MonArch: Scalable Monitoring and Analytics for Visibility and Insights in Virtualized Heterogeneous Cloud Infrastructure

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering

UNIVERSITY OF TORONTO

Jieyu Lin

Department of Electrical & Computer Engineering

University of Toronto

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science

©Copyright by Jieyu Lin 2015
Abstract

MonArch: Scalable Monitoring and Analytics for Visibility and Insights in Virtualized Heterogeneous Cloud Infrastructure

Jieyu Lin
Master of Applied Science
Department of Electrical and Computer Engineering
University of Toronto
2015

With rapid development of cloud computing technologies, cloud data centers continue to grow in scale and complexity. Monitoring and analytics are the key enablers to provide visibility and insights in a large-scale infrastructure.

In this thesis, we present a monitoring and analytics system, called MonArch, designed for monitoring data collection, storage and analytics in a virtualized heterogeneous cloud environment. MonArch provides high scalability and flexibility to enable multi-layer heterogeneous resources monitoring. Scalable anomaly detection, root cause analysis, and graph processing capabilities are offered in MonArch to simplify resource monitoring and support real-time identification of system anomalies. MonArch supports a wide variety of use cases including security detection, system diagnosis, resource allocation, and IoT sensor data storage and processing. MonArch is implemented and deployed in the SAVI Testbed. The evaluation results demonstrate MonArch’s scalability and efficiency in data monitoring, as well as its effectiveness in anomaly detection and root cause analysis.
To mum, dad, Caroline,

and the memory of my grandma
Acknowledgements

I would like to first express my most sincere appreciation and gratitude to my supervisor, Professor Alberto Leon-Garcia for his guidance, support and understanding during the course of my thesis work. I am grateful for the amount of trust and inspiration from him that led my learning, exploration, and completion of my thesis. I learned so much from Professor Leon-Garcia not only academically, but also in many other aspects of life.

I would like to thank the members of my committee: Professor Ben Liang, Professor Elvino Sousa, and Professor Cristiana Amza for their evaluation of my work and valuable comments.

I am thankful to the SAVI Testbed architect, Hadi Bannazadeh for the valuable discussion and help throughout my study. I started working with Hadi in the last year of my undergraduate study and I appreciate the knowledge I have learned from him since then.

I would also like to thank the SAVI colleagues Qi Zhang, Byungchul Park, Thomas Lin, Elliot Joon-Myung Kang, Petros Spachos, Hesam Rahimi, and Rajsimman Ravichandiran for their collaboration and feedback. I would like to especially thank Qi and Byungchul for teaching me many aspects of academic research and providing invaluable comments and feedback for my work. I would also like to thank the other colleagues and friends Weiwei Li, Sai Zhang for their support and encouragement.

I am also thankful for the assistance and administrative support from staff member Vladimirio Cirillo.

Last but not least, I would like to thank my mum and dad for the unconditional love and support throughout all my studies. My thesis would not have been possible without them. I would also like to thank Caroline for her love, support, and encouragement during the time of my thesis work.
# Contents

List of Figures ix

List of Tables xi

## Chapter 1 Introduction

1.1 Vision & Motivation ........................................... 1
1.2 Research Goal & Challenges ................................. 2
1.3 Thesis Structure .............................................. 4

## Chapter 2 Background and Requirement Analysis

2.1 Software-Defined Infrastructure & SAVI Testbed ............... 6
  2.1.1 SDI Architecture .......................................... 6
  2.1.2 SAVI and SAVI Testbed .................................... 9
2.2 Monitoring and Analytics System Requirements .................. 10
  2.2.1 Requirements for the Base Monitoring and Analytics System ... 10
  2.2.2 Requirements for Anomaly Detection ........................ 13

## Chapter 3 Related Work

3.1 Monitoring System ............................................. 15
3.2 Analytics System .............................................. 17
  3.2.1 Monitoring System with Analytics Capability ............... 17
  3.2.2 Large-Scale Processing and Analytics Engine ............... 18
  3.2.3 Anomaly Detection ........................................ 19

## Chapter 4 MonArch System

4.1 Overview of Monitoring and Analytics System .................. 20
4.2 Architecture .................................................. 23
  4.2.1 Acquisition Layer ........................................ 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.5 Anomaly Detection &amp; Root Cause Analysis Implementation</td>
<td>54</td>
</tr>
<tr>
<td>5.2.5.1 Offline Entity Behavior Profiling</td>
<td>55</td>
</tr>
<tr>
<td>5.2.5.2 Online Anomaly Detection</td>
<td>56</td>
</tr>
<tr>
<td>5.2.5.3 Root Cause Analysis</td>
<td>58</td>
</tr>
<tr>
<td>5.3 MonArch Use Cases</td>
<td>58</td>
</tr>
<tr>
<td>5.3.1 Security Detection</td>
<td>58</td>
</tr>
<tr>
<td>5.3.2 System Diagnosis</td>
<td>60</td>
</tr>
<tr>
<td>5.3.3 Resource Allocation</td>
<td>61</td>
</tr>
<tr>
<td>5.3.4 Smart Room Monitoring</td>
<td>61</td>
</tr>
<tr>
<td>5.4 MonArch Portal</td>
<td>62</td>
</tr>
<tr>
<td>Chapter 6 Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>6.1 System Performance Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>6.1.1 Collection and Analytics Scalability</td>
<td>64</td>
</tr>
<tr>
<td>6.1.1.1 Monitoring Data Collection and Storage</td>
<td>64</td>
</tr>
<tr>
<td>6.1.1.2 Data Processing - Graph Construction</td>
<td>65</td>
</tr>
<tr>
<td>6.1.2 Overhead Analysis</td>
<td>69</td>
</tr>
<tr>
<td>6.2 Anomaly Detection and Root Cause Analysis Evaluation</td>
<td>70</td>
</tr>
<tr>
<td>6.2.1 DoS Attack and Resource Contention</td>
<td>72</td>
</tr>
<tr>
<td>6.2.2 Unexpected Access</td>
<td>80</td>
</tr>
<tr>
<td>Chapter 7 Future Work and Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>7.1 Future Work</td>
<td>82</td>
</tr>
<tr>
<td>7.2 Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>References</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Figures

2.1 SDI Architecture ......................................................... 7
2.2 System Architecture .................................................. 9
2.3 SAVI Architecture ..................................................... 10

4.1 Collection, Storage, Analytics Relationships ....................... 21
4.2 MonArch Architecture ................................................ 24
4.3 General Agent Internal ............................................... 25
4.4 MonArch Implementation .............................................. 32
4.5 User Agent Web GUI Screenshot ..................................... 39

5.1 Cloud Infrastructure Resource Graph Representation in MonArch .... 43
5.2 Motivating Example for Anomaly Detection ......................... 46
5.3 System Architecture .................................................. 47
5.4 Anomaly Propagation Paths Example ................................ 51
5.5 Special Cases Examples .............................................. 54
5.6 Anomaly and Root Cause Analysis Implementation in MonArch .... 55
5.7 Special Cases Examples .............................................. 59
5.8 MonArch Portal Home Page .......................................... 62
5.9 MonArch Portal Multi-layer Graph Visualization .................. 63

6.1 Scalability of Data Collection Throughput .......................... 66
6.2 Scalability of Graph Construction Processing ...................... 67
6.3 Parallelism of Graph Construction Processing ..................... 68
6.4 DoS Attack and Resource Contention Experiment Setup ............ 72
6.5 CPU and Network Utilization of VMs Belonging to the Two Applications . 74
6.6 Network Utilization of Physical Machines .......................... 75
6.7 Attack Time Period and Real-Time Anomaly Detection Results ........ 76
6.8 Relationship Graph Constructed with Anomalous Entities .......... 78
LIST OF FIGURES

6.9 Propagation Graph Constructed by Root Cause Anomaly Algorithm . . . . 79
6.10 Attack Time Period and Real Time Anomaly Detection Results . . . . . 80
List of Tables

4.1 OpenFlow Agent Monitoring Capabilities .................................. 35
4.2 NetFlow Formatting Method .................................................. 37
4.3 Physical Server Agent Monitoring Capabilities ......................... 38

5.1 Profiling Metrics for Resource Entities ................................. 57
5.2 Anomaly Detection Decision Table ....................................... 58

6.1 Agent Average CPU Utilization and Overhead ....................... 70
6.2 Agent Average Memory Utilization and Overhead .................... 71
6.3 Agent Average Network Receive/Transmit Bandwidth Utilization and Overhead ............................................. 71
6.4 DoS and Resource Contention Use Case Anomaly Detection Hints ........... 77
6.5 DoS and Resource Contention Use Case Root Cause Analysis Propagation Graph Ranking .................................................. 78
6.6 Unexpected Access Use Case Anomaly Detection Hints ............. 81
Chapter 1

Introduction

IT infrastructure has been evolving rapidly in the last two decades fuelled by continuous improvement of computing hardware and development of new generation technologies such as virtualization, smart mobile devices, and sensors. As never before, a wide variety of applications are being developed and deployed to bring more values to people’s lives. Data centers containing hundreds of thousands of computing servers are being built worldwide to support the scale and demand of these applications.

1.1 Vision & Motivation

Cloud computing infrastructure has become virtualized, heterogeneous, and multi-tiered. Virtualization is a powerful technique that enables the partitioning of physical resources (e.g. server and switches) into one or more virtual resources, each of which can be assigned to applications in a flexible and efficient manner. With rapid development of cloud computing and virtualization technologies, cloud data centers today support the virtualization of a large variety of components ranging from physical servers [33], switches, routers [11, 16, 43], middleboxes [41], wireless access points [73], and GPUs, to programmable hardware such as FPGAs [36]. To improve latency and content delivery, companies and researchers are building data centers that are geographically distributed with high bandwidth interconnectivities. This creates a multi-tiered cloud environment. An example of a multi-tiered cloud infrastructure is the Smart Application on Virtual Infrastructure (SAVI) Testbed, which has a three-tiered architecture: core, Smart Edges, and virtual Customer Premise Edge (vCPE)/sensors.

The heterogeneity of the cloud environment provides more functionalities and flexibilities for applications, but at the same time it brings more complexity for infrastructure management. Current cloud management systems often use separate management tools for different virtualized resources, resulting in high management and maintenance overhead. To address this limitation, we have introduced the concept of Software-Defined Infrastructure
1. INTRODUCTION

(SDI) that provides integrated management of virtualized resources [47]. The benefit of SDI lies in its capability of deploying complex virtual infrastructures, while facilitating efficient control and management of cloud resources for supporting diverse needs (e.g. performance, reliability and security) of today’s cloud applications.

With the growing scale and complexity of data centers and novel computing devices such as smart sensors and mobile phones, the traditional ways of infrastructure and application management by humans are no longer effective or cost efficient. Thus intelligent management is crucial for today’s IT Infrastructures and applications. One of the ultimate forms of intelligent management is autonomous computing, which refers to self-managing IT infrastructures that can adapt to unpredictable changes with minimum human intervention. In order for the infrastructure to make informed decisions for self-management purposes, infrastructure visibility and self-awareness is essential. Monitoring and analytics are the main enablers for providing visibility and insights in a large-scale cloud infrastructure. With the wide variety of monitoring targets and the high data generation rate, a large amount of infrastructure state information needs to be collected, monitored, and properly processed to simplify infrastructure monitoring by administrators.

Furthermore, instead of having administrators spending large amounts of time on identifying problems, performing diagnoses and reacting to unexpected events, the monitoring and analytics system should be able to automatically identify system anomalies/problems and suggest root causes. In particular, anomaly detection is a major aspect of cloud monitoring that aims at identifying abnormal network and system behaviors such as operator errors, hardware or software failures, and resource over- and under-provisioning. Given the ever-increasing scale coupled with the high complexity of software, applications, and workload patterns, anomaly detection methods must operate automatically at run time and without the need for prior knowledge about normal or anomalous behaviors. Moreover, the analytics algorithms should be sufficiently general to apply to multiple levels of abstractions and subsystems for the different metrics and data formats (e.g. both numerical and text-based log data) used in large-scale systems.

1.2 Research Goal & Challenges

The previous section briefly described the importance of monitoring and analytics for providing visibility and insights in today’s cloud infrastructure. We will formalize the research goal in this section and identify the research challenges.
The goal of this thesis is to investigate, design, implement, and deploy a monitoring and analytics system that is capable of creating visibility and insights in a heterogeneous, virtualized, and multi-tiered cloud infrastructure. To achieve this goal, the monitoring and analytics system should contain three main basic functionalities: collection, storage, and analytics of monitoring data. This novel system is different from many of the existing monitoring systems, so the requirements are studied and defined based on the objectives of today’s cloud infrastructure. Virtualized cloud infrastructure often contains three layers: physical, virtual, and application layers. In general, applications run top of virtual machines and virtual networks that are hosted on the physical servers and switches. The monitoring and analytics system should collect and store data from resources in all layers. In terms of creating visibility and extracting insights, we believe the monitoring and analytics system should perform analytics on the collected data to 1) provide a view of the resources in the infrastructure to understand the state of infrastructure; and 2) learn about the system’s normal behaviors in order to detect anomalies and provide hints to assist identification of root causes. The objective here is to develop generic (non-application-specific) analytics techniques that can be used by the tenants and administrators in a shared cloud infrastructure. The infrastructure insights generated by the system can be used for many use cases such as security detection, diagnosis, and resource allocation. To support interoperability, the monitoring and analytics system should be designed to target the generic cloud environment. For evaluation purposes, this system is deployed on the SAVI Testbed.

Next we discuss a number of challenges when designing and implementing the monitoring and analytics system discussed above.

**Size of monitoring data in cloud infrastructure:** With the size of modern cloud infrastructure, the size of monitoring data generated is very large. Hence, scalability of the monitoring and analytics system is crucial. In a data center, there are large amounts of data available to be collected for monitoring purposes. Some of these data are generated automatically by the infrastructure such as logs (e.g. system log, IP table logs, services log, etc), and infrastructure information (e.g. configuration files, access control rules, VM ownership information, etc). Other types of data need to be sampled and collected such as CPU utilization, disk IO, and network bandwidth utilization. Furthermore, there are multiple layers in today’s cloud infrastructure: physical layer, virtual layer, and application layer. Resources in each of these layers constantly generate monitoring data. To estimate the data size, we can make a back of the envelope calculation. Assume a data center has 10,000 physical machines and 1000 switches, and each physical server can host 15 virtual machines. Each
machine has five metrics (e.g. CPU utilization, Network IO, Disk IO, memory) and we monitor each of them every minute, with the average size of each message being 100 bytes. Also in each server, there is a 100 KB of log generated every minute. In each switch, there are 100 KB of data generated every minute (for example, using NetFlow or OpenFlow). If we collect monitoring data for six months, the total monitoring data size is: 3.81 PetaBytes. Clearly, scalability for data collection, storage and processing is one of the challenges for this research.

**Generic analytics algorithms for wide range of applications with high dynamism:** Cloud is a multi-tenant shared environment in which many different kinds of applications are operating. These infrastructure and application behaviors can change over time due to many reasons, including user demand, new feature releases, and software updates. Therefore, it is challenging to design generic anomaly and root cause analysis algorithms that can operate for a wide range of applications without sacrificing accuracy. When collecting monitoring data, monitoring metrics are similar in the physical layer and virtual layer (e.g. CPU utilization, network bandwidth, and memory). However, monitoring metrics in the application layer can vary drastically between applications, and it is not realistic to cover all the possibilities. Moreover, due to privacy reasons, some application owners may not want to expose the monitoring data to the infrastructure. As a result, the analytics algorithms need to function with limited or no information from the application layer.

**Coverage of monitoring data collected and resulting overhead:** Visibility to the infrastructure is limited by the amount of information contained in the monitoring data, so it is important to cover as much information as possible. However, as we discussed in the previous challenge, a cloud infrastructure contains a large number of monitoring data sources and data collection would introduce overhead, so another challenge is to maximize the coverage of monitoring data collected and minimize the system overhead.

### 1.3 Thesis Structure

This thesis is organized according to the following structure:

**Background and Requirement Analysis:** In this chapter, background information related to SDI and SAVI Testbed is provided. Based on this background, we analyze and define the requirements for the monitoring and analytics system.

**Related Work:** Work related to this thesis is presented in this chapter. The chapter is organized into two sections, focusing on the monitoring part of the system, and the analytics
part of the system.

**MonArch System**: In this chapter, we describe the MonArch monitoring and analytics system and how it addresses the challenges and meets the specified requirements. Architecture and implementation are the two main sections in this chapter that discuss details of the design and implementation of the MonArch system.

**Analytics and Intelligence in MonArch**: This chapter focuses on the analytics and intelligence part of the MonArch system. We start by discussing graph processing, anomaly detection and root cause analysis in the MonArch system, followed by the use cases of these analytic techniques to demonstrate the effectiveness of the MonArch system. Then we present the MonArch’s web portal used for visualization.

**Performance Evaluation & Discussion**: The evaluation results and discussion of the MonArch system are presented in this chapter. The first section provides results to show the system’s performance and scalability for monitoring data collection and graph processing. Next, the anomaly detection and root cause analysis algorithms are evaluated through some use cases. This chapter is followed by the conclusion chapter that summarizes this thesis and discusses future work.
Chapter 2

Background and Requirement Analysis

In this chapter, we describe the background of this thesis and analyze and define the requirements for the monitoring and analytics system. This chapter is organized into two sections. The first section describes the SDI concept and the SAVI Testbed. Then, based on the background and challenges, we define the requirements for the monitoring and analytics system.

