A Defense in Depth Approach for Software as a Service Cloud Applications

Shabir Abdul Samadh

School of Computer Science
McGill University
Montreal, Canada

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of
Master of Science

August 2018

I would like to dedicate this thesis to my beloved dad, who dreamt for me when I knew not how to…

Acknowledgements

Notes of sincere gratitude to:

Prof. Muthucumaru Maheswaran, *for his timely advices and lucid guidance*
for his absolute patience with me and the confidence he confided in me
(My supervisor throughout this research)

Team CIENA, *for their collaboration, timely feedback and their generous support*
(Engineering team from CIENA who were part of this *MiTACS* collaboration)

Heather McShane, *for her genuine concern, love and understanding*
for the many things I've learnt from her and for making any conversation free & easy
(Catalyst-in-Chief, McGill Sustainability Systems Initiative; where I worked part-time)

ANRL Group, *for all the fun they brought to the research atmosphere*
(The Advanced Networks Research Lab of McGill University)

Montreal, *for being such an international city and for connecting me to many wonderful people*
(Especially to my dear friend Akash Singh for sharing this journey with his silly super-hero stories)

Pubudu & Keheliya [aiyya], *for putting up with all my intricacies &*
for the untold episodes of excitement with the McGill New Rez across the street
(My two wonderfully tolerant room-mates for the past 2 years)

Agilen, *for the immense support he brought and the concern he showed*
(Childhood friend from my high school [back in Sri Lanka] living in Montreal)

Manna Nana & Family, *for their love and unwavering involvement in my well-being*
(My mom's cousin living in Toronto whose family made Canada feel like home to me)

Siblings & Brothers-In-Law, *for being the reason, strength and force that keeps me striving*
(Especially to my beloved brother Jameel Hassan for owning responsibility in my absence &
my brothers-in-law for giving me confidence merely by their presence)

Mom, *for the strength she is and the love she brings!*

Dad, *for who he was!*

God, *for who HE is...*

ABSTRACT

Cloud based deployments have been moving towards a positive trend over the past decade. A variety of tooling has been introduced and the deployment procedure itself has been immensely eased. Thus, many large scale businesses are considering a move towards a fully cloud based solution. This enables them to take complete advantage of the data intensive computing facilities of the cloud. Nevertheless, the decision to move towards a complete cloud-deployment does pose some important concerns; the top priority concern being *data security*. The current service-levels *(in terms of data security)* provided by the cloud providers are proven to be strong. However, the threat model addressed by such SLAs are related to protection of the cloud Virtual Machine(VM)s from external access and against cross-communication between adjacent VMs. Yet, an orthogonal security concern is ensuring security over internal threats. This requires a defense in depth mechanism enforced on the processes internal to the deployment. The threat model for such internal-threats is derived from the bogus behavior of applications in the deployment. The study that follows produces a solution that is immune to such threats in scheduled events with large volumes of sensitive tenant information in the cloud.

In this thesis we produce a proof-of-concept framework that provides dedicated and isolated data processing pipelines for different tenant data in cloud deployments. The isolation is achieved via containerizing each application in the processing pipeline. Communication between these containerized applications is enabled by stitching them together via a virtual switch and dynamically publishing flow-control rules as per SDN specifications. The framework is made generic such that the addition of a new work-flow is made straight-forward. We also define the container environment to be generic such that it enables monitoring of the work-flow status. We evaluate the robustness of the framework with various workloads and produce comparative results between containerized & non-containerized pipelines. The results show that these defense features can be implemented with a minimal runtime overhead of $< 5\%$. We also see from our results that the framework imposes an additional resource overhead of $\approx 7.5\%$ CPU and $\approx 8.25\%$ memory.

ABRÉGÉ

Les déploiements basés sur l'infonuagique ont la cote ces dernières années, et de nombreux outils ont été introduits pour faciliter ces procédures de déploiement. Ainsi, plusieurs entreprises à grande échelle songent à changer pour des solutions complètement basées sur l'infonuagique. Cela leur permet de tirer pleinement parti des capacités de calcul supérieures de l'infonuagique quant au traitement de volumes élevés de données. Toutefois, la décision de faire la transition complète vers le déploiement en infonuagique soulève nombre de questionnements importants, le plus prioritaire étant celui de la sécurité des données. Les accords de service présentement offerts sur le marché fournissent déjà des garanties de sécurité assez fortes. Néanmoins, le modèle de menace actuel couvert par ces accords concerne les machines virtuelles du nuage et porte seulement sur les accès externes et les communications croisées entre machines adjacentes. Pourtant, un important problème de sécurité orthogonal pour les fournisseurs de service, quant il s'agit d'infonuagique, est de garantir la sécurité des systèmes contre les menaces internes. Cela requiert un mécanisme approfondi de défense appliqué au niveau du processus interne de déploiement. Le modèle de menace pour de tels risques internes est issu de comportements fictifs d'applications lors de leur déploiement. L'étude qui suit propose une solution qui serait imperméable à de telles menaces et pratiques erronées associées lors de la planification d'événements avec de grandes quantités de données client sensibles sur une plateforme infonuagique.

Dans cette étude, on fournit une validation de concept pour un cadre logiciel (framework) qui fournit des pipelines dédiés et isolés afin de traiter des données pour différents clients séparés lors de déploiements en infonuagique. L'isolement est obtenu grâce à la conteneurisation de chaque programme s'exécutant dans le pipeline. La communication entre ces programmes est permise par le rattachement entre eux via un commutateur virtuel et la publication dynamique de leurs règles de régulation selon les spécifications SDN. Le cadre logiciel est conçu de façon générique afin que le rajout de nouvelles règles de régulation soit simple. On définit aussi l'environnement du contenant de façon générique afin qu'il soit possible de surveiller le déroulement des opérations. On mesure la résilience du cadre logiciel obtenu en le testant avec différentes charges de travail afin de produire des résultats qu'on peut utiliser pour comparer les pipelines conteneurisés et ceux qui ne le sont pas. Les résultats montrent que ces caractéristiques défensives peuvent être implémentées avec une faible surcharge d'exécution de moins de $< 5\%$. On voit aussi, à travers nos résultats, que le cadre logiciel requiert une surcharge de ressources de $\approx 7.5\%$ sur le processeur et de $\approx 8.25\%$ sur la mémoire.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Cloud computing offers many benefits. It offers computing power as a utility for which customers can pay according to their usage. It has transformed the IT industry giving rise to easily deployable applications [9]. Developers need not worry about large capital investment to get their innovative ideas up and running. They can minimize the risk of taking on ideas that do not attract enough popularity. Moreover, companies can scale their computing needs as the tasks scale depending on the volume of the business.

Even with such clear benefits, enterprises have major concerns when transitioning to a fully cloud-based solution. This is mainly attributed to the fact that cloud computing brings many new security threats into play while the enterprises rely on the cloud provider for security [50]. This is evident from the surveys conducted by leading business intelligence research organizations like Gartner, IDC and Unisys. Those results show security problems are an important factor when businesses select cloud-computing options [1][21][39][44].

Security concerns in cloud deployments emerge due to many factors. These factors are not only technical concerns but also include non-technical factors, such as lack of coherent management policies and ignorant deployment practices [33]. Moreover, these security concerns can be categorized into two different types of threat models: *external* and *internal* to the cloud deployment. External threats are initiated from outside the deployment boundary itself while internal threats emerge from within the system. Academic and industrial researches have focused on security concerns over the years. Nevertheless, there still seems to be an important void in safeguards against threats emerging from **within** the cloud-deployment. Also, it is interesting to observe that security promises against threats that arise due to internal negligence have been omitted in SLAs of leading cloud providers. Hence, an important orthogonal perspective to security in the cloud is to ensure that the applications deployed are themselves benign and do not act against the SaaS provider's tenants.

When considering the internal threats, one of the key concerns is that the myriad of in-house applications have a tendency to reach production with minor bugs. Such threats are specific to the SaaS deployment model. These threats are of utmost concern to tenants who entrust service-providers with their data for analytical insight. Knowing that their data is being processed in a common pool of resources amongst others' data raises a red flag. Hence, it is essential for SaaS providers to gain the trust and confidence of their tenants by showing provable security and isolation mechanisms when hosting tenants. An easy way for SaaS providers to express such assurance is by means of the SLAs of the cloud-infrastructure provider. However, such SLAs do not strongly cover the internal threat model. Besides, tenants who trust SaaS providers, expect an added layer of defense for their data on top of the infrastructure-provider's SLAs. They want to know what additional measures are taken to ensure that their data is not leaked in any fashion during the computational process. Thus, arises the need for mechanisms that provide defense-in-depth to multi-tenanted data processing pipelines. Defense-in-depth measures provide multiple layers of security by means of introducing redundancy [48]. Thus, SaaS providers can present their deployment to be safe even when injected with spurious applications; showing immunity against such misbehaving applications resulting from ignorant or malicious programming practices. Such a *defense in depth* framework would provide the extra level of trust expected by the tenants.

As a start we specifically focus on *passive internal threats* that arise from negligent practices in the development and deployment of cloud applications. We mention "passive" because there are also *active threats* that contribute to the above threat model. The components of a passive threat vector are a result of negligent and unintentional practices whereas an active threat is a result of internal activity with malicious intent. Thus, for the scope of this study we only address remedial strategies for the passive internal threat model.

To of address this *passive internal threat* model within a SaaS deployment, we define a defense-in-depth framework. We build our framework using isolation via containerization. The container is the encapsulation unit for all cloud deployable applications. We use the container boundary as the point of scrutiny for the applications' activity. We control inter-application communication using *Open-Flow* flow-controls published to a virtual switch which interconnects all applications. The complete flow of the framework is controlled and managed via a manager module. We test the framework using representative data processing pipeline examples. The framework is designed such that it allows the addition of new workflows with minimal overhead. The architecture also supports easy adaptation of more administrative features and monitoring capabilities for future developments.

The major contributions of this thesis can be summarized to:

- An in-depth enumeration and discussion of the threat vectors attributed to a passive internal threat model
- A proof of concept implementation of a framework to provide defense-in-depth mechanisms to cloud deployments with passive internal threats
- The use of SDN controls as means for active monitoring and control over data processing pipelines
- The conceptual basis for using containerization as means to relieve application developers from handling security and moving such checks to a separate isolation boundary

The rest of this thesis is structured as follows: Chapter 2 provides a discussion of related work. Chapter 3 details the design requirements of the framework as presented by actual industry use-cases and requirements. Chapter 4 provides an analysis of the different threat vectors related to the problem addressed and how the framework handles them. Chapter 5 is a detailed explanation of the design and architecture of our proposed framework. Chapter 6 shows the experimental results of the framework in terms of overhead incurred. Chapter 7 summarizes the complete study and provides pointers for future work.

Chapter 2

Related Work

2.1 Overview

This chapter discusses prior work in a variety of topics that are related to the work presented in this thesis. These topics can be broken down into generic cloud security, SDN & NFV based security in cloud and security of containerized systems.

2.2 Cloud Security

Since the idea of cloud based deployment became ubiquitous, a lot of effort has been put towards the security problems. Security for cloud-deployments is held at a higher level of importance *(compared to in-house infrastructure)* given that the ownership of the hosting infrastructure moves outside a company's control. This view is shared among many from the academia [23], industry [38] and governments [15, 31]. According to a report by Gartner, cloud computing has unique attributes that require risk assessment in multiple areas such as: data integrity, recovery and privacy [14]. Two important points raised by that report is the importance of customers verifying with their vendors about: *privileged user access* and *data segregation*. In light of the requirement for proper risk-assessment, Yang et al. [50] produced an assessment model for risk in cloud computing taking into consideration the variability of the deployment setup. Their model is based on an information entropy scheme.

Even-though there are many studies in academia related to cloud security, most of them are continuations of traditional systems security research. These include studies on web-security [12, 45], data outsourcing and assurance [13, 22] and security of virtual machines [37, 47]. Very few could be narrowed down to have exclusive focus on cloud security [16]. This is expected given that topics related to cloud security expand on systems security research. This research can be related to specific components of the cloud computing model. For example, Mishra et al. [32] describe how the concepts of virtualization lead to realizing a cloud-deployment model and explains how threats also originate at the virtualization layer. They establish that the hypervisor and virtual-network-layer introduce an attack surface leading to major cloud vulnerabilities. In [35], they use Amazon's EC2 service as a case study to show that it is possible to deduce a map of the internal cloud-infrastructure; instantiate VMs until they are co-resident with a target and initiate cross-VM attacks across the virtualization layer to extract information. These studies specifically target the cloud VM security.

A comprehensive summary of the differences between security threats exclusive to cloud computing and not exclusive are enumerated by Chen et al [16]. According to them, the primary new threat vector to cloud deployments is the introduction of shared resource environments. Another key issue discussed by them is *reputation fate-sharing*: while cloud users can benefit from the concentration of security expertise of major cloud providers, a single subverter can disrupt many users. Based on their findings they produce a comprehensive discussion on the unique aspects of the cloud threat model. They establish, existence of competing businesses within the same eco-system and a longer trust chain with different deployment architectures as important aspects of this threat model. They also put forth that the security needs of businesses may vary depending on the business use case. This sets up an important argument for our study where the source of the problem is the business use case itself. We specifically study scenarios where the threat surface itself is formulated by undesirable coding practices of in-house and third-party developers.

The focus of our study is to develop efficient mechanisms to protect SaaS cloud-deployments from misbehaving applications. Our framework considers a use case where a SaaS provider has many tenants with large volumes of data for processing and analytics. Thus, the expectation for security is not only from the SaaS provider but also from its tenants in order to have confidence about how their data is handled. The threat surface in our study is internal to the SaaS provider rather than external. Most literature in academia and the industry focus on either external attacks or at-rest data protection.

2.3 SDN & NFV based Security in the Cloud

2.3.1 SDN & NFV Overview

Software Defined Networking (SDN) enables programmatic configuration of network topologies on top of underlying hardware infrastructure. It facilitates dynamic re-structuring and management of networks at the software plane. This is different from traditional networks where the setup is static and decentralized [49]. With Network Function Virtualization (NFV), functionalities traditionally implemented on dedicated hardware are written in ”software” and are deployed on standard hardware [34]. NFV by definition is the practice of virtualizing all the physical network functions into *virtual network functions* (VNF) to form a service that is required to perform a specific network operation [43, 42, 41]. Thus, NFV can be used to deploy network functions on various types of network infrastructure using commercial off-the-shelf hardware (COTS). Traditionally, network functions are fully coupled with vendor specific hardware. During the time of scaling the network, adding new network functions and services becomes inconvenient and expensive. Provisioning them is also cumbersome with a lot of dynamic network traffic and changing requirements [29]. However, the NFV approach mitigates these problems making network function deployment easy and fast [42, 41]. NFV brings the benefit of enabling SDN-controller virtualization; thus allowing dynamic mobility of SDN-controllers. On the other hand, SDN is advantageous to NFV in that, it enables dynamic network connectivity by programming the network to be optimal depending on network traffic monitoring and analysis [42].

A recent trend with cloud deployments is to use a combination of SDN and NFV techniques to provide security to the infrastructure. Security functionality, traditionally implemented on commodity hardware are separated out as VNFs and are deployed on the cloud. Thus, they can now be deployed in any hardware *(with proper virtualization managers)* used in the infrastructure. Besides, they bring in the advantage of quick and easy scaling without any overhead in terms of hardware requirements. In addition, SDN controllers can enable dynamic wiring of VNFs *(which is called Virtual Function Chaining (VFC))* to institute the necessary security functions in the network. We draw a parallel between this strategy and our framework in terms of the design and approach to instigate security. Some of the recent developments from the literature related to NFV addresses certain security concerns discussed in our threat-model. Thus, in the sections that follow, we discuss such related work from this domain. We can categorize the literature in the *NFV-Infrastructure* (NFVI) area into two separate groups based on its focus:

1. How secure is NFVI, what vulnerabilities does it pose and how to mitigate them

2. Using NFVI as means to provide dynamic security functionality to deployments at scale

We discuss the work done in each of these areas below.

2.3.2 NFV-Infrastructure Security

Lal et al. in [29] and Yang et. al in [51] discuss about the vulnerabilities posed by NFVI and how to secure it. They mention that the threats related to VNF are a combination of threats on physical networking and on virtualization technologies. They also list out the causes for such security threats in a NFV infrastructure which includes: improper isolation of the virtual functions, mis-management in terms of setting up virtual network components, independent mobility of the VNFs outside legal boundaries, forced-amplification of VNFs to exhaust resources and malicious insiders. They also outline the best-practices in terms of securing any NFV infrastructure. According to them, keeping the hypervisor up-to date (with all security patches) and enabling proper security for the SDN based virtual network is very important. They also suggest other measures such as image signing, zoning of network traffic into isolated pools, using default kernel support features *(eg: SELinux)* and encryption. While their work addresses security of virtualized *network functions*, we specifically focus on securing cloud applications. Nevertheless, some of the suggestions proposed by them are applicable when considering how our framework uses SDN to interconnect the containerized applications.

In [28], authors discuss the problem of communication between VNFs not being secured other than by the common protocols like SSL/TLS used by applications. They argue that important network data exchanged between VNFs that are not protected by SSL/TLS are still vulnerable to eavesdropping and hence can disclose valuable information. Thus, they propose a design where a VPN server is integrated into the NFVI layer of the host and managers encryption of the traffic between different VNFs. Their results shows that the encryption overhead is significant ($\approx$96%) and hence, not ready for use in an actual industry setup. Moreover, their threat model is specific to malicious *external adversary* unlike ours which targets *passive internal threats*.

2.3.3 NFV as means for Cloud Security

There is considerable amount of work done using NFVI as the means to provide security functionality to deployments. Jalalpour et al. have implemented a container based orchestration mechanism [26] to dynamically deploy network security functions to edge points of content delivery networks (CDN). They do this by binding the VNFs to an *open-virtual-switch* (OVS) and routing requests to the CDN via the OVS. Their work specifically targets network security functionality at edge-networks. Pavlidis et al. have produced a vantage-point network analyzer by leveraging docker containers as the virtualization unit in their work [34]. They have followed a similar technique to ours where docker-containers are used to encapsulate the analyzer functions. However, their architecture is not for active monitoring of applications pipelines[1]; rather it provides analytics of the network on request on set points of the infrastructure.

[1]We define the term pipeline in Chapter 3.

Cziva et al. [17] propose the Glasgow Network Functions (GLANF) framework to manage the life-cycle of container based VNFs. They use a similar approach to what we have done in terms of having the VNFs connected over an OVS and controlling communication by publishing Open Flow (OF) rules. They go on to produce a concrete application of their GLANF framework [18] by implementing it for network function chaining on edge devices. Marco et al. [19] present a case for using containers as the virtualization unit for security function chaining instead of other virtualization means such as VMs. Their work is a motivated by the lack of proper specifications for container based deployments in the NFV domain. Their work proposes a concrete structure for light-weight container based VNF infrastructures based on which solutions (similar to the ones in [17, 26, 34]) to the orchestration problem can be implemented.

2.3.4 NFV based Security - A Comparative Summary

We compare the above studies in the NFV area to illustrate how our framework also uses containers as the encapsulation unit to provide security functions. Moreover, our framework also takes advantage of the SDN stack to dynamically re-wire the container units as with VNF chaining. However, these works specifically address the NFV arena and the encapsulation unit is not the *security surface* but is the means to deploy varying network security functions. In our work, the main motivation behind using containers is to provide a surface for security, outside the application boundary. The proposed frameworks from the literature follows a model where the applications handling security are containerized. Then, these containerized applications filter the network traffic either at the edge (of the deployment itself) or in between end-points. In contrast, we propose the container surface itself to be used as the point of security while placing the business applications to run inside them. Moreover, the work by Anderson et al. [5] which evaluates the performance of container based VNFs also provides a case supporting our design. Their research concludes that docker containers are an efficient technology to enhance dynamic deploy-ability of network functions in NFV environments. Their results shows that the overhead incurred in containerizing applications is acceptable against the benefits achieved.

2.4 Security Features in Container Orchestration Platforms

The framework we propose is based on containerization for isolation and it uses *Linux container* concepts to introduce the defensive boundary around the threat surface (i.e. the applications). Thus, in addition to the above approaches we also compare our design against existing industry applications. Some of the most common container orchestration platforms in industry are: *Google's Kubernetes engine*, *Docker's Swarm mode*, *Amazon's Elastic container service* and *Microsoft's Azure container instances*. The primary purpose and motivation for such orchestration systems is to make the container deployment and management process easy & fast. They focus largely on optimizing the tasks of Dev-ops in cloud deployments.

