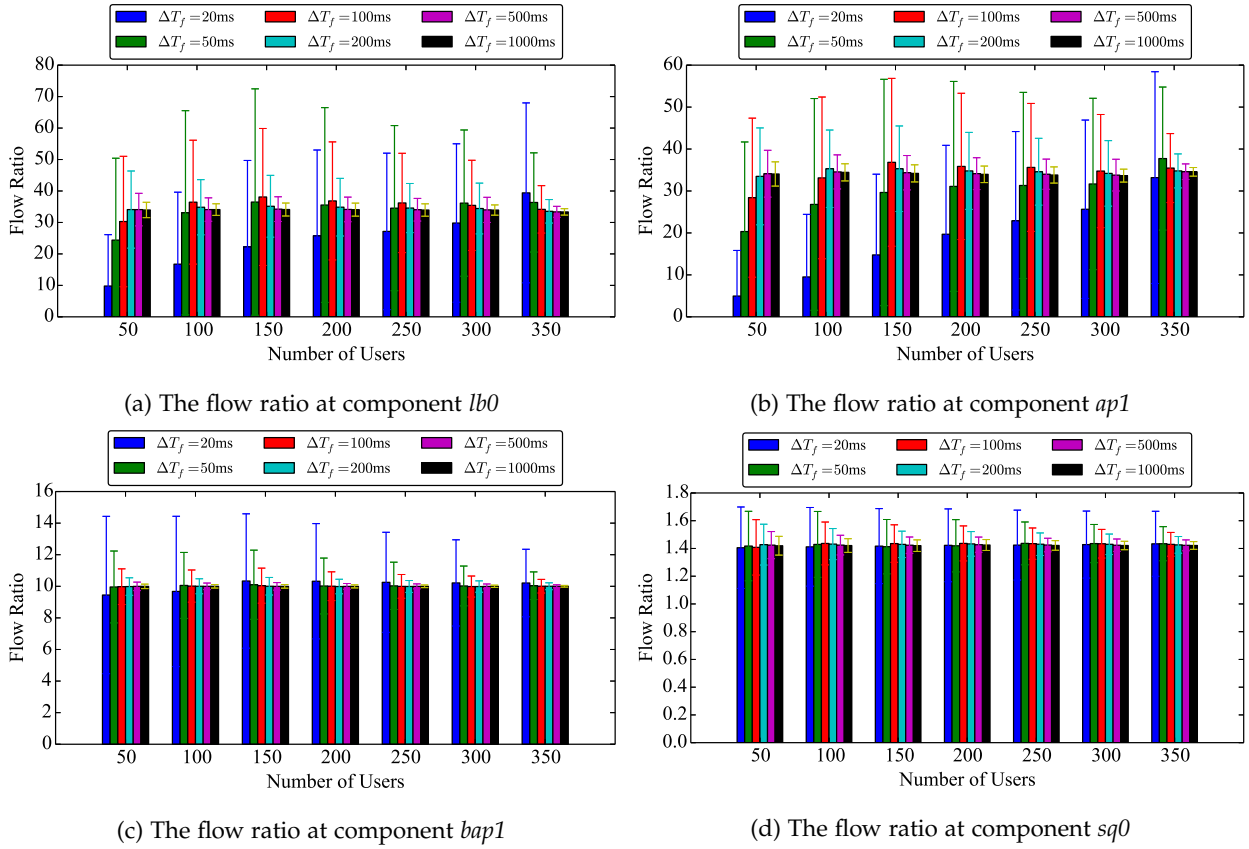
(d) The flow ratio at component *sq0*

Figure 7.4: The stability of flow ratio with different sampling intervals under varying workload intensity in Olio web application

more stable as the sampling interval increases. The flow ratio of all components suffers large fluctuation as the sampling interval $\Delta T_f \leq 100$ms as the standard deviation is almost as large as the mean flow ratio. But the mean flow ratio is stable when the sampling interval $\Delta T_f \geq 200$ms across varying workload intensity. For two application servers (*ap1* and *ap2*), the flow ratio has very large fluctuation when $\Delta T_f < 500$ms. It is stable with small fluctuation when $\Delta T_f \geq 500$ms. Comparatively, the sampling interval has a larger impact on the flow ratio of components *lb0*, *ap1*, and *ap2* than on the flow ratio of component *sq0*. It is because the response time of the load balancer and two application servers has much larger variance. The flow ratio of database component *sq0* is stable enough as the sampling interval changes. It is because the response time of component *sq0* is usually smaller than 10ms.

MediaWiki Application

Figure 7.5 shows the flow ratio at multiple components (*lb0*, *ap1*, *bap1*, *sq0*) with different sampling intervals under varying workload intensity in MediaWiki application. The flow ratio becomes more stable



(a) The flow ratio at component *lb0*

(b) The flow ratio at component *ap1*

(c) The flow ratio at component *bap1*

(d) The flow ratio at component *sq0*

Figure 7.5: The stability of flow ratio with different sampling intervals under varying workload intensity in MediaWiki application

as the sampling interval increases. The flow ratio of all components suffers very large fluctuation as the sampling interval $\Delta T_f \leq$ 100ms as the standard deviation is almost as large as the mean flow ratio. But the mean flow ratio is almost stable when the sampling interval $\Delta T_f \geq$ 200ms across varying workload intensity. For two application servers (*ap1* and *ap2*), the flow ratio has very large fluctuation when $\Delta T_f <$ 500ms. It is stable with small fluctuation when $\Delta T_f \geq$ 500ms. Comparatively, the sampling interval has a larger impact on the flow ratio of components *lb0*, *ap1*, and *ap2* than the flow ratio of components *bap1*, *bap2*, and *sq0*. It is because the response time of the load balancer and two application servers has much larger variance. For the two computational servers (*bap1* and *bap2*), the flow ratio is already stable enough as $\Delta T_f \geq$ 50ms. The flow ratio of database component *sq0* is stable enough as the sampling interval changes. It is because the response time of component *sq0* is usually smaller than 10ms.

In summary, the sampling interval is important parameter to obtain stable flow ratio to model service performance in normal operation. A small sampling interval results in unstable flow ratio as requests cannot have corresponding responses within the same interval. The obtained flow ratio has large fluctuation, and it obfuscates the difference in flow ratio between normal and abnormal operation. A large sampling

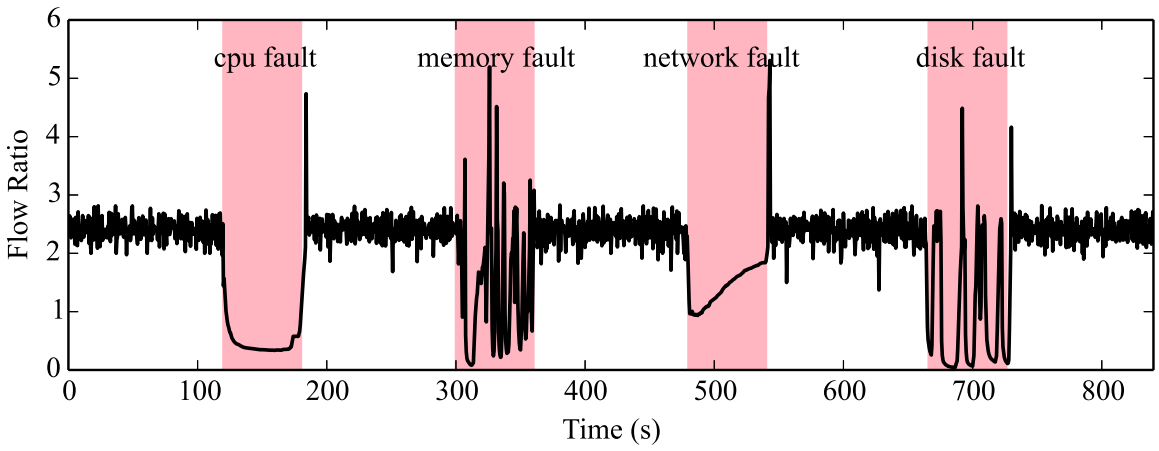


Figure 7.6: The flow ratio for resource fault detection in CloudSuite web search application

interval is more likely to produce stable flow ratio with small fluctuation. The flow ratio would be good for anomaly detection. However, an excessively large sampling interval would cover the abnormal behavior as most requests have their responses within the same interval even their response time is abnormal. DMCDL automatically selects the smallest sampling interval that achieves a stable-enough flow ratio for anomaly detection.

7.5.2 Resource Fault Detection

Figure 7.6 shows the flow ratio at component solr0 of CloudSuite web search application. Figure 7.7 shows the flow ratio at the load balancer lb0 in Olio web application. Figure 7.8 shows the flow ratio at the load balancer lb0 of the MediaWiki application.

- During the CPU fault, the component becomes slower in processing arriving requests. The response time increases, and the flow ratio decreases.
- During the memory fault, each request requiring access to memory suffers much longer delay as other processes keep spinning on "malloc()/free()" functions. When other processes free memory, the component gets enough memory to process the request and return the response. When other processes request for memory, the component gets stuck due to lack of memory. The input flow and output flow suffer frequent fluctuations during this fault.
- During the network fault, outgoing packets wait longer at the local network interface. The response time increases, and the output response flow no longer follows the input request flow within the normal sampling interval.

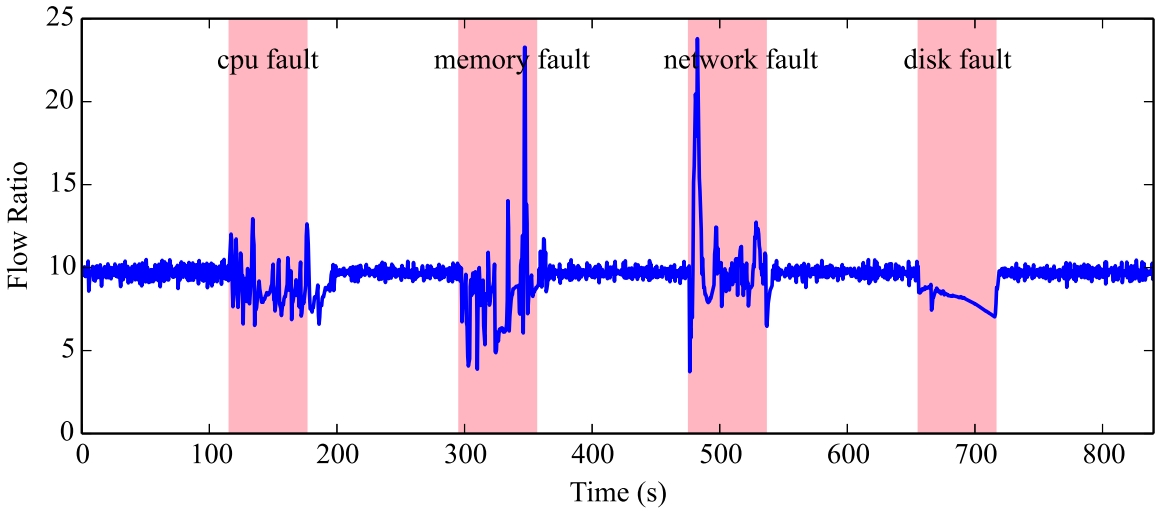


Figure 7.7: The flow ratio for resource fault detection in Olio web application

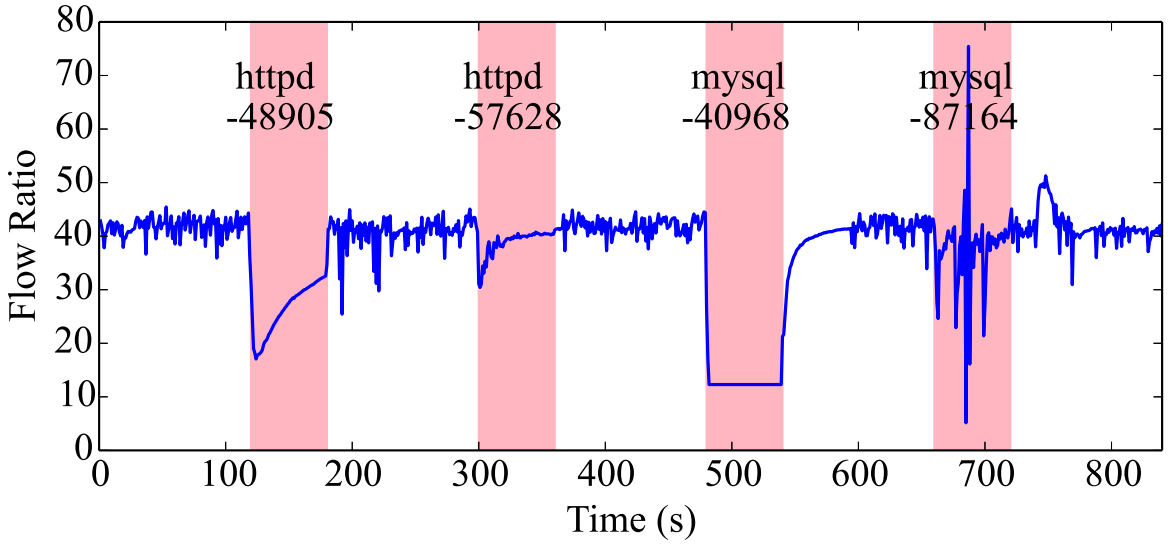


Figure 7.8: The flow ratio for resource fault detection in MediaWiki application

- During the disk fault, the request gets stuck and the server cannot process requests. In the CloudSuite web search application, requests have to fetch data from disks. When the disk fault causes disk read/write operation to be blocked, the component cannot return any response for incoming requests. The output flow is always 0 and it forces client connections to close. The flow ratio becomes very low. For MediaWiki and Olio, their requests involve very few disk accesses, and the disk fault has lower impact on the flow ratio.

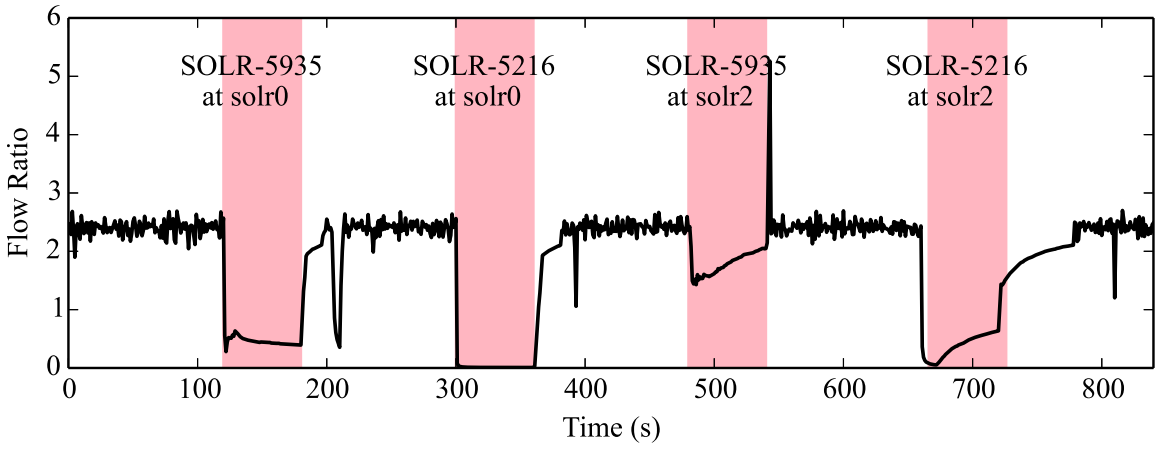


Figure 7.9: The flow ratio for performance bug detection in CloudSuite web search application

7.5.3 Common Software Fault Detection

Figure 7.9 shows the flow ratio at component *solr0* when two software bugs are injected to two components in CloudSuite web search application. The bug SOLR-5935 is injected to the component *solr0* during 120-180s, and *solr2* during 480-540s. The bug SOLR-5216 is injected to component *solr0* during 300-360s, and *solr2* during 760-820s.

- The bug SOLR-5935 at the component *solr0* causes some threads to fall into the deadlock. The component cannot return responses for incoming requests. Requests wait in the component until the deadlock is released. The bug SOLR-5935 at *solr2* does not necessarily cause the local service down. It takes longer to process incoming requests.

- The bug SOLR-5216 at the local component *solr0* causes java processes to hang out. The component cannot process incoming requests. Requests have to wait at the component until java processes recover. The bug SOLR-5216 at the dependent component *solr2* does not necessarily cause the local service down. As all the dependent components form a concurrent dependency primitive. The local component waits for responses from all dependent components until timeout. Finally, *solr0* sends incomplete responses to clients.

Figure 7.10 shows the flow ratio at the load balancer *lb0* when common software faults are injected to different components of Olio web application. Figure 7.11 shows how flow ratio changes at component *lb0* when common performance bugs are injected at different components of MediaWiki application.

- Bug HTTPD-48905: the bug injected into the load balancer *lb0* causes some child processes to hang. Those hanged processes cannot be able to return responses for incoming requests, but other normal

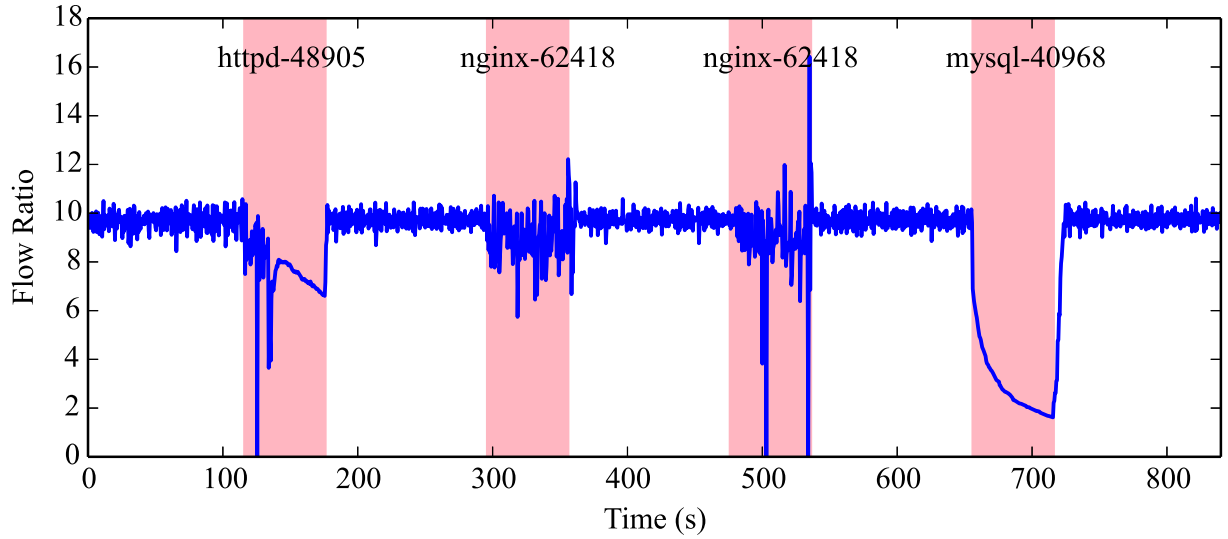


Figure 7.10: The flow ratio for performance bug detection in Olio web application

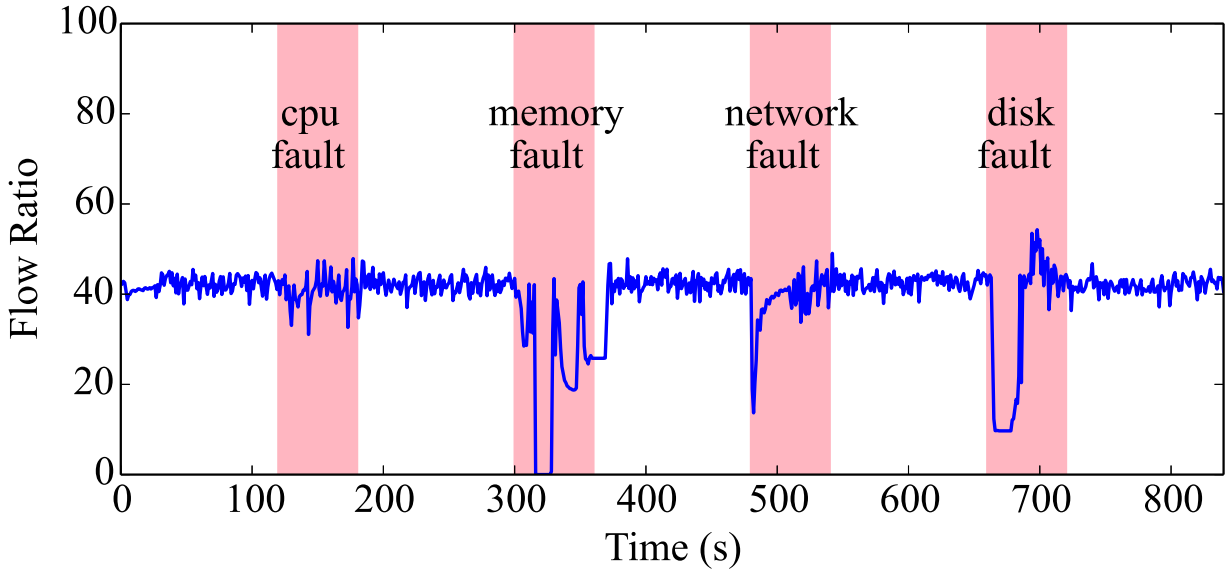


Figure 7.11: The flow ratio for performance bug detection in MediaWiki application

processes still work correctly. Many incoming requests no longer have responses going out. The flow ratio decreases compared to normal operation.

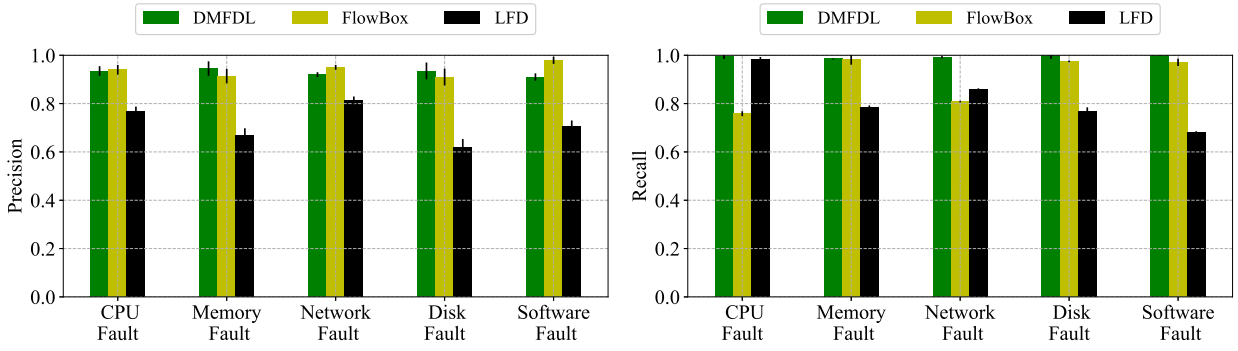
- Bug Nginx-62418: the bug injected into the Apache load balancer lb0 causes local child processes to crash. The component cannot be able to return responses for incoming requests. Requests have to wait at the component until the deadlock is released. After Nginx worker processes at *ap1* crash, the load balancer directs requests to the other server *ap2*. The flow ratio decreases but less obvious compared to the bug at the load balancer because most incoming requests are handled by the normal

application server.

- Bug MySQL-40968: the bug causes MySQL process to hang without any error message. The component would not be able to return any response for incoming requests. Requests have to wait in the component until the TCP connection timeout. The Nginx application servers returns incomplete response to the load balancer, which further sends responses back to clients. The output flow decreases a lot during the faulty period. The flow ratio decreases

7.5.4 Anomaly Detection Performance

We show the performance of DMFDL for anomaly detection in different Cloud applications: CloudSuite web search application in Figure 7.12, Olio application in Figure 7.13, MediaWiki application in Figure 7.14. We perform a large number of experiments under varying workload intensities. During the experiment, we inject different kinds of faults into different components. We compare DMFDL with two other black-box anomaly detection methods LFD [32], and FlowBox [45].

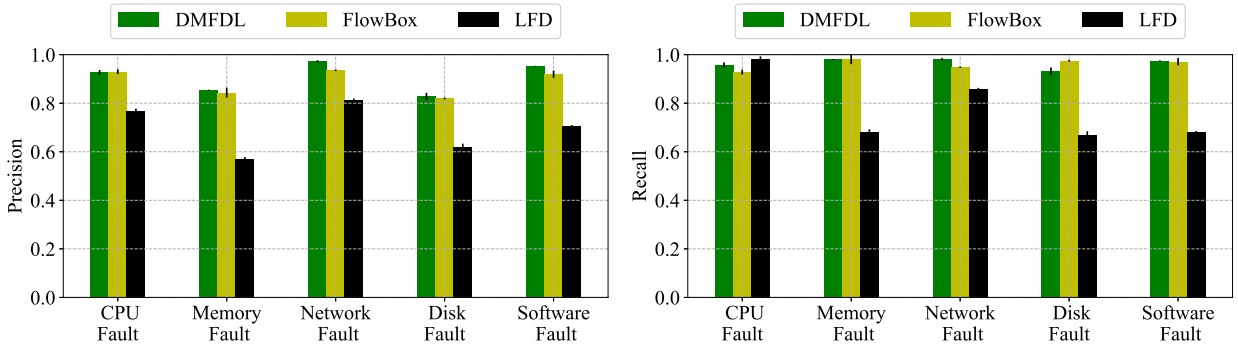


(a) The detection precision

(b) The detection recall

Figure 7.12: The detection performance of DMFDL and other methods in CloudSuite web search application.

