A GIPSY Runtime System with a Kubernetes Underlay for the OpenTDIP Forensic Computing Backend

Seyed Pouria Zahraei

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

October 2022

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Seyed Pouria Zahraei**

Entitled: **A GIPSY Runtime System with a Kubernetes Underlay for the OpenTDIP Forensic Computing Backend**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair, Examiner
Dr. Todd Eavis

_____ Examiner
Dr. Weiyi Shang

_____ Supervisor
Dr. Joey Paquet

_____ Supervisor
Dr. Serguei A. Mokhov

Approved by _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

A GIPSY Runtime System with a Kubernetes Underlay for the OpenTDIP Forensic Computing Backend

Seyed Pouria Zahraei

In this research work, we propose an underlay based on Kubernetes to enhance the scalable fault tolerance of the General Intensional Programming System's distributed run-time demand-driven backend to gather digital evidence from GitHub repositories and encode them in Forensic Lucid for further analysis in the integrated OpenTDIP environment.

We developed a solution so that forensic investigators could use GitHub to gather a dataset to investigate program flaws and vulnerabilities related to security from GitHub projects written in different programming languages. For this purpose, we design and implement a JSON demand-driven encoder to perform a Forensic Lucid conversion pipeline (data extraction, format conversion, and file compilation). In order to distribute the execution, we utilized the GIPSY distributed computing system.

We also integrated Kubernetes with GIPSY distributed computing system in order to improve the configuring, starting up and registering GIPSY nodes, so that GIPSY nodes could get registered automatically without any manual configuration. In addition, provide a mechanism to have a scalable fault-tolerant system so that when a GIPSY node dies, it will handle reallocation, configuration and registration of the GIPSY nodes automatically.

Acknowledgments

I would like to express my profound gratitude to my supervisors Dr. Joey Paquet and Dr. Serguei Mokhov for their unconditional support and patience and care during my graduate study.

I am highly grateful to have had the opportunity to work with outstanding GIPSY team members, Peyman Derafsh Kavian, Pouria Roostaei, Vashisht Marhwal.

My warmest thanks to my parents for their unrelenting encouragement that made me constantly move forward during difficulties, and to my dear sister, Seyedeh Sara Zahraei, for her constant support and unconditional love.

My thesis would not have been possible without the wholehearted support of my all great friends, especially Sepehr Jalayer, Sandy Al Akhras, Mahan Ashouri, Azar Mahmoudi, Mehrsa Armand and Happy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The FORENSIC LUCID forensic knowledge representation and reasoning language [8] is designed to automatically compute the reasoning a forensic proof with digital evidence and for event reconstruction. It can be used, for example, to build a formal case description proving or disproving a hypothesis that, in the history of GitHub commits of a certain piece of software, a specific known security defect was introduced that resulted in a compromise. Even though FORENSIC LUCID can declare this formal concept, it requires that the evidence be encoded in a manner that a FORENSIC LUCID program can understand. Thus, we require automatic extraction features from a GitHub repository and their translation (encoding, conversion) into FORENSIC LUCID format.

In this research work, we aim to show how FORENSIC LUCID can be used to distribute the computation of a forensic investigation involving evidential data extraction from a GitHub repository and translating them into FORENSIC LUCID evidential statements. As the evidential data can be quite large, performing such a task requires extensive computation. Therefore, in order to extract and translate the evidential data automatically, we use the GIPSY [9] system as a demand-driven distributed computing platform to proceed with the extraction and transformation of the evidential data and, at the same time, use it in order to evaluate the FORENSIC LUCID programs (compile) in the context of the overall Open Trusted Digital Investigation Platform (OpenTDIP) (see Figure 1), which covers all aspects

of digital forensic computing, from evidence management, a chain of custody using a blockchain, formal methods, and event reconstruction.

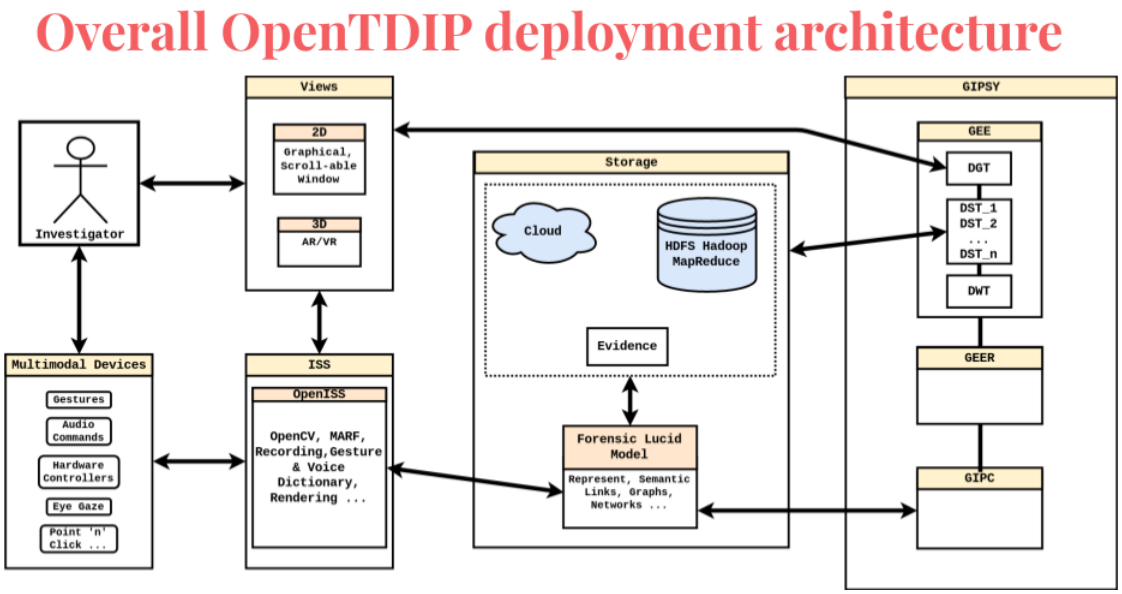

Figure 1: Overall OpenTDIP Deployment Architecture [1]

In Figure 1 the high-level architecture depicts components related to the human factors and human-computer interaction on the left-hand side using applications such as DigiEVISS [1, 10] or Ftkplipse [11, 12]. On the right-hand-side is the computation engine of the General Intensional Programming System (GIPSY) to compile and evaluate Forensic Lucid expression using its distributed runtime of the General Eduction Engine (GEE, covered later). The two sides are joined by the distributed scalable storage for the digital evidence and demand-driven computation.

However, deploying the existing GIPSY distributed system is relatively complex at scale, considering manual configuration and starting up the GIPSY system components, the compilation time for the GIPSY program that involves dozens of dependencies. Moreover, if a part of the distributed system crashes, it requires restarting the node, which currently should be done manually, which takes considerable time. This leads us to demonstrate how Forensic Lucid's execution engine can integrate with the use of the industry-standard Kubernetes [6] distributed

processing orchestration features such as the automatic and efficient deployment of the GIPSY distributed system and a scalable fault-tolerance mechanism.

1.1 Research Domain

In order to describe the domain of this research work, first, we discuss the two main concepts involved in our research – Forensic Computing and Cluster Computing.

1.1.1 Forensic Computing

By definition in the dictionary, **forensics** refers to the action of scientists who evaluate evidence to assist law enforcement in solving crimes. **Digital Forensics** is defined as a procedure of digital criminal investigations, including digital data stored on various computer devices as evidence for examination, which can be used by the court of law [4].

Nowadays, computers are used in forensic computing to analyze evidence that can be used in court to find an answer to legal questions. In order to analyze the evidence on digital devices, we need a computer and specific software, where evidence can exist in any type of storage or electronic device. In order to analyze evidence adequately, sometimes we need to collect a large amount of data and filter this data using some explicit keywords and analyze the data to deliver a piece of adequate evidence, which can help us discover more detailed information about an illegal act. The preservation, identification, extraction, documentation, and interpretation of computer data are all part of computer forensics [13–15]. There are different kinds of methodologies in forensic computing. The basic methodology consists of gathering evidence from any sort of resources, authenticating the evidence and analyzing the evidence by reconstructing a sequence of events that eventually proves or disproves a hypothesis [4, 15]. Digital Forensic investigation is required when data is too large or complex to gather manually and when the proof is too complex to manage manually.

1.1.2 Cluster Computing

Nowadays, the volume of the data to be analyzed is drastically increasing. Thus, the popularity of cluster computing is expanding with the growing demand for more powerful applications. With cluster computing, we are able to solve the load issue by combining various processors. Cluster computing is a combination of parallel, high-performance, distributed, and high-availability computing, which consists of a variety of multiple standalone computers (nodes) that are connected together and perform as a single computational unit [2, 16, 17].

As we can observe in Figure 2, a typical cluster consists of multiple standalone computers, such as PCs, powerful workstations, or SMPs (Symmetric multiprocessing or shared-memory multiprocessing) which are connected together through a high-speed network/switch [2]. There are other components, which play an essential role in the cluster architecture, such as Network Interface Hardware, which is responsible for the communication and transferring of the data packages between the nodes through a high performance network/switch, cluster middleware (Single System Image (SSI) and System Availability Infrastructure), which is software on the top of nodes and allows users to manage the cluster as a single unit, parallel programming environments, and sequential, parallel or distributed applications [17]. This architecture can help us solve problems requiring high speed, such as when we need to perform analysis and interpretation on a large amount of data. Cluster computing provides us with a high-performance, highly available and scalable computational platform.

The domain of our research is thus to employ cluster computing tools to enhance the evidence gathering and preparation for analysis in forensic investigations.

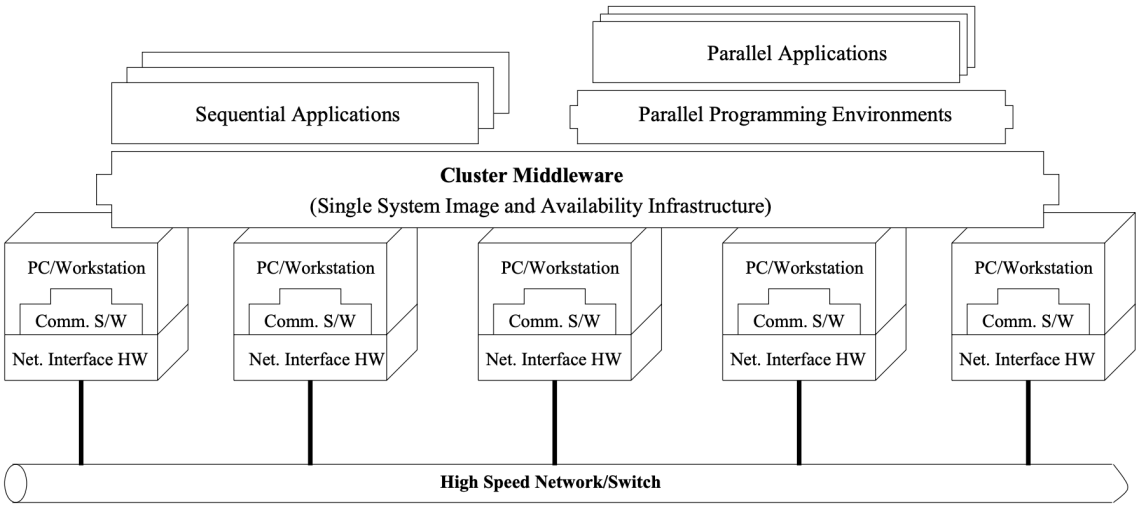

Figure 2: Cluster Computing Architecture [2]

1.2 Motivation

1.2.1 Essential Background

In this section, we discuss the motivation of the research. However, before we dive into the subject, we provide a brief background. Nonetheless, in Chapter 2, we will discuss it more in detail.

We start by giving a brief background about FORENSIC LUCID. FORENSIC LUCID [8, 18–21] is a forensic case specification language for automatic evidence composition used to formally represent and reason about digital crime incidents. FORENSIC LUCID is designed to describe forensic evidence in digital crime incidents as a context of intensional evaluations [4]. Its evaluation engine is based on the outcome of backtracing and, if a backtrace is found, to offer the path for the related event reconstructions. The results of the evaluation of FORENSIC LUCID expressions are true or false, i.e., "guilty" or "not guilty", which can be done by one or multiple backtraces [4]. FORENSIC LUCID is based on LUCID, which is an intensional and multidimensional dataflow programming language [22–26]. For instance, in MAC spoofer, forensic evidence is gathered from the log files containing the network activity for the MAC spoofer to encode in FORENSIC LUCIDformat and later reasoning [27].

The **General Intensional Programming System (GIPSY)** is a multi-language programming compiler and execution engine framework for compiling all types of Lucid dialects, including Forensic Lucid [28,29]. Using GIPSY, Forensic Lucid programs can be compiled and executed as a distributed system using a demand-driven dataflow model. In addition, GIPSY has MARFCATDGT and MARFCATDWT [30], that can feed the machine learning with weak or vulnerable code data in order to provide additional evidence to an investigation.

GIPSY consists of the General Intensional Programming Language Compiler (GIPC), General Eduction Engine (GEE), Intensional Run-time Programming Environment (RIPE), which are the primary components for compiling and executing Lucid programs, such as Forensic Lucid programs. Together they are employed to validate if the encoded Forensic Lucid program corresponds to the observation sequences via the semantic connection between distinct evidential statement objects [4].

Since the GIPSY can compile the Forensic Lucid code, the GIPSY runtime system is also used to distribute the computation of the investigation at the same time. Having such a system capable of compiling and executing the Forensic Lucid program can be advantageous since, without using the GIPSY runtime system, we need to provide a proper compiler.

As we can observe in Figure 3, the architecture of GIPSY program execution consists of three runtime components: the Demand Generator Tier (DGT), Demand Worker Tier (DWT) and Demand Store Tier (DST). A GIPSY node is a registered computer hosting one or more GIPSY tiers, where the registration of GIPSY nodes is triggered by GIPSY Manager Tier (GMT) instance, and each of these tiers works as an independent computational unit performing specific duties. Each instance manually uses a corresponding configuration file to do the registration and future configurations, which will be discussed more in detail in the next Chapter [3,4,9,29,31]. The execution architecture of GIPSY is demand-driven and the tiers generate the demands and migrate to other tiers using the Demand Store Tier (DST) [32].

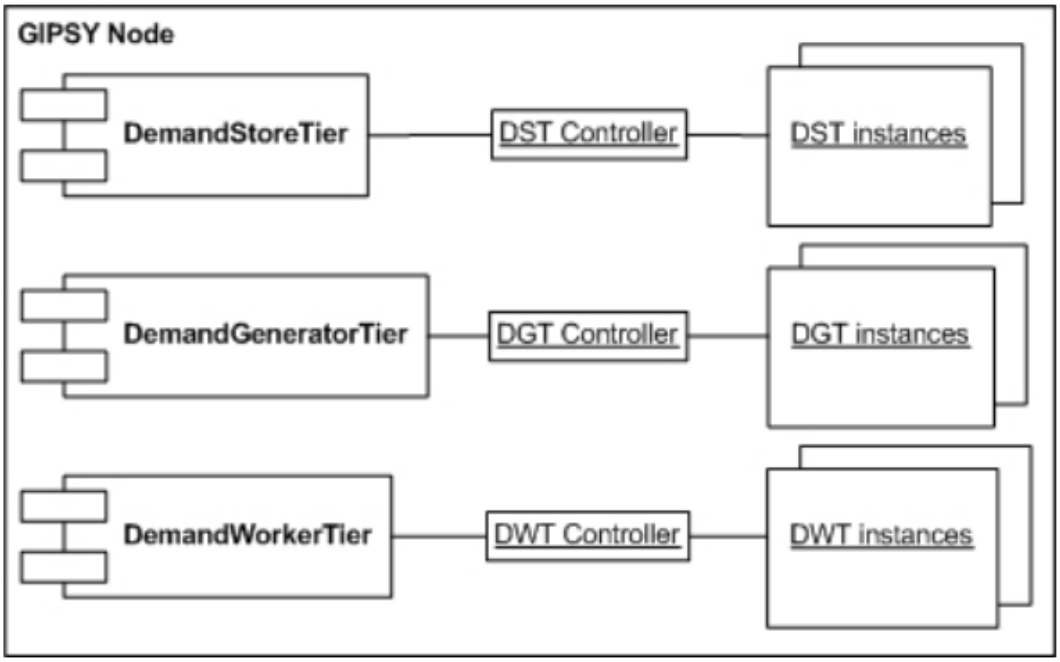

Figure 3: General Intensional Programming System Node [3]

1.2.2 Motivational Scenarios

Below we describe possible scenarios that can help us describe the solution that we will propose in the research work and illustrate some of the requirements that need to be met.

1.2.2.1 Investigating Bugs or Vulnerabilities related to breaches

In this scenario a forensic investigator wishes to encode evidential statements in the form of a Forensic Lucid program by constructing observation sequences via the semantic connection between distinct evidential statements found on a organization [33].

1. The forensic investigator primarily gathers vulnerable applications installed as a possible vector of entry correlated with the exact published Common Vulnerabilities and Exposures (CVE) and confirm with the MARFCAT-based, [30] analysis.

2. The forensic investigator gather data from source code of announced vulnerable applications as a witness statements, which are later encoded in the form of a FORENSIC LUCID observation sequence [4].

3. A set of testimonies that have been collected are encoded and defined as an evidential statement (*es*) and during the verification process, *es* are developed into the local knowledge base for the forensic case [4].

4. For evidential statements, theories in FORENSIC LUCID are defined and encoded to describe the evidence more in detail, and the theories are encoded as observation sequences *os*. They are added to *es* after being assessed against the forensic evidence acquired in *es*, and using an evaluating system we can automatically verify that theories agree with the evidence, which means the theory has an explanation for the evidential context and the theory probably explains the chain of events that led to the incident [4].

The process that we described above will be accomplished by employing the GIPSY system as we explained in Section 1.2.1 in the GIPSY node architecture. In the next chapter, we will discuss how it can be done more in detail.

1.2.2.2 Management of GIPSY Instances

In this section, we would like to discuss the scalable fault tolerance and deployment of the GIPSY computational network.

1. As discussed in Section 1.2.1, deploying GIPSY program and adding a new GIPSY node to the cluster requires quite a time-consuming procedure. Setting up a GIPSY network or adding a GIPSY node to the network requires providing the needed configuration files to each physical machine so that GIPSY node can use it to connect to the network, cloning the project from the repository, compiling the whole project, run the GIPSY node program and enter the proper command in order to join the GIPSY node to the network. However, we are looking for a mechanism to facilitate the steps that we mentioned above and

administrate all GIPSY network components on a master computer, regardless of the GIPSY node's physical location.

2. Running a GIPSY program requires some packages and tools, such as the JVM, xterm, as well as dozens of packages and libraries. We need an approach to automate the deployment of such dependencies for each machine to deploy the GIPSY program independently of operating systems and hardware platforms.

3. If a GIPSY node dies in the network, we need manually take the procedure as we explained above to join a new GIPSY node to the network. Therefore, we need a mechanism to automatically handle the fault tolerance for the GIPSY nodes in a way to avoid to have to manually restart a faild node.

The effort required to re-invent now existing open solutions there is hard to justify. Therefore, in order to achieve all the features mentioned above, we propose to employ container orchestration embedded into the existing GIPSY runtime architecture.

1.2.3 Requirements

Following the scenarios described in Section 1.2.2.1 and Section 1.2.2.2 we provide the list of requirements, which will allow us to define all the required parts of our solution. Here we provide a set of functional and non-functional requirements, in which functional requirements describe the behaviour and components of the system and non-functional requirements describe quality attributes and user expectations.

1. **The system shall have JSON demand-driven encoder:**

   We shall design and implement a demand-driven JSON encoder in order to extract data from GitHub repositories, convert them to Forensic Lucid format and compile Forensic Lucid files [7].

2. **The system shall employ the GIPSY distributed computing system for forensic computing pipeline:**

In order to execute the JSON demand-driven encoder, we shall utilize the GIPSY distributed computing system to spread the computation of the Forensic computing pipeline.

3. **The system shall return the results quickly if we execute the JSON encoder for the second attempt on the same data:**

   There shall be a mechanism where if we execute the JSON encoder multiple times for an identical set of repositories data, we shall receive the same results relatively fast without another extraction, conversion or compilation after the first execution.

4. **The system shall have an efficiently deployable GIPSY distributed system:**

   We shall have a mechanism where the GIPSY node can start on any type of operating system without installing all the dependencies, and it can start without requiring repetitive startup procedures for every machine as we mentioned in Section 1.2.2.2.

5. **The system shall have a scalable fault tolerance mechanism:**

   If a physical network node which contains one or multiple GIPSY nodes dies, it shall be able to detect and reallocate the GIPSY nodes to the next available physical network node automatically. This way, the systems administrator does not need to repeat the starting up of a GIPSY node procedure on another machine.

