LLAMA

A system for log management and analysis on a complex distributed environment

Francesco Larghi

Supervisor: Prof. Alessandro Margara

Master's degree in Telecommunication Engineering

Internet Engineering

Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria DEIB

Italy

Academic Year 2018-2019

Abstract

This thesis is about the development of a system to manage and analyze logs on a complex distributed environment. Logs are a crucial element used to check the behavior and the status of a system, but unfortunately their content and format are different in the various operating systems, software, network equipment or any other component of a system. Logs need to be collected, parsed and stored typically all in one place to be subsequently analyzed, but it's a problem to move huge quantity of data over internet. This is true both for network saturation and security or privacy issues. That's why it's safer to keep logs where they are, in a distributed fashion, also because we can split computational and storage resources in a much more efficient way. But we need a system to efficiently retrieve or analyze logs scattered over the distributed system, possibly through a central website tool.

That's why we develop a system composed by both our software and open-source public projects to have a unique tool to manage and analyze logs on a complex distributed environment, without external costs. We used the Microservices approach to design a modular lightweight system, easy to upgrade and deploy and to keep the possibility of changing only single components of the system.

The result is the project LLAMA, a system to efficiently organize and archive logs on a distributed infrastructure with a central website tool to access and analyze all of them.

Acknowledgements

First of all, I would like to thank prof. Alessandro Margara for having accepted to be my advisor and for his great support for this thesis redaction. The project was designed and implemented during my internship and my work experience in Elmec Informatica, so I would like to thank the company itself for giving me this opportunity.

Thank to Andrea Bombelli, manager of the LLAMA project, for helping me in designing the system and especially for his friendship and continuous support during my experience. Thank to Ivan Paterno, the other member of the LLAMA team, for his willingness in finding new solutions everyday and for its backing to this project together with the Security Team, which I would like to thank too. Thank to Alessio Scaglia, the leader of the R&D Team where I still work right now, for his encouragement and for his availability to let me work part-time to finish my studies and complete my thesis. Thank to Marco Mezzaro for his support with distributed and deployment technologies, and for the time spent solving various issues. Thank to all my other colleagues that have directly or indirectly participated to the implementation of this project: Andrea, Luca, Samuele, Daniele, Matteo, Emanuele, Ivan, Adriano, Marta and Samuel. It's always funny and stimulating working with you.

My life journey in the past years was not always simple, but my family and my friends always supported me. Most of all, I would like to thank my mother, my father and my sisters, Federica and Marta, for their invaluable help over these years. Finally, I would like to thank my son, Lorenzo, who this work is dedicated to. He gives me the strength to face every difficulties, no matter how hard they look. This work and my university experience overall are also a demonstration to him that everything is possible if you work and study with dedication for that.

Contents

List of Figures

Chapter 1

Introduction

The majority of our progress as humankind has been made thanks to recording and using data, since we developed the ability to write. In the past, notes were made mainly on paper and were compiled and stored by hand. Because of this documentation method, important information was also prone to being misplaced, lost, or even mishandled. Today, practically every important action - especially those that occur online and on our computers, mobile phones and tablets - is recorded somewhere in some way or another.

A log is a unit of data recording an event in some format, from software actions (e.g. well-known log-in and log-out actions), to network activity and operating system events. With the super fast growth of internet communications, employing a proper log management solution is now more crucial than ever before. Nevertheless, unlike antivirus software or similar apps, log management programs are not something that the average user is really interested in. Many probably are not even aware of how they operate and which problems they help to solve, even though log management tools form the basis of some of the most significant parts of the IT industry.

In November 2018 I had an interview with ”Elmec Informatica S.P.A”, an IT company with headquarters in Brunello (VA). This company, among the other business, is a leader in server delivery and administration. They manage both systems located in their clients structure and hosted servers in their own datacenter. They proposed me the idea of a project about a distributed infrastructure to offer log management services without high costs. Since I was very interested in this subject, I took this opportunity to write my master thesis on it and so I started an internship that lasted 6 months.

1.1 The problem context

The idea was to offer a basic log management and analysis service to the company clients. The problem is that each client has his own infrastructure with lots of servers and therefore tons of potential data to gather, process and analyze. Each infrastructure could be hosted inside the company datacenter or even physically present on the client's site. Since clients are many, the amount of data to simultaneously manage could be extremely big and often log management tools charge you for the amount of data you process too. Of course this kind of payment system is not desirable at all for business that needs to scale on number of clients and tons of data. Even if it could be sustainable, send all the logs to a single central warehouse could be very complicated, both for network saturation and security potential issues. That's why they were looking for an ad-hoc alternative, with the help of some open-source technology too, to be able to scale without problems on many clients, to grant a basic log management service without huge fees and without the need to centralize all the logs in one single point.

In the company I was inserted in the Automation&Cloud Team, where I'm actually still working right now. This project was and is still in collaboration with the Security Team since the log management is on their charge. I initially joined a group of 2 people, Andrea and Ivan, with the goal of designing and implementing a proper system.

That's how started the project LLAMA, a name that we crafted and actually stands for Logs Archive Management and Analysis. It is a system that both provide a distributed architecture for log collection and management and a tool, available on a webpage, to launch queries on this infrastructure and retrieve the relative results.

1.2 Roadmap

This thesis is mainly about distributed systems and the core part of the whole project is the message-oriented middleware for query distribution. That's why I'm going to focus more on this topic, but I will also cover log management since it's the actual use case of the system. I'm going to make a brief survey and comparison of the available software around, dealing both commercial software and open-source project with their pros and cons.

In the next chapter I'm going to make a list of all the project requirements. In chapter 3, I will give a general background about the technologies available today. Then, in chapter 4 I'm going to yield more information about the actual technologies that we decided to adopt, accordingly to the previous requirements. In the fifth chapter I'm going to focus on the design choices that we took and the overall system architecture, the security measures and the main configurations. In chapter number 6 I will go into details of the actual implementation, mainly about the critical parts of the system, the list of the main endpoints and the final user interface too. Then, in chapter 7 I'm going to give an evaluation, based on the given requirements, of the final result. Finally, in the last chapter I will make concluding considerations about the project and the future work.

Chapter 2

Requirements

The final objective of this project is to have a unique website tool to launch queries and retrieve results on a distributed infrastructure that separately store logs for each client. The project originally started with few requirements, but they rapidly increase togheter with the project growth. Here is a detailed list of all the final requirements.

2.1 Central managed distributed infrastructure for log management

The system should consist in a central website tool, communicating with a server in its turn connected to a distributed infrastructure with one node per each client. Each node is in charge of log collection and management.

2.2 Node isolation

Each node is not to communicate with other ones. This is crucial because each node handle client's private data that should not be leaked even across other nodes of the system.

2.3 JSON as unique representation of data

Data across the whole system should be formatted in JSON where possible. This should be true both for data-interchange communications and data storage. Keeping just one format should simplify the overall management of data in the system.

2.4 Efficient and reliable log management for each node

Each node should collect, transform, route, forward and store logs efficiently and quickly. This means that the node should be able to collect even thousands of logs per second from various sources in different formats, parse them into a unified format (e.g. JSON) and save them in an appropriate storage at the same time.

2.5 Log retention for a specific amount of time or space

The log storage should be able to retain and persist data for a certain amount of time (e.g. 6 months) or until a specific threshold of occupied space is reached (e.g. 100GB).

2.6 Encrypted storage for logs

Logs contains private client's data and must to be stored in encrypted secured storage. No one should be able to alter or see logs except for the system administrators or data analysts, with adequate privileges.

2.7 Replicated data storage for logs

The data storage should admit the possibility of being extended to a cluster of machines if particular fault tolerance measures or better read/write performances are needed.

2.8 Easy deployable nodes and backend infrastructure

Services composing the backend infrastructure should be easy to deploy, substitute, update and maintain in general. Also, distributed nodes should be easy to deploy efficiently with a standardized and automated procedure.

2.9 One website to launch queries on all the nodes

To launch queries on the nodes there should be a simple website. The website should allow the users to:

1. Consult logs with a simple interface
2. Create sophisticated queries for log analysis with an advanced interface
3. Launch query asynchronously and consult results
4. Schedule queries for repeated analysis or checks (see next requirement)

2.10 Query scheduling system

The system should allow to create scheduled queries that repeatedly execute on a custom interval of time. Schedules should be added and deleted transparently, without system restarts.

2.11 Authenticated endpoints and system access

Each endpoint of the backend system should be secured by authentication. Only users correctly authenticated coming from the website tool should be able to interact with the system and launch queries.

2.12 TLS and VPN secured communication

The communication between all the components and services of the system should be secured with TLS encryption. The traffic traveling over internet should also go over a private VPN connection.

Chapter 3

Background

3.1 Message-oriented Middleware for Distribution

With the ever-growing technological expansion of the world, distributed systems are becoming more and more widespread. They are a vast and complex field of study in computer science. A distributed system in its most simplest definition is a group of computers working together as to appear as a single computer to the end-user. These machines operate concurrently and can fail independently without affecting the whole system's uptime. Systems are always distributed by necessity. There are some major benefits from distributed system despite their objective complexity:

- **Scaling Horizontally** - In a standard system the only way to handle more traffic and operations would be to upgrade the hardware and this is called scaling vertically. It is all well and good while you can, but after a certain point even the best hardware is not sufficient for enough traffic. Scaling horizontally simply means adding more computers rather than upgrading the hardware of a single one. It is much cheaper after a certain threshold respect to vertical scaling. The best thing about horizontal scaling is that you have no cap on how much you can scale — whenever performance degrades you simply add another machine, up to infinity potentially.

- **Fault Tolerance** - A cluster of many dislocated machines is inherently more fault-tolerant than a single machine. Even if one place catches on fire, your application would still work.

- **Load Balancing** - With such systems it is possible to distribute workloads on many machines, optimize operations, maximize throughput, minimize response time and avoid overload of any single resource.

For a distributed system to work, though, you need the software running on those machines to be specifically designed for running on multiple computers at the same time and handling the problems that come along with it. That's why distributed systems are considerably more difficult to get right. There are many different levels and type of distribution, like distributed filesystems, computing, application or even ledgers with the more recent blockchain technology. These technologies act like a middleware, the software layer that lies between the operating system and the applications of the machines in a network. In this way, the communication level between multiple hosts of a system is abstracted. For this project we are mainly interested in messaging systems, because these kind of architecture suits our requirements [2.1].

Message Oriented Middleware (MOM) is a concept that involves the passing of data between applications using a communication channel that carries self-contained units of information (messages). In a MOM-based communication environment, messages are usually sent and received asynchronously. Using message-based communications, applications are abstractly decoupled; senders and receivers are never aware of each other. Instead, they send and receive messages to and from the messaging system. It is the responsibility of the messaging system (MOM) to get the messages to their intended destinations. Usually these system works with a "publisher-consumer" system (also known as "publish-subscribe"). The publisher process is in charge of inserting messages in one or more queues. Each queue buffers the messages until a consumer process (typically running on a different machine) takes the message and consumes it. The consumption mechanism it's often different for each technology. That's why now we are going to analyze the state of art, the current major software and projects available around as message oriented middleware.

3.1.1 RabbitMQ

RabbitMQ is an open-source MOM that originally implemented the Advanced Message Queuing Protocol (AMQP). It has later been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols. RabbitMQ supports a Publish-Subscribe mechanism. It allows a publisher object to send messages to a Queue or even to an Exchange. The latter is an object that allows to route messages to multiple Queues in different ways (Direct, Topic or Fanout) [fig. 3.1]. The connections between Exchanges and Queues are called Bindings. It also support an RPC system that automatically spawns and deletes callback-Queues that receives the result of message consumption from the consumer.