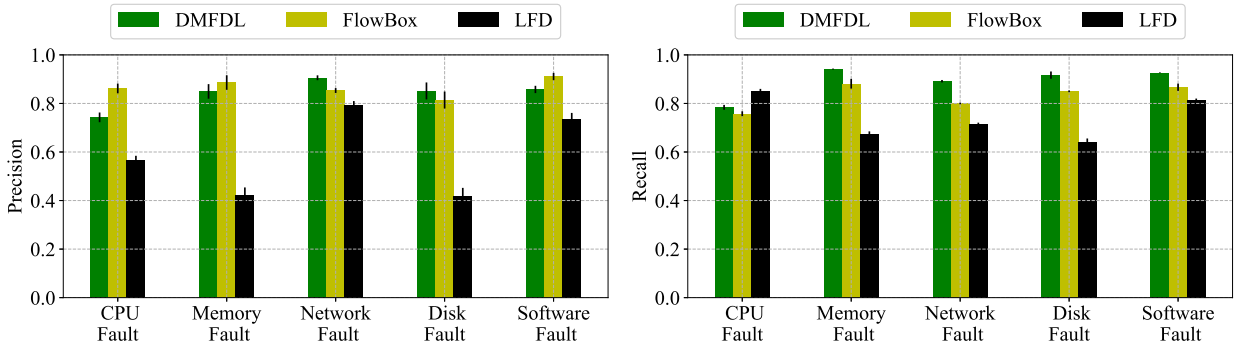
We observe that both DMFDL and FlowBox achieves high detection precision and low false positive for different faults. DMFDL also achieves lowest false negative for different faults. FlowBox has high false negative while detecting CPU and network faults. We check into those cases and found that FlowBox misses many CPU and network faults when the workload intensity is low. When the workload intensity is very low, the limited capacity is enough to process requests. Another reason is that a lower workload intensity causes the common flow ratio to be more unpredictable. But with the adjusted flow ratio, DMFDL still detects it well. LFD has the worst detection performance for all different faults. The main reason is that it selects the highest correlation between the user-space CPU utilization and other resource utilization as the metric to characterize the service performance. It is prone to high false positive and high



(a) The detection precision

(b) The detection recall

Figure 7.13: The detection performance of DMFDL and other methods in Olio web application



(a) The detection precision

(b) The detection recall

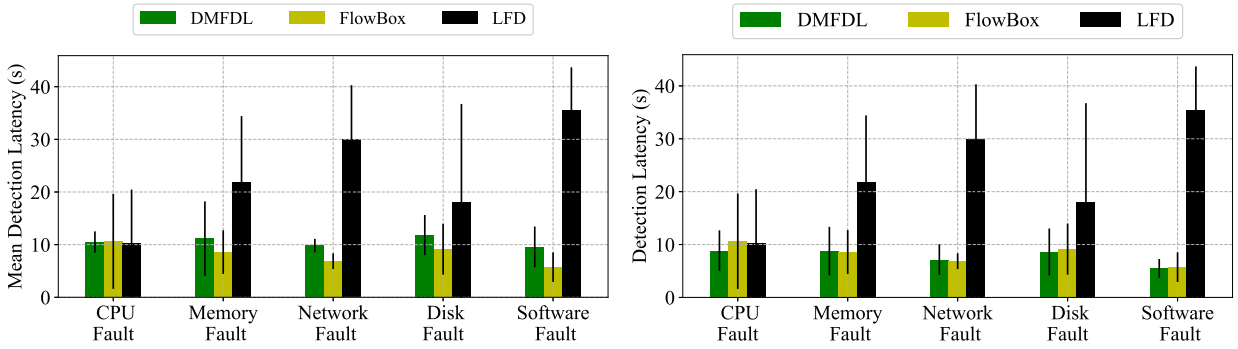
Figure 7.14: The detection performance of DMFDL and other methods in MediaWiki application

false negative. There are two main reasons. One is that if the server does not use too much user-space CPU utilization, the correlation is not meaningful. The other reason is that some resource utilization metrics are still highly correlated with the user-space CPU utilization even when the server has different faults.

Figure 7.15 shows the mean detection latency for different kinds of faults in the CloudSuite web search application. The error bar denotes the standard deviation of the detection latency. DMFDL and FlowBox achieves consistently small detection latency. LFD has much longer delay at detecting memory, network and software faults. It is because LFD uses a long-time window for computing the correlation. The fault usually causes LFD correlation to decrease gradually and the sliding window has to move long enough to trigger anomaly alert.

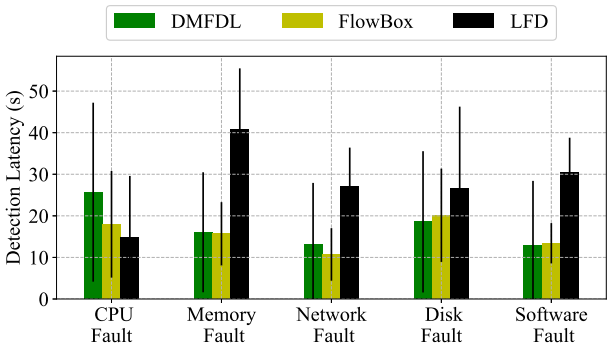
7.5.5 Anomaly Localization Case Study

In this section, we show how DMFDL does anomaly localization when performance anomalies are injected at different components in Cloud applications.



(a) The detection latency in CloudSuite

(b) The detection latency in Olio



(c) The detection latency in MediaWiki

Figure 7.15: The detection latency of DMFDL and other methods in different Cloud applications

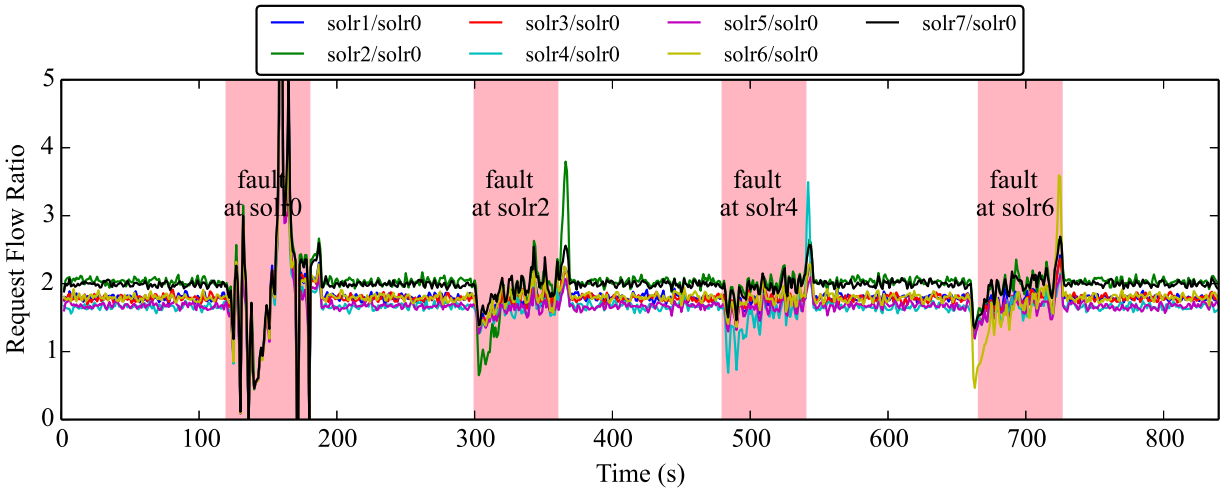
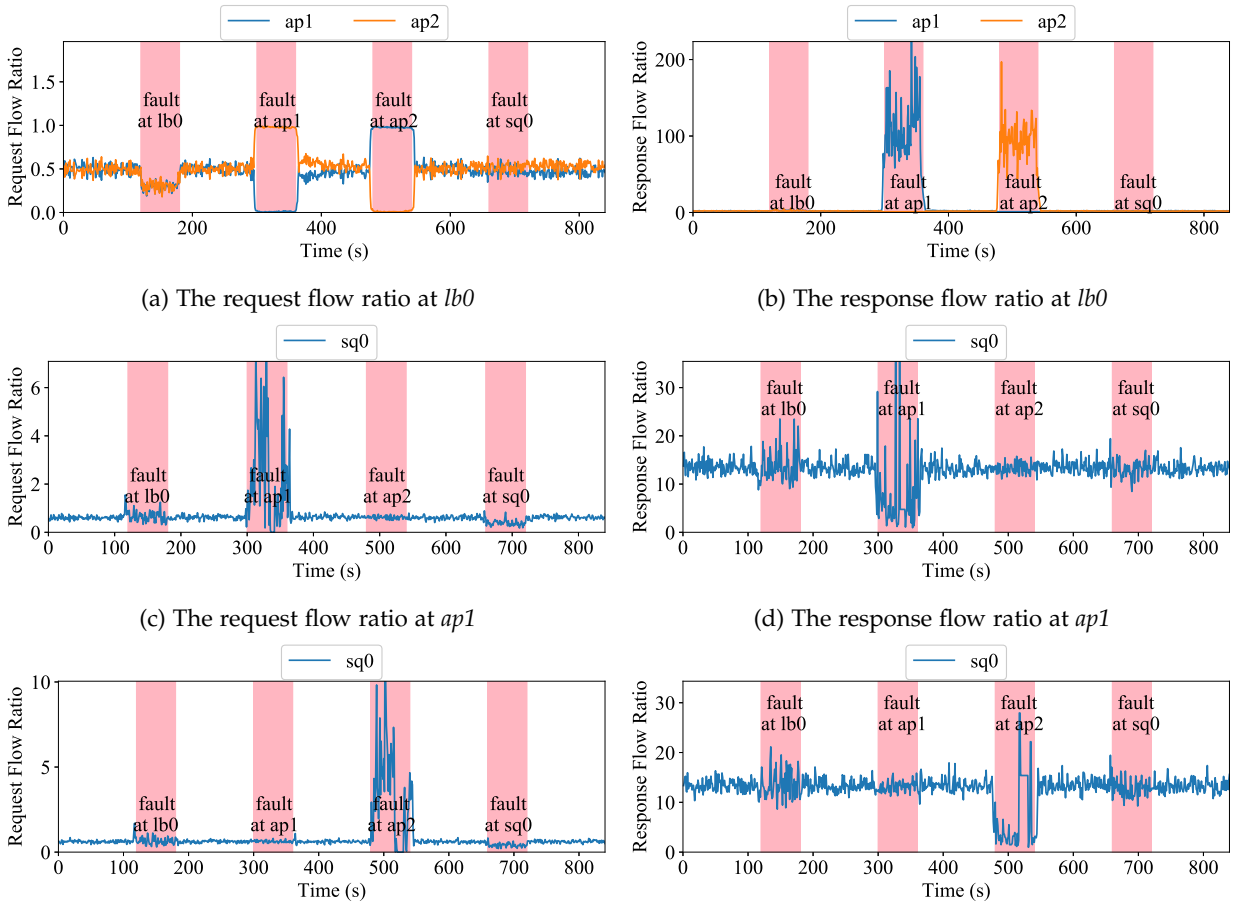


Figure 7.16: The request flow ratio for anomaly localization in CloudSuite web search application

CloudSuite Web Search Application

Figure 7.16 shows the request flow ratio analysis at the index component solr0 when faults are injected to *solr0* (120 − 180s), *solr2*(300 − 360s), *solr4*(480 − 540s), and *solr6*(660 − 720s).

When the fault is injected to *solr0*, the service time of *solr0* increases. But it has similar impact on the



(a) The request flow ratio at *lb0*

(b) The response flow ratio at *lb0*

(c) The request flow ratio at *ap1*

(d) The response flow ratio at *ap1*

(e) The request flow ratio at *ap1*

(f) The response flow ratio at *ap1*

Figure 7.17: The request/response flow ratio for anomaly localization in Olio web application

interaction between the local component and all its dependent components. The request flow ratios of all dependent components change almost similarly and concurrently. When the component *solr2* is faulty, its response time increases. It also causes requests to wait longer at the local component *solr0*, and the request flow ratios of other normal dependent components also change. The interaction between component *solr2* and the local component *solr0* is affected the most and the request flow ratio of *solr2* decreases much more than that of other normal dependent components. The same localization analysis applies when faults are injected to *solr4* and *solr6*.

Olio Web Application

Figure 7.20 shows the request flow ratio analysis and response flow ratio analysis of different components when faults are injected to *lb0* (120 − 180s), *ap1* (300 − 360s), *ap2* (480 − 540s), and *sq0* (660 − 720s).

When the fault is injected to component *lb0*, the service time of *lb0* increases. The requests wait

at *lb0* for longer time before they are distributed to backend application servers. The fault at *lb0* has similar impact on the two application servers when they interact with *lb0*. The request flow ratios of two application servers decrease simultaneously with similar scale. The fault does not impact the interaction between application servers and the database. The request flow ratios and the response flow ratios of the database at two applications servers are not impacted.

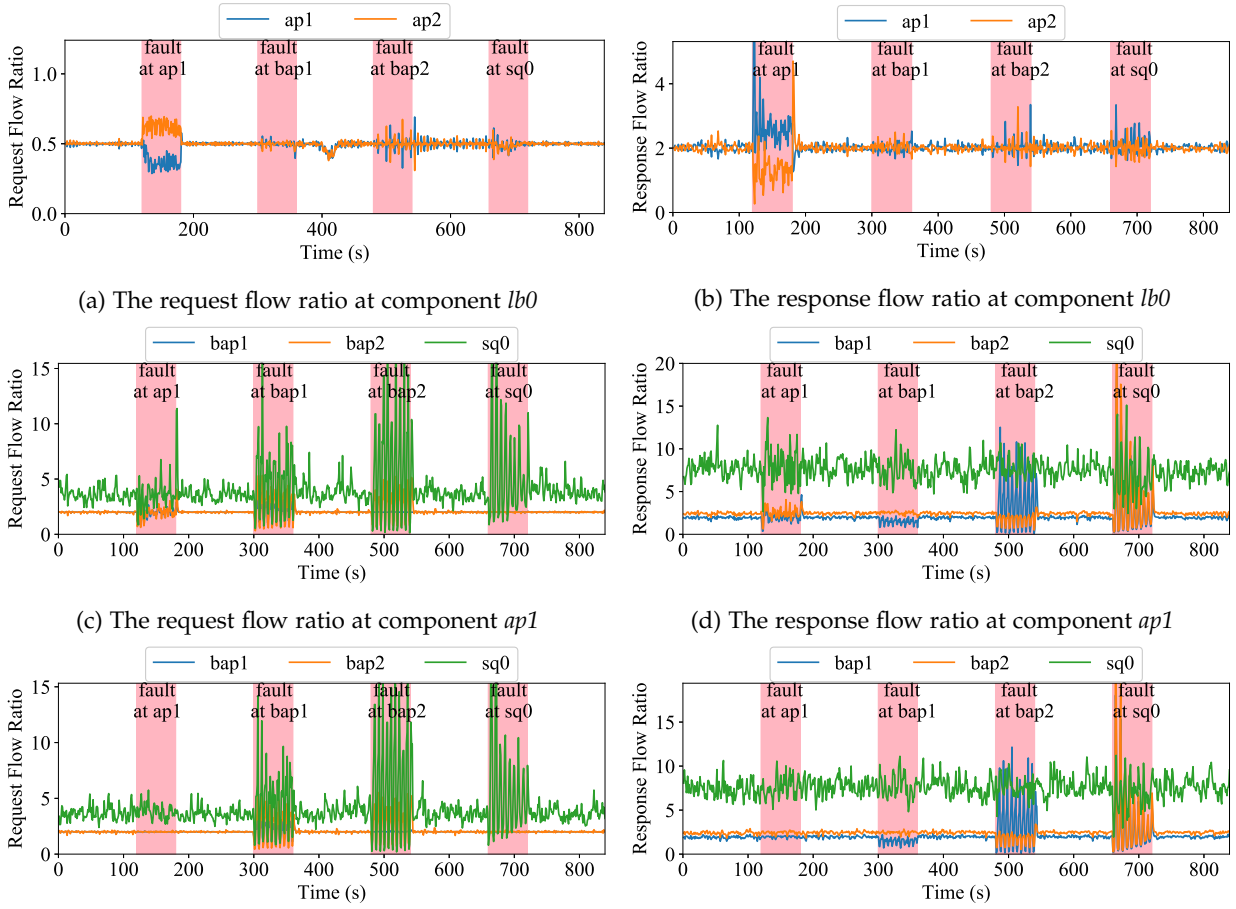
When the fault is injected to component *ap1*, the requests that should be distributed to *ap1* wait longer at the load balancer because of TCP propagation. The fault does not impact the interaction of *ap2* with *lb0*, and requests are scheduled to *ap2* normally. Most requests that arrive at *lb0* are scheduled to *ap2* during the fault. At the load balancer, the request flow ratio of *ap1* decreases, but the request flow ratio of *ap2* increases. The *ap1* return responses at a much slower speed than *ap2*, and most responses returned from *lb0ap1* are from *ap2*. The response flow ratio of *ap1* increases, and the response flow ratio of *ap2* decreases. The fault causes queries from *ap1* to *sq0* to take longer time due to TCP propagation. At component *ap1*, the request flow ratio of *sq0* decreases. When *sq0* sends responses back to *ap1*, it also takes longer for *ap1* to return responses to *lb0*. The response flow ratio of *sq0* increases. The agent of *ap2* does not detect an anomaly, and the fault does not impact the interaction between *ap2* and *sq0*. The request flow ratio and the response flow ratio of *sq0* at *ap2* is not impacted by the fault. A similar analysis applies if a fault is injected to component *ap2*.

When the fault is injected to *sq0*, the interaction between *lb0* and *ap1*, *ap2* is not impacted. At component *lb0*, the request flow ratios and the response flow ratios of two application servers are not impacted. The DMFDL agent of *lb0* determines that component *lb0*, *ap1*, *ap2* are normal. The requests that should be sent to *sq0* wait longer at the two application servers. The request flow ratios of *sq0* at *ap1* and *ap2* decrease compared to those in normal operation. The response flow ratios of *sq0* are not impacted in this case.

MediaWiki Application

In MediaWiki application, there are three components (*bap1*, *bap2*, *sq0*) forming a composite dependency primitive with components (*ap1* and *ap2*). Figure 7.18 shows the request flow ratio and the response flow ratio at different components when faults are injected to component *ap1* (120-180s), *bap1* (300-360s), *bap2* (480-540s), and *sq0* (660-720s) in MediaWiki application.

When component *ap1* is faulty, DMFDL agents for component *lb0*, *ap1*, and *sq0* detect anomalies. The agent of *lb0* further checks the request flow ratio and response flow ratio. The request flow ratio of both dependent components does not show obvious change. But the response flow ratio of *ap1* increases. DMFDL agent of *lb0* pinpoints its dependent component *ap1* as faulty. DMFDL agent of *ap1* checks the request flow ratio and response flow ratio of all dependent components. The request flow ratio and



(a) The request flow ratio at component *lb0*

(b) The response flow ratio at component *lb0*

(c) The request flow ratio at component *ap1*

(d) The response flow ratio at component *ap1*

(e) The request flow ratio at component *ap2*

(f) The response flow ratio at component *ap2*

Figure 7.18: The request/response flow ratio for anomaly localization in MediaWiki application

response flow ratio of all dependent components decrease simultaneously. DMFDL pinpoints the local component *ap1* as faulty. DMFDL agent *ap2* does not detect the anomaly, and there is no change in the request flow ratio and the response flow ratio for all its dependent components. DMFDL agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

When component *bap1* is faulty, DMFDL agents of component *lb0*, *ap1*, *ap2*, and *bap1* detect anomalies. The agent of *lb0* further checks the request flow ratio and response flow ratio. The request flow ratio of both dependent components and the subsystem almost does not show obvious change. DMFDL agent of *lb0* cannot pinpoint any component as faulty. The agent of *ap1* checks the request flow ratio and response flow ratio of all dependent components. The request flow ratio of all dependent components changes simultaneously, but the request flow ratio of *bap2* and *sq0* changes much more than that of *bap1*. The response flow ratio of all dependent components almost does not change. DMFDL agent of *ap1* pinpoints the dependent component *bap1* as faulty. Similarly, DMFDL agent of *ap2* also pinpoints *bap1* as faulty.

DMFDL agent of *bap1* does not perform localization as *bap1* does not have any dependent component.

When component *bap2* is faulty, DMFDL agents of component *lb0*, *ap1*, *ap2*, and *bap1* detect anomalies. The agent of *lb0* further checks the request flow ratio and response flow ratio. The request flow ratio of both dependent components almost does not change. The request flow ratio and the response flow ratio of its subsystem almost does not decrease. DMFDL agent of *lb0* cannot pinpoint any dependent component. The agent of *ap1* checks the request flow ratio and response flow ratio of all dependent components. The request flow ratio of dependent component *bap1* does not decrease. The request flow ratio of *bap2* and *sq0* decreases, but the request flow ratio of *sq0* decreases much more than that of *bap2*. The response flow ratio of *sq0* does not decrease. The response flow ratio of *bap1* decreases much more than that of *bap2*. It shows that the delay between when *bap1* returns responses to *ap1* and when *ap1* returns responses back to its upstream components increases a lot. DMFDL agent of *ap1* pinpoints *bap2* as faulty. Similarly, DMFDL agent of *ap2* also pinpoints *bap2* as faulty. DMFDL agent of *bap2* does not perform localization as *bap2* does not have any dependent component.

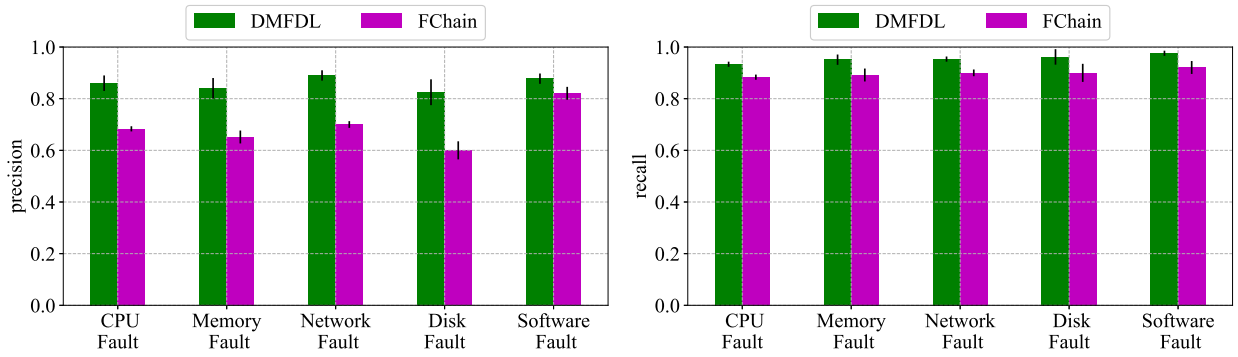
When component *sq0* is faulty, DMFDL agents of component *lb0*, *ap1*, *ap2*, and *sq0* detect anomalies. The agent of *lb0* further checks the request flow ratio and response flow ratio. The request flow ratio of both dependent components almost does not change. DMFDL agent of *lb0* cannot pinpoint any component. The agent of *ap1* checks the request flow ratio and response flow ratio of all dependent components. The request flow ratio of dependent components *bap1* and *bap2* almost does not change. The request flow ratio of *sq0* decreases a lot. It shows that *sq0* is probably faulty. The response flow ratio of *sq0* decreases a little, but the response flow ratio of *bap1* and *bap2* decreases much more than that of *sq0*. It shows that the delay between when *sq0* returns responses to *ap1* and when *ap1* returns responses to its upstream components increases a lot. DMFDL agent of *ap1* pinpoints *sq0* as faulty. Similarly, DMFDL agent of *ap2* also pinpoints *sq0* as faulty. DMFDL agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

7.5.6 Anomaly Localization Performance

To get the performance of DMFDL in anomaly location, we run extensive experiments under varying workload conditions with multiple different faults randomly injected into these applications. We compare the performance of DMFDL with FChain in anomaly localization.

We show the detection precision and recall of both DMFDL and FChain in localizing faults in different applications: CloudSuite web search application in Figure 7.19, Olio web application in Figure 7.20, MediaWiki application in Figure 7.21.

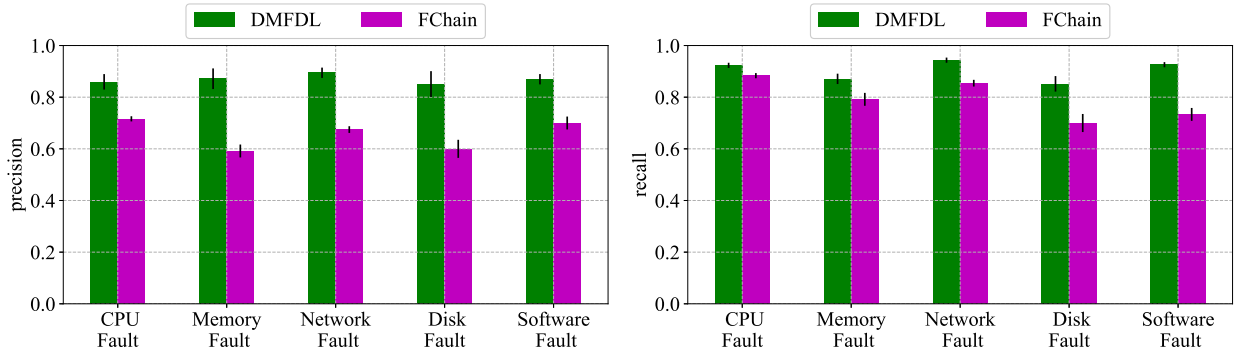
DMFDL combines the dependency model and the flow ratio model in anomaly localization. Rather than relying simply on the chronological order of changing points of different components. We show that the causal execution order is important and the flow ratio scale change is more accurate in anomaly localization. DMFDL has much lower false negative than FChain for all different faults. When DMFDL



(a) The localization precision

(b) The localization recall

Figure 7.19: The localization performance of DMFDL and other methods in CloudSuite web search application



(a) The localization precision

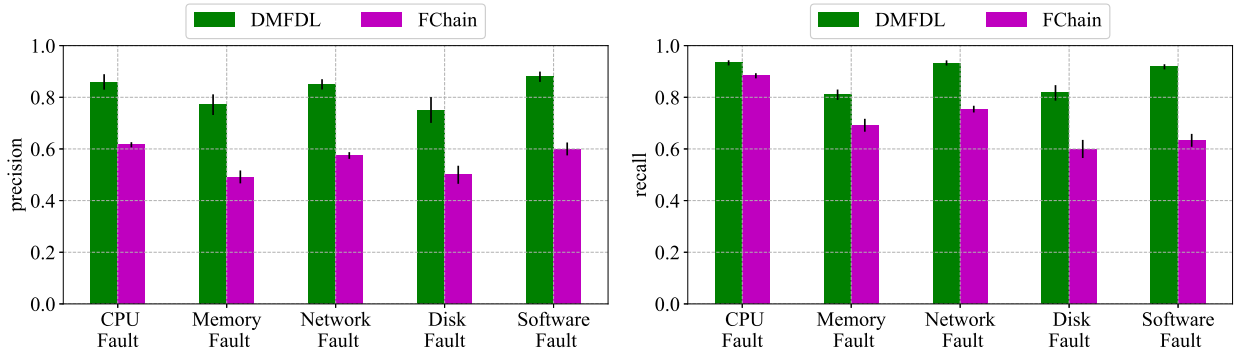
(b) The localization recall

Figure 7.20: The localization performance of DMFDL and other methods in Olio web application

pinpoints the wrong dependent component, it also represents a miss in anomaly localization. The main reason is that the fault propagates very quickly due to high throughput communication between components that interact with each other.

7.5.7 Overhead Analysis

DMFDL has lower overhead than DMADL as it just counts the number of data packets. It does not need to store any TCP flows in memory. So it has even lower overhead than DMADL. DMFDL has less than 0.5% overhead and negligible memory overhead under varying workload intensities.



(a) The localization precision

(b) The localization recall

Figure 7.21: The localization performance of DMFDL and other methods in MediaWiki application

7.6 Summary

The adaptive flow ratio model characterizes the relationship that the output response flow should changes with the input request flow in scale. Whenever, DMFDL observes the flow ratio decreases abnormally out of acceptable range, it triggers anomaly alert. However, it could produce false positive for components in applications where the relationship between output response flow over input request flow varies a lot. It is a future challenge to address this problem to deal with more dynamic applications.

Chapter 8

DMCDL: Dependency Model-based Flow Correlation Analysis for Anomaly Detection and Localization

In this chapter, we present DMCDL, a dependency model-based flow correlation analysis for detecting and localizing anomalies in multi-tier Cloud applications. DMCDL has agents distributed on physical hosts. It does not require domain knowledge from either component operating systems or underlying applications. DMCDL monitors network traffic at virtual network interface and conducts performance analysis in real time. Multi-tier Cloud applications run together on multiple components, and these components have complex interactions and behaviors depending on specific application service. DMCDL makes use of the relationship that the response data packets always follow the request data packets within a time frame corresponding to the maximum acceptable service latency as defined in the Service Level Agreement (SLA) given any component service in normal operation. The number of observed response data packets should closely follow the number of request data packets within SLA in normal operation. This is true for any component service with different configurations under dynamic workloads. It is violated in abnormal operation as the response time exceeds the SLA. DMCDL uses the Pearson correlation to characterize the relationship between the number of request data packets and the number of response data packets. VDEP [2] introduces three dependency primitives as a basis to characterize the complex interaction behavior among multiple components in distributed applications: Composite dependency, Concurrent dependency, and Distributed dependency. DMCDL combines the correlation analysis and the local dependency model for anomaly detection. DMCDL further pinpoints the faulty component by following the propagation path of anomalies. For each local component, DMCDL uses the interaction between the local component

and its dependent components to determine the source of anomaly locally.

