POLITECNICO DI TORINO
Master's Degree in Computer Engineering

Exploring the OCSF Framework in AWS: Design, Implementation and Performance Analysis of a Security Lake Platform

Supervisor
Prof. Fulvio Risso

Candidate
Stefano Gianola

Company Supervisor
Francesco Benforte

Academic Year 2023/2024

Abstract

In the cybersecurity world, identifying and contrasting cyber attacks necessitates the synergistic deployment of diverse tools. These tools generate streams of alerts and isolated data, with different log formats and data schema, often demanding manual correlation for comprehensive analysis and response. The Splunk State of Security 2023 report [1] underscores that 64% of Security Operations Center (SOC) teams face challenges transitioning between security tools due to limited integration. The collected data cannot be seamlessly combined, hindering the ability to obtain a holistic view of the security environment. Cybersecurity teams find themselves dedicating significant time and effort to manually normalize data across diverse tools. This manual effort detracts from their primary focus on detecting, investigating, and responding to security events. In essence, the data manipulation and normalization process becomes a bottleneck, impeding the efficiency of security operations.

The Open Cybersecurity Schema Framework (OCSF), unveiled during the BlackHat conference in August 2022, represents a groundbreaking initiative in the realm of cybersecurity. It is designed as an open schema standard with the key objective of offering a straightforward taxonomy that transcends supplier-specific constraints. This framework is deliberately vendor-agnostic, allowing it to be seamlessly integrated into any environment, embraced by any application, and adopted by diverse solution providers. This solution allows security teams to accelerate and streamline the data entry and analysis process, along with correlating data, without requiring time-intensive upfront standardization efforts.

In November 2022, Amazon Web Service (AWS) introduced Amazon Security Lake, a service that leverages OCSF as its foundational data schema. It is a security data lake and helps to consolidate security-related information from various sources, including AWS environments, SaaS providers, on-premises infrastructure, cloud platforms, and third-party providers. The primary function of Security Lake is to facilitate in-depth analysis of security data, empowering organizations to gain a holistic view of their security landscape across the entire enterprise. By leveraging Security Lake, users can enhance the safeguarding of their workloads, applications, and data, thereby fortifying the overall security posture of their organization.

The objective of the project is to leverage the OCSF standard and to design the architecture of a platform that is able to ingest and normalize security logs in OCSF format, integrate logs and events from external sources in the Security Lake, and develop an implementation to automate the creation and configuration of such a platform. The thesis also analyzes the scalability of the created application and measures the ability of the system to ingest data, highlighting any constraints encountered and potential alternative solutions to improve scalability.

To my family

Acknowledgment

At first, I would like to express my gratitude towards Professor Fulvio Risso for providing me with the opportunity to work on this thesis.

I am also immensely thankful to Storm Reply for warmly welcoming me and, in particular, to Francesco Benforte for supervising and guiding me throughout my thesis period.

Lastly, I would like to thank all of my friends who have supported me during my years of study.

Table of Contents

Listings

List of Figures

Acronyms

SOC: Security Operations Center
OCSF: Open Cyber Security Framework
ICD: Integrated Cyber Defense
IaC: Infrastructure as Code
AWS: Amazon Web Services
CI/CD: Continuous Integration and Continuous Deployment
HCL: HashiCorp Configuration Language
CNCF: Cloud Native Computing Foundation
AZ: Availability Zone
AZs: Availability Zones
VPC: Amazon Virtual Private Cloud
IGW: Internet Gateway
NAT: Network Address Translation
NACLs: Network Access Control Lists
IAM: AWS Identity and Access Management
CLI: Command Line Interface
ECR: Amazon Elastic Container Registry
EKS: Amazon Elastic Kubernetes Service
S3: Amazon Simple Storage Service
ACLs: Access Control Lists
KMS: Key Management Service
I/O: Input/Output
ASGs: Auto Scaling Groups
ASFF: AWS Security Finding Format
IAM: Identity and Access Management

Chapter 1

Introduction

In the cybersecurity world, identifying and contrasting cyber attacks necessitates the synergistic deployment of diverse tools. These tools generate streams of alerts and isolated data, with different log formats and data schema, often demanding manual correlation for comprehensive analysis and response. Splunk State of Security 2023 highlights that 64% of SOC teams struggle to pivot from one security tool to the next, with little integration between the tools to make it easier [1]. Cybersecurity teams face challenges in getting a complete view of the security environment due to difficulties in combining data collected from different tools. This obstacle requires significant manual effort to normalize data, diverting attention from critical tasks such as detection, investigation, and response to security events. As a result, simplifying data integration processes is critical to improving operational efficiency and strengthening cyber defense capabilities.

The Open Cybersecurity Schema Framework (OCSF), presented at the Black Hat conference in August 2022, represents an innovative initiative in the field of cybersecurity. It was designed as an open schema standard with the key goal of offering a simple taxonomy. This framework is vendor-agnostic, meaning security providers and data producers can adopt and map their existing schemes in OCSF. This solution allows security teams to accelerate and simplify the data entry and analysis process, along with data correlation, without requiring time-consuming initial standardization efforts.

In November 2022, AWS launched Amazon Security Lake, utilizing OCSF as its core data schema. This service acts as a centralized repository for security data, consolidating information from diverse sources. It enables comprehensive analysis, granting organizations a unified perspective of their security environment. Through Security Lake, users can enhance the protection of their assets, applications, and data, strengthening their overall security stance.

1.1 Goal of the thesis

After studying and exploring the OCSF, the thesis project aims to expand the functionality of the Security Lake service. The Security Lake enables data integration from vendors already using OCSF, AWS services predisposed to integration, and third-party sources. However, the data from these latest sources have their own schema, which must be manually mapped to the OCSF schema for integration into the Security Lake.

The goal of the thesis is to develop a platform that simplifies and accelerates the integration of logs from tools that still use their own schema into the Security Lake. The platform must automate the data collection, normalization of messages from the original format to the OCSF schema, and loading of data into the Security Lake.

The platform should be easily configurable, customizable, and automated to minimize manual configuration effort and optimize implementation. It should provide a repository with a pre-set configuration file where the integrator can specify the integrations they want to perform and indicate the mapping required for each integration to transform the original format into the OCSF schema. Based on the configuration file, the platform will automatically create the necessary infrastructure for all specified integrations. It is essential that the platform is scalable in relation to the volume of incoming traffic, is reliable, and is always available.

It also analyzes the scalability of the application created and measures the ability of the system to ingest data, highlighting any constraints encountered and potential alternative solutions to improve scalability and the ability of the platform to ingest data.

Chapter 2

OCSF

The Open Cybersecurity Schema Framework (OCSF) project represents a significant collaboration between cybersecurity industry leaders, spearheaded by AWS and Splunk. Built based on work done by ICD Schema in Symantec, a division of Broadcom, this initiative has evolved through close engagement with customers and a meticulous analysis of the evolving needs within the security operations market. In the beginning, OCSF boasted the participation of 18 founding members, a consortium of technology and security organizations. This collaborative effort signals a unified commitment to advancing the field of cybersecurity. The coalition comprises AWS, Broadcom Cloudflare, CrowdStrike, DTEX, IBM Security, IronNet, JupiterOne, Okta, Palo Alto Networks, Rapid7, Salesforce, Securonix, Splunk, Sumo Logic, Tanium, Trend Micro, and Zscaler. The combined wealth of experience and expertise from these pioneering organizations culminated in the unveiling of OCSF at the renowned Black Hat USA event in Las Vegas on August 10, 2022. This announcement marked a pivotal moment in the ongoing research, outlining a guideline for cybersecurity. After a year since the announcement has more than 145 organizations and 435 individual participants. OCSF is a vendor-agnostic scheme: security providers and data producers can adopt and map their existing schemes into OCSF. OCSF can be used by any cybersecurity team, in any organization, for any solution. OCSF is an open-source project and is an extensible framework for schematic development: new attributes, objects, categories, profiles, and event classes can be defined.

2.1 OCSF Taxonomy

The OCSF taxonomy encompasses various constructs, each tailored to a specific level of granularity. These constructs range from defining the kind of value to more generalized groupings of events sharing common characteristics.

Key constructs within the OCSF taxonomy include:

- **Data Types**: These define the nature or kind of data that a particular value can assume.
- **Attribute**: An attribute is a unique and identifiable name linked to a specific data type.

- **Event Class**: This is a collection of attributes that together describe an event.
- **Category**: Categories serve as containers for events sharing a common domain. They facilitate the grouping of events with similar characteristics.
- **Profile**: Profiles introduce additional, context-specific attributes to event classes and objects. They enhance the descriptive capabilities of the schema by overlaying supplementary information.
- **Extension**: The extension construct allows for the expansion of the schema without making modifications to the core structure. This flexibility enables adaptation and growth without disrupting the foundational elements.

In summary, the OCSF taxonomy employs a hierarchical arrangement of constructs, ranging from specifying data types and attributes to organizing events into classes, categories, and profiles. The extension mechanism further ensures adaptability and scalability without compromising the integrity of the core schema.

2.2 Data Types

The term Data Type indicates the specific nature of the value associated with a variable, and these types can be of different kinds: scalar or complex.

Scalar data, which forms the basic building blocks, is crafted from Base Types like strings, integers, floating-point numbers, and booleans.

In the OCSF, complex data types are called objects. An object serves as a cohesive assembly of attributes that are logically connected, typically representing a distinct entity. These attributes encapsulate relevant information about the entity, and an object may also incorporate or reference other objects, creating a hierarchical or interconnected structure.

2.3 Attributes

An attribute is essentially a distinct and identifiable name linked to a specific data type. In the OCSF schema, all attributes that can be used are present in the Attribute Dictionary. This dictionary serves as a complete list, detailing all available attributes in the schema. You can compare attributes to essential bricks, and together they build the entire OCSF scheme. The attribute dictionary consists of a structured table with five columns.

The first column, known as “Caption”, serves as a user-friendly name for each attribute. This makes the attributes easily understandable for users who may not be deeply familiar with the technical intricacies.

Moving on to the “Name” column, it contains unique identifiers for each attribute. The names are crafted with specific suffixes that convey additional meaning. The “_uid” suffix signifies an attribute’s uniqueness within the OCSF schema, often coupled with a friendly companion bearing the “_name” suffix. Together, they establish a link between a distinct identifier and a more human-readable name. This symbiotic relationship is exemplified in the Base Event Class with attributes like “category_uid”, “category_name”, “type_uid”, and “type_name”. Another suffix is “_uuid”, denoting a globally unique value. An instance of this can be found in the Session object, where the “uuid” attribute holds the universally unique identifier for a session. Attributes, that finish with “s”, are indicative of an array. The “_id” suffix in the Attribute Dictionary signifies an Enum integer value, where each integer is linked to a corresponding label. It operates in tandem with an attribute sharing the same name minus the suffix. This associated attribute steps in when there is no fitting label in the Enum or to reveal the name of the Enum label. In Enums, two default values are typically present: 0 for “Unknown” and 99 for “Other”. If you opt for the value 99, indicating “Other”, it is essential to populate the associated attribute. “_ids” signals an array of Enum attributes, introducing a collective dimension to Enum structures. Attributes like “_ip,” “_info”, “_detail”, “_time”, “_time_dt”, “_process”, and “_version” each contributes a unique characteristic to the schema. “_ip” refers to an IP address, “_info” encapsulates a value containing information, while “_detail” holds additional information. “_time” and “_time_dt” both relate to time, with the former in milliseconds since the Epoch 01/01/1970 00:00:00 UTC and the latter following the RFC-3339 format. “_process” contains information about a process, and “_version” indicates a specific version.

The “Type” column is where the fundamental nature of the attribute is defined, whether it is a basic, scalar, or complex data type. This sheds light on how the attribute is expected to behave within the schema.

In the “Referenced By” column, we find a valuable list of Class Events and Objects where each attribute can be applied. This linking ensures that attributes are utilized appropriately across various elements of the OCSF schema.

Lastly, the “Description” column offers a brief narrative about each attribute. This narrative serves as a guide, providing users with insights into the purpose and functionality of each attribute.

2.4 Event Class

Event Classes are a set of attributes that together describe and represent a specific event at a precise time. The objective of the schema is to allow the mapping of any raw event into a single class of events.

Within the OCSF schema, all Event Classes derive from the Base Event Class. This foundational class comprises a set of attributes that are common across most event classes. Specific attributes are then incorporated into each class to provide a more detailed description of the event's context. Furthermore, the Base Event Class can function as a foundation for creating new Event Classes or for mapping generic events that don't align with any specific event class.

The Base Event Class encompasses attributes found in the Attribute Dictionary. Apart from the typical Caption, Name, Type, and Description fields derived from the Attribute Dictionary, the Group and Requirement columns are introduced to enhance the overall understanding of the event attributes.

2.4.1 Group

Attributes are categorized for documentation purposes into four groups: Classification, Occurrence, Primary, and Context. The first two groups, Classification and Occurrence, are universal and not specific to any particular event class. On the other hand, Primary and Context groups are tailored to a specific event class.

Classification attributes play a crucial role in the framework's taxonomy and are designated as Required within the Base Event class. Examples are "activity_id", and "class_id" attributes.

Occurrence attributes are linked to time-related aspects and can be required, recommended, and optional based on their relevance to a particular event class. Instances are "duration" and "time".

Primary attributes signify the core semantics of an event across all use cases. They are typically marked as Required or Recommended, depending on their importance within each event class. Primary attributes in the Base Event class are applicable to all event classes. Examples are "observables" and "status" attributes.

Context attributes, on the other hand, contribute to variations in typical use cases, adding depth or nuance to the event's meaning. These attributes may have different requirement levels but are commonly marked as Optional. “metadata” and “enrichments” are attributes that context is their group.

2.4.2 Requirement

In event classes, attributes play a crucial role, with each assigned a requirement flag based on the semantics of the event class. These flags are available in three distinct types, each of which outlines a specific need in the context of the event.

Firstly, we have “Required” attributes. These are considered indispensable, constituting essential elements that must be present within the Class Event to ensure a comprehensive understanding of a particular occurrence. In instances where these attributes cannot be filled in, a default value, typically denoted as “Unknown”, is employed to maintain a baseline of information.

Next, there are “Recommended” attributes. These attributes bring certainly advantage for a more comprehensive understanding of a given event but are not obligatory additions. Their inclusion is suggested due to their potential usefulness, providing additional layers of information that can enhance the overall context. However, it is important to note that there might be instances where these attributes cannot be filled in.

Finally, we encounter “Optional” attributes. These attributes, while not imperative for basic event comprehension, serve to enrich the contextual understanding or facilitate the mapping of data with a higher information density. Their inclusion is discretionary, allowing for a more nuanced description of the event.

2.4.3 Base Event Class

The Base Event Class embodies a dual nature that perfectly integrates generality and concreteness. This fundamental construct not only represents a generic archetype of an event but also concretely outlines a standardized set of attributes that form a universal language shared between different classes of events. The base event serves as a reference point for creating new classes of events, providing a model on which specific classes can be built. One of the distinctive features of the base event lies in its agnostic nature to the predefined event categories. This inherent adaptability becomes particularly valuable when facing the challenge of classifying events that lack predefined categorizations or fall outside the boundaries of the established scheme. Let is now analyze the main attributes of the Base Event, according to the division in Group.

2.4.3.1 Classification Attributes of Base Event

The classification attributes within the framework's taxonomy play a pivotal role, contributing significantly to the overall structure and organization. These attributes serve as key elements that help categorize and define different aspects within the framework. In the Base Event, the classification attributes follow a pattern where each attribute with the suffix "_id" or "_uid" is accompanied by a corresponding attribute with the suffix "_name". An exception is made for the "severity_id" attribute, which is associated with the "severity" attribute instead of "severity_name". The attribute with the "_id" suffix is required, while its associated attribute with the "_name" suffix is optional. However, if the value of the attribute with the "_id" suffix is 99, denoting "Other", the associated attribute must be provided.

2.4.3.2 Occurrence Attributes of Base Event

In the OCSF, the occurrence attributes play a crucial role in capturing the temporal and frequency aspects of events. The framework places significant emphasis on representing time, utilizing attributes with the type "timestamp_t", which expresses time in milliseconds. An important addition to OCSF is the Date/Time profile, which introduces a new attribute type, "datetime_t", for each attribute previously defined with "timestamp_t". This allows for a more detailed description of time, adopting the RFC3339 format, such as "1999-09-07T23:20:50.52Z". In the Base Event, not all time attributes are represented as a single attribute, but some are within the "metadata" object attribute.

2.4.3.3 Primary attributes of Base Event

The primary attributes serve as clear indications of the event's meaning or context across all possible use cases. The primary attributes of a Base Event include:

- "message": Description of the event provided by the source.
- "observables": Collection of extracted information associated with the event, streamlining data display.
- "status": Represents the current state of the event, normalized to a corresponding label.
- "status_code": Reflects the current status of the event, directly sourced from the event provider.
- "status_detail": Supplementary information about the outcome of the event.
- "status_id": Standardized identifier categorizing the event into distinct states,

such as Unknown, Success, Failure, or Other.

2.4.3.4 Context Attributes of Base Event

Context attributes improve the significance and enrich the content of an event by introducing variations. Within the event base class, four specific context attributes are integrated to bring nuance and depth to the information conveyed. These context attributes serve to provide additional layers of meaning, ensuring that the event is not only comprehensible at a basic level but also capable of conveying more details and contextual nuances. They are:

- "enrichments": Enrichments attribute adds external data to events via Enrichment objects with required (data, name, value) and recommended (provider, type) attributes. Data attribute allows flexible JSON insertion. Name and value attributes are crucial for linking to specific attributes within the enriched event class. Incorporating "modified_time" into "metadata" during enrichment is recommended for accurate temporal context. For example add location information for the IP address in the DNS answers [2].
- "metadata": This object contains attributes that are filled with information external to the source event, essential attributes such as "product" and "version" are required. The "product" identifies the reporting software, while the "version" follows Semantic Versioning (SemVer) and indicates the version of the OCSF schema used. Other attributes help identify events, such as "correlation_uid" and "event_code". Attributes with the prefix "log" add details about the event registration system. Time attributes detail the timestamps of events, while "profiles" and "extensions" indicate which profiles and extensions are used.

Listing 2.1: Example of enrichments attribute.

```
[{
    name: answers.ip,
    value: 92.24.47.250,
    type: location,
    data: {city: Socotra, continent: Asia, coordinates: [-25.415,
    17.0743], country: YE, desc: Yemen}
}]
```

- "raw_data": The "raw_data" refers to the unprocessed and unstructured information obtained directly from the event source.

- “unmapped”: It refers to attributes that have not been associated with a field inside the OCSF event schema.

2.4.4 Constraints

A constraint is a documented validated rule that indicates how attributes should be used. One type of constraint is the “at_least_one” constraint. This rule acknowledges that, while not all attributes are mandatory in every use case, there should be a minimum requirement to ensure clarity and avoid ambiguity. In simpler terms, it suggests that at least one of the specified recommended attributes within the constraint must be filled. In the “actor” object, at least one attribute must be present between “process”, “user”, “invoked_by”, and “session”.