6. **The system shall have a shared file system:**

   In order to avoid moving some required configuration files and metadata between GIPSY nodes, we shall have a file system that is shared and simultaneously mounted on multiple GIPSY nodes.

1.3 Thesis Objectives

In this section, we discuss the goals and objectives of our research work. As we provided some scenarios in Section 1.2.2.1 and Section 1.2.2.2 and discussed the requirements in Section 1.2.3, we would like to provide an adjustable robust and scalable solution to oversee the forensic investigation using the GIPSY system and manage the processes utilizing container orchestration. Bellow is a brief list of objectives of this research work:

- To facilitate forensic investigator's task to efficiently exploit the collection/encoding of digital evidence, e.g., GitHub repository, for forensic investigation by employing the FORENSIC LUCID programming language.
- Design and implement node instances by utilizing container orchestration in order to have a quickly deployable, scalable fault tolerance system.

Finally, we need to evaluate and analyze to what extent we were able to satisfy what we have mentioned in the requirements section in Section 1.2.3 and make sure that we were able to meet what we have mentioned in the scenario section in Section 1.2.2.1 and Section 1.2.2.2.

1.4 Scope of the Thesis

In this research work, our goal is to achieve all the requirements that have been mentioned in Section 1.2.3. We attempted to show how the Forensic Lucid dataset can be collected for a follow-up computation of Forensic Investigation with evidence from GitHub repositories. We also demonstrate how the execution engine can integrate the use of the Kubernetes distributed processing orchestration features such as automatically scalable fault tolerance and easily manageable. Below is the list of some problems or elements related that are outside the scope of our research and development work:

- Primarily we performed the data extraction on the GitHub repositories by retrieving the dataset from the GitHub's application programming interface (API). However, we could expand our analysis datasets to other sources. There are other applications that we could conduct a forensic investigation on their dataset (e.g., Twitter, etc).

- As we discussed in Section 1.2.1, in this research work, we came up with a solution to employ a container orchestration tool in order to scale and manage the deployment of the GIPSY distributed network. There are various container orchestration tools, such as Kubernetes, OpenShift, Docker Swarm, etc. Yet, in this research work, we only utilize the Kubernetes tool.

- If a GIPSY container dies, we can handle the fault tolerance automatically. However, we cannot control the failure of a GIPSY tier in the container. In our architecture, Kubernetes will not be aware of the failure of the GIPSY tier inside the container, and it only controls the health of GIPSY container (pod [6]) in the cluster.

1.5 Thesis Contributions

In this section, we discuss the contributions that have been made through the research work. We fulfill what we have discussed in the Section 1.2.3 and Section 1.3 as well as some associated support tasks, which we discuss in the list below:

- We complete the design and implementation of a JSON demand-driven encoder to run a Forensic Lucid conversion pipeline (data extraction, converting to Forensic Lucid format, compiling the Forensic Lucid files).

- Using GIPSY's GEE runtime system, we provide an efficient solution that inherently persistently stores the resulting values of the computed demands. This effectively results in a distributed system that persistently stores the results computed by each node/tier. In this way, if a node/tier fails, it can be restarted later with minimal loss of the information that it had processed earlier.

- We provide a GIPSY container in order to automatically manage all the dependencies when deploying a GIPSY network.
- We establish and configure one of the well-known container orchestration tools (Kubernetes) in order to manage tiers and nodes for the GIPSY system.
- We design and implement the GIPSY tiers and nodes so that they can deal with fault-tolerance in a more scalable way, i.e., that each time a fault happens, the time taken for recovering from the fault is minimized, leading to scalable fault tolerance. Along the same line, the components are much more easily deployable so that adding a new one in the cluster would be only by changing the configuration file in the Kubernetes cluster, thus reducing their deployment time.
- We deploy the Network File System (NFS) server to share the file system between different GIPSY containers by mounting the NFS volume to each GIPSY container.
- We evaluated our cluster's fault-tolerant, and efficiency after employing container orchestration.

1.6 Summary

In this chapter, we began by discussing the domain of our research work in Section 1.1. Thereafter we provided our motivation in Section 1.2, in which we discussed some motivation scenarios for this work. Afterwards, we provided some functional and non-functional requirements extracted from the motivation section in Section 1.2.3. Then we described the thesis objectives in Section 1.3. Later, we specified the scope of our research work in Section 1.4. Finally, we provided the contributions of our research work in Section 1.5.

Chapter 2

Background

2.1 Forensic Lucid

Following what we described as FORENSIC LUCID in Section 1.2.1, in this section, we discuss FORENSIC LUCID more in detail. We explain what the reader needs to comprehend from the perspective of this research work about FORENSIC LUCID. So far, the reader is aware of some essential aspects of FORENSIC LUCID, such as FORENSIC LUCID is a Lucid-based program used to represent a data-flow [4] program. It is used for cases when forensic investigators attempt to reason encoded evidential statements, as described in Section 1.2.1. For instance, in the scenario in Section 1.2.2.1, forensic investigators attempt to gather a dataset from the GitHub repository by fetching, converting to Forensic Lucid and compiling the Forensic Lucid files so that later could perform the investigation regarding any security vulnerabilities in a project.

Before we go more deeply into the design of the FORENSIC LUCID, we would like to provide two basic definitions that are used in this context. At first, we start by defining the **observation sequence**, which is a list of observations arranged chronologically. These observation sequences represent a continuous witnessed story [4].

$$os = (observation1, observation2, observation3, ...) \tag{1}$$

Second, an **evidential statement** is a set of observation sequences, which is not necessarily arranged chronologically [4].

$$es = (os1, os2, os3, ...) \tag{2}$$

Below we provide a required specification template of a FORENSIC LUCID program, which after identifying forensically interesting data, for instance, as we discussed in Section 1.2.2.1, will be able semantically and syntactically check by the GIPC that we described in Section 1.2.1 [4].

```
where
  evidential statement system_es = { ... };
  // T is a theory of what has transpired
  observation sequence T = { ... };
end
```

Listing 2.1: Simple FORENSIC LUCID

An evidential statement *es* is a dictionary of the evidential observation sequences and is the highest level dimension in Lucid parlance. The *es* is in the form of $<dimension: tag>$ pairs, where dimensions are identified by their identifiers, and their tags are defined as nested values [4].

2.2 GIPSY

As we discussed in Section 1.2.1, GIPSY is a multi-language programming platform designed to compile and execute all dialects of the Lucid family of programming languages.

The core components of the GIPSY's design (GIPC, GEE and RIPE) are responsible for supporting multiple compilers and, as a result, get a binary output, which is a compiled Lucid program and can then in turn be executed by the GISPY execution engine [4].

The GIPC is a compiler framework that enables syntax and semantic analysis,

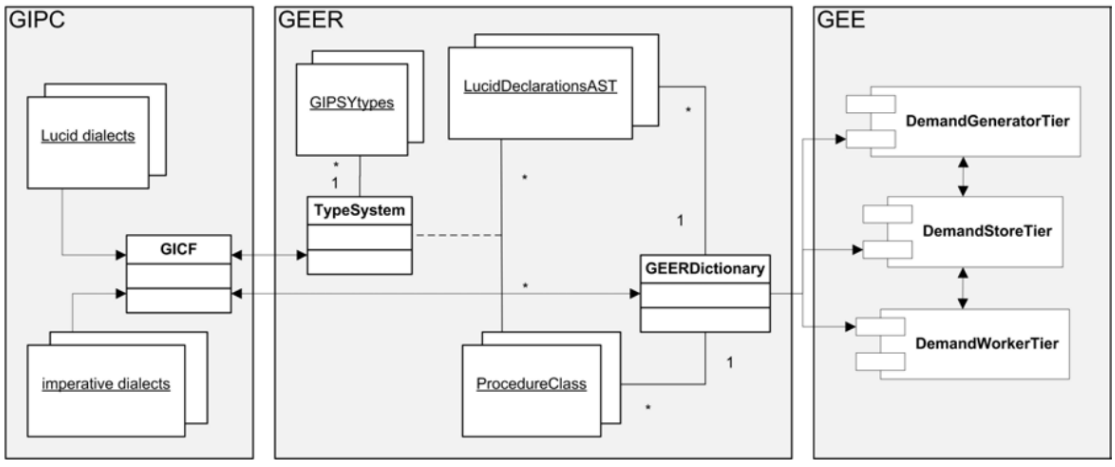

Figure 4: High-level Structure of GIPSY's GEER Flow Overview [4]

and translation of any Lucid variant. It is built on the idea of the Generic Intensional Programming Language (GIPL), which serves as the fundamental run-time language into which all other varieties of the Lucid language may be transformed and then executed. The Generic Eduction Engine Resources (GEER) is a dictionary of run-time resources compiled from a GIPL program that had previously been generated from the original program using semantic translation rules defining how the original Lucid program can be translated into the GIPL. Under the framework of GIPSY there are a number of compilers, and the corresponding run-time environment is present under the eduction execution engine (GEE) [4, 7].

GEE is the component where the Demand Migration System (DMS) and multitier architecture as a whole are dependent on the demand-driven tagged token dataflow distributed computation model [4, 34, 35]. Every tier in the architecture of this distributed system can have any number of instances where demands are distributed without knowledge of the processing or storage locations of the demands. Any tier or node failure can happen without having a fatal effect on the system while computation is taking place, nodes and tiers can be added or removed without any noticeable lag, nodes and tiers can impact how each GIPSY program is performed at runtime, meaning a specific node or tier may have different programs requiring its computational resources [4].

The Demand Migration System (DMS) can be realized as an instance of the Demand Migration Framework (DMF) [4, 35–38]. One such particular DMS that we used for the purpose of this particular research uses Jini (Apache River) to transport and store their results' demands, and a JavaSpaces repository serves as a temporary data storage for the most commonly demanded computations and their outputs. The DMF is an architecture that is based on the demand store and consists of transport agents (TAs) that implement a specific protocol to store and deliver demands across different nodes and tiers. The evaluation process starts with a demand generator that creates demands according to the problem specification (generally as represented by a Lucid program, e.g., FORENSIC LUCID), which are sent to the demand store. Form there, any other generator or worker connected to this store can pick up these demands and continue the processing further. Eventually, some of the demands will be calls to procedures (i.e., *procedural demands*), which is a specific kind of demand that can be processed by workers to perform processing on a higher granularity level by relying on a procedural programming language for their execution, e.g., Java, C++, etc. Once demands have been picked up and processed, i.e., their corresponding values have been calculated, they are put back in the store with their value embedded in the demand and are tagged as processed demands, at which point their original generator will be able to fetch them and continue its processing, until all demands have been finally processed.

Figure 5 represents a use case diagram where nodes and tiers start in a GIPSY distributing system.

In this diagram, General Manager Tier (GMT) allows the GIPSY nodes and tiers to register and allocate them to the GIPSY network instances under its management. In order to decide if additional tiers nodes need to be produced, the GMT communicates with the allocated tiers and then, when necessary, GMT sends GIPSY nodes system commands to generate new tier instances. Users may use any GMT to register a node, which alerts all the other GMTs to its existence and makes the node accessible to host new GIPSY tiers upon request from any GMT [4].

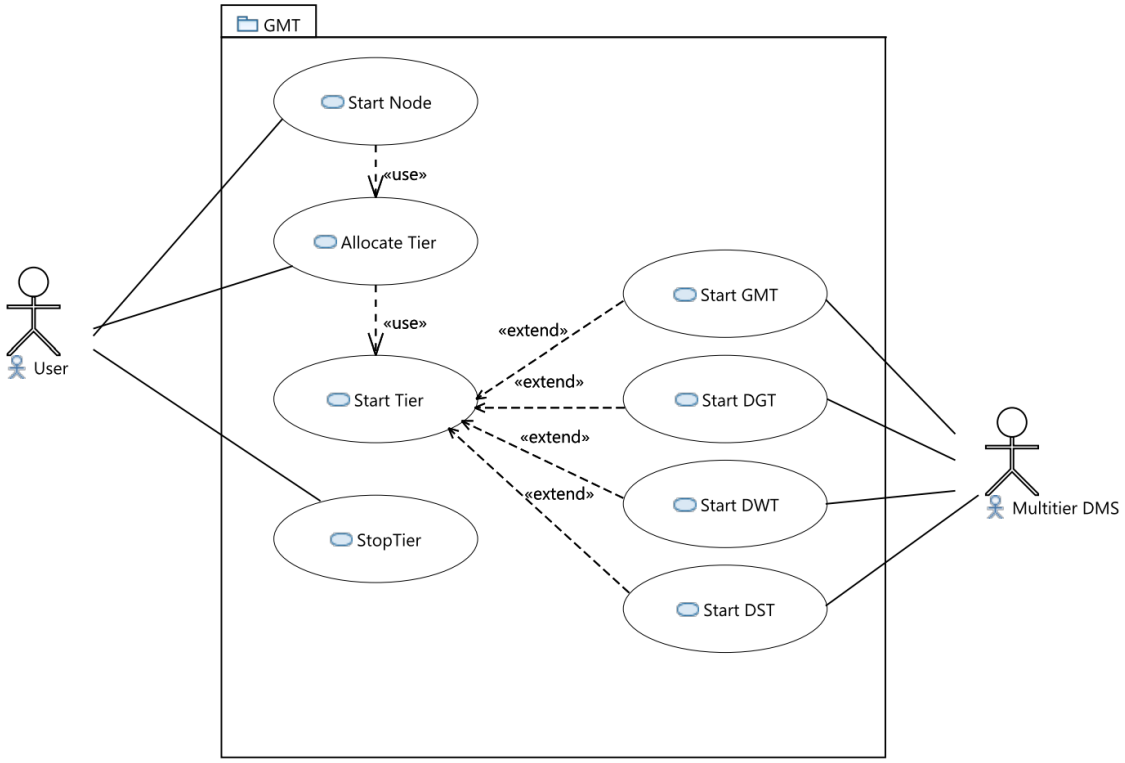

Figure 5: GMT Use Case Diagram [4]

The GIPSY program execution is divided into three jobs and delegated to different tiers in a multitier architecture where each tier is an abstract, generalized object representing a computing unit that interacts with other tiers utilizing demands to work together to execute a program as a whole. Therefore, the GIPSY multi-tier architecture is entirely demand-driven [3, 4, 31].

A GIPSY node is a physical or virtual computer registered via GMT instance and is ready to host one or more GIPSY tiers and is being controlled by GMT remotely [4].

In accordance with the program declarations and definitions stored in one of the GIPSY tiers, the Demand Generator Tier (DGT) creates demands and can be transferred to other Demand Generator tiers or Demand Worker tier instances to be processed further and for any Lucid program it can handle requests for, DGT hosts a set of tiers. The Demand Generator tier can make system demands asking for more tiers to be added to its GEER Pool thanks to a demand-driven method and allow DST instances to process requests for more programs running on the GIPSY networks

they are a part of [4].

The Demand Worker Tier (DWT) processes demands written in a procedural language or functions. Same as DGT the Demand Worker tier can make system demands through a demand-driven mechanism to add more GEERs to its GEER pool, gradually increasing its processing knowledge capability [4].

The Demand Store Tier (DST) serves as a tier middleware to move demands between tiers. It also offers persistent storage for demands and the values generated by those demands, improving processing performance by preventing the need to compute each demand's value each time it is regenerated after being processed. The Demand Store tier is made to include a peer-to-peer architecture when necessary and a way to join all of the Demand Store tier instances in a specific GIPSY network instance in order to prevent experiencing an execution slowdown in large computations. Therefore demands or the results of the demands will be stored on any available DST instance [3, 4, 28, 31, 39, 40].

2.3 Container Orchestration

Before diving into the definition of Container Orchestration, we need to describe some main terms related to Container Orchestration.

- **Container**: A container is a standardized software component that encapsulates code and all of its dependencies to ensure that an application will run in different computer environments [41, 42].
- **Container image**: A container image is a standalone software package that contains all the components and instructions required to operate a program [41, 42].

One of the most well-known container platforms is **Docker**. Docker provides the ability to package and run applications and ship them as a container image. Figure 6 illustrates the architecture of Docker, which is based on client-server architecture. In this architecture, any requests from the user end go to the Docker daemon, which

is responsible for the actions on the images, containers, volumes, etc . This daemon listens for Docker API REST requests that are sent via a network interface, then handles the action on Docker objects. The Docker client offers a command line interface (CLI) via which you may request the Docker daemon to perform some actions. For instance, in order to run a container on a Docker host, using Docker commands, initially will pull the container image from the public Docker registry (Docker Hub), which stores Docker images [5].

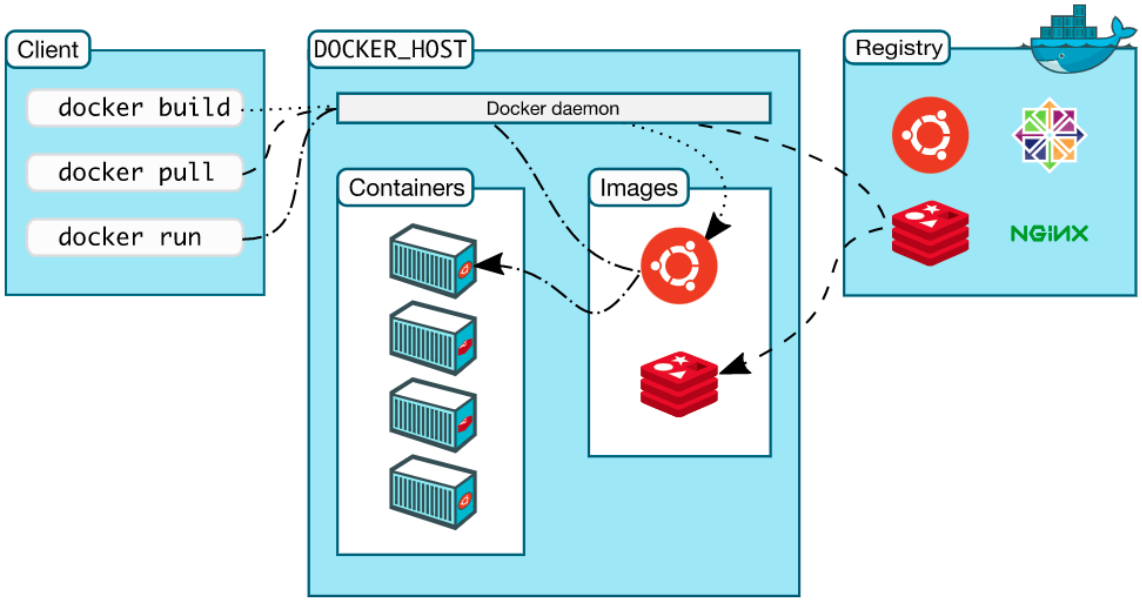

Figure 6: Docker Architecture [5]

Container Orchestration is the automatic process of provisioning, deployment, networking, scaling, availability, and lifecycle management of containers [41, 43]. Following, we cover one of the well-known Container Orchestration systems and discuss its architecture and benefits.

2.4 Kubernetes

We start by giving a brief background about **Kubernetes**. Kubernetes is an open-source container-based distributed system orchestrator which helps to deploy a highly available, reliable, and easily manageable distributed system automatically.

Kubernetes APIs are being used to offer many services via the network [6]. These APIs are frequently given through a distributed system, with the many components that implement the API running on separate machines connected by a network and coordinating their operations through network communication. Because we increasingly rely on these APIs for various areas, they must be extremely dependable. Even if a section of the system crashes or ceases operating, their failure should not lead to the general failure of the entire system, and their recovery process should be as seamless and as lossless as possible. They must also ensure availability during software rollouts. Along with the increasing number of service requests, they must ensure scalability to expand service response and keep up with ever-increasing demand without requiring a fundamental restructure of the distributed system that implements the services [41].

There are various advantages to using containers and container APIs like Kubernetes. Some of the fundamental reasons to take advantage of Kubernetes are such as [41]:

- **High availability** – Kubernetes provides a self-healing technology for its managed microservice-based applications. When a Kubernetes' host fails, it can resume failed containers and replace or reschedule them [44] to eventually resume the computation. Forensic computation potentially involves the the computing of a very large amount of data extracted for a large number of different data sources. An integrated forensic computing system is thus prone to experience failure in extracting data from any of these diverse sources, and the failure of one source should not impact the processing of other sources. Given that many such failure are likely to happen, it is primordial to have an automated mechanism to somehow automatically resume processing from a failed node execution. Hence high availability is a key factor for an integrated forensic computing system.