8.1 DMCDL overview

DMCDL consists of 5 major functions: traffic flow monitoring, dependency model extraction, flow correlation analysis, anomaly detection, and anomaly localization. The operating flow of these functions in DMCDL is shown in Figure 8.1. Flow monitoring function captures the network traffic flow outside components and analyzes only data packets. The dependency model extraction function uses the number of request and response data packets to identify the execution sequence of dependent components. The flow correlation function models the service performance with the Pearson correlation between request and response flow using an appropriate sampling interval in normal operation. The anomaly detection function compares the flow correlation with normally-profiled flow correlation and combines it with the dependency model to detect performance anomalies in Cloud applications. The anomaly localization function further analyzes the request/response flow correlation and combines it with the dependency model to pinpoint the faulty component in a decentralized fashion.

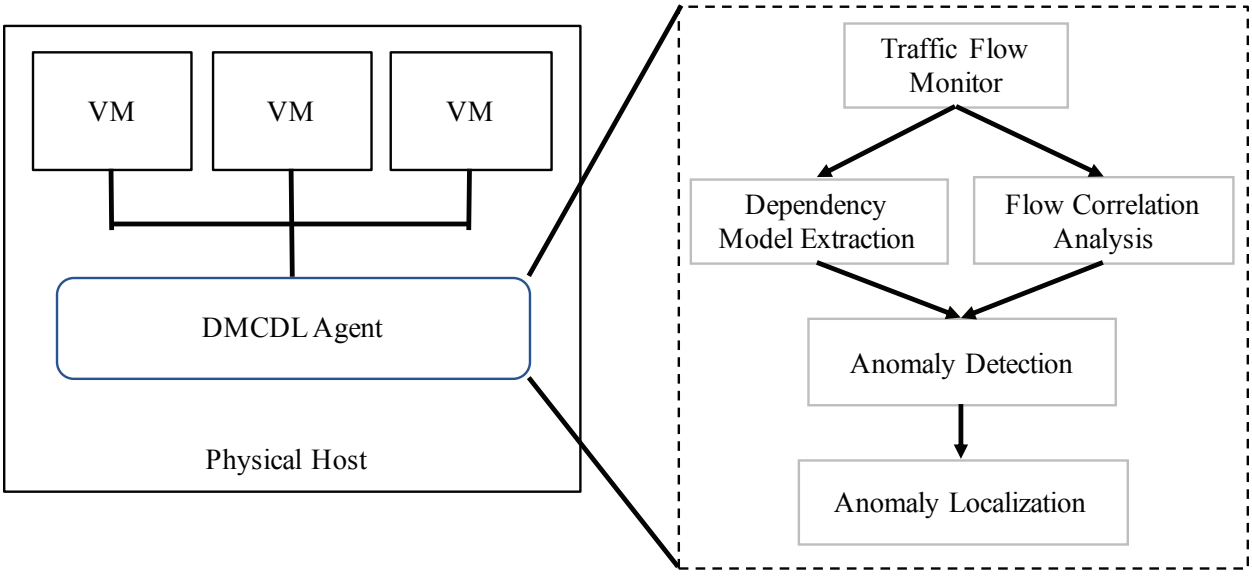


Figure 8.1: The operation diagram of DMCDL for anomaly detection and localization

8.2 Traffic Flow Monitor

For each component service, DMCDL agents monitor the traffic flows in the corresponding virtual network interface in each sampling interval $(t - \Delta T_c, t)$ (ΔT_c is the sampling interval). DMCDL uses the same network traffic flow monitor as DMFDL in Figure 7.2 for each component. At a local component

A, we denote the input request flow as $X_A(t)$ and the output response flow as $Y_A(t)$. For its dependent component B, the output request flow is denoted as $X_{AB}(t)$ and the input response flow is denoted as $Y_{AB}(t)$. DMCDL uses the Pearson correlation analysis between the input request flow and the output response flow for performance anomaly detection at the local component.

8.3 Anomaly Detection

DMCDL models the flow conservation relationship using traffic flow correlation analysis. DMCDL agents uses the flow correlation for anomaly detection at each component of Cloud applications.

8.3.1 Traffic Flow Correlation

DMCDL agents first find a proper sampling interval ΔT_c. The sampling interval should maximize the probability that request data packets and their response data packets are observed within the same interval when the service runs in normal operation. However, it should also minimize the above probability in abnormal operation.

Given a request arrived at time $t_q \in (t - \Delta T_c, t)$ and its response time is τ:

- If $\tau \geq \Delta T_c$, the request data packets and its response data packets would never be observed and counted within the same monitoring interval. The indicator function of the request data packets and corresponding response data packets in the same monitoring interval is

$$p = 0$$

- If $\tau < \Delta T_c$, the indicator function of whether the request data packets and its response data packet are observed and counted within the same monitoring interval is p.

$$p = \begin{cases} 1, & \text{if } t_q \in (t - \Delta T_c, t - \tau). \\ 0, & \text{if } t_q \in (t - \tau, t). \end{cases}$$

First, we derive the probability that request data packets and its response data packets are observed in the same monitoring interval in normal operation. We assume that the arrival time of the first data packet of requests t_q in the interval $(t - \Delta T_c, t)$ follows a certain distribution $f_a(x)$.

$$f_a(x) = P(t_q = x), x \in (t - \Delta T_c, t)$$

We assume the response time of requests in the interval $(t - \Delta T_c, t)$ follows distribution $g_n(\tau_t)$, and its cumulative distribution function $G_n(\tau_t)$. The probability that request data packets and its response data

Algorithm 6: DMCDL Sampling Interval Selection Algorithm

input : For a component service A, the time series of input request flow $X_A(t)$ and output response flow $Y_A(t)$ during normal operation. A list of optional sampling intervals $\Delta T_i (i = 1, 2, \cdots)$

output: The proper sampling interval ΔT_c.

```
for the sampling interval δT_i do
    Compute the flow correlation using the sampling interval ΔT_i during normal operation: ρ(t);
    Compute the mean of the flow correlation μ_i and the standard deviation σ_i;
    Compute the relative deviation: K_i = σ_i / μ_i;
end
for i = 2 to n do
    if ‖K_i − K_{i−1} ≤ K_th and K_{i−1} ≤ K_σ then
        break;
    end
end
The proper sampling interval is: ΔT_c = ΔT_i;
```

packets are observed in the same monitoring interval in normal operation

$$p_n(t) = \int_0^{\Delta T_c} P(t_q = t - x) * P(\tau_t \leq x)dx = \int_0^{\Delta T_c} f_a(t - x) * G_n(x)dx$$

When there is an anomaly, we assume the response time of requests in the interval $(t - \Delta T_c, t)$ follows a certain distribution $g_a(\tau_t)$, and its cumulative distribution function is $G_a(\tau_t)$. The probability that request data packets and its response data packets are observed in the same monitoring interval during abnormal operation is

$$p_a(t) = \int_0^{\Delta T_c} P(t_q = t - x) * P(\tau_t \leq x)dx = \int_0^{\Delta T_c} f_a(t - x) * G_a(x)dx$$

The proper sampling interval in ΔT_c must maximize the difference between the probability function $p_n(t)$ in normal operation and the probability function $p_a(t)$ in abnormal operation.

$$\Delta T_c = \arg\max \ [p_n(t) - p_a(t)]$$

It is not trivial to specify the service level target (SLA) for each component service or solving the above optimization problem without knowing those distribution functions. DMCDL uses Algorithm 6 to determine the proper sampling interval for the flow correlation. To obtain the proper sampling interval, DMCDL selects the proper sampling interval that achieves stable flow correlation in normal operation. DMCDL uses the selected sampling interval ΔT_c to model the relationship between the input request flow and the output response flow.

In normal operation, the input request flow $X_A(t)$ and the output response flow $Y_A(t)$ has high correlation coefficients. In abnormal operation, the system slows down and the output response flow no longer follows the input request flow in the same sampling interval. The flow correlation decreases in presence of

anomalies. DMCDL makes use of the flow correlation $\rho(X_A, Y_A, t)$ between the input request flow $X_A(t)$ and the output response flow $Y_A(t)$ within a window of W samples for anomaly detection.

$$\rho(X_A, Y_A, t) = \frac{\sum_{t=0}^{W-1} \left(X_A(t) - \overline{X_A(t)}\right)\left(Y_A(t) - \overline{Y_A(t)}\right)}{\sqrt{\sum_{t=0}^{W-1} \left(X_A(t) - \overline{X_A(t)}\right)^2}\sqrt{\sum_{t=0}^{W-1} \left(Y_A(t) - \overline{Y_A(t)}\right)^2}}$$

8.3.2 Detection Algorithm

To deal with real time detection of anomalies, we apply a sliding window strategy in our detection algorithm. The sliding window strategy is frequently used in data streaming and it assumes that the recent data is closely related to historical data in normal operation. It discards old samples from the window and adds new samples into the window. We update parameters for anomaly detection dynamically. DMCDL adapts to dynamic system for online anomaly detection.

We first train the system in normal operation for a period and learns each component's normal correlation profile. The normal correlation profile is characterized with its mean and standard deviation. In the detection phase, it continues to detect whether there are anomalies with data points in the upcoming window. Within each window, there are two parts of data samples. The first part consists of data points from historical normal operation called the reference set. The second part consists of data points from latest period waiting for evaluation called the test set. We compute the correlation of data points in each window. To quantify whether the correlation is normal or not, we maintain a normal profile for the correlation. If the obtained correlation is within the latest normal profile, the evaluated data points in the test set are normal. We should continue to update normal profile and include the data points from the test set to the latest reference set. If the obtained correlation exceeds the latest normal profile, then the data points in the test set is suspicious. In this case, we continue to evaluate the new data points using historical normal profile after sliding the window. Through updating the current window profile regularly, the sliding window strategy enables the online fault detection to adapt to the time-varying behavior of the system. Now we are going to describe our algorithm in detail. We denote the correlation window size as W, and the sliding window step size as SW.

Suppose the ith window contains the data samples from t_{i-W} to t_i, the reference set contains the latest $(W - SW)$ data samples when the system runs in normal operation S_{ref} and the test set contains SW latest data points for evaluation S_{test}. We also have the local correlation profile μ_i and σ_i. In the current window, we compute the correlation $\rho(X, Y, t_i)$.

If the current correlation stays within the normal profile (i. e., the correlation is within λ times of the standard deviation σ_i from the mean correlation μ_i): $\rho(X, Y, t_i) \in (\mu_i - \lambda\sigma_i, \mu_i + \lambda\sigma_i)$, the data in the test

set S_{test} is normal. Otherwise, we consider the recent data in S_{test} is suspicious.

DMCDL updates the normal profile only when the data in the test set S_{test} is normal. It uses exponentially weighted moving average (EWMA) to update the correlation profile.

$$\mu_{i+1} = \alpha\mu_i + (1-\alpha)\rho(X, Y, t_i)$$

$$\sigma_{i+1} = \alpha\mu_i + (1-\alpha)\max(|\rho(X, Y, t_i) - \mu_i|, \sigma_i)$$

Here, α is the weighting parameter on the historical profile. However, as we update the standard deviation σ, we use the maximum value of the difference and the historical deviation. After evaluation, we slide the window to include data points from t_i to t_{i+SW} into the test set for new evaluation.

DMCDL agent uses a look-back window to confirm whether the component really has performance anomalies or not. DMCDL triggers anomaly alert if the number of suspicious detection in the look-back window exceeds a tolerable threshold. Otherwise, it has to wait for more detection to confirm anomalies.

We give the detailed algorithm for anomaly detection based on correlation analysis between the number of incoming request data packets and outgoing response data packets in Algorithm 7

Algorithm 7: Anomaly Detection

input : Given a local component A and its m dependent components $S = \{A_{d1}, A_{d2}, \cdots, A_{dm}\}$
output: Detection result for the local component A

It selects a sampling interval ΔT_A to monitor the number of incoming request data packets and outgoing response data packets. The sampling interval ΔT_A should approximate the maximum acceptable response time at the local component.;
It monitors the number of request data packets arrived at component A: $X_A(t)$, and the number of response data packets departed from component A: $Y_A(t)$.;
It uses a window of latest W_A samples for detecting anomalies;
if $\sum_{t=1}^{W_A} X_A(t) > 0$ *and* $\sum_{t=1}^{W_A} Y_A(t) == 0$ *and* $\sum_{t=1}^{W_A-1} X_A(t) > 0$ **then**
    $state(t) = -1$;
    Anomaly Alert: A does not return responses for requests;
else
    if $\sum_{t=1}^{W_A} R_A(t) >= \mu_A(t) + k_A\sigma_A(t)$ **then**
        $suspicious(t) = -1$;
    else
        $suspicious(t) = 1$;
    end
    if $\sum_{t=1}^{W_A} suspicious(t) <= TH_A$ **then**
        $state(t) = -1$;
        Anomaly Alert: A's response time is abnormal;
    else
        $state(t) = 1$;
    end
end

8.4 Anomaly Localization

Previous methods build the dependency graph for the whole application in order to locate the faulty component. In our case, we do not need a centralized method to build the dependency graph for the whole application. The DMCDL agents distributed on physical hosts uses the local dependency model and flow correlation analysis for anomaly localization.

8.4.1 Local Interaction Behavior Characterization

Due to the fault propagation, the flow correlation at a local component decreases due to three possible causes: (1) the upstream components of the local component; (2) the local component itself; or (3) the local component's subsystem. As the flow correlation at the local component is affected by its upstream components or its dependent components, it is usually difficult to determine the state of the local component. Each DMCDL agent does localization only for the dependent components of the local component.

If the local component has dependent components and the local anomaly is caused by the upstream components of the local component, the interaction between the local component and its dependent components is still normal. If the local anomaly is caused by the local component or its subsystem components, the interaction between the local component and its dependent components are affected due to anomaly propagation. DMCDL uses the local interaction between the local component and its dependent components to determine whether the state of dependent components.

DMCDL characterizes the local interaction behavior between the local component and each of its dependent components using the request flow correlation and the response flow correlation. For a dependent component B in Figure **??**, the request flow correlation $\rho(X_A, X_{AB}, t)$ is defined as the pairwise correlation between the output request flow $X_{AB}(t)$ and the local input request flow $X_A(t)$. The response flow correlation $\rho(Y_A, Y_{AB}, t)$ is defined as the pairwise correlation between the input response flow $Y_{AB}(t)$ and the local output response flow $Y_A(t)$.

$$\rho(X_A, X_{AB}, t) = \frac{\sum_{t=0}^{W-1}\left(X_A(t) - \overline{X_A(t)}\right)\left(X_{AB}(t) - \overline{X_{AB}(t)}\right)}{\sqrt{\sum_{t=0}^{W-1}\left(X_A(t) - \overline{X_A(t)}\right)^2}\sqrt{\sum_{t=0}^{W-1}\left(X_{AB}(t) - \overline{X_{AB}(t)}\right)^2}}$$

$$\rho(Y_A, Y_{AB}, t) = \frac{\sum_{t=0}^{W-1}\left(Y_A(t) - \overline{Y_A(t)}\right)\left(Y_{AB}(t) - \overline{Y_{AB}(t)}\right)}{\sqrt{\sum_{t=0}^{W-1}\left(Y_A(t) - \overline{Y_A(t)}\right)^2}\sqrt{\sum_{t=0}^{W-1}\left(Y_{AB}(t) - \overline{Y_{AB}(t)}\right)^2}}$$

In order to find out the faulty component locally, DMCDL extracts the dependent component which is the first dependent component to execute among all dependent components. DMCDL needs to extract the first dependent component because the delay between the time when a request arrives at the local

component and the time when a local component further calls the first dependent component would increase if there is an anomaly at the local component or its subsystem components. If the request flow correlation decreases and drops out of normal correlation profile, DMCDL agent infers that local component is faulty. Otherwise, the local anomaly is caused by its upstream components.

8.4.2 No Dependent Component

The local component has no dependent component. The local flow correlation decreases due to two different cases: a fault at the local component or at its upstream components. DMCDL agent does not have monitoring data for its upstream components, it is difficult to decide the real cause of the local anomaly. If the DMCDL agents of its upstream components do not trigger anomaly alert, the local component is faulty. In an extreme case where the DMCDL agent observes only input request flow but without output response flow, the flow correlation is constantly 0 and DMCDL reports that the local component is faulty.

8.4.3 Single Dependent Component

The local component has a single dependent component. In order to find out the root cause for the anomaly. We discuss several different cases:

1. When its upstream component(s) is faulty, the local service time and its subsystem response time are not impacted. The request flow correlation and response flow correlation are still similar as in normal operation.
2. When the local component is faulty, requests arrived at the local component take longer to reach its dependent component. The correlation between the local input request flow and the output request flow to its dependent component decreases. It also takes longer time for the local component to send responses back to the upstream components after receiving responses from the dependent component. The correlation between the input response flow from the dependent component and the output response flow from the local component decreases.
3. When the dependent component is faulty, requests that should be sent to the dependent component are write-blocked due to the TCP propagation. The local waiting time increases. The correlation between the input request flow and the output request flow to the dependent component decreases. The responses arrived at the local component are sent back to the upstream components normally. The correlation between the input response flow from the dependent component and the output response flow from the local component is still normal.

DMCDL uses the correlation between the input request flow and the output request flow, and the correlation between the output responses flow and input response flow to perform the localization analysis at the local component.

8.4.4 Distributed Dependency Primitive

The local component has 2 or more dependent components, and they form a distributed dependency primitive. In distributed dependency primitive, the input request flow equals the sum of output request flows to dependent components, and the output response flow equals the sum of input response flows from dependent components.

- When the upstream component(s) is faulty, the local component still interacts with dependent components normally. The request flow correlations and response flow correlations of all dependent components are still normal.
- When the local component is faulty, requests arrived at the local component take longer to reach its dependent components. The request flow correlations of all dependent components decrease with similar scale. It also takes longer time for the local component to send responses back to upstream components. The response flow correlations of all dependent components decrease with similar scale as well.
- When a dependent component is faulty, the local component is write-blocked when it tries to send requests to the faulty dependent component due to TCP propagation. Most requests arriving at the local component are processed by other normal dependent components. The request flow correlations of non-faulty dependent components are still normal or even higher. Most responses returned from the local component are returned by non-faulty dependent components. The response flow correlations of non-faulty dependent components are still normal or even higher. But the response flow correlation of the faulty dependent component decreases.

However, this may not always hold. In some cases, the scheduling policy is important in request flow correlation and response flow correlation of dependent components. The number of requests from the local component to its dependent components may or may not be proportional to the number of requests arrived at the local component.

8.4.5 Composite Dependency Primitive

For a local component with multiple composite dependent components, DMCDL analyzes different causes and respective behavior for local anomaly localization.

- When an upstream component(s) is faulty, the local component still sends requests and returns responses normally. The flow correlation of all dependent components is still normal.
- When the local component is faulty, requests take longer to reach its dependent components. The request flow correlations of all dependent components decrease with similar scale. It also takes longer time for the local component to send responses back to upstream components. The response flow correlations of all dependent components decrease with similar scale.
- When a dependent component is faulty, the local component is write-blocked when it tries to send requests to the faulty dependent component due to TCP propagation. The local waiting time increases, and the request flow correlation of the faulty dependent component decreases. Requests arrived at the local component take longer to reach those dependent components that run after the faulty dependent component. The request flow correlation of those dependent components that run after the faulty component decreases. The request flow correlations of those dependent components that run before the faulty component almost do not change. The responses returned by dependent components that run after the faulty dependent component are sent back to upstream components normally. The response flow correlations of dependent components that run after the faulty dependent component do not change. The responses returned by dependent components that run before the faulty dependent component take longer to be returned by the local component. The response flow correlations of dependent components that run before the faulty dependent component decreases compared to normal operation.

8.4.6 Concurrent Dependency Primitive

For a local component with multiple concurrent dependent components, DMCDL analyzes different causes and respective behavior for local anomaly localization.

- When an upstream component(s) is faulty, the local component still sends requests and returns responses normally. The flow correlation of all dependent components is still normal.
- When the local component is faulty, requests arrived at the local component take longer to reach its dependent components. The request flow correlation of all dependent components decreases

with similar scale. It also takes longer time for the local component to send responses back to the upstream components. The response flow correlation of all dependent components decreases with similar scale.

- When a dependent component is faulty, the local component is write-blocked when it tries to send requests to the faulty dependent component due to TCP propagation. The local waiting time increases, and the request flow correlation of the faulty dependent component decreases. When a request arrived at the local component causes more than one request to its dependent components, the local component has to wait for responses from all dependent components before sending subsequent requests to its dependent components. The request flow correlation of all dependent components decreases, but the request flow correlation of the faulty dependent component decreases much more. When the local component receives responses from non-faulty dependent components, it still has to wait for responses from the faulty dependent component before returning responses back to its upstream components. The response flow correlation of non-faulty dependent components decreases much more than the response flow correlation of the faulty dependent component.

DMCDL has a central administrator which summarizes the reported localization results from each DMCDL agent and pinpoints the faulty component for Cloud applications.

8.5 Experimental Evaluation

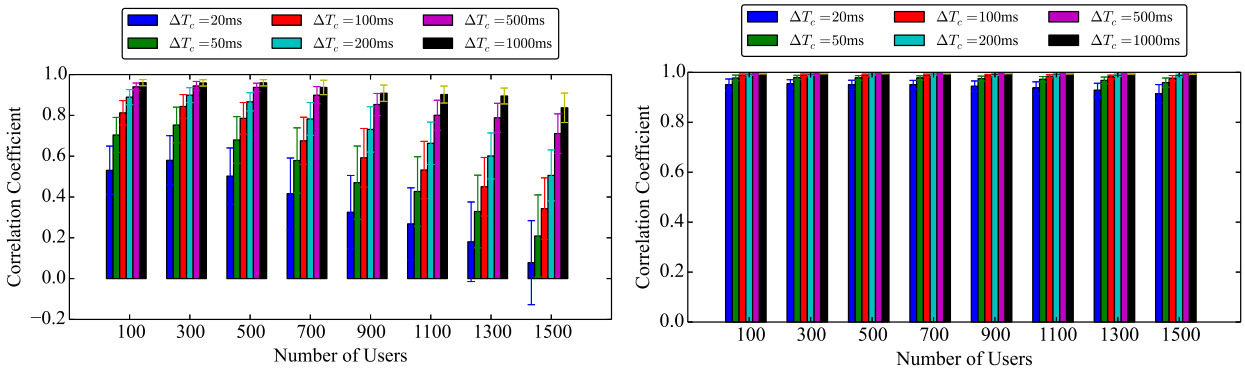
Our experimental evaluation covers three different aspects: (1) We show the stability of the flow correlation under varying workload intensity; (2) We evaluate DMCDL in detecting various kinds of performance anomalies using different fault models; (3) We evaluate DMCDL in locating anomalies injected into different components. (4) We evaluate the overhead of DMCDL.

8.5.1 Sampling Interval Selection

Figure 8.3 shows the mean flow correlation and its standard deviation with different sampling intervals under varying workload intensity. A larger sampling interval gives more stable flow correlation with smaller standard deviation under varying workload intensity. After the sampling interval increases to a critical limit, the impact on the mean flow correlation and the standard deviation is negligible. An excessively large sampling interval results in high flow correlation even when the system response time is larger than the acceptable time limit. We pick the sampling interval when it achieves a stable flow correlation and a standard deviation small enough.

CloudSuite Web Search Application

Figure 8.2 shows the flow correlation at the index component (*solr0* and one of the backend component *solr6*) with different sampling intervals under varying workload intensity in CloudSuite web search application. The flow correlation increases as the sampling interval increases for the index component *solr0*, and all its dependent components. At component *solr0*, the flow correlation is high enough as the sampling interval $\Delta T_c \geq 1000$ms. The flow correlation suffers high fluctuation as its response has larger fluctuation. For its dependent component *solr6*, the flow correlation is high enough as $\Delta T_c \geq 20$ms. It is because the response time of component *solr6* is usually smaller than 10ms. The flow correlation of other dependent components is also high and has very small fluctuation (*solr1*-*solr5*), which is similar to that of *solr6*.



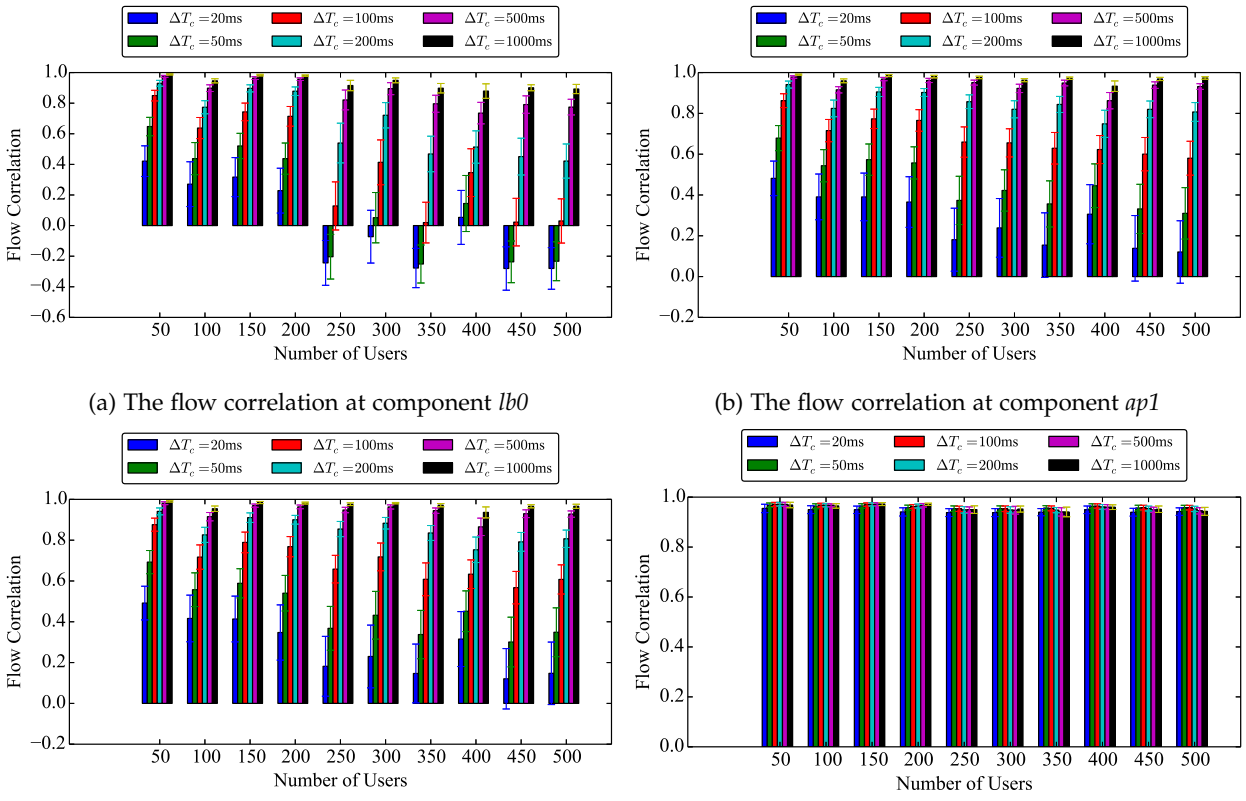
(a) The flow correlation at component *solr0*

(b) The flow correlation at component *solr6*

Figure 8.2: The stability of flow correlation with different sampling intervals under varying workload intensity in CloudSuite web search application

Olio Web Application

Figure 8.3 shows the flow correlation of different components in Olio web application with different sampling intervals under varying workload intensity. The flow correlation increases as the sampling interval increases for components *lb0*, *ap1*, and *ap2*. At the load balancer, the flow correlation is stable enough as the sampling interval $\Delta T_c \geq 500$ms. For two application servers (*ap1* and *ap2*), the flow correlation is larger than 0.8 as $\Delta T_c \geq 200$ms. The flow correlation of database component *sq0* is stable enough as the sampling interval changes. It is because the response time of component *sq0* is usually smaller than 10ms.