2.1 Software-Defined Infrastructure & SAVI Testbed

Current control and management systems mainly focus on having separated controllers for difference resources. For example, in cloud environment, compute and network resources often have separate controllers. The cloud controller in OpenStack [10] is responsible for managing virtual machines and storage, whereas the network is often managed by OpenFlow [56] or other SDN controllers. This approach is not ideal for best decision making and optimization of performance and cost. In order to make informed decisions in a heterogeneous cloud environment, it is important to have a global view of all the resources and manage them in an integrated way. Although there could be communication between controllers to facilitate this kind of decision making, it often introduces high management and maintenance overhead. To address this limitation, we have proposed SDI, which integrates the management of different resources into a logically centralized point to provide more flexibility and intelligence.

2.1.1 SDI Architecture

In this section, we briefly introduce the architecture of SDI. Figure 2.1 shows the high-level architecture of the SDI Resource Management System (RMS).

In a virtualized and heterogeneous infrastructure, we have various types of resources such as compute, network, storage, programmable hardware (FPGA), and etc. To allow resource sharing and to improve utilization, typically physical resources are virtualized, and thus we
Figure 2.1: SDI Architecture - Software-Defined Infrastructure Architecture for Integrated Resource Management
have both physical and virtual resources in the infrastructure. In Figure 2.1, different types of resources are indicated by different shapes (square, triangle, circle). Physical resources are denoted by shapes with a solid border line, and virtual resources are indicated by a dotted border line.

In this architecture, a type-specific resource controller directly controls each type of resource. As shown in Figure 2.1, resource controllers A, B, and C directly connect to and control resources A, B, and C respectively. A resource controller can be any traditional controller that is designed to manage a specific kind of resource. The use of these resource controllers allows us to leverage results from other research and projects.

Different types of resources may have interconnections or dependencies between each other, which gives rise to a need for a topology manager. The topology manager is responsible for discovering the relationship and interconnections between different resources.

A monitoring and analytics manager is used to monitor the heterogeneous resources, and extract useful knowledge and insights from the monitoring data to assist infrastructure management. This module corresponds to the monitoring and analytics system that is the focus of this thesis. The monitoring and analytics manager should provide monitoring data collection, storage, and analytics functionalities. Different from the topology manager, which may only store the latest resource topology, the monitoring and analytics manager keeps records of the historical monitoring data. From the historical data, the monitoring and analytics manager can potentially also construct a historical view of how the topology of the infrastructure has changed in the past. The monitoring and measurement manager and the topology manager together provide a multi-dimensional view of the infrastructure that includes spatial (topology), and temporal (resource states and information over time) information.

The SDI manager is the decision-making point in this architecture. It is responsible for performing integrated resource management for converged heterogeneous resources. The SDI manager obtains the resource topology information from the topology manager, and the resource state and analytics results from the monitoring and analytics manager. Based on this information about the infrastructure, the SDI manager launches the integrated management algorithms and communicates with the resources controllers to materialize various integrated resource management functionalities. Examples of integrated resource management functions include but are not limited to: resource allocation and migration, real-time diagnosis, performance optimization, green networking, and fault tolerance.

The SDI manager, topology manager, and monitoring and analytics manager have open APIs for external entities. The open APIs allow external entities to leverage the capabilities
2.1 Software-Defined Infrastructure & SAVI Testbed

2.1.2 SAVI and SAVI Testbed

The Smart Applications on Virtual Infrastructure (SAVI) is a project with the goal to investigate future application platforms. The SAVI Testbed is an implementation of the SDI concept and has been operational since July 2013. The SAVI Testbed has a three-tiered architecture: core, Smart Edge and vCPE/sensors, as shown in Fig 2.2. Core is similar to a traditional massive-scale data center that generally resides geographically far away from end users. Smart Edges are mid-size data centers that are located closer to users, designed to support low latency applications and content delivery. Besides traditional computing and networking resources, Smart Edges also provide resources such as reconfigurable hardware, GPU, and Software-Defined Radio (SDR) resources. vCPE and sensors are extension of the Smart Edge closer to the users for monitoring the physical world and providing extended service to the user’s site.

The SAVI Testbed currently has one core node located at the University of Toronto and seven Smart Edges located in seven Canadian universities: the University of Toronto, the University of Waterloo, York University, McGill University, Carleton University, the University of Calgary, and the University of Victoria. The Smart Edges and the core are connected mostly by ORION and CANARIE layer two optical networks.

SDI architecture is reflected in each SAVI Smart Edge node and core node. In a SAVI Smart Edge, as shown in Figure 2.3, we have compute, network, storage, FPGA, and other resources. OpenStack is used for managing compute, storage, and FPGA resources. OpenFlow controllers are used for controlling network resources such as switches. System states and information from the OpenStack, OpenFlow controllers and physical nodes are collected by the topology manager and the monitoring and analytics manager to generate system topology and create visibility. The processed information is used by the SDI manager for decision-
2. BACKGROUND AND REQUIREMENT ANALYSIS

2.2 Monitoring and Analytics System Requirements

To address the challenges and limitations of the existing systems, we investigate and determine the requirements for the monitoring and analytics system that is designed for modern cloud infrastructure. The requirements can be organized into two parts: 1) requirements for the monitoring and analytics system; 2) requirements for anomaly detection analytics on the monitoring and analytics system.

2.2.1 Requirements for the Base Monitoring and Analytics System

The following are the requirements for the base monitoring and analytics system:

1. **Integrated Monitoring and Analytics**
   The system should be capable of collecting monitoring data from converged heterogeneous resources in an infrastructure. These resources include compute, network, storage, GPU, programmable hardware, wireless access points and other resources...
2.2 Monitoring and Analytics System Requirements

that are either physical or virtual. On the analytics side, the system should be able to jointly process data from different types of resources and produce integrated results.

2. **Multi-Layer Monitoring and Analytics**
   The system should be able to monitor resources in the physical layer, the virtual layer as well as the application layer. The physical layer includes physical machines and network equipment, and virtual layer consists of virtualized resources such as VM and virtual network. The application layer contains applications or software that are operating on top of the virtual resources. One of the goals of the SAVI project is to provide adaptive support for future applications, which implies that the monitoring and analytics system should be able to monitor application level information.

3. **Scalability**
   As we discussed, one of the main challenges is scalability for handling large data size. Due to the size and variety of monitoring data in a cloud infrastructure, the monitoring and analytics system should be scalable. Scalability in this system consists of three main parts: 1) Data collection: as the amount of monitoring data in an infrastructure increases, the system should continue to collect and process the monitoring data in a timely manner; 2) Analytics: the analytics algorithms that are running on top of the monitoring system should be able to process more data by adding compute servers without significant slowing down in processing time; 3) Storage: The storage system should be able to handle more data by adding more servers.

4. **Support for Structured and Unstructured Data**
   Both structured and unstructured data should be supported in the system. That includes collecting, storing, and analyzing both structured and unstructured data. However, due to the complex nature of unstructured data, analytics methods may vary based on the use case, so we don’t set strict requirements for all the analytics methods.

5. **Data-On-Demand**
   Monitoring data and analytics results should be available on-demand to the users and the modules in the SDI manager. In the SDI architecture, each module in the SDI manager obtains resources state information and infrastructure insights from the monitoring and analytics manager in order to make informed decisions. In addition to the management modules, users of infrastructure should also have access to monitoring data and analytics results, if they have the privilege. Thus access control also needs to
2. BACKGROUND AND REQUIREMENT ANALYSIS

be enforced. This requirement also means that users do not need to specify the physical and virtual layer metrics to monitor; the system should continuously collect and store physical and virtual layer data and perform predefined analytic processing so that data is accessible to users upon request.

6. Storage of Historical Data
To support future analysis and data-on-demand, the system should store the historical monitoring data and analytics results. Historical monitoring data can be used to recreate a view of the infrastructure or applications at a certain time in the past. It can also be used to investigate the infrastructure or application behavior in order to improve operation and performance.

7. Support for Custom Metrics
Due to the nature of applications, the monitoring requirements will vary from application to application. Since it is neither realistic nor efficient to monitor every metric that all users might need, the system should provide a mechanism to allow users to monitor user-specified custom metrics.

8. Stream Processing
The system should be able to process monitoring data in real-time. As new monitoring data enters the system, some analytic tasks may require the data to be processed right away so that we can observe the behavior of the system to achieve real-time monitoring functions. Some examples of the monitoring functions include detection of anomalies, detection of violation of Service Level Agreement (SLA), and triggering of system migration. To facilitate the real-time analytics tasks, inputs of the system include both real-time data as well as offline analytics results.

9. Extensibility
As new types of resources are being added to the infrastructure, the system should provide extensibility to easily collect, store, and analyze monitoring data from the new data sources. One example is when adding new type of sensors to the infrastructure. We should be able to quickly extend the system to include the new sensors in our monitoring and analytics scope.
2.2 Monitoring and Analytics System Requirements

2.2.2 Requirements for Anomaly Detection

In order to design our anomaly detection algorithm, we must first define our design requirements.

1. Supporting Heterogeneous Measurement Inputs
   The proposed anomaly detection mechanism must be capable of analyzing and correlating input data in various formats. Traditional cloud monitoring systems such as Ceilometer [10] only capture numerical data. In practice, it has been shown that text data such as application and system logs are also effective for detecting system anomalies. Thus, our anomaly detection mechanism must be capable of analyzing and correlating heterogeneous measurement data.

2. Handling Unlabelled Measurement Data
   One of the key challenges we encounter in the design of the anomaly detection mechanism is the lack of labeled training measurement data. A measurement is labeled if it has been correctly identified as either normal or abnormal. In practice, getting labeled measurement data is an extremely difficult task, due to the large varieties of applications and components running in the cloud, as well as the intrinsic dynamism inherited in these systems. To keep our approach general, we require that the anomaly detection mechanism must be able to work with unlabeled data. This implies we must use unsupervised learning algorithms for anomaly detection.

3. No Assumption on Underlying Distributions
   Many anomaly detection algorithms assume the measurement data follow specific (e.g. Gaussian) distributions. This assumption generally does not hold for the cloud environment where workload can fluctuate significantly over time. For example, the daily demand fluctuation patterns can cause cloud applications to work in multiple (e.g. idle and busy) states. As a result, the measurement data can exhibit multi-modal distributions. To keep our approach general, we must not make any assumption about the underlying distributions of the monitoring data.

4. Scalability and Efficiency
   Given the large number of applications and components to be monitored, our anomaly detection mechanism must be highly scalable. This means the detection algorithm must incur both low space and time complexity. Many of existing anomaly / outlier
2. BACKGROUND AND REQUIREMENT ANALYSIS

detection algorithms, such as k-nearest-neighbor search (kNN) [46] must keep large volumes of historical data in memory, which makes them unsuitable for our purposes.
Chapter 3

Related Work

Based on the main functionalities of a monitoring and analytics system, the related work of this thesis is broken down into two main parts: 1) monitoring system; and 2) analytics system. The coverage of each part is as follows:

**Monitoring System:** For the monitoring system, we investigated the previous work that focuses on various aspects of monitoring for cloud and SDN. In comparison to our MonArch, none of the existing work provides sufficient flexibility and scalability in monitoring multi-layer heterogeneous resources.

**Analytics System:** To support scalable processing and analytics, we researched the existing generic and specialized large-scale data processing systems. We also reviewed the literature for research and algorithms related to performing anomaly detection in a cloud environment.

3.1 Monitoring System

Popular tools such as Nagios [21], Ganglia [55], ZABBIX [68], OpenNMS [22] are traditional IT infrastructure monitoring tools. They provide rich functionalities and support extensions for cloud monitoring. However, these tools only provide basic statistics of monitoring data and cannot satisfy today’s complex monitoring processing and analytics requirements.

To support shared infrastructure, there are research efforts and projects created to support multi-tenant monitoring. Ceilometer [10] is a telemetry component of OpenStack, an open source cloud platform. Ceilometer provides mechanisms for monitoring virtual compute, network, and storage resources. However, Ceilometer provides very limited analytics capability for monitoring data (only average, sum, max, and min). CloudView [61], MONaaS [19] are other systems that integrate with OpenStack, but they have the same limitations as Ceilometer. [66] proposed a multi-tenant monitoring and management system based on a widely-used open source network monitoring system. This system is implemented using OpenNMS.
3. RELATED WORK

For data collection, there are research results and technologies that support different aspects of compute, network, and physical server monitoring. Kernel-based Virtual Machine (KVM) [48] is a hypervisor that runs on top of Linux kernel with x86 hardware that natively supports virtualization. QEMU [34] is an open source emulated hypervisor for hosting virtual machines. Other hypervisors include XEN [33] and vSphere [30]. Libvirt [18] is an open source library that provides a common layer on top of hypervisor for management purposes. Libvirt provides APIs for monitoring CPU utilization, memory usage, disk IO and network IO of VMs. Since Libvirt supports a wide range of hypervisors, VM monitoring can be performed by using the Libvirt APIs.

In terms of network monitoring, there have been efforts with regard to Software-Defined Networking (SDN) monitoring. FlowSense [76] is a push-based SDN monitoring system for network utilization. It provides a basic approach for OpenFlow network monitoring. PayLess [39] proposed a network monitoring framework for monitoring OpenFlow networks, and it proposed a variable frequency flow-statistics collection algorithm to improve the monitoring overhead. OpenNetMon [67] provides per-flow metrics monitoring in an OpenFlow network. It focuses on three metrics: bandwidth, delay and packet loss. Besides OpenFlow, NetFlow [40] is an solution introduced by Cisco for collecting IP layer traffic statistics. NetFlow is widely supported and can also be used for network monitoring.

For physical hardware monitoring, Simple Network Management Protocol (SNMP) [37] and Intelligent Platform Management Interface (IPMI) are the two popular specifications for managing and monitoring hardware. By using IPMI or SNMP, we can monitor the voltage, temperature, power usage, and other metrics of physical hardware such as physical servers and network switches.

All of the aforementioned solutions do not resolve the challenges described in the Section 1.2. First, each has its own specific target resources and can not monitor heterogeneous resources. For example, Ceilometer provides all the basic statistics of the OpenStack components and resources. However, it cannot handle other resources in the infrastructure, such as Netflow data, and unstructured data from applications and system logs. Second, none of them supports flexible cross layer (application layer, service layer, and infrastructure layer) monitoring. In the cloud computing environment, various types of applications will run on the cloud and each may have different metrics to be monitored for performance or management purposes. A monitoring system should allow administrators and applications to define and utilize new metrics. Lastly, they do not provide sophisticated analytic functionalities. Their main function is to collect monitoring information and to generate event notifications.
based on it. To provide meaningful information for resource managers to make scheduling and resource allocation decisions, a monitoring and analytics system should have both data collection and analytics capability.

### 3.2 Analytics System

#### 3.2.1 Monitoring System with Analytics Capability

On the monitoring analytics side, Monalytics [50] presents a hierarchical monitoring and analytics system, where the leaf nodes are data collection points and the other nodes are data aggregation and analytics nodes. It proposes integration of monitoring and online analytics, and suggests that analytic tasks should be executed locally at the monitoring data acquisition point or along the path of a hierarchical tree structure. [69] is a continuation of the Monalytics work. This paper presents a software overlay called Distributed Computation Graph (DCG) which provides flexibility to dynamically create analytics functionalities on top of the Monalytics architecture. By using DCG, the two main technical contributions of the paper are: 1) efficient use of resources: only initiate analytics functions at necessary locations; and 2) reduce 'Time to Insight' (TTI): reduce the delay between when monitoring data is generated and the time it takes to produce the analytics results. Although this system provides stream processing of monitoring data and a flexible model for implementing analytics tasks, it provides no functionalities for analysis of historical monitoring data. Analysis of historical monitoring data is an important functionality in a monitoring and analytics system, as it provides a means to understand the system behavior and 'learn from the past'. The Monalytics system also suffers from lack of fault tolerance and scalability due to resource contention. As the number analytics jobs deployed on the Monalytics system increases, more analytics tasks will be allocated to each of the internal analytics nodes (referred to as Monitoring Broker in the paper) of the tree architecture. As a result, this would limit the scalability of the system.

The MISURE system [63] is an application-level cloud monitoring system that uses stream processing. It proposed a scalable and fault tolerant framework for monitoring applications running in a cloud environment; however, MISURE is mainly designed for application layer monitoring. It also omitted the batch processing part where historical monitoring data is analyzed for better understanding of system behaviors.

The authors of [54] present a 3-D monitoring solution for monitoring different metrics corresponding to each physical machine and application. The later work on 3-D monitoring
3. RELATED WORK

[35] presents a system called Ceiloesper which combines Ceilometer and Esper CEP [13] to support stream processing, but again it doesn’t provide analytics capabilities on historical monitoring data.

3.2.2 Large-Scale Processing and Analytics Engine

There is also a large body of research and projects related to designing and implementing large-scale processing engines. MapReduce [42] is one of the earlier works in the area of large-scale distributed computing. This paper proposed a method to process data on a distributed system by using a map function and a reduce function. Hadoop MapReduce [14] is an open source implementation of MapReduce. Due to MapReduce’s inefficiency in data reuse (i.e. writing results to disk in every iteration), it is not performance optimized for iterative machine learning, graph analysis, and ad-hoc query.