Listing 2.2: Example of “at_least_one” constraints.

```
constraints: {
    at_least_one: [
        process,
        user,
        invoked_by,
        session
    ]
}
```

The second constraint, “just_one”, takes a slightly different approach. It recognizes that, in certain scenarios, having more than one recommended attribute populated might create confusion or redundancy. Therefore, it mandates that only one of the specified attributes within the constraint should be filled.

Listing 2.3: Example of “just_one” constraints [3].

```
constraints: {
    just_one: [
        privileges,
        group
    ]
}
```

2.4.5 Associations

In the context of OCSF, the association construct is used within a class definition to denote relationships between attributes. These relationships can be either bi-directional or uni-directional. The purpose of these associations is to indicate that certain attributes within an event class may be associated with each other, and in some cases, only one of them is present in the event while the other can be added at processing or storage time. For example, let is consider an event in the "Scheduled Job Activity" Class with attributes: "actor" and "device". The association construct specifies that there is a relationship between the "actor.user" and "device". In this case, the relationship is bi-directional, meaning that if you have an attribute "actor.user", you can also access the attribute "user" and vice versa.

Listing 2.4: Example of associations into "Scheduled Job Activity" Class [4].

```
Attribute Associations
actor.user: device
device: actor.user
```

There could be situations where the association may be uni-directional, indicating that you can only access one attribute from the other.
This construct becomes particularly useful in automated processing systems where a lookup service is available for an attribute that might not be populated directly by the event source producer. In such cases, the association allows you to link attributes in a way that enables the retrieval of additional information during processing or storage. In these situations, it is important to fill the attribute "processor_time" of the "metadata" object, at the moment the association is established.

2.5 Categories

In the OCSF framework, the incorporation of event classes into distinct categories is a strategic practice that serves several purposes, contributing to the overall cohesion and efficiency of the system.

Firstly, these categories act as logical containers, grouping event classes that share common themes or belong to specific domains. This organization enhances documentation clarity and simplifies the search process. Users can easily navigate through related event classes, making it more convenient to understand and work with the system.

Beyond documentation, categorization plays a crucial role in reporting. By associating event classes with specific categories, the system can generate more targeted and meaningful reports. Storage partitioning is another key aspect influenced by categories. Grouping related event classes logically can optimize storage structures, promoting efficiency in data retrieval and management. This proves valuable for scalability and performance, ensuring that the system operates smoothly even with large volumes of data.

Additionally, categories contribute to access control strategies. By organizing event classes based on their nature or sensitivity, the system can implement security measures at the category level. This means that users or processes can be granted specific access rights, enhancing overall system security.

Each category is uniquely identified by a "category_uid" attribute, providing a distinct marker for reference. To enhance user-friendliness, these categories are associated with friendly name attributes: "category_name". These names serve as intuitive labels, facilitating effective communication and understanding among users.

2.5.1 From Categories to Profiles

Determining the optimal level of detail for categories is a crucial aspect of modeling. In handling events that might neatly fit into multiple distinct categories, the challenge is to avoid diluting the specificity and context inherent to each category. To address this, the OCSF introduces a practical solution: the concept of Profiles. Profiles serve as a tool to navigate the complexity of overlapping categorical scenarios without resorting to the creation of additional event classes that might only partially contribute new information. Instead of forming generalized categories that risk losing the nuanced context of specific events, OCSF employs Profiles to manage this overlap effectively.

2.6 Profiles

Profiles serve as dynamic extensions to event classes and objects, acting as versatile mix-ins of attributes along with their associated requirements and constraints. Unlike event classes, which specialize in category domains, profiles provide a means to enhance existing event classes by introducing a set of attributes that are independent of category considerations.

A profile improves the functionality of event classes, acting as a modular add-on, contributing to creating attributes that are tailored to specific contexts without the need to create entirely new classes for each variation. This is achieved through the optional "profiles" attribute within the Base Event class, where multiple profiles can be seamlessly added in the form of an array of profile values.

The mix-in methodology inherent in profiles promotes the reusability of event classes, eliminating the need to replicate classes that share the same attributes. By incorporating profiles, event classes, and instances can be selectively filtered using the profiles attribute, transcending the boundaries of categories and event classes. This introduces an additional layer of classification, allowing for a more nuanced and flexible approach to organizing and managing events.

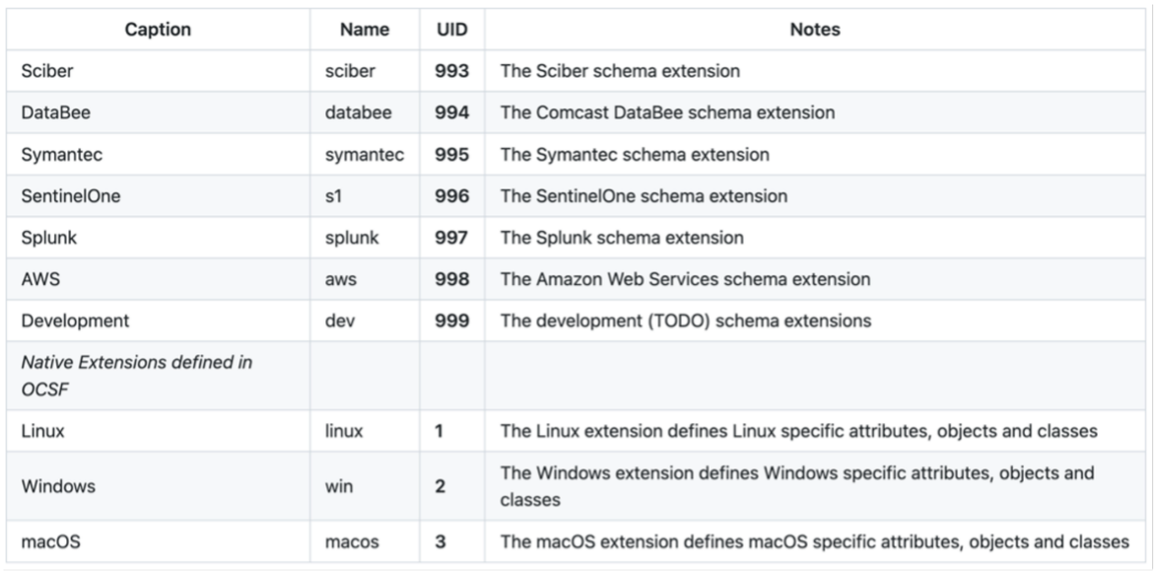
If the "profiles" attribute is absent, it signifies that no profile attributes are added, as expected. Attributes defined within a profile come with mandatory requirements that cannot be overridden, considering profiles are optional. The assumption is that the application of a profile implies a desire for those specific attributes, and they can be populated accordingly. However, certain classes, like System Activity classes, inherently incorporate the attributes of a profile. For instance, the "Host" profile attributes such as "device" and "actor" are explicitly defined within the class. Even when a class definition includes these profile attributes, the class still registers for that profile. This registration is done in the class definition to ensure compatibility with any searches across events associated with that profile. It is important to note that in such cases, the attribute requirements defined within the class take precedence over those specified by the profile.

2.7 Extensions

OCSF provides a flexible structure for defining new attributes, objects, categories, profiles, and event classes. The framework allows the extension of schemas, enabling vendors or customers to tailor the schema to their specific needs.

Extensions serve multiple purposes, such as accommodating vendor-specific requirements, enhancing an existing schema, factoring out non-essential schema domains keeping a schema small. When extending the core schema, the process involves creating categories, profiles, or event classes as needed, drawing from the schema dictionary. Additionally, extensions can introduce new attributes and objects to the dictionary.

Maintaining consistency and avoiding conflicts is crucial, and this is achieved through unique identifiers and versioning. Each extension, like categories, event classes, and profiles, is assigned a unique ID within the OCSF framework. Versioning is essential to keep track of schema changes over time, and extended events should update the "metadata.version" attribute to reflect the extended schema version. To ensure the uniqueness and visibility of extensions, developers must register their extensions in the OCSF Extensions Registry. This involves assigning a unique identifier and name to the extension. This registry plays a vital role in preventing collisions with the core schema or other extensions, making the extended schema widely accessible and identifiable. By adhering to these practices, developers can effectively manage and integrate customizations within the OCSF framework. For instance, the introduction of a new extension would be represented by a new entry in the table.



Caption	Name	UID	Notes
Sciber	sciber	**993**	The Sciber schema extension
DataBee	databee	**994**	The Comcast DataBee schema extension
Symantec	symantec	**995**	The Symantec schema extension
SentinelOne	s1	**996**	The SentinelOne schema extension
Splunk	splunk	**997**	The Splunk schema extension
AWS	aws	**998**	The Amazon Web Services schema extension
Development	dev	**999**	The development (TODO) schema extensions
Native Extensions defined in OCSF			
Linux	linux	**1**	The Linux extension defines Linux specific attributes, objects and classes
Windows	win	**2**	The Windows extension defines Windows specific attributes, objects and classes
macOS	macos	**3**	The macOS extension defines macOS specific attributes, objects and classes

Figure 2.1: OCSF Extensions Registry [5].

Expanding the schema involves the creation of a fresh subdirectory within the extensions directory. In this newly established subdirectory, introduce a new "extension.json" file that serves to articulate key details like the extension's name and unique identifier (UID). It's worth noting that the architecture of the extension's directory aligns with that of the primary schema directory. Within this framework, the extension directory can incorporate specific files such as "categories.json" and "dictionary.json". Additionally, the extension directory has the flexibility to include subdirectories for the desired type of extension. These subdirectories may be events, including, objects, and profiles, depending on the extension you want to make. This design allows for a customizable and adaptable organizational approach, ensuring that the extension directory aligns seamlessly with the intended functionality and features of the extension.

Extensions to the core schema serve another purpose by facilitating the creation of new schema artifacts. These newly developed elements may be evaluated and, if deemed appropriate, be integrated into the core schema or designated as part of a platform extension. In the OCSF schema, experimental categories and event classes featuring additional attributes and objects are denoted with a “dev” extension superscript, illustrating their status as developmental elements.

Chapter 3

Tecnologies

3.1 Terraform

Terraform is an open-source infrastructure as a code software tool created by HashiCorp [6]. It allows users to define and provision infrastructure resources such as virtual machines, networks, storage, and more, using a declarative configuration language. This means you define the desired state of your infrastructure in code, and Terraform handles the provisioning and management of those resources.

Terraform stands out as a crucial tool in modern infrastructure management for several compelling reasons. Firstly, its adoption of Infrastructure as Code (IaC) empowers teams to treat infrastructure configurations as software, enabling version control, collaboration, and automation. This shift not only enhances efficiency but also reduces errors and facilitates better team coordination.
Moreover, Terraform's declarative approach to configuration simplifies infrastructure management by allowing users to define the desired state of their systems without getting bogged down in procedural details. This abstraction enables Terraform to handle the intricacies of provisioning and managing resources, thereby improving clarity and maintainability.
Another key advantage of Terraform is its support for multiple cloud providers, ensuring compatibility across various platforms such as AWS, Azure, and Google Cloud. This capability enables organizations to avoid vendor lock-in and leverage the best features of each cloud provider while managing their infrastructure through a unified interface.
Additionally, Terraform promotes scalability and consistency by enabling easy adjustments to configuration files to accommodate changing needs. This flexibility facilitates the smooth scaling of infrastructure while also mitigating the risk of configuration discrepancies across different environments.
Terraform streamlines the complexities of infrastructure management in cloud-based environments, offering a potent solution for modern development and operational workflows.

Using Terraform with AWS provides a streamlined approach to managing your cloud infrastructure as code. Now, it describes the process step by step.
Firstly, you will need to install Terraform on your local machine or CI/CD server.

With Terraform installed, the user will write your infrastructure code using the HCL within .tf files. Here, the user will define the AWS resources you want to provision, along with their configurations. For instance, he can specify EC2 instances, VPCs, subnets, security groups, IAM roles, S3 buckets, and more. The user has to navigate to Terraform project directory and execute "terraform init". This command initializes your working directory and automatically downloads any required provider plugins. For AWS, Terraform fetches the AWS provider plugin, which enables interaction with the AWS API. Then, it can generate an execution plan by running the "terraform plan". Terraform analyzes your configuration files and compares the current state of your infrastructure with the desired state defined therein. It then outlines the actions needed to achieve the desired state. The user can review the plan output carefully to ensure it aligns with his expectations. When the user is satisfied with the plan, he can execute "terraform apply". Terraform handles the creation, updating, or deletion of resources as necessary to bring the user's infrastructure in line with the desired state. Before any changes are made, Terraform prompts you to confirm the execution plan. Should the need arise to decommission your infrastructure or clean up resources, utilize "terraform destroy". This command permanently deletes all resources managed by Terraform in your AWS account, based on your configuration files.

3.2 Kubernetes

Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers [7]. Originally developed by Google, it is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes helps manage containerized applications across a cluster of machines, providing mechanisms for deployment, maintenance, and scaling, all while ensuring high availability and resource efficiency. It abstracts away the underlying infrastructure, allowing developers to focus on application development rather than worrying about the specifics of the infrastructure.

3.2.1 Kubernetes architecture

A Kubernetes cluster is made up of a group of worker machines, also known as nodes, that execute containerized applications. Each cluster must always have at least one working node. The worker nodes are responsible for hosting the Pods, which are the building blocks of the application workload. The control plane manages both the worker nodes and the Pods within the cluster. In production environments, the control plane is typically distributed across multiple computers, and a cluster generally runs on several nodes to ensure fault tolerance and high availability [8].

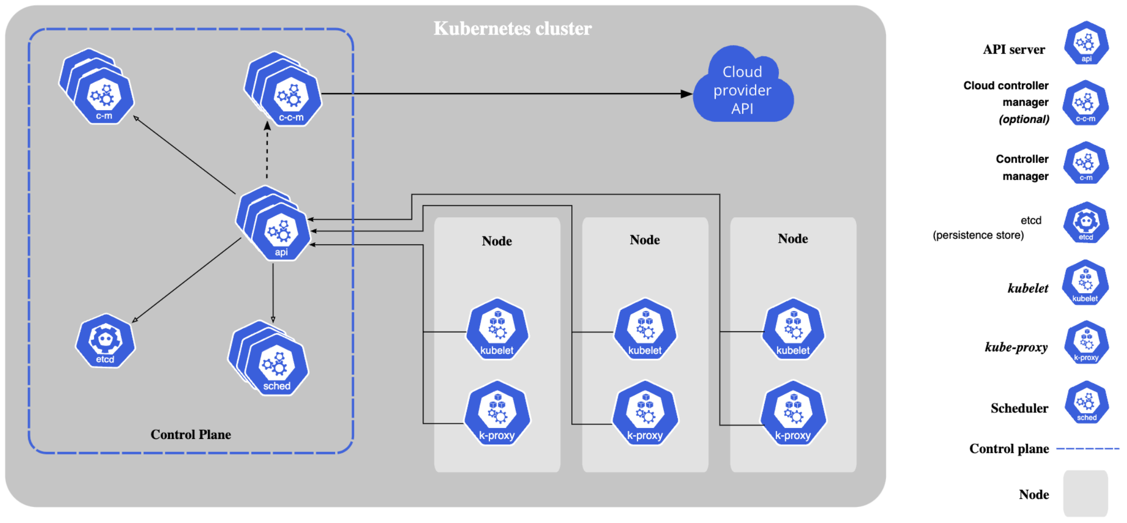


Figure 3.1: Kubernetes cluster's architecture [8].

3.2.1.1 Kubernetes Control Plane

The Kubernetes control plane, often referred to as the master in Kubernetes architecture, is responsible for managing the Kubernetes cluster. It consists of several components that work together to maintain the desired state of the cluster and to manage the deployment, scaling, and operation of applications. Here is a detailed breakdown of the components that make up the Kubernetes control plane:

- API Server: The API server is the central management entity of the Kubernetes control plane. It provides access to the Kubernetes API, enabling users, administrators, and other system components to communicate with the cluster. All operations and communication within the cluster are mediated by the API server. It serves RESTful API endpoints that clients use to interact with the cluster.
- Scheduler: The scheduler is responsible for assigning newly created pods to nodes within the cluster. It takes into account various considerations such as resource needs, quality expectations, limitations posed by hardware, software, or policies, preferences for proximity or separation of workloads, data placement, potential impact of workload interactions, and time constraints.
- Controller Manager: The controller manager is a collection of controllers that regulate the state of the cluster. Each controller works to ensure that the cluster reaches and maintains the desired state. Examples of controllers include the

Replication Controller, ReplicaSet Controller, Endpoint Controller, Namespace Controller, and Service Account Controller.

- etcd: etcd is a distributed key-value store used as Kubernetes' backing store for all cluster data [8]. It stores configuration data that represents the state of the cluster at any given time, including cluster configuration, node and pod metadata, service information, and more. Consistency and high availability are ensured through distributed consensus algorithms.
- Cloud Controller Manager: The Cloud Controller Manager is responsible for integrating the cluster with the underlying cloud provider's API. It manages the interaction between the cluster and the cloud provider's services, such as load balancers, storage, and networking. This component is only present in Kubernetes clusters deployed on cloud infrastructure and is optional.

3.2.1.2 Kubernetes Node

The Kubernetes worker node is responsible for running the workload, which typically includes containers managed by Kubernetes. Here is a detailed breakdown of the components that make up the Kubernetes worker node:

- Kubelet: Kubelet is the primary node agent that runs on each worker node. It is responsible for managing the containers running on the node and ensuring that they maintain the desired state specified in the Kubernetes manifests. Kubelet communicates with the Kubernetes API server to receive instructions about pod deployment, updates, and deletions. It manages the pod lifecycle, executing actions such as pulling container images, starting, stopping, and restarting containers as necessary.
- Kube Proxy: The Kube Proxy is a network proxy that operates on every node within the cluster. It maintains network rules on the node, allowing network communication to and from pods running on that node. Kube Proxy implements Kubernetes services abstraction by managing network traffic routing to the appropriate pods based on service selectors.
- Container Runtime: The container runtime is responsible for running containers on the node. Kubernetes supports multiple container runtimes, including Docker, containerd, and CRI-O. The container runtime is responsible for pulling container images from a container registry, creating containers based on those images, managing the container lifecycle, and providing isolation between containers.
- Addons: Worker nodes may also run optional add-ons to provide additional

functionality, such as DNS, Dashboard, Container Resource Monitoring, cluster-level logging, and Network plugins.

3.2.2 Kubernetes Objects

In Kubernetes, various objects are used to define the desired state of the system, allowing Kubernetes to manage and maintain the application environment accordingly. Here are some of the key objects in Kubernetes:

- Namespace: namespace is a way to logically divide cluster resources into separate virtual clusters. It is like creating multiple virtual clusters within a single physical cluster. Namespaces provide a scope for names so that resources of the same type can have the same name within different namespaces. This helps in organizing and managing Kubernetes objects and resources more effectively, especially in multi-tenant environments or when dealing with multiple projects or teams. Here are some common namespaces in Kubernetes:
  - default: This is the default namespace for objects which do not have a namespace explicitly specified. It is recommended not to deploy your applications into this namespace for better isolation.
  - kube-system: This namespace is reserved for Kubernetes system objects and resources created by Kubernetes itself. Components like kube-dns, kube-proxy, and others reside in this namespace.
  - kube-public: Resources in this namespace are visible and accessible to all users (including those not authenticated) and can be useful for sharing resources publicly.
  - kube-node-lease: This namespace holds node lease objects that are used for node heartbeats, determining node availability, and maintaining the high availability of the cluster.
- Pod: Pods are the smallest deployable units in Kubernetes and can consist of one or more containers. They share a common network namespace, allowing containers within the same Pod to communicate with each other via localhost. Pods can be created directly, but it is more common to define them using higher-level controllers like Deployments or StatefulSets. They are ephemeral by default, meaning they can be created, destroyed, and replaced dynamically based on the cluster's needs.
- Volumes: Volumes are directories that hold data accessible to containers within pods. They serve to persist data beyond pod lifecycles, enable data sharing

between containers, and facilitate communication with the underlying infrastructure.