(a) The flow correlation at component *lb0*

(b) The flow correlation at component *ap1*

(c) The flow correlation at component *ap2*

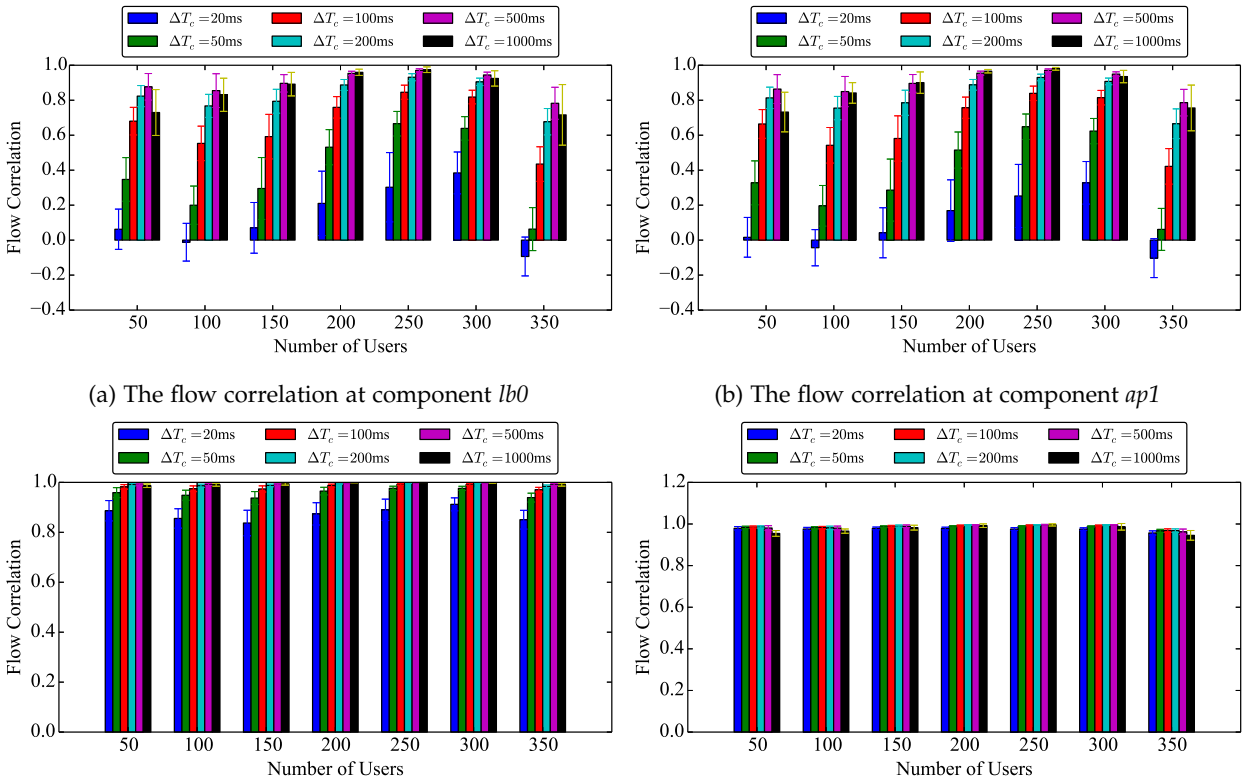
(d) The flow correlation at component *sq0*

Figure 8.3: The stability of flow correlation with different sampling intervals under varying workload intensity in Olio web application

MediaWiki Application

Figure 8.4 shows the flow correlation at multiple components (*lb0*, *ap1*, *bap1*, *sq0*) with different sampling intervals under varying workload intensity in MediaWiki application. The flow correlation increases as the sampling interval increases for components *lb0*, *ap1*, and *bap1*. But after $W_c > 500$ms, the flow correlation even decreases at different components as the interval increases. At the load balancer, the mean flow correlation is highest when the sampling interval $\Delta T_c = 500$ms. For two application servers (*ap1* and *ap2*), the flow correlation is larger than 0.6 as $\Delta T_c \geq 500$ms. For the two computational servers (*bap1* and *bap2*), the flow correlation is already high and stable enough as $W_c \geq 20$ms. The flow correlation of database component *sq0* is stable enough as the sampling interval changes. It is because the response time of component *sq0* is usually smaller than 10ms.

In summary, the sampling interval is important parameter to obtain high and stable flow correlation to model service performance in normal operation. A small sampling interval results in low flow correlation as well as large fluctuation as responses do not correspond to requests within the same interval. The



(a) The flow correlation at component *lb0*

(b) The flow correlation at component *ap1*

(c) The flow correlation at component *bap1*

(d) The flow correlation at component *sq0*

Figure 8.4: The stability of flow correlation with different sampling intervals under varying workload intensity in MediaWiki application

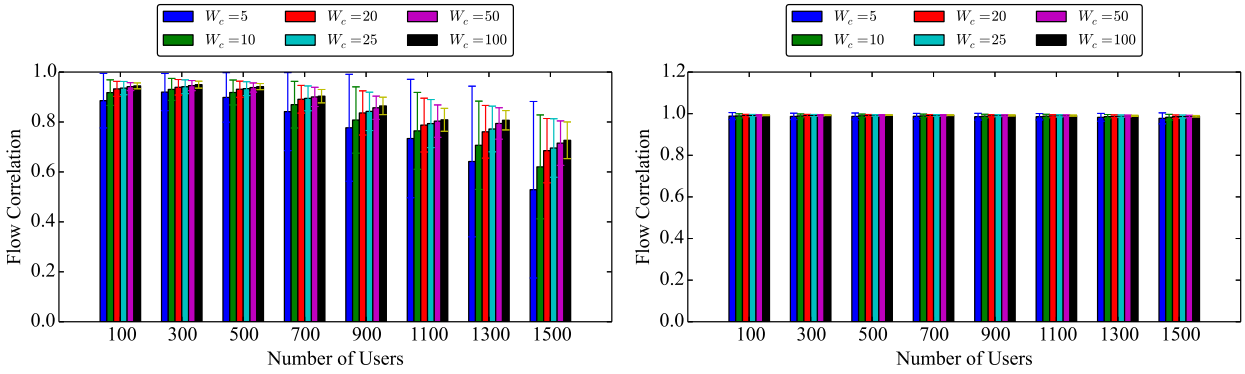
obtained flow correlation is not high and stable enough to be used for profiling service performance in normal operation. A large sampling interval is more likely to produce high and stable correlation as most responses correspond to requests within the same interval. It does not mean the larger is the sampling interval, the flow correlation is higher and more stable. DMCDL automatically selects a smallest sampling interval that achieves high and stable flow correlation for anomaly detection.

8.5.2 Correlation Window Selection

After selecting the proper sampling interval to collect the input and output traffic flow, it is also important to pick a proper window size (a proper number of samples) to compute the flow correlation.

CloudSuite Web Search Application

Figure 8.5 shows the mean flow correlation (with 95% confidence interval) with different window sizes under varying workload intensity in CloudSuite web search application. For the index component *solr0*, the flow correlation is low and not stable with small window size ($W_c \leq 20$). In comparison, the flow



(a) Flow correlation at component *solr0*

(b) Flow correlation at component *solr6*

Figure 8.5: The stability of flow correlation with different correlation windows under varying workload intensity in CloudSuite web search application

correlation of *solr6* is not that sensitive to the window size. It is because the response time of the index component usually has larger fluctuation than that of its back-end components.

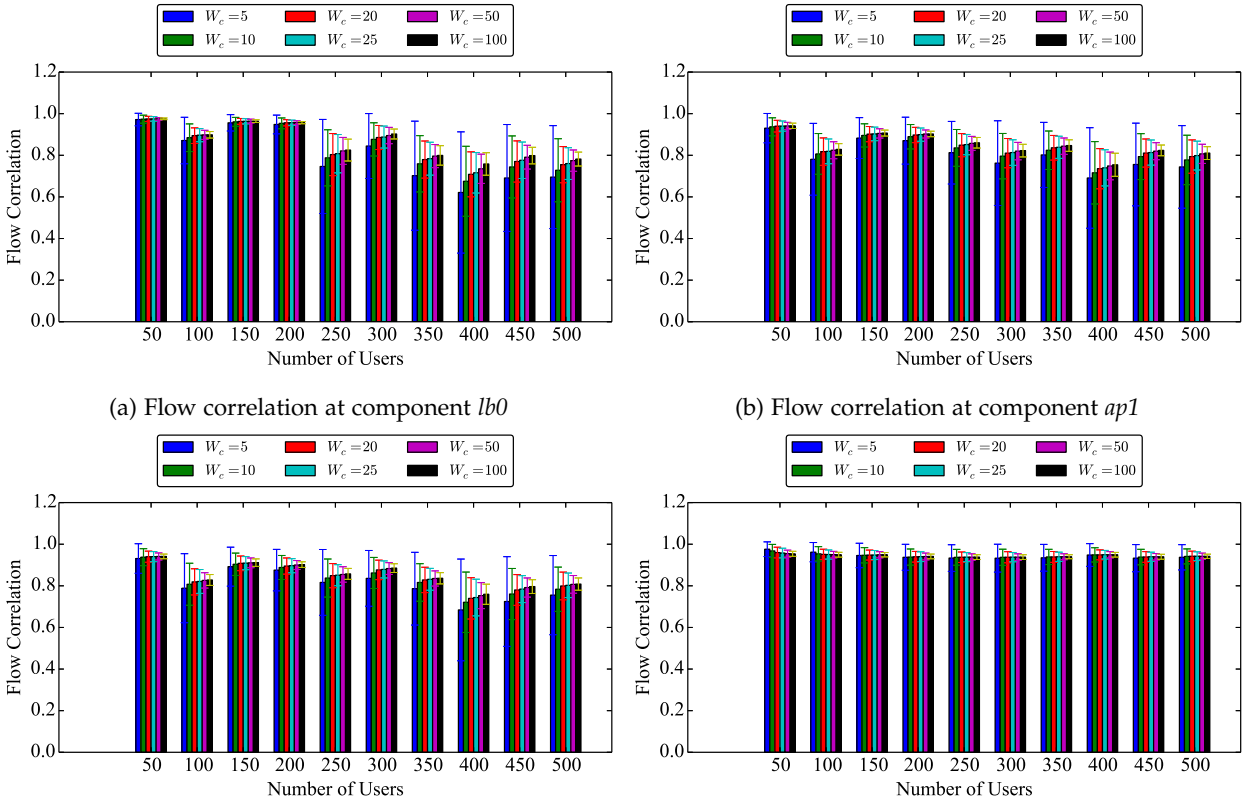
Olio Web Application

Figure 8.6 shows the mean flow correlation and the 95% confidence interval with different correlation window sizes under varying workload intensity of different components in application. A small correlation window requires less samples in order to compute the flow correlation. But the flow correlation is lower and has larger fluctuation. It poses a threat for detecting anomalies. For different components in Olio application, a window of more than 20 samples is good enough to obtain the flow correlation for modeling the service performance in normal operation.

MediaWiki Application

Figure 8.7 shows the flow correlation of different components with different correlation windows under varying workload intensity in MediaWiki application. The flow correlation is stable enough with different window sizes in MediaWiki application. The mean flow correlation is high across varying workload intensity and has small fluctuation.

In summary, the window size is important to obtain a reasonable flow correlation to model the normal performance, although the flow correlation is not that sensitive to the window size in most cases. A small window sizes has the advantage of less samples and thus less computational cost. But the obtained flow correlation may not be high and stable enough to be used for profiling service performance in normal operation. A large window is more likely to produce high and stable correlation as there are more samples to use. But it also means higher computational cost and larger detection latency. From



(a) Flow correlation at component *lb0*

(b) Flow correlation at component *ap1*

(c) Flow correlation at component *ap2*

(d) Flow correlation at component *sq0*

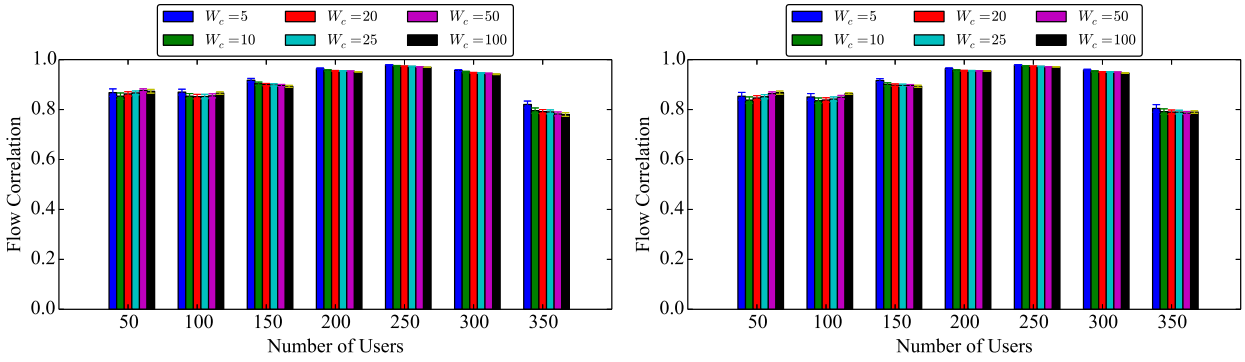
Figure 8.6: The stability of flow correlation with different correlation windows under varying workload intensity in Olio web application

all above experiments with different applications, a correlation window of size $W_c \geq 25$ is good enough for the flow correlation analysis. Users could choose a larger window for their own needs. DMCDL automatically selects a window size as the maximum of 40 samples or the number of samples within a second for a component: $W_c = \max(40, 1/\Delta T_c)$.

8.5.3 Resource Fault Detection

Figure 8.8 shows the flow correlation at *solr0* of CloudSuite web search application. Figure 8.9 shows the flow correlation at the load balancer *lb0* in Olio web application. Figure 8.10 shows the flow correlation at the load balancer *lb0* in MediaWiki application.

- CPU fault: the CPU fault is injected during the time period $120 - 180$s. The injected component becomes slower in processing arriving requests. The response time of requests increases. The flow correlation decreases as less input request flow and output response flow are correlated within a sampling interval.



(a) Flow correlation at component *lb0*

(b) Flow correlation at component *ap1*

(c) Flow correlation at component *bap1*

(d) Flow correlation at component *sq0*

Figure 8.7: The stability of flow correlation with different correlation windows under varying workload intensity in MediaWiki application

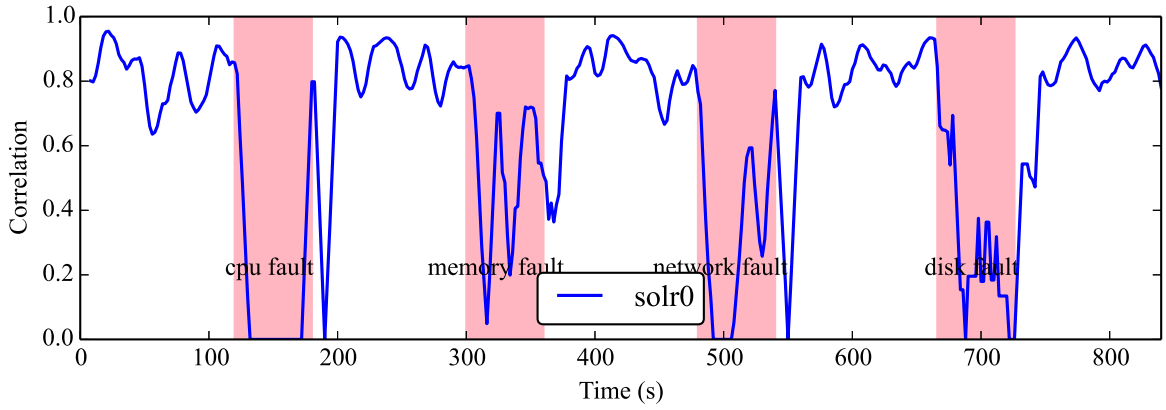


Figure 8.8: The flow correlation for resource fault detection in CloudSuite web search application

- Memory fault: the memory fault is injected into the system during the time period $300 - 360$s. During the memory fault, each request requiring to access the memory suffers much longer delay. However, as other processes keep spinning on "malloc()/free()" functions on the memory. When other processes free the memory, the load balancer gets enough memory to process the request and

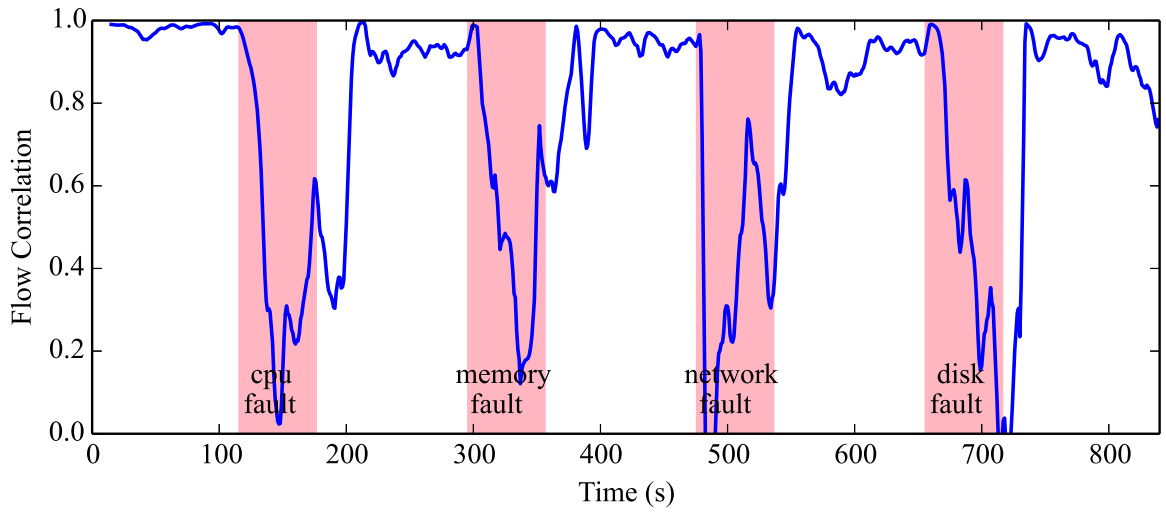


Figure 8.9: The flow correlation for resource fault detection in Olio web application

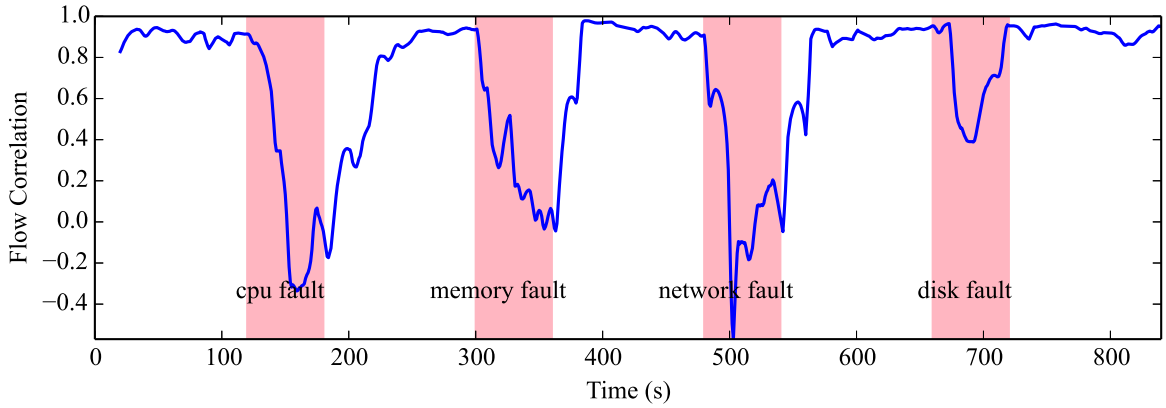


Figure 8.10: The flow correlation for resource fault detection in MediaWiki application

return the response. When other processes request for memory allocation, the service process gets stuck due to memory bottleneck. The input flow and output flow suffer frequent fluctuations during the memory fault.

- Network fault: the network fault is injected into the system during the time period $480 - 540$s. During the network fault, each outgoing packet waits longer at the local network interface. The response time increases, the output response flow no longer follows the input request flow within the normal sampling interval 500ms.

- Disk fault: the disk fault is injected into the system during the time period $660 - 720$s. The disk fault causes the subsystem to be locked. The request gets stuck and the server cannot process any request. During the fault, the load balancer cannot return any response for incoming requests. The

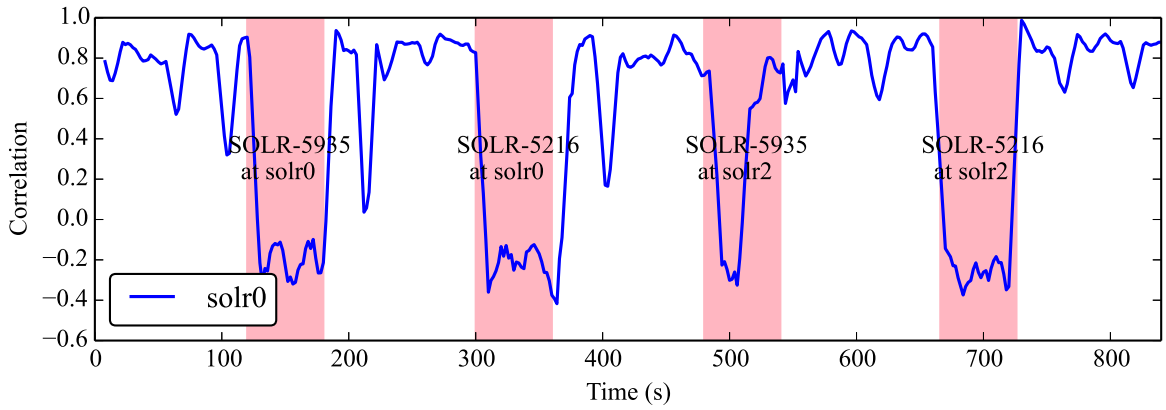


Figure 8.11: The flow correlation for performance bug detection in CloudSuite web search application

output flow is always 0 and it forces client connections to close. The flow correlation is low enough.

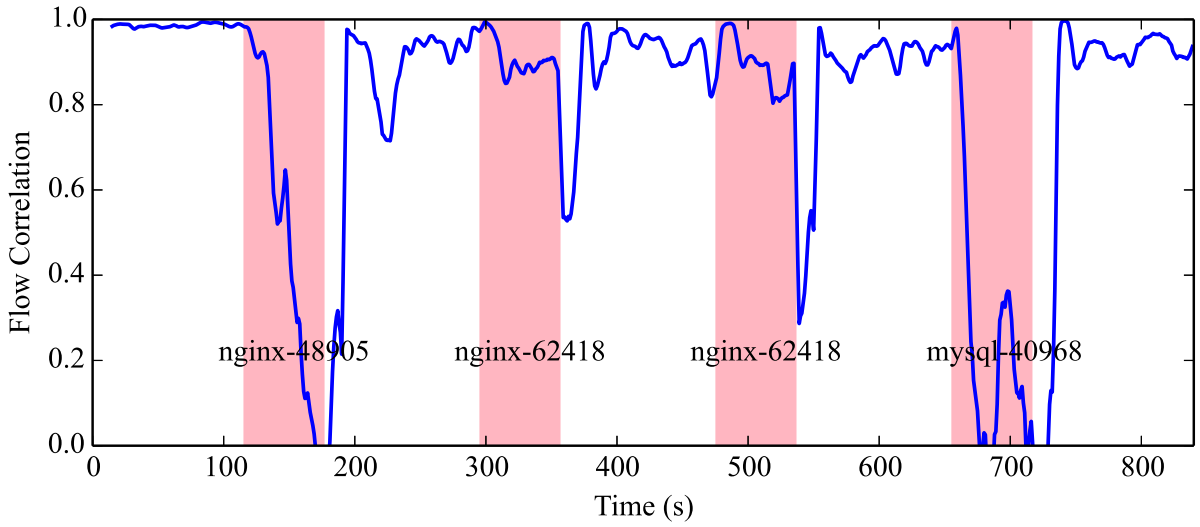
8.5.4 Common Software Fault Detection

Figure 8.11 shows the correlation coefficient between the input request flow and the output response flow at component *solr0*. The SOLR-5935 bug is injected at component *solr0* during $120 - 180$s, and at component *solr2* during $480 - 540$s. The SOLR-5216 bug is injected at component *solr0* during $300 - 360$s, and at component *solr2* during $760 - 820$s.

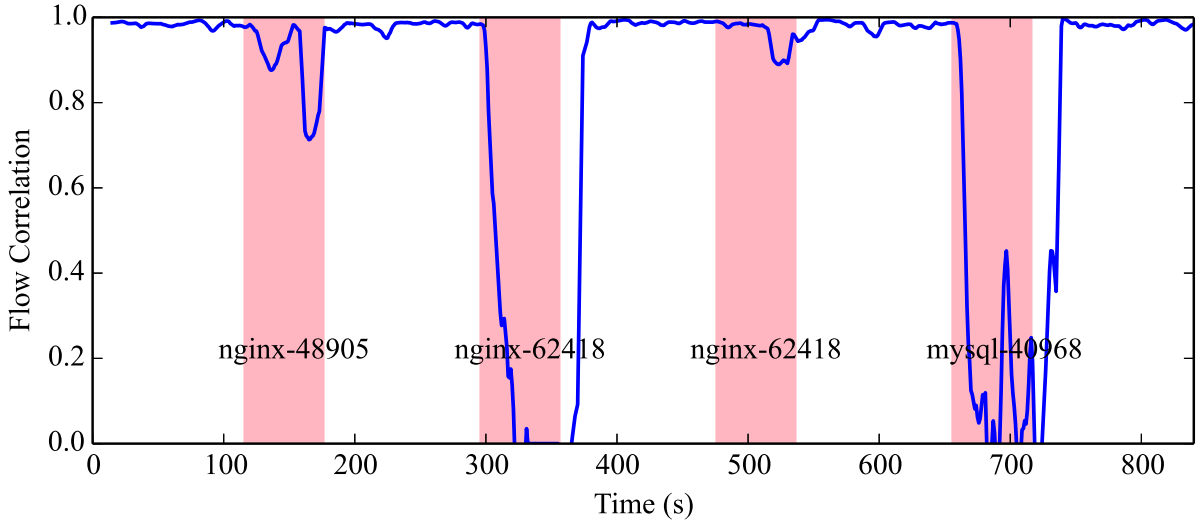
- Deadlock bug: the SOLR-5935 bug at component *solr0* causes local solr threads to fall into deadlock. The component would not be able to return responses for incoming requests. Requests have to wait in the component until the deadlock is released. The SOLR-5935 bug at the dependent component *solr2* does not necessarily cause the local service down. It takes longer to return responses for incoming requests. The number of outgoing responses per second is much lower than the number of incoming requests per second at the local component *solr0*.

