A Kubernetes Underlay for OpenTDIP Forensic Computing Backend

Pouria Zahraei
Computer Science and Software Engineering
Concordia University
Montreal, Canada
s_ahrae@cse.concordia.ca

Serguei Mokhov
Computer Science and Software Engineering
Concordia University
Montreal, Canada
mokhov@cse.concordia.ca

Joey Paquet
Computer Science and Software Engineering
Concordia University
Montreal, Canada
paquet@cse.concordia.ca

Peyman Derafshkavian
Computer Science and Software Engineering
Concordia University
Montreal, Canada
p_derafs@cse.concordia.ca

***Abstract*—We propose an underlay based on Kubernetes to enable 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.**

***Index Terms*—OpenTDIP, Kubernetes, FORENSIC LUCID, GIPSY, JSON, encoding, digital evidence**

I. INTRODUCTION

By definition, forensics is the practice of scientists analyzing evidence to help law enforcement investigators to solve crimes by reconstructing events.

As a method of conducting digital criminal investigations, digital forensics uses data preserved on various computer devices as examination evidence that may be utilized in a legal proceeding [1].

The analysis of digital evidence that can be used in court to determine legal issues is a sub-field of forensics called forensic computing, which relies a computer and specialized software to extract and assess evidence that can be found on any kind of storage or electronic device. It is sometimes necessary to gather a very large volume of data, filter it using elaborated techniques appropriate to each different kind of data, and then analyze it in order to produce a piece of appropriate proof that will allow us to learn more specific details about illegal behaviour presentable in court. Computer forensics includes preserving, identifying, extracting, documenting, and interpreting computer data [2], [3].

In forensic computing, many methods are used, but the fundamental process entails obtaining evidence from any source, authenticating the evidence, and evaluating the evidence [1], [3]. Digital forensic investigation is particularly required when the volume of data is too large or complex to gather and when the proof is too complex to manage manually.

FORENSIC LUCID [1] is a forensic case specification language for automatic evidence composition used to formally represent the knowledge of and reason about digital crime incidents. It is designed to describe forensic evidence in digital crime incidents as a context of evaluations, which they often match the original condition of the situation in relation to what was actually observed [1]. Its evaluation engine is based on outcome backtracing and, if a backtrace is found, to offer the path for the related event reconstruction. The results of the FORENSIC LUCID expressions are true or false, i.e., "guilty" or "not guilty", which can be done by one or multiple backtraces [1]. FORENSIC LUCID is based on LUCID, which is an intensional dataflow programming language [4], [5], [6], [7], [8].

GIPSY is a multi-language programming compiler and execution engine framework for compiling all types of Lucid dialects, including FORENSIC LUCID [9]. Using GIPSY, FORENSIC LUCID programs can be compiled and executed as a distributed system using a demand-driven dataflow model. While GIPSY's multi-tier architecture provides a base implementation of the underlying distributed evaluation and computation of FORENSIC LUCID expression, its setup is rather manual to configure and operate its nodes and it lacks automated fault tolerance if nodes go out of service.

In this paper, we show how a FORENSIC LUCID dataset can be collected for a followup computation in a forensic investigation with evidence from GitHub repositories. We also demonstrate how the execution engine can integrate the use of the Kubernetes distributes processing orchestration features such as automatic scaling, and automatic availability management as an underlay of the GIPSY's distributed runtime system to address the mentioned limitations while being transparent to GIPSY. We use it for both the dataset collection as well as subsequent translation, and compilation [10].

Simply presented, the forensic processing pipeline that we implement consists of the following steps:

- The forensic investigator is primarily focused on a particular software implementation vulnerability observes a

list of GitHub repositories or projects that include code related to the target vulnerability.
- For further analysis, every piece of a potentially essential data item in the repository is retrieved in JSON format.
- These "witness statements" (e.g., commits) are the encoded as part of a FORENSIC LUCID observation sequence [1].
- The collection of "testimonies" (e.g., commit history) is then encoded and specified as an evidential statement, which is then formed into the local knowledge base for the forensic case [1].
- Finally, the acquired FORENSIC LUCID dataset is validated through their compilation using GIPSY's compiler. Using GIPSY's evaluating system, one can then check that theories agree with the evidence, meaning the theory has an explanation for the evidential context and possibly explains the chain of events that led to a security incident [1] and provide the event reconstruction sequence of indicate the absence of one.