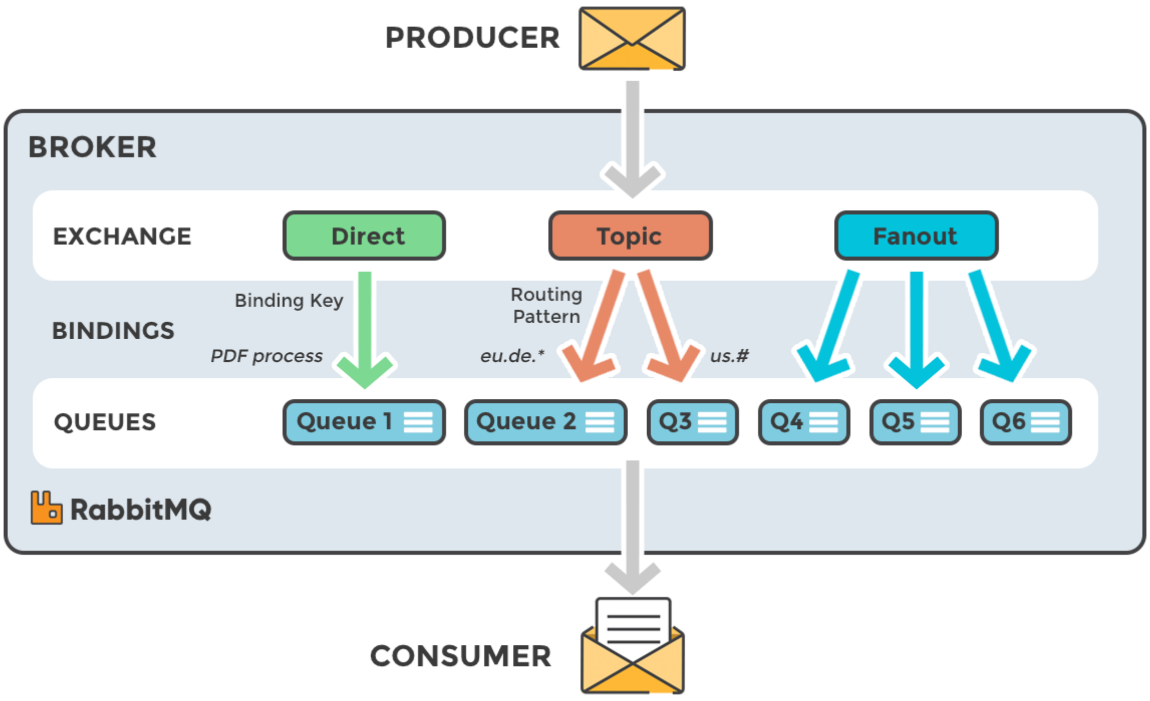


Figure 3.1: RabbitMQ

3.1.2 Apache Kafka

Kafka is a distributed messaging system providing fast, highly scalable and redundant messaging through a Publish-Subscribe model. Kafka's distributed design gives it several advantages. First, it allows a large number of permanent consumers read simultaneously from files. In fact Kafka relies on the filesystem to store permanently queues and messages instead of load them into memory. Second, Kafka is highly available and resilient to node failures and supports automatic recovery. In real world data systems, these characteristics make Kafka an ideal fit for communication and integration between components of large scale data systems. All Kafka messages are organized into Topics, that are the actual queues locally saved into files. Kafka, as a distributed system itself, runs in a cluster. Each node in the cluster is called a Broker. Topics are divided into a number of Partitions [fig. 3.2]. Partitions allow you to parallelize a Topic by splitting the data across multiple Brokers. Each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel. Consumers can also be parallelized so that multiple Consumers can read from multiple Partitions in a Topic allowing for very high message processing throughput. Kafka performances scale a lot horizontally, even with few machines available [bib. [1]].

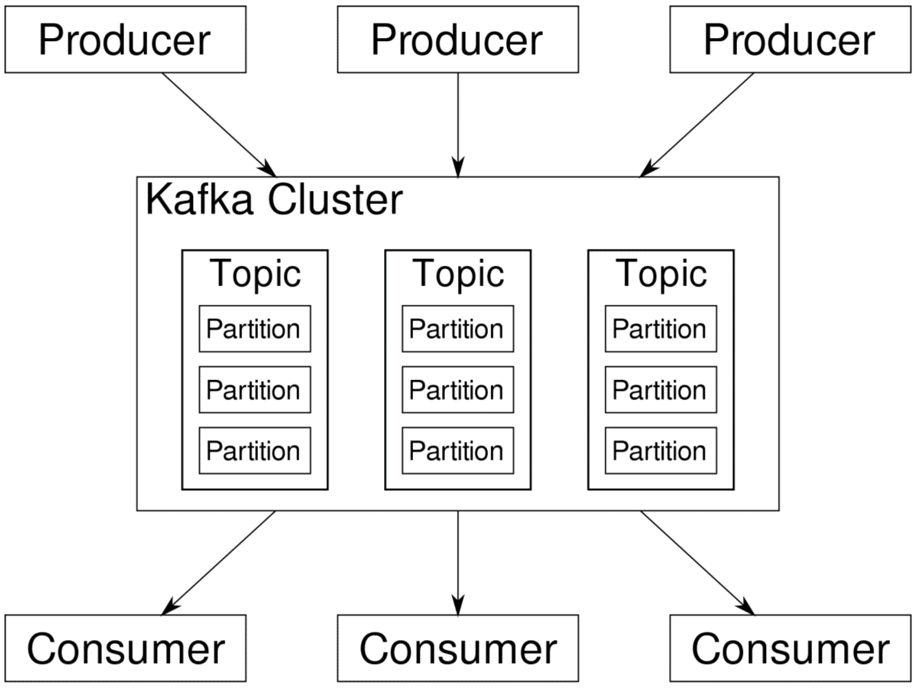


Figure 3.2: Apache Kafka

3.1.3 Redis

Redis is an open source in-memory data structure store. It's not just a MOM broker, it's also usually used as a cache or high performance database. It implements the Publish-

Subscribe paradigm, allowing client to publish message to a specific Channel (the actual queue). Redis clients could also subscribe to a Channel, consuming the incoming messages. The strength of Redis resides in its versatility of acting both as a message broker and a database as needed. This peculiarity can be exploited to store additional information, other than messages, into Redis. We will see later in much more details how we can use this feature [fig. 3.3]. Redis can also be deployed as a cluster using Sentinel or Redis Cluster. Redis grants great performances even with a single instance, compared to other in-memory storage systems [bib. [2]]. Since this is the actual Message-Oriented Middleware that we adopted for our project together with Celery (a Python client for Redis), we will go in much more details about this technologies in the Technology dedicated section [4.1].

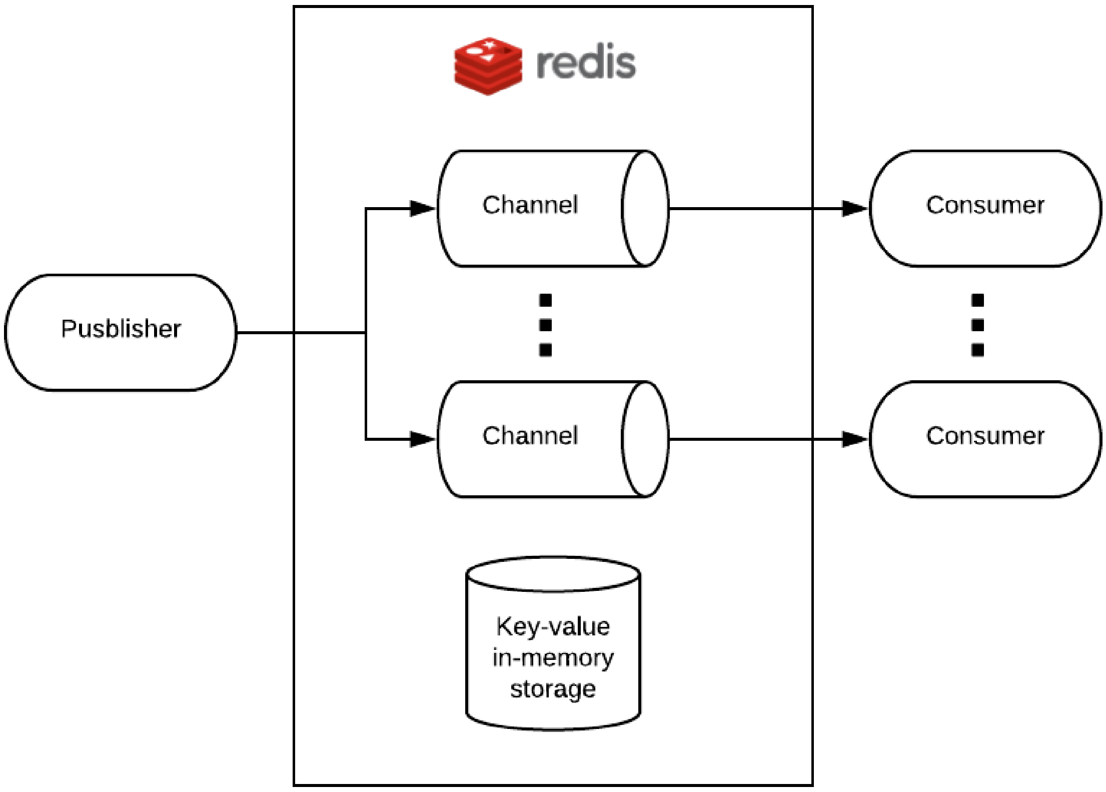


Figure 3.3: Redis

3.2 Log management

Log management gives the possibility to always monitor your systems and applications and have a detailed history of all the past events. Security specialists can act as soon as they notice any potential issues and danger to your security setup, reducing the overall risk your systems is subject to. Log management apps are an integral proactive security measure and without them, we would not know where to look at when accidents happens. We would know that something is wrong, but not be able to figure out exactly what, or at least without spending a lot of time searching for the problem. This wasted time can always be spent in better, more productive, and more strategic ways. Utilizing log management is a proactive measure that any business should take complete advantage of. IT security, by its very nature, has to be extremely adaptive and proactive, since attacks that compromise data protection evolve on a daily basis and are becoming trickier to detect and harder to overcome and repel. Regardless of company size or industry, everyone has security concerns, and with good reason. Log management is another layer of protection against unwanted incursions and data theft.

System administrators are tasked with overseeing how computer systems and servers are operating and making sure that everything is configured and working as intended. By collecting and reviewing logs, they know how the systems are supposed to function normally and can react when they notice that something irregular is happening. These logs are the first line of defense against any problem, so log management provides them a better and more precise methodological approach to their work. Log management solutions allow to generate ad-hoc rules for generating alerts and look for patterns correlating with similar events. Extensive search options enable system admins to quickly sort, find, and compare past log activity with current activity.

Last but not least, when writing code, developers often rely on log reports to figure out where bugs are located. Debugging, monitoring for errors, and troubleshooting are essential parts of software development, both during and after release. Through the use of log management apps, this often cumbersome process can be made easier, leaving developers with more time to squash bugs instead of wasting valuable resources and effort in hunting them down. Also, using the right sort of app to manage logs is crucial to any development team, since it converts unstructured data into useful and legible information they can read and make use of.

Following, we will make a survey of both paid commercial software and open-source projects.

3.2.1 Paid Commercial Software

There are tons of commercial software around for log management. Usually they charge you for the amount of data processed, that of course is not an ideal policy if you need to store and process huge amount of logs. One of the leader in log management is Splunk. It's probably the biggest tool in this space, a product who essentially created a new category. It's got hundreds of apps to make sense of almost every format of log data, from security to business analytics to infrastructure monitoring. Splunk's search and charting tools are feature rich to the point that there's probably no set of data you can't get to through its UI or APIs. Despite this, Splunk has two major cons. The first is that it's an on-premise solution which means that setup costs in terms of money and complexity are high. To deploy a high-scale environment you will need to install and configure a dedicated cluster. Splunk's second issue is that it's very expensive. To support a complex system this means tens of thousands of dollars and the process of integration is going to be slow too. There are also other popular log management "freemium" solutions such as DataDog, SumoLogic, Loggly and many others. In general the main problem of these software is of course that they are not entirely free as an open-source solution.

3.2.2 Open-source Projects

The most popular set of open-source projects for log management is by far the ELK Stack, a robust solution for search, log management, and data analysis. ELK stack consists in a combination of three open source project: Elasticsearch, Logstash, and Kibana. These projects have specific roles:

- **Elasticsearch** handles storage and provides a RESTful search and analytics endpoint.

- **Logstash** is a server-side data processing pipeline that ingests, transforms and loads data.
- **Kibana** lets you visualize your Elasticsearch data and navigate the Elastic Stack.