Nevertheless, some of these systems have support for features[2] that enable control over the communication between containers. Kubernetes had introduced this feature just last year as *network policies for defense in depth* [25]. This requires for Kubernetes to be configured with one of the few Kubernetes network plugins that has support for this feature. They prevent inter-container communication by manipulating container IP tables. Docker's cluster management system *Docker Swarm* [20] also supports a similar feature by means of virtual bridges that connect multiple containers. Containers can be segregated into different networks by creating various overlay networks. Each overlay network is managed by policies published to a virtual bridge that is specific to that network. Thus, docker's way for restricting communication is to introduce a new overlay via a new bridge. Amazon's Elastic container service (ECS) platform introduced a feature called *task networking* to overcome some of the scalability issues with docker's default networking modes [40]. AWS has a logical networking component called the *elastic network interface (ENI)* which is attached to each AWS instance. With task-networking they extend this ENI into running containers within an AWS instance. An ECS agent invokes a chain of network plugins to ensure that the ENI is configured appropriately in the container's network namespace.

While, each of these container runtimes provide features to achieve restricted inter-container communication their primary purpose is container-orchestration. Thus, their design decisions are somewhat agnostic to the idea of pipelines of workflows that process data from different tenants. This is the primary industry use case that we target in our solution. Besides none of the above implementations follow a purely SDN based approach to control communication. SDN allows dynamic changes to network policies at a single point of control *(i.e. the virtual bridge)* and enables continuous auditing/analytics capabilities. In contrast, the approach taken by systems like *Kubernetes* is a static declaration of policies for communication. Moreover, SDN based controls support multi-layer application-topology interleaving and pipeline handling. In addition, the systems discussed earlier come as a full package for container orchestration with third-party feature support. Thus, they are not the ideal choice for businesses seeking the minimal setup requirement to address the threat model we address in this study. The framework we propose addresses such industry requirements and provides a solution for the threat model discussed in chapter 4.

[2]Some introduced these features more recently while the others have had them for sometime.

Chapter 3

Design Requirements

In this chapter we present the design requirements for the framework we have developed. These requirements emerged from a series of discussions with the industry partner (Ciena) who collaborated with us in this project. The initial phase included understanding existing systems and their deployment setup. From this information we were able to classify data processing pipelines into different categories. As a start, we built a deployment setup for a generic data processing pipeline that handled the specific use case of data mis management. The knowledge gained from building this prototype was used as the basis to construct the full prototype.

3.1 Terminology

The terminology used throughout this thesis is consistent with what is used in the cloud computing domain. However, we explain some terms that we believe need introduction in the context of our industry use case.

3.1.1 Service (SaaS) Provider & Tenant

Throughout this thesis the owner of the software services or the data-processing pipelines is called either the *Service Provider* or *SaaS Provider*. They are the industry collaborators we directly engaged with. Our study identifies their requirements and provides a solution for it. It is their objective to move to a cloud service model and to ensure their application pipelines are threat free.

The data processed by the pipelines shown in Figure 3.1 is owned by a customer of the service provider. This customer is called the *Tenant*. It is the tenant's requirement to have assurance from its service provider that their data is processed securely and in isolation in the cloud.

3.1.2 Data-Processing Pipeline

The deployment we studied consists of multiple applications. These applications are connected to each other as shown in Figure 3.1. These applications process incoming tenant data and pass it forward to the next application. A chain of such applications form a pipeline for processing tenant data. We call such a collection of applications as a *data-processing pipeline*. The term *pipeline* is also used throughout this thesis to denote a chain of applications. A single *data-processing pipeline* achieves a specific task in the deployment. There can be several such pipelines in the whole deployment. Each of these different pipelines that is used to achieve a specific task is called a *workflow*.

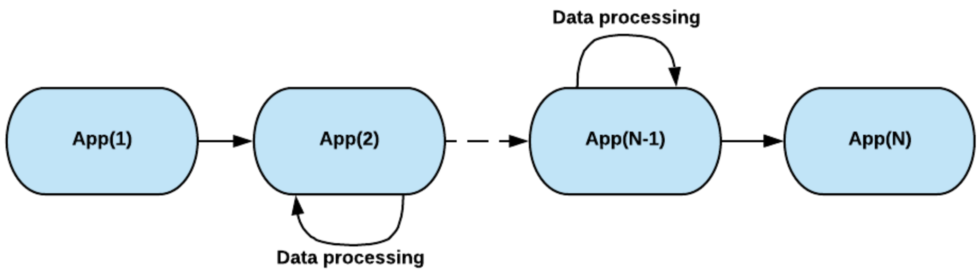


Fig. 3.1 A data processing pipeline

3.1.3 Ingress & Egress Applications

As described above each pipeline is associated with a specific *workflow*. A workflow is initiated by a *triggering event*, internal or external to the system (We discuss different types of events under Design and Architecture in chapter 5). Upon a workflow being triggered, data processing starts at one of the applications in the pipeline and ends at another. The application at which the workflow is triggered is called the *Ingress Application* while the application at which the workflow ends is called the *Egress Application*. Figure 3.2 shows 3 different types of workflows with their *Ingress* and *Egress* applications marked .

3.2 Existing Deployment Model

Our industry collaborator is a network equipment manufacturer. They have many Internet service providers who buy and use their equipment in their deployment setups. These devices generate operational statistics and other useful information over time. These devices also expose APIs that can be used to collect this information. Access to these APIs are allowed only

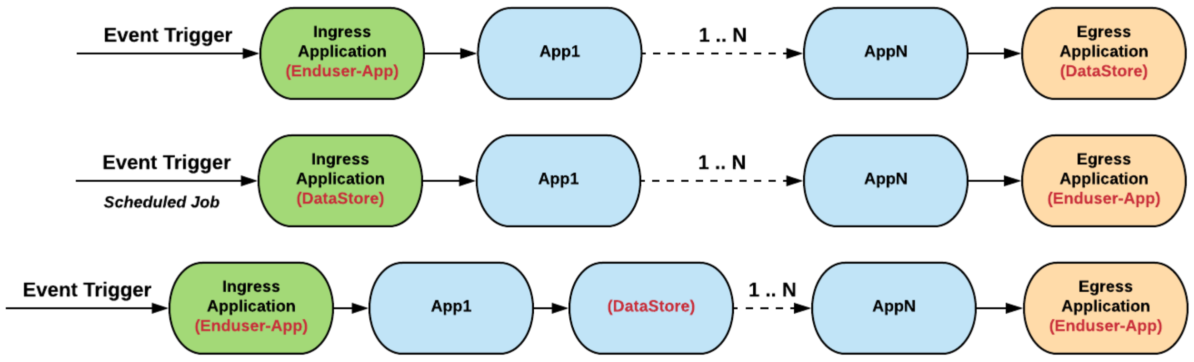

Event Trigger
Ingress Application (Enduser-App)
App1
1 .. N
AppN
Egress Application (DataStore)
Event Trigger
Scheduled Job
Ingress Application (DataStore)
App1
1 .. N
AppN
Egress Application (Enduser-App)
Event Trigger
Ingress Application (Enduser-App)
App1
(DataStore)
1 .. N
AppN
Egress Application (Enduser-App)

Fig. 3.2 Examples of different workflows with Ingress and Egress applications

from within the client site. The collected information is useful for further analysis to understand essential trends in the deployed networks and to improve operation. Such analysis is beneficial to both, the manufacturer and the client. Thus, the manufacturer also provides software services to analyze the collected information.

Currently, there are two approaches in which the information from these devices are gathered: (1) manufacturer-side engineer goes on-site to client location and collects the data by connecting to the device APIs; (2) client collects the device data themselves and uploads it to a client specific jump-server on a daily basis. Upon receiving this data, it is processed by manufacturer-side engineers to produce statistical information and useful metrics. The generated results are shared with the clients as routine reports. However, the manufacturer deems this a tedious work flow and intends to host the services via a SaaS cloud model. In such a deployment the clients (cloud tenants) can upload their data and receive reports, analytics and other information in a timely fashion. As a SaaS provider the manufacturer should fully assure the tenants that the data upload and all subsequent processing happens securely and in isolation from other tenants' data.

With this goal in mind the preliminary phase of this study was to develop a secure flow for how the incoming data can be handled in a SaaS deployment. The focus of the recommendations drafted during this phase was towards application level data-security and not network-level security. Thus, we studied one specific use case to develop an end-to-end secure flow from data upload (into the cloud) until it reaches the data store. This deliverable addresses scenarios like eavesdropping on the flow which could lead to data being compromised and erroneous scenarios that could be caused due to administrator mistakes. In the process of studying and proposing these suitable recommendations, specific research problems were identified in terms of handling security in SaaS deployments with heterogeneous applications. It is one of these problems that developed into the idea of providing isolation via containerization for applications in data processing pipelines.

The section that follows details some of the design choices and implementation specifics of this first phase of our work and the identified voids that essentially became the design requirements for our framework.

3.3 Data Upload Validation

In the transition from the current deployment to a SaaS cloud model, one of the important concerns that had to be addressed was the data upload task. That is, how the data arrives into the cloud; more specifically how the arrived data could be validated for authenticity and integrity. Several use cases (as listed below) had to be evaluated based on the discussions with industry collaborators. The following use cases include concerns raised by both the SaaS provider as well as their tenants.

- A SaaS provider side engineer uploads data to the system.
  - Engineer who has permission to upload any tenant data.
  - Engineer with limited permissions to upload only specific tenant data.
- Tenant side engineer uploads data *(eg: An engineer from Tenant-X).*
- Uploader indicates the data to be from an incorrect tenant.
  - *Tenant-X data attempted to be uploaded as belonging to Tenant-Y.*
- A SaaS provider side engineer attempts to upload data for a tenant he/she doesn't have permission for

Figure 3.3 depicts the overall picture of the steps involved in the recommended upload phase. The upload itself is accepted via a webapp **(Uploader Webapp)**. User login and authentication is handled using an *OAuth* enabled server (WSO2 APIM in our demo setup). Finally, a *microservice* **(Uploader Microservice)** is used to accept uploads and to store them. We explain each of these components next.

3.3.1 Components of the Upload Validation Phase

Uploader Webapp: This is a *java servlets* based webapp running inside a tomcat server. It has basic (username - password) login enabled. During a login attempt this webapp communicates the *username-password* pair to the *OAuth server* to generate a token against it. The *OAuth*

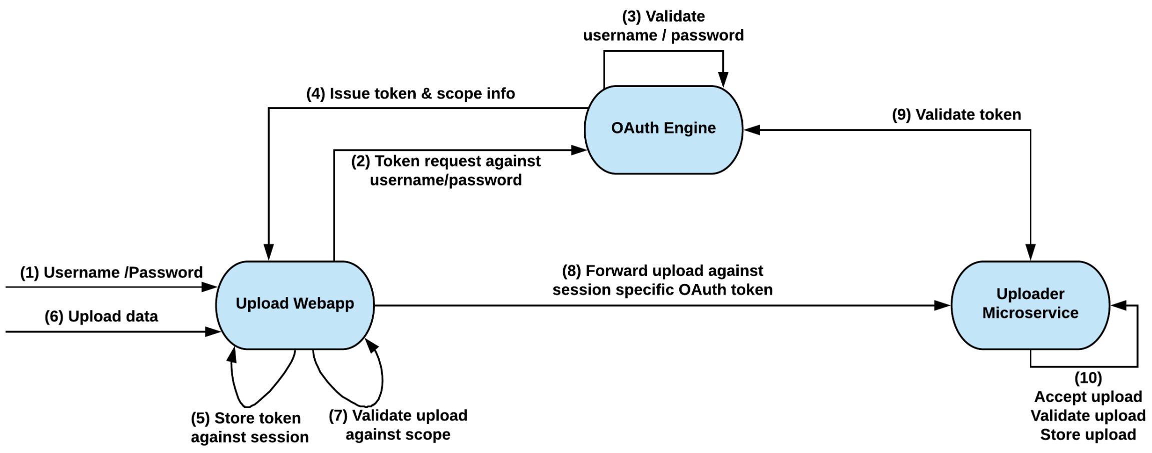

(3) Validate username / password
(4) Issue token & scope info
OAuth Engine
(9) Validate token
(2) Token request against username/password
(1) Username /Password
(6) Upload data
Upload Webapp
(8) Forward upload against session specific OAuth token
Uploader Microservice
(10) Accept upload Validate upload Store upload
(5) Store token against session
(7) Validate upload against scope

Fig. 3.3 Data upload phase

server validates the provided credentials against its user-store and replies back to the webapp with an *OAuth Token* & its related *scopes*[1]. The webapp stores the *token & its scopes* against the current session.

Whenever, an upload is triggered the webapp first validates the selected customer (during the upload phase as shown in *Figure* 3.6) against the *scopes* (explained below) attached to that session. If the current session *(logged in user)* has a *valid scope* for the selected customer, then the uploaded file is forwarded to the *Uploader Microservice* along with the session's *OAuth token*. The *token* is sent as a HTTP header parameter set as *Authorization: Bearer <TOKEN>*.

OAuth Engine: This is the *authentication server* used to validate users and associate authorization roles to them. User authentication is handled in our setup using *OAuth*. The main reason behind this is that an authenticated user could be identified using a single ***token*** amongst multiple inter-communicating processes. For the purpose of building the demo flow we have used the *WSO2 APIM* which has an inbuilt *OAuth engine*. In addition it also has an embedded user-store to create and manage users. It can also be connected to an external user-store such as *Active Directory* via LDAP.

For every login attempt, the *Uploader Webapp* communicates to the OAuth engine to generate a **token**. The OAuth server validates the login credentials against its user-store, fetches the *OAuth scopes* attached to the requested user and generates an *OAuth token* for the current login. The generated *token* along with its *scopes* is returned to the webapp. The OAuth engine

[1]Scopes are *OAuth* specific way of handling roles. Scopes are assigned to users and API resources enabling management of resources based on the user-resource scope matching.

is also contacted by the *Uploader Miroservice* (when it receives an upload from the webapp) to ensure that only valid uploads are being routed to it.

- **OAuth Scopes:** We have also leveraged the *scope* based authorization mechanism introduced in OAuth 2.0. For the demo setup we have created scopes based on imaginary tenants *(bell_upload, fido_upload etc.)* and attached them to sample users *(samadh, shabir, bell_engineer, fido_engineer and etc)*. Thus, if a logged in user attempts to make an upload for a tenant without having the tenant *scope* associated with him/her, then the upload is denied.

  An example user-scope mapping setup could be as follows:

  - **samadh** : bell_upload, fido_upload, comcast_upload, senior_engineer
  - **samsun** : junior_engineer, velco_upload
  - **bell_engineer** : bell_upload
  - **fido_engineer** : fido_upload

Uploader Microservice (UMs): A successfully validated upload *(against the user-scope)* is forwarded to this backend microservice along with the *OAuth token*. This microservice first validates the *token* it receives by communicating to the *OAuth Engine*. Then, it carries out the validation process *(explained below)* to ensure that the uploaded data corresponds to the tenant against whom the upload was made. If this validation passes then the data is allowed to follow through to the next phases of the pipeline. The sequence diagrams in *Figure* 3.4 and *Figure* 3.5 depict the login and upload flows that were described in this section.

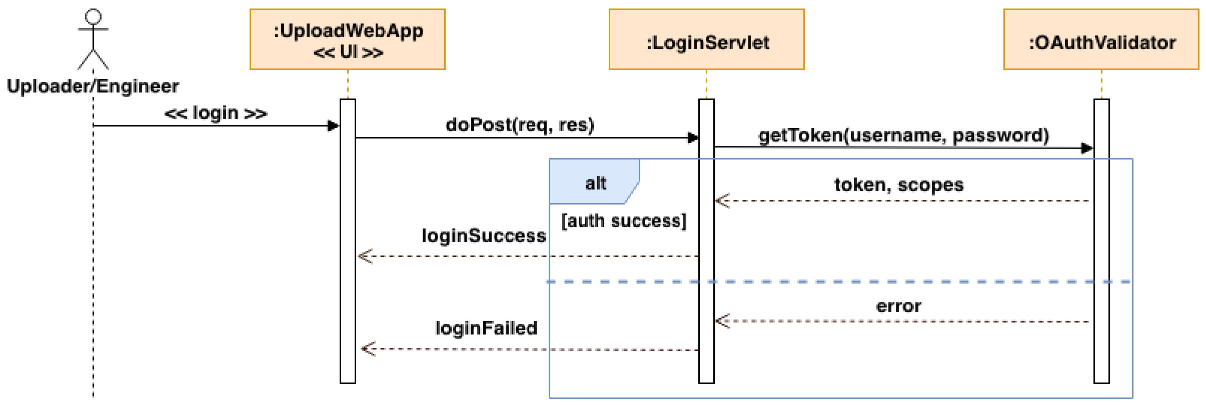


Fig. 3.4 Sequence Diagram of the Login flow

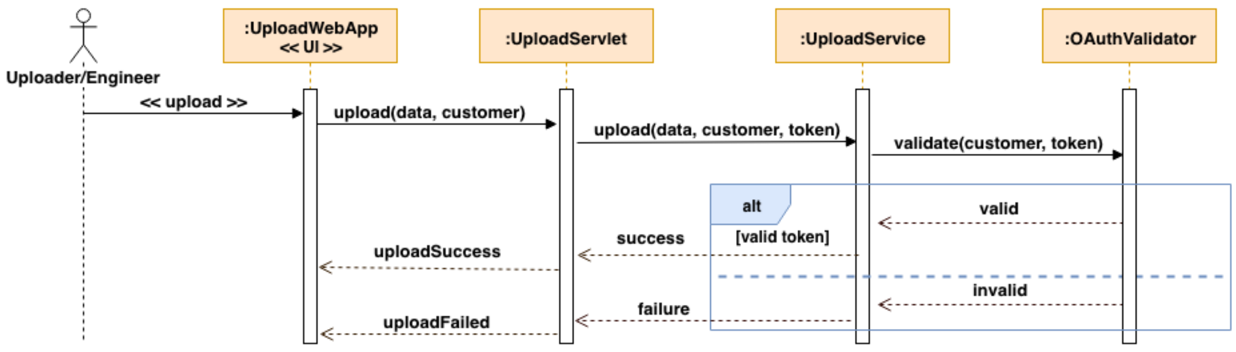


Fig. 3.5 Sequence Diagram of the Upload flow

3.3.2 Handling Incorrectly Tagged Uploads

One of the key use cases addressed in this phase is the ability to handle an upload with incorrectly tagged data. That is, with a cloud based upload scenario *(unlike something like the existing jump-server method)*, an upload can happen in two ways: *(1) Tenant side engineer uploads data* or *(2) SaaS provider side engineer uploads tenant data*. The second use case here has an overhead of the SaaS provider side engineer having to tag the data with the correct tenant. Thus, a situation in which this engineer *(or the uploader)* selects an incorrect tenant for whom he/she is making the upload for. This will lead to the uploaded data being stored in the datastore under a wrong keyspace/table. An example of this situation is shown in *Figure* 3.6. Here, the data to be uploaded *(FIDO_test.zip)* belongs to tenant *FIDO*; however, the uploader has selected the tenant as *BELL*.

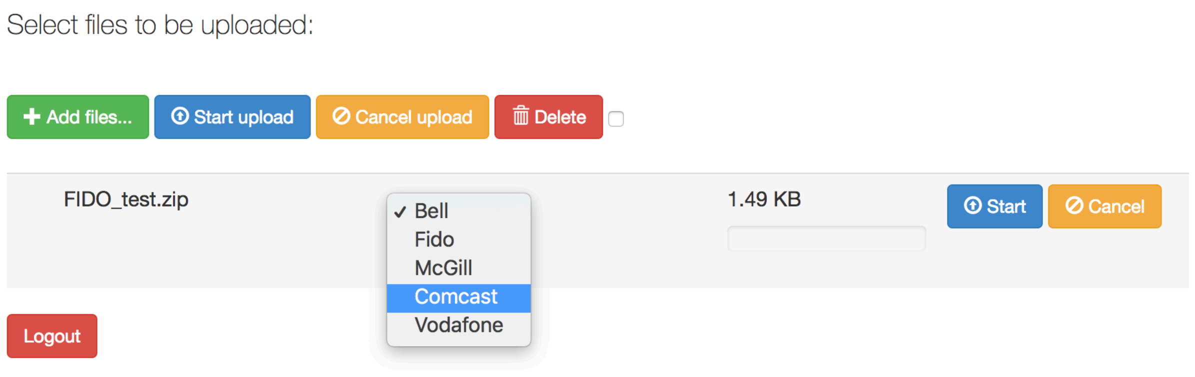


Fig. 3.6 Sample Upload scenario

Solution

The solution to the above problem would be to be able to validate every upload that has been made against the tenant for whom it was made. To be able to do that the data will have to carry some meta information that verifies its source *(i.e tenant)*. In addition, this meta information also should be such that it *cannot* be changed by anyone in the event of a malicious attempt to do so. Thus, we propose a method based on **encryption keys** to sign the data before it leaves the tenant premises'. Then, we would be able to validate this signature on every upload against the tenant to whom it was attributed to, by the uploader.