- StatefulSet: StatefulSets are used to manage stateful applications with unique identities. They provide guarantees about the ordering and uniqueness of Pods, persistent storage, and stable network identities. StatefulSets are suitable for applications like databases, queues, and key-value stores that require stable, persistent storage and ordered deployment.
- ReplicaSet: ReplicaSets ensure that a specified number of identical Pods are running at all times. They are used by Deployments to maintain the desired number of Pods and handle scaling operations. If a Pod fails or is deleted, the ReplicaSet creates a new Pod to maintain the desired number of replicas.
- Deployment: Deployments manage the lifecycle of Pods by controlling ReplicaSets. They enable declarative updates to Pods and ReplicaSets, facilitating rolling updates and rollbacks. Deployments ensure that a specified number of Pods are running and handle scaling, self-healing, and updates to the desired state. The following example creates a ReplicaSet with three nginx Pods.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 3.1: Example of a Deployment [9]

- Service: There are several ways to expose services to the outside world or to other services within the cluster. The three common ways are:
  - ClusterIP: This is the default service type in Kubernetes. It essentially makes the service accessible only within the cluster via a cluster-internal IP address. This type of service is commonly utilized for facilitating communication between various components of an application that are running within the same Kubernetes cluster.
  - NodePort: This service type exposes the service on each node's IP at a static port. It allows for accessing the service from outside of the cluster. When you create a NodePort service, Kubernetes allocates a port from a range on each node and forwards traffic from that port to the service.
  - LoadBalancer: This service type exposes the service externally using a cloud provider's load balancer. It is typically used when you want to expose your service to the internet or to an external network. The cloud provider provisions a load balancer that forwards traffic to the service. This type is only available when running Kubernetes in certain environments that support external load balancers.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 3.2: Example of a Service [10]

- Ingress: Ingress serves as a configuration API within Kubernetes, handling external access to services hosted within a cluster. It acts as a traffic controller, directing incoming HTTP and HTTPS traffic to the appropriate services based on defined rules. Ingress works with an Ingress controller, which configures load balancing or proxying to enable external access to services. It simplifies routing by allowing users to define rules for routing traffic based on hostnames, paths, and services. Additionally, Ingress controllers can handle TLS termination for secure communication.

- HPA: The Horizontal Pod Autoscaler is a feature that automatically adjusts the number of replica pods in a deployment, replica set, or stateful set based on observed CPU utilization. It continuously monitors the CPU usage of pods, compares it to a target value, and scales the number of replicas up or down accordingly to ensure optimal resource utilization and application performance. The HPA provides a dynamic feedback loop that adapts to changing workloads, helping Kubernetes clusters efficiently handle varying levels of demand without manual intervention.

3.3 Docker

Docker is a platform for developing, shipping, and running applications using containerization technology. Containers are lightweight, portable, and self-sufficient environments that package an application and all its dependencies. Docker provides tools and a platform to manage these containers efficiently. It is widely used in software development and deployment pipelines because it allows developers to package their applications with everything they need to run consistently across different environments, from development to production. With Docker, you can create, deploy, and manage containers easily using Docker Engine, which is a runtime and set of tools for managing containers. Docker Compose is another tool that simplifies the process of defining and running multi-container Docker applications.

3.3.1 DockerFile

A Dockerfile is a text file that contains a set of instructions used by the Docker engine to automatically create a Docker image. These instructions define the steps needed to assemble an image that includes everything needed to run a particular application or service inside a Docker container. Here is a breakdown of what you might typically find in a Dockerfile:

- Base Image: The Dockerfile usually starts with a “FROM” instruction that specifies the base image to use. This base image provides the starting point for your image and contains the operating system and basic runtime environment for your application.
- Working Directory: The “WORKDIR” instruction sets the working directory inside the container where subsequent commands will be executed.
- Copy Files: The “COPY” or “ADD” instructions copy files and directories from your local machine into the Docker image. This is how you add your application

code, configuration files, and other necessary assets into the container.

- Install Dependencies: Using the "RUN" instruction, you can execute commands inside the container to install any necessary dependencies or perform setup tasks. This might include installing packages using package managers, downloading files, or running build scripts.
- Expose Ports: The "EXPOSE" instruction documents which port the container listens on during runtime. It does not actually publish the ports; it is just metadata to indicate which ports are intended to be exposed.
- Define Environment Variables: The "ENV" instruction sets environment variables inside the container. This can be useful for configuring the behavior of your application or specifying runtime settings.
- Define Runtime Commands: The "CMD" or "ENTRYPOINT" instruction specifies the command that should be executed when the container starts. This is typically the main command to run your application or service.
- Additional Instructions: Depending on your specific requirements, you might include additional instructions in your Dockerfile for tasks like setting up user permissions, cleaning up unnecessary files, or configuring startup behavior.

Here is a simple example of a generic Dockerfile for a Node.js application.

```dockerfile
# Use a Node.js runtime as the base image
FROM node:14
# Set the working directory in the container
WORKDIR /usr/src/app
# Copy init.json to the working directory
COPY init.json ./
# Install dependencies
RUN npm install
# Copy the rest of the application code to the working directory
COPY . .
# Expose the port the app runs on
EXPOSE 1999
# Define the command to run the application
CMD [node, app.js]
```

Listing 3.3: Example of Dockerfile.

By writing a Dockerfile, you can define your application's environment and dependencies in a portable and reproducible way, allowing you to easily package and distribute your application as a Docker image that can run consistently across different environments.

3.4 Fluentd

Fluentd is an open-source data collection software that is designed to unify data collection and consumption. It acts as a robust and flexible data collector, allowing you to efficiently collect, process, and distribute logs, events, and other data across various sources and destinations in real time. Fluentd follows a plugin-based architecture, which means it can be extended with a wide range of plugins to support different input sources, output destinations, and data processing functions. The "fluent.conf" file is the configuration file used by Fluentd to define its behavior and settings. It is typically written in a simple declarative language that specifies input sources, output destinations, and any data processing steps that need to be applied. In the "fluent.conf" file, you can configure various aspects of Fluentd, such as input plugins (such as tailing log files, listening to TCP/UDP streams), output plugins (such as writing to files, sending to databases), and filters (such as manipulating data, filtering events). This file essentially serves as the roadmap for Fluentd, guiding how it collects, processes, and forwards data within your system. Here is an example of the "fluent.conf" file.

```
<source>
    @type forward
    port 24224
</source>

<match **>
    @type stdout
</match>
```

Listing 3.4: Example of "fluent.conf"

Chapter 4

Amazon Web Services (AWS)

Amazon Web Services, a subsidiary of Amazon, provides flexible cloud computing solutions and APIs to individuals, businesses, and government organizations. Users only pay for what they use based on their usage. AWS offers a wide range of services including computing power, storage, databases, machine learning, analytics, networking, mobile development, developer tools, IoT, security, and enterprise applications. It is one of the leading cloud service providers globally, known for its reliability, scalability, and extensive feature set. With its cloud infrastructure, you have the freedom to deploy your applications across various locations using Regions and Availability Zones ensuring optimal accessibility and performance.

4.1 Region

AWS divides its global infrastructure into regions, which are geographic areas with multiple availability zones. Each region is designed to be isolated from other regions to provide fault tolerance and minimize the impact of regional disasters.

Scalability is one of the key benefits of AWS regions. Customers can deploy their applications and resources in multiple regions to achieve geographic redundancy and scale their infrastructure globally. This allows businesses to expand their operations into new markets and serve customers in different parts of the world with low latency and high performance.

It is important to note that AWS services are not uniformly available in all regions. Some services may be launched initially in specific regions before being rolled out globally. Therefore, the choice of region can have implications for service availability, data residency, and compliance requirements.

4.2 Availability Zone

Within each AWS region, there are multiple availability zones (AZs), distinct locations with their own infrastructure and facilities. These AZs are interconnected with high-speed, low-latency links, enabling synchronous replication and real-time data processing between zones. The primary purpose of availability zones is to provide high availability and fault tolerance for applications and services deployed in the cloud. By distributing resources across multiple AZs within the same region, customers can ensure that their applications remain operational even in the event of failures or disruptions in one zone. Each AZ is designed to be isolated from failures in other zones, with redundant power, cooling, and networking infrastructure to minimize the risk of correlated failures. This fault isolation ensures that a problem in one AZ does not cascade to affect the entire region, enhancing the overall resilience of the AWS cloud.

4.3 Amazon Virtual Private Cloud (VPC)

Amazon Virtual Private Cloud (VPC) provides you with the capability to establish a private segment within the AWS Cloud. Within this segment, you can deploy AWS resources in a customized virtual network environment tailored to your specifications. This provides you with control over your virtual networking environment, including selection of your IP address range, creation of subnets, and configuration of route tables and network gateways [11].

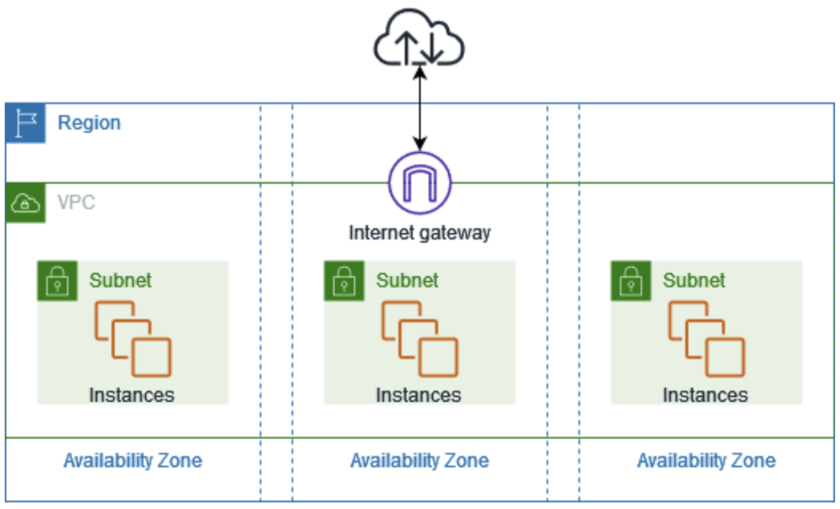


Figure 4.1: Example of VPC [12]

Below are some key components and features of AWS VPC:

- Subnets are the foundational building blocks of a VPC. They represent a segmented portion of the IP address range that you define for your VPC. Each subnet is associated with an Availability Zone within a specific region. Subnets

can be categorized into public and private.
Public subnets have a route to the internet gateway and typically host resources that need direct access to the internet, such as web servers or load balancers.
Private subnets do not have a route to the IGW and are isolated from the internet. They are commonly used to deploy backend databases, application servers, or other resources that should not be directly accessible from the internet.

- An Internet Gateway (IGW) enables the communication between instances in your VPC and the Internet. It serves as a gateway for internet-bound traffic, allowing resources in public subnets to send outbound traffic to the internet and receive inbound traffic from the internet when necessary.
- Route tables are used to determine the path that network traffic takes within your VPC. Each subnet in your VPC must be associated with a route table, which contains rules, that specify where traffic should be directed. Key routes include:
  - Local Route: Automatically added by default, allows communication within the same VPC.
  - Internet Route: Points to the IGW for public subnets to access the internet.
  - NAT Gateway Route: For private subnets to access the internet via a Network Address Translation (NAT) gateway.
- Network Access Control Lists (NACLs) act as stateless firewalls at the subnet level, controlling inbound and outbound traffic. They allow you to define rules to permit or deny traffic based on IP addresses, protocols, and ports.
- Security groups act as virtual firewalls that manage inbound and outbound traffic on an instance level. They are stateful and allow you to create rules based on security groups, IP addresses, protocols, and ports. Security groups are more granular than NACLs and provide additional security by restricting access to resources based on their roles and requirements.
- An interface endpoint in AWS is a construct that allows resources within a VPC to privately access AWS services without requiring an internet gateway or NAT instances. It essentially creates a direct network connection between resources within your VPC and the AWS service, bypassing the need to traverse the public Internet.
  When you create an interface endpoint, AWS provisions a network interface in your VPC. This network interface resides in one or more of your VPC's subnets. The interface endpoint is assigned one or more private IP addresses from the IP address range of the subnet(s) where it is deployed. Each interface endpoint

is specific to a particular AWS service (such as S3, DynamoDB, etc.). AWS provides different endpoint services for different AWS services. Any traffic destined for the AWS service associated with the endpoint is routed to the interface endpoint's private IP address within the VPC. This traffic remains within the AWS network and does not traverse the public internet. You can attach security groups to the interface endpoint, allowing you to control the inbound and outbound traffic to and from the endpoint. AWS automatically provides DNS resolution for the endpoint's private IP address. When your VPC resources make requests to the AWS service, AWS's DNS resolves the endpoint's DNS hostname to the private IP address of the endpoint.

In the first scenario, there is a VPC endpoint for Amazon CloudWatch with a single network interface located in one Availability Zone. When any resource within the VPC accesses CloudWatch through its public endpoint, the traffic is directed to the IP address of this single network interface. However, if the Availability Zone where this interface resides experiences issues, resources in other Availability Zones lose access to CloudWatch.

In the second scenario, the VPC endpoint for CloudWatch has network interfaces in two Availability Zones. When a resource in any subnet accesses CloudWatch via its public endpoint, a healthy interface is selected using round-robin selection. This ensures that traffic is evenly distributed between the two interfaces, providing redundancy and maintaining access to CloudWatch even if one Availability Zone encounters problems.

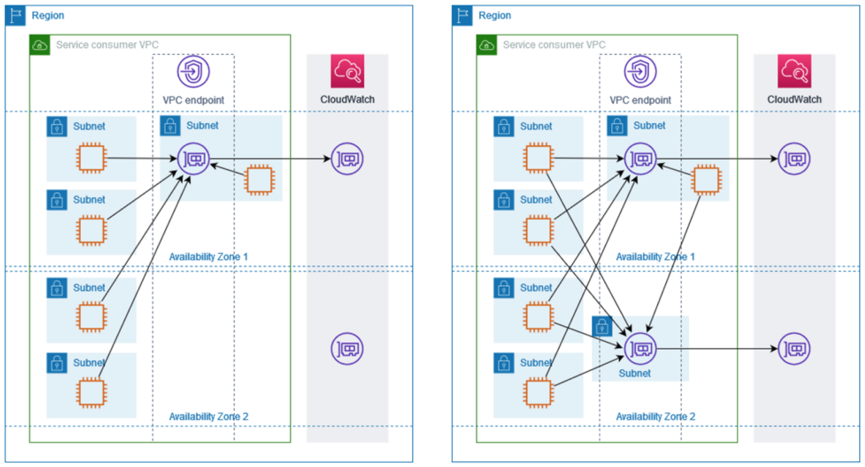


Figure 4.2: Examples of VPC Endpoint [13]

4.4 AWS CodeCommit

CodeCommit is a fully managed Git repository hosting service by AWS that offers secure and scalable storage for source code and development assets. It provides encryption for data both at rest and in transit, ensuring the confidentiality and integrity of your code. With fine-grained access control policies managed through IAM, you can regulate user access and permissions. CodeCommit supports branching and merging, allowing teams to work concurrently on different features. Collaboration features like pull requests facilitate code review and discussion. Integration with AWS services like CodeBuild and CodePipeline enables automated development workflows, including CI/CD pipelines. Audit capabilities and compliance features ensure tracking, monitoring, and adherence to security requirements. Cross-region replication ensures high availability and durability, making CodeCommit suitable for enterprise-scale applications. The figure below illustrates how to create and manage repositories using your development machine, the AWS CLI or CodeCommit console, and the CodeCommit service [13].

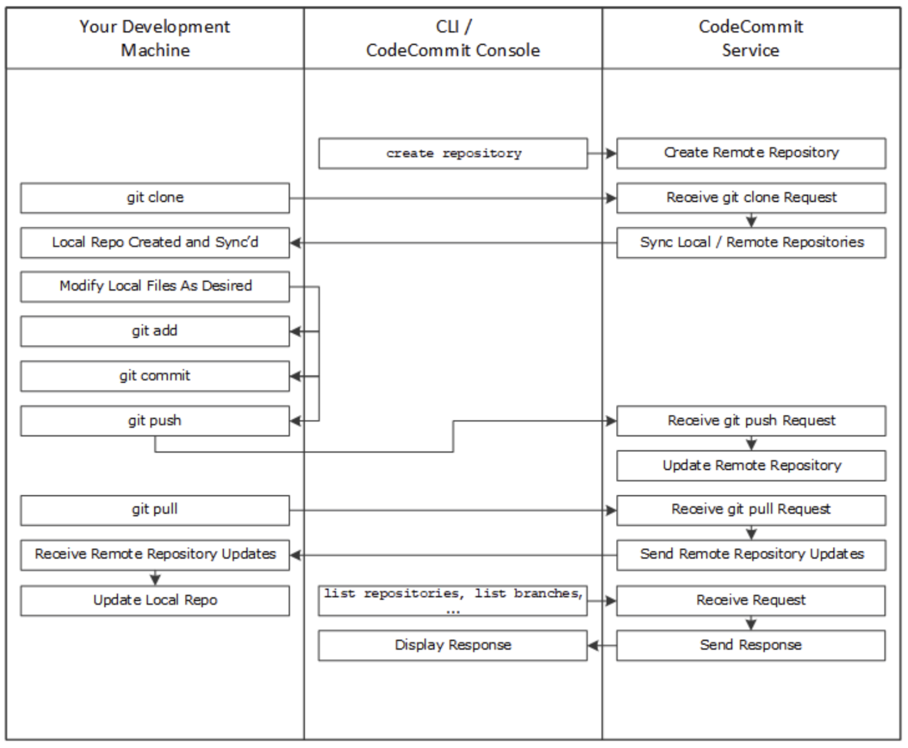


Figure 4.3: Illustration of how to create and manage repositories [14]

First, create a CodeCommit repository using either the AWS CLI or the CodeCom-

mit console. Then, on your local development machine, clone the repository using Git. This sets up a connection between your local environment and the CodeCommit repository.
Now, you can make changes to the files in your local repository: adding, editing, or deleting them. After making changes, stage them using "git add", commit them locally with "git commit", and push them to the CodeCommit repository using "git push".
To ensure you are working with the most up-to-date files, you can download changes made by other users. Simply use "git pull" to synchronize your local repository with the CodeCommit repository.

4.5 AWS CodePipeline

AWS CodePipeline is a service that handles the entire process of continuous integration and continuous delivery (CI/CD). It automates the build, test, and deployment phases of your release process for software applications, enabling you to deliver updates more reliably and rapidly.