- Service hang bug: the SOLR-5216 bug at component *solr0* causes the solr process to hang. The component would not be able to return responses for incoming requests. Requests have to wait in the component until the deadlock is released. The SOLR-5216 bug at the dependent component *solr2* does not necessarily cause the local service down. As all dependent components form a concurrent dependency primitive, the component *solr2* cannot return responses. The local component waits until the timeout. Finally, the component *solr0* returns incomplete responses back to clients.

Figure 6.11 shows the flow correlation for performance bug detection in Olio web application. Figure 8.12a shows the flow correlation at the load balancer *lb0* and Figure 8.12a shows the flow correlation



(a) DMCDL detection result at *lb0*



(b) DMCDL detection result at *ap1*

Figure 8.12: The flow correlation for performance bug detection in Olio web application

the application server *ap1*.

- Bug HTTPD-48905: the bug injected into the Apache load balancer *lb0* causes many child processes to hang. The hanged child processes cannot process incoming requests. Most incoming requests have to wait in the component until being processed by remaining normal child processes. It takes longer to return responses for incoming requests. The flow correlation decreases.

- Bug Nginx-62418: the bug injected into the Nginx application server 1 *ap1* causes Nginx worker processes to crash. The component cannot be able to return responses for incoming requests. Re-

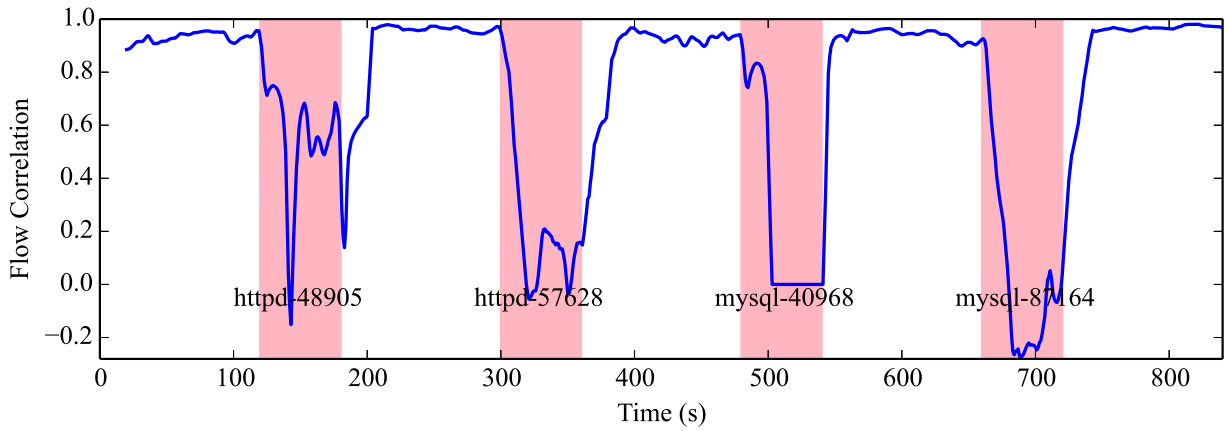


Figure 8.13: The flow correlation for performance bug detection in MediaWiki application

quests have to wait in the component until worker processes recover. The load balancer *lb0* still directs requests to the other application server *ap2*. The flow correlation does not decrease as most incoming requests are handled by the normal application server and responses are also returned by the normal application server. The same discussion applies to the case where the bug is injected to the Nginx application server 2 *ap2*.

- Bug MySQL-40968: the bug at the database component *sq0* causes the MySQL process to hang without any error message. The component would not be able to return any response for incoming requests. Requests have to wait in the component until the MySQL process recovers. The output flow is always 0 during the faulty period. The flow correlation decreases to 0 and DMCDL triggers alert at component *sq0*.

Figure 8.13 shows the flow correlation at component *ap1* by DMCDL in MediaWiki application.

- Bug HTTPD-48905: the bug injected into the Apache application server 1 *ap1* causes some child processes to hang. The hanged child processes cannot process any request. The Apache is still able to spawn new child processes to handle incoming connections and requests. Those requests waiting to be processed by hanged processes would not get their responses. Most incoming requests have to wait in the component until being processed by remaining normal child processes or new child processes. The number of responses no longer changes with the number of requests, and the flow correlation decreases.

- Bug HTTPD-57628: the bug injected into the Apache application server 1 *ap1* causes child processes to crash after they finish processing a high number of requests. The child processes run much

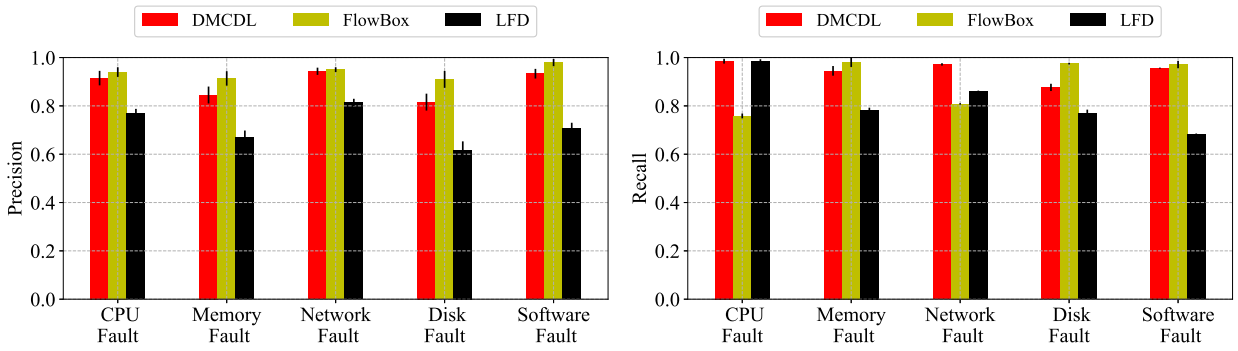
slower than normal operation. It causes the response time to increase significantly. The number of responses no longer changes with the number of requests, and the flow correlation decreases.

- Bug MySQL-40968: the bug injected at the database component *sq0* causes the MySQL process to hang without any error message. The database component would not be able to return any response for queries from *ap1* to *sq0*. Requests have to wait in the system until MySQL is normal again. The output flow is always 0 during the faulty period. The flow correlation decreases to 0 and DMCDL triggers alert at component *ap1*.

- Bug MySQL-87164: the bug injected into the database *sq0* causes MySQL to run much slower than normal operation. The queries sent from *ap1* to *sq0* experience much higher response time. The local response time of *ap1* also increases as a result. The flow correlation decreases at *ap1*.

8.5.5 Anomaly Detection Performance

We run extensive experiments under varying workload scenarios. In each experiment, we inject different kinds of faults randomly into different components. We show the performance of DMCDL for anomaly detection for various Cloud applications: CloudSuite web search application in Figure 8.14, Olio application in Figure 8.15, MediaWiki application in Figure 8.16. We compare DMCDL with two other black-box anomaly detection methods LFD [32], and FlowBox [45].

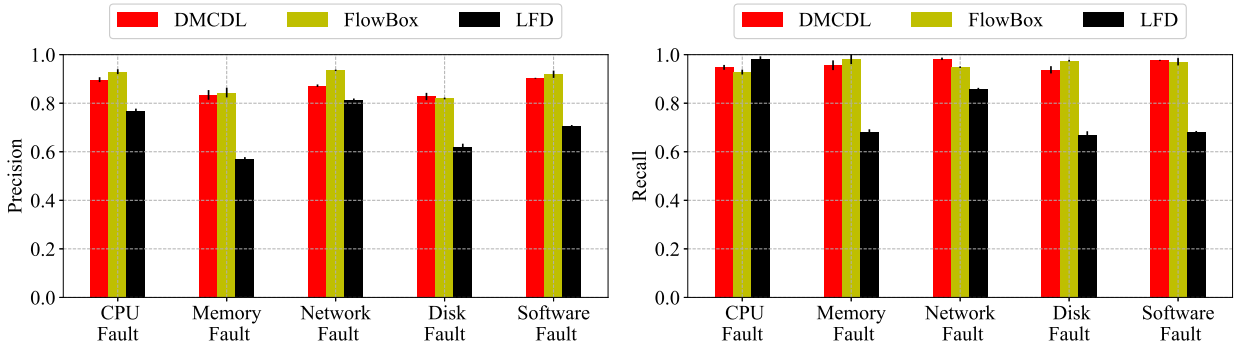


(a) The detection precision

(b) The detection recall

Figure 8.14: The detection performance of DMCDL and other methods in CloudSuite web search application

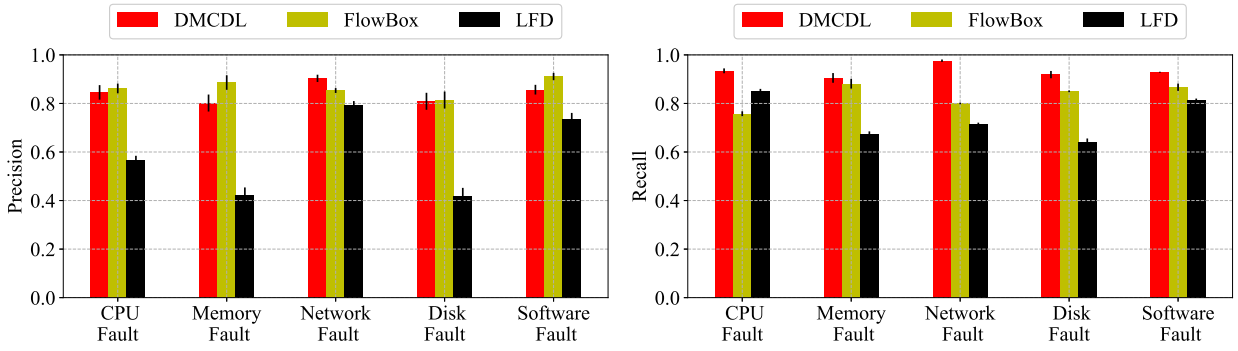
We observe that both DMCDL and FlowBox achieves high detection precision and low false positive for different faults. LFD has the worst detection performance for all different faults. The main reason is that it selects the highest correlation between the user-space CPU utilization and other resource utilization as the metric to characterize the service performance. It is prone to high false positive and high false negative.



(a) The detection precision

(b) The detection recall

Figure 8.15: The detection performance of DMCDL and other methods in Olio web application



(a) The detection precision

(b) The detection recall

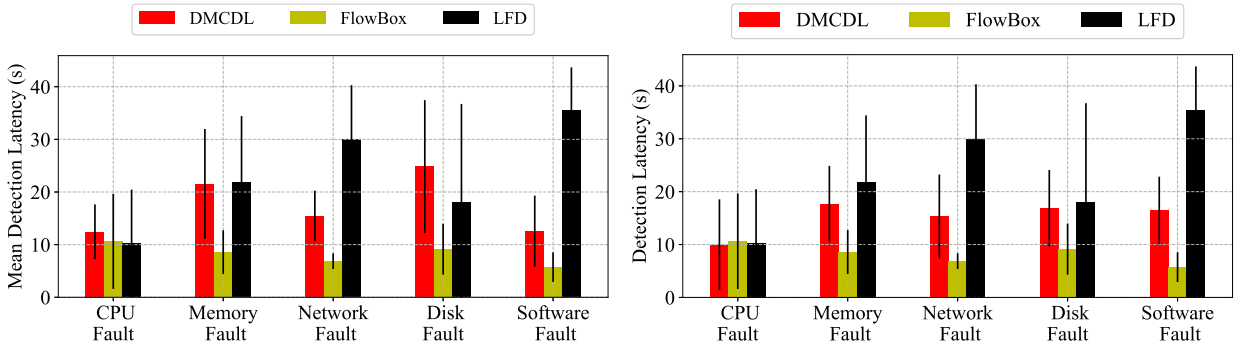
Figure 8.16: The detection performance of DMCDL and other methods in MediaWiki application

There are two main reasons. One is that if the server does not use too much user-space CPU utilization, the correlation is not meaningful. The other reason is that some resource utilization metrics are still highly correlated with the user-space CPU utilization even when the server has different faults.

Figure 8.17 shows the mean detection latency for different kinds of faults in the CloudSuite web search application. The error bar denotes the standard deviation of the detection latency. FlowBox achieves consistently small detection latency. DMCDL and LFD have longer delay at detecting memory, network and software faults. It is because LFD uses a longer time window for computing the correlation coefficient. The fault causes the correlation coefficient to decrease gradually and the window has to move long enough to include certain amount of abnormal samples to trigger anomaly alert.

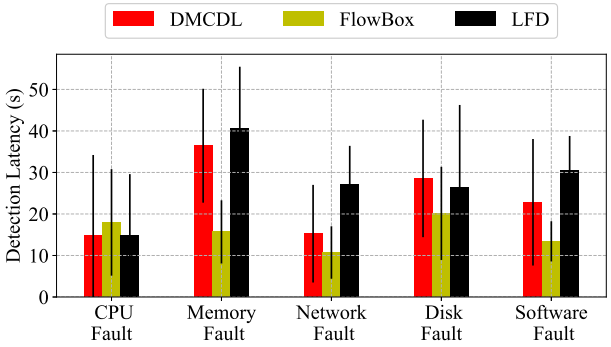
8.5.6 Anomaly Localization Case Study

In this section, we evaluate DMFDL for anomaly localization in different Cloud applications when performance anomalies are injected at different components.



(a) The detection latency in CloudSuite

(b) The detection latency in Olio



(c) The detection latency in MediaWiki

Figure 8.17: The detection latency of DMCDL and other methods in different Cloud applications

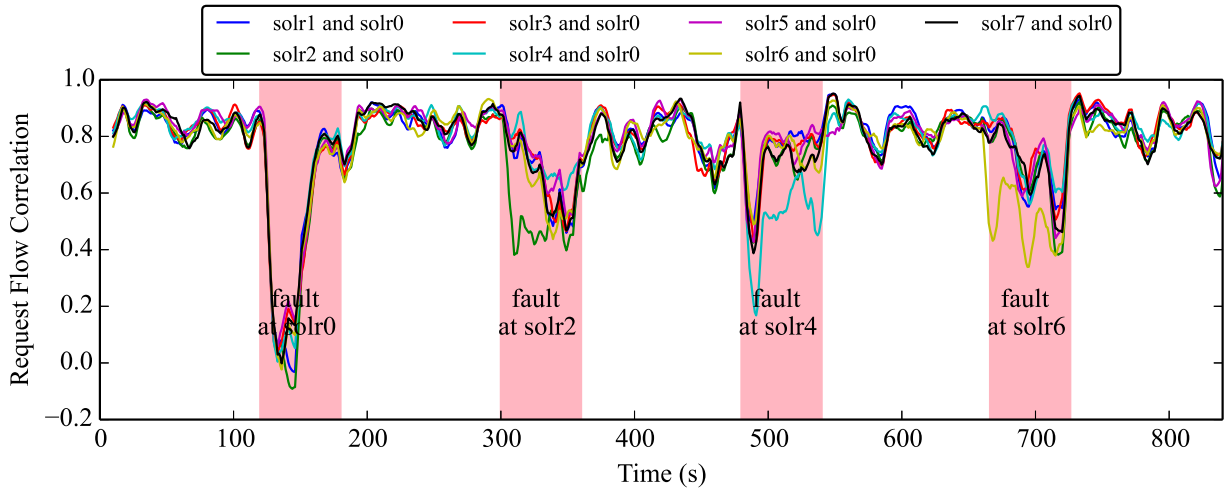


Figure 8.18: The request flow correlation for anomaly localization in CloudSuite web search application

CloudSuite Web Search Application

Figure 8.18 shows the request flow correlation at component solr0 when faults are injected to *solr0* (120 − 180s), *solr2* (300 − 360s), *solr4* (480 − 540s), and *solr6* (660 − 720s).

When the fault injected at component *solr0*, the service time of *solr0* increases. But it has similar

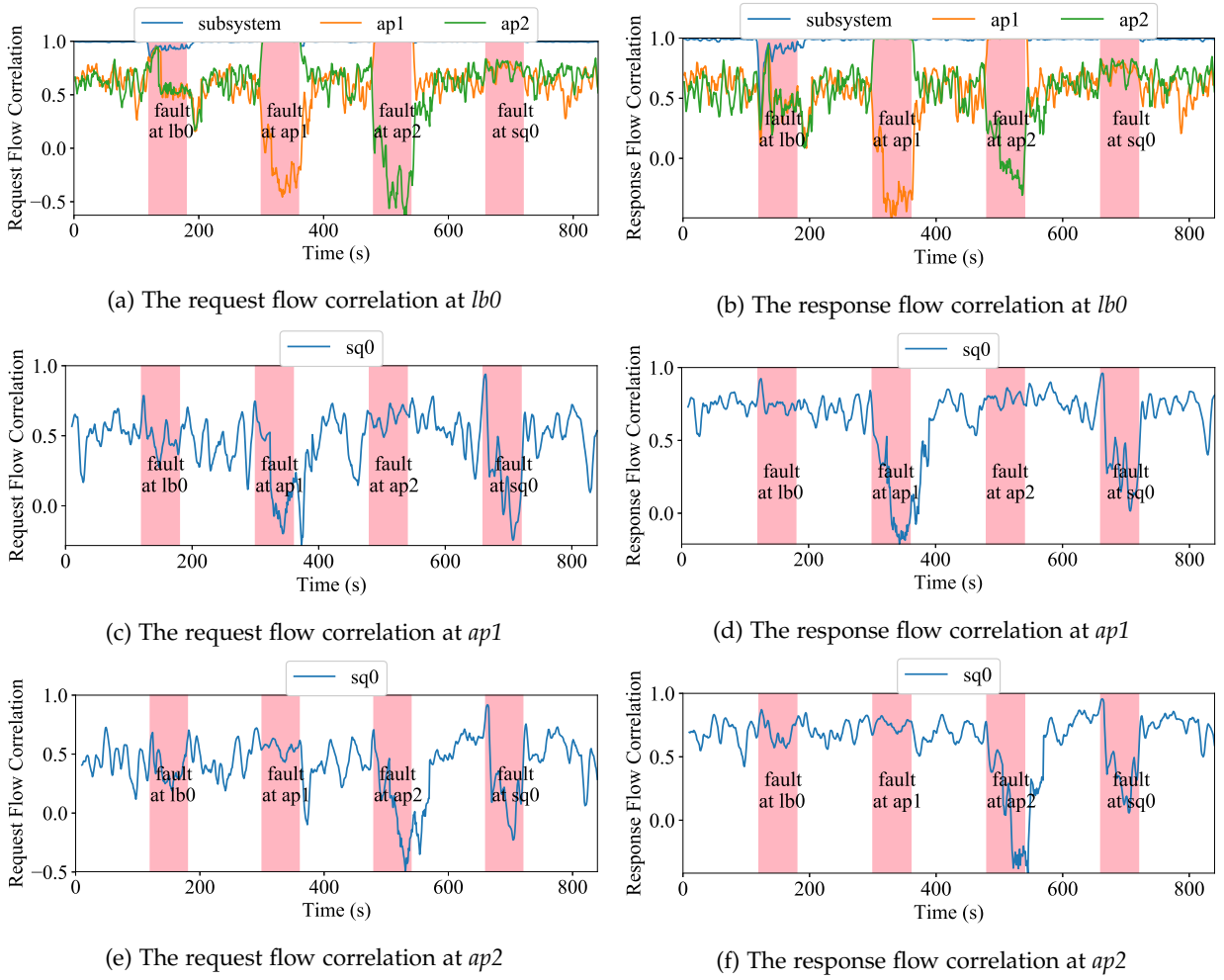
impact on the interaction between the local component and its dependent components. The request flow correlation of dependent components decreases almost similarly and concurrently. When component *solr2* is faulty, its response time increases. It also causes requests to wait longer at component *solr0*, and the request flow correlation of other normal dependent components also decreases. The interaction between component *solr2* and component *solr0* is affected the most and the request flow correlation of *solr2* decreases much more than that of other normal dependent components. The same localization analysis applies to fault at *solr4* and *solr6*.

Olio Web Application

Figure 8.19 shows the request flow correlations and response flow correlations at different components of Olio web application when faults are injected to *lb0* (120-180s), *ap1* (300-360s), *ap2* (480-540s), and *sq0* (660-720s). The load balancer is configured with the session-based scheduling policy. The load is equally distributed to two application servers. DMCDL detects faults obviously when there are faults at the load balancer,

- Figure 8.19a shows the request flow correlation at component *lb0*. The fault at *lb0* causes requests to stay longer time at the load balancer. The request flow correlation of component *ap1* and *ap2*, and the correlation between the local input request flow and the sum of output request flow to both dependent components (denoted as "subsystem") decreases as well. The fault at *ap1* causes arrived requests at *lb0* to wait longer time when it tries to send requests to *ap1*. The delay between the input request flow and the output request flow to *ap1* increases, and it causes the request flow correlation of *ap1* to decrease. The fault at *ap2* causes arrived requests at *lb0* to wait more time when it tries to send requests to *ap2*. The delay between the input request flow and the output request flow to *ap2* increases, and it causes the request flow correlation of *ap2* to decrease. The fault at *sq0* does not affect the waiting time and service time at *lb0*. The delay between the input request flow and the output request flow to its dependent components.

- The request flow correlation of *ap1* and *ap2* does not decrease. Figure 8.19c shows the request flow correlation at component *ap1*. The fault at *lb0* causes requests to spend more time at the load balancer. The input request flow at *ap1* may be affected, but the interaction between *ap1* and its dependent components is almost not affected. The request flow correlation of its dependent components (*bap1, bap2, sq0*) decreases a bit. When *ap1* is faulty, the local input request flow waits longer before the causal request flow to dependent components. The request flow correlation of all dependent components decreases a lot. When *ap2* is faulty, the interaction between *ap1* and its



(a) The request flow correlation at *lb0*

(b) The response flow correlation at *lb0*

(c) The request flow correlation at *ap1*

(d) The response flow correlation at *ap1*

(e) The request flow correlation at *ap2*

(f) The response flow correlation at *ap2*

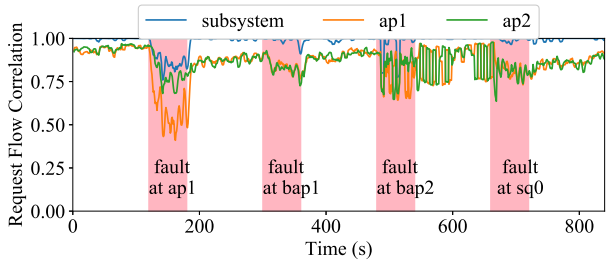
Figure 8.19: The request/response flow correlation for anomaly localization in Olio web application

dependent components is still normal. The request flow correlation of *ap1*'s dependent components is normal.

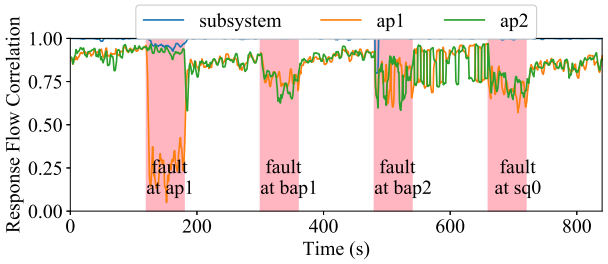
The fault at *sq0* causes longer waiting time at *ap1* when *ap1* tries to send requests to *sq0*. The request flow correlation of *sq0* decreases a lot compared to normal operation. The other two dependent components *bap1* and *bap2* are still normal and *ap1* still interacts normally with them. The request flow correlation of *bap1* and *bap2* is normal.

MediaWiki Application

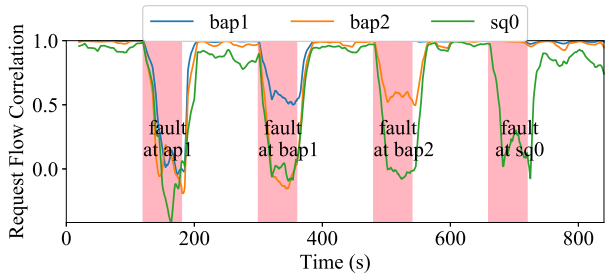
In MediaWiki application, there are three components (*bap1*, *bap2*, *sq0*) forming a composite dependency primitive with components (*ap1* and *ap2*). The faults are injected to component *ap1* (120-180s), *bap1* (300-360s), *bap2* (480-540s), and *sq0* (660-720s).



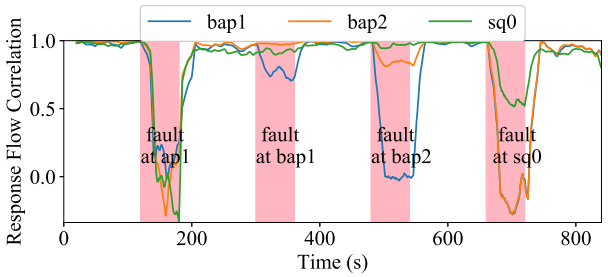
(a) The request flow correlation at *lb0*



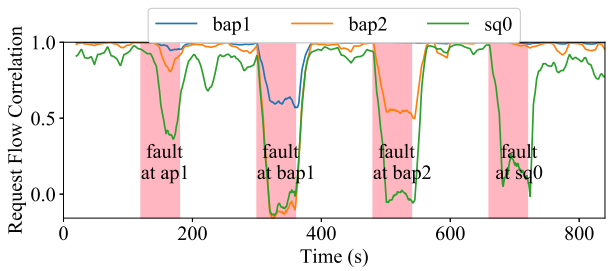
(b) The response flow correlation at *lb0*



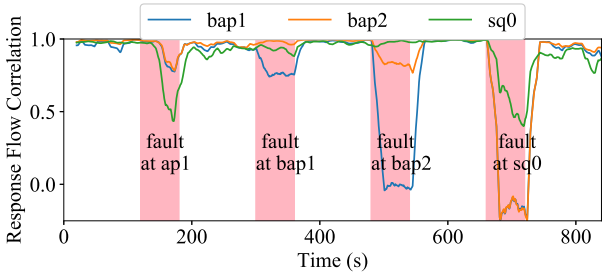
(c) The request flow correlation at *ap1*



(d) The response flow correlation at *ap1*



(e) The request flow correlation at *ap2*



(f) The response flow correlation at *ap2*

Figure 8.20: The request/response flow correlation for anomaly localization in MediaWiki application

Figure **??** shows the request flow correlation and Figure **??** shows the response flow correlation at different components. When component *ap1* is faulty, DMCDL agents for component *lb0*, *ap1*, and *sq0* detect anomalies. The agent of *lb0* further checks the request flow correlation and response flow correlation. The request flow correlation of *ap1* decreases much more than the other dependent component *ap1*. DMCDL agent of *lb0* pinpoints its dependent component *ap1* as faulty. The agent of *ap1* checks the request flow correlation and response flow correlation of all dependent components. The request flow correlation and response flow correlation of all dependent components decrease. DMCDL pinpoints the local component *ap1* as faulty. The agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

When component *bap1* is faulty, DMCDL agents for its upstream component *lb0*, *ap1*, *ap2*, and *bap1* detect anomalies. The agent of *lb0* further checks the request flow correlation and response flow correlation. The request flow correlation of both dependent components and the subsystem almost does not

decrease. DMCDL agent of *lb0* pinpoints the subsystem as faulty. The agent of *ap1* checks the request flow correlation and response flow correlation of all dependent components. The request flow correlation of all dependent components decreases, but the request flow correlation of *bap2* and *sq0* decreases much more than that of *bap1*. The response flow correlation of all dependent components almost does not decrease. DMCDL agent of *ap1* pinpoints the dependent component *bap1* as faulty. Similarly, DMCDL agent of *ap2* also pinpoints *bap1* as faulty. DMCDL agent of *bap1* does not perform localization as *bap1* does not have any dependent component.