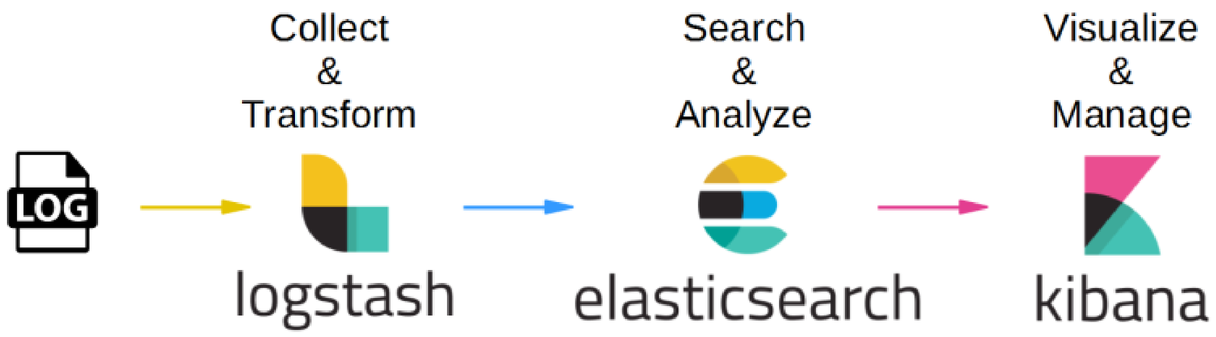


Figure 3.4: ELK stack

These tools combine to provide an all in one platform for logs storage, retrieval, sorting and analysis. ELK is becoming the most common open-source, log management platform used globally. It is a very good set of tools and, since it is open-source, it's also free. The throwback is that the whole set of software is actually suitable for most use cases, but not everyone. In those cases it's not easy to fix or customize ELK on your needs. For example, Elasticsearch is actually very fast for searching (its main purpose), but not as reliable and fast as other NoSQL databases in insertion. Logstash too, which is the core tool for log management, has its limits. For example, even if it is extensible through plug-ins, making your own custom ones it's not a simple process. You need to publish them in the official public repository before being able to use them. Furthermore, Logstash is a very heavy and resource demanding application running on JVM and this is not a trivial problem if you would like to deploy it multiple times on lightweight technologies such as containers. There are also alternatives to ELK and Logstash, such as Fluentd.

As the project website claims "Fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data". Since this is the software that we actually chose, we will later discuss in much more details its features in a dedicated section [4.2] and it's configuration for our project [5.3] togheter with the configuration of the relative log storage. Instead of other solutions (such as Elasticsearch), we decided to adopt MongoDB as our main storage for log collection and we will introduce this technology too in a future section [4.3].

Chapter 4

Technologies

4.1 Celery for Redis

Celery is a task queue implementation for Python web applications used to asynchronously execute work outside the HTTP request-response cycle [bib. [3]]. Task queues are used by Celery as a mechanism to distribute work across threads or machines. A task queue's input is a unit of work called a task. Dedicated worker processes constantly monitor task queues for new work to perform. Celery communicates via messages, using a broker to mediate between clients and workers. To initiate a task the client adds a message to the queue, the broker then delivers that message to a worker [fig. 4.1]. A Celery system can consist of multiple clients, workers and brokers, giving way to high availability and horizontal scaling. Celery supports different brokers like RabbitMQ, Redis, Amazon SQS, and more. In addition to this, it provides concurrent execution (through prefork, Eventlet, gevent) and serialization (with JSON, pickle, yaml, msgpack, zlib or bzip2 compression). As main broker we have chosen Redis, which is the official supported one, togheter with RabbitMQ. In this way Redis channels are directly manipulated by Celery as Celery queues, to route its task execution mechanism. Task execution logic can be exploited to distribute query execution across our system [2.1]. I will go into details of how we implemented Celery tasks and workers to achieve our goal in the Implementation chapter [6.1].

Celery also has a built in daemon service to schedule tasks, called Celery Beat. The default class of Beat that handle schedule storage saves data on a local file. This is

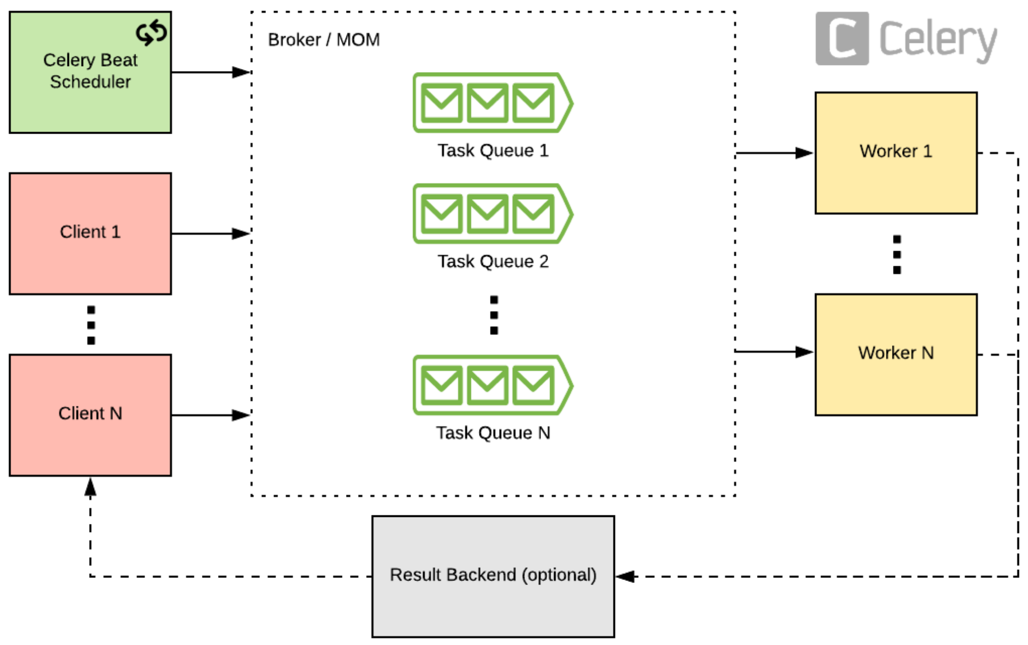


Figure 4.1: Celery architecture

an issue, because if you need to modify the schedule (adding or removing scheduled tasks) you need to restart the daemon service to upload the changes. Fortunately, it can also be substituted with custom ones. In our case, we substitute it with RedBeat, an open-source Celery Scheduler class developed and adopted by Heroku. RedBeat stores the scheduled tasks and runtime metadata directly on Redis [bib. [4]]. It gives various performance improvement in addition to the possibility to dynamically change, insert or delete scheduled tasks at runtime. Moreover, RedBeat also grants greater stability respect to the default Beat class [bib. [5]]. This matches our requirements of having a query scheduling system [2.10]. The software provides also external tools integration like Flower [bib. [6]], to monitor task success or failure, and Jobtastic to estimate and report execution progress. We also make use of Redis Commander, an open-source client to directly monitor Redis' status of channels and database [bib. [7]]

4.2 Fluentd

Fluentd is an open-source log collector and manager which lets you unify filter, buffer and route data incoming from different sources and formats [bib. [8]]. It has a built-in reliable buffering system and minimum resource requirements (it's implemented in C and Ruby). Moreover, Fluentd tries to structure data as JSON as much as possible: this allows Fluentd to unify all facets of processing log data: collecting, filtering, buffering, and outputting logs across multiple sources and destinations. This is a major plus for our purposes since we decided to adopt JSON as main data representation and communication model [2.3]. Fluentd's routing engine redirects messages to one or more destinations based on their source, format, or metadata, in accordance with user needs. Fluentd also supports filtering messages, adding custom fields, and basic data stream manipulation, that our system requires [2.4]. It is a fully pluggable architecture easy to extend. You can implement your own plugins in Ruby and directly use them in your Fluentd instances. As a Cloud Native Computing Foundation (CNCF) project, Fluentd integrates with Docker and Kubernetes as a deployable container too, which make it easy to deploy also in multiple instances [2.8].

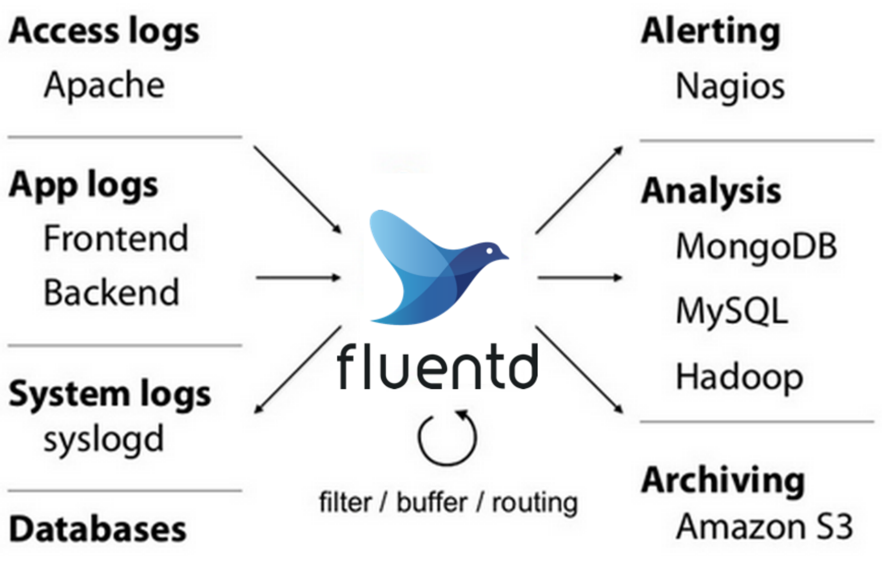


Figure 4.2: Fluentd

4.3 MongoDB

MongoDB is a document-oriented NoSQL database used for high volume data storage. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other. The values of fields may include other documents, arrays, and arrays of documents. This storage fashion is ideal for logs. MongoDB represents JSON documents in binary-encoded format called BSON, which extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages. This means that MongoDB gives users the ease of use and flexibility of JSON documents together with the speed and richness of a lightweight binary format. This fits our needs to keep JSON as a standard across the whole architecture [2.3]. Moreover, MongoDB grants a super fast insertion mechanism, more than 50k document inserts per second on a common hardware setup with 4 cores CPU [2.4]. Performances are also greater since the introduction of WiredTiger engine with MongoDB 3.0 [bib. [9]]. Space or time based document retention are easily implementable respectively with Capped collection and Indexes expiration [2.5]. Also, MongoDB allows for replication through Replica-set system (which grants both fault tolerance and better read performances) and data-set partitioning called Sharding (which allows read and write scale on multiple parallel fragments) [2.7]. Last but not least, MongoDB features powerful tools to analyze documents. For example, the Aggregation framework is a powerful way to run on collections a pipeline of various commands, such as filtering, grouping, sorting and even more complex statistical calculus, in sticking with our requirements of providing an interface featuring an advanced way for creating analysis queries [2.9]. I'll explain later the actual implementation of query execution system on MongoDB in our distributed system [6.1].

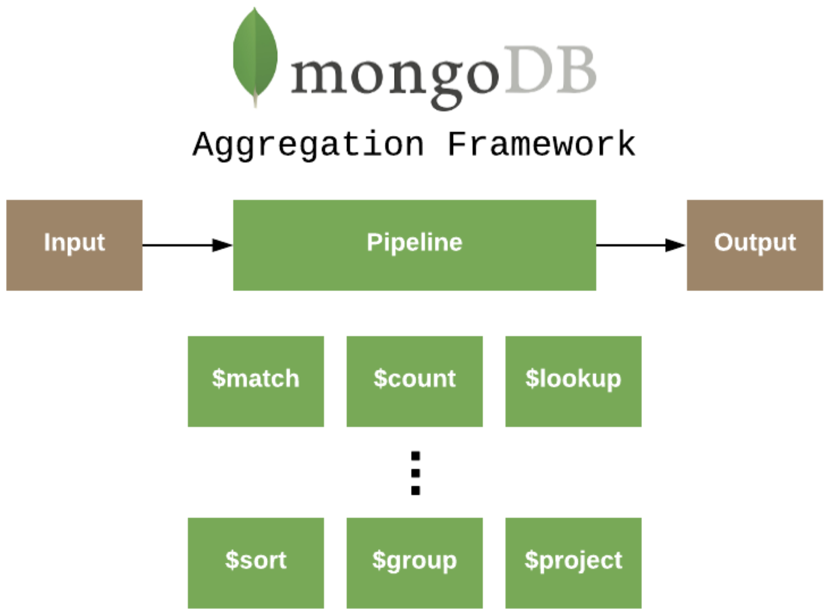


Figure 4.3: MongoDB Aggregation Framework

4.4 Flask

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks. Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy. One of the main libraries we adopted is Flask-RESTful. It is an extension for Flask that adds support for quickly building REST APIs. You can easily define endpoints and relative callable actions exploiting Python's decorators.

4.5 Containers on Docker and Kubernetes

Containers, along with containerization technologies (like Docker and Kubernetes), have become common components in many developers' toolkits. The goal of containerization, at its core, is to offer a better way to create, package, and deploy software across different environments in a predictable and easy-to-manage way. Containers are an operating system virtualization technology used to package applications and their dependencies and run them in isolated environments. They provide a lightweight method of packaging and deploying applications in a standardized way across many different types of infrastructure. Respect to VMs, rather than virtualizing the entire computer, containers virtualize the operating system and the application directly. They run as specialized processes managed by the host operating system's kernel, but with a constrained and heavily manipulated view of the system's processes, resources, and environment. We can actually interpret containers as applications togheter with folders, libraries, frameworks, etc. that they depend on. Docker is by far the most common way of building and running containers. It is a set of tools that allow users to create container images, push or pull images from external registries, and run and manage containers in many different environments.