Spark [77] is a general large-scale processing system that uses the concept of Resilient Distributed Datasets (RDD) to support both in-memory and on-disk data processing sharing. According to the paper, Spark is able to perform 20 to 40 times faster than Hadoop MapReduce. A number of libraries also developed on top of Spark to support stream processing, graph analysis, interactive query, and large-scale machine learning.

On the stream processing side, Apache Storm [6] is a popular distributed stream processing engine that allows users to specify a stream processing topology and logic by using the spout and bolts abstraction. One of the limitations of Storm is that it does not support update of the topology. Apache Samza [5] is another distributed stream processing engine that runs on top of Apache Hadoop YARN [7]. Spark Streaming [78] is a library that runs on top of Spark to provide distributed stream processing. Unlike the other two stream processing engines, where streaming data is continuously processed, Spark Streaming processes streaming data in mini batches. By discretizing stream processing, Spark Streaming can provide a uniform programming model with the Spark batch processing and enables more deterministic processing of data. One drawback of discretized stream processing is its limitation in latency.

There are also a number of specialized processing systems. Pregel [52] is a distributed graph processing engine. Dremel [57] and Apache Drill [2] are created to support interactive analysis and ad-hoc query. Apache Lucene [3] and Elasticsearch [12] are designed to support text search.
3.2.3 Anomaly Detection

Anomaly detection is a well-studied research topic. There has been extensive study of the problem under many different settings using various statistics tools and machine learning algorithms [38]. However, in the context of virtualized data centers, the problem is still relatively new.

For anomaly detection algorithms, the EBAT system uses an entropy-based approach to detect change in statistical distribution of input metrics. The authors demonstrated that it outperforms threshold-based anomaly detection algorithms [70]. PREPARE uses Tree-Augmented Naive (TAN) Bayesian network to predict online anomalies and proactively take prevention actions [65]. However, these algorithms mainly works at application level, and have not considered issue of root cause analysis. Many other existing techniques only apply to the single metric case, where only a single metric (e.g. CPU utilization) is considered for anomaly detection (e.g. [70][71]). Furthermore, because cloud applications often show fluctuating resource demand (e.g. operating under the time-of-the-day effect) that typically follow multi-modal distributions, statistical anomaly detection techniques (e.g. [75]) that assume a pre-defined (e.g. Gaussian) distributions do not work well for our case.

For root cause analysis, Sherlock [32] is an anomaly detection system that uses bayesian networks to infer the root cause of anomalies. However, it is a supervised learning approach that requires extensive training. FChain [59] uses Fast Fourier Transform (FFT) to learn the burstiness of the input signal and use the information to raise alarms, and similar to ours, it proposes the concept of anomaly propagation. However, it uses timestamp information to infer the original cause of the anomaly, which may not be accurate in a distributed system where time on each machine may be desynchronized. CloudPD [60] uses k-nearest neighbor for anomaly detection and metric correlation for anomaly classification. However, this approach does not consider anomaly propagation. Finally, the authors of [72] proposed a technique for root cause analysis in component-based systems. However their approach focuses at application-level anomaly correlation, and does not consider the effect of network and system virtualization.
4.1 Overview of Monitoring and Analytics System

This section provides an overview of a monitoring and analytics system.

Traditional monitoring systems focus on collecting monitoring data and storing them for visualization and simple alarm functionality, but most of the time these systems have very limited processing and analytics capabilities. As the size and complexity of cloud infrastructure continues to grow, we see an increasing demand for analytics on infrastructure data to provide more visibility and management assistance. We believe a cloud infrastructure should have a monitoring and analytics system that is capable of collecting infrastructure data, and storing and analyzing them.

The relationship between data collection, storage, and analytics is shown in Figure 4.1. Data collected by the system can be stored in the storage module to maintain historical records and support user access. Collected data can also be sent directly to the analytics module for real-time analytics, such as intrusion detection and anomaly detection. The analytics module can access data from the storage module to perform off-line processing. Some examples of off-line processing include machine learning training tasks, and graph processing. The off-line processing results can also be used to support real-time processing.

In the rest of this section, we will break down each part (collection, storage, processing) of a monitoring and analytics system to better understand the components involved, as well as their mutual dependencies and constraints. This analysis is the foundation for our system design and implementation. We try to make this discussion general and system-agnostic.

**Collection:**

There are three parts related to monitoring data collection: acquisition, raw data transformation, and monitoring data transportation and aggregation.

Monitoring data can be polled periodically or pushed by data generators. Then raw data needs to be transformed into proper structures and sent to the destination. During collection, system overheads including network IO, disk IO, and CPU utilization are introduced. Data
pushed automatically often can produce less overhead compare to polling; however, some continuous data such as CPU utilization, need to be sampled and thus require use of the polling method. When polling for monitoring data, one of the decisions to make is the polling period. That is related to how fine-grain the monitoring data needs to be. Polling too often might result in unnecessary overhead. Therefore the system needs to strike a balance between the amount of data to collect and the monitoring overhead.

Depending on the system’s architecture, data may or may not be sent over network. That’s another architectural decision that can affect the overall performance of the system. The design of monitoring data flows affects the latency and system overhead.

Storage:

Although there are many distributed databases already available such as HBase (NoSQL) [15], Cassandra (NoSQL) [9], and MongoDB (NoSQL) [20], each database is optimized for different purposes and thus may not fit all the requirements for a storage module in the monitoring and analytics system. To evaluate a storage system, we need to evaluate the involving disk IO, CPU utilization, memory utilization, and scalability, based on the use case of the target system.

The goal of monitoring data storage is to support data analysis and on-demand data queries from users. Scalability is crucial for both data analysis and data query; however, storage scalability requirements for data analysis and data query can be different. To support
data queries, a system needs to perform searches on large amounts of data and often return data of a small size. On the other hand, data analysis often involves processing of large amounts of data and requires storage to be optimized for disk IO with large blocks of data (to reduce random disk IO). This difference brings a challenge for the design of a storage system for a monitoring and analysis system, which supports both query and analysis of the monitoring data.

To design a storage module for the monitoring and analytics system, we need to first understand the characteristics of monitoring data. First, we can recognize that monitoring data is time series data. Each data point, regardless of whether it is structured or unstructured, has a time stamp attached to it. Hence, data can be organized and optimized based on the time value. Next, since monitoring data is generated continuously, the storage module needs to support continuous and potentially large amounts of write operations. At the same time, analytics and data query will involve data read operations.

When designing the storage module, there are several factors that need to be considered including storage format (e.g. file format), data indexing (e.g. database indexing, file indexing), data organization and distribution (e.g. file folder structure, data distribution amount servers), and data fault tolerance. For both data query and analytics, it is important to quickly locate the data of interest and read them from disk. Well designed data indexing and data organization are important in this case. In this distributed environment, data needs to be properly sharded and replicated among different servers to improve IO. Proper storage format is also important since it can help reduce unnecessary disk IO. For example, there are column-oriented file formats and row-oriented file formats. If the data have a large number of columns and the system often only needs to access some columns for analysis, then column-oriented file format can reduce disk access, as it does not require reading the entire row. Besides performance, storage format often also affects the compression rate of the data. Due to the lack of format, unstructured data needs to be stored differently compared to structured data.

Another consideration of the storage module is memory usage. Caching data in memory can drastically improve performance compared to reading directly from disk. However, since analytics modules will also require large amounts of memory during analytics tasks, memory needs to be allocated properly.

**Analytics:**

Analytics of the monitoring and analytics system contains two parts: the analytics platform and analytics algorithms. The analytics platform is used to support the execution of the
analytics algorithms. The analytics platform needs to operate with the underlining storage system in an efficient manner. Since we emphasize scalability of the system, the analytics platform also needs to be scalable. That means the analytics platform should work on top of a cluster of machines and efficiently distribute computation tasks to the cluster while maintaining global execution orders and consistency. Moreover, the analytics platform needs to support both batch analytics/processing and stream analytics/processing.

**Monitoring as a Service:**

Monitoring as a Service (MaaS) is another important feature of a monitoring and analytics system. MaaS is key to providing an open monitoring environment that can support the needs of multiple tenants in shared resource environments. The system can then provide open APIs to allow users to: 1) stream/submit custom monitoring data to the system for storage and analysis purposes; 2) submit analytics tasks and retrieve the results on demand; 3) access historical monitoring data-on-demand (with access control). The monitoring system can leverage its large volumes of cross-layer, heterogeneous monitoring data to conduct infrastructure-wide analytics that benefit its tenants. For example, such analytics can include security detection and auditing, dynamic resource allocation, and automatic system diagnosis.

### 4.2 Architecture

The high-level logical architecture of MonArch is shown in Figure 4.2. This architecture can be divided into four vertical layers: acquisition, streaming, batch analytics, and user access layer. Monitoring data generally moves from left to right in this architecture. The acquisition layer obtains or generates monitoring and measurement data. These data is then sent to the streaming layer for transport and stream processing. The raw data and optionally the stream processed data are then sent to the analytics layer for storage and batch analytics. Users can access the monitoring data and analytic results through the access layer. By using this architecture, the system can achieve monitoring data collection, storage, and analytics with scalability and flexibility.

#### 4.2.1 Acquisition Layer

The acquisition layer is responsible for acquiring and obtaining monitoring data and submitting it to the streaming layer. There are three kinds of components in this layer: Agent, User Agent, and User Agent Receiver.
4. MONARCH SYSTEM

Agent: Agents are components that acquire infrastructure level monitoring data and send it to the streaming layer. These data include virtual and physical server states, network states, and all other metrics that need to be constantly monitored. As there are multiple types of metrics that agents can monitor, there can exist multiple types of agents. Figure 4.3 shows the general internal pipeline of an agent. Monitoring data is first acquired by the Data Collector, then data is formatted, processed and published to the streaming layer. In the agent, the Light Weight Processor is only responsible for light weight processing such as calculating differences, calculating the rate of change, and evaluation of threshold violation. The Processing Tasks Store keeps a list of processing tasks that the Light Weight Processor needs to execute for different monitoring metrics. We believe more complex processing and analytics should be done in a more centralized way in the streaming and the batch analytics layer (Refer to 4.2.5.1 for a detailed discussion). The advantage of providing light weight processing capabilities in the agent is to support the ultra low latency alarm feature. An example use case of this feature is to support applications that require simple threshold-based evaluation on the monitoring data to provide immediate notification (e.g. within 500 ms) of rule violation. When an alarm is triggered here, the Light Weight Processor sends a message to the Notifier of the MonArch system to notify users or trigger predefined management actions. Since we should minimize the monitoring overhead at the data collection point, only a limited amount of light weight processing should be placed in the agent. The Metrics

Figure 4.2: MonArch Architecture
4.2 Architecture

Store maintains a list of targeting monitoring metrics and their polling frequency. The Data Collector acquires monitoring data based on the information provided by the Metrics Store. The Metrics Store allows easy control of the monitoring metrics and monitoring period.

**User Agent**: The User Agent is a component that allows users to monitor custom metrics. By using the User Agent, users can specify the metrics to monitor with a configurable monitoring frequency. Examples of custom metrics include web server metrics (e.g. requests per second, request latency, request location, number of connections), database throughput, and number of online applications. The User Agent has two modes: polling mode and passive mode. In polling mode, the user specifies a metrics information and monitoring frequency, then the User Agent periodically retrieves the monitoring data and sends it to the User Agent Receiver module. In the passive mode, the user specifies metrics information and the User Agent waits for input data. When input data is received, it formats the data and sends it to the Data Admission Gateway. In both modes, the User Agent can handle both structured data and unstructured data.

**Data Admission Gateway**: The Data Admission Gateway receives monitoring data from User Agents, checks the formats, and performs access control. If there is no problem with the monitoring data, it sends the data to the streaming layer. The main purpose of this component is to prevent the system from user attacks using the User Agent. This could potentially also be used in the future for billing purposes. One of the important requirements
4. MONARCH SYSTEM

of this component is its scalability. It should be able to scale up when the number of User Agents increases and provide consistent performance. Since this component only needs to store the access control information for each user, it can be easily scaled up if the access control rules are properly sharded among a number of Data Admission Gateways.

4.2.2 Streaming Layer

The streaming layer is responsible for transporting the monitoring data to the storage module and performing stream processing at the same time for real-time analytics purposes. This layer includes a messaging system and a stream processing system.

**Messaging System:** As the monitoring data is generated and sent to the streaming layer, it is important to have a messaging system that buffers the data before it is processed by the stream processing module. Another benefit of using a messaging system is the flexibility to allow multiple stream processing tasks to subscribe to the same input data stream. This is useful when multiple stream processing tasks, performing different streaming analysis, need to access the same set of streaming data asynchronously (i.e. access the data at a different time). We emphasize that this messaging system should be distributed and scalable. One of the challenges for designing a scalable messaging system is providing global ordering of messages. Since the goal of having the messaging system maintaining the global message ordering is to ensure processing correctness in the stream and batch analytics tasks, we made a system-wide decision to attach a time stamp to each message and relax the global ordering requirement of the messaging system. (The details regarding consistency and global ordering will be discussed in the discussion section in this chapter.) By using this design, the messaging system only needs to maintain ordering of each individual agent and thus allowing it to scale up.

**Stream Processing:** The stream processing module is responsible for real-time processing of the incoming monitoring data. It retrieves monitoring data from the messaging system and performs the required processing tasks and actions. Some examples of stream processing for monitoring data include detecting threshold-based violation, finding temporal or spatial correlation of the monitoring data across different resources, and detecting anomaly based on a predefined or learned pattern. Since stream processing may require a series of historical monitoring data points for processing tasks, this component is able to keep a window of monitoring data when needed. For long-term temporal information, the stream processing component can contact the storage module. One of the important requirements for this stream processing module is scalability. There are two aspects to scalability of this mod-
4.2 Architecture

1) the system can be scaled up to support higher throughput as the amount of streaming monitoring data increases; 2) this module can be scaled up to support a higher number of real-time analytics tasks. The first aspect is highly dependent on the design of analytics algorithm such as data dependency and analysis complexity. On the system side, since data is streamed to a centralized cluster for stream analytics, we can easily improve scalability by adding processes that fetch data and conduct processing. For the second aspect, proper data sharding and replication in the message system and appropriate allocation of stream tasks can provide the required scalability.

4.2.3 Batch Analytics Layer

This layer is responsible for storage and analysis of the historical monitoring data. There are two modules in this layer: the storage module and the analytics module. These two modules operate cooperatively with the stream processing module in the streaming layer.

The storage module stores the raw monitoring data as well as some analytics results from the analytics module and the stream processing module. Raw monitoring data is sent to the storage module by a stream processing task which does formatting and organizing. To support scalability of the batch analytics tasks, the storage module is designed on top of a distributed file system or distributed database. Proper data format and organization are important for performance of analytics tasks. Data formatting will be discussed in detail in the implementation section.

The analytics module is similar to a batch processing system. It is responsible for analyzing the monitoring data stored in the storage module to discover patterns, trends, and useful information that could benefit infrastructure management and users of the infrastructure. Analytics results of this system are stored in the storage module to allow access from the stream processing module and the modules in the access layer. Useful analytics results can help improve the performance and accuracy of the stream processing module. For example, off-line machine learning algorithms running on top of the analytics module can use the historical monitoring data to find resource usage patterns. Machine learning training results can be written to the storage module, and stream processing module can use it to detect anomalies.

There are many existing generic large-scale distributed processing engines, such as Apache Spark and Apache Hadoop, that support distributed data processing. The goal of the these engines is to provide high-level processing APIs, such as map, reduce, and groupbykey, to simplify implementation of parallelized data extract, transform, load (ETL) and at the same
4. MONARCH SYSTEM

time offer fault tolerance. In addition to the performance of the processing engine, performance is highly dependent on the analytics algorithms and optimization of algorithms based on the underlying engine. The data organization in storage also has a big impact on the analytics performance. In the next chapter, we will describe some of the analytics we designed and implemented in the analytics module.

The analytics module and the storage module should integrate closely with the stream processing module. Some stream processing tasks may require access to the historical monitoring data, so it is important that historical monitoring data is accessible to the stream processing module with low latency when needed. Moreover, under different conditions, the streaming processing module may require some batch processing of the historical data, and since these batch processing tasks may not be known in advance, the analytics module should be integrated closely with the stream processing module and be able to handle processing tasks from the stream processing module effectively.

4.2.4 User Access Layer

The user access layer is an interface layer to users of the system. As we mentioned, users should be able to access current and historical monitoring data-on-demand. This layer provides the capabilities to allow on-demand access of monitoring data. There are three modules in this layer: Notifier, API, and Graphical User Interface (GUI). In general, there are two ways a user can get monitoring data or analytics results from the system: 1) poll or 2) push mechanisms. MonArch supports both methods.

**API**: When using the poll method, a user actively queries the system for interested monitoring data. The API module is created for this purpose. Users can send a request to the API module to access certain monitoring data. The API module also allows users to submit analytics criteria or tasks. This module is connected to the storage module for data access and adding analytics information.