The pipeline can be created through the AWS Management Console or the AWS CLI. A pipeline consists of a series of stages, each representing a phase of your release process, such as source, build, test, and deploy. Here are the main stages of a pipeline:

1. Source Stage: The first stage of the pipeline is typically the source stage, where CodePipeline retrieves the source code from a version control system like AWS CodeCommit, GitHub, or Amazon S3.
2. Build Stage: After the source code is retrieved, the next stage is the build stage. Here, AWS CodePipeline invokes AWS CodeBuild or other build tools to compile the source code, run tests, and produce deployable artifacts such as executable binaries, Docker images, or static website files.
3. Test Stage: Once the build stage is complete, the pipeline can include a test stage where automated tests are executed against the artifacts generated in the build stage. These tests can include unit tests, integration tests, regression tests, and other forms of automated validation to ensure the quality of the software.
4. Deploy Stage: After the code has been built and tested successfully, the pipeline proceeds to the deploy stage. In this stage, AWS CodePipeline deploys the artifacts to the target environment, which could be AWS Elastic Beanstalk, Amazon ECS, AWS Lambda, EC2, or any other computing service supported by AWS.

Throughout the entire process, AWS CodePipeline monitors the execution of each stage and provides real-time visibility into the progress of the pipeline through the AWS Management Console or CloudWatch logs. If any stage fails, the pipeline stops execution, and you receive notifications to investigate and resolve the issue. This diagram illustrates an example release process using CodePipeline [15].

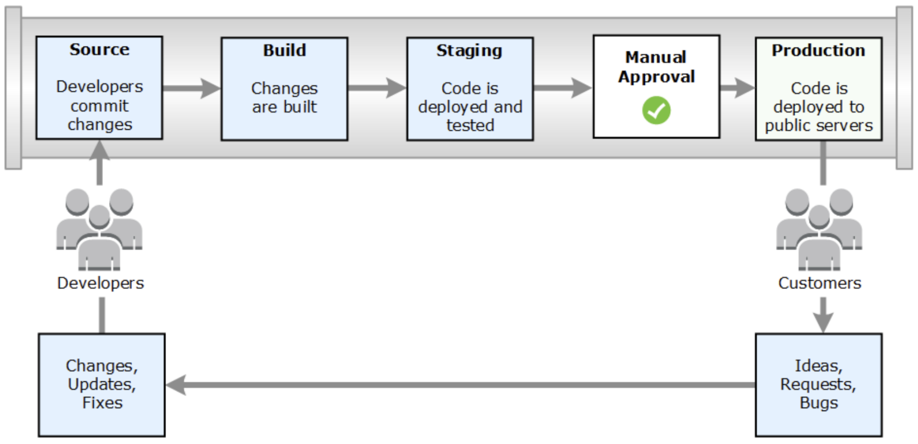


Figure 4.4: Illustration of how to create and manage repositories [15]

Developers make changes to the code and save them in a source repository. CodePipeline detects the changes and proceeds to build the code, runs tests (if they were set up), and deploys it to staging servers. Additional tests like integration or load tests are then performed on the staging servers. Once the code passes these tests, it requires manual approval before it can be deployed to production servers.

4.6 AWS CodeBuild

AWS CodeBuild is a fully managed continuous integration and continuous deployment (CI/CD) service provided by AWS. It is designed to help you automate your software development processes, from building and testing code to deploying it to various environments. CodeBuild can be run either from the AWS CodeBuild console or through AWS CodePipeline. Visualized in the diagram below, you can integrate CodeBuild into your pipeline within AWS CodePipeline, seamlessly incorporating it into either the build or test stage.

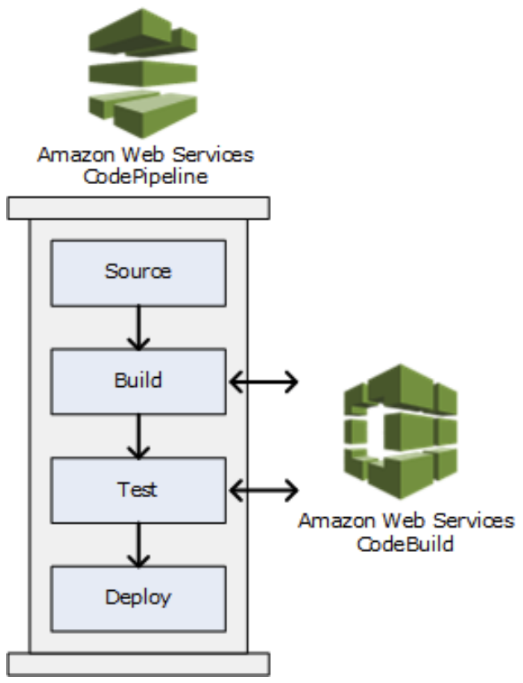


Figure 4.5: CodeBuild integrates with CodePipeline [16].

The key codebuild components are:

- Source: This is where your application's source code resides. CodeBuild supports various source code management systems such as AWS CodeCommit, GitHub, Bitbucket, and Amazon S3. When you set up a build project in CodeBuild, you specify the location of your source code repository. The source code is a buildspec.yml file which is a configuration file used to define the build process for your application. Key Components:

    1. Version: Specifies the version of the build specification syntax being used.
    2. Phases: Defines the different build phases and the commands to be executed within each phase. Phases typically include install, pre_build, build, post_build, and post_deploy, but you can customize them according to your project's needs.
    3. Artifacts: Specifies the files and directories to be packaged as build artifacts and the location where they should be stored. You can specify individual files or directories to include, set a base directory, and choose

whether to discard paths. Artifacts are typically uploaded to an Amazon S3 bucket or another destination for further deployment.

- Environment: CodeBuild provides customizable build environments where your code is built, tested, and packaged. You can choose from a variety of pre-configured build environments that come with different combinations of operating systems, programming language runtimes, and build tools. You can also create custom Docker container environments to meet your specific requirements. Each build environment consists of:
  - Compute resources: These are the virtual machines or containers where your build runs. You can choose the size and type of compute resources based on the complexity and resource requirements of your build.
  - Build tools and runtime: CodeBuild environments come with pre-installed build tools and runtime environments for popular programming languages such as Java, Python, Node.js, etc. You can also install additional dependencies and tools as needed.
  - Environment variables: You can define environment variables that are available to your build process. These variables can be used to configure your build, pass secrets securely, or provide other runtime information to your build scripts.
- Artifacts: After the build process is complete, CodeBuild produces artifacts, which are the output files generated by the build. Artifacts can include compiled code, executable files, documentation, test results, or any other files generated during the build process. CodeBuild can store artifacts in various locations such as Amazon S3, AWS CodeArtifact, or any other compatible storage service. You can specify the location and configuration of artifact storage when you define your build project.

CodeBuild is an AWS service for automating builds. You start by setting up a build project, and defining your source code repository and build environment. When triggered, CodeBuild retrieves the source code, runs build commands specified in your configuration, and monitors the process. After completion, it produces artifacts and uploads them to a specified location. Notifications or downstream actions can be set based on the build outcome.

4.7 AWS Elastic Container Registry (ECR)

AWS ECR is a managed Docker container registry service. It provides secure, scalable storage for Docker container images, allowing developers to push, pull, and manage their images in a central repository hosted on AWS infrastructure. ECR integrates with AWS IAM for access control, supports lifecycle policies for automated image cleanup, and encrypts images at rest and in transit for enhanced security. It seamlessly integrates with other AWS services like Amazon ECS and Amazon EKS, facilitating the deployment of containerized applications on AWS.

4.8 Amazon Elastic Kubernetes Service (EKS)

Amazon Elastic Kubernetes Service is a managed Kubernetes service provided by AWS [17]. Amazon EKS abstracts away the complexities of managing Kubernetes clusters, allowing developers to focus on building and deploying their applications without worrying about the underlying infrastructure.

4.8.1 Amazon EKS Architecture

Amazon EKS conforms to the typical cluster structure found in Kubernetes.

4.8.1.1 Control Plane

In Amazon EKS, the control plane refers to the layer of management and orchestration that handles the deployment, scaling, and management of Kubernetes clusters. This layer includes components such as the API server, scheduler, controller manager, etcd, and other supporting services required for the proper functioning of a Kubernetes cluster. Amazon EKS takes care of managing the control plane for you, so you do not need to install, operate, or maintain it yourself. You can interact with the control plane through the Kubernetes API server, which exposes the functionality of the cluster to users and applications. When you create an EKS cluster, AWS automatically provisions and manages the control plane on your behalf, ensuring that it is highly available and securely configured. This allows you to focus on deploying and managing your containerized applications without worrying about the underlying infrastructure.

4.8.1.2 Node

Aside from the control plane, an Amazon EKS cluster consists of worker machines known as nodes. Choosing the right type of node for your Amazon EKS cluster is vital to tailor to your particular needs and make the most of your resource allocation. Amazon EKS provides various options for managing the nodes within your Kubernetes cluster:

- AWS Fargate: This serverless compute engine for containers simplifies infrastructure management by automatically handling provisioning, scaling, and maintenance based on your application's resource needs. It is perfect for those who prioritize simplicity and want to focus on application development rather than infrastructure management.
- Karpenter: As a flexible Kubernetes cluster autoscaler, Karpenter dynamically adjusts compute resources to match application demands, optimizing both availability and efficiency. It is capable of provisioning resources precisely when needed, ensuring your workload requirements are met efficiently.
- Managed Node Groups: This option combines automation and customization for managing Amazon EC2 instances within your EKS cluster. AWS takes care of routine tasks like patching, updates, and scaling, while also supporting custom kubelet arguments for advanced resource management. Additionally, it enhances security through IAM roles for service accounts, streamlining permissions management across clusters.
- Self-managed Nodes: For users seeking complete control over their EC2 instances within the EKS cluster, self-managed nodes provide the ultimate level of customization. You are responsible for managing, scaling, and maintaining these nodes, offering granular control over the underlying infrastructure. This option suits those willing to invest time in infrastructure management to achieve specific customization and control requirements.

4.9 AWS CloudFormation

AWS CloudFormation is a powerful service provided by AWS that enables users to define and manage their infrastructure as code. This means that instead of manually creating and configuring individual AWS resources, users can use CloudFormation templates to describe the desired state of their infrastructure in a declarative manner. These templates are written in either JSON or YAML format and can include a wide range of AWS resources, including compute instances, storage buckets, databases, networking components, and more. Key Features and Concepts:

- Templates: CloudFormation templates serve as blueprints for provisioning and managing AWS resources. They provide a structured way to define the desired configuration of infrastructure components, including their properties, dependencies, and relationships. Example of YAML template.

```yaml
AWSTemplateFormatVersion: 2010-09-09
Description: Example
Resource:
     EC2InstanceExample:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: ami-0ff8a91507f77f867
    InstanceType: t3.micro

```

Listing 4.1: Example of CloudFormation YAML template

- Resources: Resources are the fundamental building blocks of CloudFormation templates. They represent the various AWS services and components that make up the infrastructure, such as EC2 instances, S3 buckets, RDS databases, IAM roles, and more.
- Stacks: Stacks are logical groups of resources that are managed together as a single unit by CloudFormation. When a template is deployed, it creates a stack containing all the resources defined within it. Stacks can be created, updated, and deleted as needed, providing a way to manage the lifecycle of infrastructure components.
- Parameters: Parameters are customizable values that can be passed to CloudFormation templates at runtime. They allow users to specify input values such as instance types, storage sizes, and network configurations when creating or updating stacks, making templates more flexible and reusable across different environments.
- Mappings: Mappings provide a way to define conditional values within CloudFormation templates based on input parameters or other criteria. They are useful for specifying different configurations for different environments or regions, allowing templates to adapt dynamically to changing requirements.
- Outputs: Outputs are values that are returned by CloudFormation after a stack has been created or updated. They provide a way to retrieve important information such as resource identifiers, endpoint URLs, or configuration settings for use in other parts of the infrastructure or applications.
- Rollback: CloudFormation includes automatic rollback functionality to ensure the integrity and consistency of stacks during deployment. If any resources fail

to be provisioned or updated during a stack operation, CloudFormation will automatically roll back the changes to their previous state, helping to prevent downtime or data loss.

Here is an illustration of the CloudFormation workflow for creating stacks [18].

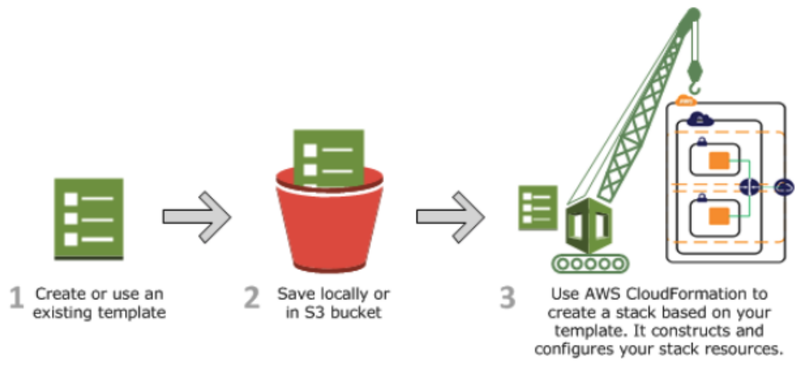

1 Create or use an existing template
2 Save locally or in S3 bucket
3 Use AWS CloudFormation to create a stack based on your template. It constructs and configures your stack resources.

Figure 4.6: CloudFormation workflow for creating stacks [18].

When you want to make changes to your stack's resources, you can simply update the stack's template instead of creating a new stack and deleting the old one. To do this, you submit a modified version of the original stack template, adjust input parameter values, or both, to create a change set. This change set reflects the differences between the modified template and the original one, listing the proposed alterations. Once you review these changes, you have the option to either proceed with updating the stack using the change set or create a new change set. The following figure illustrates the various steps to update a stack.

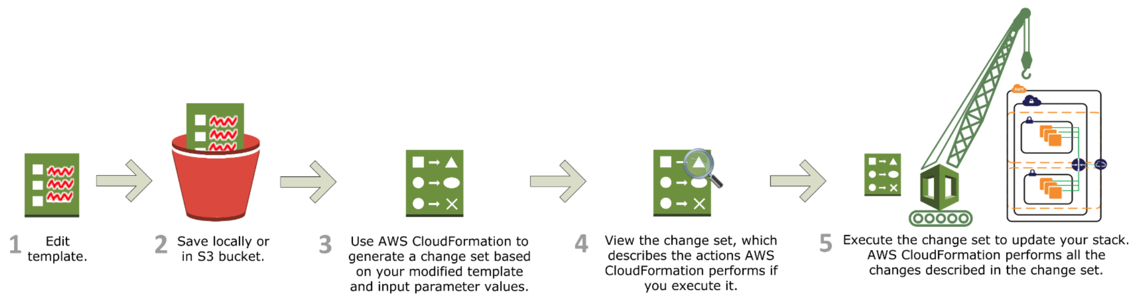

1 Edit template.
2 Save locally or in S3 bucket.
3 Use AWS CloudFormation to generate a change set based on your modified template and input parameter values.
4 View the change set, which describes the actions AWS CloudFormation performs if you execute it.
5 Execute the change set to update your stack. AWS CloudFormation performs all the changes described in the change set.

Figure 4.7: CloudFormation workflow for updating stacks [18].

4.10 AWS Simple Storage Service (S3)

Amazon Simple Storage Service is a cornerstone of AWS, providing businesses and developers with highly scalable and durable object storage in the cloud. At its core, S3 allows users to store and retrieve virtually unlimited amounts of data, from small text files to massive datasets, with high availability and low latency.

One of S3's key features is its scalability. Users can start with small amounts of storage and seamlessly scale up as their needs grow, without the need for upfront provisioning or capacity planning. This scalability is essential for businesses dealing with fluctuating data volumes or rapid growth.

S3 ensures the durability and availability of stored data through automatic replication across multiple geographically dispersed data centers. This redundancy means that even in the event of hardware failures or catastrophic events affecting one data center, data remains accessible and intact.

Security is paramount in S3, with robust encryption options available for both data in transit and at rest. Users can encrypt data using server-side encryption with keys managed by AWS Key Management Service (KMS) or client-side encryption for added security.

Access control mechanisms in S3 enable administrators to manage who can access data and how they can interact with it. This includes bucket policies, access control lists (ACLs), and integration with AWS Identity and Access Management (IAM), allowing for fine-grained control over permissions.

S3 offers lifecycle management policies, allowing users to define rules for automatically transitioning objects between storage classes or deleting them after a specified period. This feature helps optimize storage costs by moving less frequently accessed data to lower-cost storage tiers over time. Versioning support in S3 enables users to have multiple versions of an object stored in the same bucket, protecting against accidental deletion or modification of data.

Cross-region replication allows users to replicate data across different AWS regions, providing additional resilience against regional outages and ensuring data availability and continuity of operations.

S3 integrates seamlessly with other AWS services, enabling users to trigger events based on changes to objects in S3 buckets. This integration facilitates real-time processing of data and automation of workflows, enhancing the agility and efficiency of cloud-based applications.

Overall, Amazon S3 is a versatile and reliable storage solution that caters to a wide range of use cases, including backup and restore, content distribution, data lakes, and big data analytics. Its pay-as-you-go pricing model makes it cost-effective for businesses of all sizes, from startups to enterprises.

4.11 AWS CloudWatch

AWS CloudWatch is a robust monitoring and observability service offered by AWS for tracking the performance and health of applications and infrastructure deployed on the AWS cloud. It provides a comprehensive suite of tools and features to collect, visualize, analyze, and act upon data generated by various AWS resources and custom applications.

With CloudWatch, users can monitor metrics, log files, and events in real-time, gaining insights into the utilization, performance, and operational behavior of their AWS resources. Metrics monitoring allows tracking of key performance indicators such as CPU utilization, network traffic, and storage usage, while log monitoring enables the analysis of log files generated by AWS services and applications to identify and troubleshoot issues.

CloudWatch Events enables users to automate workflows by responding to system events and triggering actions based on predefined rules. This facilitates the implementation of automated remediation and event-driven architectures. Additionally, CloudWatch Alarms provide the capability to set thresholds on metrics and receive notifications or initiate automated actions when thresholds are breached, ensuring proactive monitoring and timely responses to performance issues. The service also offers customizable dashboards for creating visual representations of metrics and alarms, allowing users to monitor the health and performance of their AWS resources in a single, centralized view. Advanced analytics features such as metric math expressions, anomaly detection, and metric filters further enhance the ability to gain actionable insights from monitoring data.

4.12 AWS Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) offers a flexible and scalable cloud computing platform. EC2 offers users the opportunity to lease virtual servers, commonly referred to as instances, where they can deploy and operate their applications. This service is designed to provide developers with complete control over their computing resources while minimizing the need for upfront investment in hardware. EC2 instances come in various configurations to suit different workloads, ranging from small instances suitable for testing and development to powerful instances optimized for high-performance computing or memory-intensive tasks. Users can choose from a wide selection of operating systems, including popular choices like Linux and Windows, and customize their instances with additional software and configurations as needed. One of the key features of EC2 is its scalability. Users can quickly scale their compute capacity up or down based on demand, either manually or automatically using features like Auto Scaling. This elasticity allows applications to handle fluctuations in traffic or workload without the need for extensive capacity planning or provisioning.