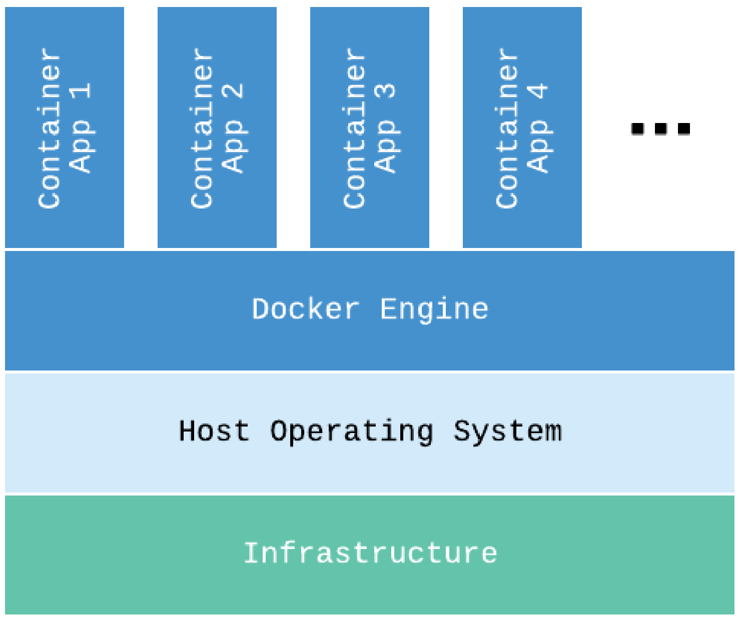


Figure 4.4: Containers on Docker

As backend production environment, we actually have in place a Kubernetes cluster (that also hosts many other applications developed in the company). Kubernetes (also written as k8s) is an open-source container-orchestration system for automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation. It aims to provide a platform for automating deployment, scaling, and operations of application containers across clusters of hosts. Kubernetes configure application through YAML files. Through Kubernetes Deployments, we can deploy containerized application with Pods and Services. A Pod is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, being executed on the same physical or virtual machine would mean being executed on the same logical host. Services are an abstract way to expose an application running on a set of Pods as a network service. They basically allow to map Pods exposed ports to other ports open in the external environment, outside of the inner network system.

I'm not going into further details about these technologies because this is not the goal of this writing, but these are the solutions that we implemented to fulfill our requirements on the infrastructure deployment [2.8]. We will see in the next chapter how the whole system was actually designed also thanks to these technologies.

4.6 Vue.js

VueJS is an open source progressive JavaScript framework used to develop interactive web interfaces. It is one of the famous frameworks used to simplify web development. VueJS focuses on the view layer. It can be easily integrated into big projects for front-end development without any issues. VueJS makes the use of virtual DOM, which is also used by other frameworks such as React, Ember, etc. The changes are not made to the DOM, instead a replica of the DOM is created which is present in the form of JavaScript data structures. Whenever any changes are to be made, they are made to the JavaScript data structures and the latter is compared with the original data structure. The final changes are then updated to the real DOM, which the user will see changing. The data binding feature helps manipulate or assign values to HTML attributes, change the style, assign classes with the help of binding directive called v-bind available with VueJS. Components are one of the important features of VueJS that helps create custom elements, which can be reused in HTML. VueJS provides various ways to apply transition to HTML elements when they are added/updated or removed from the DOM. VueJS has a built-in transition component that needs to be wrapped around the element for transition effect. We can easily add third party animation libraries and also add more interactivity to the interface. VueJS is very lightweight and the performance is also very fast. It is ideal to develop our website tool [2.9].

4.7 Ansible

Ansible is an open-source IT engine which automates application deployment, intra-service orchestration, cloud provisioning and many other IT tools [bib. [10]]. Ansible is easy to deploy because it does not use any agents or custom security infrastructure. It works by directly connecting to the nodes through ssh, Kerberos, etc. Ansible uses playbook to describe automation jobs. A playbook is a series of tasks described using very simple language i.e. YAML, which is very easy for humans to understand, read and write. After connecting to your nodes, Ansible pushes small programs called as “Ansible Modules”. Ansible runs that modules on your nodes and removes them when finished. It uses the hosts file where one can group the hosts and can control the actions on a specific group in the playbooks. We used Ansible playbooks to automate the deployment of our distributed nodes, with the whole configuration and software installation, as our requirements asked [2.8].

Chapter 5

Design

5.1 High level architecture

This project is designed taking account of our main requirements [2.1]. The overall architecture of the system can be divided into 3 main sections [fig. 5.1].

The first part is the software running on the main network of the company (Elmec Informatica) with the website (frontend), the backend services and the central database. The website consists in a tool inside the company official website for internal operations (”Automation Site”). This tool is called ”LLAMA”, as the project itself. It is implemented with Vue.js, a frontend framework technology we have briefly described in the previous chapter [4.5]. I will show later the final UI of this website tool [6.3]. On the ”Elmec Network” we also have the backend part hosted on Kubernetes. We will see in the next section a detailed description of the backend design. Lastly, here we also have the main database cluster implemented with MongoDB. On this database we store all the results of the queries, scheduled queries info, and other metadata about the overall system.

The second one is the broker, the MOM used for message distribution, hosted on the DMZ (demilitarized zone) network. It is a Redis instance, the core part of the whole infrastructure since it connects the central logic with the distributed machines. It is hosted on the DMZ network of the company because it also receives traffic from internet, which could be insecure and it’s a good practice to keep this components outside the inner network. Anyway, Redis is connected to the distributed nodes with a secured VPN and, as we will see later in the Security section [5.4], the traffic is encrypted too.

Finally, the third and last part is the set of nodes composing the distributed infrastructure. We called the single node DMI-Log (also written DMILOG, or DMI-LOG). DMI is actually an acronym already used for another distributed infrastructure of the company which stays for Distributed Management Infrastructure, so we kept the name adding "Log" to the end to identify the purpose of these machines. I'll show in section [5.3] how a DMI-Log is actually designed.

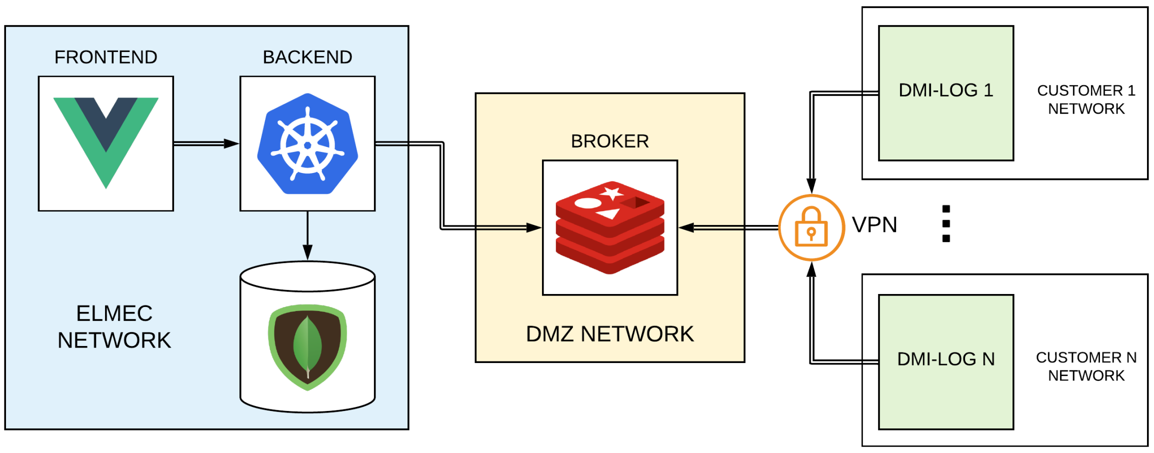


Figure 5.1: LLAMA high level architecture

5.2 Backend design

Here resides all the main logic of the architecture. The logic parts were split into different services deployed as containers on Kuberentes. This style is also known as Microservices architecture. This architectural style structures an application as a collection of services that are highly maintainable and testable, loosely coupled, independently deployable and organized around business capabilities. Containerized application are separately deployed with different resources allocation, scalability or replication configurations. On the back-end we have 4 main containers:

- **LLAMA API**: it contains the RESTful API implemented in Python using Flask framework with the Flask-RESTful extension introduced in the technology chapter

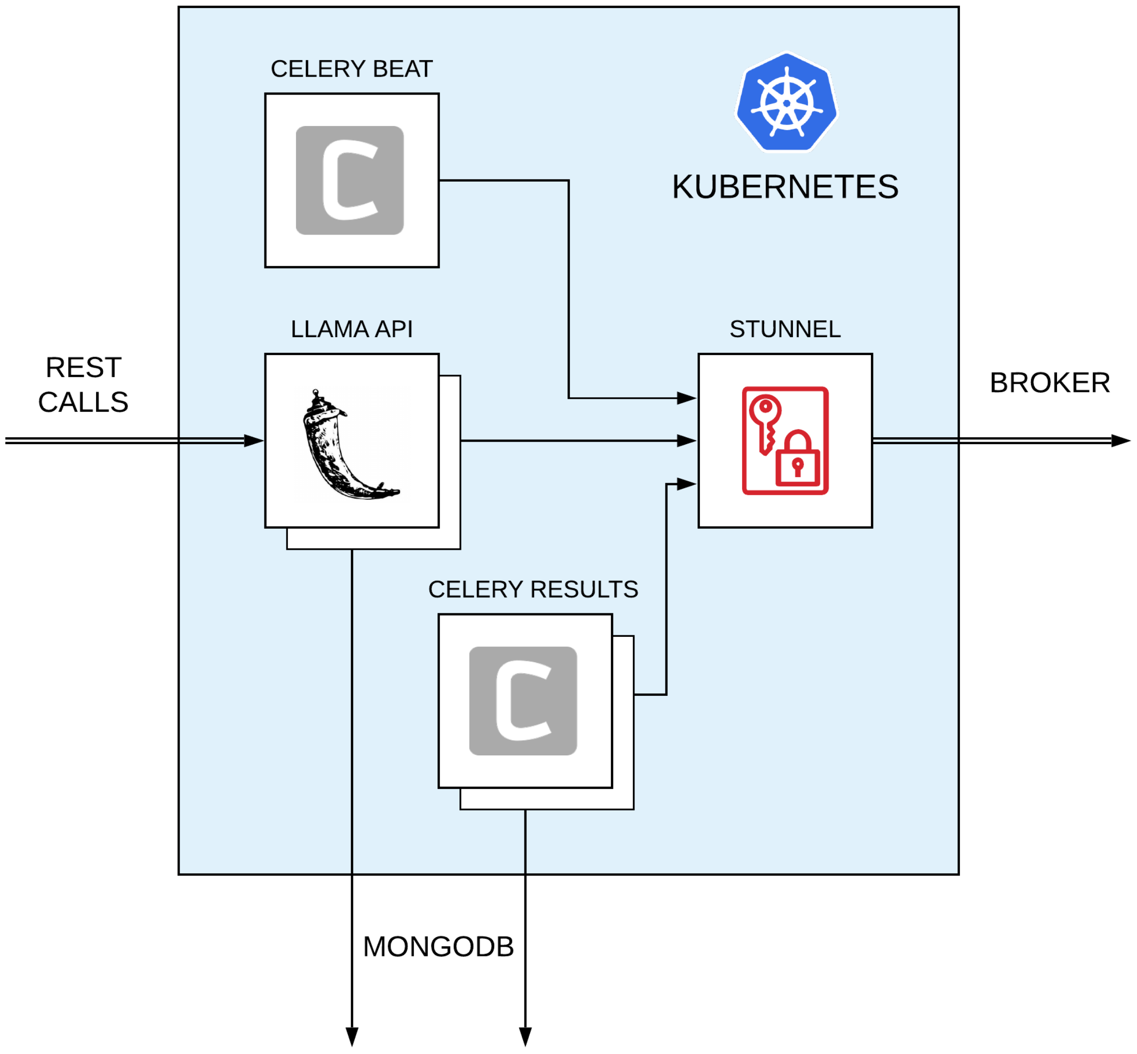


Figure 5.2: Backend design on Kubernetes

[4.2]. It exposes the port 5000 for incoming calls and communicates with both MongoDB cluster and Redis through Celery. A list of all the main endpoints is available in the next chapter [6.2].

- **CELERY RESULTS**: a lightweight Python container that runs a Celery worker listening on the ”llama_results” queue. After receiving queries’ results messages from Redis, it stores them into the central MongoDB proper database. I’ll go into details about Celery queues and workers implementation in the next chapter [6.1].

- **CELERY BEAT**: a Python container that simply runs an instance of Celery Beat daemon, the actual system scheduler. This actually relies on the RedBeat scheduler class to handle the task scheduling. As we introduced in the previous chapter, RedBeat allows to store schedules directly on Redis, and expose some Python methods to dynamically add or delete the scheduled tasks.