Encryption Scheme for data signing

This proposed scheme is based on the combination of *RSA*[2] *Private-Public key and AES*[3] *symmetric* crypto systems. A *Public-Private key pair* would be generated per tenant during the on-boarding process. The *Private keys* will be held securely by each tenant. The *Public keys* of all tenants will be held inside the *keystore* specific to the *Uploader Microservice*.

Any data collected from the tenant devices *(via the service provider's device APIs)* will be encrypted with the *Public Key* of that specific tenant. Hence, data is uploaded to the system or is given to a service provider side engineer (for upload) only in an encrypted form. Thus, whenever an upload happens (against a specific tenant), the *Uploader Microservice* first attempts to decrypt the data using the *Private Key* that corresponds to the tenant *selected* during upload. If the decryption attempt succeeds, then we can be assured that the uploaded data corresponds to the selected tenant. However, a failed decryption would denote a mismatch between the ownership of the data and the tenant who was selected during upload. Such, a scenario is treated by discarding the upload completely and alerting the engineer about the incident.

Data Encryption Agent

A *data encryption agent* was developed to evaluate the process of encrypting/decrypting tenant data. The tenant data is **not** encrypted/decrypted by the *RSA cryptosystem based key pair* mentioned in the above passage per-say; alternatively a *randomly generated key* is used to encrypt the data using the **AES** encryption scheme.

Issues in using RSA crypto system to encrypt/decrypt large files [36]:

- RSA is slow/compute intensive compared to symmetric (AES) encryption, especially for decryption. It is almost 4 decimal orders of magnitude or more compared to symmetric

[2]RSA is one of the first practical public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and differs from the decryption key which is kept secret.

[3]The Advanced Encryption Standard (AES), also known by its original name Rijndael, is a specification for the encryption of electronic data.

encryption. RSA's execution time grows much faster than linearly with the modulus size used in the RSA algorithm (even much faster than quadratically, for decryption).

- Safe RSA-based encryption schemes expand the size of the ciphertext. For example RSA-OAEP with SHA-256, when using 2048-bit RSA, uses 2048-bit ciphertext blocks (256 bytes) which convey at most $2048 - 2 * 256 - 16 = 1520$ bits (190 bytes), loosing over 25% of the bandwidth. This can only be improved to some degree (for n-bit security, there must be at least n bits lost per block).
- RSA with long-term keys does not offer *forward secrecy*[4].

To tackle the above issues and still be able to use the key-pair based scheme explained earlier we use the **Hybrid-Encryption**[5] technique.

Hybrid-Encryption scheme [Encryption]:

1. Generate a random *AES-key*.
2. Encrypt the tenant data (which sometimes approximates to over $5Gb$) via the *AES encryption scheme* using the generated random *AES-key*.
3. Calculate the *MD5* message digest of the encrypted data.
4. Concatenate the **(message digest + the encrypted data)** and then *base64 encode* them together.
5. Encrypt the generated *AES-key* using the **tenant specific public-key**.
6. Create the final upload file by **zipping** together the encoded file and the encrypted *AES-key* used for data encryption.

Hybrid-Encryption scheme [Decryption]:

1. *Decode* the encoded unit and retrieve **(message digest + the encrypted data)**.
2. Calculate the message digest of the encrypted data portion above.
3. Compare the message digest from step-2 against the one obtained from step-1
    - If they don't match, then alert user and disallow upload

[4]In cryptography, forward secrecy (FS), also known as perfect forward secrecy (PFS), is a property of secure communication protocols in which compromise of long-term keys does not compromise past session keys.

[5]In cryptography, a hybrid cryptosystem is one which combines the convenience of a public-key cryptosystem with the efficiency of a symmetric key cryptosystem.

- If the message digests are the same then go to step-4

4. Retrieve the *AES-key* by decrypting the key-file using the *tenant specific private-key*.
5. Retrieve the actual data by decrypting the encrypted data using the *AES-key* obtained from the above step.
    - If decryption fails then alert user and disallow upload

File size	Encryption time	Decryption time	Main memory	Disk memory
465 Mb	2.5 sec (40 sec)	2 sec (12 sec)	32 bytes	2 * File size (900Mb)
920 Mb	5 sec (1.5 min)	3 sec (23 min)	32 bytes	2 * File size (1.8Gb)
1.4 Gb	6 sec (2 min)	4.5 sec (35 min)	32 bytes	2 * File size (2.8Gb)

Table 3.1 Encryption-Decryption Process

Table-3.1 is an analysis of the time taken for the *encryption & decryption* process. Three different file sizes have been studied to understand the time taken and memory consumption for this. The file sizes were chosen based on a range of daily average upload sizes as indicated by our industry partner. It is noteworthy that the whole encryption process includes two rounds of encryption *(the data & the symmetric-key)*, message digest calculation, encoding the encrypted data and finally zipping the contents *(encrypted data & symmetric-key)* together. Thus, the times shown in Table-3.1 denote the time taken for the encryption/decryption alone; while the times inside the braces denote the time taken for the entire process. The main memory footprint here is the portion of the file read into the encryption buffer at a single time. It was observed that the time taken by the AES encryption algorithm was too high when large portions of the file were read at once. Thus, an iterative algorithm was implemented to read the file chunk by chunk and flush to disk immediately upon encryption. The reverse of the same flow was followed for decryption. This implementation of the **data encryption agent** is currently running in the production environment of one of the collaborators of this project.

Moving forward from the data-upload validation phase we studied the complete pipeline for how the data is processed until it reaches the datastore. The findings from these studies helped define the scope and vision for the framework that is delivered.

3.4 Design Requirements Emerging From the Initial Study

Once the *data-upload step* has been validated for authenticity and integrity, it must be ensured that the data reaches the datastore via a secure channel. It was observed that the pipeline that carries the uploaded data from the beginning until the end (datastore) may consists of a myriad of different software tools and technology stacks. Based on the variety of software and technology stacks used, a combination of multiple security protocols could be enabled to ensure that the flow is isolated and is secure from external tampering. This is the same for all types of data processing pipelines made up of heterogeneous applications. An example of another type of an observed pipeline was when a user triggers for analytics from the stored data. In this use case the data is already found in a datastore. So the pipeline begins from the store and ends at the user-facing application (the opposite of the upload scenario).

However, one of the common traits of all such pipelines is the heterogeneity of the many applications that make up the pipeline. In such cases each of these different applications expose their own ways to instigate security on and around them. This necessitates that the application developers will have to be aware of the different security protocols that each of these applications operate on. They also have to pay attention to the dependencies between different versions of such protocols to ensure lucid integration. This prevents quick and easy addition/removal of new applications to/from a data-processing pipeline.

In addition, a more significant issue is that the above characteristics are a hurdle to the main goal of this discussion which is to protect data-processing pipelines from *passive internal threats*. As discussed in the previous chapters, our primary objective is to protect the data and the pipeline from threats that occur due to *(passive)* developer errors. If the application security checks are also done by the application developer then they too are prone to errors. Thus, a key design requirement emerging from this prototyping phase was to provide a defensive boundary around the applications. In addition, it was required that the developers only focused on the core logic of the application while tenant specific parameters are externally made available to the application environment. To achieve this, it was required to define a surface outside the applications itself to instigate security checks. This surface at which the security checks would happen must be independent of the applications itself but yet close enough to encapsulate them (the applications) as independent logical units. Hence, we build this surface by containerizing the applications of the pipeline; then we define the container boundary as the interface for setting up the necessary security checks. This unique boundary allows us to setup defense-in-depth controls on the applications. In addition to the applications themselves being isolated, the data-processing pipelines (for different tenants) also must have isolation. That is, different tenant data should be processed in dedicated pipelines as a defense-in-depth measure. We enable this in our framework by not reusing container instances of one tenant to process data from another tenant. We also ensure that there is no crosstalk between different tenant specific containers using SDN controls as discussed next.

With the above requirement, we now have to define the means for communication between the pipeline applications. It is important to be able to dynamically control the communication topology while actively monitoring the channel. Dynamic topology control allows re-routing data-flows via different application instances for added security. Active monitoring enables real-time capturing & alerting of bogus activity of a specific application. Thus, defining a proper communication mechanism is a crucial requirement to the design. We achieve this using the control-plane & data-plane separation enabled by software defined networking (SDN).

With proper mechanisms for application isolation *(to provide defense-in-depth)* and communication channels *(to enable dynamic reconfiguration and monitoring)*, the next requirement would be to ensure that the design is modular. A modular component based framework is important so that new feature inclusion and framework updates are made easy. This also makes troubleshooting easy and the framework more scalable. Another important design requirement that was identified is that the framework should support easy integration of different types of workflows. The data processing pipelines may change over time. New tools may replace older ones. Intermediary applications may be injected to an already existing flow. Thus, the framework should make the on-boarding process (for a new pipeline) straightforward. We achieve this by separating the implementation details of each pipeline into a generic interface which can be re-used for new pipelines. Thus, a new pipeline can be added to the framework within hours by providing the right interface implementation and including its definition in the configuration.

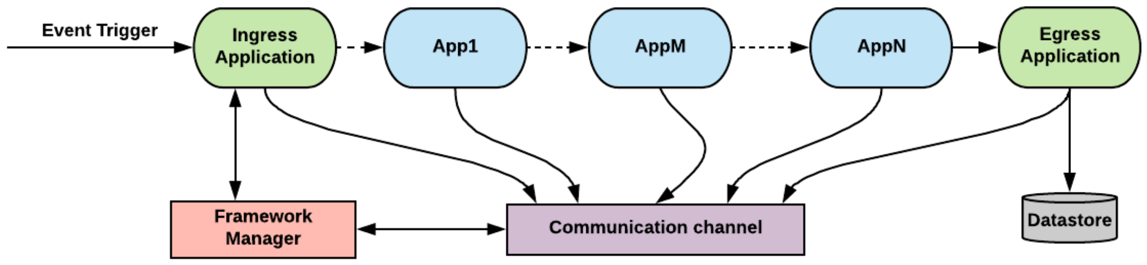


Fig. 3.7 Expected system model from identified requirements

Figure 3.7 is a high-level depiction of the system model emerging from the identified requirements. We describe the implementation and functionality of each component of this model in Chapter 5. To summarize, the goal of the system is to protect tenant data from *passive-internal threats* - threats arising from unintentional developer mis-practices when building applications.

Accordingly, the system requirements established from the evaluated industry use cases are as follows.

- The framework shall provide a strict isolation boundary for applications to instigate defense measures.
- The framework shall make available tenant specific configurations/information to the application developer from an external isolation boundary.
- The framework shall enable the application developer to write business logic using tenant specific information made available at the isolation boundary.
- The framework shall ensure different tenant data are processed by different isolated pipelines.
- The framework shall provide communication means between the application isolation units.
- The framework shall enable network level monitoring of the application communication.
- The framework shall ensure real-time evasion of unintended crosstalk between pipeline applications.
- The framework shall allow dynamic wiring of applications to enable communication between them.
- The framework shall allow new workflows to be added with less effort than deploying them without the framework.
- The framework shall allow existing pipelines to be extended by introducing new applications to it.

The threat model addressed by these requirements is discussed in Chapter 4 and the implementation specifics of these requirements are explained in Chapter 5.

Chapter 4

Threat Model

In this chapter we discuss the threat model addressed by the proposed framework. As discussed in the previous chapters our focus is to handle *passive-internal threats* in SaaS based cloud deployments. We enumerate the different threat vectors that are of concern for such SaaS deployments. We then explain how our framework is able to avoid these threat scenarios and how it enables easy integration of additional security measures.

4.1 Threat Vectors

The threat vectors of the *passive-internal model* are related to how the data is handled in SaaS clouds. The cloud-infrastructure providers by default have their own *service level agreements* with regards to protecting the cloud VMs. However, SaaS providers who have their software services deployed in the cloud need assurance against threats that are different from the ones addressed by the cloud-infrastructure providers' *SLA*s. The threat model here is focused towards the behavior of the deployed applications and how data is processed by them. We target SaaS deployments where a set of applications create a data-processing pipeline. Such pipelines have an **ingress** and **egress** application. The data enters the pipeline through the ingress application and reaches its resting point via the egress. The applications in the pipeline transfer data in a specific order such that it is processed accordingly.

An important aspect of such scenarios is that the application development and deployment are two different processes. Thus, in most companies they are handled by different teams. The cloud administrative team is mostly unaware of the internals of the applications. They just provide the system logistics to enable these applications to interact as requested. However, an important point-of-vulnerability are the applications themselves. The primary source of threat in this model are improperly/maliciously behaving applications. The reason for such unintended behavior could be anything from *ignorant programming errors* to *maliciously written programs*. Thus, the end-goal of the framework is to ensure protection of data against the effects of such bad practices and unintended errors.

4.1.1 Application Crosstalk

This type of threat arises when a certain application of the pipeline attempts to communicate / connect-to applications that are not one of its immediate *pipeline-neighbors*[1]. Such an application might alter the processing order of the data or may threaten the stability of the system. This in return will lead to the data losing its integrity. The effects of this threat could create a chain of problems to the system since it is carried with the data. It could lead to incorrect/misleading information being written to the final datastore.

4.1.2 Data Leak

Data leak related threats arise when an application **writes** any processed data *(to some location)* while the expected behavior is only to process the data and transfer it to the next application in the pipeline. Such behavior could be attributed to *partially reviewed code* before deployment and also *maliciously written programs* with the intention of gaining access to the data. Leakages due to ignorant coding/deployment practices makes the data easily accessible in the event of the cloud-VM being compromised. On the other hand, intentionally leaked data could effect the system immediately and be used for other malicious activity. In addition to unintended writes, data leaks could also happen via in-memory data hijacking attack.

4.1.3 Data Mixing

One of the key requirements of cloud deployments is to promise secure processing and transmission of data that are owned by different tenants[2]. The SaaS provider should be able to provide this promise to its tenants whose data, the SaaS applications will process in the cloud. A pipeline of applications that process data owned by multiple tenants poses the important concern of **data-mixing**. These pipeline-applications must use appropriate *processing-parameters* and *process-logic* depending on the tenant whose data is being processed. Any errors *(intentional or ignorant)* could lead to data from different tenants being mixed and incorrectly tagged. Even worse the incorrectly tagged data could end up in the wrong datastore, leading to erroneous results. Thus, there should be proper isolation mechanisms to ensure different tenant data are handled by different channels. In addition, the defense mechanism should be able to capture any such erroneous behavior and stop bogus applications from altering the integrity of the data by mixing it with others.

[1]Pipeline-neighbors means applications that are logically next to each other when processing the data.
[2]Tenants here are *customers* of the SaaS provider.

4.1.4 External Access

Any application internal to the pipeline could attempt to communicate outside the deployment boundary itself. This would not only load the network activity of the deployment server but more importantly allow data to be leaked elsewhere. Thus, applications should be prevented from trying to access anything outside the pipeline's network boundary. All pipeline-applications have the single task of processing the data and transferring it to the neighboring applications. The only exception would be the *ingress* or *egress* applications that could be user-facing to receive event triggers and to provide results. Hence, all other applications' network activity should be restricted to a strict boundary.

4.1.5 Unconstrained Resource Utilization

Applications could hold server resources for a period more than what is required. In addition they could be performing computation that is not part of the pipeline-processing. This would lead to un-accounted utilization of the resources for tasks that is not of benefit for the SaaS provider. Bogus applications, could keep hold of server resources for irrelevant/malicious computation. Huge losses could have been incurred by the time this is realized and the deployment is cleaned. Thus, measures should be taken in order to restrict the resource utilization of the applications. In addition, it must also be possible to audit the resource utilization of specific applications over a period of time.

4.2 Defense Measures

As discussed at the beginning of this chapter the common characteristic of the threat vectors listed above is that the point-of-vulnerability is the applications that are deployed. Hence, the focus of the framework is to capture bogus behavior of these applications and prevent them from causing any harm to the system and the data. The primary challenge in achieving this is on how to define a rigid defensive boundary for the applications. Such a boundary must be independent of the implementation of the applications. It must be decoupled and farther away from the application such that it does not impose additional rules for the application programmer; acting as a transparent shield. In addition, the boundary must also be close enough to the application so it creates a suitable and accountable encapsulation unit; one which is well separable from other applications.

We define this defensive boundary by containerizing *(as explained in the previous chapter)* each application that is part of a data-processing pipeline. The container boundary is a stable encapsulation unit that provides the environment for the application to run independently but also defines a barrier before it could reach outside. By default Linux containers provide features such as isolation from host processes and fine-tuned resource allocation. Our framework makes use of these features and builds into these containers the defensive measures necessary to protect the application & data from each of the threat vectors discussed earlier. In addition, the architecture is made modular such that it allows for future improvements in terms of security measures. Communication between these containerized units is enabled via connecting them to an *Open-Flow* enabled virtual switch. The **controller** module connected to this switch provides the defense mechanisms required in terms of external-reachability and communication of the applications. A **manager** module takes responsibility of monitoring on-going events and the activity of the applications in different pipelines. The **manager**[3] is made capable to trigger alerts when anomalous behavior is detected.

Application Crosstalk: Upon, receiving a new *event-trigger* it is the responsibility of the manager to inform the controller about the *type of event* that was triggered. In addition it also notifies the logical organization of the applications in the event pipeline. As per this information the controller will publish ***flow-control rules*** to the switch, telling it what are all the allowed flows (communication) per application in the pipeline. Any, unauthorized crosstalk between the applications will be captured and disallowed. Moreover, this information is logged enabling future audit of the behavior of each application. Thus, the controller acts as the strict enforcer of communication rules between the applications and provides defense against any bogus application crosstalk.

Data Leak: The container boundary is the surface with which each application is allowed to interact with. Applications, when encapsulated within a container will not be able to see any of the host processes nor will they have access to the host file-system. Their runtime environment is restricted to what is provided by the container. Hence, whatever local *writes* they do are only visible within the container and is only available for the lifetime of the container. The framework brings down all the containers contributing to a specific event upon successful termination of that event. Hence, any local writes during a single event will be automatically cleaned upon completion of that event. As a result data-leaks *(intentional or accidental)* will not have an effect in the long run. All disk writes are cleared during the event termination phase and the system returns to a fresh state. The memory surface of each container is also completely isolated from each other and will only be valid as long as the container is running.

Data Mixing: As indicated in the previous passage, every new event that is triggered has its own set of fresh containers that form the data-processing pipeline. Thus, no instance of any

[3]The operations of the **manager** and the **controller** are explained in detail in Chapter 5

application in a pipeline, processes data from two different events. Every batch of incoming data are provided with a dedicated pipeline of applications from *ingress* to *egress*. Whilst containerization provides isolation of the applications, isolation amongst different pipelines must be achieved at network level. Our framework achieves this via separately published flow-rules per pipeline to the virtual switch. These rules ensure that no mixing of data is allowed between any two different pipelines. This creates a separation between different tenant data and serves as the mechanism to evade data-mixing.

External Access: In the same way that the virtual-switch protects data from leaking between pipelines, it also ensures that data is not leaked outside the deployment environment. In addition, the virtual-switch is setup fully internal to the host and is cut off from any external connectivity. It sits inside the deployment host as the conductor for enabling connectivity between the containers while preventing all external access. The switch is controlled via the **controller** module as explained earlier. In addition, the framework controls *hostname resolution* to enable applications to reach out to neighboring applications. Thus, applications must stick to application specific hostnames in trying to communicate to neighbors. The manager-module ensures that proper hostname resolution per container is configured depending on the application it hosts. Thus, connectivity outside the container is allowed only as required by the pipeline.

Unconstrained Resource Utilization: The Linux-kernel by default provides hooks to control the amount of system resources allocated per container. This sets a limit on how much resources a containerized application can use for its computation. We make use of this functionality to build into the framework the capacity to control resource allocation per container *(application)*. The manager-module provides hooks to define container specific resources. In addition the controller-module supports gathering of network activity of different applications. Together the framework can support fine-tuned resource allocation per isolation unit *(container-level and pipeline-level)* while providing alerting and monitoring services regarding how the resources are occupied/utilized. The potential to integrate this feature into the framework ensures that applications are prevented from unconstrained resource utilization.