- **Scalability** – Kubernetes provides scalability by separating components from one another using specified APIs and service load balancers. APIs and load

balancers keep each component of the system separate. Load balancers offer a buffer between operating instances of each service, whereas APIs provide a buffer between the implementer and the user. This design makes it easy to scale and increase the size of the program without having to alter or reconfigure any of the other layers. Additionally, when it comes to scaling the services, since the containers are immutable and the configurations are declarative, scaling up the services is only a matter of modifying the configuration file [6, 41]. Given the potential extreme volume of data that may need to be processed by an integrated forensic processing system, scalability is an essential characteristic for its useful and viable implementation.

- **Abstraction** – Kubernetes provides abstraction features, which allow the applications to be moved across any environment after deploying and managing it. When developers build their applications, moving the applications across various environments happens merely by moving the declarative configuration to adapt to the different context [6, 41]. Given that an integrated forensic computing system is by definition extracting data from sources that vary widely in their context and configuration, it is essential that its various computation nodes be abstractly defined and executable in different contexts.

- **Efficiency** – Kubernetes offers several valuable features for resource management. In addition, one of the benefits of abstraction in Kubernetes is that since there would be no need to think about the machines, applications may coexist on the same machines without any issues; thus, several user applications can be merged into fewer machines [45] [41].

Using Kubernetes you can automate many tasks related to the management of the applications using Kubernetes since it comes with built-in commands that will take care of these tasks. Kubernetes handles all the computation, networking, and storage. Therefore, developers focus on the applications rather than the underlying environment and only interact with Kubernetes API . Kubernetes also has a mechanism that when a container fails, Kubernetes restarts the container and only

makes the service available to users after confirming that it is operating. Kubernetes continually monitors the health of your services [6, 41]. In the following, we begin by illustrating the architecture of the Kubernetes.

2.4.1 Kubernetes Architecture

Before we illustrate the architecture of the Kubernetes, we begin by providing some fundamental components.

- **Pods** are the smallest units in the Kubernetes cluster, which can be created and managed by developers. It contains one or multiple containers with shared storage and network resources. The pod packages all containers, storage assets, and a temporary network identity as a single unit. A pod's IP address and port space are shared by the containers in it [6].

- **Deployment** is another abstraction on the top of pods in Kubernetes, which is used to create or modify instances of the pods. It can scale up and scale down the number of replicas of pods [6].

- **Services** are an abstraction that defines a set of pods and a policy that provides the IP address and DNS name to access the pods. By creating and deleting pods, each gets its IP address, therefore, it would be challenging to communicate with the pods via their IP address. To avoid this problem, Kubernetes allows us to assign a label to the pods and select them according to their service labels. This way, once a pod is deleted and recreated, it has the same label [6].

- **Volume**: In order to preserve the data produced by the container and for scenarios like sharing file systems among containers or backing up the data, Docker has a volume object, which mounts these file systems on the Docker container, and they are preserved on the host machine. Containers virtualize an operating system, allowing us to run multiple containers on a single machine and a shared operating system [6].

- **Persistent Volumes (PV)**: PV is an API that abstracts the implementation of physical storage for the pod's usage, whose lifecycle is independent of the pods. They can last longer, even they can get accessible after a container restarts [6].

- **A Persistent Volume Claim (PVC)**: It is a request from a user for PV storage, which specifies the level of resources, size and access modes for the storage. PVCs use up PV resources [6].

- **Network File System (NFS)**: It is a shared filesystem volume which allows to mount an NFS share into a pod. The data of this volume is kept intact, so when a pod is destroyed, the data in the NFS will not be deleted. It also allows data to be pre-loaded and shared between pods [6].

A Kubernetes cluster consists of a control plane (master) node and multiple worker nodes. For high availability, Kubernetes as multiple worker nodes. The control plane runs on a cluster machine and has all Kubernetes objects. The control plan manages the object states and any modification on the cluster. It decides on the cluster's big picture [6, 41].

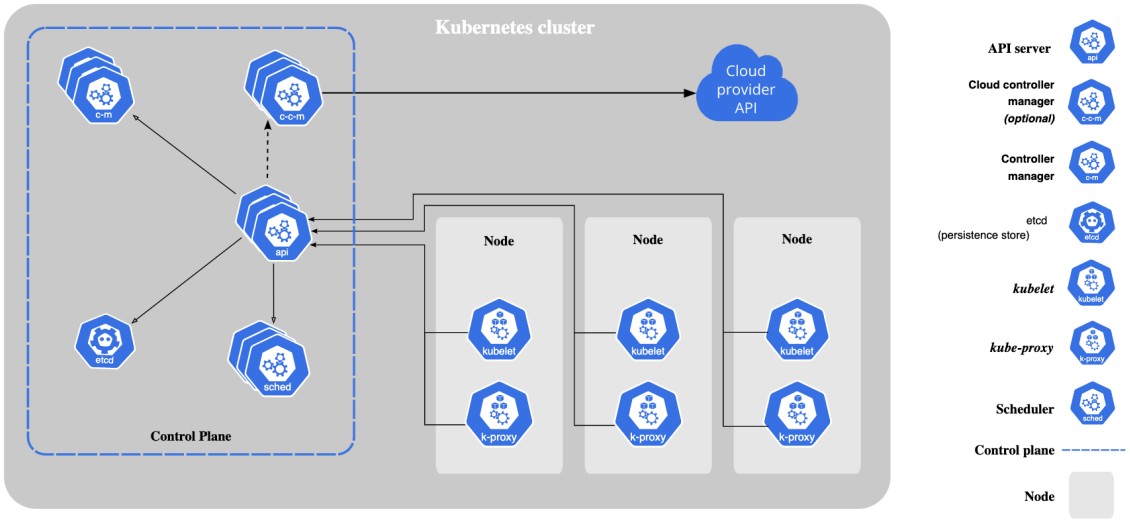

Figure 7: Kubernetes Architecture [6]

2.4.1.1 Control Plane

The control plane consists of multiple essential components [6]:

- **kube-apiserver**: Responsible for exposing the Kubernetes API, which supports REST operations and acts as a front-end for all components to communicate with the clusters' shared state.
- **etcd**: Consists of a key-value store, which is used for keeping the cluster configuration. It is also used for storing the state of the cluster, and we can monitor the modifications on the cluster.
- **kube-scheduler**: Responsible for assigning a created pod to a node, which is designated based on some parameters such as resource provisions, etc.
- **kube-controller-manager**: Responsible for managing various controllers in Kubernetes. These controllers are non-terminating control loops that monitor the cluster's condition and request or make modifications in the cluster. Examples of existing controllers: *Node Controller*, which is responsible for detecting when nodes go down, *Replica Controller*, which is responsible for monitoring the state of replica sets and if a pod dies, will reproduce another one, etc.

2.4.1.2 Worker Node

As we mentioned above, each worker node runs Kubernetes workloads and hosts one or multiple pods inside of it. Below are the essential components of each node [6]:

- **kubelet**: It is responsible for the running pods in the Kubernetes nodes and monitors the health of containers.
- **kube-proxy**: It is a network proxy that runs on each node and is responsible for maintaining network rules on each node, enabling communication across the network from network sessions both inside and outside of your cluster to your pods.

- **Container runtime**: It is the software component that is used for running containers. It is responsible for pulling down container images, managing the lifecycle of containers, and isolating its resources for containers.

2.4.1.3 Kubernetes Objects

Kubernetes objects objects are used to represent the state of the Kubernetes cluster. These objects inform us what containers are running on which nodes, the resources available for the containers, IP endpoints for a container group, the number of replicas of a container, policies for the container's behavior, etc. These objects are "record of intent", by creating the object, Kubernetes attempts to ensure that the cluster is in the desired state by comparing it with the current state of the Kubernetes cluster. Creating, modifying or deleting the Kubernetes objects are achievable via Kubernetes API using the Kubernetes command-line tool (`kubectl`). Kubernetes object consists of two essential object fields, object spec and object status. Object spec is the desired state, which describes the attributes of the object and object status is the current state of the object, representing the updated Kubernetes system. In order to create a Kubernetes object, you need to describe the system's desired state, which is possible by providing the object state. Mostly the object information is provided to `kubectl` in the `.yaml` file, where kubectl converts the `.yaml` file to a JSON file to send as an API request to Kubernetes API [6].

Below is an example `.yaml` file, which describes required fields and objects that need to be specified when we create a deployment. The required fields consist of [6]:

- **apiVersion**: It is the Kubernetes API version that you are utilizing to construct this object.
- **kind**: It indicates the type of object you wish to create (pod, deployment, service, etc.).
- **metadata**: It is the data that helps us to determine the object (name, etc.).
- **spec**: It describes the desired state of the object. Depending on the object

type, the spec file may also vary.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 2.2: Simple deployment

In our research, we take advantage of all mentioned features in order to have a scalable fault-tolerant and easily manageable system as we mentioned in our scenario in Section 1.2.2.2.

2.5 Summary

In this chapter, we began by providing a background related to FORENSIC LUCID, in Section 2.1. Thereafter, we discussed the GIPSY distributed system in Section 2.2 and described the components of the GIPSY distributed system in more detail. Afterwards, we briefly discussed containers, container images, a software platform (Docker) that we use in order to manage the containers, and Container Orchestration in Section 2.3. Finally, we described Kubernetes, discussed its architecture and presented the features we employ in our research in Section 2.4.

Chapter 3

Methodology

3.1 Solution Overview

In this section, we first discuss our solution to achieve the requirements that we mentioned in the Section 1.2.3 regarding the GitHub demand-driven JSON to Forensic Lucid Encoder Formalization in GIPSY thereafter, we explain the reason for choosing our solution related to the Kubernetes cluster for the GIPSY system and different methods that we used for designing and implementing our proposed solution.

3.1.1 GitHub JSON to Forensic Lucid Encoder

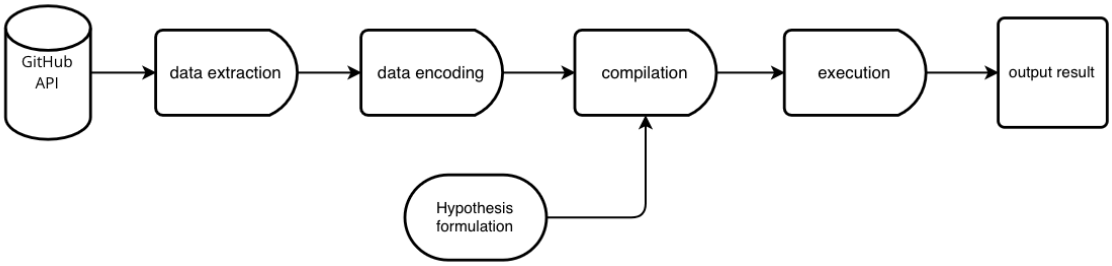

Figure 8: Forensic computing pipeline

The JSON to Forensic Lucid encoder is a Java program created to transform a JSON string into a Forensic Lucid program which is then provided to the GIPSY system for compilation.

In order to implement different JSON parsers for various sources such as Twitter, GitHub, etc., a `IJsonParser` interface has been developed so that we can, according to the different applications, implement a corresponding JSON parser [7, 33]. For instance, in order to create a JSON parser, as we discussed in the Section 1.2.2.1, we can implement `IJsonParser` corresponding to the GitHub JSON structure.

In Figure 8, we illustrate the pipeline for Forensic Computing, which consists of multiple primary steps such as fetching data in JSON format from the GitHub API repository, converting all JSON files to Forensic Lucid program, compile the Forensic Lucid files using the hypothesis formulation that is provided by the forensic investigator and finally execute them and returning the results.

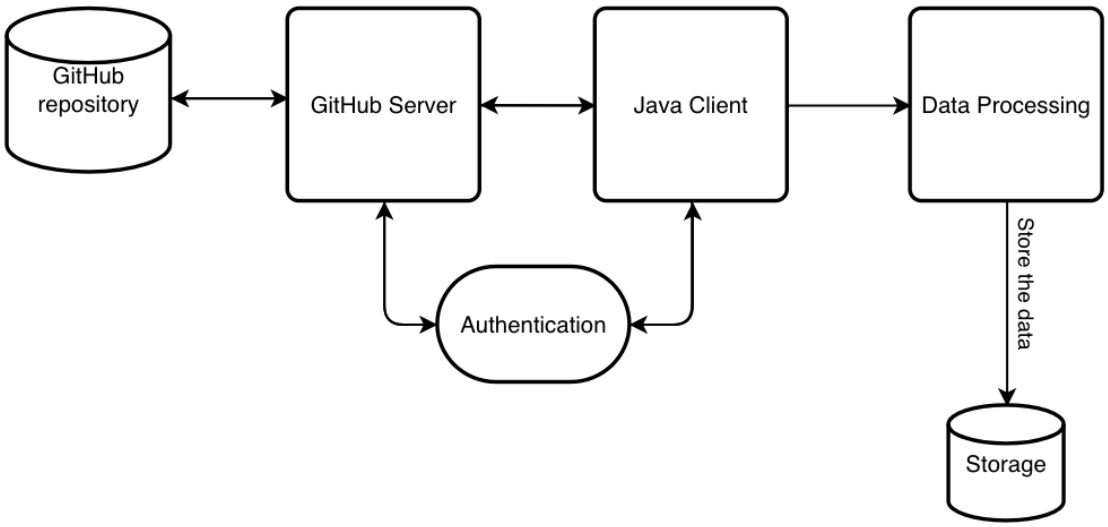

Figure 9: Data extraction form GitHub repository

Figure 9 illustrates the fetching process from the GitHub repository. At first, the Java client sends a REST API request by providing a token to the GitHub server in order to authenticate, and after successful authentication, the Java client sends another REST API request to the GitHub server for the specific project from the GitHub repository. The GitHub server responds with a JSON response. Once the Java client receives the JSON response from GitHub, the Java client will pass the response to extract the relevant details and keep all the processed data in storage.

By creating observation sequences using the semantic connections between various evidential statement objects on a GitHub repository, which describe reported security vulnerabilities in a project code, a forensic investigator can encode evidential statements in the form of a Forensic Lucid program, see Requirement 1.

In Figure 10 one can observe the use case diagram for the JSON to Forensic Lucid Encoder.

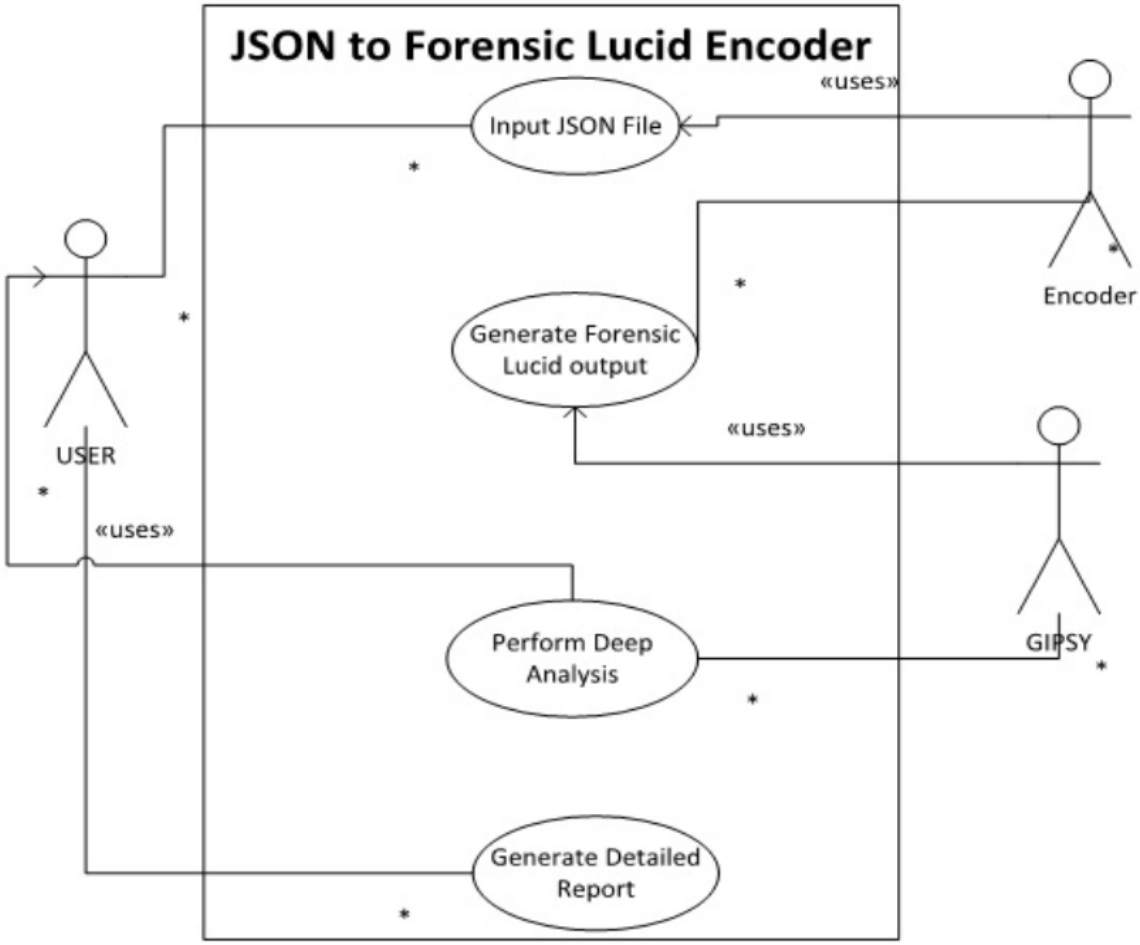

Figure 10: JSON to FORENSIC LUCID Encoder Use Case Diagram [7]

3.1.2 Integrating Kubernetes and GIPSY distributing system

With what we described in Section 2.2 the GIPSY architecture, we would like to take advantage of Kubernetes to have a distributed system that can be automatic scalable

fault tolerant and easily manageable.

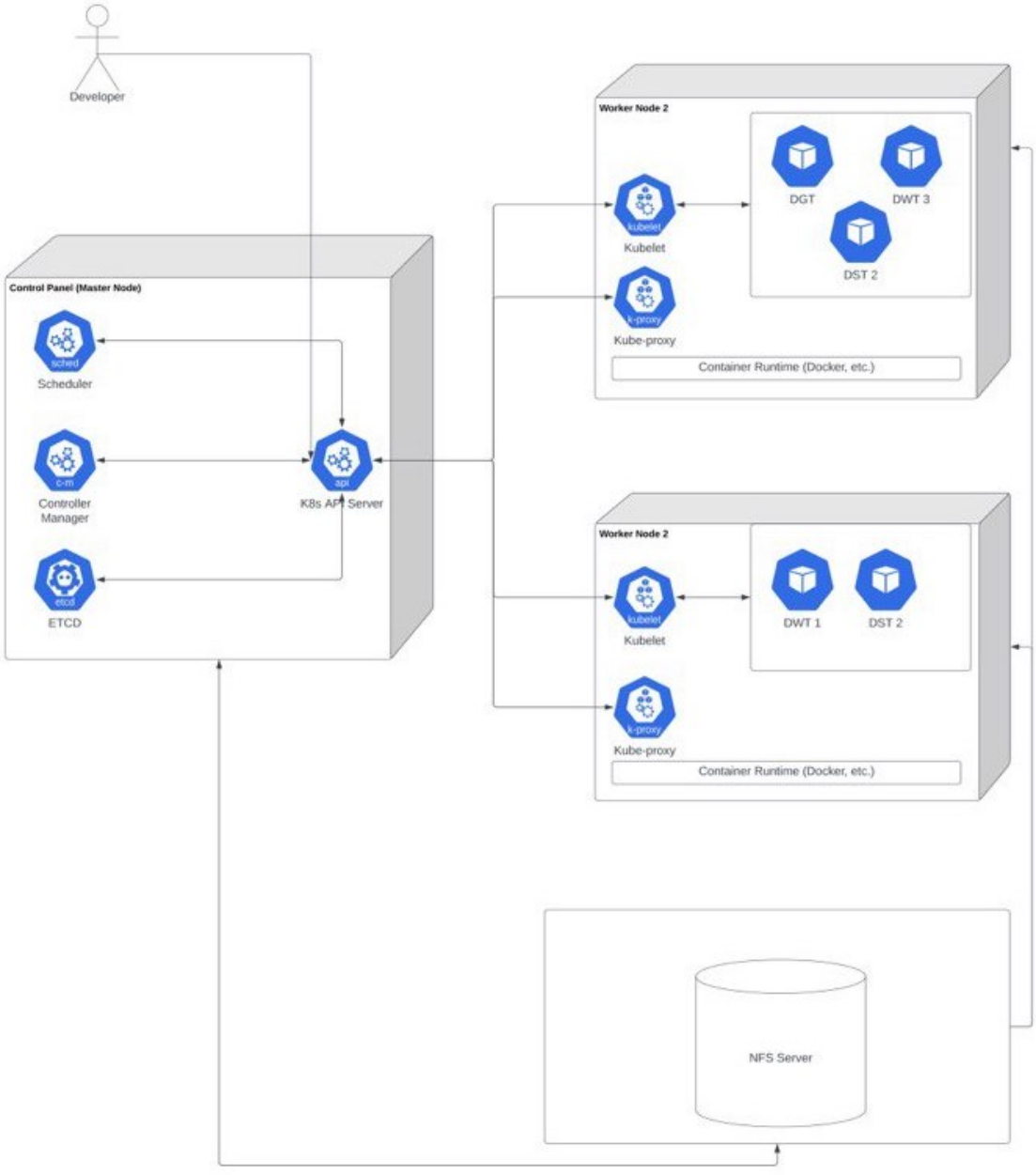

Figure 11: Kubernetes Integration with GIPSY

As we illustrate in the Figure 11, we have one Control Plane and multiple Worker nodes in our architecture. Control Plane consists of essential components as we described in the Section 2.4. In order to work with the cluster, Developer will transmit its request to the Kubernetes API Server, which is running on the Control Plane node

and will take care of the requests by communicating with Kubelet. Each Worker node contains Container Runtime, Kubelet, Kube-proxy and multiple pods, which we will describe in more detail. In this paperwork, we utilized Docker as a Container Runtime platform.