**Notifier**: We created the notifier module for the push method. Since it is not efficient to stream all the monitoring data to users, the notifier module only notifies users under certain conditions. The notification condition, along with the notification method (e.g. HTTP callback, email), can be specified by users. One may debate that monitoring data live streaming to the users is important since users can implement custom stream processing on those data. However, we believe that analytics tasks should be done by the MonArch system and users can submit the processing tasks to the system. This way, we can minimize the unnecessary network traffic for transportation of monitoring data to users. It also frees the users from
needing to create and deploy analytics tools. The analytics module in MonArch also handles scalability and fault tolerance, which makes implementation and deployment of analytics tasks simpler for users.

**GUI:** The GUI module supports mainly poll methods. It provides users with a graphical view of the monitoring data and analytics results. Visualization is an important component in data analytics, so the GUI module plays an important role in this system.

### 4.2.5 Discussion

The key design decisions and system characteristics of the MonArch architecture are discussed in this section.

#### 4.2.5.1 Logically Centralized Monitoring and Analytics

MonArch stores all the monitoring data in a distributed file system for analytics and processing. This provides a global view of the system which enables correlation of metrics from different monitoring sources. An example of this is finding the infrastructure usage pattern by looking at CPU utilization of physical servers and VMs, bandwidth usage from OpenFlow switches, and NetFlow from routers. We believe more complex processing and analytics should be done in a more centralized way for three reasons: (1) Resource utilization: Since agents are generally located closer to the resources being monitored, they may be in an environment with a limited or competitive amount of resources (CPU, memory, storage). For example, a VM monitoring agent could be placed in a physical machine that hosts VMs in order to facilitate its communication with the hypervisor for getting VM monitoring data. In this case, heavy resource utilization of the monitoring agent could affect the performance of other management modules or VMs, thus we should minimize the agent resource utilization when possible. (2) Fault tolerance: When data is processed in a logically centralized processing system, it is much easier to handle fault tolerance. We only need to focus on the fault tolerance of the processing system. Failure of an agent should not affect the processing and analytics of the whole system. (3) Centralized knowledge: When monitoring data is processed in a logically centralized manner, the system has more information about the overall infrastructure and can extract information such as correlation of data collected by different agents.
4. MONARCH SYSTEM

4.2.5.2 High Scalability

Scalability is one of the most important objectives in the MonArch design. As we have discussed above in the descriptions of each individual component in the architecture, all components are designed with scalability in mind and can be scaled up. More monitoring data can be collected by running more agents. With today’s scalable messaging system, scaling up basically means adding servers and partitions. Processing and analytics tasks are executed in a centralized cluster, so we can easily provide high scalability by using the existing distributed analytics engines. The analytics algorithms running in the MonArch system are also highly scalable. (a more detailed discussion is provided in Chapters 5 and 6.) Thus MonArch provides higher scalability than other existing monitoring systems such as Ceilometer. One example of this is in the alarm mechanism. Ceilometer has an Alarm Evaluator component which evaluates violation alarms by looking at alarms one by one and requesting statistics for each alarm for evaluation. Since Alarm Evaluator runs as a single process, it has a scalability bottleneck. In the MonArch system, alarm evaluation tasks are done in a distributed fashion.

4.2.5.3 Monitoring and Analytics Support for a Wide Variety of Data Sources

Traditional monitoring systems focus on monitoring basic compute (e.g. CPU and memory) and network (e.g. bandwidth) resources. Many systems also monitor different layers of the cloud system (e.g. physical layer, virtual layer, or even application layer). We believe that the ability to monitor all kinds of monitoring information across different layers is important. Our system can collect and process structured and unstructured data such as VM CPU usage, physical machine memory usage, system log, user specified application information, and NFV logs (e.g. DPI logs). As such, our system achieves integrated monitoring and analysis of all types monitoring information across multiple layers.

We would like to emphasize two types of monitoring information in our system that other existing monitoring systems do not take into account for integrated analysis. (1) Human interaction with the system, which includes the administrator’s command line input and database modification. Frequently system errors are caused by human interaction with the system such as upgrade or maintenance. In a traditional monitoring system, we can only see the symptom, not the root cause. Therefore, collecting all the users’ interaction information and jointly analyzing with other monitoring data can shorten the time used for system diagnosis. (2) Infrastructure services monitoring: In an OpenStack environment, cloud service components, such as Nova, Neutron, and Ryu controller, run in the system, but the current
4.2 Architecture

cloud monitoring models does not take into account the performance and information from these services. User experience and application performance is highly dependent on the performance of these services. For example, some user applications running in VMs can retrieve monitoring data from Ceilometer for scaling purposes. In this case, Ceilometer’s API request latency and system VM boot up time should be factored in when making user scale up/down decisions. If the cloud system can boot up a VM quickly, the application should scale up even if it only experiences high traffic load for a short period of time. On the other hand, if VM boot up time is long, then it may not be cost-effective to scale up.

4.2.5.4 Data Ordering

In the MonArch system, as monitoring data is collected and generated in a distributed fashion and processing of monitoring data from multiple sources is needed to create a global view of the system, data ordering is important and needs to be carefully designed. One option is to use the messaging system to conduct global ordering of all the monitoring data and assign a time stamp as the data comes in. However, due to the unknown delay between the agents sending data, and the data being received by the messaging system, ordering of monitoring data can be captured incorrectly by the messaging system. Another total ordering method in a distributed system is having the agents communicate with each other to agree on a global order. These kinds of methods would introduce large amount of computation and network overhead and thus they are not suitable for a monitoring system where potentially monitoring data can be generated at a high rate. The third option is to attach a time stamp to each piece of monitoring data at the collection point. But one of the problems of this approach is that there could be a time drift between servers and the time stamp of monitoring data from different servers may not be consistent. In MonArch, we choose the third method. To solve the problem of time inconsistency, we have the agents periodically synchronize the machine time. Although the time between servers can still drift after each synchronization, this would be relatively small. By properly adjusting the time synchronization period with respect to the monitoring period, in most cases the time drift would be small enough to be ignored compared to the monitoring period. For the very high rate monitoring agent (e.g. with a 50ms monitoring period), the monitoring data can be sent in batches by the agent to the messaging system. Then based on the synchronized time, the agent can more accurately give time a stamp to each piece of monitoring data (or give one time stamp to the whole batch). Since monitoring at a very high rate can introduce high overhead, MonArch does not use very high rate monitoring agents. Thus proper time synchronization between servers
4. MONARCH SYSTEM

4.2.5.5 Extensibility

As new hardware, software and new applications are introduced in a cloud environment, extensibility of a monitoring system is important to easily provide support for monitoring of new resources. With MonArch’s architecture, a User Agent can be quickly used to monitor new types of resources. Specialized monitoring agents can also be developed and plugged into the system for new monitoring data collection tasks. The benefit of developing a specialized monitoring agent, instead of using a User Agent is that its monitoring overhead can be optimized, based on the characteristics of the data collection task. With its modular design, MonArch is highly extensible for new monitoring and data collection tasks. New analytics tasks can also be easily added to the MonArch system for processing of the newly added metrics.

4.3 Implementation

We have implemented MonArch and deployed it in the SAVI Testbed. Figure 4.4 shows the detailed implementation of the system.
The MonArch system provides monitoring capabilities for physical, virtual, and application layers in a cloud infrastructure. A number of specialized agents are provided in the MonArch system for monitoring resources in the virtual and physical layers. These agents include: 1) Physical Agent: monitors the CPU usage, memory, and disk of the physical server; 2) OpenFlow Agent: responsible for OpenFlow network monitoring; 3) NetFlow Agent: monitors TCP/IP traffic by using NetFlow specification; 4) Ceilometer Compute Agent: Compute Agent that is responsible for monitoring the CPU, disk, and network usage of the virtual machine in OpenStack; 5) IPMI Agent: monitors physical devices by using IPMI specifications. Since we implemented MonArch to work closely with OpenStack, we integrated the Ceilometer Compute Agent as one of the data collection agents.

For application layer monitoring, MonArch also provides a generic data collection agent called User Agent that allows users to monitor custom metrics. The User Agent allows users to specify monitoring metrics with different acquisition periods and actions. When the monitoring data is acquired by the User Agent, data is sent to a Data Admission Gateway, which does admission control and subsequently forwards data to the messaging system (Kafka).

Since the system receives a large amount of information from a wide variety of sources and performs stream processing, this puts more load on the messaging system. The messaging system should be scalable and provide high throughput. RabbitMQ is a message queuing system widely used by Ceilometer and other components in the OpenStack ecosystem. However, RabbitMQ is not suitable for large-scale resource monitoring due to its limited throughput. In MonArch, we choose an alternative message system, Kafka, which provides higher scalability and throughput than RabbitMQ.

One of the objectives of the MonArch system is to have the ability to monitor and analyze a wide range of system data, including structured data and unstructured data. HDFS is a good option for this. It provides fault tolerance and good integration with existing processing engines. It also gives flexibility on file placement and table placement for performance optimization. Hence, we use HDFS for our storage module. There are four kinds of data stored in HDFS: 1) structured and unstructured monitoring data; 2) stream analytics results; 3) batch analytics results; 4) list of analytics tasks to be executed (potentially periodically). These data are properly formatted and organized in HDFS to optimize for performance.

Another reason for storing monitoring data is to provide Monitoring as a Service in MonArch. That means users should be able to query monitoring data of a resource for a certain period of time with some requirements. Since HDFS, by default, is not optimized for query, we have created a Monitoring Data Store that is used to support on-demand access of
monitoring data. The Monitoring Data Store is implemented on top of Apache Cassandra, and is optimized for user queries.

Spark is used for both streaming and batch processing in our system. Spark is a distributed data processing system like Hadoop. Spark’s Resilient Distributed Dataset (RDD) design provides superior performance compared to Hadoop. For time sensitive analysis, Spark streaming programs take input from the messaging system directly. For batch processing, we leverage Spark and the libraries (GraphX, MLlib, and SparkSQL) on top of Spark.

For user access, an API module is provided in the MonArch system to support tenant-based: 1) access of monitoring data; 2) access of analytics results; 3) submission of analytics requests or custom analytics logics. A notifier module in MonArch performs notification actions when the stream processing tasks detect certain conditions that require notification of users or triggering of management actions. The stream processing tasks are responsible for sending notification requests to the notifier by using Kafka.

In the following subsections we describe the detailed implementation of each component of the MonArch system.

4.3.1 Agent Implementation

The architecture and general pipeline of an agent is discussed in the section and Figure 4.3. To implement this pipeline and at the same time provide extensibility, we use a modular and pluggable framework so that different kinds of data collectors, formatters, light weight processors, and publishers can be dynamically activated and deactivated. The agents are implemented in Python language for better interoperability with OpenStack components. For the specialized agents (i.e. OpenFlow, NetFlow, Physical Server, and IPMI Agent), we use a library called stevedore for dynamic loading of modules. Stevedore builds on top of setuptools entry points to support dynamic adding and removing extension modules. That allows the administrator dynamically control of the metrics to monitor, the formatting method, the processing logic as well as how and where to publish the monitoring data.

For the purpose of formatting monitoring data, we first define some terminologies:

- **meter name**: a meter name is given to each type of metric (e.g. openflow_flow_bw means the openflow flow bandwidth metric)

- **resource ID**: each monitoring target is considered as a resource and is given an identifier (e.g. VM resource_id in OpenStack)
4.3 Implementation

- **resource metadata**: the metadata associated with the resource such as VM size, OpenFlow flow source MAC, etc. This field can contain multiple key value pairs. It is mainly used for queries with conditions related to resources.

- **monitoring data unit**: the unit of the monitoring data, if it is metric data

- **data**: the actual monitoring content. This would be numeric when monitoring structured data and text when monitoring unstructured data.

The collected monitoring data in all the agents are formatted into the fields above. This data is serialized when published to the messaging system.

4.3.1.1 OpenFlow Agent

The OpenFlow Agent is responsible for collecting network data from OpenFlow enabled devices. The metrics that the OpenFlow Agent can monitor are listed in Table 4.1.

<table>
<thead>
<tr>
<th>Meter Name</th>
<th>Resource</th>
<th>Acquisition Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Byte Count</td>
<td>Flow (DPID + hash(flow_info))</td>
<td>Collected</td>
</tr>
<tr>
<td>Flow Packet Count</td>
<td>Flow (DPID + hash(flow_info))</td>
<td>Collected</td>
</tr>
<tr>
<td>Flow Duration Second</td>
<td>Flow (DPID + hash(flow_info))</td>
<td>Collected</td>
</tr>
<tr>
<td>Flow Duration Nano Second</td>
<td>Flow (DPID + hash(flow_info))</td>
<td>Collected</td>
</tr>
<tr>
<td>Flow Bandwidth</td>
<td>Flow (DPID + hash(flow_info))</td>
<td>Calculated</td>
</tr>
<tr>
<td>Port Receive Byte Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Receive Packet Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Receive Bandwidth</td>
<td>Port (DPID + port number)</td>
<td>Calculated</td>
</tr>
<tr>
<td>Port Receive Drop Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Receive Error Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Transmit Byte Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Transmit Packet Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Transmit Bandwidth</td>
<td>Port (DPID + port number)</td>
<td>Calculated</td>
</tr>
<tr>
<td>Port Transmit Drop Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
<tr>
<td>Port Transmit Error Count</td>
<td>Port (DPID + port number)</td>
<td>Collected</td>
</tr>
</tbody>
</table>

Table 4.1: OpenFlow Agent Monitoring Capabilities

In OpenFlow, there are two main types of monitoring entities: port and flow. In the data plane, counters are attached to each port and flow, therefore, we can obtain packet and
4. MONARCH SYSTEM

byte count including number of drops and errors of each flow and port. To calculate the
bandwidth, we use the following formula with the collected counter values and their time
stamp:

\[
Bandwidth^c = \frac{B_{t2}^c - B_{t1}^c}{T_{t2}^c - T_{t1}^c}
\]  (4.1)

where \(B_t^c\) denotes the number of bytes at time \(t\) for counter \(c\), and \(t2\) is a time stamp
in seconds later than \(t1\). This calculation is done in the Light Weight Processor in the agent
pipeline.

During data collection, the OpenFlow Agent sends a ports statistics request and a flows
statistics request to the OpenFlow Controller (Ryu [26] in our implementation). The Open-
Flow controller sends a request to the switch to obtain the counter values of all the ports
and flows and forwards the results to the OpenFlow Agent. Since OpenFlow Agent acquires
monitoring data by using the APIs of the OpenFlow controller, it can leverage monitoring
optimizations that are implemented in the OpenFlow controller. The clear interface between
the OpenFlow Agent and the OpenFlow controller will minimize implementation changes if
OpenFlow protocol evolves and changes in the future.

4.3.1.2 NetFlow Agent

NetFlow is a specification that allows monitoring of IP and Transport layer statistics. It
provides information including packet counts, byte counts, number of flows, duration of
flows, transport layer protocol, source IP and port number, and destination and port number.
These statistics are collected in the NetFlow capable switches and periodically sent to an
external endpoint.

Since NetFlow is a push-based protocol, the data collector in the NetFlow Agent is de-
signed to listen for NetFlow packets and submit them through the agent pipeline. As each
NetFlow message contains multiple metrics related to a resource, we reformat the message
into the form shown in Table 4.2.

4.3.1.3 Physical Server Agent

For a Physical Server Agent, we use a library called psutil [25] for obtaining physical server
CPU, memory, disk, and process information. The metrics that the Physical Server Agent is
capable of monitoring are list in Table 4.3.
4.3 Implementation

<table>
<thead>
<tr>
<th>Meter name</th>
<th>packet count, or byte count, or number of flows, or duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource ID</td>
<td>concatenation of switch ID, Protocol, Source IP, Source Port, Destination IP, Destination Port</td>
</tr>
<tr>
<td>Resource metadata</td>
<td>Map{protocol: text, source_IP: text, source_port: int, destination_IP: text, destination_port: int}</td>
</tr>
</tbody>
</table>

Table 4.2: NetFlow Formatting Method

4.3.1.4 User Agent

The User Agent is designed to allow collection of custom metrics. One requirement of the User Agent is the ability to dynamically add new monitoring tasks or remove them. Since stevedore is built on top of the Python setuptools entry point, new monitoring tasks needs to be installed using setuptools before it can be loaded by stevedore. This is not flexible enough for a non-specialized agent such as a User Agent. To address this issue, we implemented the User Agent on top of the Tornado \[29\] framework. Tornado is an asynchronous framework in Python. It allows us to implement multiple monitoring tasks and execute them in an asynchronous manner. This solves the blocking problem that occurs when tasks are executed sequentially. Another benefit of using Tornado is its native support for web development. We want to make the User Agent as easy to use as possible, so we provide a web GUI in the User Agent for management and modification of monitoring tasks.

For monitoring data collection, the User Agent supports two types of data acquisition methods: poll and push. For the poll method, the User Agent periodically executes the monitoring tasks provided by the users. The User Agent supports three ways to specify a monitoring task: Python script, Shell command, and Shell script. Each of these scripts or commands is required to output the monitoring data (value or text) to the stdout, where the User Agent picks it up and sends it through the agent pipeline. For the push method, a RESTful API is provided by the User Agent to allow submission of monitoring data.

For monitoring tasks management, another RESTful API is provided to allow adding and deleting of monitoring tasks. A web GUI, shown in Figure 4.5, is also provided to allow easy management of monitoring tasks.