4.3 Counter Measures

Application crosstalk is blocked at IP level. Each application is encapsulated in a container and is assigned a dynamic IP address. Given the containerized isolation, applications can reach each other only via network communication. Thus, the defense has to be introduced at the IP level. During the *setup phase*, flow rules are published as explained in Chapter 5. A rule table per application (in the pipeline) is created in the virtual switch. The rule table contains one rule per neighboring application and a general **drop** rule for all other applications in the pipeline. Each rule is matched against the IP address, the OVS-Port and in some cases the MAC-address of the incoming packets. This can be further extended to create alerts on every drop incident. Moreover, the developers are expected to use predefined URLs (instead of IP addresses) to reach neighboring applications. During the *setup phase* the correct IP address to URL mapping is automatically configured into each container by altering the *hosts* file. Thus, the correct IP address is resolved for the right URLs used in application configurations. The flow rules published to the virtual-switch can be varied depending on the application type and the protocols that must be monitored and allowed. This is a major advantage of using SDN based network management where the control plane can be dynamically configured.

With network isolation **Data Leaks** and **Data Mixing** is only possible when the persistence-disk is shared between applications and pipelines. As discussed earlier, containerization completely isolates the writes of each application from those of others applications. The only shared persistence medium are shared-volumes used with file based applications *(eg: logstash)*. Data-mixing *(between multiple flows of a single tenant and flows of different tenants)* in such scenarios is completely evaded by having dedicated shared-volumes for each pipeline. No two flows share the same volumes for data transfer between applications. This is configured into the container instance during the *setup phase* of a new pipeline. In addition, the shared-volumes are configured as a temporary data transfer mechanism *(only when needed like with logstash)* between applications. Thus, upon termination of a pipeline these volumes are complete removed and cleared. The only data at rest will be at the datastore *(eg: Cassandra*) which will be encrypted. Any data maliciously leaked will be completely lost upon termination of the pipeline. Besides, the malicious actor will need root access to the system to take advantage of any leaked data even if it is persisted. Thus, with only minimal setup requirements, our containerization approach ensures no data leakages are allowed. Our design also easily encapsulates all internal writes from mixing with others. Besides, there is no additional overhead in cleaning up since the container ecosystem automatically takes care of it once a container is stopped and removed.

The default Ethernet interface of the host-machine is removed from the virtual-switch cutting off any **External Access**. The application-containers are initially spawned with no network interfaces assigned to them. Then they are attached to the OVS using a veth (Virtual Ethernet) pair. The OVS-port to container mapping is maintained in a separate structure along with the IP address assigned to the application. Hostname resolution is done as explained above. Thus, none of the applications can reach outside of the system. The **ingress** application-container, alone is assigned an additional network interface. Along with the veth pair connecting it to the OVS, this container is also attached to the docker-host bridge. This is to allow for the tenants to be able to spawn jobs in the system to process their data. However, the *ingress* and *egress* applications are written and managed by a high level authority in the company. Thus they are not considered amongst those applications which are deemed harmful. Moreover, communication channels from the border-containers are also controlled by flow-rules in the OVS. Thus, by having the OVS completely isolated from external connectivity and ensuring that the containers have restricted network mediums the framework maintains stability. No knowledge of the host machine can allow a malicious actor to reach the framework and tamper a flow from outside.

Finally, the framework is implemented with extensibility in mind. This is one primary advantage catered by having the framework modular and using SDN based controls. New flows can be easily integrated into the framework, by setting up the appropriate container-images and including the required configurations in the main configs file. This makes way for setting fine-tuned control over how much resources are allowed per container-image and what percentage of system resources can be utilized. The controller has full capability to restrict resource utilization. In addition, the flow controller can integrate newer modules to support additional alerting and monitoring features to the framework. Overall the framework provides robust defensive measures against the threat model discussed above while providing the necessary extensibility to support emerging threats and business requirements.

Chapter 5

Design and Architecture

In this chapter we first define the design choices for the framework. We then provide a detailed description of the architecture and how each component is connected to each other. While most design choice tools are a result of being freely and easily available, any similar technology can be used as a replacement for an implementation of the framework. A key characteristic of the design was to utilize existing tools and systems infrastructure to quickly and easily achieve *defense-in-depth* for enterprise cloud deployments with scheduled workflows.

5.1 Overview

The design of the proposed framework is based on systems with workflows of multiple applications that handle large volumes of tenant data. Figure 5.1 is a depiction of such workflows. As can be seen in the figure such a workflow contains an event inlet (called the **Ingress Application**) and an event outlet (called the **Egress Application**). In between the *Ingress* and the *Egress* there may be any number of intermediary applications that process the tenant data. However, it is noteworthy that the flow of the data need not necessarily be linear as shown in Figure 5.1. Certain applications in the workflow may communicate with more than one application; thus denoting that they have multiple neighbors. This study only considers workflows where such applications *(ones with multiple neighbors)* talk to only one neighbor at any moment. That is, no concurrent communication happens amongst multiple neighbors. Figure 5.2 is an example of one such workflow. In addition, the design specifically takes into consideration two different types of workflow scenarios:

- `[1]` Off-line batch processing workflow.
- `[2]` On-line user-engagement based workflow.

To understand the differences between these two types, we define what the triggering **event** is. The *"event"* can be a simple user-trigger to perform a specific task that expects a feedback *(eg: fetch me statistics of sales for last month)* or it could also be a one-way trigger. A one-way

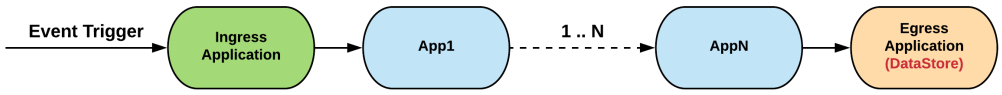


Fig. 5.1 A layout of a sample application workflow

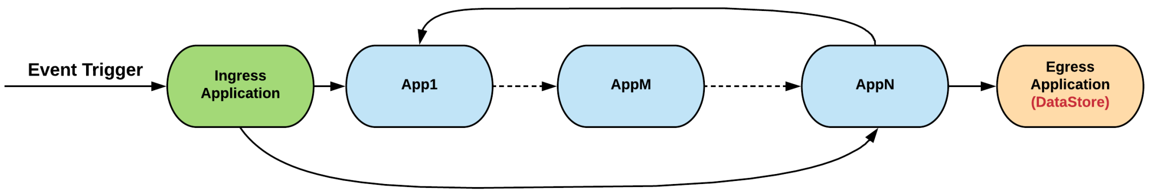


Fig. 5.2 A workflow example with non-linear cross communication

trigger is an event that requests for the execution of a specific workflow but does not expect an immediate response. A typical example of this is one where tenant data is uploaded to the system to be processed and stored. To further define the scope of a workflow we also note that the *egress* application is either a datastore or a user facing app such as a web UI.

5.1.1 Off-line Batch Processing Workflow

Off-line batch processing workflows are one-way triggered pipelines as explained above. The user triggers a specific workflow expecting a task to be performed internal to the system without expecting any response. Thus, the framework sets up the applications of the workflow, processes the tenant data, puts it into an *egress* application (i.e. datastore). Depending on the workflow an *egress* can be an intermediary application of the flow while also being at the tail end. We explain this by the example of a flow where an event triggers some tenant data to be fetched from a **datastore** *(egress)*, processed via a pipeline of applications and written back to the **datastore** (once again the *egress*).

5.1.2 On-line User-engagement Based Workflow

In contrast to the above, an *on-line user-engagement based workflow* is one where an event is triggered and a response is expected. In such scenarios the *ingress* and *egress*, are both the same application that enables event triggering.

We focus the development of the framework to *off-line batch processing workflows*. Thus, it is based on the example of a specific workflow which receives an upload event, processes the uploaded data through a pipeline of applications and deposits the end-results to a datastore.

We then show that the implementation is extensible to different types of workflows with proper handles for flow-termination.

5.2 Design Process

The key target of the framework is to provide **Defense In Depth** functionality to workflows that process different tenant data in cloud environments. As discussed in the introduction to this study, *defense in depth* promises security controls at different layers of the data and application life cycle. Our framework specifically targets the problem of applications with unintended behavior. Such behavior could occur as a result of multiple reasons. The threat model for such unintended behavior stems from the basis of improper programming practices by application developers. This could be a malicious act or a mistake out of ignorance.

Thus, the primary goal of the framework is defining an **isolation boundary** for the applications to enforce security controls. Upon, introducing this boundary we note that the primary act of threat is unintended network traffic to and from these applications. Hence, next we define the **inter-application communication mechanism** and how such cross-communication is controlled. A sub-problem of the design of communication is **hostname resolution**. Then, to provide completely isolated data processing pipelines for different tenants we introduce **isolated pipeline allocation** for each tenant in the framework. That is, data specific to two different tenants are not processed via the same pipeline of applications. Each one gets its own channel from *ingress* to *egress*. Having dedicated and `active` pipelines for different tenants enforces, resource sharing within the system even when some pipelines are idle. Thus, our framework sets up processing pipelines **on demand** for new tenant events and strips them down upon completion. The focus of the development of this framework was based on a requirement that has scheduled and infrequent events. In addition, the template use cases used for the development of this framework consist of self contained applications that can be dynamically spawned and terminated. Thus, it is acceptable that the framework adapts a design choice to not persist active pipelines beyond their active requirement. It is also noteworthy that only the intermediary applications of the pipeline follow this non-persistent design of the framework. The *ingress* and the *egress* applications are kept running throughput the lifetime of the system. This is because these are the boundary points where the event *initiation* and *termination* are captured.

The choice to setup applications on demand *(i.e. when a new event occurs)* poses an important problem. That is, how can the framework correctly decide that the applications of the pipeline are ready for action. To elaborate more, let us take the example of a *data-upload* workflow where the uploaded data is processed via multiple applications and is deposited to a datastore. With our on demand pipeline model, when an event is triggered an initial delay is observed for the flow to be setup. This steps involves spawning the applications that are part of the pipeline that handles the event in question. Since, these applications are spawned independently, we need a mechanism to deterministically conclude that all of them have duly started and are ready to accept the event flow. We cannot assume that the applications will be up and running in time when the data arrives. We handle this problem using a **notification mechanism** that informs the framework that a particular application is up and running. The final piece of the design process is proper **termination of the pipeline**. This includes terminating all intermediary applications and cleaning up of all resources incurred during setup. In the following sections we discuss how each part of design process (discussed above) is delivered in the framework.

5.2.1 Application Isolation

A key aspect of the framework is defining an isolation boundary for applications such that security controls could be enforced. The framework achieves this by **containerizing** all the applications that are part of a workflow. A container [46] is a lightweight virtual execution environment that includes everything needed to run a specific piece of software. Containers completely isolate applications from its surrounding and enables applications to be run in a platform agnostic manner.

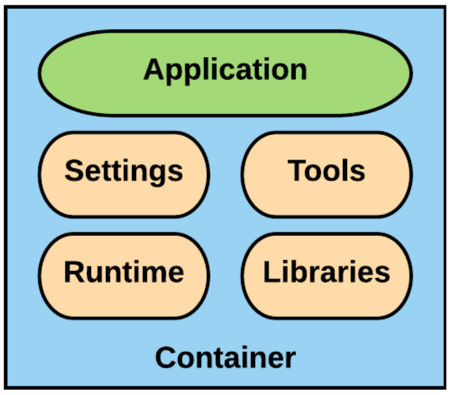


Fig. 5.3 A Container hosting an application with the required tools

Figure 5.3 depicts how a containerized application would be. For the purposes of our framework containerizing applications provides us a nice boundary to introduce security controls to the application behavior. This boundary is close enough *(to the application)* such that it does not overlap with other applications' execution environment and is far enough *(to the application)* such that different controls could be enforced. Thus, the framework enforces applications to be containerized first to ensure *application level* isolation. The isolated environment emulates a lightweight `linux` machine that makes use of the underlying host-kernel. Thus, all isolation units share the same host-kernel but within their own execution boundary. The isolation itself (at the host kernel level) is achieved via **linux namespaces and cgroups** that enables partitioning of kernel resources . In our implementation of the framework we use **Docker** container platform to containerize the workflow applications. Containerization is a one-time process where the container configurations *(application source, libraries, other tools and execution command etc.)* are provided as a config file and a corresponding *container-image* is built. Then, this image can be used to spawn as many containers as required. This image is the source code equivalent of a running process *(in this case a container instance)*. This *image* defines the necessary security controls and required scripts to manage it, in addition to the application related binaries and libraries.

5.2.2 Inter-Application Communication Mechanism

With application isolation in place, the framework needs to define a mechanism for communication between the applications. Since application isolation has been achieved via containerization, our unit of control is now the container within which the application resides. Thus, the discussion of enabling communication between applications necessarily means how to enable networking between the containers.

The *docker* [8] container platform by default supports multiple networking modes to allow inter-container communication. The simplest mode is one where all containers share the host networking stack as shown in Figure 5.4. In this mode all containers (even though isolated) share the host *TCP port range*. This means these containers will act as processes on the same host with inter-process-communication (IPC) enabled via socket communication. In addition, if the host can reach the Internet, so can the containers as well. This is not what was intended in terms of providing *"defense in depth"* via application isolation. Moreover, this also eliminates the possibility of segregating pipelines of different tenants.

Another *docker networking* approach is where custom *virtual bridges* can be formed per tenant pipeline as shown in Figure 5.5. With this approach a dedicated *virtual network bridge* can be created per tenant and each container in the pipeline is attached to this dedicated bridge. It is also noteworthy that in this mode the containers are fully isolated from the host networking stack. Thus, with this setup we gain true isolation of pipelines for different tenants. However, we still do not have control over restricting communication between containers *(applications)* that fall within the same pipeline (connected via a single bridge). This will require re-writing of a new docker networking plugin to support dynamic controls on the bridge. In addition, scaling

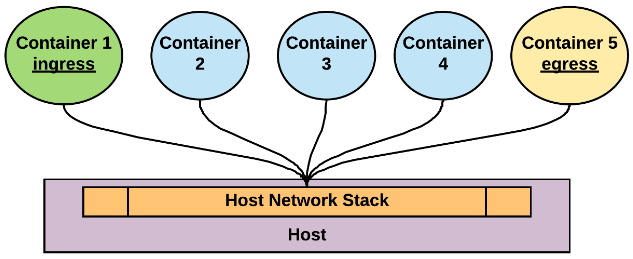


Fig. 5.4 Docker Networking: Host mode

and updates to the plugin is restricted by the level of control enabled by *docker* on its networking stack.

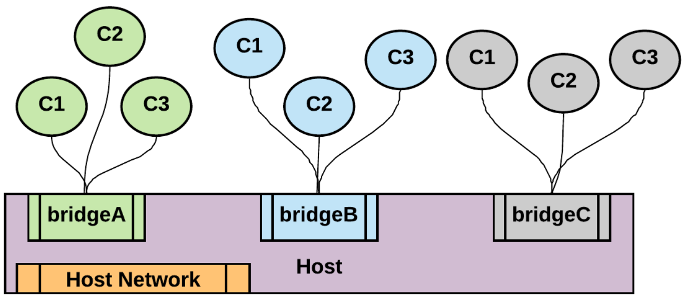


Fig. 5.5 Docker Networking: Custom bridge mode

Given, the drawbacks of the two default networking options of docker we devise a different solution to enable inter-application communication in our framework. We look towards a fully **software based** networking option to have finer control over communication between same-pipeline applications. Our design is based on pure *software defined networking (SDN)* principals.

Software Defined Networking (SDN)

SDN technology is an approach to facilitate dynamic network restructuring and management programmatically. This is unlike the traditional networking setup which is static and decentralized. An SDN based networking architecture enables dynamic re-wiring of the networking nodes and also enables enforcing fine tuned controls over the communication channel. Using an SDN based networking model gives us the freedom to (programmatically) manage the communication between the nodes and also supports for monitoring and analysis. Figure 5.6 is a depiction of a traditional SDN setup. All the communicating nodes are connected to a switch. An SDN switch is just like any traditional switch with additional support for the *OpenFlow* protocol. A **controller** program listens on a specific port *(default:6653)* for control messages from the switch. These control messages are sent via the OF protocols. The controller can monitor the activity on the switch and also can publish control rules to the switch which will be enforced for any traffic flowing through the switch. Whenever, the switch encounters a packet from a previously unseen host, it forwards it to the controller, which then decided what to do with it.

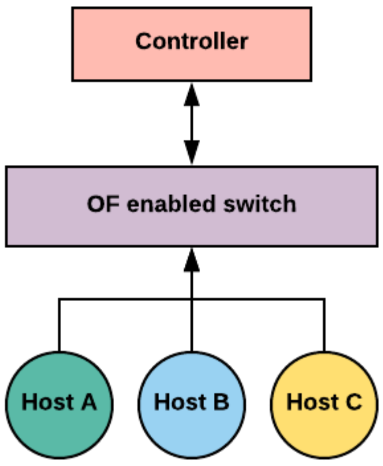


Fig. 5.6 An example of a traditional SDN setup

SDN based Inter-Container Communication

Our framework proposes a design where all containers are initially started completely isolated from the network stack. At this point no container *(i.e. application)* can reach any adjacent application. We then connect all these containers to a **virtual switch** [7], setup in the cloud host. Figure 5.7 is depiction of how this is achieved. The *virtual switch* could be any implementation that supports the *open-flow* protocol, making it compatible for SDN. This virtual switch is isolated from the host network and is not allowed external/Internet connectivity. The idea is to provide the minimal channels for inter-application communication shielding the applications from any unintended cross-communication. We then control communication between these containers by dynamically publishing control rules to the switch via the *flow-controller module* (which we discuss later in this chapter). This mechanism enables control over communication between the applications with as little overhead as setting up a single virtual switch as the bridge between them.

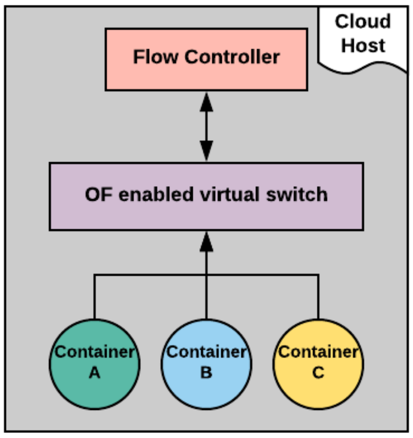


Fig. 5.7 Inter-container communication via OF enabled virtual switch

Hostname resolution

Upon achieving **application-isolation** and **inter application communication**, we still have to figure out a way for *hostname resolution*. That is, we cannot allow application developers to freely use any `IP` address (in their code) to reach other applications. This will be a mechanism for security breach and will restrict us in how dynamic our framework can be. In order to enable application development in a fully framework agnostic manner, we must decouple how applications use `IP-addresses` from its code base. We achieve this by enforcing the application developers to use pre-defined **hostname** strings in their final revision of the code. Proper *hostname* string usage must be verified via thorough code reviews before going into production. The framework, takes care of dynamically mapping these application-hostnames to correct `IP-addresses` at the isolation boundary (i.e. the container boundary). Similar applications of different pipelines may share the same hostname, but the framework dynamically binds them to different `IP-addresses` and ensures there is no conflict. If an incorrect *hostname* string is used then the framework would disallow the workflow to continue.

5.2.3 Isolated Pipeline Allocation

In the preceding sections we described how the framework achieves **application isolation** and **inter application communication** across the isolation boundary. Now we describe how the framework supports isolation of pipelines for different tenants. As described earlier the isolated containers are all stitched together via the OF enabled virtual switch. At this point by default all communication is allowed. However, we define pipeline isolation by restricting applications of a specific tenant to a communication pool. Per-tenant *(per pipeline)* control rules are maintained in the switch separately. Each pool enforces rules specific to the applications of the pipeline it corresponds to. When an application attempts to communicate to another application, the switch first delivers the message to the pool it corresponds to. Then, control checks in that pool are executed to see if the flow is allowed within that specific pipeline. Thus, on a single virtual switch, we manage multiple isolated pipelines via separate control pools.