In order to make this integration work, we containerized the GIPSY program with Docker so a GIPSY tier can operate inside each pod. By deploying each tier inside a pod, we can quickly bringing the cluster up and down by adding a new pod or removing a pod from the Kubernetes cluster. This modification would not affect other GIPSY tier pods' performance.

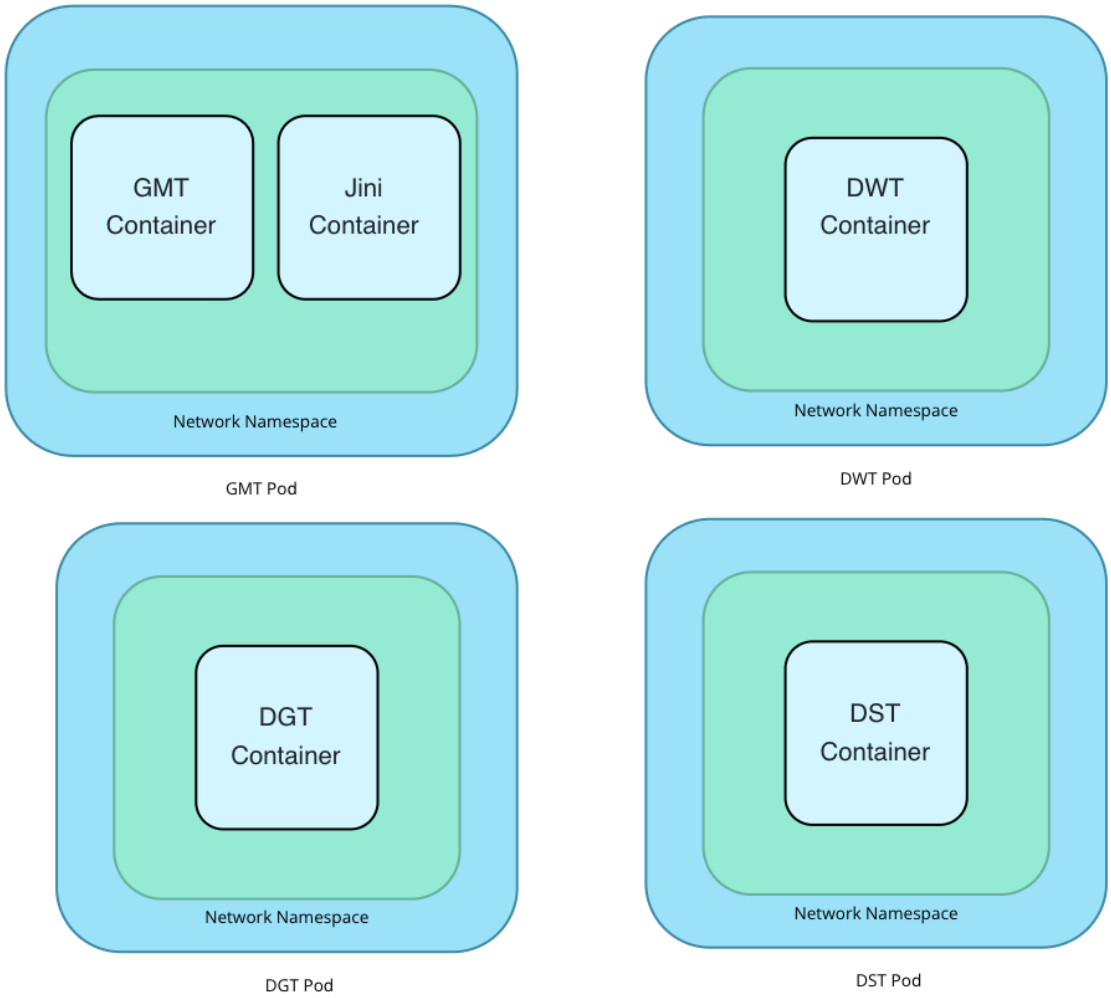

Figure 12: GIPSY Tier Pods

In Figure 12 one can observe that each pod consists of a Network Namespace,

GIPSY tier Container (GMT, DGT, DWT, DST) and additional supporting containers if needed. In our architecture, as an additional supporting container, we have a Jini Container to transport and store their results' demands for the DGT pod. In addition, we added an NFS Server to our cluster to help us with sharing the configuration files in the initializing state of the cluster and in future to have a backup from DST, which we will discuss more in detail in the future work section. As depicted in Figure 13, for each GIPSY pod, a PersistentVolumeClaim (PVC) with a specific amount of storage and with certain access modes is created, and the PersistentVolume is attached to the NFS server. Kube-controller-manager (control loops that watch the state of the cluster) look for new PVCs, find a matching PV (if possible), and bind them together. Once the claim is bound, each pod has access to the shared storage resources.

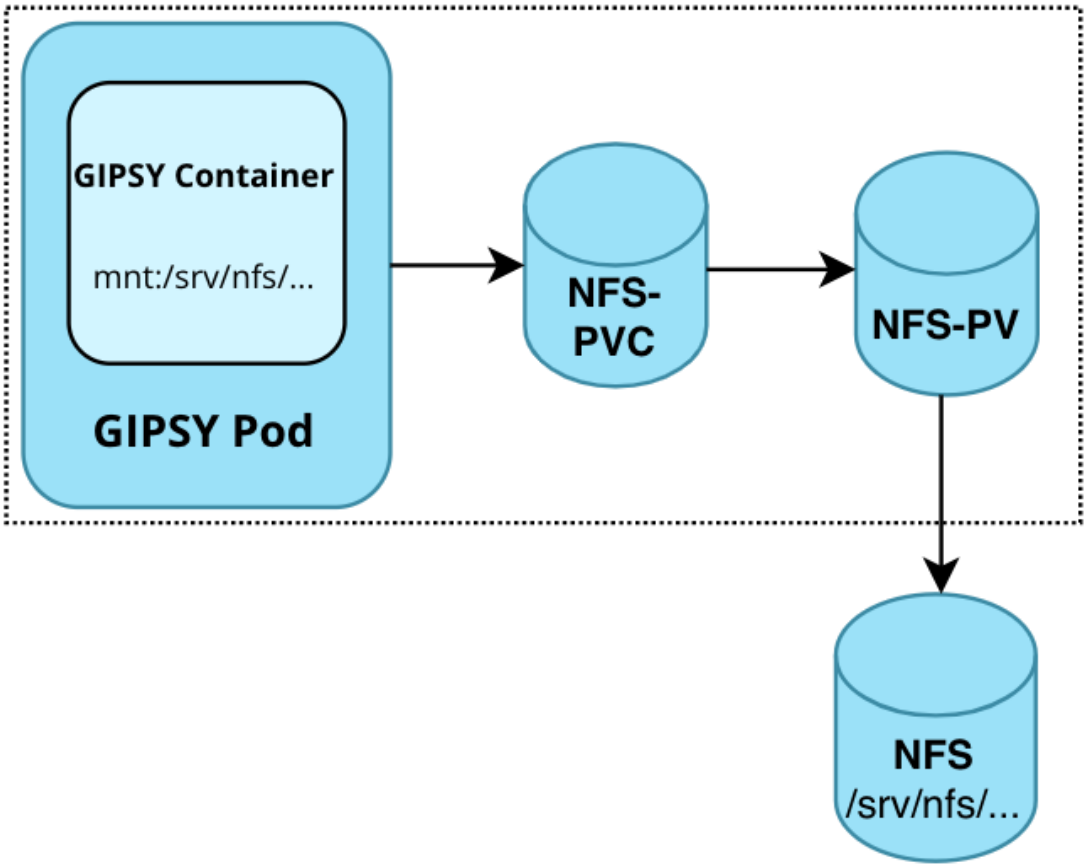

Figure 13: NFS

3.2 Evaluation Methodology

In this section, we continue by providing the reader with methods we follow in this research work to evaluate that our solution truly solves the problems we mentioned in Section 1.2.2 and that our solution meets the requirements we mentioned in the Section 1.2.3.

We proposed the following experiments to evaluate our solution in practice:

- In order to evaluate our JSON demand-driven encoder and verify Requirement 1, we collect various publicly announced computer security flaws that were committed in the GitHub repositories and test our program using the list of the commits URLs that we gather for different repositories. We use the GIPSY distributed computing system to spread the pipeline's execution. Therefore, we allocate a DGT, a DST and multiple DWTs to run the experiments.

- Then, in order to evaluate the integration of the GIPSY and Kubernetes, we first set up the NFS server on the control plane node and the worker nodes and deploy persistent volume (PV) and persistent volume claim (PVC) to mount to the shared storage directory so that each GIPSY pod has access to the shared files and directories, see Requirement 6.

- Subsequently, on the control plane node we create a .yaml file corresponding to each pod for GIPSY nodes and the number of replicas for the workers. Then by running the Kubernetes command line and applying each configuration file, we have a GMT pod and multiple regular pods automatically registered. Later, the GIPSY tiers will be assigned to each regular pod, see Requirement 4.

- Once the GIPSY node is registered in the cluster (at first, we run one pod per GIPSY node), again on the control plane node, we begin to allocate the GIPSY tiers, and we test the JSON demand-driven encoder on the Kubernetes cluster by providing the same list of the commits URLs as we collected before. We run various experiments for JSON demand-driven encoder in the Kubernetes cluster

by bringing up and down the number of GIPSY pods to verify the performance of the Kubernetes cluster depending on the number of GIPSY pods. Then we attempt to register various numbers of GIPSY nodes in one pod and run the same experiments for the JSON demand-driven encoder.

- Finally, to verify the fault tolerance using Kubernetes integration (Requirement 5), we attempt to switch off one of the worker nodes and prove if Kubernetes re-allocate the running GIPSY pods to the next available worker node and verify if GIPSY nodes inside the pods get registered automatically.

3.3 Summary

In this chapter, we offered the reader an overview of our approaches to accomplish our objectives and meet our varied requirements. We will go into more detail about how to create and develop our solution in the following chapter, as well as how to put it into practice.

Chapter 4

System Design and Implementation

In this chapter, we discuss our solution's design and implementation in more detail. Chapter 3 described the theories and methods that we employed in our solution to have a GitHub demand-driven JSON to Forensic Lucid Encoder Formalization in GIPSY and integrating Kubernetes with GIPSY in order to distribute the execution. In the following, we will discuss the classes and architecture of our JSON to Forensic Lucid Encoder implementation. Afterward, we will represent how we implement and establish the Kubernetes integration with the GIPSY distributing system alongside UML class and sequence diagrams..

4.1 GitHub JSON to Forensic Lucid Encoder

Before diving into the encoder, we need to get familiar with the **JSON** (JavaScript Object Notation). JSON is a language-independent and data interchange text format which is used for storing and transmitting data and makes it easy for the server and applications to write and read data. Over the last years, many different web service APIs have utilized JSON as a data format [7]. Below we present a sample of a specific JSON data file that is passed to the encoder.

```
{
    "sha": "156c1bd163fbc735d07a98970a504408d5a6a9e3",
    "node_id": "MDY6Q29tbWl0Mzg1NjkyNzAwOjE1NmMxYmQxNjNmYmM3MzVkMDdhOTg5NzBhNTA0NDA4ZDVhNmE5ZTM=",
    "commit": {
```

```
      "author": {...},
      "committer": {...},
      "message": "...",
      "tree": {...},
      "url": ...
      "comment_count": 0,
      "verification": {...}
    },
    "url": "...",
    "html_url": "...",
    "comments_url": ".../comments",
    "author": {... },
    "committer": {...}
}
```

Listing 4.1: JSON data format sample

GitHub APIs allow end-users to fetch or extract data from any repository on GitHub easily. Using an authorized access token, users of GitHub APIs may communicate with GitHub and retrieve the needed data. In order to access repositories that are not accessible to the general public, users can additionally authenticate with regard to the username using the GitHub REST API [46]. In this research work, we put more of an emphasis on the Commits API, which gives you access to commit comments and statuses as well as a list, view, and comparison of all the commits in a repository.

As depicted in Figure 14, the conversion is a linear dataflow graph that consists of three steps, who each are associated with their own kind of demand to be processed: (1) the extraction of the data, which is provided by the forensic investigator in the format of a list of commit URLs for one or different repositories, from the GitHub repository (*fetch demands*) in JSON format; (2) the transformation of the extracted JSON data into Forensic Lucid code (*convert* demands); and (3) the compilation of the Forensic Lucid code (*compile* demands). The conversion pipeline involves at least one of the three GIPSY tiers (DGT, DWT and DST) that will generate and/or process these demands.

In our architecture, each demand can be in a PENDING or COMPLETED state, in which PENDING indicates that the demand requires to be processed and

COMPLETED indicate that the demand's processing is complete. The DGT is responsible for the generation of the demands in the order that is represented in the dataflow graph of the transformation pipeline. Once the demands get into the DST, they are put in the pending state. The DWT (which there can be several execution instances of) is responsible for the execution of these demands, and to put back their corresponding results into the demand itself, at which point the demand is put in the processed state.

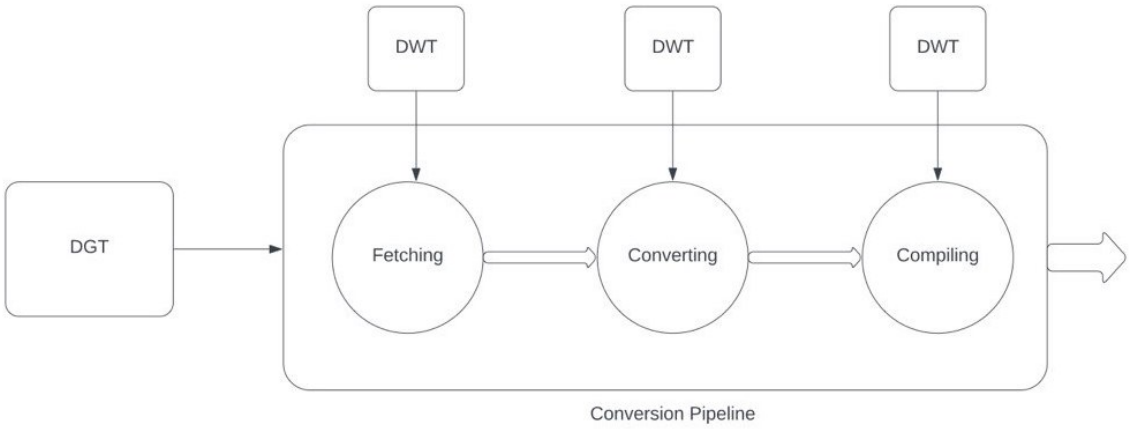

Figure 14: Conversion Pipeline

The pipeline starts by the demand generator (DGT) to generate fetch demands for the processing of specific GitHub queries according to kind of URLs that forensic investigator provides, which in this case are a list of commit URLs. Once the DGT has created the fetch demands, a DWT begins to perform the fetching, and DGT waits for these demands to be processed. After a DWT finishes the fetching process, it proceeds to generate a convert demand to generate the Forensic Lucid code that corresponds to the JSON GitHub fetch results. Later, a DWT by receiving the converting demands, proceeds to convert the the JSON files to Forensic Lucid code and generates a corresponding compile demand to generate an executable version of this Forensic Lucid code, see Requirement 1.

Figure 15 illustrates the sequence diagram for the JSON to Forensic Lucid Encoder.

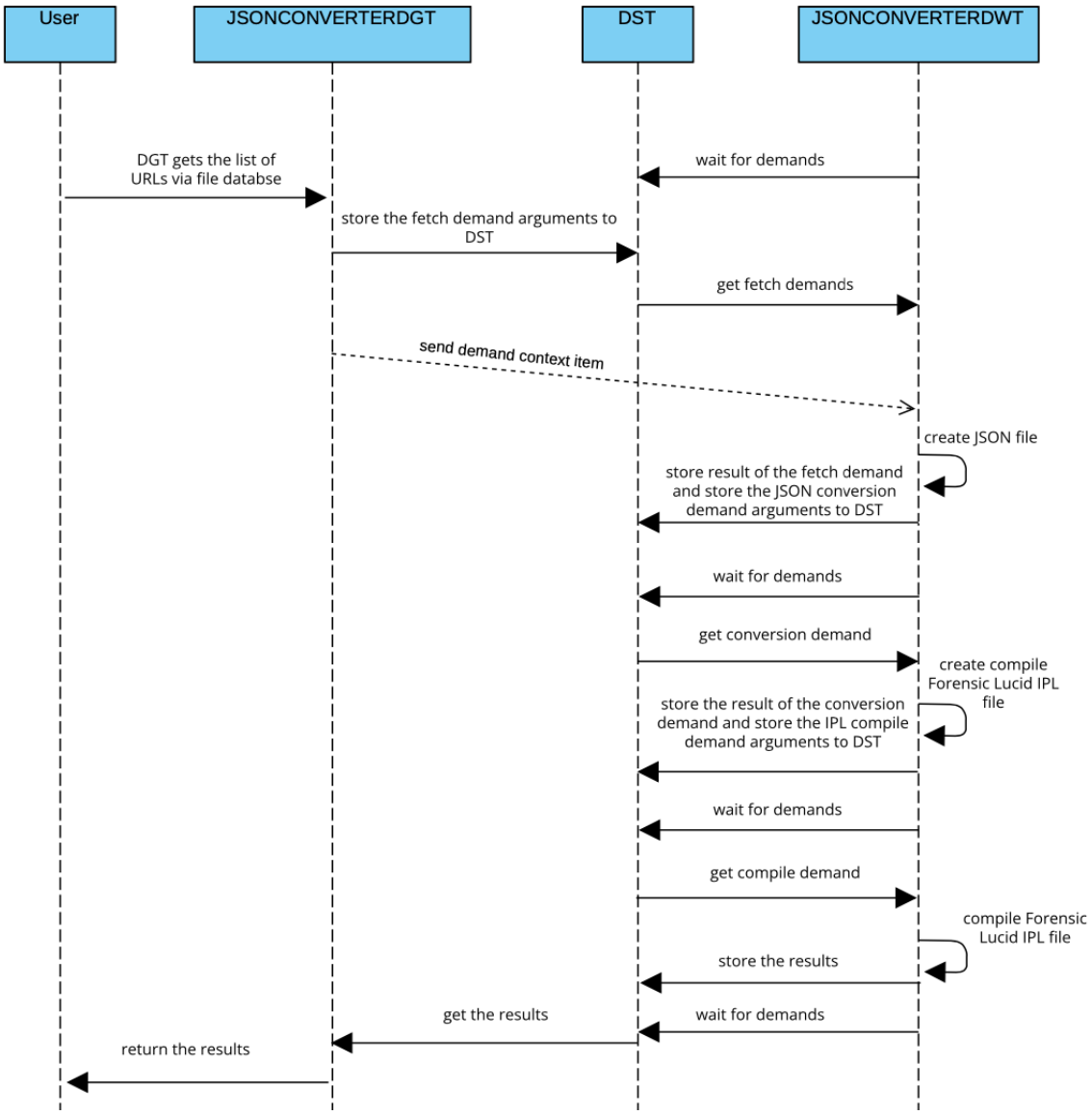

Figure 15: Sequence Diagram for GitHub JSON to Forensic Lucid Encoder

In this diagram, the `JSONCONVERTERDGT` is the demand generator (DGT), `JSONCONVERTERDWT` is our worker (DWT), and DST, as we discussed in the Section 1.2.1, is Demand Store tier, where the demand generator and worker store the demands and the results. The user provides a list of URLs of the GitHub repositories that are forensically-interesting. The demand generator will watch for user requests for the JSON parser and builds a fetch demand index with a status of the demand in the DST and pass it to the worker. DGT uses the URLs as a demand signature

in order to avoid re-generating the same demand, so next time for the same demand from the user, the DGT notices the existence of the same demand signature and instead of creating new demand, since it has the results already in the DST, it merely returns the results, see Requirement 3.