Since the User Agent can be deployed on any system, we would like to perform admission control so that only authorized users can submit data to the MonArch system. For this reason, instead of sending the monitoring data directly to the messaging system, it is sent to a Data Admission Gateway for admission control.
### Table 4.3: Physical Server Agent Monitoring Capabilities

<table>
<thead>
<tr>
<th>Meter Name</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical CPU Time (user time, nice time, system time, idle time, io wait time, interrupt request time, soft interrupt req time, steal time, guest time, guest nice time)</td>
<td>Server name + core number</td>
</tr>
<tr>
<td>Physical CPU Percentage</td>
<td>Server name + core number</td>
</tr>
<tr>
<td>Physical disk IO read bytes</td>
<td>Server name + disk name (e.g. sda)</td>
</tr>
<tr>
<td>Physical disk IO write bytes</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk IO read request count</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk IO write request count</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk IO read time</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk IO write time</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk free space</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk total space</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk used space</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical disk usage percentage</td>
<td>Server name + disk name</td>
</tr>
<tr>
<td>Physical swap memory free</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical swap memory usage percent</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical swap memory used</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical swap memory total</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical swap memory swap in bytes</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical swap memory swap out bytes</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory active</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory inactive</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory available</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory buffers</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory cached</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory free</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory usage percentage</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory total</td>
<td>Server name</td>
</tr>
<tr>
<td>Physical virtual memory used</td>
<td>Server name</td>
</tr>
</tbody>
</table>
4.3 Implementation

4.3.1.5 Data Admission Gateway

The Data Admission Gateway is responsible for receiving data from the User Agents, performs admission control and forwards the authorized monitoring data to the messaging system. The Data Admission Gateway supports two types of data submission mechanisms: RESTful and UDP. The User Agent can publish data using either of these mechanisms to submit data to the Data Admission Gateway.

For authentication purposes, each message of the monitoring data is signed with a secret key in the User Agent. When messages are received in the Data Admission Gateway, based on the user ID, a corresponding secret key is used to validate the message. To improve throughput, secret keys are cached in memory in the Data Admission Gateway. For secret key distribution, an API is provided in the API module that allows users to obtain a secret key by provide authentication credentials. In the current implementation, the API module forwards the authentication credentials to the OpenStack Keystone [23] component for authorization purposes.

4.3.2 Storage Format and Organization

For data storage purpose, the structured monitoring message is shown below:

- **meter_name**: String,
- **user_id**: String,
- **timestamp**: Timestamp,
4. MONARCH SYSTEM

resource_id: String,
unit: String,
data: Double,
tenant_id: String,
resource_metadata: Option[Map[String, String]]

When storing the monitoring data in HDFS, we choose the Parquet format. Parquet is a column-based storage format that is efficient when data queries often only need to access selective columns. In MonArch, we choose Parquet because of the following reasons: 1) analytics tasks may not always need to access all the fields in the structure above (especially the resource metadata which can be a relatively large size; 2) some of fields in the structure above, such as resource_id and meter_name, only have a small set of value among all the resources and thus are highly repetitive. Using the Parquet format, data can be easily compressed to save disk space.

Formatted monitoring data is organized by time slots into files based on the time stamp of each monitoring message and stored in the HDFS under the following file structure:

- /short_interval_storage/meter_name/year/month/day/hour/
data_file (configurable interval)/data_file_partitions
- /long_interval_storage/meter_name/year/month/
parquet_data_file (one for every day)/parquet_data_file_partitions

In this file structure, monitoring data is stored in two top level directories: short_interval_storage and long_interval_storage. New monitoring data is first stored under the short_interval_storage directory where each file contains data of a small time interval. The file format in the short_interval_storage folder is configurable (e.g. Parquet, JSON). Later on, data from the short_interval_storage folder is aggregated and organized into a Parquet file that contains one day of data, and stored in the long_interval_storage. Monthly interval level files can also be added based on the specific situation.

The folder structure is designed this way due to the following reasons:

- **Batch Analytics Time to Sight**: Batch Analytics Time to Sight in this context is defined as the time between generation of the monitoring data and the time when the monitoring data is accessible by batch analytics jobs. Since the Parquet format is not designed for appending, monitoring data can be stored as Parquet files with a shorter interval of data in the short_interval_storage folder, so that data does not buffer in
Kafka for too long. Another option is to use JSON format and continuously appending new monitoring to the JSON file in the short_interval_storage. In either case, the batch analytics task can quickly take into account the new monitoring data. However, using only small Parquet files or JSON is not the best for processing efficiency and disk utilization, so we created the long_interval_storage folder.

- **Batch Analytics Performance**: Since performance of batch analytics is dependent on the number of partitions of the input files, processing a large number of small Parquet files in the short_interval_storage may not be efficient. Therefore, by aggregating the files daily into the long_interval_storage with a proper number of partitions, we can improve performance of analytics tasks.

- **Storage Size**: If JSON format is used in the short_interval_storage folder, we can reduce the number of files as appending is supported. However, compared to Parquet, JSON takes up much more disk space and is not efficient for column-based query. Hence, JSON data should also be aggregated and converted to Parquet files and stored in the long_interval_storage folder.

### 4.3.3 API Module & Notifier Module

The API module in MonArch allows users to submit analytics tasks and access monitoring data and analytics results. To access monitoring data, users specify the name of the metric, resource ID and the relevant time period, then the API module will query the requested data and reply to the users. For analytics results, users can access the results of the custom submitted analytics tasks, as well as the results generated by the generic analytic algorithms in MonArch, such as anomaly detection and multi-layer graph construction. This API module is implemented using the Tornado framework.

We use the Alarm Notifier module from Ceilometer as the notifier module in MonArch. However, we extended its implementation to support interfacing with Kafka for receiving notification messages from the streaming and batch analytics jobs. In the current implementation, we support both email notification and webhook notification.
Chapter 5

Analytics and Intelligence in MonArch

This chapter describes the analytics algorithms that are provided in the MonArch system.

5.1 MonArch Graph Processing

In this section, we discuss the graph construction and processing techniques that are provided by the MonArch system.

5.1.1 Multi-layer Graph in Virtualized Cloud Environments

As we have discussed previously, a virtualized cloud environment contains resources in different layers. For example, we have VMs in the virtual layer, physical machines in the physical layer, and applications in the application layer. Many of these resources are interconnected or related. To better understand the influence of each resource and relationships among the resources, we express the relationships among resources using a graph representation, which in turn can be used for graph analysis.

In our graph representation, as shown in Figure 5.1, there are five types of vertices: 1) physical machines in the physical layer, 2) physical network devices in the physical layer (e.g. switches, routers), 3) VMs in the virtual layer, 4) virtual network devices in the virtual layer (e.g. gateways, DHCP servers, routers), and 5) applications in the application layer. There are two types of relationships: 1) link connection that connects vertices in the same layer such as physical network link (this relationship is directed), and 2) host relationship that connects vertices from different layers such as physical machine hosting virtual machines and virtual machine hosting applications (this relationship is undirected).
5.1 MonArch Graph Processing

5.1.2 Multi-layer Graph Construction & Processing

5.1.2.1 Graph Construction

To construct the graph mentioned in the last section, MonArch utilizes both monitoring data and system information. The following are the steps for constructing this graph:

- Obtain physical layer vertices and connections information from the infrastructure (i.e. from the cloud controller and the network controller)

- Get a list of VMs by processing VM monitoring data (and potentially calculate average utilizations)

- Extract virtual layer connectivities (MAC address to MAC address) by processing OpenFlow monitoring data. In MonArch, we consider a directed connection exists between two VMs when one communicates with the other by sending packets

- Retrieve the port (i.e. MAC address) to VM resource ID mappings from the infrastructure and construct the virtual layer graph

- Obtain the physical machine and VMs hosting relationships from the infrastructure

- Create application vertices and hosting information from the user custom monitoring data collected by the User Agents
5. ANALYTICS AND INTELLIGENCE IN MONARCH

Since the virtual connections between VMs can change quickly, a new graph is generated periodically to capture the latest topology. As the graphs generated from different times are all stored in the MonArch, it also allows us to examine how the topology changes over time.

5.1.2.2 Graph Processing

For processing purposes, we mainly focus on VMs and their connectivities. The following is the processing provided by MonArch

- **Input Degree**: the input degree of each vertex. It represents the number of other VMs that connect to this VM.

- **Output Degree**: the output degree of each vertex. It shows the number of other VMs that this VM is connected to.

- **PageRank**: the PageRank value of each vertex. This is used to evaluate the importance of a node in the cluster. This value will often change if the connection pattern changes.

- **Triangle Count**: the number of triangles in the graph. This is used to show the characteristics of the graph.

By looking at the connection pattern between VMs and the pattern changes over time, we can better understand the characteristics of the applications running on top of the infrastructure. The connection pattern information is helpful when we need to detect anomalies or security attacks.

5.2 MonArch Anomaly Detection & Root Cause Analysis

Motivated by the limitations of the existing anomaly detection methods, we propose a new anomaly detection system running on the MonArch system for virtualized cloud data centers. Leveraging data analytics frameworks such as Apache Spark, our system provides scalability to efficiently handle heterogeneous monitoring data in various formats. In particular, we leverage unsupervised learned techniques for identifying anomalies at resource entity level. Given the anomalies detected, we propose a technique that can quickly identify the source of the anomaly in the system.

For discussion in this section, we will first define the following term:
5.2 MonArch Anomaly Detection & Root Cause Analysis

- **Resource Entity**: We define a resource entity as one instance of a type of resource at different layers of the cloud infrastructure. For example, a physical machine, a switch and a link are each considered as a resource entity in the physical layer, whereas VMs and virtual links are considered virtual layer resource entities.

5.2.1 Motivating Example

To motivate the importance of anomaly detection and root cause analysis in a virtualized cloud environment, we will provide an example with the corresponding data where automatic anomaly detection and root cause analysis can reduce management and diagnosis overhead.

For the sake of simplicity, we only focus on a small part of a large cloud infrastructure (shown in figure 5.2) with two physical servers, a network switch, and a gateway. Each physical server is connected to the switch through a physical link, and communication traffic between the two physical servers passes through the switch. For Internet access, the gateway is also connected to the switch. Next, we assume there are two tenants in this environment, and each of them is deploying a three-tiered web application on VMs that are hosted by the two physical servers. In each application, the web server and application server are co-located in one VM and the database is located in another VM (i.e. each user has two VMs). Managed by a VM scheduler, these four VMs are allocated in a way that the two web server VMs are hosted in the physical machine 1 and the two database server VMs are hosted in the physical machine 2. As everything works so far, the two tenants will be happily serving their own web applications to their users by using this cloud shared environment.

Unfortunately, one day a web hacker decides to attack the web application of tenant 1, and he launches a DDoS attack. As a result, large amounts of web requests are sent to the web server 1, which brings down the web application of tenant 1. Due to the large size of the traffic produced by the attack, unexpectedly, the web application of the other tenant is also affected. The reason is that the web server 2 VM and web server 1 VM share the same physical network link for database access and Internet access. Since the attack is almost saturating the physical link, it affects the performance of tenant 2’s web application as well. In such a shared environment, tenant 2 can see the performance problem of his web application, but he has no clue about the cause. At this point, the cloud provider should be responsible for handling the issue, but the existing monitoring system cannot identify the root cause automatically. Due to the scale of the cloud infrastructure, manually tracing the problem is also difficult and inefficient.

To validate the case that we are discussing above, we have produced such an environment
If anomaly detection and root cause analysis can be executed in real time automatically, the system would be able to send an alarm to the administrator and suggest the web server 1 and its input traffic as the root cause. We will further discuss and evaluate this example in the evaluation section of this thesis.

### 5.2.2 Anomaly Detection System Architecture

The goal of the anomaly detection system contains two points: We first want to identify anomalies in the system and report them to the cloud administrator. Because multiple correlated anomalies can be detected simultaneously, we need to analyze the correlations among the anomalies, and identify the resource entity that most likely has caused the observed anomalies.

The architecture of our system is shown in Figure 5.3. Given the variety of metrics collected for the resources being monitored, the *Entity Anomaly Detectors* identify the abnormal behaviors of individual resource entities and raise alarms once they are detected. In our current implementation, there is one anomaly detector for each resource entity in the SAVI Testbed. The data is shown in Section 6.2.1.
cloud infrastructure. To identify anomalies, each anomaly detector builds a profile for each resource entity through historical traces. At run-time, each anomaly detector checks whether the current measurements of a resource entity deviate from the profile predicted values. If so, an alarm is raised to indicate that the resource entity is experiencing an anomaly.

Once a set of anomalies have been identified at a given time, the information regarding the anomalies are sent to the Anomaly Correlation Engine, which tries to identify the source of the anomalies through root cause analysis. To achieve this objective, the Anomaly Correlation Engine relies on domain knowledge captured by the current network and system configurations. This includes data center topology, server and network configurations, as well as virtual machine and application characteristics and settings.

In the next two subsections, we first present the detailed design of an Entity Anomaly Detector, followed by describing the implementation of an Anomaly Correlation Engine in Section 5.2.4.

### 5.2.3 Anomaly Detection Algorithm

In order to identify anomalies of individual resource entities, we first need to capture the normal behavior of each resource entity by creating entity profiles. This is achieved by running unsupervised learning algorithms on historical traces. Specifically, let $\mathcal{E}$ denote the set of resource entities that comprise the cloud system, and let $T$ denote the duration of the historical traces being considered. For each resource entity $e \in \mathcal{E}$, we assume at a given time $t$ there is a set of measurements captured by a vector $x^e_t = \{x^e_{1t}, x^e_{2t}, \ldots x^e_{n_et}\}$ that captures the state of resource entity $e$ at time $t$, where $n_e$ is the number of metrics being monitored for $e$. 
With the assumption that the entity \( e \) operates in normal conditions most of the time during time period \( T \), our goal is to use an unsupervised learning algorithm to create a model that best captures all the observations in \( x^e_t \) for all \( 1 \leq t \leq T \).

While there are many potential learning algorithms (e.g. Statistical Testing [53], Hidden Markov Model [44], One-Class Support Vector Machines [45]) we can use, and we do not restrict the algorithm to be used only for individual entities, in our implementation we resolve to use \( K \)-means clustering due to its simplicity and its intuitiveness. Specifically, we are given a set of historical measurements \( X^e = \{x^e_1, x^e_2, \ldots x^e_T\} \), each can be mapped to a \( n_e \)-dimensional space. We first normalize each measurement \( x^e_t = \{x^e_{1t}, x^e_{2t}, \ldots x^e_{n_et}\} \in X^e \) using standard z-score measures. Specifically, we define the mean \( \mu_i^e \) and standard deviation \( \sigma_i^e \) for dimension \( i \) (\( 1 \leq i \leq n_e \)) overall the samples in \( X^e \):

\[
\mu_i^e = \frac{1}{T} \sum_{j=1}^{T} x_{ij}^e, \quad \sigma_i^e = \sqrt{\frac{1}{T} \sum_{j=1}^{T} (x_{ij}^e - \mu_i^e)^2}
\]

We then create normalized measurement \( \bar{x}^e_t = \{\bar{x}^e_{1t}, \bar{x}^e_{2t}, \ldots \bar{x}^e_{n_et}\} \) as:

\[
\bar{x}_{it}^e = \frac{x_{it}^e - \mu_i^e}{\sigma_i^e}
\]

The \( K \)-means clustering algorithm aims at finding \( K \) centroids \( \{\bar{c}^e_1, \bar{c}^e_2, \ldots, \bar{c}^e_k\} \) in a \( n_e \)-dimensional feature space and assigning each measurement vector \( x^e_t \) to one of the centroids. Let \( N_k \) denote the measurement vectors assigned to centroid \( \bar{c}^e_k \), the goal of the \( K \)-means algorithm is to minimize the following similarity score:

\[
\text{score} = \sum_{i=1}^{k} \sum_{j \in N_k} ||\bar{c}^e_j - \bar{x}^e_t||^2
\]

where \( ||a - b|| \) denotes the Euclidian distance between two points \( a \) and \( b \) in the feature space. To determine the value of \( K \), we adopt a common approach, which is to pick the value of \( K \) such that increase the number of clusters from \( K \) to \( K + 1 \) does not achieve significant better gain in terms of minimizing the \( \text{score} \). Furthermore, if a cluster contains only a small number of elements \( \alpha \) (e.g. less than 3, or less than 1\%), we consider the cluster as an anomaly cluster. \( \alpha \) is a configurable parameter.

Once we have obtained the centroids, we need to specify a threshold \( \theta \) that is used to separate normal measurements from abnormal observations. Specifically, for each centroid
5.2 MonArch Anomaly Detection & Root Cause Analysis

We compute the mean $\mu^e_w$ and standard deviation $\sigma^e_w$ of Euclidean distances between the points in $N_w$ and the centroid in cluster $w$. At run time, for each newly arrived measurement $x^e_j$, we first normalize it by applying the normalization equation above to compute $\bar{x}^e_j$. Then we compute the anomaly intensity of the measurement as the ratio between the normalized Euclidean distance from $x^e_j$ to its nearest centroid $k_{\text{nearest}}$ and $\theta$:

$$I(x^e_j) = \frac{||x^e_j - \bar{c}_{k_{\text{nearest}}}|| - \mu^e_{k_{\text{nearest}}}}{\theta \cdot \sigma^e_{k_{\text{nearest}}}}$$

We say an observation $x^e_j$ is experiencing an anomaly if $I(x^e_j) > 1$ for an extended period of time $T_m$, i.e. $I(x^e_j) > 1$ for all $1 \leq j \leq T_m$.