When component *bap2* is faulty, DMCDL agents for its upstream component *lb0*, *ap1*, *ap2*, and *bap1* detect anomalies. The agent of *lb0* further checks the request flow correlation and response flow correlation. The request flow correlation of both dependent components decreases simultaneously. The request flow correlation and the response flow correlation of its subsystem almost does not decrease. DMCDL agent of *lb0* reports the local component *lb0* as normal, and its subsystem as faulty. The agent of *ap1* checks the request flow correlation and response flow correlation of all dependent components. The request flow correlation of dependent components *bap1* does not decrease. The request flow correlation of *bap2* and *sq0* decreases, but the request flow correlation of *sq0* decreases much more than that of *bap2*. The response flow correlation of *sq0* does not decrease. The response flow correlation of *bap1* decreases much more than that of *bap2*. It shows that the delay between when *bap1* returns responses to *ap1* and when *ap1* returns responses back to its upstream components increases a lot. DMCDL agent of *ap1* pinpoints *bap2* as faulty. Similarly, DMCDL agent of *ap2* also pinpoints *bap2* as faulty. DMCDL agent of *bap2* does not perform localization as *bap2* does not have any dependent component.

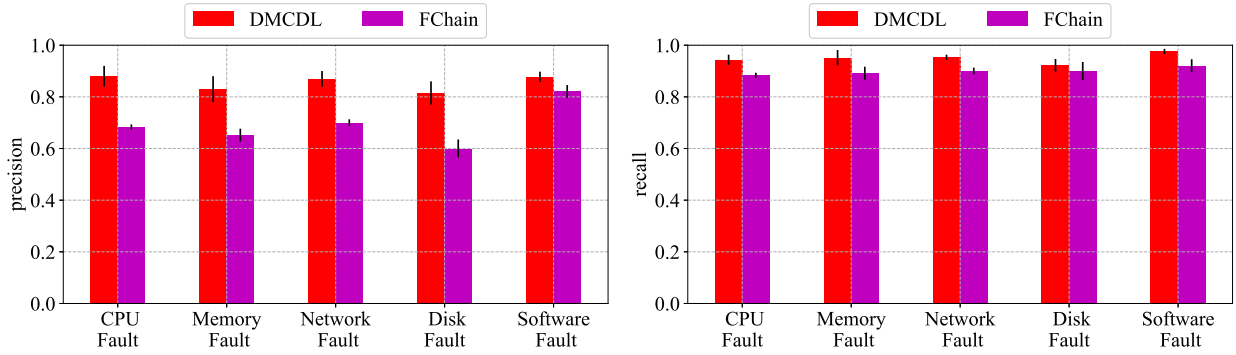
When component *sq0* is faulty, DMCDL agents for its upstream component *lb0*, *ap1*, *ap2*, and *sq0* detect anomalies. The agent of *lb0* further checks the request flow correlation and response flow correlation. The request flow correlation of both dependent components decreases simultaneously. The request flow correlation and the response flow correlation of its subsystem almost does not decrease. DMCDL agent of *lb0* reports the local component *lb0* as normal, and its subsystem as faulty. The agent of *ap1* checks the request flow correlation and response flow correlation of all dependent components. The request flow correlation of dependent components *bap1* and *bap2* does not decrease. The request flow correlation of *sq0* decreases a lot. It shows that *sq0* or its subsystem component is probably faulty. The response flow correlation of *sq0* decreases a little, but the response flow correlation of *bap1* and *bap2* decreases much more than that of *sq0*. It shows that the delay between when *bap2* returns responses to *ap1* and when *ap1* returns responses back to its upstream components increases a lot. DMCDL agent of *ap1* pinpoints *sq0* as faulty. Similarly, DMCDL agent of *ap2* also pinpoints *sq0* as faulty. DMCDL agent of *sq0* does not perform localization as *sq0* does not have any dependent component.

8.5.7 Anomaly Localization Performance

To get the performance of DMCDL in anomaly localization, we run extensive experiments under varying workload conditions with different faults randomly injected into components of these applications. We compare the performance of DMCDL with FChain in anomaly localization.

We show the precision and recall of both DMCDL and FChain in localizing faults in different applications: CloudSuite web search application in Figure **??**, Olio web application in Figure 8.19, MediaWiki application in Figure **??**.

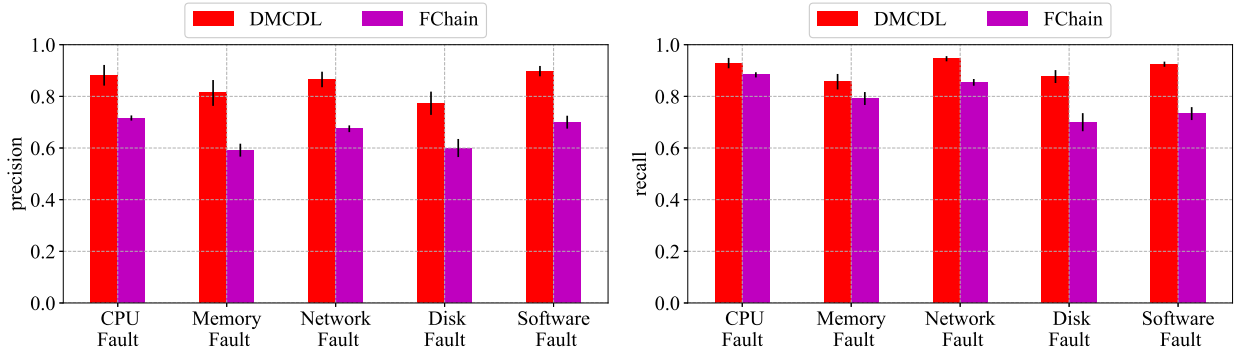
DMCDL combines the dependency model and the flow correlation analysis in anomaly localization. Rather than relying simply on the chronological order of changing points of different components. We show that the causal execution order is important and the flow correlation change is more accurate in anomaly localization. DMCDL has much lower false positive than FChain for all different faults. The



(a) The localization precision

(b) The localization recall

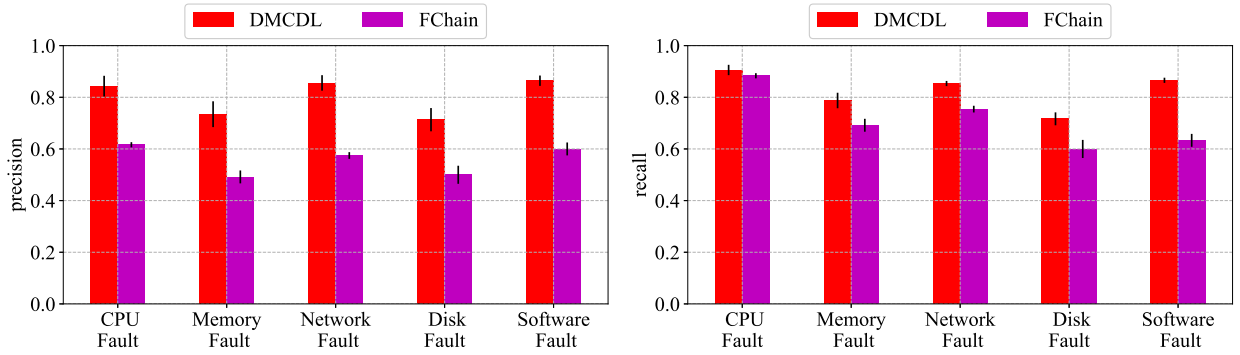
Figure 8.21: The localization performance of DMCDL and other methods CloudSuite web search application



(a) The localization precision

(b) The localization recall

Figure 8.22: The localization performance of DMCDL and other methods in Olio web application



(a) The localization precision

(b) The localization recall

Figure 8.23: The localization performance of DMCDL and other methods in MediaWiki application

main reason is that the fault propagates very quickly due to high throughput communication between components that interact with each other.

8.5.8 Overhead Analysis

DMCDL has lower overhead than DMADL as it only counts the number of data packets. It does not need to store any TCP flows in memory. So it has even lower overhead than DMADL. In varying workload intensities, DMCDL achieves less than 0.5% CPU overhead and negligible memory overhead.

8.6 Summary

The flow correlation analysis characterizes the relationship that the output response flow usually follows the input request flow within an acceptable time limit. The correlation coefficient evaluates how strong the relationship is linear. In real cloud applications, the request flow correlation is more obvious than the response flow correlation. We show that they together captures the performance of each component and can be used for anomaly localization.

Chapter 9

Performance Comparison

In previous chapters, we show the performance of DMADL, DMFDL, and DMCDL in detection of common resource bottleneck faults and software performance anomalies. In this chapter, we evaluate our non-intrusive methods using extensive chronic faults. Different from resource bottleneck faults or common software performance bugs. Chronic software fault could hide in the system for a long time before it causes serious performance degradation. Early detection of chronic software faults is important for further debugging and system recovery. We evaluate our methods (DMADL, DMFDL, and DMCDL) in detection of chronic software faults. After evaluation using extensive chronic faults, we do comparative analysis of DMADL, DMFDL, and DMCDL in anomaly detection and localization.

9.1 Case Study of Extensive Chronic Faults

We study extensive chronic faults in target software components using the Olio application. The application and software setup we used for extensive chronic fault evaluation are shown in Figure 9.1. We adjust the client workload to reach medium capacity on the whole system. The workload intensity is then kept steady till the end of the experiment. All experiments use the same workload configuration to maintain consistency. For each chronic fault in our evaluation, we repeat it for 5 times to avoid the effect of accidental factors in evaluation and ensure the correctness and consistency of our evaluation result.

9.1.1 Apache HTTP Server

We show the comparative analysis of DMADL, DMFDL, and DMCDL in detection of extensive chronic faults in Apache HTTP server.

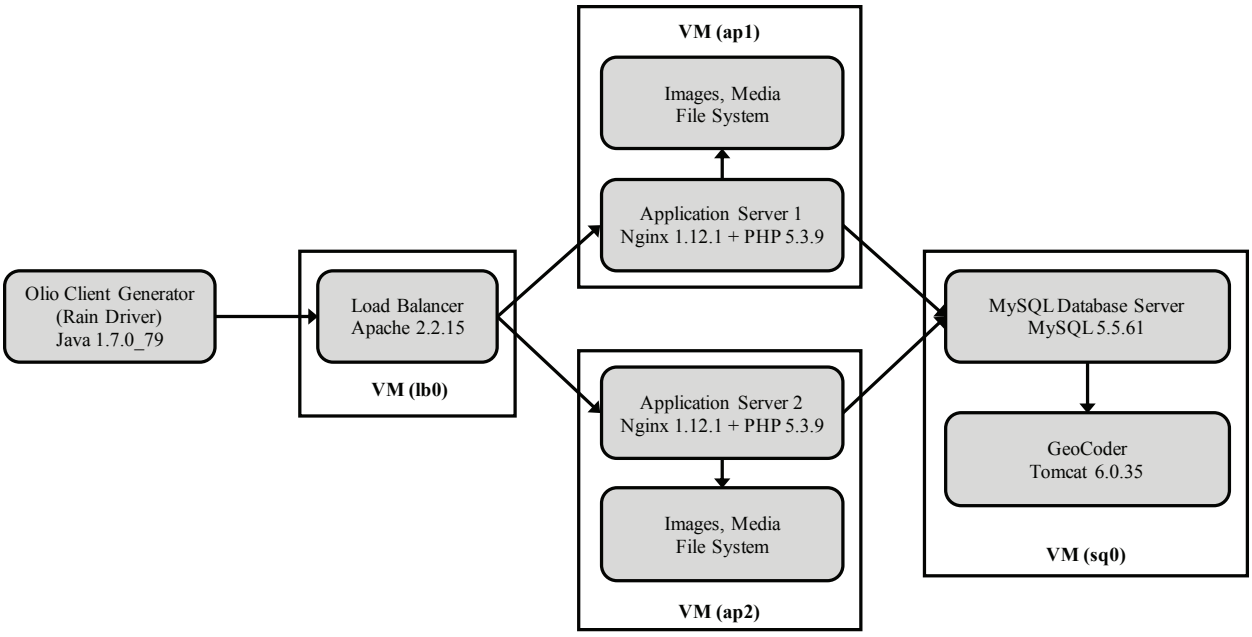


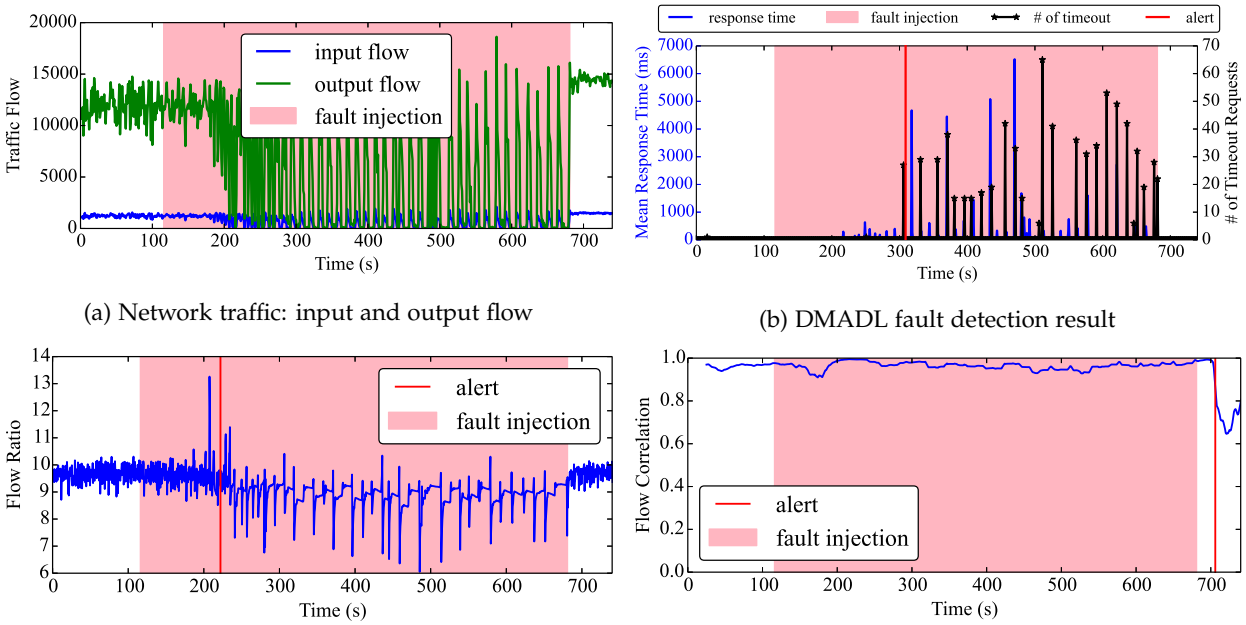
Figure 9.1: The software detail for extensive chronic fault detection

Slow Memory Leakage

Many chronic faults cause memory leakage (Apache #25667, #35404, #43223, #44975, #53435, #55296, #56271, #497077, #849272) or high memory utilization (Apache #33899, #44026, #44783, #61222) in Apache HTTP server. These faults are caused by software bugs or bad memory-related configurations. Chronic memory leak could exist in the system for a long time before it causes serious performance degradation.

The memory leak is injected to the Apache web server from time 116s at a speed of 335 KB/s. At the beginning of the memory leak, the component still has enough free memory to handle client connections and requests. When the memory leakage uses up all free memory, the system no longer has available memory to handle client connections and it has to swap data from memory to the disk. The output flow has many periodical spikes during the faulty period. In this case, requests have to wait a longer time and the response time increases. The Apache http server maintains many child processes to handle requests in the prefork module. Each process handles one request at a time and a few idle processes are maintained as slack to cater to request spikes. The critical amount of leaked memory causes some child processes to crash because of memory allocation failures. There are fewer processes to handle requests as there are fewer memory available. The server fails to respond quickly and the response time of requests increase gradually. When the memory leak reaches a critical threshold, the system completely crashes and restarts.

Figure 9.2 shows the detection result of our methods. The memory fault injection lasts for 565 seconds.



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

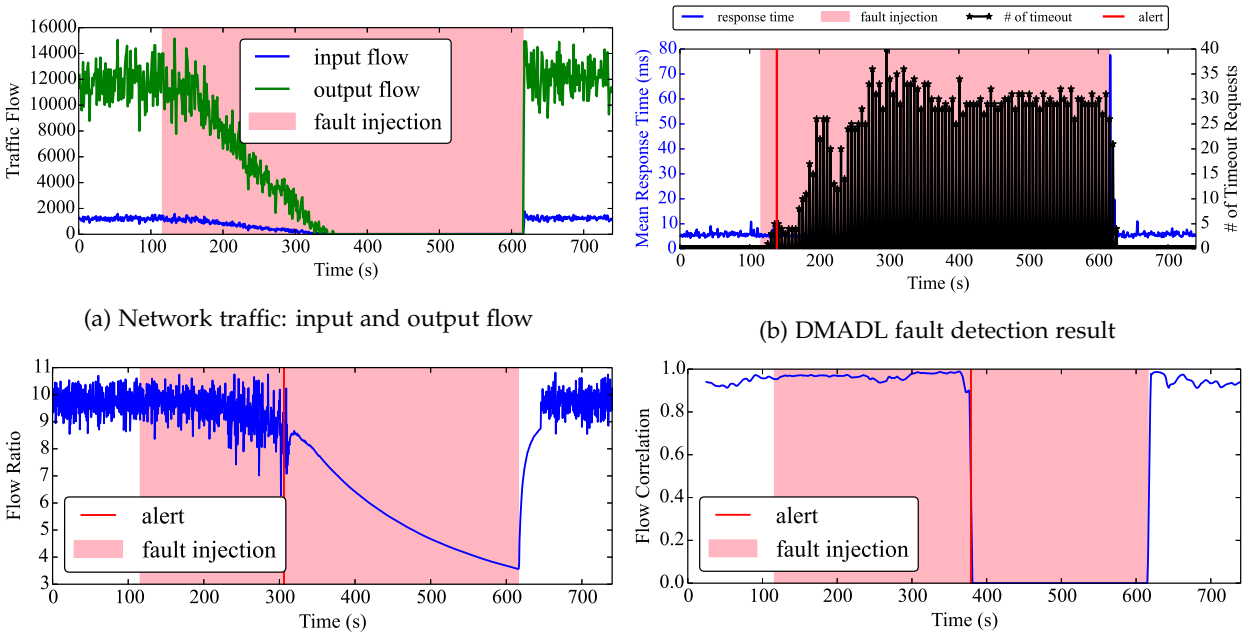
Figure 9.2: Chronic memory fault detection in Apache HTTP server

The memory leak causes spikes in the mean response time after time 200s. DMADL observes a few spikes in the mean response time at the beginning, but it does not trigger anomaly alert as it has to confirm that the frequent spikes are not caused by dynamic workload. DMADL detects the fault at time 309s. The detection latency reaches 193 seconds. The flow ratio decreases as the response time increases and DMCDL detects shortly after a few spikes. DMFDL triggers alert at time 222s, and the detection latency is much smaller (i.e., 106s). DMCDL does not detect the memory fault as the flow correlation does not decrease during the faulty period.

Child Process Failure

Many chronic faults cause child processes to crash or hang in Apache HTTP server (Apache #10266, #47370, #50702, #59798, #60071, #98979, #119128, #641968). The Apache "prefork" module has many child processes running. It cannot cause a complete server failure when some child process crash. If a child process is processing a request at the time of crash, the server will fail to respond to that request. But other processes are still working properly and returning responses. The server tries to spawn a new child process to process requests if the number of forked child processes has not reach the maximum. When the number of crashed child processes reaches a critical threshold, there is performance degradation as the server cannot fork enough processes.

The Apache server is in prefork module and it forks new processes to handle new connections. From



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

Figure 9.3: Chronic concurrent fault detection in Apache HTTP server: child process hang failure

time 116s, child processes start to crash at a speed of 1 process per second. Figure 9.3 shows the detection result of our methods for the chronic Apache concurrency fault. At the beginning of fault, the Apache server can fork new processes to process requests when there are child processes crash. Those hanged processes cannot return responses. DMADL observes timeout requests a short time after the fault injection. But those timeout requests account for only a small part of the load, and most requests are still being processed properly. So the flow ratio and flow correlation does not show significant changes. DMFDL and DMCDL do not detect the fault at the beginning. After the number of forked child processes reach the allowed maximum value, the server cannot fork new processes to replace hanged processes. The input and output flow start to decrease. The server has limited capacity to process a few requests, many requests have to wait in the server. Most input request flow cannot have the output flow within the same sampling interval. The flow ratio decreases and the flow correlation decreases. DMFDL and DMCDL detects the fault after a critical number of child processes hang in the Apache server.

In real scenarios, the child processes may crash faster or slower than our setup. If there is a high workload and a segmentation fault, all child processes could crash at once. In this case, the server cannot process any request. DMADL could observe a large number of timeout requests. There is only input flow but without any output flow. The flow ratio and the flow correlation decrease to 0 at once. All our methods could detect such a fault with a short delay. If child processes crash slowly under a certain

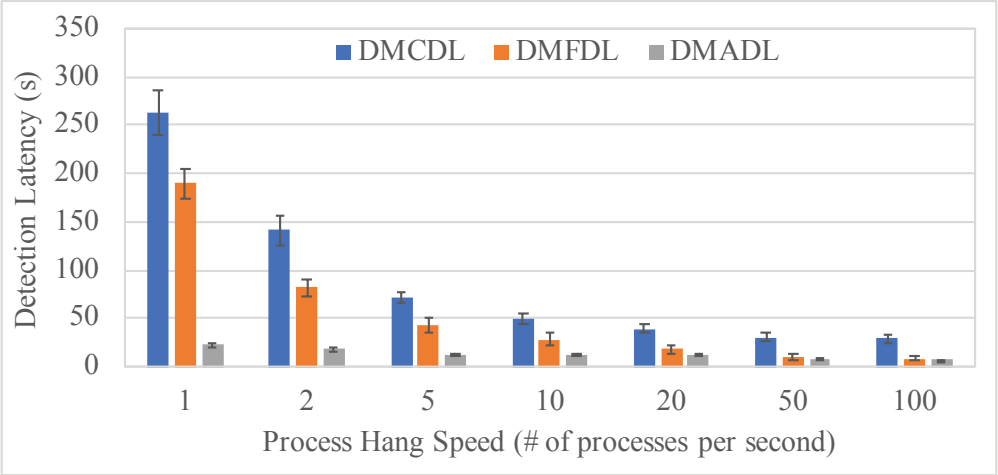


Figure 9.4: The detection latency at different process hang rates.

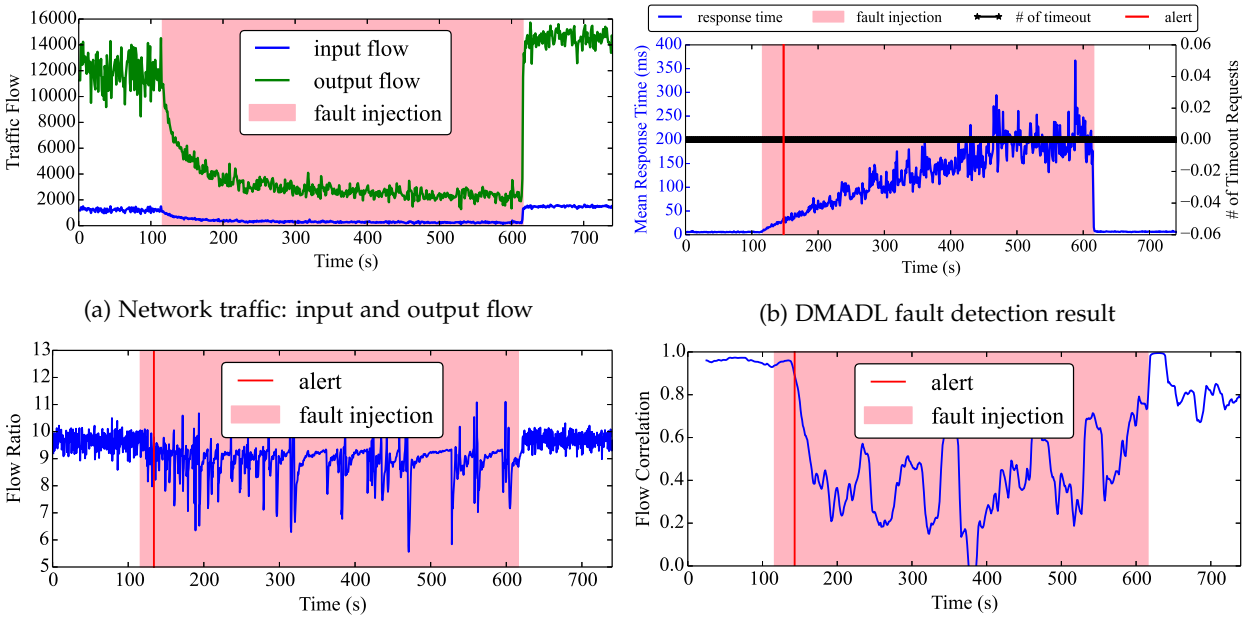
condition, DMCDL and DMFDL may not be able to detect it at the beginning. They only detect the fault when a critical number of child processes hang. But DMADL could detect the fault much earlier as it starts to see timeout requests for those hanged processes. Figure 9.4 shows the detection latency of our methods when child processes crash at different speeds in Apache.

Some software faults in Apache fail to terminate some idle child processes on time. The idle child processes consume resource that can be better used by active child processes for processing requests. The mean response time does not increase because of non-terminated idle child processes. The flow correlation and flow ratio also do not show any change under these faults. DMADL, DMFDL, and DMCDL cannot detect such kind of faults.

CPU Usage Spikes

Many chronic faults cause high CPU occupancy rate in the Apache http server (Apache #5225, #37680, #52858, #57544, #57800, #117832). The CPU usage spikes are caused by unexpected user operations or software bugs. We emulate the high CPU usage by running some CPU-intensive processes with the Apache component. We start to inject the CPU contention from time 116s. As time goes, we start more processes competing for CPU cycles and the Apache child processes suffer higher competition.