Once the demand is stored in the DST, the worker will retrieve the serialized object from the DST, meaning the worker should select the appropriate JSON parser, which in this case is Git, and its job will be to parse the JSON file, fetching the data from GitHub repository, creating a Java object with the JSON structure, send the object back as a CTX file, which includes the list of JSON file as well as the FORENSIC LUCID observation structure, and include a demand to let the next available workers know that the serialized object has been computed.

These data are fetched from the "commit" endpoint on the GitHub repository. As you can observe, it consists of an evidential statement es, and es has an observation sequence (commit), which consists of multiple observations such as SHA and commit object, and commit object consists of data that are forensically interesting for us.

```
OResult
where
evidential statement es = {o_sequence};

        observation sequence o_sequence = {o_commit_1,o_commit_2,...};
        observation o_commit_1 = {commit, author, committer, parents, stats , files,...};
        commit = {[author : {[name : "..."], [email : "..."], [date : "..."]}], ...};
        author = {[login : "..."], [id : ...], [node_id : "..."], ...};
        committer = {[login : "..."], [id : 17151892], [node_id : "..."], ...};
        parents = {[[sha : "..."], [url : "..."], [html_url : "..."]]};
        stats = {null};
        files = {null};

        observation o_commit_2 = {commit, author, committer, parents, stats , files,...};
        commit = {[author : {[name : "..."], [email : "..."], [date : "..."]}], ...};
        author = {[login : "..."], [id : ...], [node_id : "..."], ...};
        committer = {[login : "..."], [id : 17151892], [node_id : "..."], ...};
        parents = {[[sha : "..."], [url : "..."], [html_url : "..."]]};
        stats = {null};
        files = {null};

        ...
```

```
end
```

Listing 4.2: Simple FORENSIC LUCID from GitHub repository

Once the worker finishes the job, it will send back the results and attach a new demand type of conversion demand to convert the JSON files to Forensic Lucid. DWT will create a Java object with the CTX file containing the list of JSON files and store the demand in the DST.

Then the next available worker will perform the same procedure as it did in fetching the JSON files, it will convert all the JSON files to FORENSIC LUCID format and create a list of IPL files corresponding to the JSON files and store back the list of the IPL files as a result and attach the new demand type of compile demand to compile all the FORENSIC LUCID IPL files. DWT will create a Java object with the CTX file containing the list of IPL files and store the demand in the DST.

Once the DWT grabs the compilation demand from the DST, it will begin to compile the Forensic Lucid files and return the output of the compilation as results to the DST including the list of JSON files, IPL files and GIPSY files. Eventually, DGT will pass the final results to the user, see Requirement 2.

The system sequence diagram for JSON to FORENSIC LUCID encoder is illustrated in Figure 16.

4.1.1 Context File Item

As we discussed before, the CTXFileItem object contains the FORENSIC LUCID observation structure and JSON file contents, which is used to store in the DST. The class diagram in Figure 17 illustrates the content of this object.

- `stringParserType` - It represents the type of JSON parser.
- `time` - It represents the creation time of the `CTXFileItem` object.
- `url` - It represents the GitHub repository URL we seek to fetch the data in order to perform the conversion.

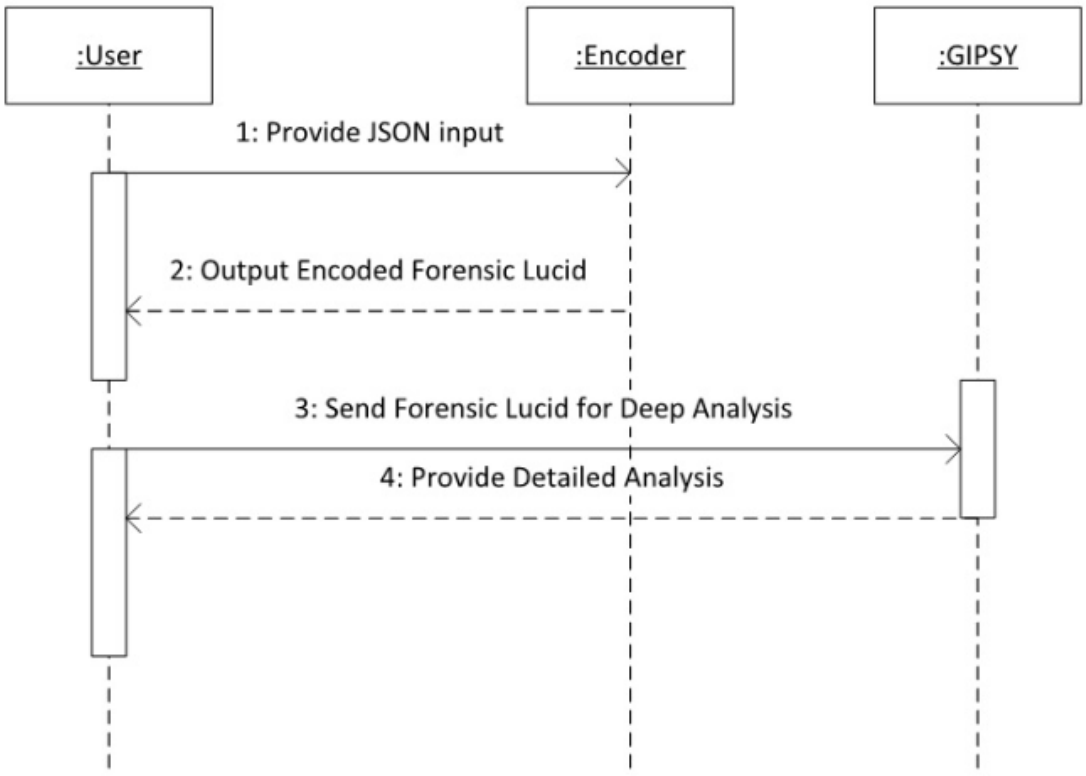

Figure 16: System Sequence Diagram for JSON-to-FORENSIC LUCID Encoder [7]

- `urlSince` - It eliminates all commits made before the specified date.
- `urlUntil` - In contrast to "urlSince" this excludes any commits made after the specified date.
- `jsonDir` - It represents the directory to the list of JSON files after the fetching demand is processed.
- `IplDir` - It represents the directory to the list of IPL files that is stored after the conversion demand is processed.
- `jsonFiles` - It represents the list of JSON files in the `ArrayList` of `String` type that is stored after the fetching demand is processed.
- `iplFiles` - It represents the list of IPL files in the `ArrayList` of `String` type that is stored after the Conversion demand is processed.

- `gipsyFiles` - It represents the list of GIPSY files in the `ArrayList` of `String` type that is stored after the Compile demand is processed.

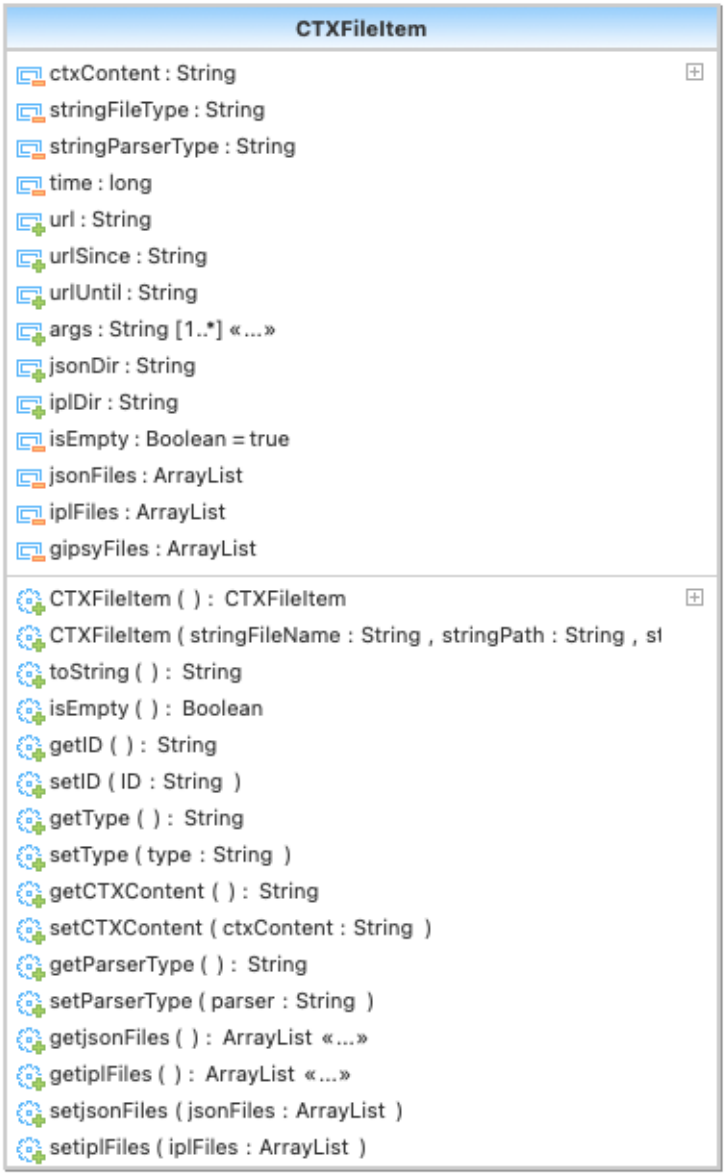

Figure 17: CTX Class Diagram

4.1.2 GIPSY Demands

The three primary demand type classes, as depicted in Figure **??**, are `JsonDemand` (fetching step), `JsonConverterDemand` (converting step) and

`FlucidCompileDemand` (compiling step).

- `time` - It represents the time of generated demand.
- `jsonParserType` - The JsonParserType class contains an enumeration of all the available JSON parsers (Facebook, Twitter, etc.), which in this case is git.
- `ctxFile` - As depicted in Figure 17, it contains the context file item value.

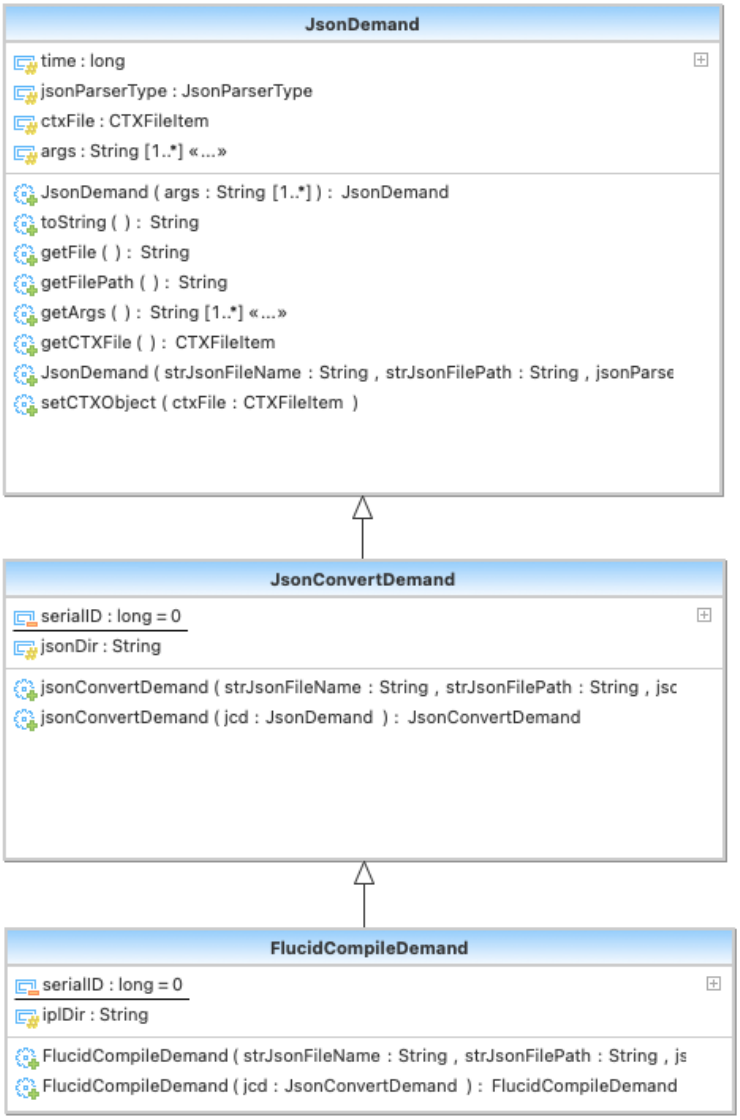

Figure 18: GIPSY Converter Framework Demand Type

4.1.3 JSON to Forensic Lucid Encoder

As discussed in Section 4.1, `JSONCONVERTERDGT` is our demand generator Java program that is responsible for generating and storing the fetching demand into DST, and `JSONCONVERTERDWT` is the worker Java program that is responsible for processing the demands.

Below, we describe the functionality of various methods that are implemented for `JSONCONVERTERDGT`.

- `init()` - This method performs the initialization before the demand generator starts, such as specifying the environment's directory where we wish to store the resource, etc. Create a TA instance
- `startTier()` - This method is responsible for starting the tier instance, creating a TA instance and middle-ware `DemandDispatcher` instance.
- `stopTier()` - As the name suggests, this method is responsible for terminating the tier instances.
- `execute()` - After defining the type of the parser, which in this case is git, this method calls a corresponding function to start the demand generator.
- `generateJsonConversionDemand()` - This method is responsible for starting the demand generator. It reads the list of URLs which the user provides, creates a fetch demand, including a CTX object file, stores the demand to the DST and waits for the results when the demand status is labelled as completed.

Below, we represent the functionality of various methods implemented for `JSONCONVERTERDWT`.

- `init()` - This method performs the initialization before the workers start, such as specifying the environment's directory.
- `startTier()` - Like the demand generator, this method is responsible for starting the tier instance and creating a TA instance.

- `stopTier()` - This method is responsible for terminating the tier instances.
- `execute()` - This method is responsible for defining the type of the parser, providing configuration files and calling the function for starting the demand worker.
- `process()` - This method starts the demand worker and waits for a demand. Once it receives a demand, it verifies the demand type (`JsonDemand`, `JsonConvertDemand`, `FlucidCompileDemand`) and perform the conversion pipeline by passing the type of demand and the CTX file to the `parseJsonFile()` function. In this pipeline, in the case of receiving the demand type of `JsonDemand`, the demand worker, after receiving the results from `parseJsonFile()`, will create a demand type of `JsonConvertDemand`, label the demand as `PENDING`, attach the results of the fetching and store it back in the DST. Accordingly, after receiving the demand type of `JsonConvertDemand`, it will generate a demand type of `FlucidCompileDemand`, attach the results of the converting and store it back in the DST. Finally, once the demand worker receives the demand type of `FlucidCompileDemand`, it will label the demand as `COMPLETED` and store the final results back to the DST.
- `parseJsonFile()` - As explained above, once this function obtains the CTX file and the type of demand, it will begin to perform the conversion pipeline by passing the arguments to the `ParserGitAPI`.

Figure 20 illustrates the class diagram for the `ParserGitAPI`. This class is designed to parse the demands that the demand worker sends including the CTX file and the type of the demand as an argument.

- `parse()` -This method is responsible for the received from the demand worker. It calls the proper method corresponding to the type of demand and returns the send back to the demand worker.

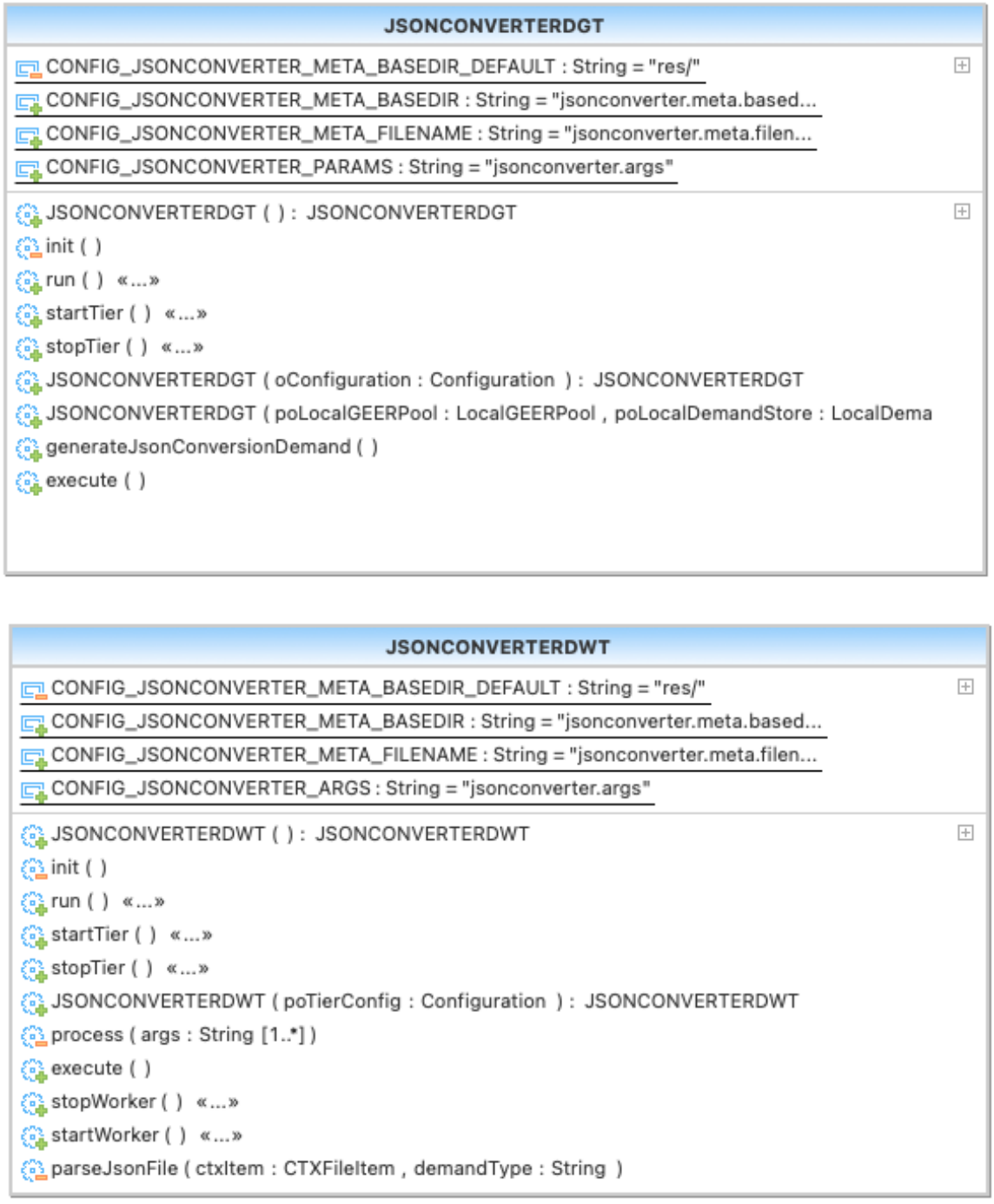

Figure 19: JsonConverterDWT and JsonConverterDGT Class Diagram

- `fetch()` - As the name suggests, this method is responsible for fetching the data from GitHub API. It utilizes the `check_git_limit()` method, which is implemented in order to check the rate limit status for the authenticated user by providing a token.

- `convert_json_ipl()` - As the name suggests, this method is responsible for getting the list of JSON files using the directory that the demand worker delivers and converting it to the FORENSIC LUCID format.

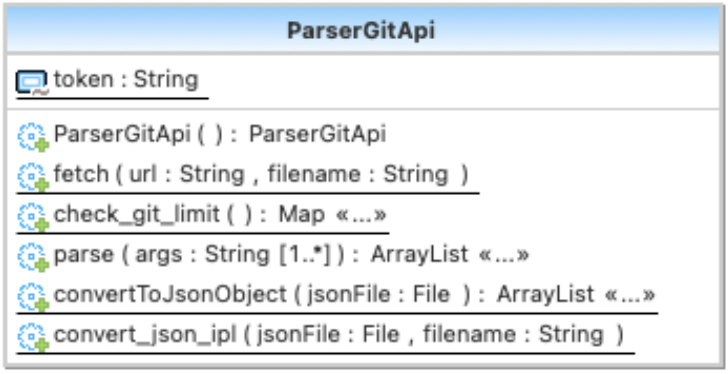

Figure 20: ParserGitAPI Class Diagram

4.1.4 GitHub Generic JSON

As depicted in Listing 4.1, we described the format of the JSON, which is received after we send a request to fetch the list of commits from GitHub API for a particular repository. Using GSON, which is an open source Java library provided by Google, the JSON file and its structure will be taken and mapped to respective java objects. In this research work implementation, the JSON file has a Java object representation called Commit, which contains multiple entries; thus, `ArrayList` of `Commit` is created. The method `fromJson()`, a GSON instance, is called, and it parses JSON string into java Commit object, which is illustrated in Figure 21.