- **STUNNEL**: all the traffic going towards Redis, is actually pointing to this container that redirects the traffic to Redis after having encrypted it with the proper keys and certificates. I'll give some more information about this technology in the security section [5.5].

Each container is deployed as a Pod that automatically restarts containers in case of failure. Pods also replicates containers for load balancing, except for Celery Beat which it's not replicated because, since it's a scheduler, there is a potential risk of sending multiple schedules at the same time. Also, there is a Service that expose outside of Kuberentes the port 5000 of LLAMA API to make it reachable from the outside (for the frontend application).

5.3 DMI-LOG design and configuration

A DMI-LOG is the single node of the distributed architecture. It is the machine in charge of log collection, management and storage. A DMI-LOG consists on a Centos 7 Linux machine (both virtual or physical) configured to be both inside the customer network and a VPN that is connected to the MOM (the Redis instance). To efficiently deploy and configure this machine, we used Ansible, a technology we introduced in the chapter before [4.7]. After having installed a Centos 7 machine, it's possible to configure it using the proper tool on the internal company website. This tool basically launches an Ansible playbook that automatically installs Docker and a MongoDB Replica Set on the machine, besides other custom configurations. Then, it automatically pulls from a private registry the updated images of the containers and runs them.

There are 2 containers:

- **CELERY CUSTOMER**: a lightweight Python container that runs a Celery worker listening on the queue assigned to that customer (company client). Each customer has his message channel on Redis set up during machine automatic configuration. As for "CELERY RESULTS" container, I'll go into details about the actual Celery logic in the next chapter [6.1].

- **FLUENTD LLAMA**: this is a custom container created starting from the original available image of Fluentd published on Docker Hub. Besides official plug-ins installation, we also added our own custom plug-ins to support particular log formats, such as Cisco ASA codes, CEF and so on.

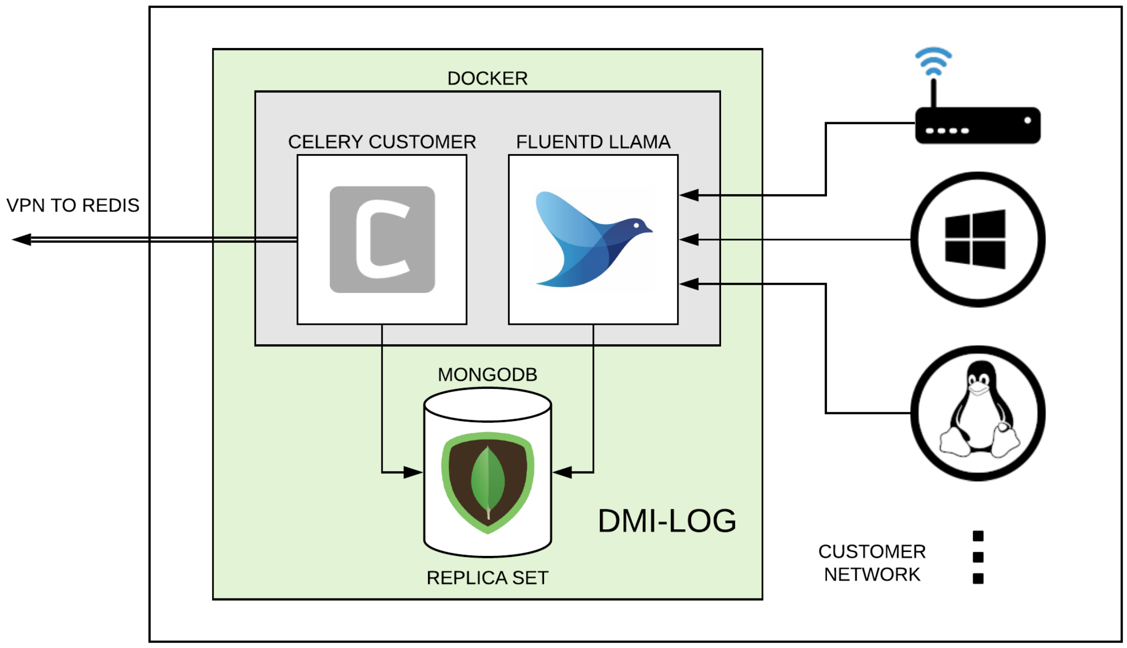


Figure 5.3: DMILOG design

Both these containers runs in host mode, that means they share the same network of the host machine, mapping their exposed port and listening ports on the host.

The Fluentd instance listens on specific configured ports. Each DMI-LOG could have its own configurations based on the needs of the customers, the technologies monitored and type of logs collected. For Linux and Windows log collection we use log shippers (such as Fluent bit, Filebeat or Winlogbeat based on the OSs), whereas network equipment (like routers, switches, IPSs, IDSs etc.) can usually forward logs (e.g. with Syslog) directly towards the relative ports. So, each port corresponds to a specific technology source and the relative traffic will be accordingly parsed and treated with the relative plug-in. Then the logs are inserted in their relative collection on the MongoDB ”logs” database (one collection for each source). Fluentd has a specific plug-in also to insert logs as documents in the MongoDB Replica Set. This is done very efficiently, in fact we reach thousands of inserts per second (in some cases we have up to 5000/second without issues). Fluentd can be nicely tuned to achieve great performances, for example thanks to the use of multiple threads at the same time. Usually we adopt 4 parallel threads for a standard instance.

For what concerns MongoDB, as I previously said it is configured as a Replica Set. By default, of course the Replica Set only contains the local instance of MongoDB that it's the primary one, but still there is the possibility to add more instances on other machines to create a proper cluster if needed, as our requirements asked [2.7]. MongoDB collections are configured to have an index on the always present timestamp field, to speed up all the queries and also to automatically set an expiration time after which the document is automatically deleted (usually after 6 months). The instance it's also configured to be accessed only with authorization. This means that only users registered by the admin during the set-up, having specific privileges, can access (read or write) the logs. The logical volume on which MongoDB saves the database is also encrypted. I'll go into more details about this and other security measurements we took in the next section.

5.4 Security

In this project the security aspect is fundamental. The data we handle is private and contains sensitive information about the client companies and their employees. As the CIA paradigm states, we need to provide confidentiality (information can be accessed only by authorized entities), integrity (information can be modified only by authorized entities and in the way they meant to) and availability (information must be available to all the parties who have a right to access it). To grant a secure enough system, we adopted various technologies, some of which we already cited before and some not.

5.4.1 Stunnel and TLS

Stunnel is a proxy designed to add TLS encryption functionality to existing clients and servers without any changes in the programs' code. Its architecture is optimized for security, portability, and scalability (including load-balancing), making it suitable for large deployments. Stunnel uses the OpenSSL library for cryptography, so it supports whatever cryptographic algorithms are compiled into the library. As shown in the previous sections, we used Stunnel to add TLS encryption to our backend. That's because in this way we don't need to add specific Redis configuration to support encryption. In this way we can also decouple this feature from the code and separately modify and deploy it as a portable container. The traffic going from the backend to the Redis instance in DMZ is securely encrypted. The Redis instance is configured to receive TLS incoming traffic. The traffic between Redis and the DMI-LOG nodes travels inside a VPN and is encrypted with TLS, fulfilling our requirements [2.12].

5.4.2 HTTPS and JWT

As we said before, "LLAMA" is a web tool available inside the internal company website. The website is secured through HTTPS and the outgoing API calls contains authorization tokens implemented with JWT, released from the official JWT server of the company. The tokens are then validated in the backend thanks to an ad-hoc library for Flask endpoint authentication that we developed from scratch. The name of this library is *flask-elmecauth.py* and allows to simply add authentication to Flask REST endpoints thanks to Python decorators. The decorator *@authenticated* simply placed before the endpoint declaration, wraps the action method and checks the validity of the JWT for the requested action before the actual execution, as our requirements asked [2.11]. The execution proceeds only if the issuer is valid and not the token is not expired.

5.4.3 Security for log storage

To fulfill our requirements [2.6], we adopted different solutions. As I showed in the previous section, the logs are stored on DMI-LOGs inside MongoDB collections. These collections resides in a database simply named "logs". Only 2 predefined and password secured users can interact with the database. The user ”fluentd”, which is used by the ”FLUENTD LLAMA” container, can insert documents and add new collections. Instead the user ”llama”, used by the container ”CELERY CUSTOMER”, can only read documents. The Celery task (that is a Python script) cannot hurt the system because, since it's executed inside a container, it cannot interact with other elements or file on the DMI-LOG machine, except for MongoDB on the port 27017 (with only read permission on logs). Moreover, the Celery worker runs with a Linux user which privileges are limited inside the container too.

To grant complete obfuscation of data for those who have are not accessing as MongoDB users, the whole database is saved on an encrypted LUKS partition. LUKS is the standard for Linux hard disk encryption. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passwords. LUKS stores all necessary setup information in the partition header, enabling to transport or migrate data seamlessly. To automate the process of retrieve a key for the encryption/decryption of the partition we adopted Clevis together with Tang. Clevis is a pluggable framework for automated decryption. It can be used to provide automated decryption of data or automated unlocking of LUKS volumes. Tang is a server implementation which provides cryptographic binding services without the need for an escrow. Clevis has full support for Tang. The whole process is well explained in the article reported [bib. [11]]. The server is hosted on separate machines and services. The LUKS volume is initialized during the DMI-LOG configuration with Ansible, before MongoDB installation.

Chapter 6

Implementation

In this chapter we will analyze some practical aspects of the actual implementation of the system. First of all we will see how we used Celery to implement asynchronous execution for our queries and the relative result collection, the most important logic of the system. Then we will make a list of the endpoints exposed by the API and finally we will show the website final UI, illustrating how to craft queries and interact with the system.

6.1 Celery tasks and workers for query execution on MongoDB

The overall idea of the query execution mechanism is to wrap a Python script (i.e. the query, implemented with PyMongo library) in a Celery task that will execute only on a specified target. The target uniquely identifies a company client on which we want to execute the query. The target is also the actual name of the Celery queue (and so the Redis channel). Only the Celery worker running on the target client is listening on this channel and will consume the execution. In this way, we can easily specify where to execute the query without conflicts. On the LLAMA webpage, the user actually has a Python editor and a target selector. The editor is where he can write the code to build queries to retrieve or analyze logs stored on the DMI-Log. I'll show in the last section of this chapter the UI of the webpage. After the query execution, the main task calls a new one that will execute only on the results worker running on "CELERY RESULTS" container in the backend, that will insert the result in the results database.

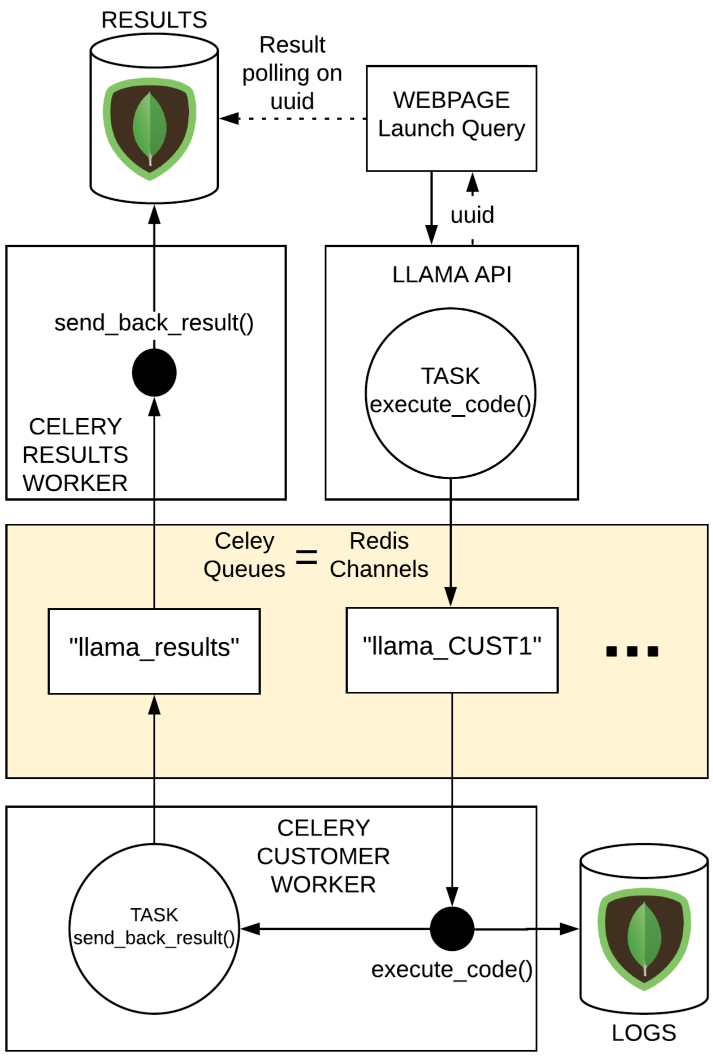


Figure 6.1: Celery tasks execution

So, there are 2 main Celery tasks:

- *execute_code()*: this task is launched with the *apply_async()* Celery method when a user submit a query execution. It contains the query written on the editor, that will execute on the remote console worker. This method take some parameters: "code" is the actual Python script, the query that we need to execute on the worker; "uuid" is the generated uuid that univocally identifies the task; "time_frame" contains the time frame specified in the frontend on which the user wants to execute the query.