Figure 9.5 shows the detection result of our methods for the chronic Apache concurrency fault that cause CPU spikes. The fault injection period lasts for 500 seconds. During the faulty period, the server processes requests slower and the requests have a larger response time. DMADL triggers alert shortly after the injection (i.e., 32 seconds) as the mean response time increases from around 5ms to around 40ms. Many requests cannot have corresponding responses within the acceptable time limit. The flow



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

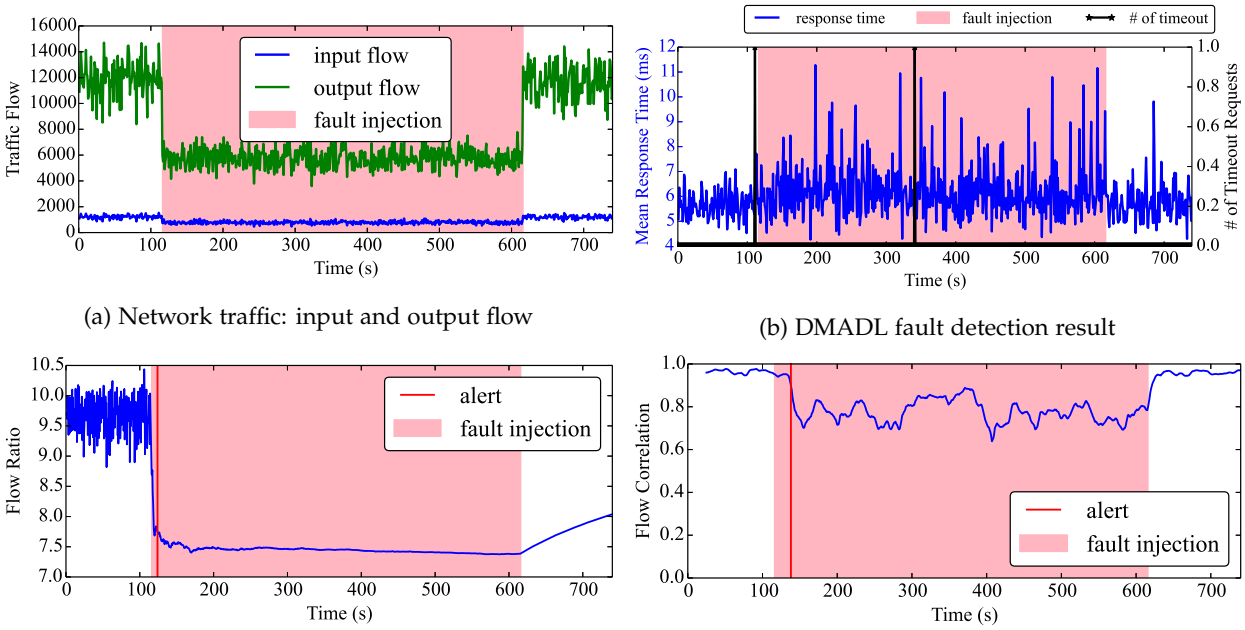
Figure 9.5: Chronic concurrency fault detection in Apache HTTP server: CPU usage spikes

ratio decreases and DMFDL triggers alert with a smaller detection delay (i.e., 18 seconds) than DMADL. DMCDL triggers alert with 27 seconds delay as the flow correlation also decreases.

Apache Truncated Response Fault

Some chronic faults cause the truncated response body in the Apache web server (Apache #27292, #50481, #56176, #57476, #61147,#908583, #1569081). Some faults cause truncated response content if the response size is too large and Apache returns an HTTP 206 status code. They can be detected by parsing the HTTP response partial content code. For some special faults, the Apache web server still responds with an HTTP 200 status code. They are caused by improper function of Apache modules returning a partial response just ends up corrupting the response. The truncated response fault is injected to the Apache web server from time 116s and the fault lasts for 500 seconds. The response time does not increase and it may decrease instead as there are less response data to transmit from the server.

Figure 9.6 shows the detection result of our methods for the chronic Apache semantic fault that causes truncated response content. DMADL fails to detect the truncated response fault as the estimated mean response time does not increase and there are no timeout requests. The number of outgoing response data packets decreases compared to normal operation. The flow ratio decreases and DMFDL detects the fault shortly after the fault injection (i.e., 8 seconds delay). The flow correlation also decreases at fault injection and DMCDL also detects the fault with a delay of 22 seconds). The DMFDL and DMCDL perform better



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

Figure 9.6: Chronic semantic fault detection in Apache HTTP server: truncated response fault

than DMADL in this case.

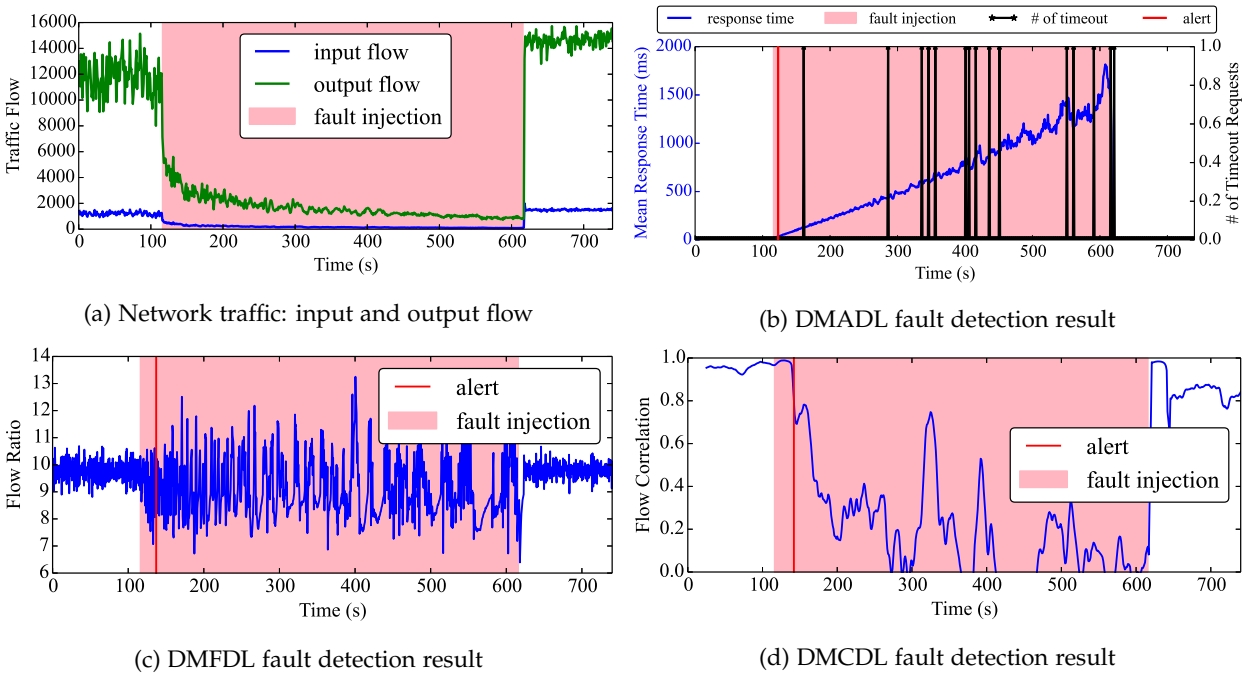
However, other faults result in malformed or corrupted response content, but the sizes of responses do not change. The response time, the flow ratio, and the flow correlation fail to detect such faults as long as malformed responses are sent out from the component within the acceptable time limit. Clients continue to behave normally on receiving a corrupted or malformed response.

Increasing Queuing Latency

Some chronic faults are not necessarily caused by memory or concurrent faults. They can be caused by human or operator mistakes, such as improper configuration of parameters in the operating system (e.g., the queue length in the network stack or server configuration scripts (e.g., the number of server processes). These chronic faults do not cause service performance degradation immediately. These faults start to affect the service performance under certain workload conditions or human operations. They cause the service response time to increase gradually and their effects are usually negligible at the beginning.

We emulate such chronic faults by gradually increasing the waiting time in the network queue. We start to increase the waiting time of each outgoing packet in the network by 5ms in the Apache component. We gradually increase the waiting time of each outgoing packet by 1ms per second until the waiting time reaches 900ms. The fault injection lasts for 500s. During the faulty period, each packet going out of the Apache server component has to wait longer. It first causes requests going to its subsystem to wait and it

also causes more responses wait at the output queue.



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

Figure 9.7: Chronic semantic fault detection in Apache HTTP server: gradually increasing queue latency

Figure 9.7 shows the detection result of our methods for the chronic Apache semantic fault that causes gradually increasing latency in the output queue of the component. The mean response time increases, and DMADL detects the fault shortly after the injection (i.e., 7 seconds) with the estimated mean response time increases from 5ms to around 20ms. The response time increases and there are less responses going out in each sampling interval. The flow ratio has a larger variance and it detects the fault with a larger detection delay (i.e., 17 seconds) compared to DMADL. DMCDL detects the fault as all responses wait longer at the Apache component and the output flow no longer changes with the input flow within the same sampling interval. DMCDL detects the fault with a larger latency of 26 seconds.

9.1.2 Nginx Application Server

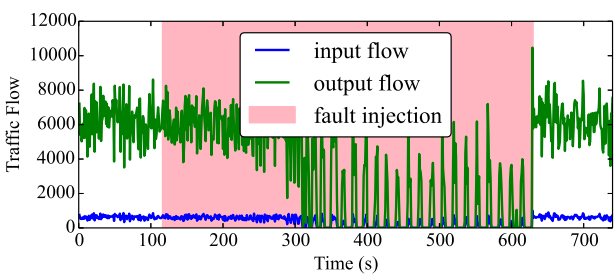
We show the comparative analysis of DMADL, DMFDL, and DMCDL in detection of extensive chronic faults in Nginx application server.

Slow Memory Leakage

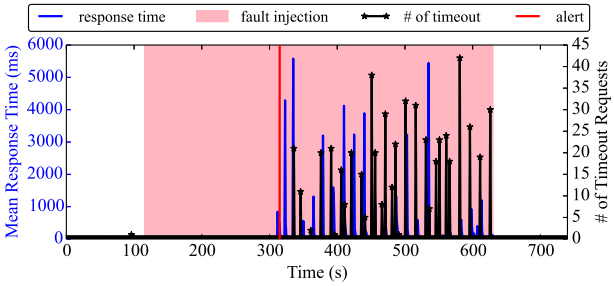
Many chronic faults cause memory leakage (Nginx #568 #871, #996, #1482, #1509, #1587) in the Nginx server. These bugs are usually caused by loaded modules or improper server configurations. Chronic

memory leak could exist and hide in the system for a long time before the Nginx server experiences performance degradation.

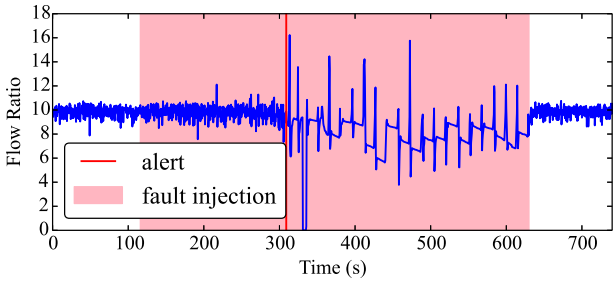
The memory leak is injected to the Apache web server from time 116s at a speed of 3485 KB/s. At the beginning of the memory leak, the component still has enough free memory to handle client connections and requests. As the leakage progresses, the Nginx server has no enough memory available and Nginx worker processes have to swap data to the disk. A critical amount of leaked memory causes worker process to hang due to memory allocation failures. The server fails to respond quickly and the response time of requests increase gradually. When the memory leak reaches a critical threshold, the system completely crashes and restarts.



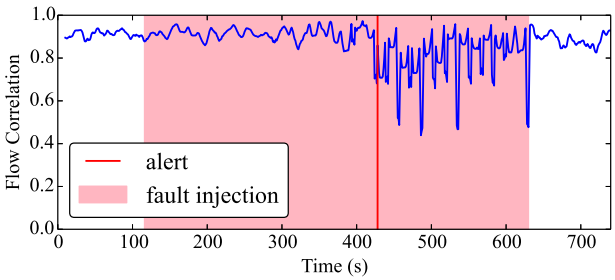
(a) Network traffic: input and output flow



(b) DMADL fault detection result



(c) DMFDL fault detection result



(d) DMCDL fault detection result

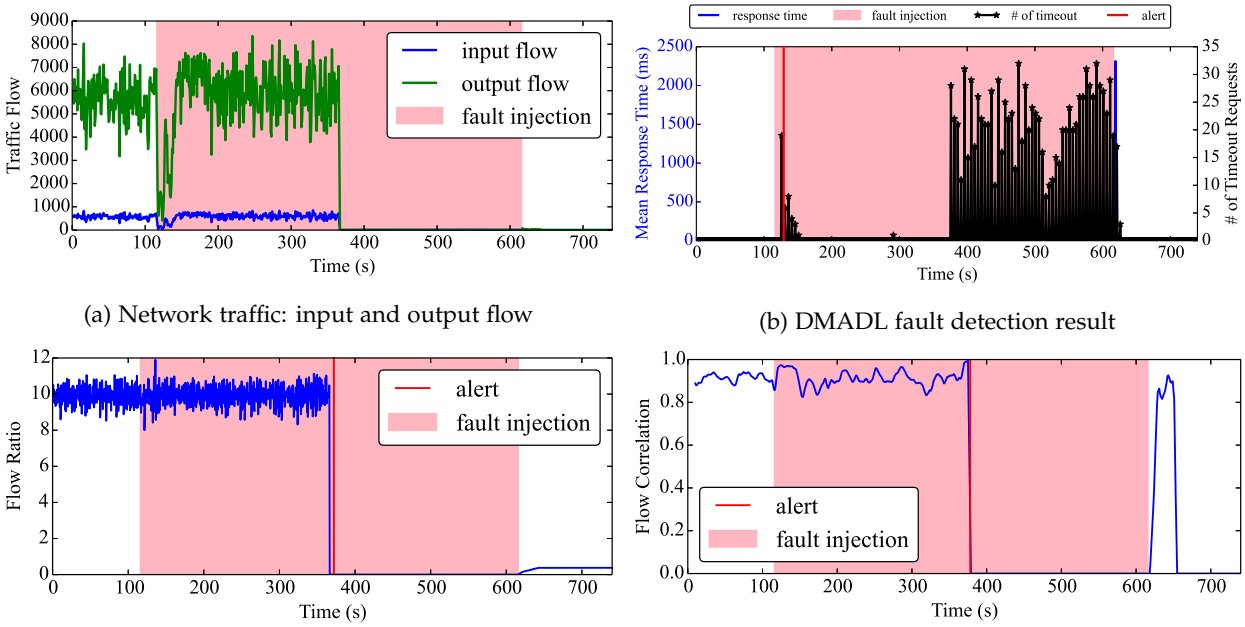
Figure 9.8: Chronic memory fault detection in Nginx HTTP server

Figure 9.8 shows the chronic Nginx memory fault detection result of our methods. The memory fault injection lasts for 514 seconds. The memory leak causes spikes in the mean response time after time 300s. DMADL triggers alert at time 315s. The detection latency reaches 99 seconds. The flow ratio decreases as the response time increases and DMFDL triggers alert at time 309s, and the detection latency is shorter (i.e., 93 seconds). The flow correlation decreases and DMCDL triggers alert after time 400s with a detection delay of 325 seconds.

Worker Process Failure

Many chronic faults in Nginx cause worker processes to crash or hang (Nginx #912, #822, #192). The Nginx server has a master process and one or multiple worker processes. Each worker process could process a large number of simultaneous connections. If a server has more than one worker processes, it does not crash completely when a worker process crashes. If the worker process is processing a request at the time of crash, the server fails to respond to that request. The client connections being handled by the crashed process are also affected and cannot have responses any more. But other worker processes still work normally and corresponding client connections still receive normal service. But it causes performance degradation gradually as each remaining normal worker process has to handle more simultaneous connections. If more worker processes crash, the service performance degradation is more obvious and our methods could detect the fault easier.

We configure Nginx application server 1 (*ap1*) with 1 master process and 2 worker processes, and each worker process could handle 1024 connections. The first worker process crashes at time 116s, and the second worker process crashes at time 366s. After both worker processes crash, there is no worker process to handle any connections at component *ap1*.



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

Figure 9.9: Chronic concurrency fault detection in Nginx HTTP server: worker process failure

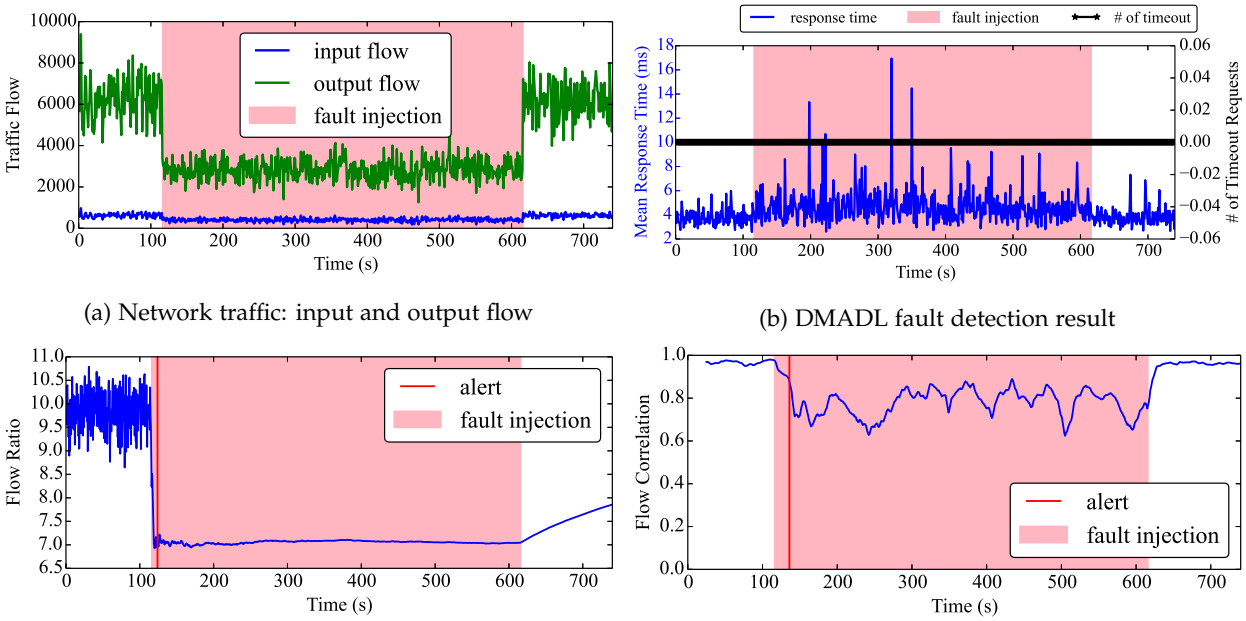
Figure 9.9 shows the detection result of our methods for the chronic Nginx concurrency fault that cause worker process failure. The input flow and the output flow decrease when the first worker process crashes.

But the load recovers very soon as the remaining normal worker process starts to handle new connections and the server still processes requests as in normal operation. There is no output flow but only very few input flows after both worker process crash. After the fault injection ends, the server starts to send responses for those queuing requests. But most of the requests are already directed to the application server 2. There is almost no workload to the application server 1. DMADL triggers anomaly alert 13 seconds after the first worker process crashes. Those connections that are being handled by the crashed worker process would no longer be responded by the server. But after that, new incoming connections are handled by the remaining normal worker process. It shows that the server still works normally after the first worker process crashes. However, after the second worker process crashes, the server has no worker process to handle any connections any more. All requests are not responded and DMADL detects timeout requests. DMFDL does not trigger any anomaly alert when the first worker process crashes. It is because only the other worker process is still processing requests normally. The flow ratio does not show significant change. When the last remaining normal worker process crashes, the server cannot return responses. The flow ratio decreases to 0 until the end of fault injection. The flow correlation also does not decrease when the first worker process crashes. It triggers alert as both worker processes crash at time 478s. After the fault injection ends, the server starts to process those requests queued inside the server and the correlation recovers for a short time. But there is almost no requests going to the server as most requests are sent to application server 2 (*ap2*). The flow correlation and flow ratio does not recover as there is almost no input flow or output flow.

Partial Content Error

Some chronic faults cause the HTTP 206 partial content error in responses (Nginx #683, #1014, #1304, #549, #1357, #1550) in the Nginx HTTP server. These faults are caused by slice module closing client connection unexpectedly or unknown errors in transferring HTTP responses. They can be detected by parsing the HTTP response status code. For some special faults, the Nginx web server still responds with an HTTP 200 status code. The 206 partial content fault is injected to the Nginx HTTP server at time 116s and the fault lasts for 500 seconds.

Figure 9.10 shows the detection result of our methods for the chronic Nginx semantic fault that causes HTTP 206 partial content error in HTTP responses. The response time does not increase and it may decrease instead as there are less response data to transmit from the server. DMADL fails to detect the truncated response fault as the estimated mean response time does not increase and there are no timeout requests. The number of outgoing response data packets decreases compared to normal operation. The flow ratio decreases and DMFDL detects the fault shortly after the fault injection (i.e., 8 seconds delay).



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

Figure 9.10: Chronic semantic fault detection in Nginx HTTP server: HTTP 206 partial content fault

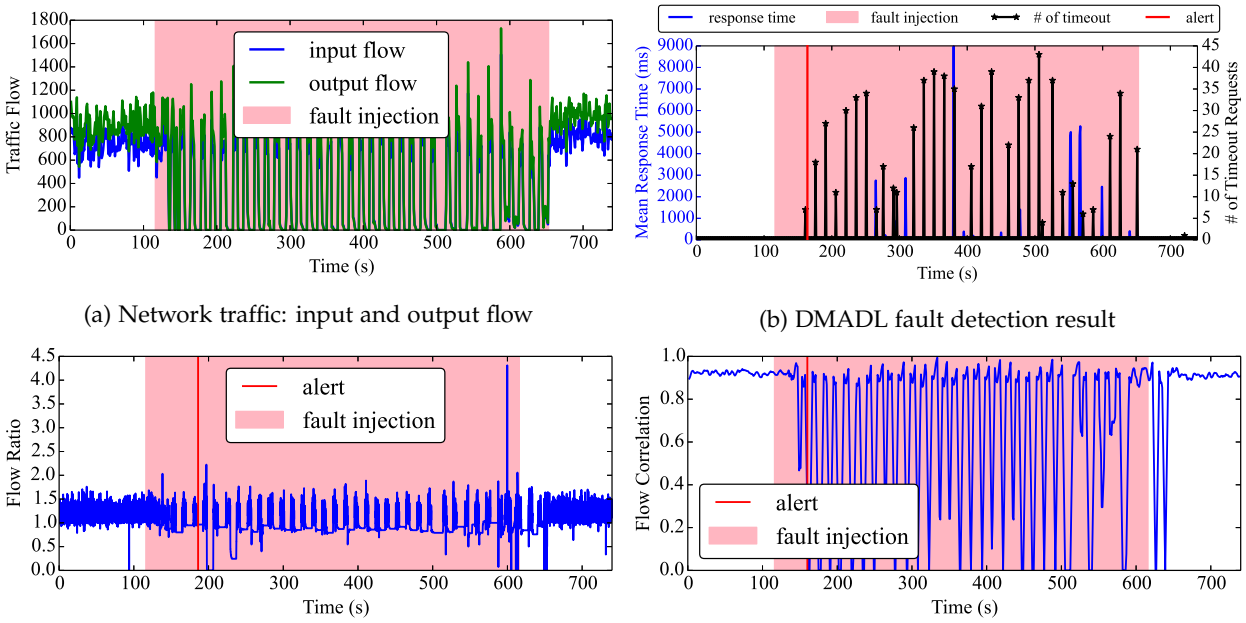
The flow correlation also decreases at fault injection and DMCDL also detects the fault with a delay of 22 seconds). The DMFDL and DMCDL perform better than DMADL in this case.

9.1.3 MySQL Database Server

We show the comparative analysis of DMADL, DMFDL, and DMCDL in detection of extensive chronic faults in MySQL database server.

Slow Memory Leakage

Many chronic faults result in memory leaks (MySQL #56924, #66740, #72885, #83047, #86082, #87501, #852477) or high memory utilization (MySQL #68287, #68514, #68980, #77403) in the MySQL database. The faults are mostly caused by clients' big queries or bugs in the software code. We inject a memory leak into the MySQL component for 6649KB/s. At the beginning of the memory leak, the component still has enough memory to handle connections and requests. When the memory leak uses up all free memory, the nginx server has no enough memory to handle connections and the worker processes have to swap data to the disk. In this case, requests have to wait a longer time and the response time increases. When the memory leak reaches a critical threshold, the system crashes and restarts. The server fails to respond quickly and the response time of requests increase gradually.



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

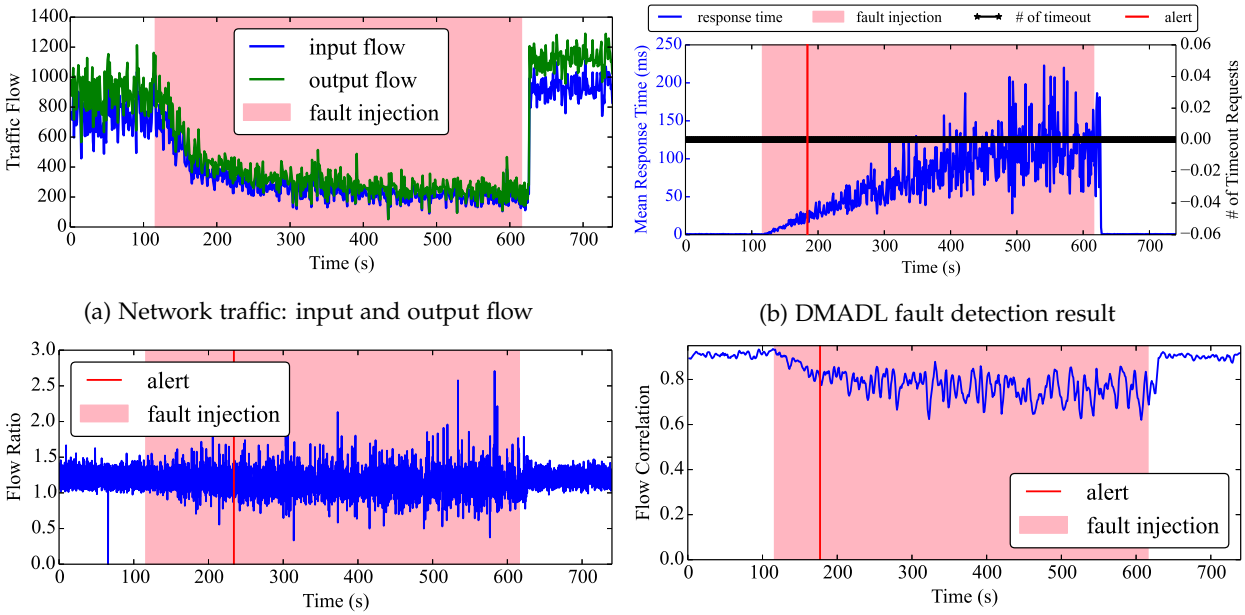
Figure 9.11: Chronic memory fault detection in MySQL database server

Figure 9.11 shows the detection result of our methods for a chronic MySQL memory fault. The memory leak fault injection lasts for 537 seconds. The memory leak causes spikes in the mean response time. DMADL detects many timeout requests triggers alert at time 164s. The detection latency is about 48 seconds. The flow ratio decreases as the response time increases and DMFDL triggers alert at time 186s, and the detection latency is a little longer (i.e., 70 seconds). The flow correlation decreases and fluctuates between 0 and 0.9. DMCDL triggers alert at time 160s with a detection latency of 44 seconds.