4.2 Integrating Kubernetes and GIPSY Distributed Computing System

Following what we stated in Chapter 3, in this section, we deliver each component of our cluster's architecture and characterize configuration files in detail in order to

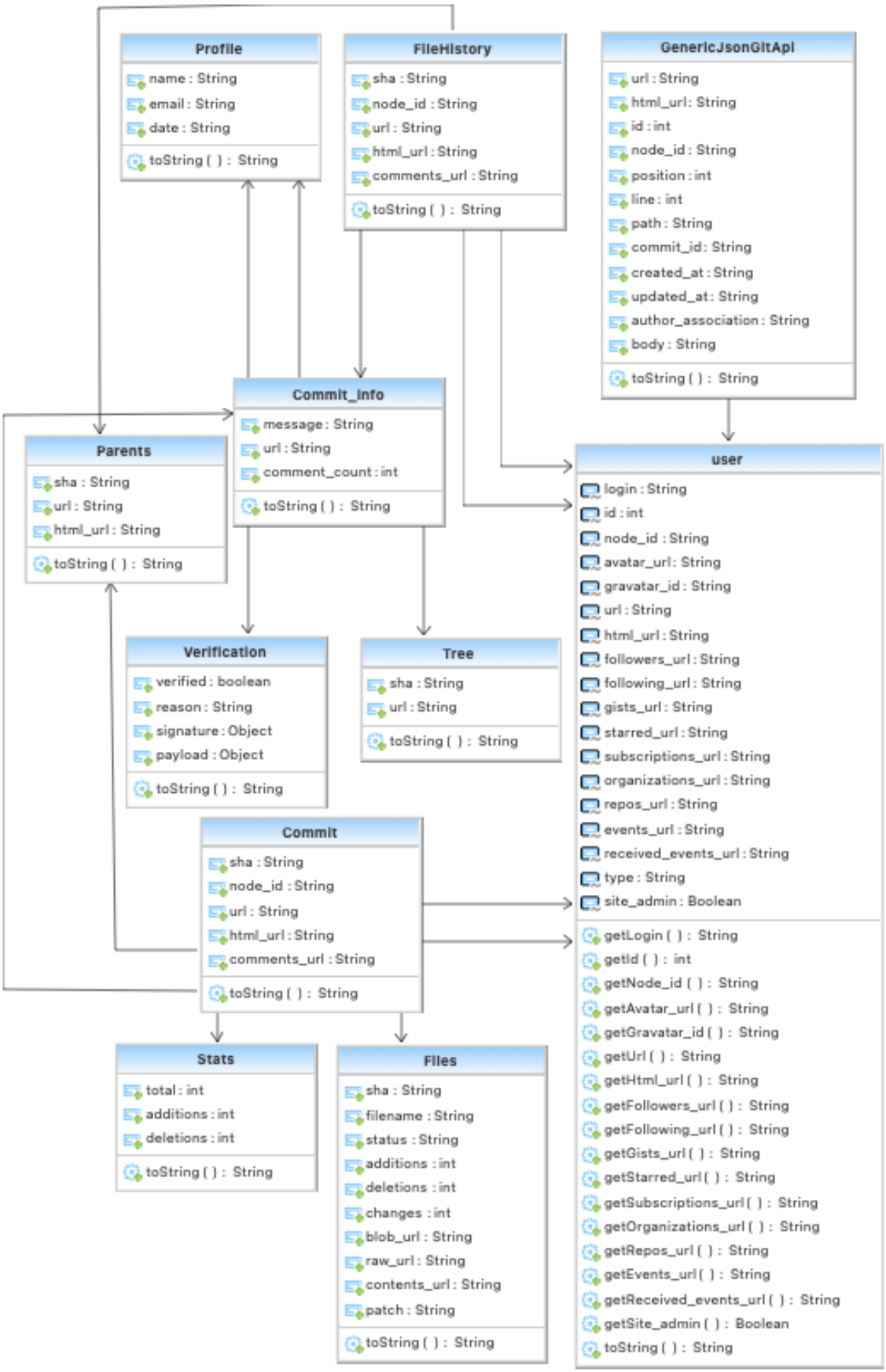

Figure 21: GitGenericJson Class Diagram

integrate Kubernetes with the GIPSY system to distribute the execution of JSON to the FORENSIC LUCID Encoder.

4.2.1 Containerization GIPSY

As we discussed in the Chapter 3, each pod in the GIPSY Kubernetes cluster consists of a GIPSY tier (GMT, DST, DGT or DWT). Therefore, we containerized each of these GIPSY tiers using the Docker platform. In order to build a Docker image, we need to create a Dockerfile, a text document containing all the commands that a user usually calls on the command line to create an image. After creating the Dockerfile, we can build the image and push our container to the Docker hub repository.

This Dokerfile in Listing 4.3 consists of some essential commands such as FROM, RUN, WORKDIR and ENTRYPOINT:

- The **FROM** keyword inform the Docker to use a base image, which in this example is the Ubuntu image repository.
- The **RUN** keyword tells the Docker to execute some instructions when building the image.
- The **WORKDIR** keyword specifies the working directory of the Docker container.
- The **ENTRYPOINT** keyword allows specifying a command along with the parameters when a container is started from a Docker image.

```
FROM ubuntu:18.04
SHELL ["/bin/bash", "-c"]



# Add dependencies
RUN apt-get update && \
        apt-get install --yes build-essential && \
        apt-get install --yes git && \
        apt-get install --yes openjdk-8-jdk && \
        apt-get install --yes xterm && \
        apt-get install --yes iputils-ping && \
```

```
        apt-get install --yes vim && \
        apt-get install --yes net-tools && \
        apt-get install --yes xauth && \
        apt-get install --yes sudo && \
        apt-get clean

# Setup user + sudo
RUN useradd -m gipsy && (echo gipsy:gipsy | chpasswd)
RUN adduser gipsy sudo
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
USER gipsy

# Setup GIPSY repository
WORKDIR /home/gipsy

# build.sh prepares this on the host from the repo
COPY gipsy-src-json-encoder .

# Inside the container you'd need your own ssh keys
# to interact with git or use HTTPS

ENTRYPOINT ["/bin/bash"]
```

Listing 4.3: GIPSY Dockerfile

4.2.2 Configuration and Setting up Kubernetes

In this section, we continue to discuss the configuration and how we set up Kubernetes in order to integrate it with GIPSY, as demonstrated in Figure 11. In order to install the Kubernetes corresponding to Figure 11, we have different machines (virtual machine or physical machine), one of which we will be using as a Control Plane node and the others as a Worker nodes.

In order to build Kubernetes clusters on these machines, we employ **Kubeadm Tool**. `kubeadm` ensures that essential effort has been taken to have a Kubernetes cluster up and running [6]. Kubeadm will automatically handle the installation and configuration of Kubernetes components such as the API server and Controller Manager.

Another required tool is kubelet, which is a node agent that controls node-level

processes that is installed on each node. It handles pod specifications defined in YAML or JSON format, takes the pod specifications, and determines whether the pods are operating healthily. In addition, in order to communicate via API Server with the Kubernetes cluster, Kubernetes provides us with the `kubectl` tool, which is a command line tool (CLI).

Below are the steps that need to be taken in order to achieve our goal [6].

- Install and activate Docker on Control Plane and Worker nodes.

```
$ sudo apt update
$ sudo apt install docker.io
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

- Install Kubernetes on all of the nodes.
  - Updating the *apt* package index and installing packages needed to use the Kubernetes *apt* repository.

```
$ sudo apt-get update
$ sudo apt install apt-transport-https curl
```

  - Download the Google Cloud public signing key and add the Kubernetes *apt* repository.

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
$ sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

  - Eventually, update *apt* package index, install kubelet, kubeadm, kubectl and kubernetes-cni.

```
$ sudo apt update
$ sudo apt install kubeadm kubelet kubectl kubernetes-cni
```

- If memory swapping is enabled on the host machine since the Kubernetes scheduler is responsible for selecting the most available node, it can affect Kubernetes' performance. Therefore, we need to disable Swap Memory.

```
$ sudo swapoff -a
$ sudo nano /etc/fstab #Inside this file, comment out the /swapfile line.
```

- In order to run Kubeadm, we executed the following commands on the Control Plane node:

```
kubernetes-master:~$ sudo kubeadm init
kubernetes-master:~$ mkdir -p $HOME/.kube
kubernetes-master:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
kubernetes-master:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

- Deploy a pod network on the Control Plane node:

```
$ kubectl apply -f https://.../kube-flannel.yml
$ kubectl apply -f https://.../kube-flannel-rbac.yml
$ kubectl get pods --all-namespaces
```

- Eventually, in order to join each Worker node to the cluster, we had to run the following command:

```
$ sudo kubeadm join xxx.xxx.xxx.xxx:xxxx \
    --token ... --discovery-token-ca-cert-hash \
    sha256:...
```

4.2.3 Deploying GIPSY System in Kubernetes

In Section 4.2.2, we discussed some essential configurations that needed to be done in order to run a Kubernetes Cluster. We continue with describing the configurations we applied to deploy the GUPSY pods on Kubernetes Cluster, as discussed in Figure 11.

- **Deploying NFS**

  There are some configuration files which are essential in order to start each tier (GMT, DST, DGT, DWT). For instance, `RegDSTTA.config` is a configuration file consisting of URI, hostname, and some unique id which is generated after

starting the GMT. In order to share these types of data across the network, we should use the Network File System (NFS).

After installing the NFS on the machines, we need to provide the PV and PVC to mount the NFS volume to each pod, see Requirement 6.

– By creating the following file, we define a PV. This `yaml` contains the PV's name, which is used in pod definitions, storage space allocated to this volume, `accessModes` labels that are used to pair a PV with a PVC, the volume reclaim policy `Retain` that is used to ensure the volume will be preserved after the pod's termination, defining the volume type being used (NFS) and indicate the NFS mount path and the IP address of NFS server.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-www
spec:
  storageClassName: ""
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /srv/nfs/kubernetes
    server: 132.205.40.48
    readOnly: false
```

– After creating Persistent Volume, we need to create the Persistent Volume Claim (PVC) to bind between a pod and Persistent Volume.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-www-pvc
```

```
spec:
  storageClassName: ""
  volumeName: nfs-www
  accessModes:
    - ReadWriteMany
  volumeMode: Filesystem
  resources:
    requests:
      storage: 100Mi
```

- **Deploying GIPSY**

  As mentioned in the Figure 12, the GMT pod consists of a GMT container and a Jini container in order to transport and store the results of demands for the DGT pod.

  This `yaml` file contains a service configuration that provides the IP address and DNS name to access the pods and the pod configuration.

  Inside the pod configuration, we deploy one container to run the GMT and another to run the Jini. In the GMT configuration, we specify the name of the container, the name of the image that we are going to pull, define `volumeMounts` which contains the NFS volume name and the directory of the required configuration files and directories that we wish to share and store and express `command` in order to run when creating the pods. Specifying a `command` will override the default command provided by the container image. The command `./start.sh` is used to run the Jini. In the GMT pod, we run the `./startJSONCONVERTERGMTNode.sh` using `gmtd`, a GMT daemon, to avoid the GMT Console. Once the GMT pod is created, we will be able to start and allocate DST, DWT and DGT to each regular node (DGT, DST and DWT).

```
apiVersion: v1
kind: Service
metadata:
  name: gmt-svc
spec: ...
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: gipsy-gmt
  labels:
    name: gipsy-gmt
spec:
  hostname: gmt
  subdomain: gmt-svc
  containers:
  - image: s4lab/gipsy-json-u18:02-18.04
    name: gipsy-json-gmt
    volumeMounts:
    - name: nfs-vol
      mountPath: /home/gipsy/gipsy-src-json-encoder/configs/gmtc.sh
      subPath: gmtc.sh
    - name: nfs-vol
      mountPath: /home/gipsy/gipsy-src-json-encoder/bin/multitier/RegDSTTA.config
      subPath: RegDSTTA.config
    command: ["/bin/bash","-c",
    "cd /home/gipsy/gipsy-src-json-encoder/configs
    && ./gmtc.sh -c
    && cd /home/gipsy/gipsy-src-json-encoder/bin/multitier
    && ./startJSONCONVERTERGMTNode.sh -n gmtd"]
  - name: gipsy-jini
    image: s4lab/gipsy-json-u18:02-18.04
    command: ["/bin/bash","-c","cd gipsy-src-json-encoder/bin/jini && ./start.sh"]
  volumes:
  - name: nfs-vol
    persistentVolumeClaim:
      claimName: nfs-www-pvc
```

- When the GMT pod is up and running, we will be able to run the GIPSY tiers (DST, DWT, DGT) pods. Since we can have numerous Workers and DSTs, we use deployment in the following `yaml` file. By defining deployment, we will be able to scale the GIPSY cluster up and down as demanded. In this configuration file, `replicas` define the number of pods that we wish to deploy, we specify the container image, mount the NFS volume to the pod in order to use the generated configuration file by GMT and define the `command` to run when creating the pods. `./starJSONCONVERTERRegularNode.sh -r`

will register a GIPSY regular node in the cluster.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gipsy-regularNode
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gipsy-regularNode
  template:
    metadata:
      labels:
        app: gipsy-regularNode
    spec:
      hostname: regularNode
      subdomain: gmt-svc
      containers:
      - name: gipsy-json-regularNode
        image: s4lab/gipsy-json-u18:02-18.04
        volumeMounts:
        - name: nfs-vol
          mountPath: /home/gipsy/gipsy-src-json-encoder/configs/gmtc.sh
          subPath: gmtc.sh
        - name: nfs-vol
          mountPath: /home/gipsy/bin/multitier/res/iplFiles
          subPath: iplFiles
        - name: nfs-vol
          mountPath: /home/gipsy/bin/multitier/res/jsonFiles
          subPath: jsonFiles
        - name: nfs-vol
          mountPath: /home/gipsy/gipsy-src-json-encoder/bin/multitier/RegDSTTA.config
          subPath: RegDSTTA.config
        command: ["/bin/bash","-c","cd gipsy-src-json-encoder/bin/multitier
        && ./starJSONCONVERTERRegularNode.sh -r"]
        stdin: true
        tty: true
      volumes:
      - name: nfs-vol
        persistentVolumeClaim:
          claimName: nfs-www-pvc
```

By entering the `kubeclt apply -f <name_of_yaml_file>` for both `.yaml`

configuration files that we defined above for the GMT node and the regular node, the pods start running in the cluster. The administrator only needs to enter the `kubectl` command on the Control Plane node, and no other actions are required to register the GIPSY nodes, see Requirement 4.

Once the nodes are registered in the cluster, we can begin by allocating each pod to a particular GIPSY tier (DST, DWT, DGT). `kubectl exec` enables us to establish connections to the containers in the cluster, execute `/bin/sh` inside the gipsy-gmt pod's container, and pass the input and output streams from the terminal to the container's process. Following is the command in order to connect to the GMT pod.

```
kubectl exec -it gipsy-gmt /bin/sh
```

Once the connection has been established with the GMT container, we will be able to allocate each of the regular nodes to a GIPSY tier (DST, DWT or DGT) by specifying the index of the node and Index of the DST and the number of instances that we wish to run. Following are the commands in order to allocate and start the GIPSY tiers.