5.2.4 Ready State Notification Mechanism

The next important aspect of the framework is managing how application status is monitored to schedule data flow without intermediary stoppages. A problem observed in the build up to the framework was that with dynamically spawned application instances it is difficult to tell when exactly an application is completely up & running and is ready to receive data. The framework needs to know if all the applications in the pipeline have reached a **ready-state** before it can initiate the data processing. We handle this by implementing a notification mechanism by observing application logs. The application developer is expected to provide the final **start-up log** as part of the documentation of the source. The framework then uses this as the monitoring point to confirm that the application is **ready**. We can also enforce a specific *startup log template* as a coding practice to all developers, enabling a unified *ready-state log* pattern for all applications handled by the framework. The framework maintains a *state-queue* per pipeline and waits until all applications in the queue have reached the **ready-state**. Once, that is achieved the data processing is started.

5.2.5 Termination of the Pipeline

The final building block of the framework is a proper mechanism to deduce when to terminate the pipeline and clean resources. For applications that process data as a batch and transfer them to the application next in line, we can use a *log based* notification mechanism as described above. However, most scenarios within the use case boundary of this study have been applications that **stream-in** and **stream-out** data. For such streaming scenarios, an **end-of-stream (EOS)** notification needs to sent. The identification of the end-of-stream is very difficult unless an EOS message is piggybacked at the end of the flow. Even, then it gets more complicated in that, each application needs to know how to handle this message and it must be expected of the developer to program this. Moreover, this only works for linear streams. In case of a parallel stream, it is difficult to tell when all of them have ended. Thus, the termination mechanism is left to be handled independently as best suited for the workflow in question. Since, we had focused our implementation on an actual data-upload and data-processing pipeline (with a few set applications), we introduce a termination mechanism that suits this flow. We also generify this approach to be suitable for any file based streaming flows. In our approach we force input a **termination file** to the flow which can be understood by the pipeline applications. The penultimate application that writes data to the *egress datastore* checks the streamed data for a **termination string**. If such is found then a termination event is triggered. We explain the specifics of how this is handled in the sections that follow.

5.3 Architecture and Implementation

In the previous section we described the design considerations for the framework and how we address each one of them. In this section we produce the entire architecture of the framework as a whole, by putting together each of the sections discussed earlier. We also provide implementation details as to how we handle each scenario. We build our framework based on an actual data-processing pipeline as shown in Figure 5.8. We then run multiple tests on this flow to evaluate the overhead incurred in using our proposed framework. As shown in Figure 5.8,

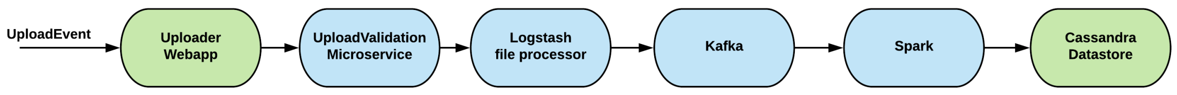


Fig. 5.8 A workflow example for data upload and processing pipeline

the flow starts by *data-upload*, then the data is processed through various applications of the pipeline and finally is written to a *Cassandra* datastore. We build the framework to align with this specific flow and enable hooks to extend it to any similar flows. First, we describe the architecture as a flow of events for a generic data-processing pipeline and then extend the discussion to the specific flow shown in Figure 5.8, with implementation specifics.

Figure 5.9 is the complete view of the framework architecture. We start by establishing that the processes running as normal processes (denoted as bare-processes) in the cloud host are the: **Container Manager (CM), Flow Controller (FC), Docker and MQTT broker**. Note that none of the applications of the actual pipeline are running as bare-processes. We also note that all communication between the two framework modules *(flows 4, 5, 7 and 15)* and between the *CM and Ingress (flows 2 and 7)* are done using the MQTT protocol. This was a choice keeping in mind the ease-of-scalability of the framework to integrate newer modules. MQTT is also lightweight in the event of distributing the framework across multiple nodes as a future improvement. Finally, we setup a virtual-switch (VS) in the system to manage and control the

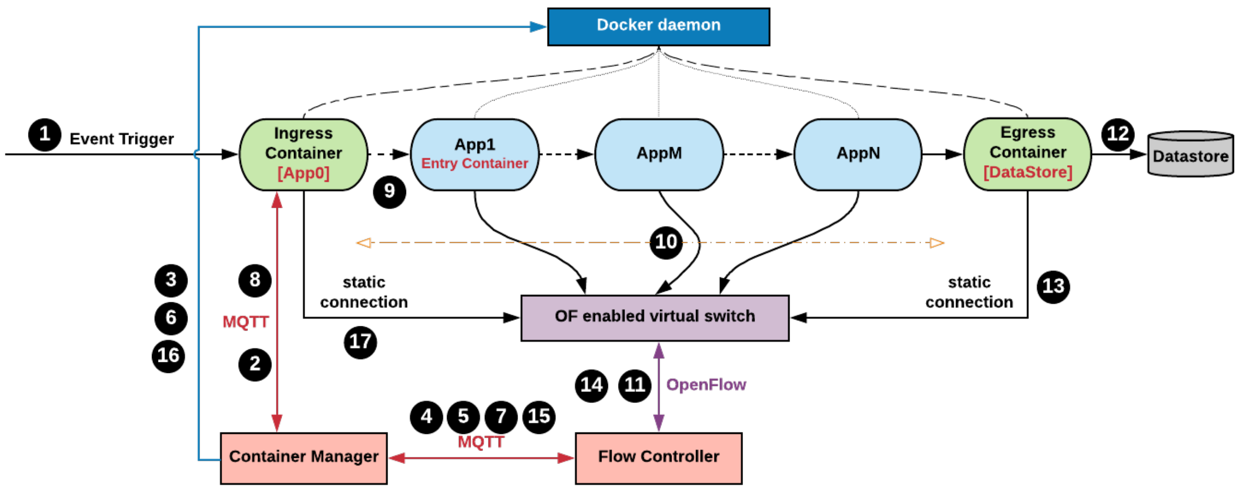


Fig. 5.9 The architecture and flow diagram of the framework

communication channels. The *flow-controller* module listens on the *virtual-switch* for *open-flow* messages indicating traffic that flows through *(flows 11 and 14)*. The *virtual-switch* is fully isolated from external/Internet connectivity.

We start by isolating the two border containers - **ingress and egress** - via containerization. These *border containers* are set to keep running throughout the lifetime of the system. I.e. they are not spawned and terminated for every incoming event. This also means that all flows initiating from here are separated into different pipelines for different tenants. This design choice is required since the border containers are either the user-facing applications or the ones that interact with a datastore. Thus, they need to be active at all times and are useful isolation boundaries at which important security checks can be implemented. Since the **ingress** container shares the lifetime of the system, we assume that the application run within the *ingress* is developed and reviewed with higher priority than the normal pipeline applications. We also, have the *ingress and egress* containers attached to the *virtual-switch* with static IP addresses assigned.

5.3.1 Flows 1 to 6

As an event is triggered *(flow 1)* in one of the *ingress* applications *(ex: data-upload)*, an **event handler** is created with a unique `EVENT_ID`. Then this event is notified to the *container-manager* via a simple MQTT message *(flow 2)*. The *container-manager* then loads a pre-defined **configuration** file (like the example shown in Appendix:A.1). This holds, all the necessary information about all the different workflows supported by the framework. Based on this configuration the *container-manager* spawns the required application containers (for this specific workflow) by contacting the docker daemon *(flow 3)*. These containers are spawned in **network=none** mode making them completely isolated. At this point the spawned containers are also attached to the *virtual-switch*. However, no traffic is generated since the pipeline-applications within the containers are yet to be executed. Upon, successfully spawning the containers, the *container-manager* forwards the necessary information to the *flow-controller (flow 4)*. A sample of the information forwarded to the *flow-controller* is shown in Appendix:A.3. The *flow-controller* then builds the necessary data-structures to handle this specific event/pipeline and responds `OK` to the *container-manager (flow 5)*. Now, the *container manager* **executes** the pipeline-applications within their specific containers by contacting the docker daemon *(flow 6)*.

5.3.2 In between Flows 6 and 7

While most applications have a purely stand-alone startup, some applications are dependent on another application running. The framework handles this by explicitly requiring the *execution-order* of the pipeline-applications to be included in the configuration. Upon execution, the framework redirects all the logs produced by the pipeline-applications to a pre-defined location within the container boundary. The log output of these applications are monitored for a pattern matching the "`start-up log`" parameter of the configuration (as explained in the design section above). Upon a match being found the container sends a `UDP` packet to the *virtual-switch* indicating the **ready-state** of that specific container-application. Since, this `UDP` packet will be one of the first of its kind seen by the *virtual switch*, it will be forwarded to the *flow-controller*. The *flow-controller* recognizes this as a **ready-state** packet and sets the state of this specific application in an internal queue to "`READY`". The **ready-state** packets contain the following information: `||<EVENT_ID>:<CUSTOMER>:<HOSTNAME>:READY||`. In addition, the *flow-controller* also publishes a control rule to the *switch*, forcing all future `UDP` packets to be forwarded to the controller.

5.3.3 Flows 7 to 11

Once all the applications in the pipeline have reached the "`READY`" state, the *flow-controller* informs the *container-manager* that the pipeline for this specific event is ready to process incoming data *(flow 7)*. The *container-manager* in return sends a "`READY`" message (MQTT) to the **ingress** application specific to this event *(flow 8)*[1]. This "`READY`" message contains the *EntryIP* indicating the next point of entry for the data from the *ingress*. Different tenant events will have different isolated pipelines created. They will all take separate and unique routes at this point where each of their *EntryIP*s will point to different isolated containers. This is denoted

[1]Note that all the events between flows 6 and 7 are explained as a single unit without having each internal flow broken down into smaller sub-flows.

by *flow 9* in the architecture diagram above and is also shown in Figure 5.10 below. Once the

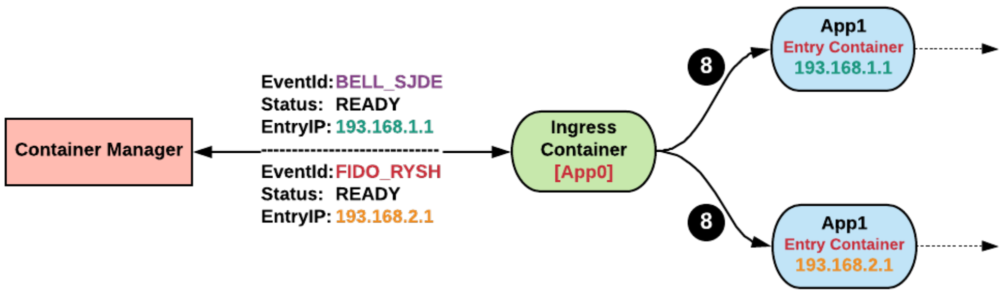


Fig. 5.10 An example of flow separation at *ingress* for two different tenant events

ingress has the `EntryIP` and starts forwarding the incoming request (data) to the first application in the pipeline, traffic is generated between all the application-containers in the pipeline. All such traffic must pass through the *virtual-switch* since the container interfaces are attached to the switch and are isolated from external communication *(flow 10)*. As packets start flowing through, the *virtual switch (VS)* checks if it has any control-rules setup for that specific packet. Since, it is the first (non-`UDP`) packet of its kind, the *VS* forwards them to the *flow-controller (FC) (flow 11)*. The *flow controller* analyses the packet and sets up appropriate control-rules for flows from that specific host *(i.e: application-container)*. The process of how the *FC* analyses an *open-flow* message and sets up control rules is explained later under Algorithm-1.

5.3.4 Flows 12 to 15

Now, applications are allowed to communicate to each other (in achieving the data processing task) according to the control-rules established by the *FC*. Any malicious or restricted crosstalk will be disallowed at the *switch*. Thus, the applications will be only allowed to send and receive data from its white-listed neighbors. Once the processed data has passed through the entire pipeline and reached the **egress** container, it will write the data to the *datastore (flow 12)*. Immediately after flushing all the data to the *datastore*, it will trigger a *"termination event*[2]*"* by sending a notification as a `UDP` message to the switch *(flow 13)*. As per the established control rules (for `UDP` packets) the switch will forward the notification packet to the *FC (flow 14)*. The *FC* analyses this packet, cleans up all internal flow-structures for this event and informs the *container-manager* about the termination-notification *(flow 15)*.

[2]The termination event is workflow specific as explained in the design section of this chapter. We explore a specific termination mechanism we employ in our experiments in Chapter 6.

5.3.5 Flows 16, 17 and Termination

Upon receiving the notification for termination for a specific **event**, the *container-manager* immediately stops (and removes) all the application-containers specific to that `EVENT_ID` *(flow 16)*. It then, triggers a **flow-deletion** message (encapsulated as a `UDP` packet) to be sent to the *switch* from within the event specific **ingress-container**. This message is denoted by *flow 17*. The *switch* upon receiving this `UDP` message immediately forwards it to the *FC*. The *FC* finds the *flow-deletion* message and publishes *flow-deletion-rules* to the switch. At this point all the data structures specific to this event are cleared at the *container-manager* and the *FC*. In addition the pipeline-containers setup for the workflow are also completely removed. The flow-rules specific to any of these containers are also deleted from the *virtual-switch*.

In the next section we explain the functionality of the *Container-Manager (CM)* and the *Flow-Controller (FC)*. We also describe how the flow controller analyses OF messages and controls flow-rules in the switch.

5.3.6 Container Manager (CM)

The *CM* acts as the conductor for the whole framework. It communicates with the *FC* to setup proper flow-control rules (on the *virtual-switch*) to isolate pipelines and applications. It also communicates with the *Docker daemon* to spawn, stop and remove containers and also to trigger events from within the containers. In addition, it communicates with the **ingress** containers to get-notified when a new event arrives and to inform them of the *entry-points* of different pipelines. All communication to and from the *CM* is carried out via the *MQTT* protocol. This is primarily because the protocol is lightweight and enables scalability of the framework with ease when adding newer modules. It also enables event and message separation simply by using *different topics*.

The *CM* operates using a **configuration file** which carries all the information pertaining to every workflow supported by the system. A sample of this file is shown in Appendix:A.1. Each workflow is given a **unique name** in the configuration which will be used throughout the system. In addition, the *CM* expects a *"handler-class"* for each workflow. It is this class that will be used to handle events of that specific workflow. The class is expected to be a sub-class ***(extends*** *class)* of the `abstract class AbstractEventHandler`. The `AbstractEventHandler` class is an implementation of the interface `EventHandler` (shown in Appendix:A.2). This interface provides a skeleton for a workflow developer to understand the tasks required to integrate a new workflow into the system. However, not all of them need to be implemented, since most of them have been already defined in the `AbstractEventHandler`. This design choice is made in order to have leeway for newer workflows to be integrated with ease.

5.3.7 Flow Controller (FC)

The flow-controller is the connection point to the *virtual switch (VS)*. It waits for *new-event-notifications* from the *CM* and sets up the required flow-rules on the *VS*. When a new packet arrives at the *VS*, it is forwarded to the *FC* encapsulated as an *open-flow (OF)* message. Algorithm-1 describes how the *FC* processes an incoming *OF* message. As shown in Algorithm-1, the *flow-controller (FC)* sets up all the event specific data-structures first. The *FC* does this when it receives an event-notification from the *CM* (denoted by *flow 4* in the architecture diagram). Now, when the *FC* receives an *OF* message from the *VS* it checks the originating *port*[3]. It then stores a mapping between the *vs-port* and the container IP.

Next, the transport packet of the OF message is evaluated. If the transport protocol is `UDP` and the packet did not originate from the *switch* itself, we know that it must have arrived either from one of the application-containers in the pipeline or a *border-container*. If it originated at a *border-container* then it is understood that the packet carries one of the two *pipeline-**event**-notifications:* **Event Termination** or **Delete Flow-Controls**. The type of the notification is deduced by looking at the contents of the packet-payload (shown below).

- `Event Termination: ||<EVENT_ID>:<CUSTOMER>:<ERROR>:<TERMINATION_MSG>||`
- `Cleaning Flow-Controls: ||<EVENT_ID>:DELETE_FLOWS||`

If the `UDP` packet had originated from one of the application-containers, then it carries an *application-**ready-state**-notification*. The contents of a *ready-state* payload looks as follows: `||<EVENT_ID>:<CUSTOMER>:<HOSTNAME>:READY||`.

Event Termination and Cleaning Flow-Controls

From the two *pipeline-event-notification* messages the first one to arrive would be the **event-termination** message. This is sent by the **egress** container once a termination condition is met *(e.g.: a Cassandra trigger executed upon finishing all writes)*. Upon receiving this message, the flow controller cleans all internal data-structures pertaining to the terminating event and immediately informs the *container-manager*. The *CM* stops all application-containers of the specific event (as described earlier) and triggers the **"clean-flow-controls"** `UDP` packet to be

[3]Note that this port is not the usual TCP port, but the switch port to which the host (in our case the container) is attached to. This will be used later to clear all control-rules when an event has terminated.

sent from within the event-specific **ingress** container. When the *FC* receives this message, it sends control-messages to the *virtual-switch* to clear all flow-rules set-up for the specific event.

Algorithm 1: Handling *Open Flow(OF)* messages from the *Open Virtual Switch(OVS)*

```
1  Function set_up_event_specific_structures();
2  Function receive_OF (ovs_switch, OF_message, ctx);
   Input  : Switch information, OF message and the message context
   Output: Flow controls published to the OVS
           Or container state published to container-manager
3  if (NetworkProtocol == IPv4) then
4      addSwitchPortToSrcIPMapping(srcIP, inPort);
5      if (TransportProtocol==UDP && srcMac!=OVS_Mac) then
6          if ( isBorderContainer(srcIP)[a] ) then
7              processEVENTStatusUDP(UDPPacket);
8          else
9              allReady ← processREADYStatusUDP(UDPPacket);
10             if (allReady) then
11                 setupContainerSpecificFlowEntries();
12             end
13         end
14     else
15         if (( isBorderContainer(srcIP) && isNeighbourOfBorderC(dstIP) )
16         OR ( isBorderContainer(dstIP) && isNeighbourOfBorderC(srcIP) ))
            then
17             addAllowIngressToNeighbourFlow();
18         else if ( isOVSSwitchMAC(srcMac) OR isOVSSwitchMAC(dstMac) ) then
19             addAllowFlowsToAndFromOVS();
20         else
21             setupContainerSpecificFlowEntries();
22         end
23     end
24 end
```

[a]Note that for the purposes of describing this pseudo-code the **ingress** and **egress** containers are called the **border containers** and their neighbors *(i.e. entry-container and the penultimate container in the pipeline)* are called the **neighbors of the border containers.**

Ready-State Notification

If the incoming `UDP` packet is a *ready-state* notification, the flow-controller first looks up the specific event. Then it switches the "**state**" of this event-specific host (the container which sent the notification) to be "**`READY`**". In the meantime, it also checks if all other containers of this event have reached their "`READY`" state. If **yes**, then flow entries specific to this container are setup. How the flow-entries are setup is explained below in this chapter under the section *Publishing Flow Entries*.

Non-UDP Traffic

If the incoming *OF* message contains a non-`UDP` packet, then the *FC* does the following checks:

1. If the communication is at the edge of a pipeline *(i.e. to or from an ingress or egress)*
2. If the source or the destination of the message is the *virtual-switch* itself
3. Any other flow

`[CASE 1]`
If the packet is from a *border-container*, then the *FC* immediately checks if the `destinationIP` is of a container that is a neighbor to the *border-container*. If **yes**, then control-rules are published to allow communication from that *border-container* to this `destinationIP`. The vice-versa of the above applies to `(sourceIP)` packets that are destined to *border-containers* but originate from elsewhere. These *border-container* specific flow-entries are published to the default `ZERO` flow-table of the *virtual-switch*.

`[CASE 2]`
If the packet is from the *switch* or is destined to the *switch* then those flows are also allowed. Thus, the *FC* publishes a control-rule to allow such communications. Flow entries specific to flows related to the switch are also published to the default `ZERO` flow-table.

`[CASE 3]`
For packets that do not match the previous cases, flow-control rules are published according to their neighbors in the workflow. This information is found in the **configuration-file** provided to the *container-manager* that explains all the flows in the framework. When a new event is triggered this neighbor information is also sent to the *FC* by the *container-manager* in its initial message *(flow 4)*. The *FC* uses this information to publish flow-control rules for the `IP` from where the current packet originated.