If anomalies are detected, further analysis is conducted to better characterize each anomaly. We would like to understand how the measurement point deviates from the nearest centroid $k_{\text{nearest}}$ in each dimension. To do that, during the profiling phase, we compute the mean $\mu^e_{w,d}$ and standard deviation $\sigma^e_{w,d}$ of points $N_w$ in cluster $w$ for dimension $d$. Then at run time, if measurement point $\bar{x}^e_j$ is detected as an anomaly, we compute the dimensional anomaly intensity of the anomaly point as the ratio between normalized distance from $x^e_{dj}$ to the corresponding dimension $d$ of its nearest centroid $k_{\text{nearest},d}$ and $\bar{\theta}$, where $\bar{\theta}$ is a per dimension threshold:

$$I_d(x^e_{dj}) = \frac{||x^e_{dj} - \bar{c}^e_{k_{\text{nearest},d}}|| - \mu^e_{k_{\text{nearest},d}}}{\bar{\theta} \cdot \sigma^e_{k_{\text{nearest},d}}}$$

By evaluating the deviation pattern of the anomaly measurement point, we can understand which metrics are the main contributor to the anomalies and which direction (i.e. increasing or decreasing) the metric values are deviating. This could be helpful for the administrator when debugging anomalies, and it can also provide more hints to the root cause analysis module, which will be discussed in the next sub section.

5.2.4 Root Cause Analysis

The technique presented in the previous section allows us to identify resource entities that show anomalous behaviors. However, as mentioned previously, a single anomaly in the system can often cause multiple resource entities to misbehave. To help facilitate the diagnosis and troubleshooting process, we introduce an automatic root cause analysis technique that
5. ANALYTICS AND INTELLIGENCE IN MONARCH

can identify the entity that is most likely to be the source of observed anomalies.

The main challenge in root cause analysis is to identify the correlation between anomalous system observations. Due to the complex dependencies between resource entities in virtualized clouds, we need a way to capture the causal relationship between observed anomalies in the system based on our knowledge of the network / system configurations and their resource consumptions.

5.2.4.1 Anomaly Propagation Path

In order to perform root cause analysis, we need to understand how anomaly can propagate from entity to entity. We identify two types of anomaly propagation paths:

- **Vertical propagation**, which refers to performance interferences due to virtualization and VM collocations. There are three possible scenarios: (1) an anomaly of a physical machine (PM) causes a VM to behave abnormally, (2) an anomaly of a VM causes a PM to behave differently, and (3) two VMs collocated in the same PM may interfere with each other. Notice that even though we may consider the third scenario as a combination of the first two scenarios (i.e. a VM causes a PM to be behave differently, which in turn affects another VM), doing so requires that we consider the PM as an anomalous entity, which may not always be the case. For example, if a PM is constantly operating at near maximum outgoing bandwidth utilization, while one VM all of a sudden starts consuming significant bandwidth usage, which in turn cuts down the bandwidth utilization of another VM. In this case, the PM is behaving normally, while the anomaly of one VM still affects the other VM. This example illustrates that we have to consider the third scenario as a separate anomaly propagation scenario.

- **Horizontal propagation**, which refers to anomaly propagation due to network communications of the same application. For example, when a web application is experiencing a Denial-of-Service attack, the high request rate of the application tier may cause the database tier to behave erroneously.

In both cases, we require both the source and the destination of the anomaly propagation to show abnormal behavior simultaneously. While there are cases where anomalies may propagate without meeting this requirement, in general these cases are very rare, yet difficult to be analyzed automatically. Therefore, in these cases our system would report multiple, separate anomalies instead of reporting anomalies originating from a single source.
We provide an example to illustrate the concept of anomaly propagation paths (shown in figure 5.4). Consider a simple virtual compute cluster that consists of 3 physical machines (PM1, PM2 and PM3) which hosts two applications (App1 and App2), each of which consists of two VMs. The virtual links (VL1 and VL2) are embedded along the shortest paths between the physical machines that host the corresponding VMs. Suppose VM1 is the faulty entity which causes anomalous observations in VM2, VM3, PM1 and PM3 (highlighted in red). The anomaly in VM1 may manifest according to 2 anomaly propagation paths: VM1 → PM1 and VM1 → VM2 → PM2 → VM3, through a sequence of vertical and horizontal propagation directions.

We now consider these two types of propagations separately, and identify the conditions under which anomaly propagations may occur.

5.2.4.2 Vertical Propagation

In vertical propagation, the anomalies of entities in the same physical resource entity may propagate from one to another. We consider three sub-cases:

- A VM propagates anomalies to a PM: A VM may propagate anomalies to a PM through significant change in the resource usage pattern. Since the VM is hosted on top of PM, resource usage is reflected in the PM. However, since PM is shared by many VMs, it would require anomalies at a serious level in the VM to cause similar anomalies in the PM. For example, a VM may abnormally consume a large amount of memory, causing the PM to show low memory availability. To detect this condition,
we first identify a single resource for which both PM and VM show usage deviation from their normal conditions, then we check to make sure they have the same deviation direction.

• A PM propagates anomalies to a VM: This refers to the cases where the PM is either experiencing a hardware failure or the Operating System/management software of the PM is malfunctioning. In this case, both VM and PM must be identified as misbehaving. There are two possible cases: (1) The PM resource usage is much higher than the sum of VM utilization for a particular resource (e.g. CPU, memory disk, bandwidth); then we say most likely the PM is causing the VMs to misbehave. (2) If the PM resource usage is very low, causing all VMs to have resource usage much lower than their typical values, then we also say that it is likely the PM is causing the VMs to misbehave. An example of this is network adapter failure, where network readings for both PM and VMs are zero. As both VM and PM are misbehaving and VM bandwidth usage is lower than usual, we may deduce that the PM is causing the anomaly in the VM.

• Performance interference between VMs: This happens when there is a resource for which the PM is using maximum percentage or is at a performance interference point, while certain VMs show a significant increase in resource utilization, and some other VMs show a decrease in resource utilization. For example, when a physical network link is operating at near its maximum capacity, if one VM is consuming much higher than normal bandwidth usage, the other VMs on the same PM using the same link may experience low network performance due to resource contention.

In all other cases, we assume there is no vertical propagation, i.e. the anomalies in the collocating entities are independent and do not affect each other.

5.2.4.3 Horizontal Propagation

In horizontal propagation, a misbehaving entity can propagate its anomaly to its adjacent entities through network communications. However, given two misbehaving entities that communicate with each other, it is difficult to determine which entity is affecting the other one. Therefore, by default, we assume that all horizontal propagation paths are bi-directional. However, there are cases where we may clearly identify the propagation paths. Specifically, when two VMs communicate with each other, typically the network traffic follows a specific rate (i.e. one VM sends its request to another, while the other responds to the request). Even
though in practice it is hard to guess which VM is the sender, we can monitor the ratio of the bandwidth usage between VMs. If the ratio deviates significantly from normal behavior, we may infer that the VM whose outgoing traffic becomes significantly lower is the cause of the anomaly. In addition, if VM failure is detected (i.e. no monitoring data generated) in a cluster and other anomalies are detected at the same time, then we can consider the failure VM as a source of the anomaly propagation.

5.2.4.4 Analyzing Propagation Graphs

Once we have identified the anomaly propagation paths, we can construct a set of anomaly propagation graphs. Formally, an anomaly propagation graph is a direct graph that consists of a source entity $e$, and a graph $G^e = (V^e, E^e)$, where $V^e$ is a set of misbehaving entities that can be reached from $e$ through propagation paths, and $E^e$ are the set of links that captures all the propagation paths. A propagation graph may contain cycles. For example, a bidirectional horizontal propagation path forms a loop in the propagation graph.

At a certain time instance, there could be multiple clusters of anomalies that are caused by different root cause entities. To separate different resource entity clusters, we first filter the multi-layer resource graph to contain only the anomalous entities, then in the resulting graph, each connected graph is considered as an anomaly propagation cluster. One set of anomaly propagation graphs is generated for each anomaly propagation cluster, and the root cause analysis is conducted separately.

In an anomaly propagation cluster, for each entity as a propagation source, we construct an anomaly propagation graph that represents the possible propagation of anomalies in the system. Therefore, the root cause detection problem becomes finding a set of misbehaving entities whose propagation graphs best capture all the observed anomalies. We use a ranking method for selecting the best propagation graph.

For anomaly propagation graphs, we rank them and display them to the operator based on their scores. The score function of a propagation graph $G^e$ is the minimum total distance from the source anomaly entity to all other entities:

$$score(G^e) = \sum_{v^e, v^i} d_{v^e, v^i}$$

where $d_{v^e, v^i}$ denotes the shortest propagation distance from the source entity $v^e$ to entity $v^i$. When an entity $v^i$ in the propagation graph is not accessible by another entity $v^e$, the distance between these two entities is considered as infinite. By using this method, there are two other
cases we need to consider: 1) **Score Tie**: multiple best propagation graphs exist and their scores are not infinite; 2) **Infinite Score**: all the propagation graphs have infinite scores. The following is how we handle these two cases:

- **Score Tie**: This shows that there are multiple high potential root cause entities. An example is shown in Figure 5.5a where four VMs in a cluster are detected as anomalous with the communication pattern shown in the figure, and none of the VMs match the horizontal propagation pattern discussed, then VM1 and VM4 have the same score of 6. To consider this case, among the best propagation graphs we select the one whose source entity has the highest anomaly intensity. The anomaly intensity is defined by:

\[
    \text{intensity}(v^e) = \frac{||x^e_j - \bar{\mu}^e_{k\text{nearest}}||}{\bar{\sigma}^e_{k\text{nearest}}}
\]

- **Infinite Score**: This will happen when no entity can propagate to all other entities. For example, as shown in Figure 5.5b if there are multiple VM to VM vertical propagations, the propagation score can be infinite. In this case, we will choose the propagation graph where the source entity can access the most number of entities. We break ties using the method discussed above.

### 5.2.5 Anomaly Detection & Root Cause Analysis Implementation

The anomaly detection and root cause analysis algorithms are implemented in the MonArch system by leveraging Apache Spark framework to process multi-layer monitoring data. There
5.2 MonArch Anomaly Detection & Root Cause Analysis

are three modules to this implementation: 1) Offline entity behavior profiling: responsible for processing historical monitoring data to profile each resource entity for the purpose of capturing normal behavior; 2) Online anomaly detection: performs real-time processing of the new monitoring data and determines if each resource entity is in an anomalous state by comparing to the offline profiling results; 3) Root cause analysis: by analyzing anomalies and hints from the online anomaly detection module, this module tries to locate the resource entities that are causing the anomalies. The implementation is shown in Figure 5.6.

5.2.5.1 Offline Entity Behavior Profiling

In a multi-layer virtualized cloud environment, there are often entities in each layer. In this module, we use K-means unsupervised machine and the techniques discussed in Section
5. ANALYTICS AND INTELLIGENCE IN MONARCH

5.2.3 To profile each entity in all the layers.

To perform offline profiling, this module contains three steps:

1. **Data Formatting**: Obtain and format interested historical monitoring data to create multi-dimension points for machine learning training purposes. This module accesses the historical monitoring data that is stored in HDFS with Parquet format. Then it creates an RDD of arrays (similar to a 2-D array) for each entity, where each array represents data from a time slot and each entry in the array contains the value for one metric. Table 5.1 shows the metrics we use for each type of resource. We would like to emphasize the input degree, output degree, and PageRank metrics for the VM resource entities. These metrics are graphical metrics, and we extract them by using the graph algorithm mentioned in section 5.1. Notice that we use the GraphX library for graph processing, so the graph processing is scalable.

2. **Training**: Perform K-means algorithm to cluster the input points and discover usage patterns. We use MLlib library on top of Spark to perform scalable K-means training on the input data. One of the requirements of K-means is to predefine the value K, but since we do not know the number of clusters in the environment, we run K-means for K from 1 to 20 and select the best K using the method discussed in section 5.2.3. From our experience, usually the number of clusters is less than 10, thus setting a upper limit of 20 should be sufficient. Due to the random start nature of the K-means algorithm, for each K, we run the training multiple times to make sure it does not get stuck in the local minimum.

3. **Post-Processing**: Process the input points with training results to generate the final profiling model. The training data is used again to compute 1) mean, standard deviation, and max distance between each training point and its cluster centroids; 2) per cluster and per dimension distance mean, standard deviation, max positive, and max negative. Along with the resource ID, centroids information, and normalization parameters, the results for all the resource entities are written back to HDFS to allow access by the Online Anomaly Detection module.

5.2.5.2 **Online Anomaly Detection**

One of the main implementation objectives of this module is to design a scalable algorithm so that we can minimize the processing time and detect anomalies in real time. To minimize
5.2 MonArch Anomaly Detection & Root Cause Analysis

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM (Virtual Machine)</td>
<td>CPU utilization, disk IO read bytes, disk IO write bytes, network IO receive bandwidth, network IO transmit bandwidth, OpenFlow aggregate receive bandwidth, OpenFlow aggregate transmit bandwidth, input degree, output degree, PageRank</td>
</tr>
<tr>
<td>Virtual Link</td>
<td>bandwidth utilization</td>
</tr>
<tr>
<td>Physical Machine</td>
<td>average CPU utilization, disk IO read bytes, disk IO write bytes, virtual memory usage percentage, swap memory usage percentage</td>
</tr>
<tr>
<td>Physical Link</td>
<td>bandwidth utilization</td>
</tr>
</tbody>
</table>

Table 5.1: Profiling Metrics for Resource Entities

The processing time, our online anomaly detection algorithm is executed in parallel for all the resource entities of the same type. The algorithm obtains the latest trained model in each process job, so it can always use the latest model created by the Offline Entity Behavior Profiling module. We implemented this module on top of Spark Streaming to leverage its high-level APIs for scalable stream processing. During real-time detection, we use a sliding window (i.e. looking at the latest few minutes of data) to obtain more stable monitoring input data. By using a sliding window, we can also increase our evaluation frequency (i.e. by sliding the window with a small time difference). Due to the unknown distribution of each cluster of each resource entity, we have created two anomaly detection triggering parameters: 1) $I(x^e_j)$ (as shown in section 5.2.3), and 2) $I_{max}(x^e_j)$. We denote the max distance between the training points to the centroid in a cluster as $d_{max}$. Then we can define $I_{max}(x^e_j)$ as:

$$I_{max}(x^e_j) = \frac{||x^e_j - c^e_{knearest}||}{d_{max}}$$

For $I(x^e_j)$, we select $\theta$ to be 2 so we would be able to have an accuracy of around 95% in a Gaussian distribution. $I_{max}(x^e_j)$ is useful for detecting serious anomalies or non-Gaussian distribution. If $I_{max}(x^e_j) > 1$, then we know that this is a serious anomaly since this distance is higher than any training points we have seen. On the other hand, if $I_{max}(x^e_j) > I(x^e_j)$, then it provides a hint that the distribution of the cluster may not be Gaussian. With these two parameters, we decide the anomaly detection result using logic shown in Table 5.2. Then finally the anomalies and hints are written to the HDFS.
5. ANALYTICS AND INTELLIGENCE IN MONARCH

5.2.5.3 Root Cause Analysis

This module takes the results from the Online Anomaly Detection module and analyzes the root cause. A multi-layer graph is constructed from the monitoring data to allow analysis of the relationship between the anomaly entities. Then we perform root cause analysis using the algorithms discussed in Section 5.2.4. For computing the scores of propagation paths in a propagation graph, we use a back propagation iterative graph algorithm. As shown in Figure 5.7 assuming a propagation graph is constructed as shown in sub-figure 5.7a, we then reverse the direction of all the edges to create sub-figure 5.7b. Then in every iteration, each vertex in the graph sends a message to its neighbor about the distance information, and it also updates itself and keeps the lowest distance to all accessible vertices. Once this processing converges, each vertex would be able to calculate the propagation score for the propagation graph for which it acts as the source. For optimization purposes, in each iteration, each vertex can send out only the distance information that was updated in the last iteration. The benefit of using this graph algorithm is its scalability for supporting large graphs.

5.3 MonArch Use Cases

With MonArch’s monitoring and analytics capabilities, it enables a wide variety of use cases. In this section, we will introduce some use cases that are supported by MonArch.

5.3.1 Security Detection

Security is an important aspect of a cloud environment. Detecting and preventing security attacks is not a simple task. Traditional signature-based security attack detection is no longer as effective today due to attacks such as zero-day attacks, which are attacks that system has no previous knowledge about. In this case, anomaly-based security detection would be important and more effective when detecting unknown attacks.
5.3 MonArch Use Cases

The anomaly detection and root cause analysis capabilities provided by MonArch are useful for anomaly-based security detection. By using machine learning techniques, MonArch learns about the normal behaviors of the resources in the infrastructure. If an application running on the cloud infrastructure is attacked, its resources usage would most likely deviate from its normal state. Then MonArch’s anomaly detection can detect the anomaly and quickly report to the administrators. For example, when a DDoS attack is launched at a web server running in a VM, there would be significant increase in the VM's CPU and network utilization, and if the VM normally does not get as busy and have as much traffic, it would be considered an anomaly. Since the anomalies are detected in real time in MonArch, the
administrators or the management modules can quickly react and stop the attack or migrate the service.