```
./gmtc.sh allocate NodeIndex DST JiniDST.config           [number of instances to start]
./gmtc.sh allocate NodeIndex DWT JsonConverterDWT.config DSTIndexAtGMT [number of instances to start]
./gmtc.sh allocate NodeIndex DGT JsonConverterDGT.config DSTIndexAtGMT [number of instances to start]
```

As we mentioned above, we have deployment configurations for all pods and shared the `RegDSTTA.config` file for registering any new GIPSY node. Kube-controller-manager, which is responsible for monitoring the status of replica sets, verifies the number of pods available. If a node dies, Kubernetes will create another GIPSY pod on the next available node. Since all the GIPSY node configurations are available for registration, each new GIPSY node running in a pod will automatically get registered, see Requirement 5.

4.3 Summary

This chapter focused on the design and implementation, including classes, their definitions, and corresponding UML diagrams related to our research work. We first discussed the implementation part of our GitHub JSON to FORENSIC LUCID Encoder, then described the implementation part of the Kubernetes and GIPSY integration in more detail. In the next chapter we continue by evaluating and analyzing the requirements described on the Chapter 1.

Chapter 5

Evaluation

In Chapter 3 and Chapter 4, we explained how we managed to implement a GIPSY demand-driven pipeline that allows us to pull evidence-based data from the GitHub repositories and translate it such that the Forensic Lucid can execute it by designing and implementing a JSON demand-driven encoder and distribute its computation by employing GIPSY distributed computing system. In addition, we provide our solution regarding integrating the GIPSY with Kubernetes in order to achieve the requirements as we mentioned in Section 1.2.3.

In this chapter, we follow-up from our evaluation methodology described in Section 3.2 and describe in more detail the evaluation methodology that we employed to demonstrate that our solution actually meets the requirements that we had stated in the first place in Section 1.2.3. In doing so, we provide the results of our evaluation by using specific experiments to provide an objective way to demonstrate that some of the requirements are effectively met.

In our evaluation experimentation, we perform two classes of experiments. At first we conduct data extraction from the GitHub repository and translate them to the Forensic Lucid format using JSON demand-driven encoder and GIPSY distributed computing system without employing the Kubernetes infrastructure. Then we perform the same evaluation experiments by integrating Kubernetes with GIPSYdistributed system. In the following sections, we will discuss these two experiments in more detail.

5.1 Evaluation Environment

Before we dive into the evaluation, we first describe the environment where we perform our experiments. We provide information regarding the CPU, Memory size, GPU, operating system, etc. As we illustrated in the Figure 11, for our experiments, we use three physical machines/nodes. The hardware specifications for all machines are depicted in Table 1. In addition, in Table 2 we describe the tools and applications that we employ in our research, where depending on the role of each machine (Master/Worker), we installed corresponding tools and applications.

Configuration	Component	Specifications
Control Plane Node	Memory	8 GB
	Processor	Intel Core i5-760 CPU @2.80GHz × 4
	Graphics	NVIDIA Quadro FX 580 (512 MB) × 2
	OS	Ubuntu 20.04.3 LTS x86-64
Worker Node 1	Memory	4 GB
	Processor	Intel Core i5-2400 CPU @ 3.10GHz × 4
	Graphics	NVIDIA Quadro 600 (1 GB)
	OS	Ubuntu 18.04.6 LTS x86-64
Worker Node 2	Memory	4 GB
	Processor	Intel Core i5-2400 CPU @ 3.10GHz × 4
	Graphics	NVIDIA Quadro 600 (1 GB)
	OS	Ubuntu 20.04.3 LTS x86-64

Table 1: Hardware environment for the tests.

Name	Version
Docker	v20.10.7
Kubernetes	v1.23.3
NFS	v4

Table 2: Tools and applications versions.

5.2 Evaluation of the GISPY JSON Demand-Driven Encoder

For gathering a dataset, as we mentioned in Section 3.2, we mainly aim for the cybersecurity vulnerabilities that exist publicly and fetch them from the GitHub commit API since it contains various practical information such as commit messages,

comments and patches, which show the changes made to the files. A system called Common Vulnerabilities and Exposures (CVE) offers a way for the public to exchange knowledge about cybersecurity vulnerabilities and exposures [30]. CVE vulnerability data is accessible at www.cvedetails.com.

Below is a sample list of URLs we selected from the CVE website for some popular projects. We extracted the commit URLs from these discovered vulnerabilities and used them as input to run our proposed pipeline to encode as evidence.

- https://github.com/tensorflow/tensorflow/security/advisories/GHSA-79h2-q768-fpxr
- https://github.com/pjsip/pjproject/security/advisories/GHSA-rwgw-vwxg-q799
- https://github.com/pypa/pipenv/security/advisories/GHSA-qc9x-gjcv-465w
- https://github.com/grafana/grafana/security/advisories/GHSA-c3q8-26ph-9g2q
- https://github.com/solidusio/solidus/security/advisories/GHSA-qxmr-qxh6-2cc9
- https://github.com/github/codeql-action/security/advisories/GHSA-g36v-2xff-pv5m
- https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-hpqh-2wqx-7qp5
- https://github.com/netty/netty/security/advisories/GHSA-f256-j965-7f32
- https://github.com/swagger-api/swagger-codegen/security/advisories/GHSA-pc22-3g76-gm6j
- https://github.com/http4s/blaze/security/advisories/GHSA-xmw9-q7x9-j5qc

In order to run the experiment, we first run and register the GMT node in one of the worker machine and in order to distribute the computation, in Worker Node 1 and Worker Node 2 machine, we start and register the regular nodes, to which later we will assign the GIPSY tiers. Figure 22 illustrates that all ten nodes are successfully registered into the GIPSY network.

We continue with allocating the DST to the registered GIPSY nodes, then allocate DWTs. Finally, to start the experiments, allocate the DGT using the commands we illustrate in Listing 4.2.3. Once the DGT starts running, DGT will create a demand for each of the URLs we mentioned above, assign each URL as a demand signature, and put it in the DST. Once the demands are stored in the DWT, the DWTs begin to perform the process that we described in Figure 15.

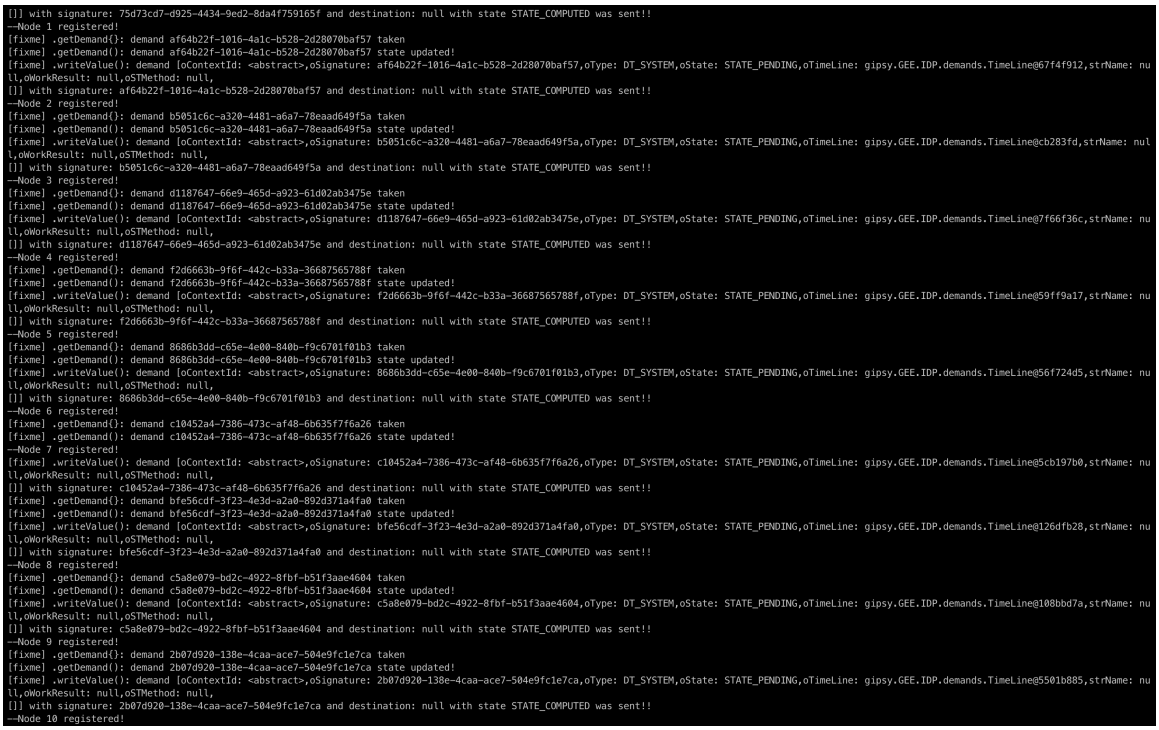

Figure 22: GMT Logs

Once the execution is accomplished, the DGT will show as an output the list of JSON file names, IPL file names and the compiled output for each demand. In addition, we collect our dataset containing 1000 JSONfiles, see Appendix A, 1000 IPL Forensic Lucid files, see Appendix B and corresponding compiled Forensic Lucid files.

Once the results are accomplished, DGT and DWTs wait for new demands to be processed, and DST holds the previous demand and its results. Therefore, we continue experiments by providing the same URLs to generate the demands as we did previously. Since we use the URLs as a demand signature and DST stored it already, the DGT merely returns the results.

This essentially constitutes the requirements as described in Section 3.2:

- **Requirement 1: The system shall have JSON demand-driven encoder**
- **Requirement 2: The system shall have employ GPSY distributed computing system for the forensic computing pipeline**

- **Requirement 3: The system shall return the results quickly if we execute the JSON encoder for the second attempt on the same data**

We follow our experiments by finding a relation between the number of workers (DWTs) and the number of URLs. We vary the number of DWTs and perform the same experiments as we did above for the same URLs.

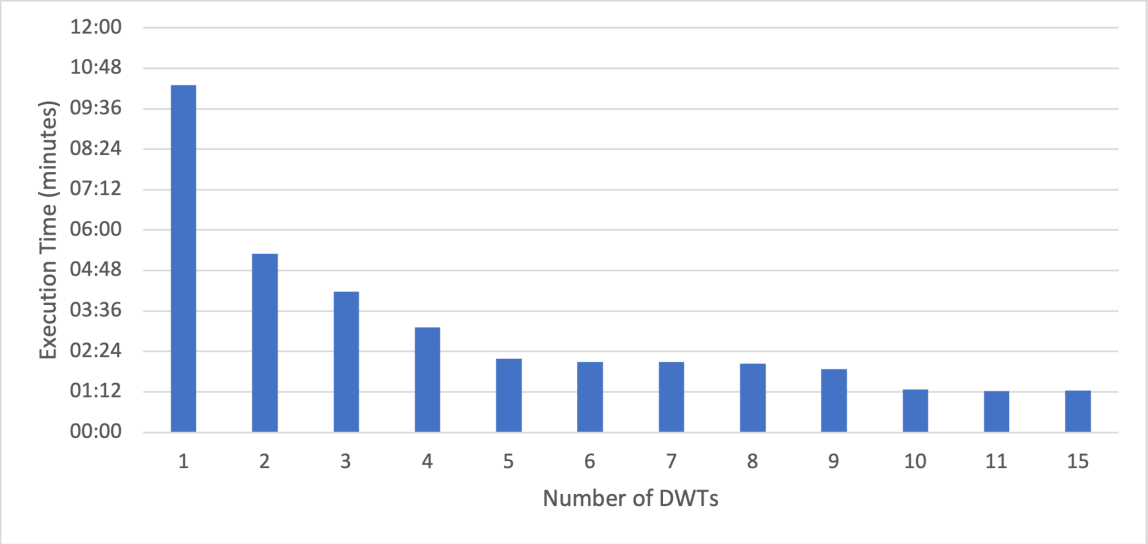

Figure 23: Execution time depending on the number of DWTs (workers)

Figure 23 depicts the total execution time for each amount of workers. As we can observe in the figure, the performance significantly improves once we employ more than five DWTs. However, the execution time remains constant once there are ten workers, which is equal to the number of URLs.

5.3 Evaluation of Integration of Kubernetes in the GIPSY Distributed Execution System

In this section, we attempt to perform the same experiments we conducted in Section 5.2, except this time, we attempt to achieve the same results by integrating the distributed processing orchestration features of Kubernetes. Before we begin to perfom the integration, we begin with deploying the GIPSY network without Kubernetes infrastructure and estimate the time it takes to configure and register

the GIPSY GMT and regular nodes (ten nodes in this test).

```
NAME      STATUS   ROLES                  AGE    VERSION   INTERNAL-IP
h-lai     Ready    <none>                 12d    v1.23.4   192.168.132.74
kmaster   Ready    control-plane,master   162d   v1.23.3   132.205.40.48
knode     Ready    <none>                 162d   v1.23.3   192.168.132.73
```

Figure 24: Kubernetes Nodes

Figure 24 illustrates three nodes registered in Kubernetes, consisting of one Control Plane node and two Workers nodes, where all the administration will happen in the Control Plane node, and our GIPSY pods will run on two Worker nodes.

Now we begin with registering GIPSY pods in the Kubernetes cluster by applying the `.yaml` configuration files we provide in Section 4.2.3. By defining the number of replicas in the `.yaml` configuration file, which in this experiment is ten, we specify the number of regular pods we wish to register. Once we apply the configuration file using the Kubernetes command line, all the GIPSY nodes inside each pod will automatically get registered.

Since we use the NFS in order to share the required configuration files, which are produced by the GMT pod, all the regular pods have access to that configuration file in order to get registered to GIPSY network.

```
NAME                          READY   STATUS    RESTARTS   AGE   IP               NODE
gipsy-dst-5b6c49bc5f-5n2k8    2/2     Running   0          14m   172.16.177.227   knode
gipsy-dst-5b6c49bc5f-8qjhk    2/2     Running   0          14m   172.16.177.238   knode
gipsy-dst-5b6c49bc5f-9j86h    2/2     Running   0          14m   172.16.81.61     h-lai
gipsy-dst-5b6c49bc5f-bfwqv    2/2     Running   0          14m   172.16.81.60     h-lai
gipsy-dst-5b6c49bc5f-mxql4    2/2     Running   0          14m   172.16.81.62     h-lai
gipsy-dst-5b6c49bc5f-nhbqz    2/2     Running   0          14m   172.16.177.236   knode
gipsy-dst-5b6c49bc5f-rlqjk    2/2     Running   0          14m   172.16.81.54     h-lai
gipsy-dst-5b6c49bc5f-vd7bg    2/2     Running   0          14m   172.16.177.245   knode
gipsy-dst-5b6c49bc5f-w2vwp    2/2     Running   0          14m   172.16.81.56     h-lai
gipsy-dst-5b6c49bc5f-xd27z    2/2     Running   0          14m   172.16.177.235   knode
gipsy-gmt                     3/3     Running   0          15m   172.16.81.55     h-lai
```

Figure 25: GIPSY nodes registration using Kubernetes infrastructure

Figure 25 illustrates that all the GIPSY pods are up and running, and the GIPSY nodes inside each GIPSY pod are registered successfully to the GIPSY network. It also demonstrates how Kubernetes distribute the GIPSY pods depending on the resource availability of the machines. If we take a look at the logs of the `gipsy-gmt` pod, we see the same output as in Figure 22.

Table 3 depicts the comparison of the estimated time for registration of ten

Type	Time
without integrating k8s	5 minutes
with integrating k8s	50 seconds

Table 3: Estimated time for configuring and registring 10 GIPSYnodes

GIPSY nodes. As we can observe from these results, registering GIPSY nodes using Kubernetes resulted in dramatic reduction of deployment time. However, in this estimation, we did not count the compilation time, which for each time would take approximately 9 minutes, which is significantly time-consuming considering if we would like to run multiple nodes.

Once GIPSY nodes inside the pods registered to the GIPSY network, we followed experiments with allocating the DGT, DWTs and DST the same as we did in Section 5.2. Except we need to run the allocation commands inside the GMT pod as we mentioned in Listing 4.2.3. Eventually, when the DWTs finish the execution, the collected dataset is the same as Section 5.2, which can be verified in the NFS shared directories.

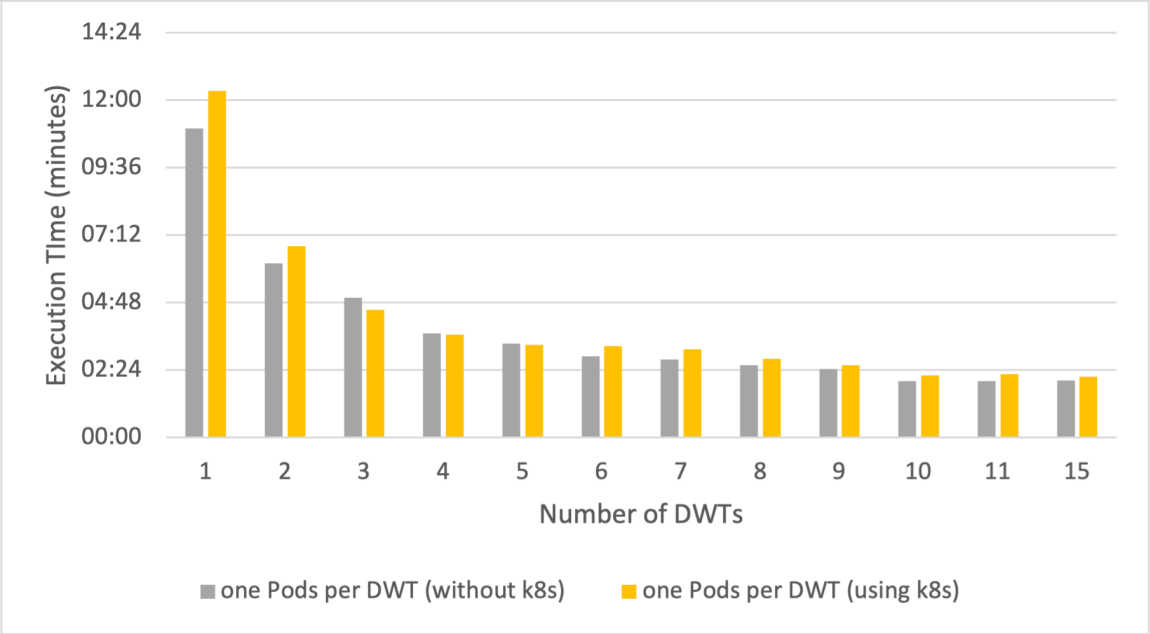

Figure 26: Execution time Comparison

In Figure 26, we compare our experiments with and without Kubernetes infrastructure. Same as Section 5.2 we perform testing for different numbers of DWT

pods to estimate the execution's performance depending on the number of GIPSY pods.

Type	Fetching	Converting	Compiling	Total
without integrating k8s	1m 21s	6.5s	0.8s	1m 31s
with integrating k8s	1m 21s	6.5s	0.7s	1m 31s

Table 4: Comparison the Execution Time of Fetching, Converting and Compiling (Minute:Second)

As we can observe, the performance of Kubernetes remains almost the same . However, the testing results depend on the fetching, converting and compiling time can vary. Therefore, we came up with the following experiment in order to estimate the execution time for fetching, converting and compiling separately. Table 4 demonstrates the average estimation of each step in our experiment.