These parameters are then passed to the Python *exec()* method. This method executes the dynamically created program, which in our case is a simple string. It creates a separate environment with its own process on which the code is executed. Finally the result is actually sent back calling *send_back_result()* having always as target the queue "llama_results".

- *send_back_result()*: As we just said this method is invoked always at the end of *execute_code()* to retrieve back the result. It is always executed by the results Celery worker that runs on the "CELERY RESULTS" container. This worker just consumes messages from "llama_results" queue. The task simply insert a JSON object, containing the result and some execution additional info, in the main MongoDB results database.

The system already provides the import of the most useful libraries such as "PyMongo", the official python distribution containing tools for working with MongoDB, or "Datetime", the main library to handle timestamps and dates. The system automatically imports also a set of additional methods to speed up the creation of queries. The method *llama_email()* allows to send email from a SMTP server. This can be used for scheduled queries to inform someone about some problems. For example, we can schedule a query to search a specific string that should not appear inside the logs. If the string appears, the task will send a notification email to the interested user. The idea is to implement the whole logic inside the task. We also implemented methods to rapidly build MongoDB aggregation pipelines [4.3]. The Aggregation Framework is the main tool used and suggested to make queries, since it's very powerful, but sometimes it takes a lot of lines to write some always present pipeline stage. So, the method *llama_time()* takes the "time_frame" object of the task and returns a formatted pipeline stage that will limit the execution only inside the time frame. Again, the method *llama_span()* returns a formatted pipeline stage to divide in time spans the space of the results, to group them maybe.

6.2 API endpoints

Here is a list of the main endpoints available on the RESTful API service.

6.2.1 /queries

ACTIONS: POST

This is the main endpoint used to launch a query. This endpoint is not inserting objects in our main backend database (MongoDB), instead these actions interacts with the class "LlamaController.py" that is in charge of handling the whole query execution and implements the Celery tasks invocation through the method "apply_async()". Basically, with the POST we submit a new request execution which is actually a message on Redis (the Celery task). Each time we create a new task, we associate it with a "uuid", a unique id, so we can monitor the task status on celery (with Flower for example), retrieve the relative result later or delete the execution.

6.2.2 /query/:uuid

ACTIONS: GET - DELETE

This is the main endpoint used to poll the result of the asynchronous query execution from the results collection in the database. We basically continuously look in the database if the task with our target uuid has returned its results. With the DELETE action we abort the execution, deleting the message on Redis from the relative channel and consequently stopping the consumption on the worker. This is always done thanks to the uuid.

6.2.3 /targets

ACTIONS: GET - POST

This endpoint is used to get the available targets (queues on which the DMI-LOGs are listening) and insert them in the database.

6.2.4 /scheduled-tasks

ACTIONS: GET - POST - DELETE

This endpoint is used to get the current scheduled tasks, delete them or add new ones. Insertion and deletion involve the invocation of the dedicated class ”BeatController.py” that exploits the RedBeat class to organize the schedules details directly on Redis storage. As said in the Design chapter, this class provides methods to save or remove tasks schedules at runtime on Redis without the needs of restarting the scheduler daemon.

6.2.5 /statistics

ACTIONS: GET

This endpoint is used to get general statistical information collected in the database about the status of the overall system, such as the number of queries, the scheduled queries executed etc.

6.2.6 /saved-queries

ACTIONS: GET - POST - DELETE

This endpoint is used to insert, delete or get favorite queries that the user want to save for future use and quick access.

6.3 Llama web page UI

In this last section I will introduce the LLAMA web page user interface. It is a single page tool, divided into 4 sub-pages. The first two pages are used to launch and schedules queries, the features we are mainly interested in. The Dashboard and Management page are currently under development and are not part of this project thesis.

6.3.1 Query page

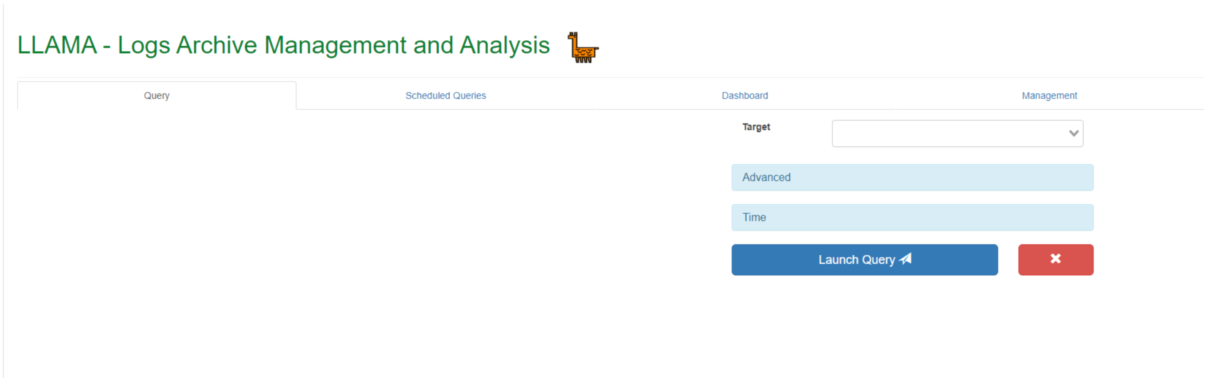
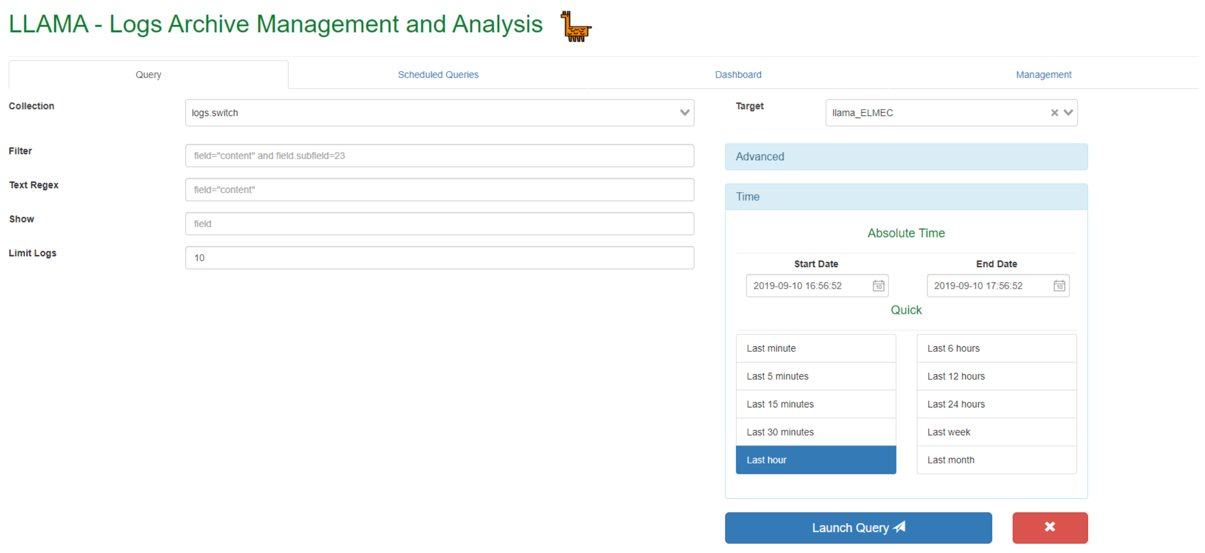


Figure 6.2: Query page

This is the main page of the tool, where you can craft a query and launch it [fig. 6.2] First of all, the user has to choose a target (that corresponds to the Celery queue). This will trigger automatically a query that will load the available collections on the relative customer. We can see in the example "llama_ELMEC" as target, which points to the DMI_LOG that collects logs inside the own company systems (the company is the customer of itself in this case) [fig. 6.3]. By default, the "Simple Mode" interface will show up after the collection names are retrieved.

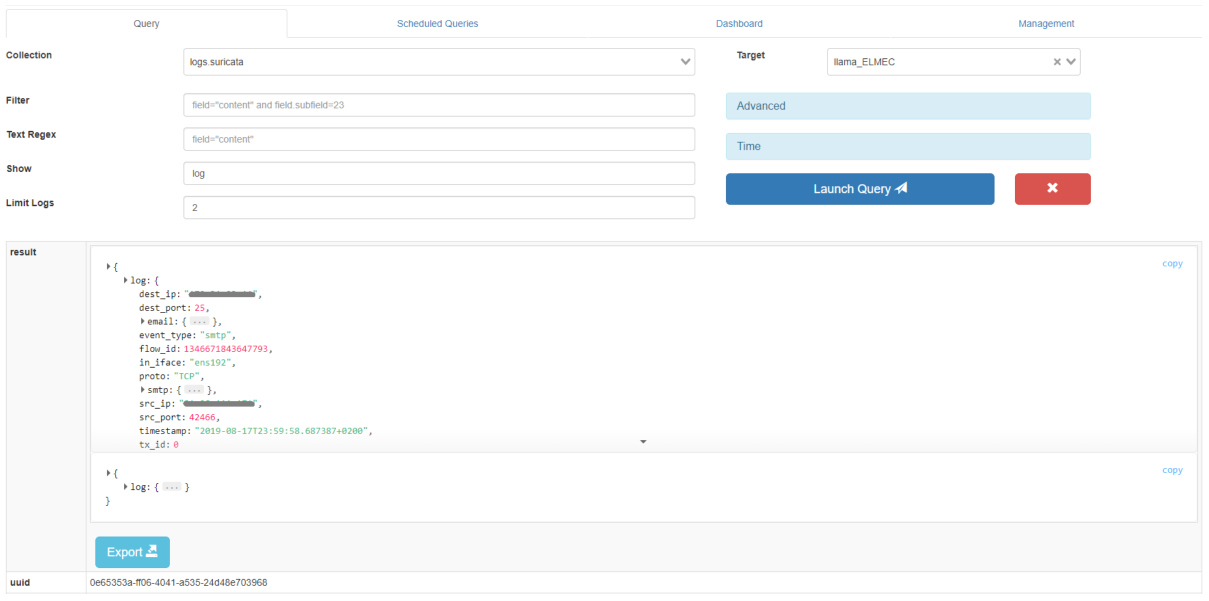
In this mode, we can make simple queries to just retrieve some logs and check their content. The user has some basic filter fields, which actually are a wrapper to the actual Python code. The auto generated code consists in a MongoDB aggregation pipeline. For example, the "Filter" consists in just a "$match" pipeline stage, the "Text Regex" exploits "$regex" pipeline stage, "Limit Logs" maps to "$limit" and so on. In the special area on the side, the user can specify the time frame on which we want to execute our query. It provides both an absolute time picker and a relative one.



Collection: logs.switch
Filter: field="content" and field.subfield=23
Text Regex: field="content"
Show: field
Limit Logs: 10
Target: llama_ELMEC
Start Date: 2019-09-10 16:56:52
End Date: 2019-09-10 17:56:52

Figure 6.3: Simple Mode

The "Launch Query" button just launch the asynchronous execution calling the "POST queries" endpoint, that will return the relative uuid and then it starts to poll over "GET query
:uuid". Clicking the red button will stop and abort both the asynchronous execution and the polling. This mode is intended to use to just check the log content, parsed structure and make simple queries.