The objective of this work is, therefore, to facilitate the forensic investigator's task to collect and examine the digital evidence, e.g., from GitHub repositories, for forensic investigation by encoding it in FORENSIC LUCID programming language at large. In order to accomplish this, we design and implement General Intensional Programming System's (GIPSY) runtime compute network node instances by using Kubernetes container orchestration to have an automatically highly-available underlying system. This is done in the context of the big picture of the Open Trusted Digital Investigation Platform (OpenTDIP), which is poised to cover all aspects of digital forensic computing from evidence management, chain of custody, formal methods, and event reconstruction, among others.

II. BACKGROUND

A. FORENSIC LUCID

Following the definition of FORENSIC LUCID in Section I, in this section, we describe what the reader needs to understand from the perspective of this paper. The reader is already familiar with some of the essential features of FORENSIC LUCID, such as the fact that it is a Lucid-based application that represents a tagged token dataflow program [1]. Before delving further into FORENSIC LUCID's architecture, we present two fundamental concepts employed 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 [1].

$$os = (observation_1, observation_2, observation_3, ...) \quad (1)$$

Second, an **evidential statement** is a set of observation sequences, which is not necessarily arranged chronologically [1].

$$es = (os_1, os_2, os_3, ...) \quad (2)$$

Below is an example of a typical patter specification for a FORENSIC LUCID program. After locating forensically interesting data, this program, for instance, will be able to perform a semantic and syntactic check by the GIPSY's compiler, and in doing so it can assist the forensic investigator in scripting the forensics events using an autonomous forensic computing system.

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

Fig. 1. Simplified FORENSIC LUCID digital evidence context specification.

B. General Intensional Programming System (GIPSY)

GIPSY is a multi-language programming platform created to compile and run all the languages in the Lucid family of computer languages dialects. The compiler framework (GIPC), the execution engine (GEE), and the programming environment (RIPE) make up the basis of the GIPSY's architecture. These three elements work together to support various compilers, and as a consequence, binary output is produced that is a compiled version of the Lucid program [1] that can then in turn be executed by the GISPY execution engine. The GIPC is a

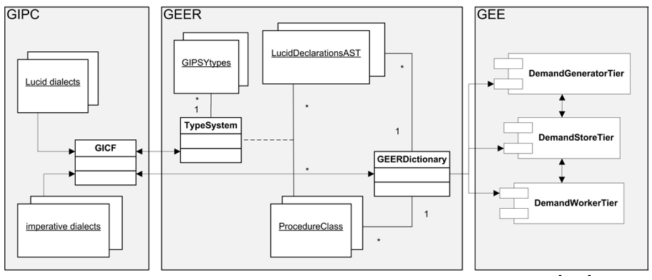

Fig. 2. High-level structure of GIPSY's GEER flow overview [1].

compiler framework that enables syntax and semantic analysis, 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) [11], [1]. 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 [1], [12], [13]. 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 [1]. The Demand Migration System (DMS) can be realized as an instance of the Demand Migration Framework (DMF) [13]. 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.

C. Kubernetes

Kubernetes is an open-source container-based orchestrator that aids in the autonomous deployment of scalable, reliable, and manageable distributed systems. In order to achieve these goals, Kubernetes provides numerous services that are provided through the network via APIs [14]. 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 capacity and keep up with ever-increasing demand without requiring a fundamental restructure of the distributed system that implements the services [15]. 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 [15]:

- **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 [16] to eventually resume the computation. Forensic computation potentially involves 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 [14], [15]. 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. When developers build their applications, moving the applications across various environments happens merely by moving the declarative configuration to adapt to the different context [15], [14]. 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 features for resource management. At the container level, parameters can be assigned to containers for setting minimum or maximum values for the resource assignment to the container, such as CPU load and RAM volume. Given that forensic computation potentially involves very resource-intensive demands on the computation nodes, such efficiency-related features provided by Kubernetes is necessary in our context [15], [17].