Another example is Slowloris attack which tries to hijack a web server by opening a large number of connections and trying to keep the connections open by using a minimum amount of bandwidth. Since some web servers create a new thread for each connection and have a limit for the maximum number of threads running at the same time, Slowloris attack will occupy all the threads and the web server will not be able to handle requests from real users. This kind of low and slow attack is hard to detect since it does not take up a large amount of bandwidth like normal DDoS attacks. In MonArch, with the anomaly detection capability, we can detect this kind of attack by jointly looking at the CPU utilization and TCP connection number, duration and transmitted size. Assume a web server normally has 40 to 100 TCP connections and the CPU utilization is between 30 and 60. If more users are sending requests to this web server, the number of TCP connections and the CPU utilization will increase. However, when the server is attacked by Slowloris, the number of TCP connections and the connection duration will increase, while the CPU utilization will decrease. This unknown correlation between the CPU utilization and the connection metrics will be captured as an anomaly by MonArch’s real-time anomaly detection algorithm (based on K-means algorithm), and the administrator will be notified.

5.3.2 System Diagnosis

For system diagnosis, we focus on two types of diagnosis: 1) failure detection, and 2) performance diagnosis. In both cases, one of the objectives of our system is to identify problems and suggest the resource entity that is the root cause. The anomaly detection and root cause analysis algorithms in MonArch can be used for these types of diagnosis.

For failure detection, physical resources failure detection is straightforward, since there will be no monitoring data for that resource. For VM failures and software failures, the MonArch’s anomaly detection algorithm should be able to detect most cases since we also consider the graph structure of the virtual clusters. For example, if a database in a VM fails and cannot handle any request, the output degree of that VM would be much smaller than the normal case, since it does not send results back to the requesters.

Performance problems are more difficult to detect and to locate the root cause. In a cloud environment, one of the most common performance problems is performance interference. For example, when two VMs are sharing the same physical network link, there would be performance interference when the sum of their required bandwidth is higher than the link
5.3 MonArch Use Cases

capacity. Part of the root cause analysis algorithm in MonArch is designed to detect resource contention and thus can be used for this case. Other types of performance problems can be detected by MonArch if they cause utilization anomaly. For instance, if the disk in the physical machine has a problem and is slowing down disk IO, then the VMs, hosted on this physical machine, that have heavy disk usage would experience the same issue. In this case, MonArch can detect the problem and identify the physical server as the root cause. Last but not least, since MonArch supports monitoring of custom metrics from the application layer, users can choose to send application layer performance data to MonArch to allow detection of performance problems.

5.3.3 Resource Allocation

In a virtual cloud environment, resources allocation is vital to enable efficient and high performance applications, as well as facilitating resource management and infrastructure maintenance. MonArch can be used to provide useful resource allocation suggestions to help the management module make better allocation decisions. For example, if a user needs to boot up VMs with high network IO and low CPU utilization, the infrastructure scheduler can send a query to MonArch to find a physical server that has low network IO and high CPU utilization, and allocate the VMs there. By allocating in this way, application performance can be guaranteed while minimizing the overall physical resource usage. Another example is if a user needs to get a VM that requires high availability, the analysis can be run on the physical machine monitoring data to find the historical availability of each physical machine, and based on the results, suggest the physical machine with the highest availability.

5.3.4 Smart Room Monitoring

Since MonArch is extensible for collecting, storing, and analyzing new types of monitoring data, one of the use cases is to use MonArch for sensor data storage and analytics.

As new buildings are becoming more energy efficient and airtight, Indoor Air Quality (IAQ) has become an important health and safety concern for indoor environments. However, health and safety detection mechanisms are lacking in many indoor environments. In [64] and [51], we have demonstrated a real-time smart room monitoring system by integrating MonArch, a wireless ad-hoc sensor network that supports carbon dioxide monitoring, and the SAVI virtual customer-premises edge (vCPE).

In this system, the wireless ad-hoc sensor network contains three components: Sensor
nodes, Relay nodes, and Control room. The sensor nodes are responsible for wireless sensing of carbon dioxide to generate monitoring data and the Relay node handles packet forwarding of monitoring data towards the Control room. The sensor receiver (i.e. one of the Relay nodes) is connected to a vCPE, which in turn is connected to the SAVI Testbed. MonArch’s User Agent is installed in the vCPE to collect sensor data and send it to the streaming layer of the MonArch system. By using MonArch, users can leverage the MaaS functionalities, alarm mechanisms, and anomaly detection algorithm directly on the sensor data without modification of the MonArch system. This eliminates the need to setup storage and analytics system when users want to collect sensor data.

5.4 MonArch Portal

To provide the functionalities to MonArch users in a user-friendly way, we have created a web portal for interacting with the MonArch system and visualizing the results. In this section, we will demonstrate the web portal of MonArch running on top of the SAVI Testbed.

Figure 5.8 shows the home page of the MonArch portal. It shows some high-level statistics including number of VMs, number of anomalies, number of regions in the Testbed. The location of each region of the Testbed is also shown using a Google map. This web portal provides the following functionalities: 1) visualization of the multi-layer resources graphs that are created by the MonArch system; 2) providing a list of anomalies and presenting the root cause analysis results; and 3) supporting query and visualization of the monitoring data.

For visualization of the multi-layer resource graphs, users can see the multi-layer graph...
with the ability to select which layer and what type of resources to show. In Figure 5.9, we are showing the infrastructure administrator’s view of the virtual and application layer graph. In the figure, the black-border nodes are virtual machines and the blue-border nodes are applications running on top of virtual machines. The fill color of a VM node represents the CPU utilization (red denotes higher utilization). The links in this figure are OpenFlow-based links. They are generated by processing OpenFlow monitoring data. The color of each link represents its bandwidth utilization. In this figure, we can see many different kinds of applications running in the SAVI Testbed. Some examples of these applications include Hadoop clusters (mesh network with high link bandwidth usage), clusters with a load balancer (star topology), and network experiment clusters.

For viewing of anomalies and investigating the root causes, we have created the anomaly page to list the anomalies detected by the system at different times. The anomalies are grouped by detection time. By selecting the anomaly group, the portal provides the detailed anomaly information and suggests the root cause resource entity of these anomalies.

This web portal is implemented using the Tornado framework in the backend, and Bootstrap [8] and AngularJS [1] in the frontend.
Chapter 6

Evaluation

In this chapter, we provide the functional and performance evaluation of the MonArch system. The evaluation results are organized into two sections: 1) a system performance evaluation section that focuses on performance of the MonArch system; and 2) an anomaly detection and root cause analysis evaluation section that evaluates the analytics algorithms that are running in the MonArch system.

6.1 System Performance Evaluation

The MonArch system is evaluated in three areas: 1) scalability; 2) analytics performance; and 3) system overhead.

6.1.1 Collection and Analytics Scalability

Scalability is an important factor for a monitoring and analytics system. Scalability includes two aspects: 1) monitoring data collection and storage scalability: the number of resources in the infrastructure that the monitoring system is capable of monitoring; 2) data processing scalability: analytics efficiency (i.e. processing time) when processing different amount of monitoring data.

6.1.1.1 Monitoring Data Collection and Storage

To evaluate the monitoring scalability of the MonArch system, we study the data throughput of the system with different cluster sizes. In this experiment, a cluster of 6 VMs (1 master node and 5 worker nodes) is used for processing and storage purposes, and 1 VM is used for hosting 4 Kafka processes for message queueing. To emulate a large data center, we collect historical monitoring data trace from the SAVI Testbed CORE node, and replay the data into the compute cluster of this experiment at a much higher rate. Since the data collection rate in MonArch is mainly limited by the processing power and scalability of the compute cluster...
6.1 System Performance Evaluation

(used for message queuing, processing and storage), replaying the monitoring data at a high rate can very closely emulate a large data center where monitoring data is generated at a high rate by many Agents. In this experiment, we measure the data throughput when the MonArch system collects, formats, and organizes new monitoring data and stores them in HDFS using Parquet format.

The experiment result is shown in Figure 6.1. It can be seen from the figure that the MonArch system provides scalability for supporting high monitoring data throughput. As we increase the cluster size, the data throughput of the system scales up almost linearly. Since a typical physical server, running a Compute Agent, an OpenFlow Agent and a Physical Server Agent, on average generates about 6 to 7 messages per second (measured from the SAVI CORE node), we can estimate that by utilizing 15 worker CPU cores (and 2 CPU cores for the master) in a 6 VMs cluster, MonArch is capable of monitoring 60 to 69 physical servers and OpenFlow enabled Open vSwitchs, and around 840 to 966 VMs (assuming on average each physical server hosts 14 VMs). In this case, the overhead of the MonArch cluster is around 0.77% to 0.89% (calculated by dividing CPU cores used for monitoring purposes by the total number of physical cores that the system is capable of monitoring, assuming a similar environment as this experiment where each physical server contains 32 cores). In addition to data collection, the cluster can also be used for other processing and analytics tasks, and thus the overhead of the monitoring and analytics cluster for each analytics and storage job can be lower when the cluster operates under high utilization.

Although MonArch provides scalability for supporting a large number of resources, the data formatting and organizing processing efficiency (which is the performance bottleneck) can still be further improved. To minimize disk read for other analytic tasks, in the current implementation, the incoming monitoring data is organized and stored by their metric name under different folders. The performance of this processing task can be further optimized by improving the Spark and Spark SQL framework to improve parallelism.

6.1.1.2 Data Processing - Graph Construction

In Chapter 5 Section 5.1, we discussed MonArch’s multi-layer resource graph construction capability. In this section, the performance of this graph construction module will be evaluated focusing on the processing time and scalability. In our implementation, the graph construction algorithm not only constructs the multi-layer graph but also computes the average bandwidth utilization of the links and average CPU utilization of the physical and virtual machines. As a result, this evaluation will also show the scalability of the basic statistic
To evaluate the scalability and processing time of the graph construction algorithm, we would like to understand the following two aspects:

1. **Scalability**: How processing time changes as we linearly increase both the data size and the amount of compute resources. This shows us whether the algorithm can handle large data sets without sacrificing performance.

2. **Parallelism**: How processing time changes as we fix the data set size and increase the amount of compute resources. This can help us understand the potential minimum processing time of the algorithm, and find out the time bottleneck in the distributed processing flow.

In this experiment, we run the graph construction program in the production MonArch cluster in the SAVI Testbed CORE region using production data. The cluster contains 6 VMs (1 VM is used as the master node and 5 VMs are used as worker nodes). These VMs are large size flavor with 4 virtual CPUs, 8GB RAM, and 320GB hard disk. Spark master (standalone mode) and HDFS name node are installed in the master node, and Spark worker and HDFS data nodes are installed in the worker nodes. A production dataset is used for this experiment.
For this experiment, we use a partial or complete set of one day of historical monitoring data collected in the SAVI core region, and we launch the processing job with a different number of worker cores.

For the scalability evaluation, we linearly increase both the data size and the number of CPU cores, and measure the processing time. The results are shown in Figure 6.2. The x-axis in this figure represents the number of cores and the number of hours of data we use. For example, 8 in the x-axis means we are using 8 compute cores and 8 hours of monitoring data, and vice versa. As we can see in the graph, the processing time has a small increment when scaling up the data size and the cluster. That shows the algorithm is scalable, and it is capable of handling large data sets without a large impact on its processing time. It also demonstrates that the algorithm has very few computations that are done in a centralized way (i.e. in a single server). However, there is still a small increase in the total processing time as we scale up. By further investigating in the Spark job logs, we found that the major time increment is due to delay in starting the distributed job in the cluster. We believe that is caused by
the delay in the driver program, which is responsible for constructing the processing plan and submitting it to the cluster scheduler. In this experiment, as we scale up, we increase the number of hours of data. Since the monitoring data are stored in files in five-minutes slots, an increment in the data time period will result in an increment in the number of files, and thus increase time in the driver program to obtain the file schemas and construct the processing flow. As the 5 minutes monitoring data files in MonArch are combined into a daily file every day, we can execute the processing algorithm by using the daily files if we continue to scale out, and that would reduce the number of input files and therefore reduce the extra processing time in the driver program. (The next experiment verifies this conclusion.)

![Figure 6.3: Parallelism of Graph Construction Processing](image)

Figure 6.3 shows how the processing time changes when a fixed size input data is used and the number of computing cores varies. Each line in the figure represents the processing time with a certain input data size. From the figure, we can see that as we increase the number of CPU cores in the cluster, the processing time decreases because some processing tasks are done in parallel in different machines by different compute cores. This demonstrates the
parallelism of the graph construction processing algorithm. In all four input data sizes (i.e. 5 minutes, 1 hour, 10 hours and 1 day), we can achieve reasonably low processing times as we increase the cluster size. Although this graph construction algorithm provides parallelism, there is still a limit for the minimum processing time as we increase the cluster size. The minimum processing time is limited by the computations in the algorithm that are not parallel including setting up of clusters, constructing processing flow (i.e. directed acyclic graph (DAG) construction in Spark), shuffle actions that require all-to-all communication in the cluster, and the struggler task in each iteration. Future work can be done on these aspects to further reduce the minimum processing time of the graph construction algorithm.

In addition to parallelism, there are two other observations we made in this experiment:

1. **Number of Files and Processing Time**: The processing time for 1 day of monitoring data is shorter than the processing time for 10 hours of monitoring data. This is due to the different number of input files in these two cases. (One combined daily Parquet file that contains 24 hours of data is used for the one-day processing job, and 120 Parquet files each contains 5 minutes of data are used for the ten-hours processing job.) This shows that a large number of input files can reduce performance in the Spark framework. It also shows the benefit of files aggregation in the MonArch system.

2. **Number of Partitions and Processing Time**: When using 10 hours of data for graph construction, a ‘coalesce’ operation (i.e. decreasing the number of partitions in an RDD) is executed to reduce the processing time. When the number of partitions is much larger than the cluster size and each partition is relatively small, high scheduling overhead would be introduced and thus reduce the utilization of the computing cluster. Therefore, combining partitions is important when processing large number of small partition in the Spark framework.

### 6.1.2 Overhead Analysis

As a monitoring system, we would like to evaluate and understand the monitoring overhead of the system. In this section, we evaluate the monitoring overhead of the MonArch system focusing on CPU, memory, and the network overhead of the monitoring Agents.

For this experiment, a SuperMicro server is used to host the agents and virtual machines. The Compute Agent, OpenFlow Agent, and Physical Agent are deployed on this server. Then we measure the CPU, memory, and network utilization of the agent process by running one type of agent at a time. To better understand the overhead of agents under different
6. EVALUATION

conditions, we run the experiment with different numbers of VMs running on the physical machine.

Table 6.1, 6.2, and 6.3 show the average CPU, memory and network usage of each type of Agent. We can see that the CPU, memory, and network overhead incurred by the agents is minimum. For bandwidth usage, in the current implementation, the messages are serialized using JSON format. The reason for using JSON is due to its readability. We can future optimize the bandwidth usage by using more bandwidth efficient serialization methods such as Google Protocol Buffer [42] and Thrift [62].