The performance of the MySQL server degrades as the memory leak happens. When the memory contention is not high enough, the response time of MySQL queries is almost not affected. The mean response time, the flow ratio, and the flow correlation coefficient is not affected much as a result. When a critical amount of memory is leaked, the database is unable to maintain caches due to memory allocation failures. The response time increases dramatically and all our methods could detect them.

CPU Usage Spikes

Many chronic faults cause high CPU occupancy rate and slow queries in the MySQL database (MySQL #34312, #65778, #76402, #87637). The CPU usage spikes are caused by unexpected user operations or MySQL software bugs. We emulate high CPU usage by running some CPU-intensive processes with the MySQL component. We start to inject the CPU contention from time 116s. As time goes, we create more processes competing for CPU cycles and the MySQL process suffers higher competitions.



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

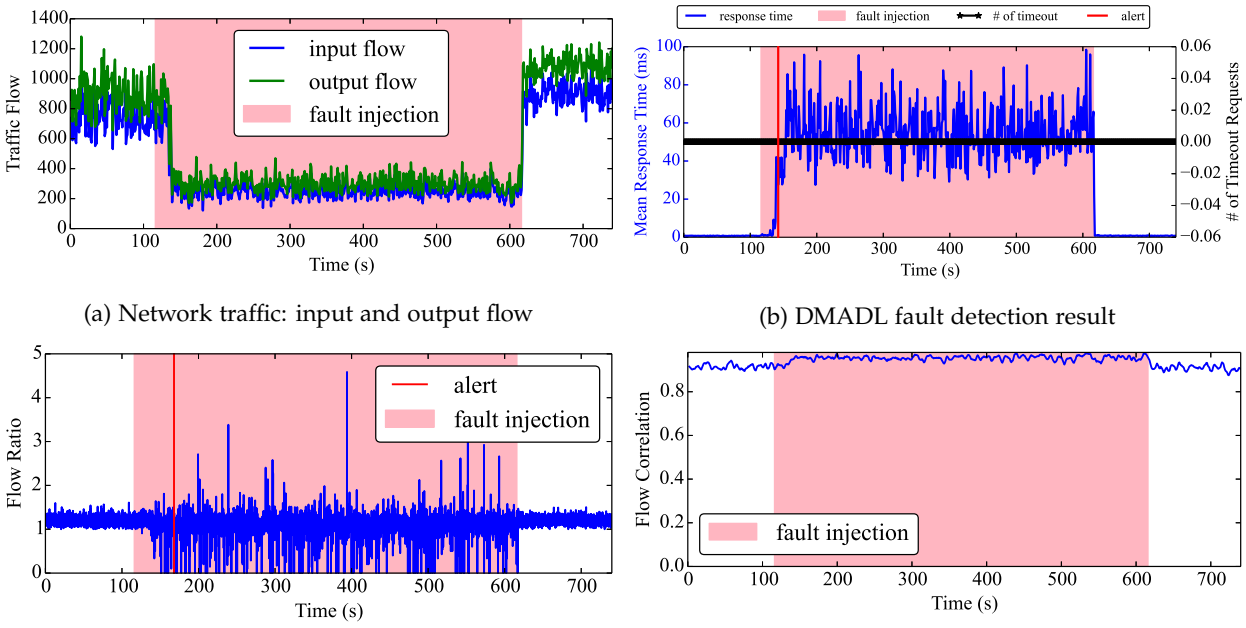
Figure 9.12: Chronic concurrency fault detection in MySQL database server: CPU usage spikes

Figure 9.12 shows the detection result of our methods for the chronic MySQL concurrency fault that cause high CPU usage. The fault injection lasts for 500s. During the faulty period, the server processes requests slower and queries have a larger response time. DMADL detects the fault as the mean response time increases gradually from less than 1ms to around 20ms. The response time increases gradually, and less requests have corresponding responses within the same interval. The flow ratio and the flow correlation decreases until the response time reaches a critical limit. The flow ratio decreases slowly and DMFDL detects the fault with a larger detection delay (i.e., 68 seconds) than DMADL. DMCDL detects the chronic fault with the largest detection latency (i.e., 124 seconds).

Poor Service Performance

Many chronic faults cause performance degradation and slow queries in the MySQL (MySQL #15815, #36525, #37633, #67252, #71130, #80989, #86215, #88834). They are not always related to CPU usage spikes, and they are caused by problems with MySQL threads. We emulate poor service performance in the MySQL database by limiting the CPU usage of MySQL threads and they compete for limited CPU cycles to process queries. We start to limit the CPU usage of threads from time 116s. As time goes, we put a lower limit on the CPU usage of MySQL threads and competition among MySQL threads increases. The fault injection lasts for 500s.

Figure 9.13 shows the detection result of our methods for the chronic MySQL concurrency fault that



(a) Network traffic: input and output flow

(b) DMADL fault detection result

(c) DMFDL fault detection result

(d) DMCDL fault detection result

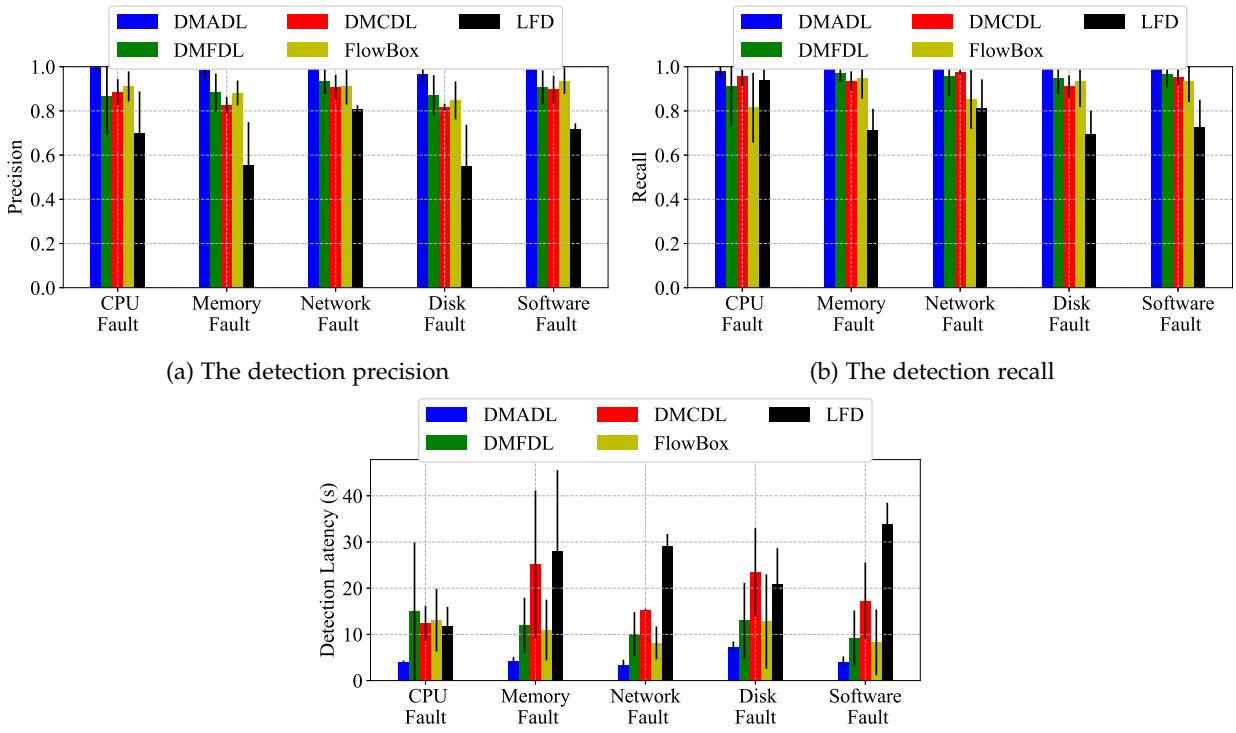
Figure 9.13: Chronic concurrency fault detection in MySQL database server: poor service performance

cause poor service performance but not CPU usage spikes. During the faulty period, the server processes requests slower and the queries have a larger response time. DMADL detects the fault shortly after the injection (i.e., 26 seconds) as the mean response time increases from less than 1ms to around 40ms. The response time increases and less requests have corresponding responses within the same interval. The flow ratio decreases and DMFDL detects the fault with a larger detection delay (i.e., 53 seconds) than DMADL. DMCDL fails to detect the faults as most requests are still responded correctly and the output flow still changes with the input flow. The flow correlation has high values in this case.

We find that four chronic faults in MySQL result in incorrect response data. One of these faults also results in user data corruption. These faults have no effect on response time and the server appears to behave correctly. The response size does not change, but the content is corrupted. The flow ratio and flow correlation do not change as well. Our methods cannot detect such kind of faults.

9.2 Overall Anomaly Detection Performance

Here we show their overall detection and localization performance of different methods in Cloud applications in Figure 9.14. The error bar shows the 95% confidence interval. DMADL achieves the best performance in anomaly detection and smallest detection latency as well. DMFDL, DMCDL, and FlowBox achieve similar detection precision and recall. LFD has the worst detection performance. It is especially



(a) The detection precision

(b) The detection recall

(c) The detection latency

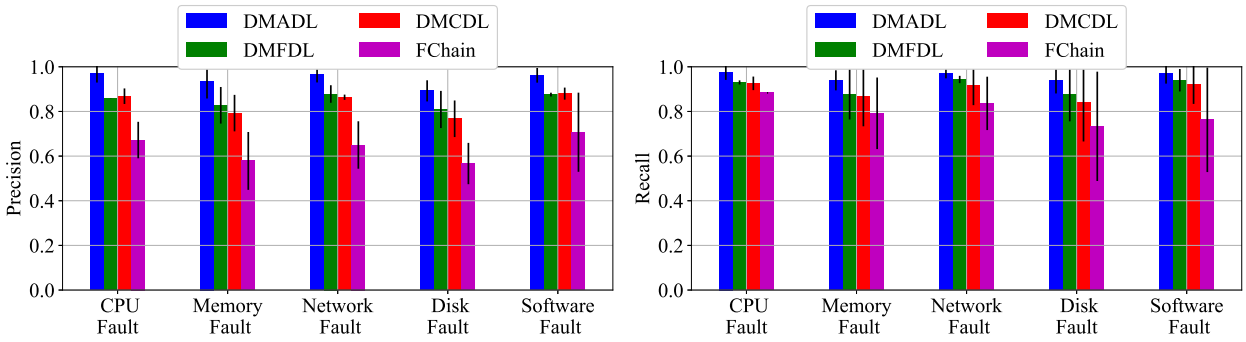
Figure 9.14: The performance of different methods for anomaly detection in Cloud applications

obvious for memory-related and disk-related faults. The component performance starts to fluctuate, and it also affects the input and output flow a lot. It is hard to tell whether it is caused by fluctuating workload or service performance. In this case, the correlation varies a lot. It is hard to determine a single threshold to achieve good detection performance. The high recall is because when there is disk or memory faults, other resource usage metrics could still show high correlation with user-space CPU utilization.

9.3 Overall Anomaly Localization Performance

Figure 9.15 shows the overall performance of our dependency model-based methods with FChain. The error bar shows the 95% confidence interval. DMADL has the best localization performance and recall. It shows the estimated mean response time and the dependency model accurately models the performance at each component. The component impact reflects how the faulty component impacts the performance at its upstream components. The DMCDL and DMFDL use the relationship between input request flow and output request flow to dependent components and the relationship between output response flow and the input response flow from dependent components to characterize the interaction between the local component and its dependent components together with the dependency model. FChain compares the

chronological changing order of different components and follows the dependency path to pinpoint the faulty component as the one which shows the earliest change. It pinpoints all components that do not have dependency path. We show that the anomaly could propagate to other components that does not have dependency path but involve in the local dependency model.



(a) The localization precision

(b) The localization recall

Figure 9.15: The performance of different methods for anomaly localization in Cloud applications

9.4 Discussion

We evaluate our non-intrusive anomaly detection methods through three different classes of chronic faults in common software systems: memory, concurrency, and semantic faults. The results show our methods can detect most of these faults. Our methods have low monitoring overhead of less than 0.3% CPU usage and negligible memory cost.

For most injected chronic software faults, DMADL could almost detect them with a very small latency. DMFDL and DMCDL have a larger detection latency because the flow ratio and correlation start to have abnormal changes only the response time exceeds a critical threshold. DMCDL has the largest delay as it uses a correlation window that containing many samples. The flow correlation decreases obviously only after there are enough abnormal samples.

Many chronic concurrency faults in Apache, Nginx and MySQL cause the server to fail to send responses or increase the server response time. The DMADL, DMFDL, and DMCDL method detects these faults with almost 100% accuracy. Some chronic concurrency faults are not detected as they either have no effect on response times (sending incorrect data to clients) or do not change the size of responses. All our methods fail to detect chronic semantic faults that cause incorrect response data. They can be detected through intrusive log analysis or feedback from clients.

Chapter 10

Conclusion

We develop a decentralized application of non-intrusive methods for performance anomaly detection and localization in Cloud applications. At each component, the service response time is a key metric representative of service performance. But it is difficult to obtain the service response time without intrusive logging operations. Many existing methods infer the service performance through resource utilization metrics. We propose to characterize the performance of each component from two perspectives: (1) its interaction behavior with other components; (2) the responsiveness of the component service. We propose DMADL to estimate the service response time using the data packets of incoming requests and outgoing responses. For services whose response times are difficult to estimate, we propose DMFDL and DMCDL to model a behavior that the response flow always follows the request flow at each component using the flow ratio and the flow correlation. Our non-intrusive methods incorporate the dependency model to characterize the interaction behavior among components. The experimental results show the effectiveness of our methods in anomaly detection and localization of faulty components in Cloud applications.

Our agents run the localization analysis for each component using the local dependency model. Compared to methods that use the global application topology, we break the localization problem into subproblems, and each agent solves in a distributed fashion. Our methods are effective and have lower complexity because of two main reasons. First, each agent only considers the local interaction behavior between the local component and its dependent components. The faulty propagation from other components does not change the local interaction behavior when all components involved in the local dependency model are normal. This insight helps us filter out those normal components quickly and reduces computation in localization of faulty components in Cloud applications. Earlier methods based on the resource utilization cannot process locally and in distributed fashion as the fault propagation changes the resource usage of all components in the application. Second, our methods analyze how a faulty com-

ponent affect the local interaction behavior under different dependency primitives. With this information, we avoid isolated analysis of each component. Existing methods do not consider the dependency primitive among components in anomaly detection. We find that a faulty dependent component affects the resource usage and behavior of other components. Tracking resource utilization without the consideration of dependent components leads to either false positive or false negative in anomaly detection. The dependency model helps us in localization of faulty components.

We show the advantages and disadvantages of our proposed methods. All our methods rely on the traffic flows for inferring service performance. The random behavior of each individual request has a negative impact on the performance of our methods. Under high workload intensities, the behaviors of individual requests are multiplexed together, and our methods are able to characterize the performance of most requests. Under low workload, the characterization of the behavior is challenging due to the randomness of individual requests. Therefore, our methods behave better under medium to high workloads.

DMADL works well in most cases as the estimated response time characterizes how fast the component responds to users' requests. It assumes that the mean response time do not change abruptly in any component. For components whose mean response time varies significantly in normal operation, DMADL cannot be used for detecting anomalies. DMADL detects most of faults that slows down the service operation. But for faults that do not necessarily cause unacceptable response time, such as partial content fault, DMADL does not work well. In DMADL, the mean response time of a component incorporates the subsystem response time. It is difficult to separate the fault propagation impact from subsystem components. When two interacting components have faults at the same time, DMADL cannot locate both of them at the same time as DMADL prefers to choose the one with higher impact in the response time.

DMFDL and DMCDL perform well when the response flow follows the request flow within a specified sampling interval. The sampling interval is critical to the performance of DMCDL and DMFDL. DMFDL does not work well for components with a very small response time. The sampling interval should also be small to properly model the service performance as specified in DMFDL and DMCDL. DMFDL uses the flow ratio between the output response flow and the input request flow. A smaller interval causes less request data packets to be considered in each interval. The flow ratio suffers the effect of randomness in individual requests and cannot represent the service performance. In this case, DMCDL performs better as it correlates all samples in a window to see their performance. For a component with small service response time, DMADL and DMCDL performs significantly better than DMFDL.

DMFDL and DMCDL do not capture well the behavior of components at which the ratio between the number of response data packets and the number of request data packets varies significantly. For example, when a component provides both uploading and downloading service of large files, the output

response flow does not always change with the input request flow. In this case, neither the flow ratio nor the flow correlation is meaningful to reflect the service performance. DMADL method is preferred and performs much better for those components. Based on the advantages and disadvantages of our individual method, an alternative method for automating the model selection process could be developed to improve anomaly detection and localization of faulty components. Our methods allow understanding the behavior of each component in Cloud environments. Many Cloud applications are developed using microservices. Multiple applications can access the same microservice. Cloud operators are managing extremely large hybrid Cloud environment. The performance of a Cloud application is closely related to other applications. It is no longer effective to view and analyze problems in each application separately and independently. Our work provides insights for a performance management framework where the behavior of individual components is modeled by the local dependency model.

The main aim of this work is to explore non-intrusive methods for performance anomaly detection and localization in IaaS Cloud applications. Although we conduct extensive experiments with several different Cloud applications, there are still some limitations and further work.

The adaptive learning process is an important part of our methods to model the behavior of components in normal operation. For each component, the multiplexing of different types of requests is an important factor in the performance of our methods. This is true in most cases except for some special occasions when users have identical behavior. For example, those online shopping websites may suffer burst workloads for best-discounted products during holidays or discounting seasons. Each component in the application process all requests of a single type. The behavior of components can be completely different from their behavior in the past. The learned model cannot recognize the behavior of components and make false decisions. To solve this problem, our methods can learn the pattern of the system during special periods and consider the pattern as an exception for anomaly detection and localization.

All our experiments run in a clean and well-controlled environment. The virtual machines are running on separate physical hosts. In production situations, multiple virtual machines may run on the same physical host. The performance of virtual machines may interfere with each other because of resource contention. The resource contention among virtual machines should be considered to deploy in a more realistic environment. We also do not consider the security perspective of proposed methods.

We do not evaluate our dependency extraction algorithm using situations where a component has different dependency primitives with different subsets of dependent components. Although our method still works, the time complexity of the dependency extraction process increases quadratically with the number of dependent components. The process has up to $O(N^2)$ time complexity where N is the number of dependent components as the algorithm is for each pair of dependent components.

Bibliography

[1] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of International Conference on Dependable Systems and Networks*, pp. 595-604, 2002. 14

[2] Y. Fu, A. Vahdat, L. Cherkasova, and W. Tang, "EtE: Passive end-to-end internet service performance monitoring," in *USENIX Annual Technical Conference*, pp. 115-130, 2002. 8

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: online modelling and performance-aware systems," in *Proceedings of the 9th conference on Hot Topics in Operating Systems*, pp. 15-15, 2003. 14

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM symposium on Operating systems principles*, pp. 74-89, 2003. 14

[5] D. P. Olshefski, J. Nieh, and E. Nahum, "ksniffer: determining the remote client perceived response time from live packet streams," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, pp. 333-346, 2004. 8, 9

[6] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier Internet services and its applications," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS)*, pp. 291-302, 2005. 9

[7] S. Lu, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, 2005. 22

[8] M. Grottke and K. S. Trivedi, "A classification of software faults," in *Journal of Reliability Engineering Association of Japan*, pp. 425-438, 2005.

[9] D. Olshefski and J. Nieh, "Understanding the management of client perceived response time," in *ACM SIGMETRICS Performance Evaluation Review*, pp. 240-251, 2006. 8, 9

[10] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: Black-box performance debugging for wide-area systems," in *Proceedings of 15th International World Wide Web Conference*, pp. 347-356, 2006. 14

[11] J. Wei and C. Xu, "sMonitor: A non-intrusive client-perceived end-to-end performance monitor of secured internet services," in *USENIX Annual Technical Conference*, pp. 243-248, 2006. 8, 9

[12] G. Jiang, H. Chen, and K. Yoshihira, "Modeling and tracking of transaction flow dynamics for fault detection in complex systems," in *IEEE Transactions on Dependable and Secure Computing*, pp. 312-326, 2006. 20

[13] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 25-33, 2006. 20, 22

[14] N. Etemadi, "Convergence of weighted averages of random variables revisited," in *Proceedings of the American Mathematical Society*, pp. 2739-2744, 2006. 76

[15] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 13-24, 2007. 15

[16] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier Applications", in *Proceedings of International Conference on Autonomic Computing (ICAC)*, pp. 11-15, 2007. 9

[17] S. Pertet, R. Gandhi, and P. Narasimhan, "Fingerpointing correlated failures in replicated systems," in *USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, pp. 1-6, 2007. 20

[18] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, "Constellation: automated discovery of service and host dependencies in networked systems," in *MSR-TR-2008-67 TechReport*, 2008. 16

[19] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: experiences, limitations, and new solutions," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 117-130, 2008. 16

[20] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Automated anomaly detection and performance modeling of enterprise applications," in *ACM Transactions on Computer Systems*, pp. 1-32, 2009. 10

[21] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *ACM SIGCOMM Computer Communication Review*, pp. 243-254, 2009. 17

[22] J. Terrell, K. Jeffay, F. D. Smith, J. Gogan, and J. Keller, "Passive, streaming inference of TCP connection structure for network server management," in *Proceedings of the First International Workshop on Traffic Monitoring and Analysis (TMA)*, pp. 42-53, 2009. 8, 9

[23] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009. 9

[24] Q. Fu, J. G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Nineth IEEE International Conference on Data Mining (ICDM)*, pp. 149-158, 2009. 9

[25] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in Cloud data center," in *IEEE International Conference on Cloud Computing (ICAC)*, pp. 370-377, 2010. 10

[26] J. P. Magalhaes and L. M. Silva, "Detection of performance anomalies in web-based applications," in *Proceedings of the 9th IEEE International Symposiumon Network Computing and Applications*, pp. 60-67, 2010. 12, 20

[27] C. Wang, V. Talwar, K. Schwan and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *IEEE Network Operations and Management Symposium*, pp. 96-103, 2010. 13, 20

[28] H. Kang, H. Chen, and G. Jiang, "PeerWatch: a fault detection and diagnosis tool for virtualized consolidation systems," in *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*, pp. 119-128, 2010. 12, 20

[29] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, D.A. Patterson, "Rain: A workload generation toolkit for Cloud Computing applications," in *Technical Report UCB/EECS-2010-14*, 2010. 26

[30] H. Nguyen, Y. Tan, and X. Gu, "PAL: Propagation-aware Anomaly Localization for cloud hosted distributed applications," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML)*, pp. 1-8, 2011. 17

[31] D. J. Dean, H. Nguyen, and X. Gu, "UBL: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pp. 191-200, 2012. 11

[32] J. Tan, S. Kavulya, R. Gandhi, "Light-weight black-box failure detection for distributed systems," in *Proceedings of the workshop on Management of Big Data Systems*, pp.13-18, 2012. 12, 20, 60, 92, 122

[33] H. Kang, X. Zhu, and J. L. Wong, "DAPA: diagnosing application performance anomalies for virtualized infrastructures," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, pp. 8-8, 2012. 20

[34] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani and D. Rajan, "PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems," in *IEEE 32nd International Conference on Distributed Computing Systems*, pp. 285-294, 2012. 20

[35] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 37-48, 2012. 25, 26

[36] F. Doelitzscher, M. Knahl, C. Reich and N. Clarke, "Anomaly detection in IaaS Clouds," in *IEEE 5th International Conference on Cloud Computing Technology and Science*, pp. 387-394, 2013. 11

[37] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem determination and diagnosis in shared dynamic clouds," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1-12, 2013. 12

[38] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "FChain: Toward black-box online fault localization for cloud systems," in *IEEE 33nd International Conference Distributed Computing Systems*, pp. 21-30, 2013. 17, 20, 29

[39] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: end-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pp. 217-231, 2014. 34

[40] P. Chen, Y. Qi, P. Zheng, and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE Conference on Computer Communications*, pp. 1887-1895, 2014. 16

[41] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot and C. Konig, "PAD: performance anomaly detection in multi-server distributed systems," in *IEEE 7th International Conference on Cloud Computing*, pp. 769-776, 2014. 12

[42] W. Wu, K. He, and A. Akella. "PerfSight: performance diagnosis for software dataplanes," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, pp. 409-421, 2015. 24

[43] A. Sangpetch and H. S. Kim, "VDEP: VM dependency discovery in multi-tier Cloud applications," in *IEEE 8th International Conference on Cloud Computing (CLOUD)*, pp. 694-701, 2015. 2, 16, 32, 33

[44] J. Kim and H. S. Kim, "PBAD: perception-based anomaly detection system for Cloud Data Centers," in *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 678-685, 2015. 8, 9

[45] S. Fu, H. Kim, and R. Prior, "FlowBox: Anomaly Detection Using Flow Analysis in Cloud Applications," in *IEEE Global Communications Conference (GLOBECOM)*, pp. 1-6, 2015. 60, 75, 92, 122

[46] S. Fu, H. Kim, and R. Prior, "FSAD: flow similarity analysis for anomaly detection in Cloud applications," in *IEEE 7th International Conference on Cloud Computing Technology and Science*, pp. 426-429, 2015.

[47] S. Barbhuiya, Z. Papazachos, P. Kilpatrick, and D. S. Nikolopoulos, "A lightweight tool for anomaly detection in Cloud data centres," in *Proceedings of 5th International Conference on Cloud Computing and Services Science*, pp. 343-351, 2015. 13, 20

[48] T. Wang, W. Zhang, J. Wei and H. Zhong, "Fault detection for cloud computing systems with correlation analysis," in *IFIP/IEEE International Symposium on Integrated Network Management*, pp. 652-658, 2015. 13, 20

[49] T. Palit, Y. Shen, and M. Ferdman, "Demystifying Cloud Benchmarking," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 122-132, 2016. 25, 27

[50] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "PerfCompass: online performance anomaly fault localization and inference in Infrastructure-as-a-Service Clouds," in *IEEE Transactions on Parallel and Distributed Systems*, pp. 1742-1755, 2016. 14, 20

[51] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for Cloud-hosted Web applications," in *Proceedings of International Conference on World Wide Web*, pp. 469-478, 2017. 14

[52] A. Mdhaffar, R. B. Halima, M. Jmaiel, and B. Freisleben, "Reactive performance monitoring of Cloud computing environments," in *Cluster Computing*, pp. 2465-2477, 2017. 15

[53] H. Zhang, S. Chen, J. Liu, Z. Zhou, and T. Wu, "An incremental anomaly detection model for virtual machines," in *PLoS ONE*, 2017. 11

[54] K. Wang and H. S. Kim, "PCAD: Cloud Performance Anomaly Detection with Data Packet Counts," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 106-113, 2017.

[55] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri, and G. D. S. Silvestre, "Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned", in *The Journal of Systems and Software*, pp. 84-106, 2018. 11

[56] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson, "Deepview: virtual disk failure diagnosis and pattern detection for Azure", in *15th USENIX Symposium on Networked Systems Design and Implementation*, pp. 519-532, 2018. 18