Publishing Flow Entries

From the above description we saw that if all the containers of an event have reached the "`READY`" state or if non-of the first 2 cases of non-`UDP` traffic matches the incoming packet, then flow-entries are created for that IP. Flow-entries specific to an application-container are published to a **unique-flow-table** dedicated to that container. The *FC* maintains an internal map of *flow-tables* and corresponding application containers. Per `IP Address` the *FC* publishes 4 different flow-entries:

- `[GOTO Flow Table]` This rule is published in the *default table-`ZERO`* to redirect rule checking to continue in the *container specific flow-table*
- `[Allow traffic to neighbors]` A rule is published per neighbor of this container *(as received from the CM)* to allow traffic to and from them. Published in the *container specific flow-table*
- `[Allow UDP traffic to OVS]` Rule allowing all `UDP` traffic to the switch from the container, This is to support notifications from the container
- `[Drop all other traffic]` Rule to drop any other unintended traffic

That concludes, the discussion about the complete architecture of the proposed framework. In the experimentation and evaluation sections of the study we elaborate on how an actual application pipeline was tested using this framework.

Chapter 6

Experimental Results and Discussion

In this section we describe the framework setup and its experimental results for an actual industry use case. The mechanisms used in the framework to achieve defense-in-depth are explained in the previous chapters. In this chapter we discuss the overhead incurred (in the system) by adopting this framework and its defense mechanisms. We first define the industry use case based on which the experiments were conducted. Then we explain the server configurations and environment of the cloud host. Finally we discuss different performance metrics in terms of overhead incurred when comparing non-framework and with-framework setups.

6.1 Experimental Use Case

We focus our framework's adaptation for a specific industry scenario that intakes different tenant data, processes it through a series of applications and finally produces some results. There are three different kinds of workflows associated with the above use case.

1. A user triggers an event *with some data*; the data gets processed via a set of applications and is written to a datastore
2. A user triggers an event *for some data*; data is fetched from the datastore and gets processed via some applications and is produced as results to the user
3. An event is triggered as a routine job (internally); data is fetched from the datastore and is processed via some applications and is re-written to the datastore

Figure 6.1 is a depiction of the three workflows listed above. We denote a single instance[1] of one of the above workflows as "a pipeline". As explained in Chapter 5, the applications at the edges of such a pipeline are called *ingress* & *egress*. An ingress application is user facing and is the point of trigger for any new event. The data is transferred from application to application

[1]We say "single instance" because as explained in Chapter 5 and Chapter 4, different events are processed by a dedicated set of similar applications.

through the pipeline until it reaches some egress point. Usually, the egress is an application at the end of a pipeline that writes the processed data into a datastore. However, for workflows 2 and 3, the ingress and the egress will be the same applications. This is because the application at which events initiate is also the application where the events will terminate. From the three listed workflows above, we run our tests against the first workflow where some data comes into the system, is processed and is written to a datastore. Whilst, the framework allows for easy adaptation of new workflows, the results of one of them should validate the performance overhead for all since they have similar pipeline structure: *An event trigger, One IO bound application and multiple intermediary applications*.

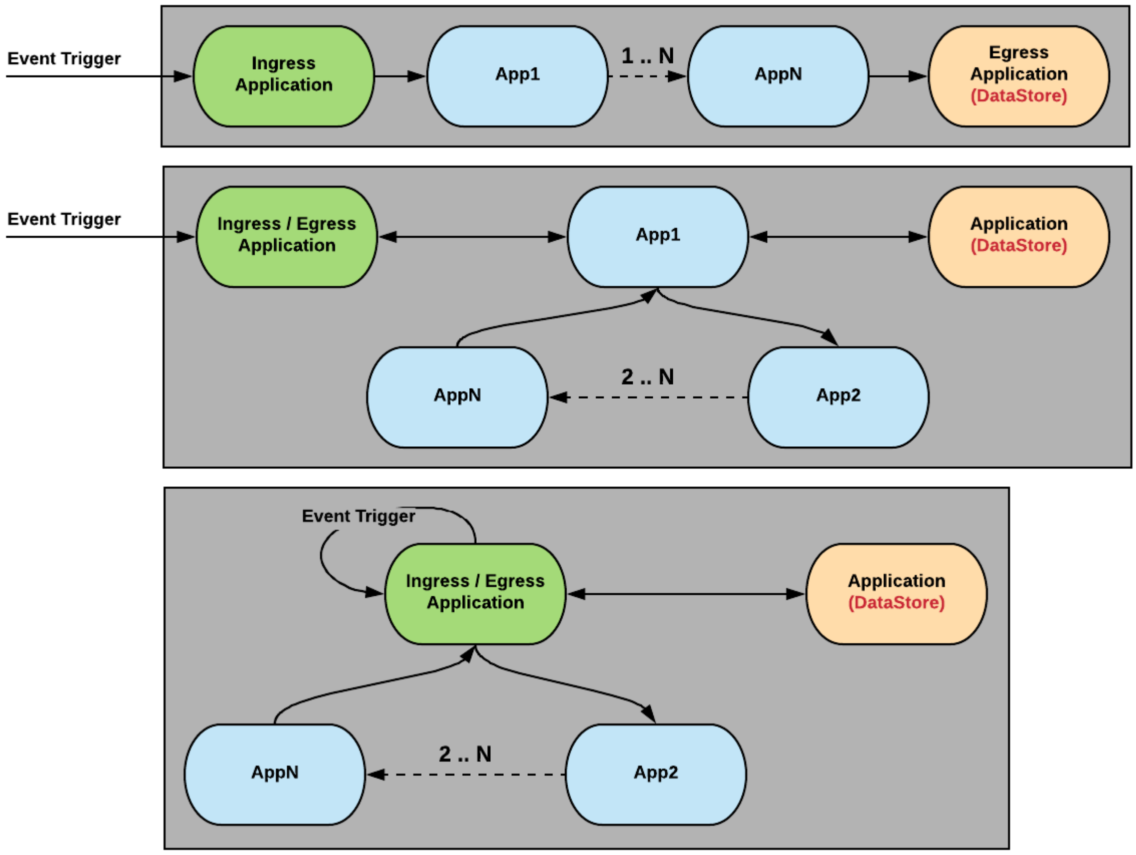


Fig. 6.1 Three different types of workflows specific to the experimentation use case

Figure 6.2, shows the actual applications involved in the sample pipeline we have used in our experimentation. We start our experimentation with the same setup as shown in the figure. Then, we extend this setup by increasing the number of intermediary applications in the pipeline to further test the framework.

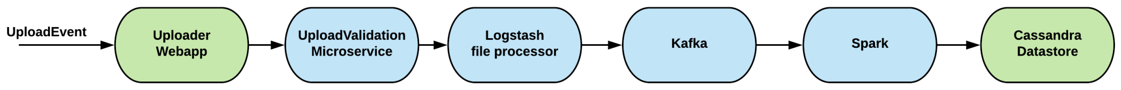


Fig. 6.2 The “Data upload” workflow

Data Upload Flow: The flow starts with a user *(tenant)* uploading their data to the ingress application. The ingress in our setup is a web application which receives the upload and forwards it to a microservice. This microservice validates the incoming upload against the correct tenant. Upon validation, it stores the uploaded data into a specific directory on which *Logstash* [10] is listening for changes. In addition the microservice also spawns a *Spark* [52] job. Logstash, upon observing new uploads in the said directory, reads this data, processes it and publishes the results to *Kafka* [6]. In the meantime, the spawned Spark job retrieves the data from Kafka and writes it to *Cassandra* [27]. Figure 6.2 shows the order in which the data flows from application to application. However, in our containerized setup the *Spark* process runs in the same container as the *Upload validation microservice*. This is because it is the microservice that spawns the spark job. Hence, when these applications are containerized they make a logical flow connection as shown in Figure 6.3.

6.2 Host Server Configurations

The setup and experiments were conducted in one of Systems Group’s[2] compute clusters. The server runs on *Ubuntu 16.04* and has 32 CPUs of Intel Xeon 2.40GHz. It also has a 20Mb L3 cache and a main memory capacity of 125Gb with an additional swap space of 15Gb. The server is shared between multiple students of the systems group. Hence some of the experiments had to be done while the server was being used for computations of other students. However, almost all test cases were run with a maximum effort on ensuring that the *load average* of the server from other computations was $\approx 6.25\%$. That is ≈ 2 CPUs on average were always occupied by other processes in the system. Version 17.05 of the *Docker* containerization tool was used along with *Open vSwitch* $v2.7.0$. The *Mosquitto broker* $v3.1$ [30] was used as the MQTT broker for communication between the framework modules.

[2]**bmj-cluster** owned by the Systems Group of McGill University - School of Computer Science.

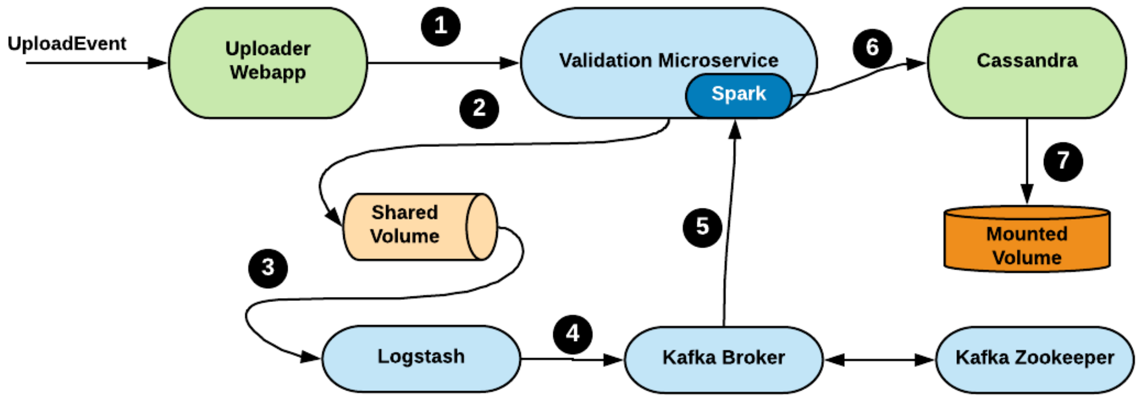


Fig. 6.3 Containerized application setup for the "Data upload" flow

6.3 Results & Discussion

All tests were executed on two different setups. One, with a deployment based on our framework and the other without it. We denote the usual deployment (without our framework) as "'bare-metal" setup while the one adopting our framework is called the "framework" setup. Figure 6.2 portrays the bare-metal setup where all applications run as normal processes on the host, while Figure 6.3 shows the containerized application arrangement in our framework. We gathered the results for both setups and compared the performance overhead incurred in adopting our framework against the bare-metal one[3]. Multiple batches of different workloads were tested in both cases.

6.3.1 Runtime Analysis

We perform runtime analysis by first measuring the time taken for complete flows - *from Ingress to Egress* - in both setups. For the setup based on our framework, we separately identify the time taken for different steps of the pipeline. This is because, with our framework there is an *initial* and *terminal* delay.

- **Initial delay**: This includes the time taken to *spawn containers* and to *publish flow controls* to the switch before the actual data processing starts.
- **Terminal delay**: This is time taken to terminate all containers and to clean all flow controls from the switch upon completion of the flow.

The actual data flow and processing happens in between these two delays. We denote this portion of the framework-flow as the *"Actual flow"*. To calculate flow overhead we compare the *actual*

[3]The term *both-setups* always denotes the framework and bare-metal based deployments.

flow time against the *bare-metal flow time*. An example of these two time measurements (for a specific load) is shown in Figure 6.4. We then denote the difference between the averages of those times (for a specific load) as *"Runtime difference"*. Such runtime differences (in seconds and percentages of the total time - *RDF and RDF%*) are shown in Table 6.1.

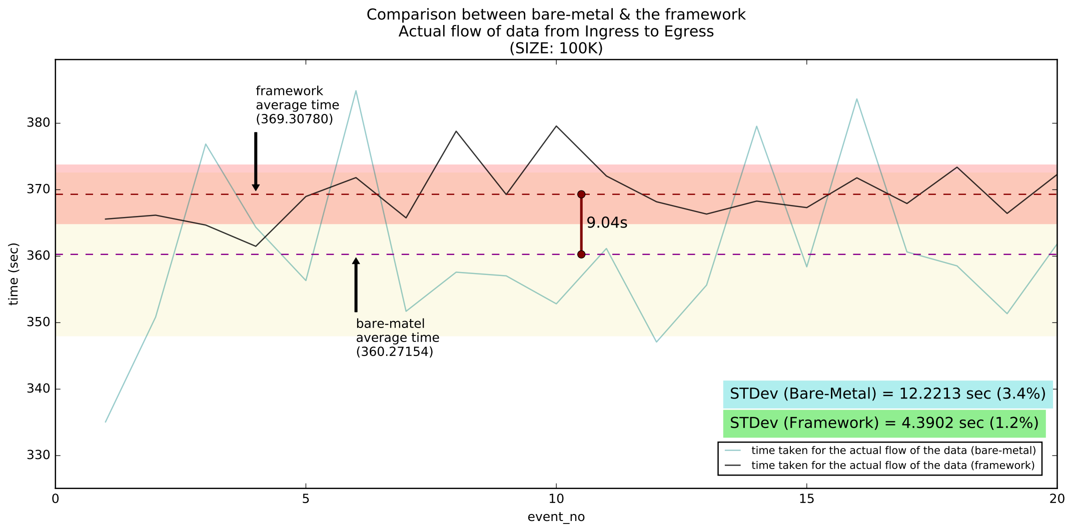


Fig. 6.4 An example of runtime measurements for a specific load

Load	**10K**	**25K**	**50K**	**75K**	**100K**	**200K**	**500K**	**750K**
Runtime difference *(RDF)*	0.52	1.22	2.44	5.94	9.04	33.20	90.07	125.19
Overhead (RDF%)	0.92%	1.10%	1.24%	2.09%	2.45%	4.56%	5.09%	4.67%
Container spawning (CP)	4.46	4.52	4.38	4.26	4.46	4.80	5.00	4.53
Publish flow-controls (PFC)	11.84	11.97	11.79	11.73	11.60	13.57	12.98	11.68
IO = CP + PFC	16.24	16.50	16.18	16.00	16.07	18.38	17.99	16.21
Termination overhead (TO)	1.71	1.70	1.67	1.66	1.62	1.64	1.85	1.75
TSO = IO + TO	17.96	18.20	17.86	17.66	17.69	20.02	19.84	17.97

Table 6.1 Runtime overhead incurred for the proposed framework *(in seconds)*

TSO: Total setup overhead
IO: Initialization overhead

In addition to the *actual flow*, Table 6.1 also provides the time overhead in terms of the *initial* and *terminal* delays. Rows in yellow *(CP, PFC, IO)* denote the overhead incurred during the initialization phase while the penultimate row *(TO)* shows the overhead at the end of a flow. We can see from the results that the time taken for container spawning (CP) and to setup flow-controls (PFC) for any load is $\approx$ 5s and $\approx$ 12s respectively. In addition, the time overhead at the end of a flow is also always $\approx$ 2s. Thus, it is evident that the framework setup specific overhead (at the beginning and at the end of the flow) is constant around $\approx$ 19s. This does not grow with the load with which the framework is stressed. This is as expected since, the steps involved in setting up the framework is not dependent on the load. Similarly, we observe that the overhead in terms of the actual flow (RDF) is also between $0-5\%$ for varying loads.

6.3.2 Resource Utilization

For studying the resource utilization we measure *CPU* and *Memory* usages of both setups. We use a combination of multiple tools to measure these resources. For both setups we measure the above resources using the *System Activity Monitor (sar)* [24] tool. In addition, for processes that run on the host (without containerization) we also use the *CPU%* and *MEM%* outputs of the **top** utility. For applications running on docker (in a containerized environment) the **docker stats** utility is used. These additional measurements are used since the output of *sar* is the total resource utilization of the system. However, as mentioned earlier the server is shared amongst many students of the systems group. Thus, we use these additional tools to deduce the correct measurements for the processes of our setup. We achieve this via conducting our measurements per process and per system user.

Our measurements are carried out at every 3 second interval over the duration of a pipeline. The interval is set to 3, since the smallest load takes on average $30-35$s to complete; thus to ensure at least 10 readings on the smallest experiment we set it to 3. This experiment is carried out approximately for 100 iterations per system load and the average over these measurements is reported. It is also noteworthy that the outputs of the above tools are only an approximation of the instantaneous CPU usage over a most recent past. We also carry out our experiments throughout the day one after the other. Hence, some of the results indicate some skewness when colliding with scheduled *sys admin* activity or some docker based computations of another user. Figures 6.5 & 6.6[4] show comparisons of CPU and memory utilization between the two setups. The annotations on top of each test case report the resource overhead in our framework when compared against the bare-metal setup. We observe that the CPU utilization overhead *(Figure 6.5)* fluctuates between $5-8\%$ while for a single test case it reaches above 10%. We believe this to be a case of our experiments conflicting with some other running docker containers.

[4]The numbers that were used to draw these charts are provided in Appendix-A.1 & A.2.

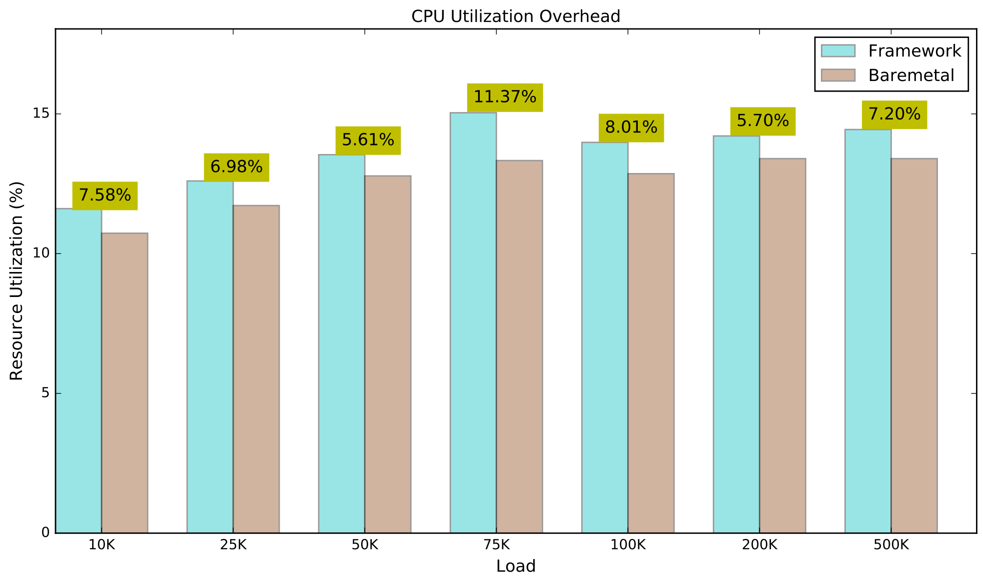


Fig. 6.5 Comparison of *CPU* utilization
(The numbers above each bar denote the overhead percentage)

The main point of strain in our framework is the docker eco-system and the management of containers. The host-processes[5] add minimal CPU overhead *(Appendix-A.1)*. From the overall, CPU utilization in our framework only 4% of it was due to the host-processes and the rest was from docker.

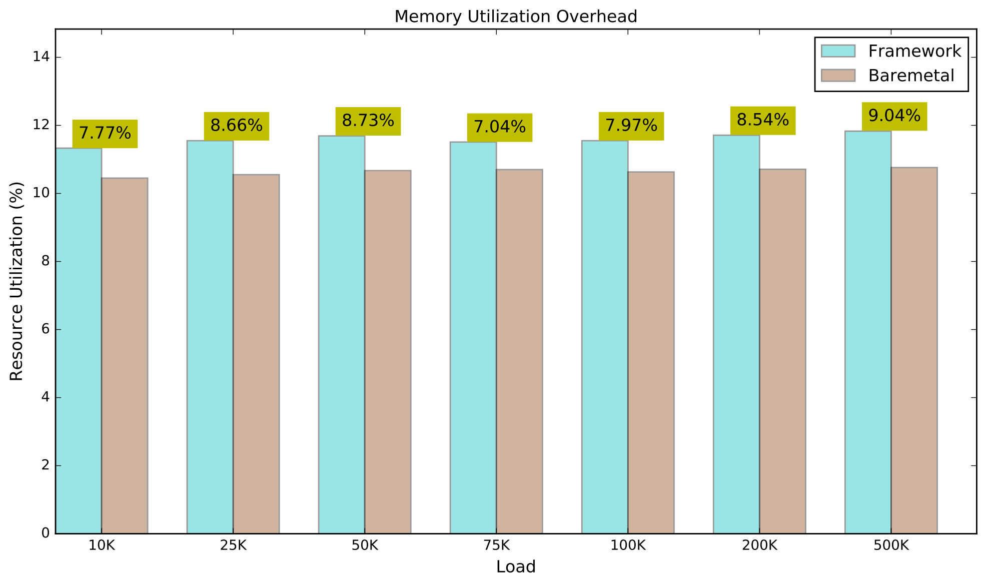


Fig. 6.6 Comparison of *Memory* utilization
(The numbers above each bar denote the overhead percentage)

[5]The controlling modules - *container-manager and flow-controller* - are the ones denoted by the term “host-processes” since they are the only un-containerized applications during a flow using our framework.