Before we illustrate the architecture of Kubernetes (Figure 3) and how it is used in our forensic computing solution as part of the GISPY execution engine, we here provide a brief introduction to some fundamental components of Kubernetes used in our solution:

- A **Pod** is the smallest unit in a 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 [14].
- A **Deployment** is an abstraction used to create or modify instances of the pods. It can scale up and scale down the number of replicas of pods [14].
- A **Service** is 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 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 [14].
- A **Volume** is used to preserve the data produced by a 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 [14].
- A **Persistent Volume** is an API that abstracts the implementation of physical storage for the pod's usage whose lifecycle is independent of the pods [14].
- A **Network File System** 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 [14].

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 panel manages the object states and any modification on the cluster. It decides on the cluster's big picture [15], [14].

III. Solution

In this section, we first describe the architecture of our solution to achieve the requirements regarding the GitHub demand-driven JSON to Forensic Lucid encoder formalization in GIPSY. We then present the integration of the Kubernetes cluster for the GIPSY system and different methods we used to design, implement, and evaluate our proposed solution.

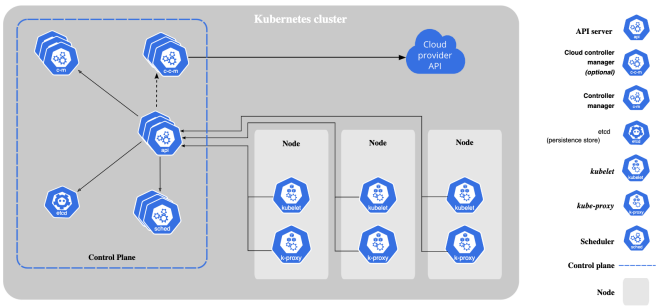

Fig. 3. Kubernetes architecture [14].

A. GIPSY's 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 [11]. As a result a lot of digital evidence online comes in this format. Figure 4 presents a sample of a specific GitHub JSON data file that is passed to the encoder. To reason about the JSON-based digital evidence in our system it has to be encoded in Forensic Lucid first, which is true for all evidential sources. The JSON-to-Forensic Lucid encoder is a Java program created to transform a JSON string into Forensic Lucid code which is then provided to the GIPSY system for compilation [11]. In Figure 5 you can observe the use case diagram for the JSON to Forensic Lucid encoder. 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 [11]. For instance, in order to create a JSON parser, we can implement `IJsonParser` corresponding to the GitHub JSON structure. The sequence diagram for JSON to Forensic Lucid encoder is illustrated in Figure 6. As depicted in Figure 7, 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 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 he three GIPSY tiers (DGT, DWT and DST) that will generate and/or process these demands. 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

```
{
  "sha": "0bf3ce8e57f539673f06971d95c171d2b599ab71",
  "node_id": "MDY6Q29tbWl0MzglNjkyNzAwOjBiZjNjZThlNTdmNTM5NjczZjA2OTcxZDklYzE3MWQyYjU5OWFiNzE=",
  "url": "https://api.github.com/repos/OpenISS/librealsense/git/commits/0bf3ce8e57f539673f06971d95c171d2b599ab71",
  "html_url": "https://github.com/OpenISS/librealsense/commit/0bf3ce8e57f539673f06971d95c171d2b599ab71",
  "author": {
    "name": "Eran",
    "email": "librealsense.eran@gmail.com",
    "date": "2021-06-13T15:46:01Z"
  },
  "committer": {
    "name": "GitHub",
    "email": "noreply@github.com",
    "date": "2021-06-13T15:46:01Z"
  },
  "tree": {
    "sha": "ef174f834eb7b36af67e3737ec314828bd1a52ed",
    "url": "https://api.github.com/repos/OpenISS/librealsense/git/trees/ef174f834eb7b36af67e3737ec314828bd1a52ed"
  },
  "message": "PR #8156 from Andrew: rs-record-playback now writes bag file to temp dir",
  "parents": [
    {
      "sha": "c40c65e65c18f4af356713c765320464fa2808fd",
      "url": "https://api.github.com/repos/OpenISS/librealsense/git/commits/c40c65e65c18f4af356713c765320464fa2808fd",
      "html_url": "https://github.com/OpenISS/librealsense/commit/c40c65e65c18f4af356713c765320464fa2808fd"
    },
    {
      "sha": "b646e014bb71ef14755b4d3b28d6af87f2a1a717",
      "url": "https://api.github.com/repos/OpenISS/librealsense/git/commits/b646e014bb71ef14755b4d3b28d6af87f2a1a717",
      "html_url": "https://github.com/OpenISS/librealsense/commit/b646e014bb71ef14755b4d3b28d6af87f2a1a717"
    }
  ],
  "verification": {
    "verified": true,
    "reason": "valid",
    "signature": "-----BEGIN PGP SIGNATURE-----\n\nwsBcBAABCAAQBQJgxig5CRBK7hj4Ov3rIwAAGX0IAK+i+lXoZALQmbQxzficQaYR\n621obIFi284cQ4isvxlfpaVSuYwzz92ApMasWAIQMPXm+7lMMDyXRhbcaLesmyCO\n",
    "payload": "tree ef174f834eb7b36af67e3737ec314828bd1a52ed\nparent c40c65e65c18f4af356713c765320464fa2808fd\nparent b646e014bb71ef14755b4d3b28d6af87f2a1a717\nauthor Eran <librealse
  }
}
```