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>number of VMs</th>
<th>CPU utilization (%)</th>
<th>Total CPU Cores</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Agent</td>
<td>4</td>
<td>0.80</td>
<td>32</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.47</td>
<td>32</td>
<td>0.046</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2.04</td>
<td>32</td>
<td>0.064</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.04</td>
<td>32</td>
<td>0.064</td>
</tr>
<tr>
<td>OpenFlow Agent</td>
<td>4</td>
<td>2.45</td>
<td>32</td>
<td>0.077</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.05</td>
<td>32</td>
<td>0.095</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3.49</td>
<td>32</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3.58</td>
<td>32</td>
<td>0.11</td>
</tr>
<tr>
<td>Physical Agent</td>
<td>4</td>
<td>0.85</td>
<td>32</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.87</td>
<td>32</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.87</td>
<td>32</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.87</td>
<td>32</td>
<td>0.027</td>
</tr>
</tbody>
</table>

Table 6.1: Agent Average CPU Utilization and Overhead

6.2 Anomaly Detection and Root Cause Analysis Evaluation

In this section, we will evaluate the anomaly detection and root cause analysis algorithm on the MonArch system against different use cases.
### 6.2 Anomaly Detection and Root Cause Analysis Evaluation

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>number of VMs</th>
<th>Memory utilization (MB)</th>
<th>Total Memory Size (MB)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Agent</td>
<td>4</td>
<td>42.8</td>
<td>49152</td>
<td>0.087</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>42.8</td>
<td>49152</td>
<td>0.087</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>42.8</td>
<td>49152</td>
<td>0.087</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>42.8</td>
<td>49152</td>
<td>0.087</td>
</tr>
<tr>
<td>OpenFlow Agent</td>
<td>4</td>
<td>45.7</td>
<td>49152</td>
<td>0.093</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>45.7</td>
<td>49152</td>
<td>0.093</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>83.6</td>
<td>49152</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>83.6</td>
<td>49152</td>
<td>0.17</td>
</tr>
<tr>
<td>Physical Agent</td>
<td>4</td>
<td>9.8</td>
<td>49152</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9.8</td>
<td>49152</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>19.6</td>
<td>49152</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>19.6</td>
<td>49152</td>
<td>0.04</td>
</tr>
</tbody>
</table>

**Table 6.2:** Agent Average Memory Utilization and Overhead

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>number of VMs</th>
<th>Network utilization Rx/Tx (Mbps)</th>
<th>Network Link Capacity (Mbps)</th>
<th>Overhead Rx/Tx (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Agent</td>
<td>4</td>
<td>0.017/0.032</td>
<td>10240</td>
<td>0.00017/0.00032</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.034/0.065</td>
<td>10240</td>
<td>0.00033/0.00063</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.046/0.089</td>
<td>10240</td>
<td>0.00045/0.00087</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.045/0.087</td>
<td>10240</td>
<td>0.000440.00085</td>
</tr>
<tr>
<td>OpenFlow Agent</td>
<td>4</td>
<td>0.010/0.76</td>
<td>10240</td>
<td>0.00010/0.00074</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.011/0.93</td>
<td>10240</td>
<td>0.00011/0.00070</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.012/1.1</td>
<td>10240</td>
<td>0.00012/0.011</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.013/1.1</td>
<td>10240</td>
<td>0.00012/0.011</td>
</tr>
<tr>
<td>Physical Agent</td>
<td>4</td>
<td>0.0075/0.11</td>
<td>10240</td>
<td>0.000074/0.0010</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0076/0.11</td>
<td>10240</td>
<td>0.000074/0.0011</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.0077/0.12</td>
<td>10240</td>
<td>0.000075/0.0011</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0075/0.11</td>
<td>10240</td>
<td>0.000074/0.0011</td>
</tr>
</tbody>
</table>

**Table 6.3:** Agent Average Network Receive/Transmit Bandwidth Utilization and Overhead
6. EVALUATION

6.2.1 DoS Attack and Resource Contention

Security attack and performance interference are two of the use cases discussed in Section 5.3. In this evaluation, we combine these two use cases and evaluate the anomaly detection and root cause analysis algorithms using the scenario discussed in the motivating example in Section 5.2.1.

This experiment is conducted in the SAVI Testbed’s CORE node. The setup is shown in Figure 6.4. Two tenants are each deploying a three-tiered web application in the cloud infrastructure where the web server and application server are co-located in one VM and the database is running in another VM. The two web servers are allocated in physical machine 1, and the two database servers are allocated in physical machine 2. The physical machines that are hosting the VMs are SuperMicro servers, each having two 2.6 GHz Xeon E5-2650 CPUs and up to 128GB of RAM. These servers are connected to a 10 Gigabit Ethernet OpenFlow enabled switch. The VMs that are used in this experiment are small VM instances (1 virtual CPU core, 2G memory, and 20G hard disk).

In the web servers, a simple Tornado-based web application is setup with one API that allows accessing of data stored in the database. PostgreSQL [24] is used as the database and is installed in the database servers. To simulate user activities, we created a user simulator that sends HTTP requests to the web server with a variable frequency. One instance of the user simulator is launched for each web application. As such, their load can be different at certain times.
6.2 Anomaly Detection and Root Cause Analysis Evaluation

We keep this setup and allow it to run for a few days. Then we run MonArch’s anomaly detection training algorithm on the historical monitoring data to learn the normal behavior of these two web applications. Real-time anomaly detection is then launched to detect any anomalies in these two applications.

For a DoS attack, we send large amount of large size UDP and TCP packets to the web server 1. For testing purposes, the attack runs for 1000 seconds.

First, we will take a look at the monitoring data and manually observe the pattern of the data. Figure 6.5 shows the CPU utilization and network traffic pattern of the four VMs before, during and after the attack. We can see that during the attack (between 22:35 and 22:43), CPU utilization and incoming traffic of the web server 1 VM increase significantly and the CPU utilization and network traffic of the other VMs decrease. On the web client side, the two user simulators also experience higher delay and request failure rate. In this case, all four VMs should be considered as anomalies since they are behaving abnormally compared to their normal behavior.

During the attack, the two physical machines also experienced network IO changes due to the effects from virtual machines. The incoming and outgoing bandwidth utilization of physical machine 1 and physical machine 2 is shown in Figure 6.6. We can see that the incoming traffic of physical machine 1 is very high (around 6.5 Gbps) and is much higher than the normal level. The outgoing traffic, on the other hand, is lower than in normal conditions. Physical machine 2 also experiences a decrease in network bandwidth utilization. The two physical servers should also be considered as behaving abnormally due to their abnormal network IO behavior.

Next, we will look at the real-time anomaly detection results from MonArch (shown in Figure 6.7). To better evaluate the accuracy of anomaly detection and the detection delay, we also plot the actual attack time in this figure, shown by the line ‘DoS Attack Period’ (the times stamps are recorded when we launch and end the attack). It can be seen that the anomaly detection system in MonArch is accurately detecting all six anomalies with relatively short delay. The start and end of the attack are both captured in the anomaly detection results. In this experiment, the first anomaly is detected for one VM with a 11 second delay, and all 6 anomalies are detected as anomalies with a delay of 2.2 minutes. In this setup, the monitoring data is collected by the MonArch Agents every minute, so detection delay is reasonable with respect to the monitoring data collection rate. We can further improve it by increasing monitoring frequency. Moreover, we keep a window of 5 minutes of new monitoring data and use the average value as input for anomaly detection. This explains the
6. EVALUATION

Figure 6.5: CPU and Network Utilization of VMs Belonging to the Two Applications

(a) Web Server 1 VM CPU and Network Utilization
(b) Database 1 VM CPU and Network Utilization
(c) Web Server 2 VM CPU and Network Utilization
(d) Database 2 VM CPU and Network Utilization

relatively higher delay between the end of the attack and the end of the anomaly alarms. The window size can be adjusted based on sensitivity and delay requirements.

In addition to raising the alarm, the anomaly detection algorithm in MonArch also provides hints to assist understanding of the anomalies. Table 6.4 shows the hints provided by the real-time anomaly algorithm during the attack detection period. In this graph, we can see that the algorithm has been able to accurately identify the metrics that are causing the anomalies. These hints also provide other information that we did not catch in the manual observation shown above including: 1) Graph structure changes: For web server 1, two of the hints listed in the table are increase of graph input degree and output degree. This tells us that more than normal amounts of other resources are connecting to the web server 1 VM; 2) For VM2 and VM3, only the OpenFlow network bandwidth metrics are detected as anomaly
6.2 Anomaly Detection and Root Cause Analysis Evaluation

but not the VM incoming and outgoing bandwidth that are measured by the hypervisor. This shows that the amount of network traffic these VMs are trying to send and receive are within normal range (although they are less than the mean values), but the actual network traffic passing through the network decreases significantly. This suggests that there are a lot of packet drops in the network.

To better understand the cause of the anomalies, we run the root cause analysis algorithm in MonArch. The propagation graph ranking result from the root cause analysis algorithm is shown in Table 6.5. The highest ranking propagation graph suggests that the anomaly in Web Server 1 VM causes anomalies in Physical Machine 1, Database 1 VM (which propagates the anomaly to Physical Machine 2), and Web Server 2 VM (which in turn causes anomaly in Database 2 VM). In this propagation graph, anomaly in Web Server 2 VM is caused by performance interference from Web Server 1 VM, and the anomalies in Database 1 VM and Database 2 VM are the results of horizontal propagation caused by communication with Web Server 1 VM and Web Server 2 VM respectively. This propagation suggestion aligns closely with our expectation and the real cause – the DoS attack that causes application 1 to behave abnormally and the bandwidth contention causes abnormal behavior in application 2. By using these root cause analysis results, administrators or management modules can look into the root cause node (i.e. the Web Server 1 VM in this case) and find out that the cause of the problem is due to large amounts of traffic generated by DoS attack from a certain source. Then the administrator can block the source to prevent attack.

To illustrate how the root cause analysis processes the anomaly results and suggests the best root cause propagation graph, we will go through the analysis procedures. First, the al-
Anomaly Detected

Figure 6.7: Attack Time Period and Real-Time Anomaly Detection Results - 1 denotes light anomaly, 2 denotes serious anomaly, and 3 denotes when the attack is running
### 6.2 Anomaly Detection and Root Cause Analysis Evaluation

<table>
<thead>
<tr>
<th>Resource</th>
<th>Anomaly Seriousness</th>
<th>Anomaly Dimensions (deviation direction, seriousness)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM 1 (Web Server 1)</td>
<td>Serious</td>
<td>CPU utilization (positive, serious), Disk Write (-, light), VM incoming bandwidth (+, serious), VM outgoing bandwidth (-, serious), OpenFlow Aggregated Incoming Bandwidth (+, serious), OpenFlow Aggregated Outgoing Bandwidth (-, serious), Input degree (+, serious), Output degree (+ serious)</td>
</tr>
<tr>
<td>VM 2 (Database 1)</td>
<td>Serious</td>
<td>CPU utilization (negative, serious), OpenFlow Aggregated Incoming Bandwidth (-, serious), OpenFlow Aggregated Outgoing Bandwidth (-, serious)</td>
</tr>
<tr>
<td>VM 3 (Web Server 2)</td>
<td>Serious</td>
<td>CPU utilization (negative, serious), OpenFlow Aggregated Incoming Bandwidth (-, serious), OpenFlow Aggregated Outgoing Bandwidth (-, serious)</td>
</tr>
<tr>
<td>VM 4 (Database 2)</td>
<td>Serious</td>
<td>CPU utilization (negative, serious), VM incoming bandwidth (-, serious), VM outgoing bandwidth (-, serious), OpenFlow Aggregated Incoming Bandwidth (-, serious), OpenFlow Aggregated Outgoing Bandwidth (-, serious)</td>
</tr>
<tr>
<td>Physical Machine 1</td>
<td>Serious</td>
<td>OpenFlow Aggregated Receiving Bandwidth (+, serious), OpenFlow Aggregated Transmitting Bandwidth (-, serious)</td>
</tr>
<tr>
<td>Physical Machine 2</td>
<td>Serious</td>
<td>OpenFlow Aggregated Receiving Bandwidth (-, light), OpenFlow Aggregated Transmitting Bandwidth (-, serious)</td>
</tr>
</tbody>
</table>

Table 6.4: DoS and Resource Contention Use Case Anomaly Detection Hints

Algorithm constructs a graph, as shown in Figure 6.8. Based on this graph, the program tries to separate connected graphs and works on each one of them individually. In this case, there is only one connected graph. Then the program determines the vertical and horizontal propagation direction. Since the incoming bandwidth utilization of the Physical Machine 1 reaches the interference point and the incoming traffic of the VM 1 (Web Server 1) increases while we measure the interference point in advance and set it as a threshold parameter in this program. We think the reason bandwidth interference happens at a lower value than the maximum bandwidth is due to the...
the incoming traffic of VM 3 (Web Server 2) decreases, the program determines that there is a vertical propagation caused by bandwidth contention. As a result, the propagations from VM 1 to VM 3 as well as from VM 1/VM 3 to Physical Machine 1 are set up. For Physical Machine 2, since there is no performance interference between VMs and no propagation from PM to VMs, the propagation is considered as from VM 2/VM 4 to Physical Machine 2. For horizontal communication propagation, based on our algorithm, it uses bidirectional propagation since no special case is found. From this propagation path analysis, the program creates the propagation graph shown in Figure 6.9. Next, we run the back propagation iterative algorithm on this graph and find the minimum total distance from each vertex to all the other vertices, and rank them to get the results in Table 6.5.

In the rest of this section, we will provide discussion regarding this use case:

1. **Support Other Kinds of Attack**: In this experiment, we used a DoS attack, so the first background traffic in the data center. Future work can be done to dynamically discover this interference point.
question to consider is whether this algorithm would be effective for DDoS attacks. In
the SAVI Testbed (and most other OpenFlow environments), the application infra-
structure often is deployed in its own subnet and layer 2 connectivity is provided. Since
the OpenFlow monitoring data is layer 2 based data, DDoS from outside the cloud
environment would not affect the internal virtual layer topology, and thus this method
would still apply. Other kinds of more sophisticated attacks such as SlowLoris attacks
would cause anomalies in the number of TCP connections and CPU utilizations, and
thus can be detected by this method by including NetFlow data. For attacks that do
not cause value changes in our existing metrics, we then must incorporate application
layer data into the anomaly detection and root cause analysis algorithm. Due to the
complex nature of application data, root cause analysis with application layer data is
not within the scope of this thesis and it will be one of our future area of work.

2. Environment Variation: For environment variation, we consider the effects of the
other VMs that are running on the same physical machines (Physical Server 1 and
Physical Server 2). There are two other possible anomaly detection results for the
physical servers due to effects of co-located VMs. These are: 1) Physical Server 2 is
not considered as anomalous: this case would happen if Physical Server 2 normally
has big fluctuation in network and the changes caused by the two database VMs do not
cause abnormal behavior in Physical Server 2; 2) Physical Server 1 is not considered
as anomalous: this is when the incoming bandwidth utilization of Physical Server 1 is
normally near the maximum but does not causes anomaly. In both of these situations,
the root cause analysis algorithm can still find the correct propagation path.
6. EVALUATION

6.2.2 Unexpected Access

To improve security in a cloud environment, administrators often define accessing policy to prevent communication between VMs and physical servers; however, sometime insider security attacks can happen without violating the policies. In this section, we will evaluate MonArch’s anomaly detection capability by simulating a low-profile password attack from an insider node.

![Figure 6.10: Attack Time Period and Real Time Anomaly Detection Results](image)

**Figure 6.10: Attack Time Period and Real Time Anomaly Detection Results** - 1 denotes light anomaly, 2 denotes serious anomaly, and 3 denotes when the attack is running

In this experiment, two VMs are used to host a three-tiered web application similar to the previous experiment. (VM1 hosts web and application server, and VM2 hosts database.) Then we run a password attack at the database VM from another VM in the same subnet. (We assume that this VM is already compromised.) Figure 6.10 shows the real-time anomaly detection result before, during, and after we launched the attack. We can see that an anomaly is detected after the attack is launched with a relatively short delay. To understand how the anomaly detection mechanism detects this attack. We can look at the hints provided by the anomaly detection algorithm shown in Table 6.6.

80
### Table 6.6: Unexpected Access Use Case Anomaly Detection Hints

As shown in the table, the anomaly detection algorithm detects an unexpected increase in input and output degree in the database server VM, and thus is raising the alarm. Since the database VM normally is only being accessed by the web server VM, the anomaly detection system learns about this normal condition during profiling stage. Then when an unexpected connection is established from the attacker VM to the database VM, the anomaly detection algorithm is able to identify the anomalous activity. In this case, the benefit is that the administrators do not need to define the exact communication pattern of the VMs. Instead, MonArch’s anomaly detection mechanism is able to learn from the historical data and extract this communication pattern. From this detection result, we can also see that even though some attacks may not create large amounts of traffic or affect CPU utilization of the target VM, MonArch anomaly detection can still detect it if it does not follow the normal connection and communication pattern.
Chapter 7

Future Work and Conclusion

7.1 Future Work

For future work, there are three main areas we can continue to improve: For anomaly detection and root cause analysis, one extension is to include application level data in the root cause analysis algorithm. The root cause analysis effectiveness can be further improved if we can jointly analyze application level information. Since MonArch also provides monitoring capability for application data monitoring, the main challenge is to create a generic mechanism that is able to capture complexity and diversity of the application level monitoring data and incorporate it into the anomaly detection and root cause analysis. Another area we can work on is to improve analytics and processing performance by optimizing the analytics algorithms and the Spark processing framework. The main objective for this work is to identify the performance bottlenecks and address them to improve the efficiency and parallelism of the analytics algorithm in MonArch. Last but not least, other kinds of unsupervised machine learning algorithms can be implemented in the MonArch system and evaluated against the existing methods. By providing different kinds of anomaly detection algorithms, we can potentially improve the anomaly detection accuracy of the system.

7.2 Conclusion

In this thesis, we presented a monitoring and analytics system, MonArch, for virtualized heterogeneous cloud environment. MonArch provides a data collection, storage, and processing platform, as well as analytics algorithms including anomaly detection, root cause analysis, and graph processing. A wide range of use cases such as security, resource allocation, and system diagnosis are supported by the MonArch system. MonArch’s scalability and efficiency is demonstrated in the evaluation section.
References

REFERENCES

[27] setuptools. https://pypi.python.org/pypi/setuptools [34]
[28] stevedore. https://pypi.python.org/pypi/stevedore/ [34]


[33] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37(5):164–177, 2003. [1,16]


[40] Benoit Claise. **Cisco systems NetFlow services export version 9.** 2004. [16]

[41] Chunfeng Cui, Hui Deng, Deutsche Telekom, Uwe Michel, and Herbert Damker. **Network Functions Virtualisation.** In *SDN and OpenFlow World Congress 2012*, 2012. [1]


[59] Hiep Nguyen et al. FChain: Toward black-box online fault localization for cloud systems. In IEEE ICDCS, 2013. 19


REFERENCES


[74] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, page 2. ACM, 2013. [34]


[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and