From Figure 6.6 we can see that the overall memory footprint overhead is within $7-9\%$. The main memory utility for different workloads are almost the same for both setups. This is because, the nature of all the applications is that the uploaded data is streamed (from file) one after the other in a sequential form. Thus, the contributors to memory overhead are the host-processes and docker itself.

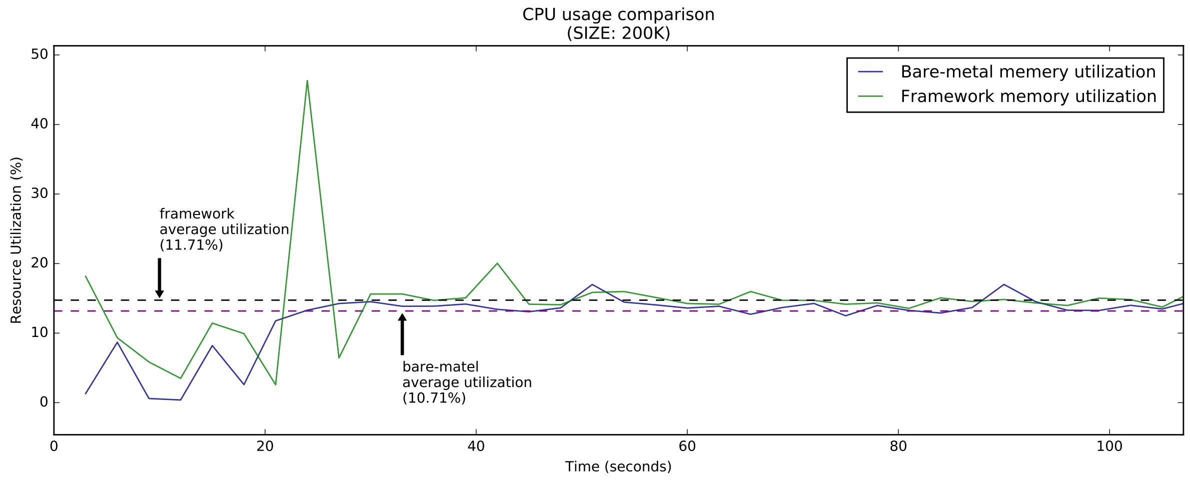


Fig. 6.7 An example of *CPU* utilization measurements for a specific load

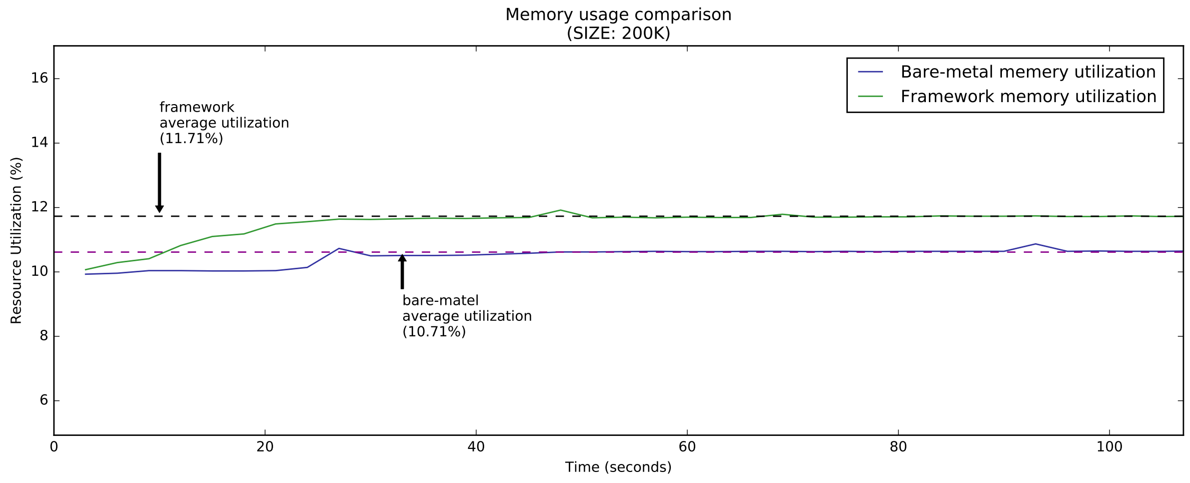


Fig. 6.8 An example of *Memory* utilization measurements for a specific load

Figures 6.7 and 6.8 are examples of a single experiment for both resources discussed. The evident spike in Figure 6.7 ($\approx 23sec$) is the time around which the uploaded data is transferred to the first application from the ingress. This is roughly the point during which the data enters the container eco-system. This can be cross referenced with Table 6.1 where the average initialization delay for loads of $200K$ is ≈ 18.4s.

We can see from the above results that the resource utilization (CPU and memory) are within a strict range of $5-10\%$. The average overhead across all different experiments is 7.49% for CPU and 8.25% for memory. It is noteworthy that the above resource utilization measures are only present during the time of an ongoing event. With our framework, when there are no active events in the system, none of the applications will be active (because of the on-demand container provisioning). The only long running processes are the host-processes and the docker daemon. Thus, the above overhead measures would not be a constant strain on a typical cloud deployment. On the other hand, with a bare metal setup, all applications would be continuously running at all times. Even though, they might not be doing any computation they would take up considerable amount of memory and CPU while in an active state. It addition, we recall the possibility of them behaving maliciously during idle time.

6.3.3 Pipeline Extension Based Performance Analysis

In addition to the above analysis we also conduct further experiments by extending the number of applications in the pipeline. We do this by replicating the number of *Logstash* instances in the original pipeline shown in Figure 6.3. We define, *"extension count"* as the no of logstash instances for each experiment. We test, pipelines of different *extension counts* using a single constant load **(of 100K)** to see how it performs. Table 6.2 shows the results for runtime analysis of five different extension counts.

Extension Count	**1**	**2**	**3**	**4**	**5**
Runtime difference *(RDF)*	9.03	39.40	40.43	31.56	44.76
Overhead (RDF%)	2.45%	10.30%	10.22%	7.82%	10.57%
Container spawning (CP)	4.46	5.96	6.61	8.38	9.94
Publish flow-controls (PFC)	11.60	13.13	13.91	16.31	18.05
IO = CP + PFC	16.07	19.09	20.53	24.69	27.99
Termination overhead (TO)	1.62	2.19	2.54	3.30	3.70
TSO = IO + TO	17.69	21.28	23.08	28.00	31.70

Table 6.2 Runtime overhead incurred with varying *extension count (in seconds)*

TSO: Total setup overhead, **IO:** Initialization overhead

Extension count 1 in Table 6.2 denotes the original pipeline and reports the same results of load $100K$ from Table 6.1. We can infer from it that the average of all overhead percentages (RDF%) for all the extension counts other than the original (i.e.1) has increased to $\approx 9.73\%$ [average(10.30%, 10.22%, 7.82%, 10.57%)]. This is a significant rise when compared to the overhead range of $0-5\%$ for varying loads on the original pipeline *(Table 6.1)*. In the original pipeline, logstash processes the uploaded data and directly publishes it to Kafka. However, when the pipeline in extended with a new logstash instance, a new shared volume is mounted between the old and new logstash containers. It is by *writing to* and *reading from* this shared volume that these instances channel the data forward. Thus, we attribute this sudden increase in runtime overhead to the effects of homogeneous applications with shared volumes as explained by Bhimani et al. in [11]. It is also evident from our results that this sudden rise is only for the first additional logstash instance. As we increase the extension count, the RDF% is steady at $\approx 9\%$.

When we compare the initialization (IO) and termination (TO) delays, we see an average increase of $\approx 3s$ and $\approx 0.5s$ respectively. This is as expected, since with increasing number of applications in the pipeline, the time taken to setup the flow-containers (spawning containers and starting the processes) and to clean the setup upon completion also increases. Thus, we observe an average framework setup time (TSO) increase of $\approx 3.5s$ as the extension count increases. Thus, from the comparative analysis of varying loads (Table 6.1) and varying pipeline length (Table 6.2), we can conclude that the framework setup overhead is constant for a fixed pipeline while it increase by $\approx 3.5s$ with every new application added to the pipeline. Moreover, we see that the RDF%(s) for a specific type of pipeline setting *(e.g.: with/without shared volumes)* only changes by $0-3$ for both varying loads and varying extension counts.

Figures 6.9 and 6.10 show the overhead incurred in terms of CPU and memory utilization when the pipeline was extended while keeping a constant load. The average CPU utilization overhead over all different extension counts is $\approx 11.36\%$. This again is high when compared against a fixed-pipeline-varying-load overhead of $\approx 7.49\%$ *(Figure 6.5)*. We also see a steady trend of increasing CPU overhead as the number of logstash instances increases. Consider again the impact of the number of shared volume based applications (as explained Bhimani et al. [11]), on the framework. Shared volume based IO has certain performance implications with docker [11]; while the IO wait time is also reported as CPU time utilized by our measurement tools. Thus, as the extension count increases (with more shared volume based operations) we see an increasing trend in CPU utilization. Amaral et al. [2] discuss such effects and Bhimani et al. [11] provide containerization strategies ideal for such scenarios.

In terms of memory overhead (Figure 6.10) we see an average of $\approx 5.05\%$ across all extension counts. It is well within the ranges reported by our experiments with varying loads

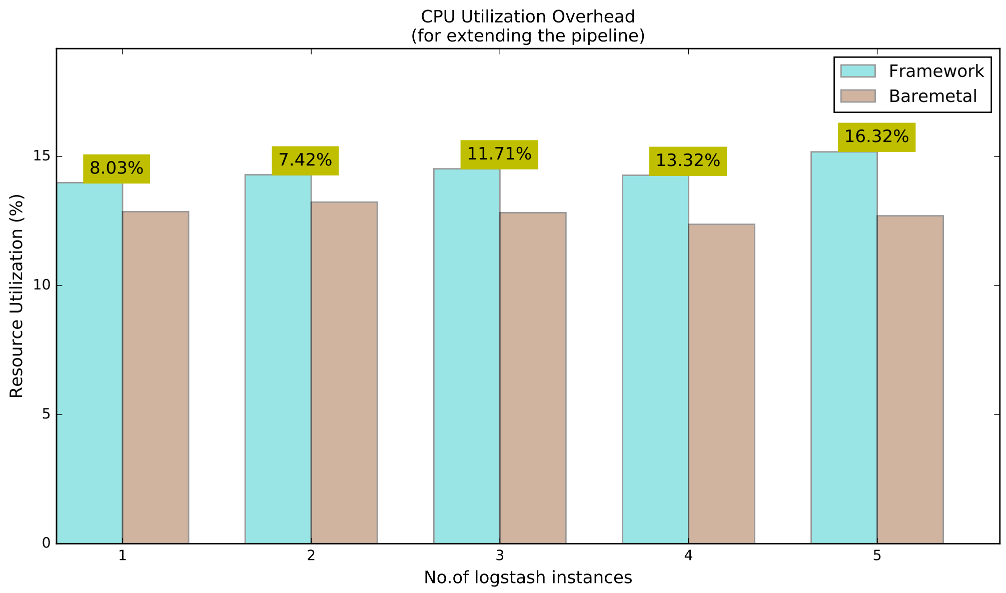


Fig. 6.9 Comparison of *CPU* utilization
(varying extension counts))

(Figure 6.6). Thus, the number of applications in the pipeline effects only the CPU overhead. This also happens only when it involves many shared volumes between the applications, which is very rare. Almost all applications transfer data and communicate via sockets since they are designed with support for deployment in distributed environments. whilst enabling access over the network.

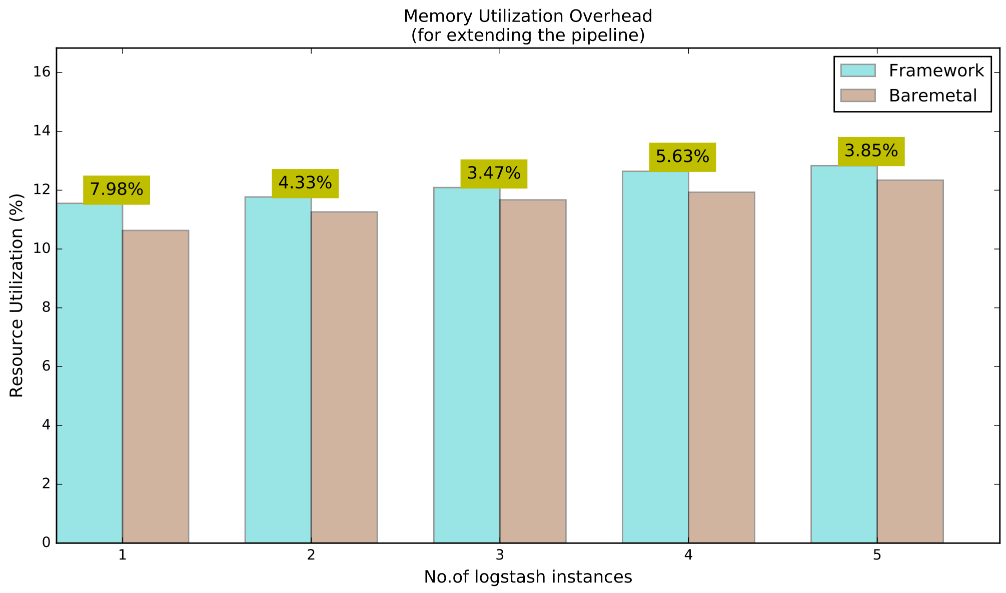


Fig. 6.10 Comparison of *Memory* utilization
(varying extension-counts)

Chapter 7

Conclusion and Future Work

7.1 Contributions and Conclusion

Service providers who adapt an SaaS cloud model require mechanisms to provide defense-in-depth to the data of their tenants and the applications that process this data. Traditional cloud security mechanisms focus on the defense of the cloud VM and data at rest. However, an additional requirement for SaaS providers is being able to capture and evade the threats posed by mis behaving applications internal to the SaaS setup. Such threats can be classified as *passive internal threats*. We provide a proof-of-concept implementation of a framework that provides security against the passice internal threat model in SaaS cloud deployments. Our framework tackles the problem of defining a boundary for applications where security checks can be made and defensive mechanisms be implemented. We build this boundary using the support for containers in the Linux kernel. We use concepts from the software defined networking (SDN) technology to enable connectivity between the isolated environments.

Our framework makes design choices based on the requirements of an industry collaborated pilot project. As part of this pilot study, we produce an authentication mechanism to validate incoming data using a hybrid encryption scheme as explained in Chapter 3. In addition, our framework provides an easy interface to add new flows to the framework by means of a centrally managed configuration. Finally, we evaluate the framework with actual workflows based on industry use cases. We then provide comparative results against a traditional deployment setup.

The results show that the said defense features can be implemented with a minimal runtime overhead of $< 5\%$ in addition to a constant setup overhead of $\approx 19s$ (for a given data processing pipeline). We see from our results that the framework imposes an additional resource overhead of $\approx 7.5\%$ CPU and $\approx 8.25\%$ memory. This overhead is only effective during active events and at all other times the framework has $< 1\%$ resource utilization. This (as explained in Chapter 6) is because container instances specific to an event are spawned on demand and at other times the system stays idle. In contrast, in a traditional deployment all applications pertaining to all workflows are kept alive throughout even though there are no active events. In addition, we also evaluate our framework against pipelines with varying number of applications. We notice that the framework setup overhead increases by $\approx 3.5s$ as the pipeline extends. The memory and CPU utilization overheads are within the same ranges reported with varying load based experiments (except for when the pipeline involves *shared volume* based processing). Finally, from our work we derive the more broader research question of whether developers can be completely relieved from having to handle application level security. That is, can an external eco-system provide the framework for supporting various security protocols with the application left only to be implemented to do the business logic.

7.2 Limitations and Future Work

The framework has the following limitations and thus room for improvement as future work:

- *Event termination detection:* In the current implementation, the event termination tracking mechanism is workflow dependent. Thus, for every workflow added a specific detection mechanism needs to be integrated. However, a more abstract mechanism could enable easy integration of various types of workflows. Moreover, a robust mechanism is required to support non sequential streaming applications, since tracking termination in such scenarios will be difficult.

- *Support for applications dependent on in-memory data:* The discussed work is based on a set of applications that do not share information between different events. This is because it was designed based on actual industry use cases (as explained in Chapter 6) that carry out completely independent tasks. However, to support workflows with applications relying on information from previous events we need a common memory persistent structure. This is a problem that needs to be addressed on its own to be integrated into the proposed framework.

In addition to the above limitations the framework design opens up avenues for various other improvements. It could be extended with a complete administrative tool with proper UI interfaces to interact with the *container-manager* and *floodlight-controller*. Interactive analysis and reporting of application behavior could be built into it. In addition, the OVS based networking supports scaling across pipelines. Thus, it could be used for nested topologies in future use cases of pipelines with overlapping applications. The use of SDN controls allows for dynamic and easy manipulation of reachability between applications. Furthermore, an interesting development would be to deduce an optimal way to encapsulate more than one application into a single container based on the logical neighbor graph of applications for a given workflow. Even though, the goal of the framework is not to be another container orchestration platform, the model could be developed into one with inherent defense-in-depth mechanisms. Nevertheless, the most significant improvement would be (as explained at the end of the previous section) to define a container boundary that can fully take control of various application security protocols.

References

[1] Ahmad, M. (2010). Security risks of cloud computing and its emergence as 5th utility service. In *International Conference on Information Security and Assurance*, pages 209–219. Springer.

[2] Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., and Steinder, M. (2015). Performance evaluation of microservices architectures using containers. In *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, pages 27–34. IEEE.

[3] Amazon (August 2016). Amazon web services - aws security best practices. *AWS Whitepapers*.

[4] Amazon (July 2015). Amazon web services - overview of security processes. *AWS Whitepapers*.

[5] Anderson, J., Hu, H., Agarwal, U., Lowery, C., Li, H., and Apon, A. (2016). Performance considerations of network functions virtualization using containers. In *Computing, Networking and Communications (ICNC), 2016 International Conference on*, pages 1–7. IEEE.

[6] Apache Software Foundation (February 2017a). *Apache Kafka*. 0.10.2 edition.

[7] Apache Software Foundation (February 2017b). *Open vSwitch*. 2.7.0 edition.

[8] Apache Software Foundation, Hykes, S., and Docker Inc. (2017). *Docker*. San Francisco, CA, 17.05.0-ce edition.

[9] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

[10] Banon, S., Elastic Search, and Apache Software Foundation (May 2017). *Logstash by Elastic Search*. 5.4.0 edition.

[11] Bhimani, J., Yang, J., Yang, Z., Mi, N., Xu, Q., Awasthi, M., Pandurangan, R., and Balakrishnan, V. (2016). Understanding performance of i/o intensive containerized applications for nvme ssds. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE.

[12] Biddle, R., Van Oorschot, P. C., Patrick, A. S., Sobey, J., and Whalen, T. (2009). Browser interfaces and extended validation ssl certificates: an empirical study. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 19–30. ACM.

[13] Bowers, K. D., Juels, A., and Oprea, A. (2009). Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM.

[14] Brodkin, J. (2008). Gartner: Seven cloud-computing security risks. *Infoworld*, 2008:1–3.

[15] Catteddu, D. and Hogben, G. (2009). Cloud computing risk assessment. *European Network and Information Security Agency (ENISA)*, pages 583–592.

[16] Chen, Y., Paxson, V., and Katz, R. H. (2010). What's new about cloud computing security. *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, 20(2010):2010–5.

[17] Cziva, R., Jouet, S., White, K. J., and Pezaros, D. P. (2015). Container-based network function virtualization for software-defined networks. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 415–420. IEEE.

[18] Cziva, R. and Pezaros, D. P. (2017). Container network functions: bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31.

[19] De Benedictis, M., Lioy, A., and Smiraglia, P. (2018). Container-based design of a virtual network security function. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 55–63. IEEE.

[20] Docker Inc. and Apache Software Foundation (2017). *Docker Swarm Mode*.