Fig. 4. GitHub JSON data format sample.

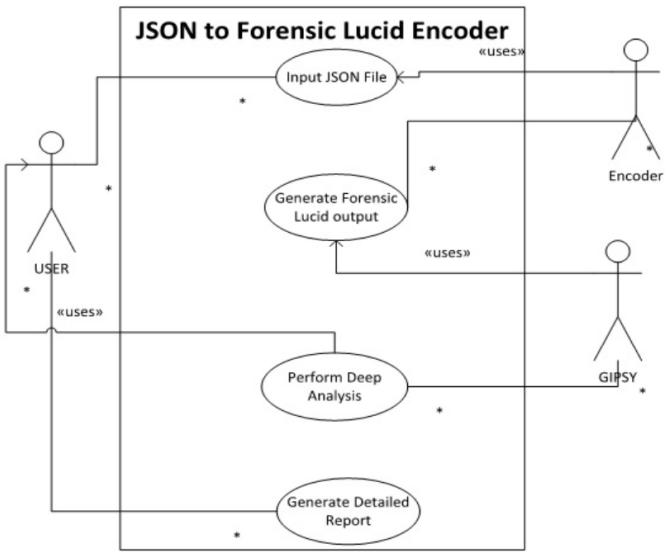

Fig. 5. JSON to FORENSIC LUCID encoder use case diagram [11].

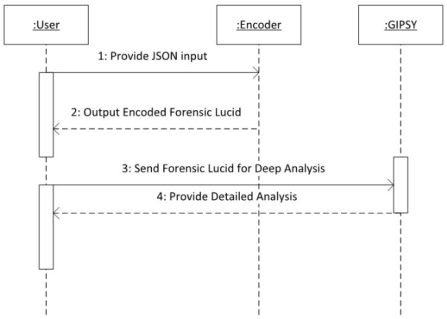

Fig. 6. System-level sequence diagram for JSON to FORENSIC LUCID encoder [11].

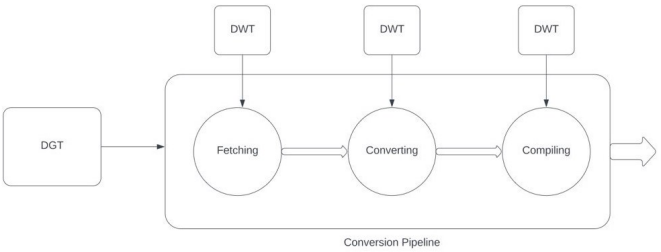

Fig. 7. Conversion pipeline.

into the demand itself, at which point the demand is put in the processed state. The pipeline starts by the demand generator (DGT) to generate fetch demands for the processing of specific GitHub queries according to the forensic investigator's specifications. Once the DGT has created the fetch demands, it will wait for these demands to be in the processed state. Once it finds some of these demands to be processed, it will proceed to generate a corresponding convert demand to generate the corresponding FORENSIC LUCID code that corresponds to this JSON GitHub fetch demand, will put it in the DST and wait for it to be processed. Once it finds some of these demands to be in the processed state, it will proceed to generate a corresponding compile demand to generate an executable version of this FORENSIC LUCID code. Once all the demands have been processed, the processing will be over.