EC2 offers a range of purchasing options to accommodate different use cases and budgetary requirements. On-demand instances provide flexibility with no long-term commitments, while Reserved Instances offer significant cost savings for predictable workloads with a one- or three-year commitment. Spot Instances allow users to bid for unused capacity at reduced rates, making them ideal for cost-sensitive or time-flexible workloads.

There are several types of EC2 instances tailored to different workloads and use cases, each with varying combinations of CPU, memory, storage, and networking capacity. Some common types include:

- General Purpose (e.g., t2, m5): Balanced CPU, memory, and network resources suitable for a wide range of applications.
- Compute Optimized (e.g., c5): Optimal for tasks demanding substantial computational power, particularly those reliant on high-performance CPUs.
- Memory Optimized (e.g., r5): Designed for memory-intensive applications such as databases, in-memory caches, and real-time big data analytics.
- Storage Optimized (e.g., i3): Optimized for applications that require high I/O performance, such as NoSQL databases, data warehousing, and Elasticsearch.
- Accelerated Computing (e.g., p3, g4): Equipped with specialized hardware accelerators like GPUs for demanding computational tasks such as machine learning, graphics rendering, and high-performance computing.
- Burstable Instances (e.g., t3): Designed for workloads that do not consistently require high CPU performance but may spike in usage.

4.13 AWS EC2 Auto Scaling

EC2 Auto Scaling offers a dynamic solution for managing the compute resources of EC2 instances. Its main goal is to automatically adjust the number of EC2 instances based on changes in demand, ensuring that applications maintain optimal performance without manual intervention. At the core of EC2 Auto Scaling are Auto Scaling Groups (ASGs). These groups serve as logical containers that house a collection of EC2 instances, defining the boundaries within which scaling operations occur. Users establish minimum, maximum, and desired instance counts within ASGs, providing guardrails for scaling actions.

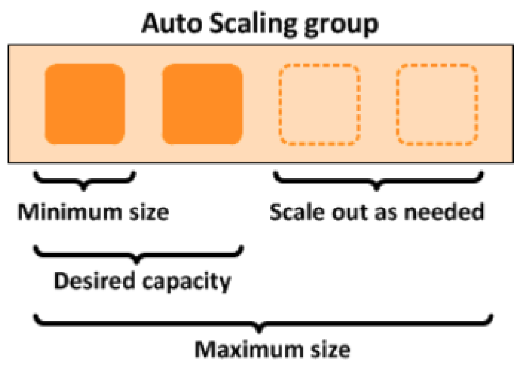


Figure 4.8: Auto Scaling Group [19].

The effectiveness of EC2 Auto Scaling lies in its ability to respond to varying workload patterns through the implementation of scaling policies. These policies can be categorized into two main types: dynamic scaling policies and scheduled scaling policies.

Dynamic scaling policies dynamically adjust the number of instances based on real-time metrics, such as CPU utilization, network traffic, or custom CloudWatch metrics. They enable the infrastructure to scale out during periods of increased demand and scale in during periods of decreased demand, ensuring that resources are aligned with workload requirements.

Scheduled scaling policies, on the other hand, allow users to define predetermined scaling actions based on anticipated workload changes. For instance, organizations can schedule scaling events to coincide with known traffic spikes or recurring operational patterns, preemptively adjusting instance capacity to accommodate expected changes in demand.

Integral to the functioning of EC2 Auto Scaling is its seamless integration with Amazon CloudWatch. CloudWatch serves as the monitoring and alarm system, collecting metrics from various AWS services, including EC2 instances. Users can configure alarms based on specific thresholds or conditions, triggering Auto Scaling actions when predefined criteria are met.

Furthermore, EC2 Auto Scaling incorporates health checks to maintain the overall health and reliability of the fleet of instances. Instances undergo periodic health checks to verify their status, and any instances deemed unhealthy are automatically replaced to maintain the desired level of service availability.

4.14 AWS Lambda

AWS Lambda stands as a pioneering service in the realm of serverless computing, offering developers an innovative approach to executing code without the need for traditional server provisioning and management. At its core, AWS Lambda enables event-driven execution, responding to a myriad of triggers from various AWS services or custom events defined by developers. This event-driven paradigm empowers developers to build highly scalable and responsive applications that react to changes in data, user actions, or system events in real-time.

Lambda functions, within the context of serverless computing, can be invoked in either synchronous or asynchronous manners, each catering to specific needs and architectural considerations. Synchronous invocation involves the direct triggering of a lambda function by a caller, which then waits for the function to complete its execution before proceeding further. This type of invocation is characterized by its immediacy and deterministic behavior. It is well-suited for scenarios where the caller requires an immediate response or relies on the result of the lambda function to continue its operations seamlessly. For instance, synchronous invocation is commonly employed in real-time processing tasks or when strict dependencies exist between the caller and the lambda function.

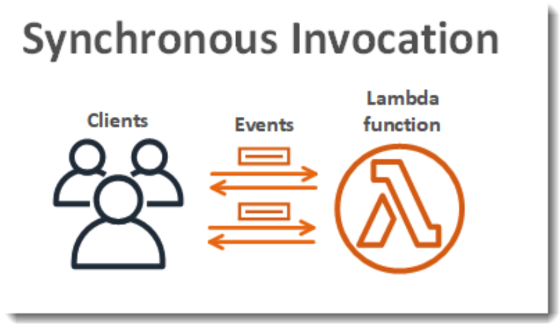


Figure 4.9: Synchronous Invocation of Lambda [20].

Conversely, asynchronous invocation does not entail immediate waiting for the completion of the lambda function. Instead, the caller initiates the function's execution and continues with its own tasks without blocking or waiting for the result. This asynchronous approach is particularly advantageous in scenarios where the caller does not need an instantaneous response or where parallel processing of tasks can enhance system throughput and efficiency. Asynchronous invocation aligns well with event-driven architectures, where lambda functions react to events generated by various sources, such as HTTP requests, database modifications, or message queue notifications. Furthermore, an asynchronous invocation is suitable for handling batch processing tasks or long-running operations where the immediate return of results is unnecessary, allowing resources to be utilized more efficiently.

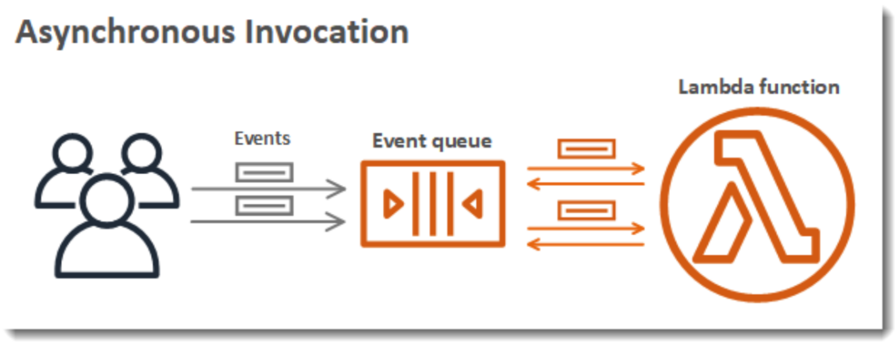


Figure 4.10: Asynchronous Invocation of Lambda [21].

One of the key features of AWS Lambda is its support for multiple programming languages, including Node.js, Python, Java, Go, Ruby, and .NET Core. This versatility allows developers to leverage their existing skills and choose the language that best suits their application requirements. Furthermore, AWS Lambda operates on a pay-per-use pricing model, where users are only charged for the compute time their code consumes and the number of requests it handles.

AWS Lambda's automatic scaling capabilities further enhance its appeal, dynamically provisioning resources to match the workload demands and scaling down to zero when idle. Lambda functions on AWS are designed to scale effortlessly based on incoming invocations. As more requests pour in, AWS automatically allocates additional resources to handle the load, ensuring optimal performance without any manual intervention. However, this scalability is not limitless. AWS imposes certain limits, one of which is the concurrency limit. This limit determines how many instances of a function can run simultaneously. When the number of incoming invocations surpasses this limit, AWS throttles the requests, preventing new invocations from being processed immediately. This inherent scalability ensures optimal performance and cost-efficiency, as users only pay for the resources they actually use. Additionally, AWS Lambda functions execute in stateless environments, eliminating the need to manage server state and facilitating seamless horizontal scaling.

Moreover, AWS Lambda integrates smoothly with an extensive array of AWS services, enabling developers to create complex serverless architectures that leverage the capabilities of other AWS offerings. Whether it's building data processing pipelines, web applications, or microservices, AWS Lambda provides the flexibility and scalability to support diverse use cases.

Chapter 5

AWS Security Lake

Amazon Security Lake is a fully managed service designed to streamline the centralization of security data from various sources, including AWS environments, SaaS providers, on-premises setups, cloud platforms, and third-party providers. The service creates a specialized data lake within your AWS account, utilizing Amazon Simple Storage Service buckets to store the information securely. All data stored in Security Lake follows the OCSF scheme.

Security Lake facilitates a comprehensive analysis of security data, offering a holistic view of the organization's security posture. This aids in enhancing the protection of workloads, applications, and data. The service can be centrally enabled across all available regions and multiple AWS accounts.

Data consumers referred to as subscribers, can access the stored data based on the level of access granted by the data owner.

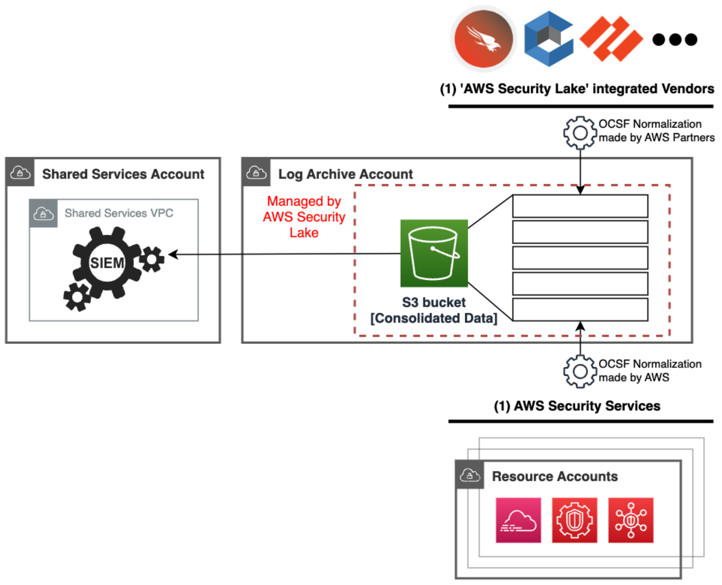

(1) 'AWS Security Lake' integrated Vendors
OCSF Normalization made by AWS Partners
Shared Services Account
Shared Services VPC
SIEM
Log Archive Account
Managed by AWS Security Lake
S3 bucket [Consolidated Data]
OCSF Normalization made by AWS
(1) AWS Security Services
Resource Accounts

Figure 5.1: Overview of Security Lake.

5.1 Source of Amazon Security Lake

The Amazon Security Lake can gather data from various origins, such as natively supported AWS services, integrated vendors with AWS Security Lake, and custom sources from third-party providers.

5.1.1 AWS Service Sources

Amazon Security Lake is capable of natively ingesting logs and events from various AWS services. It converts the Data into the OCSF scheme and saves them in the Security Lake in the efficient Parquet format. These services are:

- AWS CloudTrail is a service that tracks and records AWS API calls made within your account, capturing information from various sources such as AWS Management Console, SDK, command line tools, and specific AWS services. It provides a detailed history of who made API calls, what services were accessed, source IP addresses, and timestamps. Security Lake is capable of collecting logs for CloudTrail Management and CloudTrail Data events related to S3 and Lambda. These three types of events are treated as separate sources within Security Lake, each with a different source name.
- AWS Security Hub provides insights into your AWS security status and lets you assess your environment against security standards. It collects information from various sources, such as AWS services, third-party integrations, and checks against Security Hub controls, all presented in a standard format called AWS Security Finding Format (ASFF). When you link Security Hub findings with Security Lake, the platform instantly starts collecting these findings directly from Security Hub through an independent and duplicated stream of events. Furthermore, Security Lake converts the findings from ASFF to the OCSF. Importantly, Security Lake does not manage or alter your Security Hub findings or settings. It simply enables the collection and transformation of these findings for analysis, seamlessly integrating with your existing Security Hub configuration.
- Route 53 resolver query logs offer insights into the DNS activities of resources in your Amazon VPC, aiding in application performance analysis and security threat detection. By incorporating these logs into Security Lake, the platform efficiently acquires them directly from Route 53 through a redundant and independent event stream. Importantly, Security Lake's integration does not interfere with your existing Route 53 resolver query logging configurations. It smoothly collects and manages logs, adding an extra layer of security monitoring without disrupting your established setup.

- Amazon VPC Flow Logs is a feature that captures details about the IP traffic moving through network interfaces within your Amazon VPC environment. When you integrate VPC Flow Logs with Security Lake, the platform promptly begins gathering this information directly from Amazon VPC. It does so through a separate and duplicative stream of Flow Logs, without influencing the management of your VPC Flow Logs or making any changes to your Amazon VPC settings.

5.1.2 Integrated vendors

Amazon Security Lake integrates with several external vendors. Vendors implement one or more features among source integration, subscriber integration, or service integration. Source integrations send data to Security Lake in Apache Parquet format, adhering to the OCSF schema. Subscriber integrations can access source data from Security Lake via HTTPS endpoints, Amazon SQS queues, or directly query AWS Lake Formation. They are capable of reading data in Apache Parquet format and according to the OCSF schema. Examples of integrated vendors are CrowdStrike – Falcon Data Replicator, CyberArk – Unified Identify Security Platform, and Palo Alto Networks – Prisma Cloud.

5.1.3 Custom sources

Amazon Security Lake facilitates the integration of logs and events from external custom sources. It streamlines the process by assigning a unique prefix for each source in your Amazon S3 bucket and establishing an IAM role, granting the custom source the necessary permissions to deposit data into the security lake. Additionally, the service automatically creates an AWS Lake Formation table to efficiently organize the incoming data and configures an AWS Glue crawler. This crawler not only partitions the source data but also populates the AWS Glue Data Catalog with relevant table information, facilitating schema discovery for new source data.

To integrate a custom source with Security Lake, it must have the capability to write data as S3 objects under the assigned prefix. For sources with diverse data categories, each distinct OCSF event class should be treated as a separate source. Objects need to be partitioned based on source location, AWS Region, AWS account, and date, following a specific path format: “bucket-name/source-location/region=region/accountId=accountID/eventDay=YYYYMMDD”. The data in S3 objects should be formatted in Apache Parquet files, with a 1 MB limit for uncompressed data page size and a compressed row group size not exceeding 256 MB. Zstandard compression is preferred for Parquet objects. To maintain consistency, the same OCSF event class should be applied to all records within a Parquet-formatted object, and records should be ordered by time within the object to optimize data querying costs. When adding a custom source using Security Lake API or AWS CLI, it is essential to associate a specific IAM role to grant AWS Glue the authority to crawl the custom source data, identifying partitions for effective data organization and table management in the Data Catalog.

5.2 Subscriber of Amazon Security Lake

Amazon Security Lake enables subscribers to access logs and events with a focus on cost control and least privilege access. Subscribers can obtain Data access, receiving notifications of new source data in the S3 bucket. This access type, identified as "S3" in the CreateSubscriber API, allows notification through an HTTPS endpoint or polling an SQS queue. Alternatively, subscribers can have Query access, enabling direct querying of AWS Lake Formation tables using services like Amazon Athena. While Athena is the primary engine, other services like Amazon Redshift Spectrum and Spark SQL can also be used.

Subscribers are granted access to source data only in the selected AWS Region during subscriber creation. To extend access to multiple Regions, a roll-up Region can be specified, allowing contributions of data from other Regions. This approach ensures cost control and aligns with the principle of least privilege access.

5.3 Lifecycle of buckets S3

Security Lake allows users to customize data storage preferences in their preferred AWS Regions and set retention settings for cost-effective management. The data is stored as objects in Amazon S3 buckets, and retention settings correspond to Amazon S3 Lifecycle configurations. Users can specify the S3 storage class and time period for objects to stay in that class before transitioning or expiring at the Region level. If not configured, Security Lake defaults to storing data indefinitely in the S3 Standard storage class. Note that Security Lake does not support Amazon S3 Object Lock, and enabling it with default retention mode can disrupt data delivery.

A roll-up Region serves as a central hub that consolidates data from multiple contributing Regions. This consolidation is beneficial for ensuring compliance with regional data requirements. It helps gather information from various locations to meet specific regional data compliance standards.

Chapter 6

Design of Security Lake platform

In the realm of cybersecurity, the landscape is populated with various vendors and tools, some utilizing the OCSF language while others employ their own schemes. This diversity requires transformation and mapping in the OCSF format of data from sources that use their schemes, to enable seamless integration and interoperability. This process ensures that different tools can work together effectively, maximizing their collective potential in safeguarding digital environments against evolving threats.

6.1 Challenges

The challenge is to expand the Security Lake service of AWS, a service that allows you to already integrate tools that use OCSF, creating a platform that facilitates and speeds up the integration of third parties that use their scheme. This platform will streamline the process of transforming messages from various formats into OCSF, facilitating seamless integration with Security Lake.

Our objective is to implement Terraform code capable of deploying this platform, customizable according to the integrator's preferences. Integrators will have the flexibility to select the specific integrations they require and define the integration methods, inputting these specifications into designated fields within the Terraform configuration file.

The platform will offer the following key features:

- Message Transformation: Enable the conversion of messages from proprietary formats to OCSF, ensuring compatibility with Security Lake.
- Integration Flexibility: Empower integrators to choose the desired integrations and customize integration methods to suit their requirements.
- Efficient Deployment: Utilize Terraform automation to swiftly deploy the platform, minimizing manual configuration efforts and optimizing deployment efficiency.
- Scalability and Reliability: Design the platform to scale seamlessly with growing data volumes and maintain high levels of reliability and availability.

By developing this comprehensive platform, we aim to enhance the accessibility and usability of AWS's Security Lake service, empowering organizations to seamlessly integrate diverse security tools and maximize the effectiveness of their cybersecurity operations.

6.2 Platform's Overview

The platform is designed to have three different types of input sources and integrate the messages from these sources into the Security Lake, as illustrated in the yellow rectangle in Figure 6.1. Conceptually, the platform consists of two main parts:

- Catching part: This component, represented by the blue rectangle in Figure 6.1, manages the data acquisition process from various sources.
- Ingestion Pipeline [Normalization]: Highlighted by the red rectangle in Figure 6.1, this component normalizes data in the OCSF format and loads them into the Security Lake.

The platform provides a repository, "ADMIN Repository", for the administrator to define the integrations he wants to do. He can indicate the source, type of source, and details and methods related to each integration. Additionally, the administrator must indicate the mapping of messages from the original format to the OCSF format within this repository.

For each integration, the platform creates another repository, "Integration Repository", that allows for quick adjustments of mappings or to write of custom mapping functions tailored to specific integration needs.

By dividing the platform into these components and providing customizable integration options, the platform offers flexibility and efficiency in managing data from different sources.