Collection: logs.suricata

Filter: field="content" and field.subfield=23

Text Regex: field="content"

Show: log

Limit Logs: 2

Target: llama_ELMEC

result

```
{
  log: {
    dest_ip: "[redacted]",
    dest_port: 25,
    email: { ... },
    event_type: "smtp",
    flow_id: 1346671843647793,
    in_iface: "ens192",
    proto: "TCP",
    smtp: { ... },
    src_ip: "[redacted]",
    src_port: 42466,
    timestamp: "2019-08-17T23:59:58.687387+0200",
    tx_id: 0
```

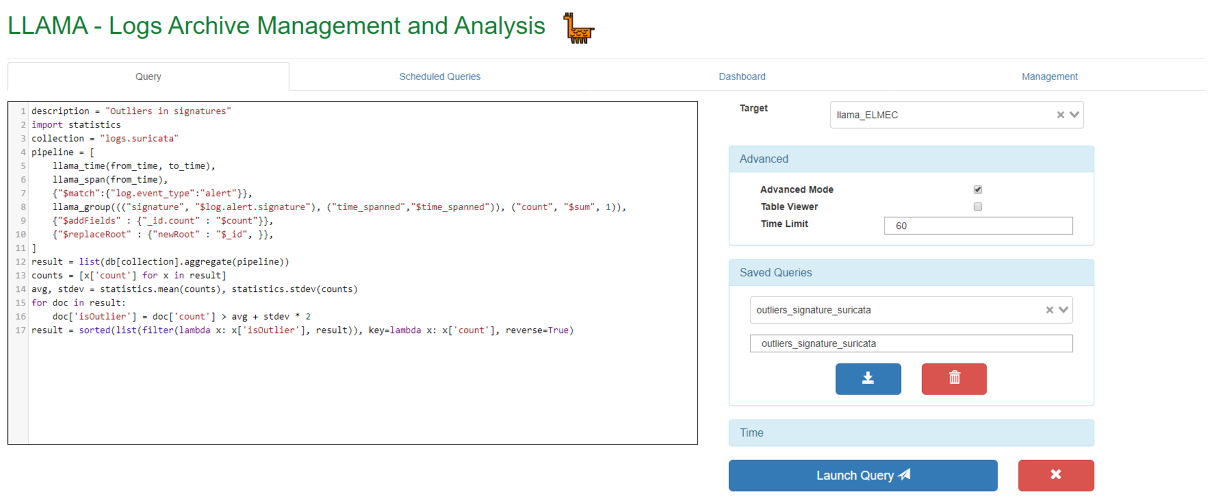
```
{
  log: { ... }
}
```

uuid 0e65353a-ff06-4041-a535-24d48e703968

Figure 6.4: Result JSON explorer

When the execution completes the result will show up below [fig. 6.4]. The user can navigate the result with the JSON explorer, expanding the single fields or the entire log. It is also possible to export the result in JSON with the relative ad-hoc button.

The access to the full features and possibilities of the tool, the user should activate the "Advanced Mode" in the proper area on the side. After the activation the simple interface is substituted with the integrated Python editor. This editor simply highlights Python syntax (it is implemented with a Javascript library called "codemirror").



```python
description = "Outliers in signatures"
import statistics
collection = "logs.suricata"
pipeline = [
    llama_time(from_time, to_time),
    llama_span(from_time),
    {"$match":{"log.event_type":"alert"}},
    llama_group((("signature", "$log.alert.signature"), ("time_spanned","$time_spanned")), ("count", "$sum", 1)),
    {"$addFields" : {"_id.count" : "$count"}},
    {"$replaceRoot" : {"newRoot" : "$_id", }},
]
result = list(db[collection].aggregate(pipeline))
counts = [x['count'] for x in result]
avg, stdev = statistics.mean(counts), statistics.stdev(counts)
for doc in result:
    doc['isOutlier'] = doc['count'] > avg + stdev * 2
result = sorted(list(filter(lambda x: x['isOutlier'], result)), key=lambda x: x['count'], reverse=True)
```

Target: llama_ELMEC

Advanced Mode ✓

Table Viewer ☐

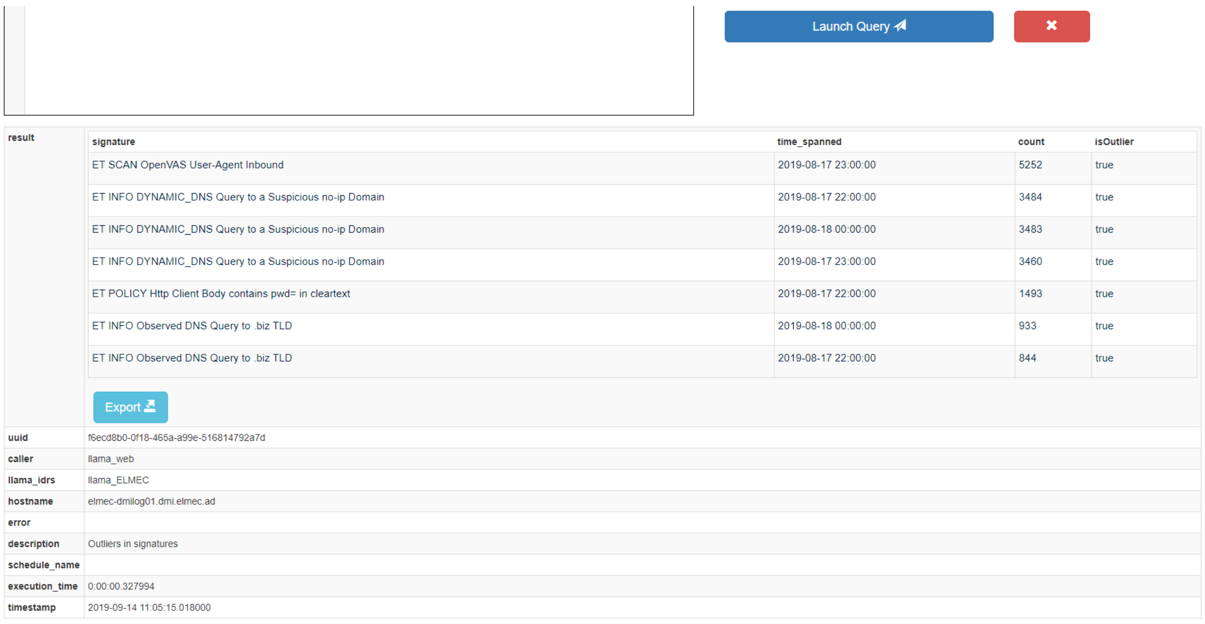
Time Limit: 60

Saved Queries: outliers_signature_suricata

outliers_signature_suricata

Figure 6.5: Advanced Mode

Here the user can write complex queries, retrieve logs or return just the results of complex analysis. In the example [fig. 6.5], there is a complex query that exploits both MongoDB aggregation framework and Python statistics library. The query was loaded using the "Saved Queries" dedicated area. In the specific case, the goal of this query is to find outliers in signatures of Suricata (and IDS software) logs. This means that when the statistical values of the matched signatures surpasses a certain threshold, the values are returned as result. The "result" and "description" fields of the result actually contains the value stored in the relative variable inside the query. So this two variables are assigned to this scope. Other special variables are "from_time" and "to_time" that will contain the value specified on the Time picker area on the side.



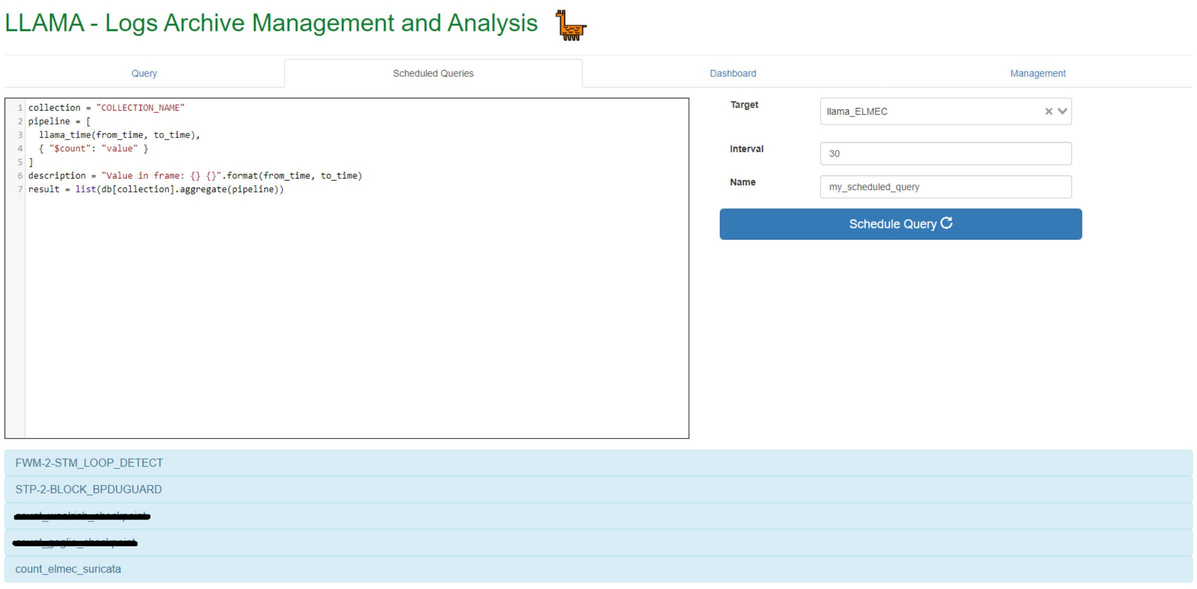
result	**signature**	**time_spanned**	**count**	**isOutlier**
	ET SCAN OpenVAS User-Agent Inbound	2019-08-17 23:00:00	5252	true
	ET INFO DYNAMIC_DNS Query to a Suspicious no-ip Domain	2019-08-17 22:00:00	3484	true
	ET INFO DYNAMIC_DNS Query to a Suspicious no-ip Domain	2019-08-18 00:00:00	3483	true
	ET INFO DYNAMIC_DNS Query to a Suspicious no-ip Domain	2019-08-17 23:00:00	3460	true
	ET POLICY Http Client Body contains pwd= in cleartext	2019-08-17 22:00:00	1493	true
	ET INFO Observed DNS Query to .biz TLD	2019-08-18 00:00:00	933	true
	ET INFO Observed DNS Query to .biz TLD	2019-08-17 22:00:00	844	true
uuid	f6ecd8b0-0f18-465a-a99e-516814792a7d			
caller	llama_web			
llama_idrs	llama_ELMEC			
hostname	elmec-dmilog01.dmi.elmec.ad			
error				
description	Outliers in signatures			
schedule_name				
execution_time	0:00:00.327994			
timestamp	2019-09-14 11:05:15.018000			

Figure 6.6: Query complete result with Table Viewer

We can see the result of this query [fig. 6.6] . In this case the "Table Viewer" feature is also activated. This will show the results in form of a table, instead of with the JSON navigator tool. The result can be exported as usual. Besides result and description fields, in this figure we can also see all the other fields present in the returned JSON, such as "uuid", "execution_time", "timestamp" etc.

6.3.2 Scheduled Queries page

This page contains another Python editor instance to write a query [fig. 6.7]. The difference is that here you can set-up a scheduled one. A scheduled query will be executed on the specified target on a regular interval. The interval can be specified (in minutes) in the dedicated input section. The user can also input a name for the schedule (the name should be unique). Clicking the "Schedule Query" button the relative endpoint will be called, inserting with RedBeat the schedule inside Redis. The Celery Beat scheduler will immediately start to execute the task, waiting for an "interval" amount of minutes between each execution.



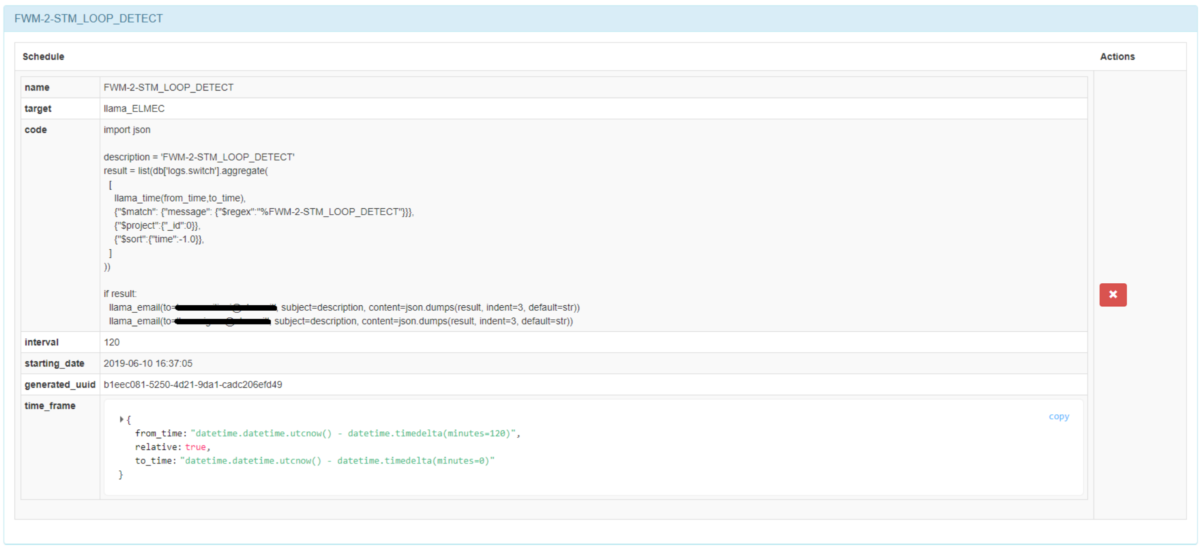
```python
collection = "COLLECTION_NAME"
pipeline = [
  llama_time(from_time, to_time),
  { "$count": "value" }
]
description = "Value in frame: {} {}".format(from_time, to_time)
result = list(db[collection].aggregate(pipeline))
```

Target: llama_ELMEC
Interval: 30
Name: my_scheduled_query

FWM-2-STM_LOOP_DETECT
STP-2-BLOCK_BPDUGUARD
[redacted]
[redacted]
count_elmec_suricata

Figure 6.7: Scheduled Queries page

Below the Scheduled Queries editor, there is a list of the running schedules. Schedules can be expanded, consulted and deleted with the dedicated button. In the example [fig. 6.8], we have a query that searches with a regex for a specific string inside a collection where we store the logs of many switches. This string identifies a loop inside the network of switches. If one or more strings are found, they are inserted in the result. Moreover, the result is also sent via email to a couple of addresses of people in charge of solving this type of problems. The query is executed from the scheduler each 120 minute, hence 2 hours.