To perform the Forensic investigation on the GitHub repository using GIPSY JSON demand-driven encoder, we shall have software that conducts a pipeline such as: fetching the

JSON data from GitHub API, converting the JSON data to Forensic Lucid file and compiling the converted file. Figure 8 illustrates the sequence diagram for the JSON to FORENSIC LUCID Encoder. In this diagram, the `JSONCONVERTERDGT` is the demand generator (DGT), `JSONCONVERTERDWT` is our worker (DWT), and DST, 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 URL demand index with a status of the demand in the DST (pending, in process, and computed) and pass it to the worker. 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 (context) file, which includes the list of JSON files, and include a demand to let the Generator know that the serialized object has been computed and store the results back to the DST. 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. After the DGT receives the results, it will create another demand to convert the JSON files to Forensic Lucid. DGT will create a Java object with the CTX file containing the list of JSON files and store the demand in the DST. Then the 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. In the last step, DGT will create a demand to compile all the FORENSIC LUCID IPL files. DGT 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. Eventually, DGT will pass the final results to the user.

B. Integrating Kubernetes and GIPSY distributing system

In this section, we are going to describe how we integrated the Kubernetes orchestration platform with the GIPSY system in order to distribute the process of what we described in Section III-A. Before we dive into the integration, we will describe the architecture of GIPSY Node and GIPSY Tiers [1]. Figure 10 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 [1]. 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 [18]. 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 [1]. 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 [1]. 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 [1]. 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 [18]. With what we described with the GIPSY architecture, we would like to take advantage of Kubernetes to have a distributed system that can be automatically scalable and highly available. As we illustrate in the Figure 11, we have one Control Panel and multiple Worker Nodes in our architecture. Control Panel consists of essential components as we described in the Section II-C. In order to work with the cluster, Developer will transmit its request to the Kubernetes API Server, which is running on the Control Panel 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. Each Pod consists of a Network Namespace, GIPSY Tier Container (DGT, DWT, DST) and additional supporting containers if needed. In our architecture, as an additional supporting container, we have a Jini Container