[21] Eling, M. and Loperfido, N. (2017). Data breaches: Goodness of fit, pricing, and risk measurement. *Insurance: Mathematics and Economics*, 75:126–136.

[22] Erway, C. C., Küpçü, A., Papamanthou, C., and Tamassia, R. (2015). Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):15.

[23] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009.

[24] Godard, S. (December 2015). *System Activity Report (sar)*. 11.2.0 edition.

[25] Google Inc. and Apache Software Foundation (October 2017). *Kubernetes Network Policies*.

[26] Jalalpour, E., Ghaznavi, M., Migault, D., Preda, S., Pourzandi, M., and Boutaba, R. (2018). A security orchestration system for cdn edge servers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 46–54. IEEE.

[27] Lakshman, A., Malik, P., and Apache Software Foundation (February 2017). *Apache Cassandra*. 3.10 edition.

[28] Lal, S., Kalliola, A., Oliver, I., Ahola, K., and Taleb, T. (2017a). Securing vnf communication in nfvi. In *Standards for Communications and Networking (CSCN), 2017 IEEE Conference on*, pages 187–192. IEEE.

[29] Lal, S., Taleb, T., and Dutta, A. (2017b). Nfv: Security threats and best practices. *IEEE Communications Magazine*, 55(8):211–217.

[30] Light, R. A. (2017). Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13).

[31] Mell, P. and Grance, T. (2009). Effectively and securely using the cloud computing paradigm. *NIST, Information Technology Laboratory*, 2(8):304–311.

[32] Mishra, A., Mathur, R., Jain, S., and Rathore, J. S. (2013). Cloud computing security. *International Journal on Recent and Innovation Trends in Computing and Communication*, 1(1):36–39.

[33] Mosa, A., El-Bakry, H. M., El-Razek, S. M. A., and Hasan, S. Q. (2016). A proposed e-government framework based on cloud service architecture. *International Journal of Electronics and Information Engineering*, 5(2):93–104.

[34] Pavlidis, A., Sotiropoulos, G., Giotis, K., Kalogeras, D., and Maglaris, V. (2018). Nfv-compliant traffic monitoring and anomaly detection based on dispersed vantage points in shared network infrastructures. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 197–201. IEEE.

[35] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM.

[36] RSA, E. (2017). Encryption of large files rsa.

[37] Santos, N., Gummadi, K., and Rodrigues, R. (2009). Towards trusted cloud computing. hot cloud 2009. In *Workshop on Hot Topics in Cloud Computing, ed USENIX Annual Technical Conference*.

[38] Shankland, S. (2009). Hp's hurd dings cloud computing. *IBM, CNET News. October*, 20.

[39] Sun, D., Chang, G., Sun, L., and Wang, X. (2011). Surveying and analyzing security, privacy and trust issues in cloud computing environments. *Procedia Engineering*, 15:2852–2856.

[40] Taber, N. and Aithal, A. (2017). *Task Networking for Amazon ECSs*.

[41] Taleb, T. (2014). Toward carrier cloud: Potential, challenges, and solutions. *IEEE Wireless Communications*, 21(3):80–91.

[42] Taleb, T., Corici, M., Parada, C., Jamakovic, A., Ruffino, S., Karagiannis, G., and Magedanz, T. (2015). Ease: Epc as a service to ease mobile core network deployment over cloud. *IEEE Network*, 29(2):78–88.

[43] Taleb, T., Ksentini, A., and Jantti, R. (2016). " anything as a service" for 5g mobile systems. *IEEE Network*, 30(6):84–91.

[44] Tanimoto, S., Hiramoto, M., Iwashita, M., Sato, H., and Kanai, A. (2011). Risk management on the security problem in cloud computing. In *Computers, Networks, Systems and Industrial Engineering (CNSI), 2011 First ACIS/JNU International Conference on*, pages 147–152. IEEE.

[45] Vikram, K., Prateek, A., and Livshits, B. (2009). Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186. ACM.

[46] Virtuozzo, IBM, Google, Biederman, E., Lezcano, D., Hallyn, S., and et.al., S. G. (January 2016). *Linux Containers*. Kernel v4.4.0-79-generic edition.

[47] Wei, J., Zhang, X., Ammons, G., Bala, V., and Ning, P. (2009). Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96. ACM.

[48] Wikipedia contributors (2004). Defense in depth — Wikipedia, the free encyclopedia. [Online; accessed 27-September-2018].

[49] Wikipedia contributors (2018). Software-defined networking — Wikipedia, the free encyclopedia. [Online; accessed 9-October-2018].

[50] Yang, M., Jiang, R., Gao, T., Xie, W., and Wang, J. (2018). Research on cloud computing security risk assessment based on information entropy and markov chain. *IJ Network Security*, 20(4):664–673.

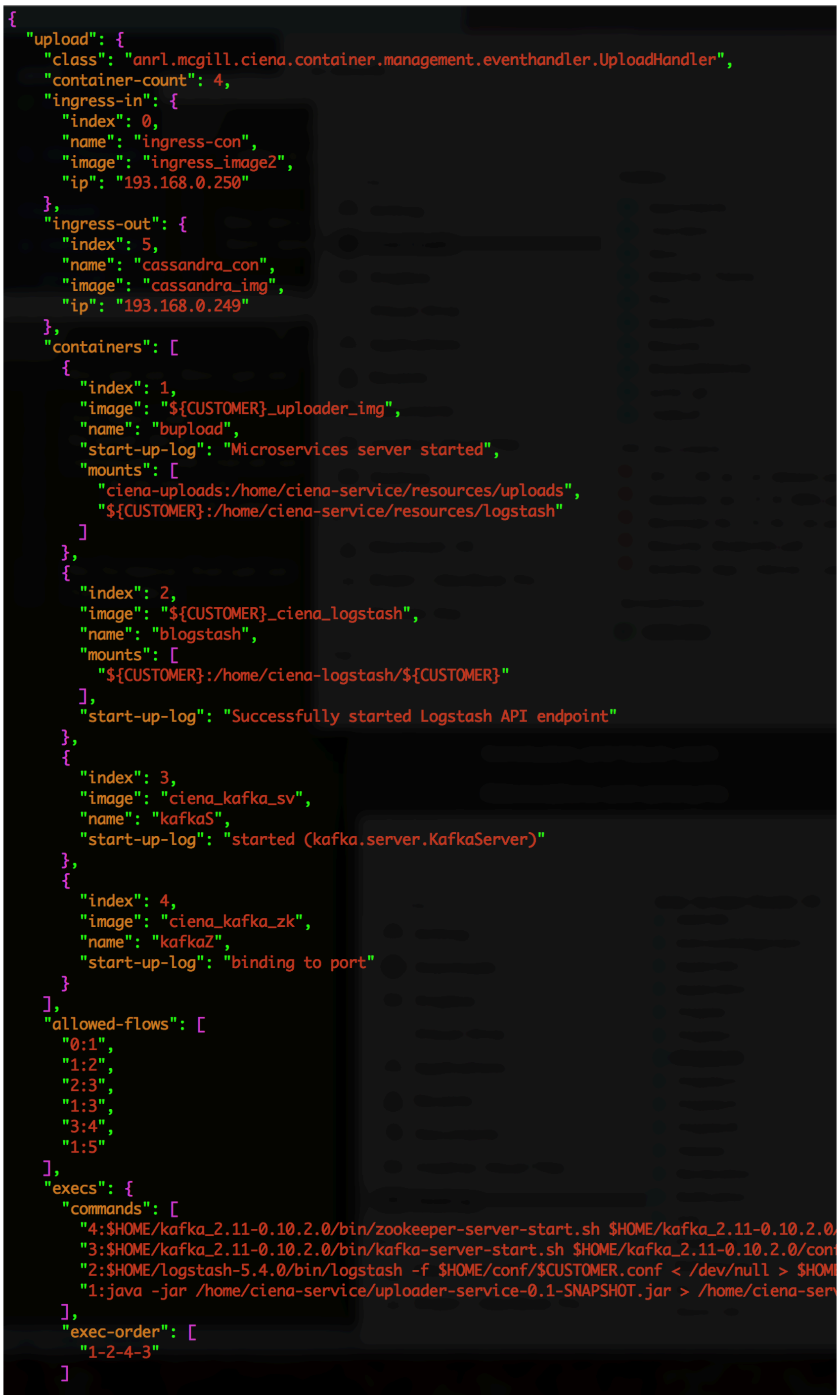
[51] Yang, W. and Fung, C. (2016). A survey on security in network functions virtualization. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 15–19. IEEE.

[52] Zaharia, M., UC Berkeley AMPLab, Databricks, and Apache Software Foundation (December 2016). *Apache Spark*. 2.1.0 edition.

Appendix A

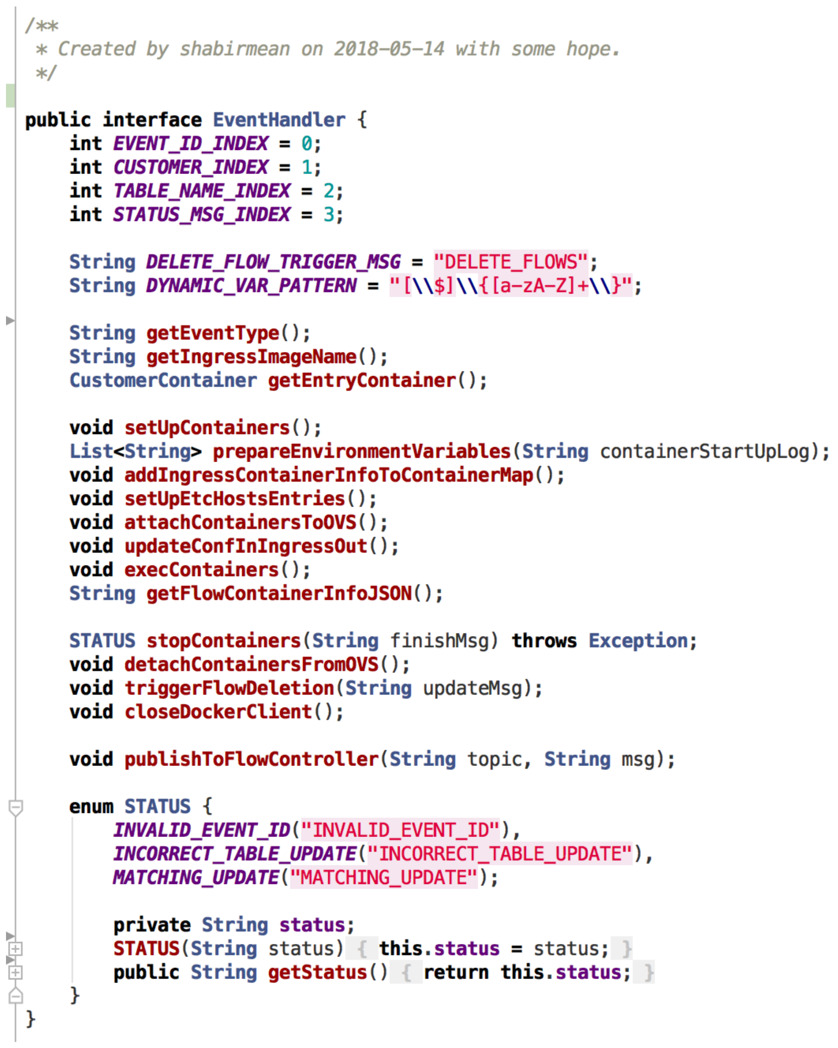
Load	Docker (%)	Host (%)	Framework (%)	Bare-metal (%)	Overhead(%)
10K	10.97	0.64	11.61	10.73	7.58
25K	12.01	0.59	12.60	11.72	6.98
50K	12.73	0.81	13.54	12.78	5.61
75K	14.41	0.63	15.04	13.33	11.37
100K	13.38	0.60	13.98	12.86	8.01
200K	13.61	0.60	14.21	13.4	5.70
500K	13.81	0.64	14.44	13.4	7.20

Table A.1 CPU Utilization for both setups



```
{
  "upload": {
    "class": "anrl.mcgill.ciena.container.management.eventhandler.UploadHandler",
    "container-count": 4,
    "ingress-in": {
      "index": 0,
      "name": "ingress-con",
      "image": "ingress_image2",
      "ip": "193.168.0.250"
    },
    "ingress-out": {
      "index": 5,
      "name": "cassandra_con",
      "image": "cassandra_img",
      "ip": "193.168.0.249"
    },
    "containers": [
      {
        "index": 1,
        "image": "${CUSTOMER}_uploader_img",
        "name": "bupload",
        "start-up-log": "Microservices server started",
        "mounts": [
          "ciena-uploads:/home/ciena-service/resources/uploads",
          "${CUSTOMER}:/home/ciena-service/resources/logstash"
        ]
      },
      {
        "index": 2,
        "image": "${CUSTOMER}_ciena_logstash",
        "name": "blogstash",
        "mounts": [
          "${CUSTOMER}:/home/ciena-logstash/${CUSTOMER}"
        ],
        "start-up-log": "Successfully started Logstash API endpoint"
      },
      {
        "index": 3,
        "image": "ciena_kafka_sv",
        "name": "kafkaS",
        "start-up-log": "started (kafka.server.KafkaServer)"
      },
      {
        "index": 4,
        "image": "ciena_kafka_zk",
        "name": "kafkaZ",
        "start-up-log": "binding to port"
      }
    ],
    "allowed-flows": [
      "0:1",
      "1:2",
      "2:3",
      "1:3",
      "3:4",
      "1:5"
    ],
    "execs": {
      "commands": [
        "4:$HOME/kafka_2.11-0.10.2.0/bin/zookeeper-server-start.sh $HOME/kafka_2.11-0.10.2.0/
        "3:$HOME/kafka_2.11-0.10.2.0/bin/kafka-server-start.sh $HOME/kafka_2.11-0.10.2.0/con
        "2:$HOME/logstash-5.4.0/bin/logstash -f $HOME/conf/$CUSTOMER.conf < /dev/null > $HOM
        "1:java -jar /home/ciena-service/uploader-service-0.1-SNAPSHOT.jar > /home/ciena-ser
      ],
      "exec-order": [
        "1-2-4-3"
      ]
```

Fig. A.1 A sample *contianer-manager* configuration file for data-upload flow

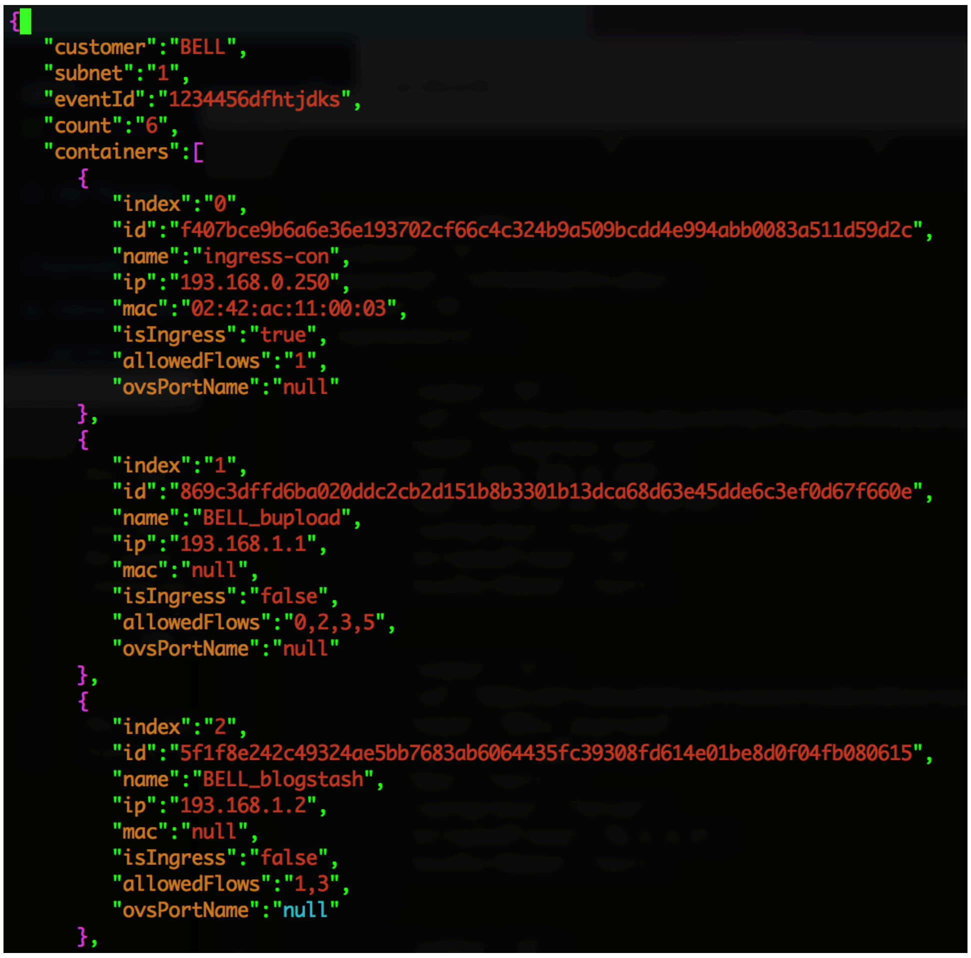


```java
/**
 * Created by shabirmean on 2018-05-14 with some hope.
 */

public interface EventHandler {
    int EVENT_ID_INDEX = 0;
    int CUSTOMER_INDEX = 1;
    int TABLE_NAME_INDEX = 2;
    int STATUS_MSG_INDEX = 3;

    String DELETE_FLOW_TRIGGER_MSG = "DELETE_FLOWS";
    String DYNAMIC_VAR_PATTERN = "[\\$]\\{[a-zA-Z]+\\}";

    String getEventType();
    String getIngressImageName();
    CustomerContainer getEntryContainer();

    void setUpContainers();
    List<String> prepareEnvironmentVariables(String containerStartUpLog);
    void addIngressContainerInfoToContainerMap();
    void setUpEtcHostsEntries();
    void attachContainersToOVS();
    void updateConfInIngressOut();
    void execContainers();
    String getFlowContainerInfoJSON();

    STATUS stopContainers(String finishMsg) throws Exception;
    void detachContainersFromOVS();
    void triggerFlowDeletion(String updateMsg);
    void closeDockerClient();

    void publishToFlowController(String topic, String msg);

    enum STATUS {
        INVALID_EVENT_ID("INVALID_EVENT_ID"),
        INCORRECT_TABLE_UPDATE("INCORRECT_TABLE_UPDATE"),
        MATCHING_UPDATE("MATCHING_UPDATE");

        private String status;
        STATUS(String status) { this.status = status; }
        public String getStatus() { return this.status; }
    }
}
```

Fig. A.2 The *EventHandler* interface based on which the work-flow handlers are written

Load	Docker (%)	Host (%)	Framework (%)	Bare-metal (%)	Overhead(%)
10K	9.05	2.29	11.33	10.45	7.77
25K	9.25	2.31	11.55	10.55	8.66
50K	9.36	2.33	11.69	10.67	8.73
75K	9.24	2.27	11.51	10.7	7.04
100K	9.28	2.27	11.55	10.63	7.97
200K	9.39	2.32	11.71	10.71	8.54
500K	9.43	2.40	11.83	10.76	x9.04

Table A.2 Memory Utilization for both setups



```
{
   "customer":"BELL",
   "subnet":"1",
   "eventId":"1234456dfhtjdks",
   "count":"6",
   "containers":[
      {
         "index":"0",
         "id":"f407bce9b6a6e36e193702cf66c4c324b9a509bcdd4e994abb0083a511d59d2c",
         "name":"ingress-con",
         "ip":"193.168.0.250",
         "mac":"02:42:ac:11:00:03",
         "isIngress":"true",
         "allowedFlows":"1",
         "ovsPortName":"null"
      },
      {
         "index":"1",
         "id":"869c3dffd6ba020ddc2cb2d151b8b3301b13dca68d63e45dde6c3ef0d67f660e",
         "name":"BELL_bupload",
         "ip":"193.168.1.1",
         "mac":"null",
         "isIngress":"false",
         "allowedFlows":"0,2,3,5",
         "ovsPortName":"null"
      },
      {
         "index":"2",
         "id":"5f1f8e242c49324ae5bb7683ab6064435fc39308fd614e01be8d0f04fb080615",
         "name":"BELL_blogstash",
         "ip":"193.168.1.2",
         "mac":"null",
         "isIngress":"false",
         "allowedFlows":"1,3",
         "ovsPortName":"null"
      },
```

Fig. A.3 A sample of the message sent to the *flow controller* from the *container manager*