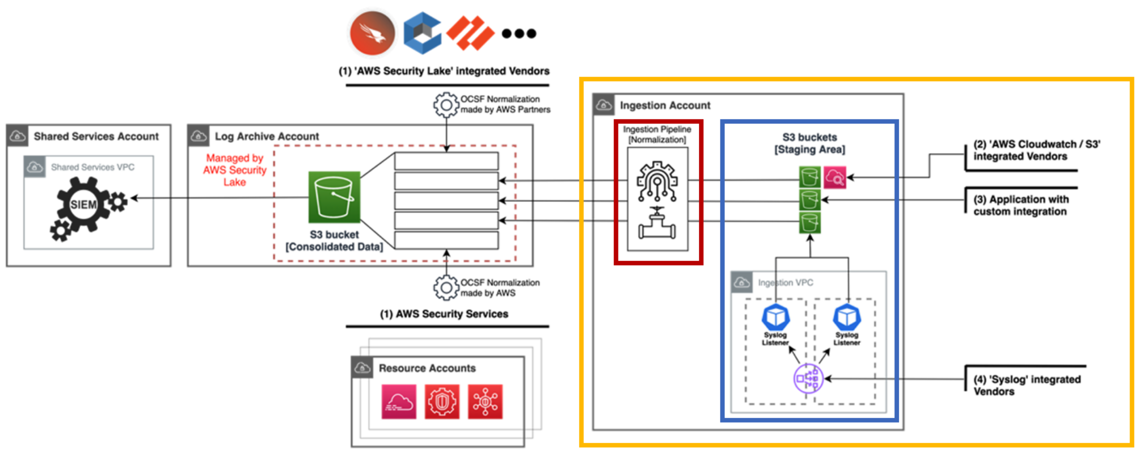

(1) 'AWS Security Lake' integrated Vendors
OCSF Normalization made by AWS Partners
Shared Services Account
Shared Services VPC
SIEM
Log Archive Account
Managed by AWS Security Lake
S3 bucket [Consolidated Data]
OCSF Normalization made by AWS
(1) AWS Security Services
Resource Accounts
Ingestion Account
Ingestion Pipeline [Normalization]
S3 buckets [Staging Area]
Ingestion VPC
Syslog Listener
Syslog Listener
(2) 'AWS Cloudwatch / S3' integrated Vendors
(3) Application with custom integration
(4) 'Syslog' integrated Vendors

Figure 6.1: Platform's Design.

6.2.1 Catching part

The platform can have three different source types:

- AWS S3 integrated Vendors: refer to third-party companies or service providers whose products or solutions have been designed to work seamlessly with Amazon S3. Integration in this context means that these vendors have developed their software, tools, or services to interact with S3, going to load messages and logs into an S3 bucket.
- An application with custom integration with S3 typically refers to a software application that interacts with Amazon S3 in a customized or tailored manner. In this case, an application that uploads logs or messages inside to an S3 bucket.
- Syslog integrated vendors: Common objects that utilize Syslog protocol. They include network devices like routers and switches, servers, operating systems, applications, and security devices.

For the first two sources, the platform develops only the Ingestion Pipeline part, because the messages are loaded directly into an S3 bucket.

As for the third source, the platform not only handles the mapping part but also facilitates the capture process. This involves generating a public IP for each integration, allowing external sources to transmit messages in syslog format.

6.2.1.1 Ingestion VPC

The platform has been created to offer an environment where each integration can have an application capable of providing a public IP. This allows the external source to send messages to the IP address, such messages are then normalized and integrated into the Security Lake. The application must buffer these messages and load them into a storage environment to make them available to the Normalization pipeline. It is essential that this application is scalable and always available. To do this, the platform creates a Virtual Private Cloud (VPC) consisting of three public subnets and three private subnets - one public and one private for each area of availability. Additionally, a Kubernetes EKS cluster is created for the three private subnets. The cluster contains nodes and pods that the application runs on, based on the integration specifications. This ensures that the application meets the requirements for scalability, redundancy, and availability.

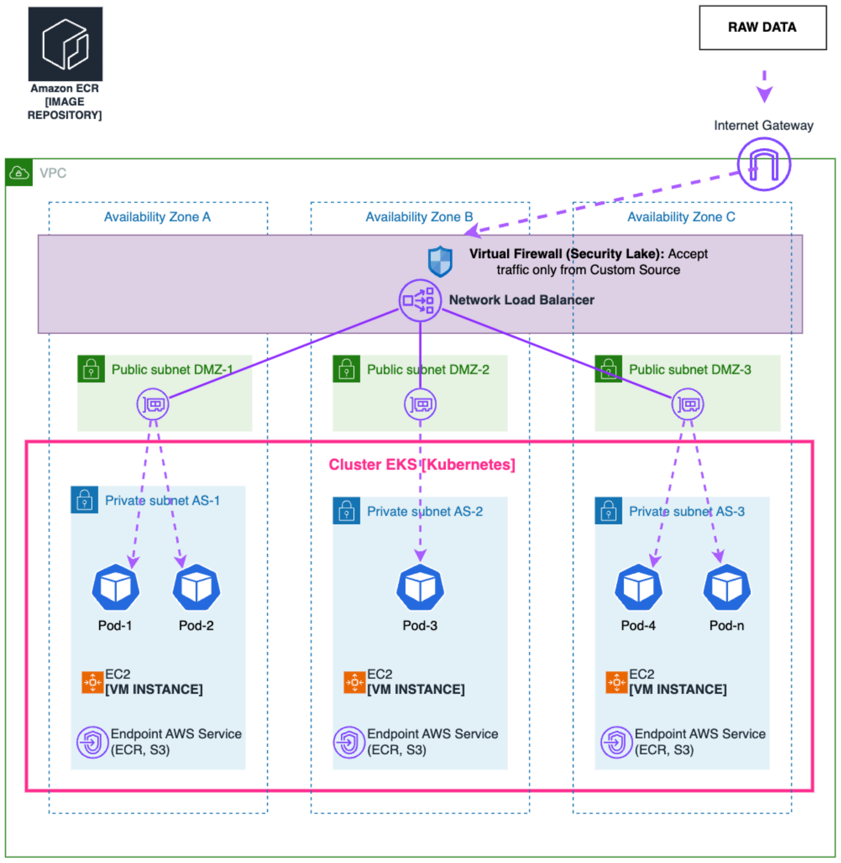


Figure 6.2: Ingestion VPC

6.2.2 Ingestion Pipeline [Normalization]

The platform employs an ingestion pipeline for each integration, designed to seamlessly handle the processing of incoming messages as they are deposited into storage containers. This pipeline is equipped to receive objects containing messages and convert each message into the OCSF format before uploading them to the Security Lake. This pipeline must be capable of efficiently handling high volumes of traffic and scaling in response to fluctuations in incoming data.

To achieve this, the platform leverages the Lambda service, creating a dedicated function for each integration. These functions are designed to be triggered automatically whenever a new object is created within the corresponding S3 bucket. Based on pre-defined configurations, each Lambda function processes the messages contained within the object, mapping into OCSF, intelligently grouping them before loading them into the Security Lake with the appropriate prefix.

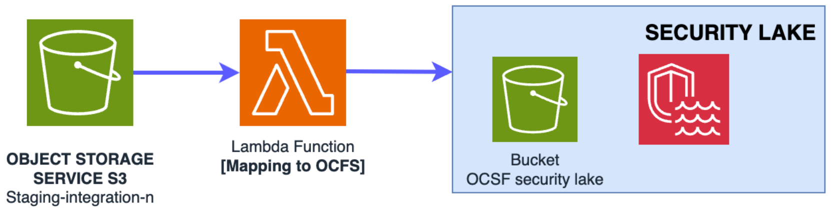


Figure 6.3: Ingestion Pipeline [Normalization].

6.2.3 Admin Repository

The idea is to create a centralized repository that allows the administrator to meticulously configure the integration, adapting it to specific requirements across different fields.

Key parameters include integration nomenclature, type (ranging from suppliers integrated in Syslog, applications with custom integrations, to vendors integrated in S3), the name of the Buckets S3 that the Security Lake uses to integrate logs, and JSON structures that host the mappings (where each key indicates a new field, OCSF format, and its value indicates how to extract the value from the original data to be assigned to the new field).

Administrators using “S3Integrated” integrations are required to provide the source S3 bucket name.

For the Syslog integration, administrators can fill in other attributes including:

- Dedicated nodes
- The desired number of nodes, together with minimum and maximum thresholds.
- Specifications such as image, disk size, and node instance type.
- Percentile consumption of cumulative CPU between nodes for dynamic scaling.
- Limits and requirements for CPU and memory per pod.
- Average CPU usage between pods for scaling purposes.
- Exposed port for communication.
- Application-specific parameters such as buffer size and flush interval inside pods.
- IP source

This approach provides granular control over integration configurations, facilitating custom configurations that support different operational needs.

6.2.4 Integration Repository

For each integration, the platform creates a dedicated repository. Its purpose is to facilitate and expedite mapping-related changes, such as reassigning specific fields to different attributes. It contains two JSON files identical to the JSONs that were placed in the previous repository during the initialization phase. These files are editable, allowing integrators to modify mappings as needed. When edited, the lambda function updates the mapping accordingly.

Moreover, integrators can directly write or modify the Python functions utilized by the lambda. This grants them the ability to create customized configurations for each integration, providing a powerful and flexible setup.

Chapter 7

Platform's Implementation

The implementation of the platform was crafted using Terraform code, a tool renowned for its prowess in orchestrating infrastructure as code. This choice was made with the intention of giving the platform flexibility and agility, enabling seamless configuration and swift deployment processes. By leveraging Terraform, we aimed to streamline the setup and management of our platform, allowing for rapid adjustments to meet evolving requirements. In addition, the adoption of Terraform represents a strategic investment in the scalability, reliability, and maintainability of our platform.

The construction of the platform unfolds through three primary steps:

1. In the first step, the builder of the platform launches the creation folder. This creates a repository dedicated to the platform administrator, where there is a platform bootstrap configuration, an administrator-configurable file, and a folder containing the files used for subsequent platform implementations. It also creates a pipeline that every time a commit is made, reads the files into the Admin Repository and, based on the previous configuration, updates/creates/deletes the resources. The first time, the pipeline reads the bootstrap file and creates the basic platform configuration.
2. In the second step, the platform integrator/administrator creates and configures preset modules within the repository. He specifies the type and method of integration desired. Once committed, the pipeline creates the infrastructure for the catching part. In addition, an Integration Repository and a pipeline are created for each integration. These repositories house essential data for the creation of the normalization ingestion phase, while the pipeline is equipped to interpret the repository files, constructing the normalization ingestion components accordingly.
3. The third step is predominantly automated. The Integration Pipeline seamlessly processes files uploaded by the initial pipeline into the Integration Repository. It then proceeds to develop the normalization component. However, users retain the flexibility to utilize the Integration Repository for swift mapping alterations (from raw data to OCSF) or to create a custom function to be used by the lambda instead of the basic one.

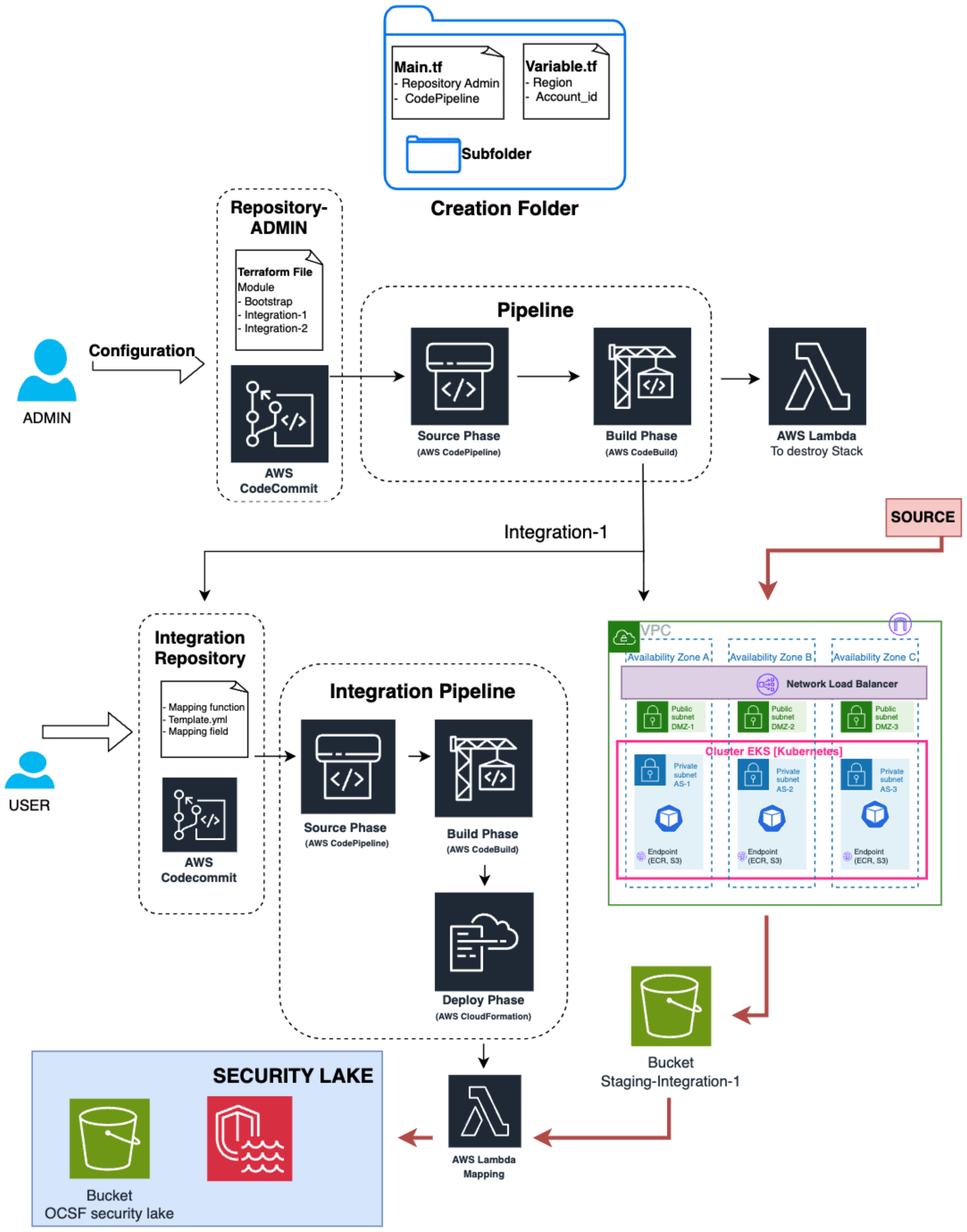


Figure 7.1: Platform's implementation.

7.1 STEP 1 of Platform's construction

Step 1 comprises two distinct sections:

- The initial part delineates the Terraform files in the creation folder and describes the resources defined within them.
- The subsequent segment delineates the services and architectures instantiated within the AWS Cloud subsequent to deploying the construction folder.

7.1.1 Platform creation folder

The platform creation folder contains multiple Terraform files and one subfolder:

- main.tf: This file contains the primary Terraform configuration for initiating the first step of building the platform. It includes resource definitions, providers, and other necessary configurations.
- variable.tf: This file contains variables related to where to create the platform.
- Subfolder: This subfolder contains additional Terraform files specifically dedicated to the second step of the platform creation process.

Organizing the Terraform project in this manner maintains a structured approach, separating different stages or components of the infrastructure deployment process to improve the clarity, versatility, and scalability of the platform.

7.1.1.1 File variable.tf

In this document, you can find the section where you can configure variables that match the location where you want to create the platform. Specifically, you will need to populate the default field pertaining to the account and region where you intend to establish the platform.

Listing 7.1: File variable.tf of creation folder

```
#region
variable "region" {
    type = string
    default = "eu-west-1"
}
#account id
variable "account_id" {
    type = string
```

```
  default = "224106000250"
}
```

7.1.1.2 *File main.tf*

The first configuration block is used to define the project requirements in detail:

- required_providers: This section instructs Terraform on which cloud service providers are needed to manage infrastructure resources. In this case, the project specifies the AWS provider provided by HashiCorp. The version key sets a restriction on the provider's version to use, indicating an approximate range of allowed versions.
- required_version: This section specifies the minimum version of Terraform required to run the project correctly.

Listing 7.2: Terraform configuration, into file main.tf of creation folder

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~>␣4.16"
    }
  }
  required_version = ">=␣1.2.0"
}
```

Next, it proceeds to configure the AWS provider by setting up the block. Within this block, the region is specified utilizing a variable named "region".

Listing 7.3: AWS Provider configuration, into file main.tf of creation folder

```
provider "aws" {
  region = var.region
}
```

The source code to establish the CodeCommit repository is delineated. This repository will serve as the container for the Terraform code required for the subsequent phase of platform creation. Additionally, it offers the admin the flexibility to configure desired integrations seamlessly.

Listing 7.4: AWS CodeCommit Repository, into file main.tf of creation folder

```
#create an Admin repository
resource "aws_codecommit_repository" "codecommit_repository_admin" {
  repository_name = "repositoryNameAdmin"
  description     = "This␣is␣the␣Repository␣of␣Admin"
}
```

To set up a CodePipeline, you need to configure some components:

1. S3 Bucket for Pipeline Artifacts: This resource establishes an S3 bucket dedicated to housing pipeline artifacts, facilitating storage and retrieval throughout the pipeline's lifecycle.
2. IAM Role for CodePipeline: This role allows CodePipeline to assume a role with the necessary permissions to perform pipeline actions. At role is associated also with policy as permissions for access to the S3 bucket of Pipeline Artifacts ("s3:GetObject", "s3:PutObject", etc), permissions related to CodeCommit that is defined before ("codecommit:GetBranch", "codecommit:GetCommit", etc), and permissions for CodeBuild ("codebuild:BatchGetBuilds", "codebuild:-StartBuild").
3. AWS CodeBuild project: The goal of the AWS CodeBuild project is to automate infrastructure construction by launching Terraform code, using AWS CodePipeline for source and artifact management.

Listing 7.5: AWS CodeBuild, into file main.tf of creation folder

```
resource "aws_codebuild_project" "create_project" {
  name          = "project"
  description   = "create_codebuild_project"
  build_timeout = "40"
  service_role  = aws_iam_role.create_project_role.arn

  artifacts {
    type = "CODEPIPELINE"
  }

  environment {
    compute_type        = "BUILD_GENERAL1_SMALL"
    image               = "aws/codebuild/standard:5.0"
    type                = "LINUX_CONTAINER"
    privileged_mode     = true
  }

  source {
    type         = "CODEPIPELINE"
    buildspec    = buildspec-tf-create.yaml
  }
}
```

Within the "aws_codebuild_project" resource block, various attributes are specified, including project name, description, build timeout, service role, and build environment settings. Notably, it is configured to utilize AWS CodePipeline for both source and artifact management, while also defining the build environment with specifics like processing type, Docker image, and privileged mode.

The source block within this resource configures how the source code is fetched for the build process, opting for CodePipeline as the source type and referencing a local YAML file known as "buildspec-tf-create.yaml". This file delineates the version and steps of the build process, consisting of three stages: install, pre_build, and build.

During the installation phase, the necessary dependencies are installed, including the download and installation of Terraform. The pre_build phase initializes Terraform, preparing it for subsequent actions. Finally, the build phase executes the Terraform commands to schedule and apply changes to the infrastructure. Generate a Terraform execution plan and apply it to make the desired changes to your infrastructure.