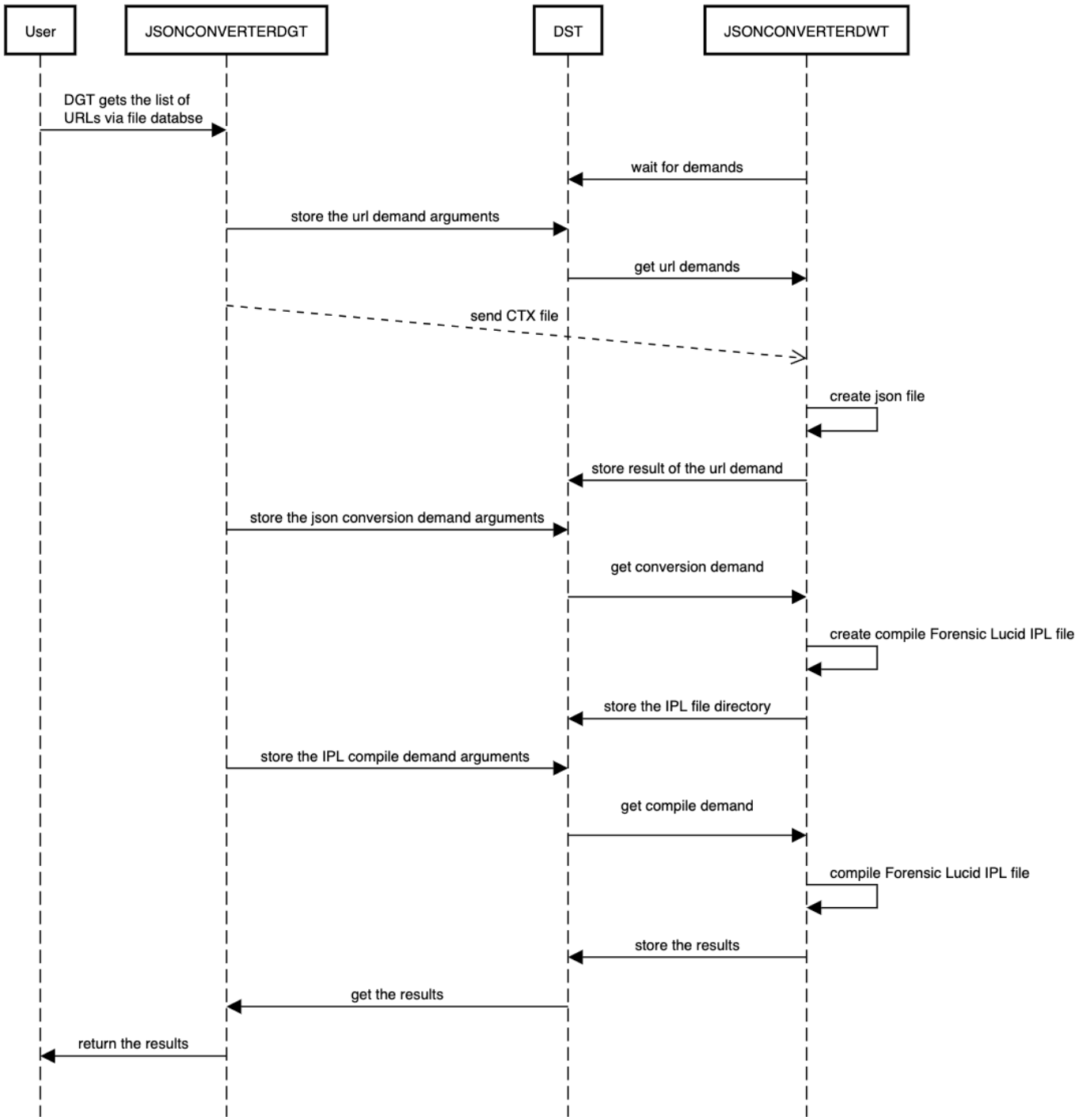

Fig. 8. Detailed design sequence diagram for Github JSON to FORENSIC LUCID encoder.

```
OResult
where
evidential statement es = {o_commit_0b};
        observation sequence o_commit_0b = {[[sha : "0bf3ce"], [commit ], committer, stats , files]};
        commit = {[author : {[name : "test"], [date : "Thu. Dec. 03 06:33:28 2020"]}]};
        committer = {[login : "test"], [id : 19864447], [node_id : "MDQ6VXNlcjE5ODY0NDQ3"]};
        stats = {[total : 369], [additions : 221], [deletions : 148]};
        files = {[[sha : "8e634f"], [filename : "common/CMakeLists.txt"], [status : "modified"]]};
        OResult = es;
end
```

Fig. 9. Simple FORENSIC LUCID from GitHub repository

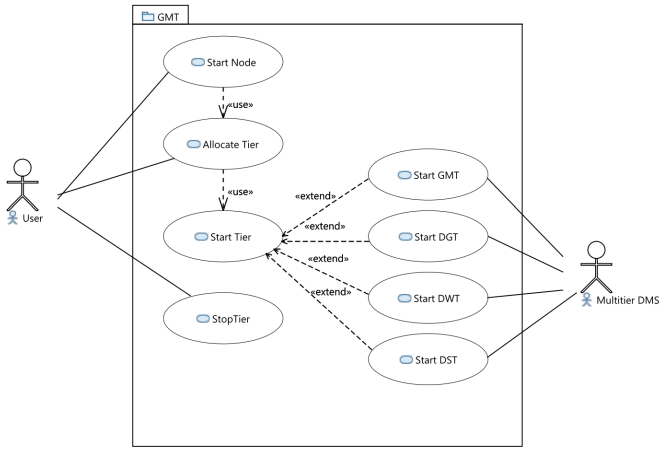

Fig. 10. GMT use case diagram [1]

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. In

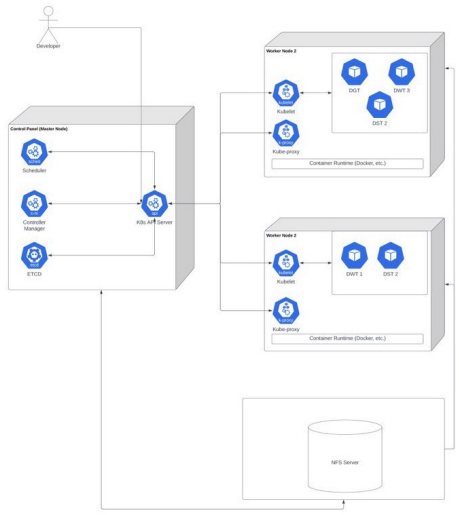

Fig. 11. Kubernetes integration with GIPSY's GEE runtime

order to make this immigration work, we containerized the GIPSY program with Docker, so a GIPSY Tier can operate inside each pod.