FWM-2-STM_LOOP_DETECT

name FWM-2-STM_LOOP_DETECT

target llama_ELMEC

code

```
import json

description = 'FWM-2-STM_LOOP_DETECT'
result = list(db['logs.switch'].aggregate(
  [
    llama_time(from_time,to_time),
    {"$match": {"message": {"$regex":"%FWM-2-STM_LOOP_DETECT"}}},
    {"$project":{"_id":0}},
    {"$sort":{"time":-1.0}},
  ]
))

if result:
  llama_email(to=[redacted], subject=description, content=json.dumps(result, indent=3, default=str))
  llama_email(to=[redacted], subject=description, content=json.dumps(result, indent=3, default=str))
```

interval 120

starting_date 2019-06-10 16:37:05

generated_uuid b1eec081-5250-4d21-9da1-cadc206efd49

time_frame

```
{
  from_time: "datetime.datetime.utcnow() - datetime.timedelta(minutes=120)",
  relative: true,
  to_time: "datetime.datetime.utcnow() - datetime.timedelta(minutes=0)"
}
```

Figure 6.8: Scheduled query example

Chapter 7

Evaluation

In this chapter I will try to evaluate the overall results, proceeding for each of our requirements.

7.1 Central managed distributed infrastructure for log management

As required in section 2.1, the system actually consists in a central web page tool, that communicates with the backend system and the distributed consoles, called DMI-LOGs, which are in charge of log collection and management. The whole infrastructure is realised with Python. Flask is the Python framework adopted for the API and Celery is the Redis client that handle the query remote execution mechanism and result retrieval.

7.2 Node isolation

This requirement 2.2 is achieved using separate queues for each customer. Moreover, Celery workers are configured with "–without-mingle" and "–without-gossip" options that block the workers from logging events happening on other workers, completely isolating the container on its console.

7.3 JSON as unique representation of data

As required in section 2.3, the system always use JSON as data representation and communication format. Fluentd forces all the incoming logs to become JSON and stores them in MongoDB which saves documents in JSON again (actually BSON as we explained in the Technology chapter [4.3]). Celery encodes it's message in JSON over Redis, and the results of the queries are again saved in a MongoDB central database. The communication with the API is RESTful, so exploiting one more time JSON format. The website shows results with a JSON explorer that allow to dynamically navigate inside JSON fields.

7.4 Efficient and reliable log management for each node

Thanks to Fluentd and MongoDB, also this requirement 2.4 is fulfilled. We have seen how Fluentd efficiently collects thousands of logs per second (even 5000+/second), parse them with relative plug-in (our custom ones too) and store them inside MongoDB, which has stunning insertion performances and search speed as well, thanks to its indexes.

7.5 Log retention for a specific amount of time or space

As required in section 2.5, MongoDB grants safe document retention and data persistence. Moreover, thanks to its indexes it can automatically delete documents after a specific amount of time. With capped collections it can also deletes documents when a specific threshold of space is reached.

7.6 Encrypted and protected storage for logs

As we have seen in details in the dedicated section [5.4.3], this requirement is satisfied too. We adopted a set of technologies to encrypt the partition on which MongoDB store the database (LUKS, Clevis and Tang). Moreover, the access to the logs database with MongoDB API is only possible with two pre-defined users protected by password ("fluentd" for insertion and "llama" for read). These users accesses are only given to Fluentd and "CELERY CUSTOMER" container. This fulfills our requirement 2.6.

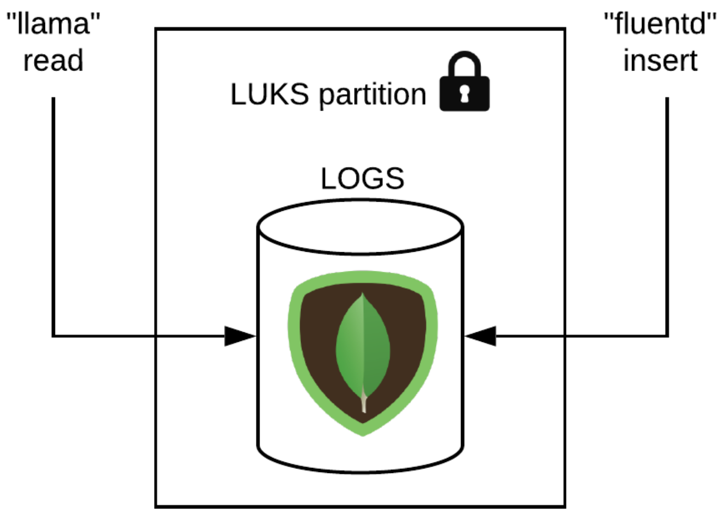


Figure 7.1: LUKS partition and users privileges

7.7 Replicated data storage for logs

As required in section 2.7, the system supports replicated data storage thanks to MongoDB Replica Set. By default it is configured as a single node Replica Set with only one Primary node, but machines could be added to provide fault-tolerance or even better read/write performances. To increase performances also data partitioning is available with MongoDB Sharding.

7.8 Easy deployable nodes and backend infrastructure

As required in section 2.8 the overall infrastructure is simple to deploy and update. For the backend infrastructure, on the git project we have a continuous integration (CI) mech-

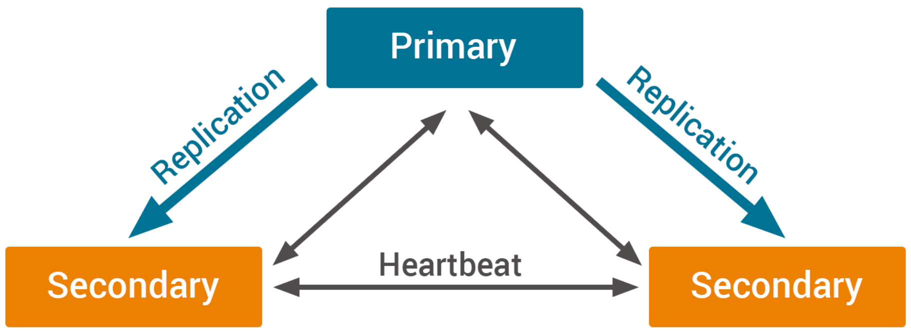


Figure 7.2: MongoDB Replica Set schema

anism to automatically launch Kubernetes deployments for each new version released. For the distributed infrastructure instead, we use Ansible to automatically deploy DMI-LOGs and install all the software and configuration needed. We also have playbooks to only update containers, or other components.

7.9 One website to launch queries on all the nodes

As required in section 2.9, there is a website to launch the queries on the distributed infrastructure. The website allows the users to:

1. Consult logs with a simple interface
2. Create sophisticated queries for log analysis with an advanced interface
3. Launch query asynchronously and consult results
4. Schedule queries for repeated analysis or checks

More details about the UI can be found in section 6.3.

7.10 Query scheduling system

This requirement 2.10 is achieved thanks to Celery Beat system. The container keeps executing the scheduled queries on the input interval. The scheduled queries could be dynamically inserted and deleted from the website thanks to the RedBeat Scheduler class.

7.11 Authenticated endpoints and system access

As required in section 2.11, all the API endpoints are secured with a proper Python library we implemented. The library wraps all the calls with a Python decorator which checks the JWT inside each call and grants access only to incoming call with valid JWTs.

7.12 TLS and VPN secured communication

As required in section 2.12, the communication in the overall system is secured with TLS encryption thanks to Stunnel technology. The connection between Redis and the distributed infrastructure is also protected with a VPN since it travels on internet.

Chapter 8

Conclusion

8.1 Summary

The idea of project LLAMA was to create a complete system able to provide log management and analysis for multiple distributed clients. We have seen that is possible to provide a solution for log management without paying for commercial software, but also that we can create a distributed system to separately store logs on different machines for each distinct client, still launching and orchestrating queries from a central system. The DMI-Log distributed architecture is completely independent from the query system for log collection and management. The centralized components are only in charge of manage the query execution and scheduling over the distributed system. The major advantage of having developed our own software for the frontend and backend parts of the central tool, is that we can continually add new features as needed and expand it as we want. Moreover, since the system is loosely coupled and fragmented in small pieces and services, we can substitute each component as we wish. For example, another choice could also have been to select Apache Kafka instead of Redis for message distribution. The major advantage of Kafka could have been greater performances and stability. Thanks to its distributed nature, it provides the possibility of deploying a huge cluster to handle a greater amount of traffic (in our case of queries). A very good comparison between those technologies is available in this article [bib. [12]] In our case, we opted for Redis because we were more interested in Python integration through Celery with its task execution logic, instead of having particular stunning performances or availability, but with the cost of a much more complex implementation of a logic for the query system.

8.2 Future work

I'm glad to say that LLAMA as already been adopted for some clients of the company, first of all by Elmec Informatica itself. The system is locally used to store hundreds of gigabytes of logs with the possibility of fast consultation and even complex analysis without any external cost. The project is still growing and in case of an increasing number of clients it's possible that the architecture will change. First of all, the actual single Redis instance is not scalable and stable enough to handle a huge number of queries. In the past we tried to extend it to a cluster with Redis Sentinel. Redis Sentinel provides high availability for Redis, creating a Redis deployment that resists to certain kind of failures. It also provides other collateral tasks such as monitoring, notifications and acts as a configuration provider for clients. Unfortunately, we had some issue in the integration of Sentinel in the system, so we switched back to a single Redis instance. Anyway, there is another new solution for Redis clustering, the project Redis Cluster. In future, we will probably study in more details this technology to consider its adoption. Alternatively, we could also think to switch to Apache Kafka in case it is needed. There is also on Celery's GitHub repository a pull request to add Kafka to the compatible brokers.

Finally, we are currently working on adding new features and pages to the website interface and new API endpoints to interact with the system. For example, the Dashboard page to visualize results, system health and performances or the Management page to centrally manage some aspect and configuration on the DMI-Log machines.

Bibliography

[1] J. Kreps. (2014). Benchmarking apache kafka: 2 million writes per second (on three cheap machines), [Online]. Available: `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`.

[2] H. Zhang. (2014). Efficient in-memory data management: An analysis, [Online]. Available: `https://www.comp.nus.edu.sg/~logbase/pdfs/vldb14_inmemorystudy.pdf`.

[3] (). Celery project repository on github, [Online]. Available: `https://github.com/celery/celery`.

[4] (). Redbeat project repository on github, [Online]. Available: `https://github.com/sibson/redbeat`.

[5] M. Sibson. (2017). Hello redbeat: A celery beat scheduler, [Online]. Available: `https://blog.heroku.com/redbeat-celery-beat-scheduler`.

[6] (). Flower project repository on github, [Online]. Available: `https://github.com/mher/flower`.

[7] (). Redis commander project repository on github, [Online]. Available: `https://github.com/joeferner/redis-commander`.

[8] (). Fluentd project repository on github, [Online]. Available: `https://github.com/fluent/fluentd`.

[9] A. Kamsky. (2015). Performance testing mongodb 3.0 part 1: Throughput improvements measured ì with ycsb, [Online]. Available: `https://www.mongodb.com/blog/`

post/performance-testing-mongodb-30-part-1-throughput-improvements-measured-ycsb.

[10] (). Ansible project repository on github, [Online]. Available: https://github.com/ansible/ansible.

[11] T. Scherf. (2018). Automatic data encryption and decryption with clevis and tang, [Online]. Available: http://www.admin-magazine.com/Archive/2018/43/Automatic-data-encryption-and-decryption-with-Clevis-and-Tang.

[12] A. Yigal. (2016). Kafka vs. redis: Log aggregation capabilities and performance, [Online]. Available: https://logz.io/blog/kafka-vs-redis/.