```
version: 0.2
phases:
  install:
    commands:
      - "curl -s https://releases.hashicorp.com/terraform/1.3.6/
    terraform_1.3.6_linux_amd64.zip -o terraform.zip"
      - "unzip terraform.zip -d /usr/local/bin"
      - "chmod 755 /usr/local/bin/terraform"
  pre_build:
    commands:
      - "terraform init"
  build:
    commands:
      - "terraform plan -out=terraform.tfplan"
      - "terraform apply terraform.tfplan"
```

```

```

Listing 7.6: File buildspec-tf-create.yaml of creation folder

Start by configuring the pipeline with a distinguished name and specifying the IAM role ARN for performing the actions. Set up the artifact store using an S3 bucket to safely store the pipeline artifacts. Next, define the "Source" step to retrieve the source code. Use CodeCommit as the source provider and configure it with the repository name and branch name. Stores artifacts generated at this stage in the designated location are typically referred to as "source_output".

Listing 7.7: Source stage CodePipeline, into file main.tf of creation folder

```
stage {
    name = "Source"

    action {
      name              = "Source"
      category          = "Source"
      owner             = "AWS"
      provider          = "CodeCommit"
      version           = "1"
      output_artifacts  = ["source_output"]

      configuration     = {
        RepositoryName   = aws_codecommit_repository.
   codecommit_repository_admin.repository_name
        BranchName       = "main"
      }
    }
  }
```

Finally, set the Terraform deployment phase to run Terraform files to advance the platform creation. Use CodeBuild for this step, specifying the input artifacts from the previous step and referring to the CodeBuild project with its name. This structured approach ensures a smooth flow from source code recovery to Terraform execution, facilitating an automated distribution process.

Listing 7.8: Build stage CodePipeline, into file main.tf of creation folder

```
stage {
    name = "terraform_create"
```

```

  action {
    name             = "terraform_create"
    category         = "Build"
    owner            = "AWS"
    provider         = "CodeBuild"
    input_artifacts  = ["source_output"]
    version          = "1"

    configuration    = {
      ProjectName = aws_codebuild_project.create_project.name
    }
  }
}
```

Then a "null_resource" called "push_to_codecommit_admin" is defined. This resource uses a local-exec provisioner to run a series of Git commands within the subdirectory with files to load into the repository. Commands include setting the default branch to "main", configuring Git credentials to interact with AWS CodeCommit, initializing a new Git repository, adding all files in the directory to the repository, committing the changes with a message indicating the addition of Terraform files and push the changes to the CodeCommit repository created earlier.
It also uses the "depends_on" attribute to specify that this resource depends on the creation of an AWS CodeCommit repository and an AWS CodePipeline. This ensures that Git commands run only after these resources have been created or updated successfully.

7.1.2 Architectural landscape after launched creation folder

Once you have navigated to the directory containing the Terraform files and executed the necessary commands (terraform init, terraform plan, terraform apply), an Amazon CodeCommit repository will be created in your AWS account. This repository will contain:

1. Modules Folder: This directory will house various files essential for building the second stage of the platform.
2. bootstrap.tf File: Within this file, you will find modules referencing Terraform files from the "modules" folder. These modules are designed to establish a common architecture present in any integration you wish to undertake.
3. main.tf File: Here, integrators or administrators can specify preset Terraform modules, detailing the type of integration they wish to implement and their

preferred configuration.

Additionally, a CodePipeline will be set up to read the Terraform files within the repository. This pipeline will automatically create AWS resources as specified in those files whenever a commit occurs. This ensures seamless deployment and resource management in response to changes made to the infrastructure configuration.

After the repository and pipeline are created, the pipeline will read the bootsrap.tf file and create the basic platform architecture. The next section describes the configuration of the bootstap.tf file and what resources it creates.

The following figure shows the architecture created by launching the creation folder.

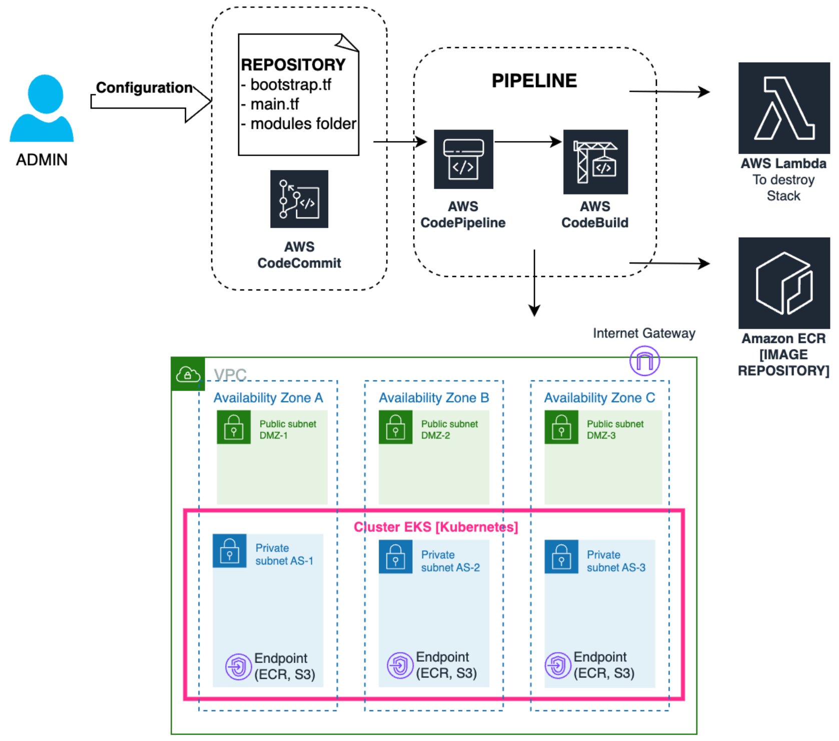


Figure 7.2: Architectural landscape after launched creation folder

7.1.2.1 File bootstrap.tf into Repository-Admin

The bootstrap file encompasses a Terraform module designed to orchestrate the creation of default platform resources. These resources include:

- VPC Configuration: Formation of a Virtual Private Cloud with a well-architected setup. Division into three public subnets and three private subnets, ensuring redundancy and fault tolerance. Allocation of one public and one private subnet for each availability zone. Establishment of a Kubernetes EKS cluster within the three private subnets for enhanced container orchestration capabilities.
- Elastic Container Registry: Creation of an Amazon ECR tailored to house the Fluentd image for seamless integration with Kubernetes Pods.
- Lambda Function: Implementation of a Lambda function specifically designed for the purpose of dismantling CloudFormation stacks. This function is pivotal in the third phase of the implementation, providing a streamlined and automated approach to stack termination.

Ingestion VPC

The platform creates a Virtual Private Cloud. It is a virtual network environment that provides and manages the isolated section of the AWS cloud. A VPC provides a high degree of control over network configuration, including the ability to define the range of IP addresses, create subnets, configure route tables, and manage security settings. Since there are no additional costs associated with using VPC, it is automatically generated as a default configuration.

In VPC configuration, the platform defines an IPv4 CIDR block, which specifies the range of private IP addresses that can be used within the VPC.

The platform activates DNS hostnames in the VPC, which on AWS automatically assigns user-friendly names to instances, simplifying identification and communication. This feature supports internal DNS resolution for service discovery and communication within the VPC. It is also critical for services like AWS Elastic Load Balancing that require DNS hostnames for proper resolution.

The platform activates DNS support in the VPC, which is crucial for smooth communication with external resources and services. Allows instances to resolve domain names, ensuring proper functionality, software updates, and access to AWS services.

The platform assigns the VPC a Tag that provides a way to facilitate identification within the AWS environment.

Listing 7.9: Virtual Private Cloud

```
#create vpc
resource "aws_vpc" "vpc" {
  cidr_block                      = "10.0.0.0/16"
  instance_tenancy                = "default"

  enable_dns_hostnames  = true
  enable_dns_support    = true
  tags = {
    Name = var.tags_vpc_name
  }
}
```

Now we go to describe one by one the components created within the VPC from the bootstrap Terraform code.

Subnet

In the VPC there are 3 private subnets and 3 public subnets. Subnet is a logical division of an Amazon Virtual Private Cloud. To specify the VPC to which a subnet belongs, you typically provide the "vpc_id" attribute in your AWS subnet resource definition.

The IPv4 CIDR block specifies the IP address range for the subnet.

Availability Zone for the subnet designates the specific availability zone in which the subnet will be created. Usually, there are 3 zones available for each region, in my implementation, it goes to create a public subnet and a private one for each zone.

In the public subnet the field "map_public_ip_on_launch" is set to true, to indicate that instances launched into the subnet should be assigned a public IP address [22]. To enable Kubernetes to provision a network load balancer on public subnets, those subnets need specific tags. These tags include "kubernetes.io/cluster/my-cluster" set to "shared" and "kubernetes.io/role/elb" set to 1.

Listing 7.10: Public Subnet

```
resource "aws_subnet" "DMZ-primary" {
  vpc_id         = aws_vpc.vpc.id
  cidr_block     = "10.0.6.0/24"
  tags           = {
    Name = "DMZ-a"
    "kubernetes.io/cluster/${var.eks_name}" = "shared"
    "kubernetes.io/role/elb" = 1
  }
  map_public_ip_on_launch   = true
  availability_zone         = data.aws_availability_zones.available.
   names[0]
}
```

Internet Gateway

Into VPC, there is an Internet Gateway to facilitate bidirectional communication between resources within the VPC and the Internet.

Listing 7.11: Internet Gateway

```
resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.vpc.id
}
```

Route Table

A route table is created and configured to create a default location for an Internet Gateway. It is associated with each public subnet (DMZ-1, DMZ-2), thus facilitating the flow of network traffic, especially for public elements.

Listing 7.12: Route table and its association

```
resource "aws_route_table" "route_table_vpc" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }
}

```

```
resource "aws_route_table_association" "assoc-DMZ-1" {
  subnet_id           = aws_subnet.DMZ-primary.id
  route_table_id      = aws_route_table.route_table_vpc.id
}
```

Endpoints

Each private subnet has two endpoints: one relative to S3, to load messages into buckets of staging, and another to ECR service to download a image to Pod.

Elastic Kubernetes Service

An Amazon Elastic Kubernetes Service (Amazon EKS) is established, within these three private subnets. An IAM role is generated specifically for the Amazon EKS cluster, accompanied by a tailored policy. This IAM policy document authorizes the EKS service to assume the designated role. Furthermore, the "AmazonEKSCluster-Policy" policy is linked to the IAM role, endowing it with the essential permissions required for operation.

Listing 7.13: Elastic Kubernetes Service

```
resource "aws_eks_cluster" "aws_eks_cluster" {
  name      = var.eks_name
  role_arn  = aws_iam_role.aws_iam_role_cluster.arn
  vpc_config {
    endpoint_private_access     = true
    endpoint_public_access      = true
    subnet_ids                  = [aws_subnet.AS-primary.id, aws_subnet
   .AS-secondary.id, aws_subnet.AS-tertiary.id]
  }
}
```

The platform also configures a Kubernetes provider for the Amazon EKS cluster. Specifies the cluster endpoint and CA certificate, and uses the AWS CLI to authenticate with the cluster and retrieve a token.

Listing 7.14: Provider Kubernetes

```
provider "kubernetes" {
  host                   = aws_eks_cluster.aws_eks_cluster.endpoint
  cluster_ca_certificate = base64decode(aws_eks_cluster.aws_eks_cluster.
   certificate_authority[0].data)
  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name", aws_eks_cluster.
   aws_eks_cluster.name]
    command     = "aws"
  }
}
```

Creation and setup of ECR

The platform creates an Elastic Container Repository where to contain the image used in Pods, in the integration with the Syslog source. To create such an image, the subfolder contains the files Dockerfile and fluent.conf used its creation. The platform takes care of launching the "docker build" commands to create the appropriate image and "docker push" to load it into the repository.

Dockerfile

The Dockerfile provided sets up Fluentd within a Docker container on a Linux platform using the base image "fluent/fluentd:v1.16-1". It employs the "USER root" directive to switch to the root account within the container, allowing subsequent commands to be executed with elevated privileges. Following this, two RubyGems packages ("fluent-plugin-s3" and "fluent-plugin-cloudwatch-logs") are installed using the gem install command. These plugins extend Fluentd's capabilities by enabling it to interact with Amazon S3 and CloudWatch Logs services, respectively. The "--no-document" flag is used to skip the generation of documentation during the installation process, reducing unnecessary overhead. Then, the Dockerfile copies a configuration file named "fluent.conf" into the "/fluentd/etc/" directory within the container. This configuration file will be described in the next section. Finally, the Dockerfile switches the user context back to fluent. This is typically done for security reasons, as running the application with root privileges can pose security risks.

```
FROM --platform=linux/amd64 fluent/fluentd:v1.16-1
USER root
RUN gem install fluent-plugin-s3 --no-document
RUN gem install fluent-plugin-cloudwatch-logs --no-document
COPY fluent.conf /fluentd/etc/
USER fluent
```

Listing 7.15: Dockerfile Fluentd

Fluentd Configuration (fluent.conf)

The fluent.conf file contains the configuration for Fluentd, specifying how it should handle incoming log data and where it should route it. Let's break down the key components:

- <source> block: This section defines a syslog source, indicating that Fluentd should listen for incoming syslog messages on a port specified by the "PORT_EXPOSE_SERVICE" environment variable. The messages are expected to arrive via TCP transport. Additionally, it binds to all available network interfaces specified by the "ADDRESS_SOURCES" environment variable. The <tags> block the incoming messages as system. The tag itself is generated by the tag prefix, facility level, and priority. tag = @tag.facility.priority. The <parse> block specifies that the message format should be detected. Supported values are rfc3164, rfc5424 and auto. Auto is useful when in_syslog receives both rfc3164 and rfc5424 messages per source. in_syslog detects message format by using message prefix and parses it [23].
- <match> block: This section matches log records with the tag "system.*.*". Log records matching this tag are then sent to an S3 bucket specified by the "STAGING_BUCKET" environment variable. The S3 bucket's region is defined by the "REGION" environment variable. Logs are stored under a path determined by the "POD_NAME" environment variable. To optimize performance and reliability, a <buffer> block is included, configuring buffering settings for the S3 output plugin. Logs are buffered to the local disk with a chunk limit size, determined by the "CHUNCK_SIZE" environment variable, and a flush interval, determined by the "FLUSH_INTERVAL" environment variable.

```
<source>
  @type syslog
  port "#{ENV['PORT_EXPOSE_SERVICE']}"
  <transport tcp>
  </transport>
  bind "#{ENV['ADDRESS_SOURCES']}"
  tag system
  <parse>
    message_format auto
  </parse>
</source>

<match system.*.*>
```

```
  @type s3
  slow_flush_log_threshold 30.0
  s3_bucket "#{ENV['STAGING_BUCKET']}"
  s3_region "#{ENV['REGION']}"
  path "#{ENV['POD_NAME']}"
  <buffer tag>
    @type file
    path /fluentd/log/
    chunk_limit_size "#{ENV['CHUNCK_SIZE']}"
    flush_interval "#{ENV['FLUSH_INTERVAL']}"
  </buffer>
</match>
```

Listing 7.16: fluent.conf

Lambda to destroy a CloudFormation Stack

The platform creates a Lambda feature capable of destroying the CloudFormation Stack, created in the third phase of the implementation of the platform. This function is invoked when an integration has been discontinued.

Configure the AWS Lambda function, it begins by creating a zip file named "deleteStack.zip" using the "archive_file" block. This zip file contains the code from the file "modules/bootstrap/deleteStack.py". The "source_file" attribute specifies the path to the Python file, while the type attribute specifies that it is a zip file. The "output_path" attribute determines where the resulting zip file will be saved.

Next, the "aws_lambda_function" block is used to define the Lambda function itself. It sets various properties:

- function_name: This is the name of the Lambda function.
- role: This role grants necessary permissions to the Lambda function, such as "cloudformation:DeleteStack", "cloudformation:DescribeStacks".
- handler: This specifies the entry point to the Lambda function within the zip file.
- runtime: The runtime environment for the Lambda function is specified as Python 3.8.
- filename: The name of the zip file containing the Lambda function code is set to "deleteStack.zip".

- environment: This block allows you to set environment variables for the Lambda function. Here, a variable named "region_name" is defined with the value of the Terraform variable "var.region".

Listing 7.17: AWS Lambda: Delete Stack

```
data "archive_file" "lambda" {
  type        = "zip"
  source_file = "modules/bootstrap/deleteStack.py"
  output_path = "deleteStack.zip"
}
resource "aws_lambda_function" "remove_stack" {
  function_name         = "deleteStack"
  role                  = aws_iam_role.role_lambda_deleteStack.arn
  handler               = "deleteStack.lambda_handler"
  runtime       = "python3.8"
  filename              = "deleteStack.zip"
  environment {
     variables = {
      region_name = var.region
    }
  }
}
```

Listing 7.18: deleteStack.py

```
import boto3
import os
import json
region_name = os.environ['region_name']
def lambda_handler(event, context):
    try:
        s = event['detail']['requestParameters']['name']
        t = s.split('-', 1)
        print(t[1])
        cloudformation = boto3.client('cloudformation', region_name=
    region_name)
        response = cloudformation.delete_stack(StackName=t[1])
        print(response)
        return response
    except Exception as e:
        return f"An error occurred: {str(e)}"
```

7.2 STEP 2 of Platform's construction

During the second stage of platform construction, the description is divided into two parts. The first part explains how administrators can compile the main.tf file in the Admin-Repository. It describes the various fields that can be filled out based on the three types of integration. The second part covers the Terraform code used to implement the catching part and the architecture that creates the normalization pipeline part.

7.2.1 Modules to declare an integration

In the "main.tf" file in the Admin-Repository, the administrator can write terraform modules with the preset fields, to declare the integrations he wants to do. One module for each integration. The modules can be of different types depending on the type of integration you want to do:

- Integration S3 integrated vendors/Applications with custom integration S3
- Syslog integrated vendors
- Syslog integrated vendors with dedicated nodes

All modules have common fields that are:

- type (string): the type of integration ("S3Integrated", "SyslogIntegrated", "SyslogIntegratedDedicatedNodes")
- integration_name (string): integration name
- source(string): consisting of "./modules/<type>", in the field between "< >" enter the integration type.
- buckets_security_lake (string): name of buckets security lake
- mapping_json: The JSON describes the mapping must do the lambda function. Keys can be in the simple case in the format "attribute_OCSF" or in the case of an object attribute in the format "attribute_OCSF.attribute_1_OCSF. ...". In the case of object attributes with multiple fields to be entered, the administrator must write an association line for each field. In JSON, such fields associated with the same object attribute must be written sequentially. The JSON can contain structured data in a variety ways:

  1. Simple value {"attribute_OCSF" : "field_initial_format"}: Where key is the OCSF attribute and value is the field name in the format to be

converted.

2. The extracted value corresponding to “field_initial_format” is compared with the keys of an enum dictionary to choose the appropriate OCSF value.

   ```
   {
       "attribute_OCSF": {
           "field_initial_format": {
               "val_1": "OCSF_value_1",
               "val_2": "OCSF_value_2",
               ...
           }
       }
   }
   ```

   Where “attribute_OCSF” is the OCSF attribute and “field_initial_format” is the field name in the format to be converted. “OCSF_val_1” is the value to be entered in the field if the value “value_1” corresponds to the value in the field “field_initial_format” of the log to be converted.

3. Extract a specific value within the value associated with the “field_initial_format” field.

   ```
   {
       "attribute_OCSF": {
           "field_initial_format": "regex"
       }
   }
   ```

   Where “attribute_OCSF” is the OCSF attribute, “field_initial_format” is the field name in the message to be converted, and “regex” is the regular expression used to capture the right value in the value associated with the “field_initial_format” key.

4. Extract a value through a regular expression from the value of a message field to be mapped. The extracted value is compared with the keys of an enum dictionary to choose the appropriate OCSF value.

   ```
   {
       "attribute_OCSF": {
           "field_initial_format": {
   ```