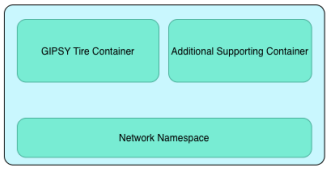

Fig. 12. Kubernetes pod and GIPSY node architecture

IV. EVALUATION

In this section, we describe the experiments that were conducted in order to gather a forensically interesting dataset. In order to achieve this dataset, we ran our experiments using the proposed solution as we described in Section III by executing the GIPSY's JSON to Forensic Lucid Encoder pipeline, i.e., extracting data from GitHub API in JSON format, converting extracted JSON data into Forensic Lucid code and compile the Forensic Lucid code. For gathering a dataset, as we mentioned, we mainly aim for the cybersecurity vulnerabilities that exist publicly and fetch them from the Commit API since it contains various practical information such as commit messages, comments and patches, which show the changes made to the file. A system called Common Vulnerabilities and Exposures offers a way for the public to exchange knowledge about cybersecurity vulnerabilities and exposures. 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*

By executing our experiment on the URLs we provided, we collected our dataset containing 1000 JSON files, 1000 IPL Forensic Lucid files, and corresponding compiled Forensic Lucid files. We followed our experiments by finding a relation between the number of workers and the number of URLs. Figure 13 depicts the total execution time for each amount of workers. As we can observe below, the execution time remains constant once there are ten workers, which is equal to the number of URLs. Following, we discuss the experiments conducted to distribute the execution of the Forensic Lucid encoder by integrating Kubernetes and GIPSY. In our experiment, we employed three physical computers, one as our Control Panel and two as Workers. Employing Kubernetes in order to have a GIPSY container orchestration allow us to have a GIPSY cluster that:

- Each tier instance can quickly get started using a *.yaml* configuration file.

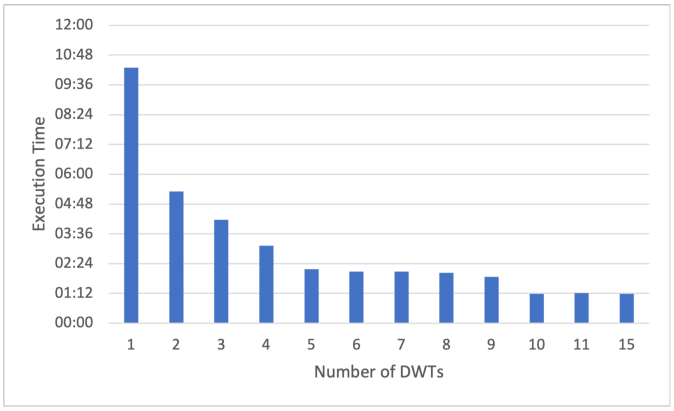

Fig. 13. Execution time depending on the number of DWTs (workers)

- Scaling up and down the cluster is only a matter of modifying the replica number in the *.yaml* file to deploy a number of GIPSY compute nodes.
- Once a Node goes down, Kubernetes will handle the relocation of the running GIPSY compute node pod to the next available node automatically.

V. Conclusion

In this section, we conclude our research based on the conducted experiments and the results that we achieved in Section IV [10].

- In our work, 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 a 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.
- In the Forensic Lucid conversion pipeline, we defined each URL 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.
- 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 a GIPSY instance, which saves a significant amount of time.
- 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.
- 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.

A. Limitations and Future Work

At the moment, despite the fact that we achieved all the requirements [10] 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 this 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, Podman, 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 [10] 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 near future.

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

REFERENCES

[1] 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.

[2] 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, pp. 108–113.

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

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

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

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

[7] 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, pp. 1266–1271.

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

[9] 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, pp. 101–109, pre-print at http://arxiv.org/abs/0906.3911.

[10] S. P. Zahraei, "A GIPSY runtime system with a Kubernetes underlay for the OpenTDIP forensic computing backend," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2022, https://spectrum.library.concordia.ca/id/eprint/991314/.

[11] 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, pp. 1–6.

[12] 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/.

[13] 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.

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

[15] B. Burns, J. Beda, and K. Hightower, *Kubernetes Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, 2019, ISBN: 9781492046530, https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/docs/vmware-kubernetes-up-running-dive-into-the-future-of-infrastructure.pdf.

[16] 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.

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

[18] 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, 2009, pp. 218–224.