```
NAME      STATUS     ROLES                  AGE    VERSION
h-lai     NotReady   <none>                 12d    v1.23.4
kmaster   Ready      control-plane,master   163d   v1.23.3
knode     Ready      <none>                 163d   v1.23.3
```

Figure 27: Kubernetes nodes status after swtiching off a node

Finally, for our last experiment, we assume that all GIPSY pods, as described in Figure 25 are up and running. In order to demonstrate the scalable fault tolerance, we switch off one of the worker nodes in order to verify how Kubernetes manage the GIPSY pods that were running on the dead node. Figure 27 depicts the Kubernetes nodes' status after switching a worker node, which turned to `NotReady`.

Once the Kubernetes node is dead, after approximately 5 minutes, it will change the status of the running pods on the dead node to `Terminating`. The reason is that if the node can restart in 5 minutes, the pods will be able to resume working. In this case, this time pass, and consequently, Kubernetes starts to create the same amount of pods on the next available node, as it is illustrated in Figure 28.

In Figure 29, we describe that once the pods got re-created, the GIPSY nodes automatically get registered in the GIPSY network.

With what we described, we accomplished the requirements as mentioned in Section 3.2:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
gipsy-dst-5b6c49bc5f-6x22f	2/2	Running	0	16m	172.16.177.234	knode
gipsy-dst-5b6c49bc5f-7v7bq	2/2	Terminating	0	16m	172.16.81.58	h-lai
gipsy-dst-5b6c49bc5f-9j8dl	2/2	Running	0	16m	172.16.177.228	knode
gipsy-dst-5b6c49bc5f-9mbt4	2/2	Running	0	16m	172.16.177.237	knode
gipsy-dst-5b6c49bc5f-fxgwr	2/2	Running	0	103s	172.16.177.248	knode
gipsy-dst-5b6c49bc5f-h7dn7	2/2	Terminating	0	16m	172.16.81.59	h-lai
gipsy-dst-5b6c49bc5f-jn9z4	2/2	Running	0	103s	172.16.177.239	knode
gipsy-dst-5b6c49bc5f-k5w9g	2/2	Running	0	16m	172.16.177.247	knode
gipsy-dst-5b6c49bc5f-q6sgq	2/2	Running	0	103s	172.16.177.241	knode
gipsy-dst-5b6c49bc5f-rnwrm	2/2	Running	0	103s	172.16.177.249	knode
gipsy-dst-5b6c49bc5f-sg2b9	2/2	Running	0	103s	172.16.177.244	knode
gipsy-dst-5b6c49bc5f-wdjj5	2/2	Running	0	16m	172.16.177.250	knode
gipsy-dst-5b6c49bc5f-wlkw6	2/2	Terminating	0	16m	172.16.81.2	h-lai
gipsy-dst-5b6c49bc5f-z7kpm	2/2	Terminating	0	16m	172.16.81.53	h-lai
gipsy-dst-5b6c49bc5f-zpbjk	2/2	Terminating	0	16m	172.16.81.57	h-lai
gipsy-gmt	3/3	Running	0	17m	172.16.177.233	knode

Figure 28: Pods status after swtiching off a node

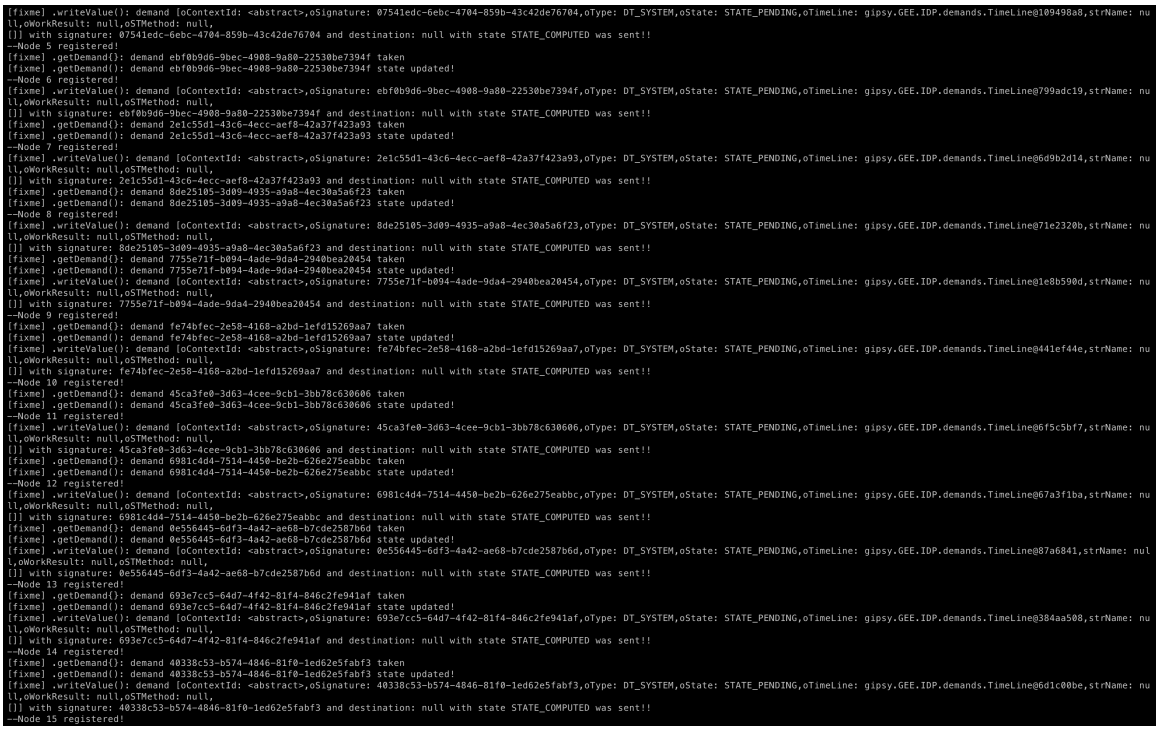

Figure 29: Automation of GIPSY nodes registration

- **Requirement 4: The system shall have an efficiently deployable GIPSY distributed system**

- **Requirement 5: The system shall have a scalable fault tolerance mechanism**

- **Requirement 6: The system shall have a shared file system**

5.4 Summary

In this chapter, we described various evaluation results of our solution to the problem stated in Chapter 1. In doing so, we have demonstrated that we have effectively fulfilled all the requirements that we mentioned in Section 1.2.3. In the next chapter, we will continue with our conclusion, describe the limitations we faced through our research and discuss future work.

Chapter 6

Conclusion and Future Work

In Chapter 1, we described the problem and provided the related motivation scenarios in Section 1.2.2 and described requirements in Section 1.2.3. In Chapter 2 we followed by representing the background and described the tools and methods that we used in this research work. Then in Chapter 3 we presented to the reader our solution regarding GitHub demand-driven JSON to Forensic Lucid Encoder Formalization in GIPSY and provided our solution related to integrating the Kubernetes cluster with GIPSY distributed computing system, and we provided the methods that we employed in our research to evaluate and verify if our solution solves the problems. Next, in Chapter 4 we described our solution's design and implementation. Then, in Chapter 5, we evaluated the fulfilment of the requirements by conducting various experiments. Eventually, in this chapter we follow by describing our conclusion for our research in Section 6.1 and representing some limitations and several portions of the work that we deliver for the future in Section 6.2.

6.1 Conclusion

In this section, we conclude our research based on the conducted experiments and the results that we achieved in Chapter 5. Below we address the fulfillment of the problems/requirements that we stated in Chapter 1.

- In our research, we devised a solution so that forensic investigators could use

GitHub to use detected vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE), which is a list of publicly disclosed computer security flaws, and gather a dataset in order to perform an investigation on program weaknesses and vulnerabilities related to security, software engineering from GitHub projects written in various programming languages. We designed and implemented JSON demand-driven encoder and we defined our classes to perform the FORENSIC LUCID conversion pipeline (data extraction, converting to FORENSIC LUCID format, and compiling the FORENSIC LUCID files). In order to distribute the execution, we took advantage of the GIPSY distributed system. Therefore we defined the required classes for distributing the pipeline execution of the JSON demand-driven encoder using the GIPSY distributed computing system, see Requirement 1.

- In the FORENSIC LUCID conversion pipeline, we defined each URLS as a demand signature to store it in the DST, which stays the same throughout the whole process of the pipeline, and once the conversion is finished, the demand signature alongside the results will be stored in the DST. By employing this approach, the pipeline does not require execution for the same demand in the DST. Therefore, if the forensic investigator requests a demand that already exists in the DST, the output would be fast since the demand results have already been stored in the DST after the first execution, see Requirement 3.

- By integrating the distributed processing orchestration features of Kubernetes with GIPSY, we improved the GIPSY in such a way that configuring, starting up and registering GIPSY nodes would happen automatically without any manual configurations. We also described that the execution time of the JSON demand-driven encoder using Kubernetes is slightly more. However, by employing the Kubernetes there would be no need for the compilation each time and installing all dependencies in order to start the GIPSY, which saves a significant amount of time, see Requirement 4.

- By integrating the Kubernetes, if a GIPSY node dies, all the pods will be

recreated on the next available machine and automatically will get registered to the GIPSY network. Therefore, there would be no need to reconfigure and startup everything manually. Thus we were able to have a scalable fault-tolerant system, see Requirement 5.

- We were able to share the directories for the initial configuration and the dataset files among the pods, by employing NFS so that each pod has access to the same directory, see Requirement 6.

6.2 Limitations and Future Work

At the moment, despite the fact that we achieved all the requirements and were able to provide solutions for our stated problem, we faced some limitations, which require work in the near future. Some of the limitations and future works are listed below:

- At the moment, we were able to collect a FORENSIC LUCID dataset from the GitHub repositories in order to conduct future investigations. However, we did not provide any hypothesis to analyze the evidential data. This can be done in future work to conduct forensic analysis and attempt to prove a hypothesis.

- In order to perform the data extraction for the GitHub API, there is a 5,000 request per hour limit for the authorized user by a user or a personal token. Therefore, our evaluation experiments were limited to not a significant amount of data to fetch. In our experiments, we gathered 1000 data element, which requires 1000 requests for each time execution. At the moment, the system has already been implemented so that once it reaches the limit of 5000 requests, it will wait until the limit rests and resume the fetching. In order to have a more accurate analysis would be better to fetch a much more important amount of data.

- In our research, we only conducted the data extraction from GitHub repositories. We did not attempt to perform the same computation for the other

open resources such as BitBucket, or other sources such as social media, e.g., Twitter, etc. It would be interesting to have the same conversion pipeline for other resources that could then be used as evidential statements to be processed by FORENSIC LUCID to prove or disprove much more diversified forensic cases.

- There are various container orchestration tools, such as OpenShift, Docker Swarm, etc., to integrate with GIPSY, which we did not attempt to employ.

- Although we were able to design and implement a system that GIPSY node can automatically get configured and startup, we need to allocate the GIPSY tiers manually. Since in order to allocate a GIPSY node, we need to specify the node index as we described in Listing 4.2.3, and the index varies for each node registration, at the moment, it is not possible to allocate them automatically. In addition, in case a node dies, we have a scalable fault-tolerant mechanism, which will automatically configure and register the GIPSY nodes. However, as we mentioned, the allocation in this scenario should happen manually by defining the index of each registered GIPSY node inside the newly recreated pod. The next step is finding a solution to perform the tier allocation process automatically.

We published our project to the Docker Hub repository as a set of Docker images, which can be found in: https://hub.docker.com/r/s4lab/gipsy-json-u18/tags. In the links below, we are releasing GitHub repositories, to which our work will be published in the future soon.

- S4L GIPSY Research and Development: https://github.com/gipsy-dev
- OpenTDIP: https://github.com/opentdip

6.3 Summary

In this chapter, we concluded our research by providing the list of achieved experiments and the results that we conducted in this research in Section 6.1. Finally,

we provided the limitation and future work for this research in Section 6.2. In addition, we submitted for publication a part of our research work for the 15th International Symposium on Foundations & Practice of Security (FPS2022) [47].

Bibliography

[1] S. A. Mokhov, M. Song, J. Singh, J. Paquet, M. Debbabi, and S. Mudur, "Toward multimodal interaction in scalable visual digital evidence visualization using computer vision techniques and ISS," in *Proceedings of the International Conference on Pattern Recognition and Artificial Intelligence (ICPRAI)*. CENPARMI, Concordia University, Montreal, May 2018. ISBN 978-1-895193-04-6 pp. 151–157, https://users.encs.concordia.ca/~icprai18/, arXiv:1808.00118.

[2] M. A. Baker and R. Buyya, "Cluster computing at a glance," 1999.

[3] B. Han, S. A. Mokhov, and J. Paquet, "Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java," in *Proceedings of the 8th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*. IEEE Computer Society, May 2010. doi: 10.1109/SERA.2010.40. ISBN 978-0-7695-4075-7 pp. 259–266, online at http://arxiv.org/abs/0906.4837.

[4] S. A. Mokhov, "Intensional cyberforensics," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2013, online at http://arxiv.org/abs/1312.0466.

[5] Docker, Inc., "Docker," [online; accessed 27-May-2019], 2019, https://docs.docker.com/get-started/overview/.

[6] Kubernetes, "Kubernetes documentation," https://kubernetes.io/docs/home/, last viewed May 2022, 2022.

[7] P. Derafshkavian, S. Huneault-LeBlanc, S. Renault-Crispo, A. Marwaha, S. A. Mokhov, and J. Paquet, "Toward scalable demand-driven json-to-forensic lucid encoder in gipsy," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, 2020. doi: 10.1109/ISNCC49221.2020.9297224 pp. 1–6.

[8] S. A. Mokhov, J. Paquet, and M. Debbabi, "Formally specifying operational semantics and language constructs of Forensic Lucid," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, ser. LNI, O. Göbel, S. Frings, D. Günther, J. Nedon, and D. Schadt, Eds., vol. 140. GI, Sep. 2008. ISBN 978-3-88579-234-5. ISSN 1617-5468 pp. 197–216, online at http://subs.emis.de/LNI/Proceedings/Proceedings140/gi-proc-140-014.pdf.

[9] J. Paquet and A. H. Wu, "GIPSY – a platform for the investigation on intensional programming languages," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 8–14.

[10] J. Singh, "Universal gesture tracking framework in OpenISS and ros and its applications," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2020, https://spectrum.library.concordia.ca/id/eprint/986429/. [accessed 20-October-2022].

[11] M.-A. Laverdière, S. A. Mokhov, S. Tsapa, and D. Benredjem, "Ftklipse–design and implementation of an extendable computer forensics environment: Software requirements specification document," 2005–2009, http://arxiv.org/abs/0906.2446.

[12] ——, "Ftklipse–design and implementation of an extendable computer forensics environment: Specification design document," 2005–2009, http://arxiv.org/abs/0906.2447.

[13] S. A. Mokhov, J. Paquet, and M. Debbabi, "Towards formal requirements specification of self-forensics for autonomous systems," 2014, in preparation for

submission to ACM Transactions on Software Engineering and Methodology (TOSEM).

[14] S. A. Mokhov, E. Vassev, J. Paquet, and M. Debbabi, "Towards a self-forensics property in the ASSL toolset," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering (C3S2E'10)*. New York, NY, USA: ACM, May 2010. doi: 10.1145/1822327.1822342. ISBN 978-1-60558-901-5 pp. 108–113.

[15] W. G. Kruse and J. G. Heiser, *Computer Forensics: Incident Response Essentials*. Addison-Wesley Professional, 2001, ISBN: 9780672334085, 0672334089.

[16] R. Buyya, H. Jin, and T. Cortes, "Cluster computing," *Future Generation Computer Systems*, vol. 18, no. 3, pp. v–viii, 2002. doi: https://doi.org/10.1016/S0167-739X(01)00053-X Cluster Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X0100053X

[17] C. S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and F. Sommers, *Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers*. Boston, MA: Springer US, 2006, pp. 521–551. ISBN 978-0-387-27705-9. [Online]. Available: https://doi.org/10.1007/0-387-27705-6_16

[18] S. A. Mokhov, "Encoding forensic multimedia evidence from MARF applications as Forensic Lucid expressions," in *Novel Algorithms and Techniques in Telecommunications and Networking, proceedings of CISSE'08*, T. Sobh, K. Elleithy, and A. Mahmood, Eds. University of Bridgeport, CT, USA: Springer, Dec. 2008. doi: 10.1007/978-90-481-3662-9_71. ISBN 978-90-481-3661-2 pp. 413–416, printed in January 2010.

[19] S. A. Mokhov and J. Paquet, "Formally specifying and proving operational aspects of Forensic Lucid in Isabelle," Department of Electrical and Computer

Engineering, Concordia University, Montreal, Canada, Tech. Rep. 2008-1-Ait Mohamed, Aug. 2008, in Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings. Online at: http://users.encs.concordia.ca/~tphols08/TPHOLs2008/ET/76-98.pdf and http://arxiv.org/abs/0904.3789.

[20] S. A. Mokhov, J. Paquet, and M. Debbabi, "The need to support of data flow graph visualization of Forensic Lucid programs, forensic evidence, and their evaluation by GIPSY," [online], Sep. 2010, poster at VizSec'10; online at http://arxiv.org/abs/1009.5423.

[21] ——, "Towards automatic deduction and event reconstruction using Forensic Lucid and probabilities to encode the IDS evidence," in *Proceedings of Recent Advances in Intrusion Detection RAID'10*, ser. Lecture Notes in Computer Science (LNCS), S. Jha, R. Sommer, and C. Kreibich, Eds., vol. 6307. Springer Berlin Heidelberg, Sep. 2010, Poster. doi: 10.1007/978-3-642-15512-3_36. ISBN 978-3-642-15511-6 pp. 508–509.

[22] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming*. London: Oxford University Press, Feb. 1995, ISBN: 978-0195075977.

[23] E. A. Ashcroft and W. W. Wadge, "Lucid – a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 5, no. 3, 1976.

[24] ——, "Erratum: Lucid – a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 6, no. 1, p. 200, 1977.

[25] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge, "Sequential demand-driven evaluation of eager TransLucid," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008. doi: 10.1109/COMPSAC.2008.191. ISSN 0730-3157 pp. 1266–1271.

[26] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[27] S. A. Mokhov, "Toward automated MAC spoofer investigations," Seminar, Mar. 2014, cIISE Security, Privacy, and Forensics (SPF).

[28] S. A. Mokhov and J. Paquet, "Using the general intensional programming system (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions," *CoRR*, vol. abs/0906.3911, 2009. [Online]. Available: http://arxiv.org/abs/0906.3911

[29] S. A. Mokhov, "Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005, ISBN 0494102934; online at http://arxiv.org/abs/0907.2640.

[30] S. A. Mokhov, J. Paquet, M. Debbabi, and Y. Sun, "MARFCAT: Transitioning to binary and larger data sets of SATE IV," [online], May 2012–2014, online at http://arxiv.org/abs/1207.3718.

[31] J. Paquet, "Distributed eductive execution of hybrid intensional programs," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*. IEEE Computer Society, Jul. 2009. doi: 10.1109/COMPSAC.2009.137. ISBN 978-0-7695-3726-9. ISSN 0730-3157 pp. 218–224.

[32] S. A. Mokhov and J. Paquet, "Using the General Intensional Programming System (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions," in *Proceedings of the 8th IEEE / ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*. IEEE Computer Society, May 2010. doi: 10.1109/SERA.2010.23. ISBN 978-0-7695-4075-7 pp. 101–109, pre-print at http://arxiv.org/abs/0906.3911.

[33] D. S. Sidhu, H. Singh, S. Kaur, and T. Paul, "Cybercrime investigations Project FLUCIDGIPSY3: Initial prototype of the GitHub and Twitter API JSON-to-Forensic Lucid encoders," INSE 6610 Cybercrime Investigations 2019, Serguei Mokhov, 2018, course project.

[34] A. H. Pourteymour, "Comparative study of Demand Migration Framework implementation using JMS and Jini," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2008, http://spectrum.library.concordia.ca/975918/.

[35] E. I. Vassev, "General architecture for demand migration in the GIPSY demand-driven execution engine," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Jun. 2005.

[36] A. H. Pourteymour, E. Vassev, and J. Paquet, "Towards a new demand-driven message-oriented middleware in GIPSY," in *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA 2007*, vol. 1, PDPTA. CSREA Press, Jun. 2007. ISBN 1-60132-020-5 pp. 91–97.

[37] ——, "Design and implementation of demand migration systems in GIPSY," in *Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'08)*. CSREA Press, Jul. 2008. ISBN 1-60132-084-1 pp. 900–907.

[38] E. Vassev and J. Paquet, "A generic framework for migrating demands in the GIPSY's demand-driven execution engine," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 29–35.

[39] J. Paquet and P. G. Kropf, "The GIPSY architecture," in *Proceedings of Distributed Computing on the Web*, ser. Lecture Notes in Computer Science, P. G. Kropf, G. Babin, J. Plaice, and H. Unger, Eds., vol. 1830. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-45111-0_17 pp. 144–153.

[40] L. Tao, "Intensional value warehouse and garbage collection in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004, http://spectrum.library.concordia.ca/8129/.

[41] B. Burns, J. Beda, and K. Hightower, *Kubernetes Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, 2019, ISBN: 9781492046530.

[42] docker, "Container," https://www.docker.com/resources/what-container/.

[43] wmware, "Container orchestration," https://www.vmware.com/topics/glossary/content/container-orchestration.html#:~:text=Container%20orchestration%20is%20the%20automation,networking%2C%20load%20balancing%20and%20more.

[44] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," Jan 2019, https://doi.org/10.48550/arXiv.1901.04946.

[45] S. Verreydt, E. Truyen, E. H. Beni, and B. Lagaisse, "Leveraging kubernetes for adaptive and cost-efficient resource management," October 2019.

[46] G. Docs, "Github rest apis," https://docs.github.com/en/rest, last viewed May 2022, 2022.

[47] P. Zahraei, S. Mokhov, J. Paquet, and P. Derafshkavian, "A Kubernetes underlay for OpenTDIP forensic computing backend," Submitted for review to the 15th International Symposium on Foundations and Practice of Security (FPS'2022), 2022.

Appendix A

Example JSON File


```
[{
    "sha": "fc9ac5879be3cd93bb04f95341dbf4d165743b01",
    "node_id": "C_kwDOArmXAtoAKGZjOWFjNTg3OWJlM2NkOTNiYjA0Zjk1MzQxZGJmNGQxNjU3NDNiMDE",
    "commit": {
        "tree": {"sha": "b4b9610692287a800788946535e048e16332e9a4","url": "https://api.github.com...
        "comment_count": 0,
        "verification": {"verified": false,"reason": "unsigned","signature": null,"payload": null}},
    "url": "https://api.github.com/repos/tensorflow/tensorflow/commits/fc9ac5879be3cd93bb04f9...
    "html_url": "https://github.com/tensorflow/tensorflow/commits/fc9ac5879be3cd93bb04f95341db...
    "comments_url": "https://api.github.com/repos/tensorflow/tensorflow/commits/fc9ac5879be3cd...
    "author": {"login": "cky9301","id": 8674768,"node_id": "MDQ6VXNlcjg2NzQ3Njg=","avatar_url":...
    "committer": {"login": "tensorflower-gardener","id": 17151892,"node_id": "MDQ6VXNlcjE3MTUx...
    "parents": [{"sha": "61c4c19fa8a1c8125a85f0dbeba0d267be1eb6cb","url": "https://api.github...
    "stats": {"total": 17,"additions": 8,"deletions": 9},
    "files": [
        {"sha": "11eeb05498a09d9e25b535203c06ac981917b883","filename": "tensorflow/...
        {"sha": "cff040345e3ccbab1931aa31f87ccb6102d6fbc6","filename": "tensorflow/...
},
{
    "sha": "fb568f9e292d853e6dfcd426f8047288a040506e",
    "node_id": "C_kwDOArmXAtoAKGZiNTY4ZjllMjkyZDg1M2U2ZGZjZDQyNmY4MDQ3Mjg4YTA0MDUwNmU",
    "commit": {
        "tree": {"sha": "4cc1395d1b1914b63600462184030a900dd002a3","url": "https://api.github.com...
        "comment_count": 0,
        "verification": {"verified": true,"reason": "valid","signature": "-----BEGIN PGP SIGNATUR...
    "url": "https://api.github.com/repos/tensorflow/tensorflow/commits/fb568f9e292d853e6dfcd426f8...
    "html_url": "https://github.com/tensorflow/tensorflow/commits/fb568f9e292d853e6dfcd426f8047288...
    "comments_url": "https://api.github.com/repos/tensorflow/tensorflow/commits/fb568f9e292d853e6...
    "author": {    "login": "Sadeedpv",    "id": 96517901,    "node_id": "U_kgDOBcC_DQ",    "avat...
```

```
    "committer": {    "login": "web-flow",    "id": 19864447,    "node_id": "MDQ6VXNlcjE5ODY0NDQ3...
    "parents": [    {      "sha": "94e3230e762b2541a7e4274e49b3a65d3826ead3",      "url": "https:...
    "stats": {    "total": 4,    "additions": 2,    "deletions": 2  },
    "files": [    {      "sha": "5fd486276d9a4bcf319a0c2108f657311849fa07",      "filename": "SEC...
}]
```

Listing A.1: Fetched JSON file from GitHub repository

Appendix B

Example Forensic Lucid File

```
OResult
where
evidential statement es = {o_sequence};

        observation sequence o_sequence = {o_commit_fcb01, o_commit_fb6e};
        observation o_commit_fcb01 = {commit, author, committer, parents, stats, files};
        commit = {[tree : {[sha : "b4b9610692287a800788946535e048e16332e9a4"], [url: "https://...
        author = {[login : "cky9301"], [id : 8674768], [node_id : "MDQ6VXNlcjg2NzQ3Njg="], [av...
        committer = {[login : "tensorflower-gardener"], [id : 17151892], [node_id : "MDQ6VXNlc...
        parents = {[[sha : "61c4c19fa8a1c8125a85f0dbeba0d267be1eb6cb"], [url : "https://api.gi...
        stats = {[total : 17], [additions : 8], [deletions : 9]};
        files = {[[sha : "11eeb05498a09d9e25b535203c06ac981917b883"], [filename : "tensorflow/...

        observation o_commit_fb6e = {commit, author, committer, parents, stats, files};
        commit = {[tree : {[sha : "4cc1395d1b1914b63600462184030a900dd002a3"], [url: "https://...
        author = {[login : "Sadeedpv"], [id : 96517901], [node_id : "U_kgDOBcC_DQ"], [avatar_...
        committer = {[login : "web-flow"], [id : 19864447], [node_id : "MDQ6VXNlcjE5ODY0NDQ3"...
        parents = {[[sha : "94e3230e762b2541a7e4274e49b3a65d3826ead3"], [url : "https://api.g...
        stats = {[total : 4], [additions : 2], [deletions : 2]};
        files = {[[sha : "5fd486276d9a4bcf319a0c2108f657311849fa07"], [filename : "SECURITY.m...
        OResult = es;
end
```

Listing B.1: Converted Forensic Lucid File