```
                    "regex": {
                        "val_1": "OCSF_value_1",
                        "val_2": "OCSF_value_2",
                        ...
                    }
                }
            }
        }
```

Where "attribute_OCSF" is the OCSF attribute and "field_initial_format" is the field name in the format to be converted, and "regex" is the regular expression used to capture the right value in the value associated with the "field_initial_format" key. Once the value is extracted from the regex, it searches the dictionary for the key "val_n" corresponding to this value. And it assigns the "OCSF_value_n" to "attribute_OCSF".

- add_field_static: JSON that contains all attributes in OCSF format with the same value in the integration. Examples are classification attributes and those that identify the source.

7.2.1.1 Modules type "S3Integrated"

In the integrations of type "S3Integrated" the platform does not develop the catching part as the logs to be converted are already loaded directly into a bucket S3. As a result, the module for these integrations must include the "buckets_source" field (string) to specify where the platform can obtain the input data.

Listing 7.19: Example of module with type "S3Integrated"

```
module "integrationWAF" {
  type = "S3Integrated"
  integration_name = "integrationWAF"
  source = "./modules/S3Integrated"
  bucket_destination_ocsf = "aws-security-data-lake-eu-west-1-kgie"
  buckets_source = "waf-1"
  mapping_json = { "dst_endpoint.ip" : "dest_ip", ...}
  add_field_static = {"class_name" : "Web␣Resources␣Activity", ...}
}
```

7.2.1.2 Modules type "SyslogIntegrated"/ "SyslogIntegratedDedicatedNodes"

Syslog integration can be done in two ways, using common nodes ("SyslogIntegrated") between multiple integrations or dedicated nodes ("SyslogIntegratedDedicatedNodes"). For this last type the fields can be filled in the form: "desired_size_nodeGroup" (int), "max_size_nodeGroup" (int), "min_size_nodeGroup" (int), "ami_type" (string), "disk_size" (int), instance_types ([string]), "average_CPU_usage_nodes" (int).

The fields common to both types of integrations are:

- "min_replicas" (int): minimum number of replicates in pods
- "max_replicas" (int): maximum number of replicates in pods
- "port_expose_service" (int): port where the service is exposed
- "pod_limits_cpu" ("milliCPU")
- "pod_limits_memory" ("Mebibyte")
- "pod_requests_cpu" ("milliCPU")
- "pod_requesta_memory" ("Mebibyte")
- "average_CPU_usage_pods" (int)
- "IP_source" (String <0.0.0.0>)
- "buffer_size" (kilobyte/kibibyte/megabyte/mebibyte)
- "flush_interval" (seconds/minutes)

Listing 7.20: Example of module with type "SyslogIntegratedDedicatedNodes"

```
module "integrationWAF" {
  type = "␣SyslogIntegratedDedicatedNodes"
  integration_name = "integrationSyslog"
  source = "./modules/␣SyslogIntegratedDedicatedNodes"
  bucket_destination_ocsf = "aws-security-data-lake-eu-west-1-kohg"
  mapping_json = { "actor.invoked_by" : "ident", ...}
  add_field_static = {"class_name" : "␣Process␣Activity", ...}
  desired_size_nodeGroup = 3
  max_size_nodeGroup = 7
  min_size_nodeGroup = 2
  ami_type = "AL2_x86_64"
  disk_size = 5
  instance_types = ["t2.micro"]
  average_CPU_usage_nodes = 60
  min_replicas = 4
  max_replicas = 50
  port_expose_service = 5140
  pod_limits_cpu = 1000m
  pod_limits_memory = "1024Mi"
  pod_requests_cpu = 1000m
  pod_requests_cpu = "1024Mi"
  average_CPU_usage_pods = 50
  IP_source = "23.0.0.23"
  buffer_size = 256k
  flush_interval = 1m
}
```

7.2.2 Implementation Catching Part

This section specifically caters to sources categorized under the "Syslog" type. This differentiation is necessary as sources labeled as "S3 integrated Vendors" and "Application with custom integration" already deposit data directly into S3 buckets. Consequently, the Ingestion Pipeline can seamlessly retrieve data from these containers.

The following constructs are built on the foundation of the Ingestion VPC and its components.

7.2.2.1 Node Group and Autoscaling policy

The administrator has the flexibility to choose between utilizing shared nodes, which are used by multiple integrations, or dedicated nodes, exclusive to their integration. If there are one or more integrations opting for shared nodes, the platform will generate a Node Group resource and associate it with an Autoscaling policy. Alternatively, for integrations requiring dedicated nodes, the platform will create a separate Node Group resource and Autoscaling policy for each integration. This allows the administrator to customize the configuration for each integration as needed. Nodes are provisioned within the private subnets of the cluster. Configuration parameters include the desired number of nodes, along with specified minimum and maximum thresholds. Additionally, settings for Amazon Machine Image type, instance type, and disk size are defined. Each group node is allocated a role governed by the following policies: "AmazonEKSWorkerNodePolicy", "AmazonEKS_CNI_Policy", "AmazonEC2ContainerRegistryReadOnly", and "CloudWatchAgentServerPolicy". Furthermore, a label is assigned to ensure pods are deployed on the appropriate nodes.

Listing 7.21: AWS EKS Node Group for dedicate Nodes

```
resource "aws_eks_node_group" "aws_eks_node_group" {
  cluster_name    = var.global_params["eks_name"]
  node_group_name = var.node_group_name
  node_role_arn   = aws_iam_role.role_node_group.arn
  subnet_ids      = [var.global_params["subenet_1_id"], var.global_params
    ["subenet_2_id"], var.global_params["subenet_3_id"]]
  scaling_config {
    desired_size = var.desired_size_nodeGroup
    max_size     = var.max_size_nodeGroup
    min_size     = var.min_size_nodeGroup
  }
  ami_type = var.ami_type
  disk_size = var.disk_size
  instance_types = var.instance_types

  update_config {
    max_unavailable = 1
  }
  labels = {
    nodegroup = "${var.node_group_name}-label"
  }
}
```

AWS Autoscaling Policy utilizes the Target Tracking Scaling methodology and seamlessly integrates with an existing AWS EKS node group configuration. The essence of this policy lies in its proactive management of resource scaling based on the average CPU utilization metric obtained from CloudWatch. By setting a target value, the policy orchestrates adjustments to the capacity of the autoscaling group, ensuring a balanced and optimal utilization of resources within the infrastructure. This adjustment, performed under the "ChangeInCapacity" adjustment type, facilitates the addition or removal of instances as necessary to maintain the desired CPU utilization target. Furthermore, the configuration incorporates an instance warm-up period. This temporal allowance enables newly launched instances to initialize and stabilize their operations before being factored into the scaling decisions, thereby enhancing the reliability and accuracy of the scaling mechanism.

Listing 7.22: AWS autoscaling policy for dedicate Nodes

```
resource "aws_autoscaling_policy" "example" {
  name                         = "cpu-scaling-policy"
  policy_type                  = "TargetTrackingScaling"
  adjustment_type              = "ChangeInCapacity"
  autoscaling_group_name       = aws_eks_node_group.aws_eks_node_group.
    resources[0].autoscaling_groups[0].name
  estimated_instance_warmup = var. instance_warmup
  target_tracking_configuration {
    predefined_metric_specification {
      predefined_metric_type = "ASGAverageCPUUtilization"
    }
    target_value = var.target_value
  }
}
```

7.2.2.2 Kubernetes components

Namespace

The platform sets up a dedicated namespace for each integration, giving it a name relevant to that integration.

Association IAM role to service-account

Applications running within the containers of a Pod can leverage either the AWS SDK or the AWS CLI to interact with AWS Services via API calls, using IAM permissions for authorization. These applications are required to sign their API requests to AWS with appropriate AWS credentials. This signing process involves associating an IAM role with a Kubernetes service account and configuring the Pods to utilize this service account. Establishing this association necessitates the creation of an OIDC IAM identity provider for the cluster. This OIDC IAM provider enables users to authenticate themselves with AWS using their credentials from an external identity provider, simplifying the integration of AWS with existing authentication systems and enabling access to AWS resources based on the authenticated user's identity.

Deployment

The platform creates Kubernetes Deployment intended for orchestrating Fluentd pods within a Kubernetes cluster environment. The deployment specification encapsulates various parameters crucial for the successful deployment and management of these pods. At its core, this configuration orchestrates the deployment of Fluentd pods, with the number of replicas specified by the variable "min_replicas". The Docker image utilized by the Fluentd container is sourced from an ECR repository. Resource constraints are imposed on the container to regulate its CPU and memory consumption, ensuring efficient resource utilization within the Kubernetes cluster. Port configuration is established to expose the necessary network interface for communication with the Fluentd pods. Environment variables are pivotal in configuring the Fluentd pods. Variables like "STAGING_BUCKET", "PORT_EXPOSE_SERVICE", "REGION", "POD_NAME", "CHUNCK_SIZE", and "FLUSH_INTERVAL" are meticulously set to ensure the creation of the appropriate image. Additionally, node affinity is employed to dictate the scheduling behavior of Fluentd pods. By specifying node selectors based on specific node labels, the deployment ensures that Fluentd pods are distributed across nodes in alignment with predefined criteria.

Listing 7.23: Deployment

```
resource "kubernetes_deployment" "rsyslog" {
  metadata {
    name      = "fluentd-${var.kubernetes_namespace}"
    namespace = kubernetes_namespace.rsyslog.metadata.0.name
  }
  spec {
    replicas = var.min_replicas
    selector {
      match_labels = {
        app = "Fluentd"
      }
    }
    template {
      metadata {
        labels = {
          app = "Fluentd"
        }
      }
      spec {
        service_account_name = var.service_account_name
        container {
          image = "${var.account_id}.dkr.ecr.${var.region}.amazonaws.com/${var.
    ecr}:latest"
          name  = "fluentd-container"
          resources {
            limits = {
              cpu    = var.cpu_limits
              memory = var.memory_limits
            }
            requests = {
              cpu    = var.cpu_requests
              memory = var.memory_requests
            }
          }
          port {
            container_port = var.port_pod
          }
          env{
            name = "STAGING_BUCKET"
            value =  var.bucket_fluentd_staging
          }
          env{
            name = "PORT_EXPOSE_SERVICE"
            value =  var.port_expose_service
          }
          env{
            name = "REGION"
            value =  var.region
          }
             env{
            name = "FLUSH_INTERVAL"
            value =  var.region
          }
             env{
            name = "CHUNCK_SIZE"
            value =  var.region
          }

          env {
            name  = "POD_NAME"
            value_from {
              field_ref {
                field_path = "metadata.name"
              }
            }
}
        }
        affinity {
            node_affinity {
                required_during_scheduling_ignored_during_execution {
                    node_selector_term {
                        match_expressions {
                            key      = "nodegroup"
                            operator = "In"
                            values   = ["${var.node_group_name}-label"]
                        }
                    }
                }
            }
         }

      }
    }
  }
  depends_on = [ kubernetes_namespace.rsyslog, kubernetes_config_map.
    pod_info]
```

```
}
```

Service LoadBalancer

The platform defines a Kubernetes service resource tailored for a Fluentd deployment. Within the "metadata" block, the service is assigned a name and the namespace for this service is obtained from the metadata of another Kubernetes resource. Moving to the "spec" block, the selector attribute designates the pods to which incoming traffic will be directed. The port configuration is meticulously defined within the "port" block. Here, the service is exposed on a specified port ("var.port_expose_service") and forwards incoming traffic to pods via the target port. Lastly, the service type is specified as "LoadBalancer", indicating that the Kubernetes platform should allocate an external IP address to enable access to the service from outside the cluster. This allows external clients to communicate with Fluentd seamlessly, enhancing the service's accessibility and utility.

Listing 7.24: LoadBalancer Service

```
resource "kubernetes_service" "rsyslog" {
  metadata {
    name      = "service-fluentd-${var.kubernetes_namespace}"
    namespace = kubernetes_namespace.rsyslog.metadata.0.name
  }
  spec {
    selector = {
      app = kubernetes_deployment.rsyslog.spec.0.template.0.metadata.0.
   labels.app
    }
    port {
      port        = var.port_expose_service
      target_port = var.port_expose_service
    }
    external_traffic_policy = "Cluster"
    type = "LoadBalancer"
  }
  depends_on = [kubernetes_deployment.rsyslog ]
}
```

Horizontal Pod Autoscaler

The platform creates a Horizontal Pod Autoscaler resource. This resource is configured to manage the scaling behavior of pods within a Kubernetes cluster. Within the spec section, parameters such as "max_replicas" and "min_replicas" are specified, indicating the upper and lower bounds for the number of pod replicas that can be created or terminated respectively. The "scale_target_ref" section specifies the target resource type (in this case, a Deployment) and the specific name of the deployment to which the autoscaler will be applied. Additionally, the "target_cpu_utilization_percentage" parameter is set, indicating that the autoscale will adjust the number of pod replicas based on the CPU utilization of the specified deployment. This setup allows for automatic scaling of pods to efficiently manage resource utilization within the Kubernetes cluster. The platform also installs a Metrics Server to collect Pod metrics used by Horizontal Pod Autoscaler.

Listing 7.25: Horizontal pod autoscaler

```
resource "kubernetes_horizontal_pod_autoscaler" "example" {
  metadata {
    name = "autoscaler-${var.kubernetes_namespace}"
  }
  spec {
    max_replicas = var.max_replicas
    min_replicas = var.min_replicas

    scale_target_ref {
      kind = "Deployment"
      name = kubernetes_deployment.rsyslog.metadata[0].name
    }
    target_cpu_utilization_percentage = var.
   target_cpu_utilization_percentage
  }
}
```

7.2.3 Implementation of architecture that creates the normalization pipeline part.

For each integration, an integration CodeCommit repository is created specifically for the Ingestion Pipeline normalization process. Within this repository, several key files are loaded:

- "mappingFiled.json": This file houses the JSON configuration that the administrator loads into the "mapping_json" field within the integration module declaration in the administrator repository.
- "addFieldStatic.json": Here lies the JSON data that the administrator loads into the "add_field_static" field within the integration module declaration in

the administrator repository.

- "template.yml": This YAML file serves as a template for the cloud infrastructure needed to establish an ingestion normalization pipeline. CloudFormation uses this file to orchestrate the creation of the Stack responsible for the creation of the part of mapping and uploading data in the Security Lake
- "mappingFunction.py": A Python script containing the code utilized by the lambda function for transforming logs into the OCSF format.

By organizing and labeling these files clearly, the integration process becomes more streamlined and manageable.

In addition, for each integration, it creates a Codepipeline that can read files in the Integration Repository and depending on the file configuration deploy a Serverless Application Model through AWS CloudFormation.

To begin, establish the pipeline configuration by assigning it a distinct name and specifying the IAM ARN role responsible for executing its actions. Then, establish the artifact store using an S3 bucket, ensuring the safe storage of pipeline artifacts.

Proceed by defining the "Source" stage to fetch the source code. Utilize CodeCommit as the source provider, configuring it with the relevant repository and branch names. Artifacts produced during this stage will be stored in the designated location commonly referred to as "source_output".

Finally, set the build stage to run the deployment of a SAM model through CloudFormation. CodeBuild is used for this step, specifying the input artifacts from the previous step. CodeBuild's goal is to automate the build and deployment of a serverless application developed with AWS SAM. The CodeBuild project is configured with specific environment variables, including the stack name, source bucket, destination buckets, region, and AWS account ID. The build process is set to run the commands defined in a "buildspec_integration_create.yaml". This YAML file orchestrates the build process in three steps:

- During Installation, the necessary dependencies, such as AWS SAM CLI, are installed.
- Pre_build bundles the SAM application and loads the artifacts into an S3 bucket.
- Finally, the Build phase deploys the packaged SAM application using Cloud-

Formation with specified parameters and features.

Listing 7.26: Codepipeline-integration

```
resource "aws_codepipeline" "codepipeline-integration" {
  name          = "pipeline-${var.integration_name}"
  role_arn      = aws_iam_role.codepipeline_role.arn
  artifact_store {
    location    = aws_s3_bucket.codepipeline_bucket.bucket
    type        = "S3"
  }
  stage {
    name = "Source"
    action {
      name             = "Source"
      category         = "Source"
      owner            = "AWS"
      provider         = "CodeCommit"
      version          = "1"
      output_artifacts = ["source_output"]

      configuration = {
        RepositoryName = var.name_repository_user
        BranchName       = "main"
      }
    }
  }
  stage {
    name = "build_stage"
    action {
      name             = "terraform_create"
      category         = "Build"
      owner            = "AWS"
      provider         = "CodeBuild"
      input_artifacts  = ["source_output"]
      version          = "1"
      configuration    = {
        ProjectName = aws_codebuild_project.create_project.name
      }
    }
  }
}
```

Listing 7.27: Codebuild SAM model

```
resource "aws_codebuild_project" "create_project" {
  name                  = "create_project-${var.integration_name}"
  description           = "create_codebuild_project"
  build_timeout         = "40"
  service_role          = aws_iam_role.create_project_role.arn
  artifacts {
    type = "CODEPIPELINE"
  }
  environment {
    compute_type                        = "BUILD_GENERAL1_SMALL"
    image                               = "aws/codebuild/standard:5.0"
    type                        = "LINUX_CONTAINER"
    privileged_mode             = true

    environment_variable {
      name  = "STACK_NAME"
      value = var.integration_name
    }
    environment_variable {
      name  = "BUCKET_LAMBDA"
      value = aws_s3_bucket.lambda_buckets.bucket
    }
    environment_variable {
      name  = "INTEGRATION_NAME"
      value = var.integration_name
    }
    environment_variable {
      name  = "SOURCE_BUCKET"
      value = var.bucket_fluentd_staging
    }
    environment_variable {
      name  = "DESTINATION_BUCKET"
      value = var.bucket_destination_ocsf
    }
    environment_variable {
      name  = "REGION"
      value = var.region
    }
    environment_variable {
      name  = "ACCOUNT_ID"
      value = var.account_id
    }
  }
  source {
    type            = "CODEPIPELINE"
    buildspec       = "buildspec-user-create.yaml"
  }
}
```

```
version: 0.2
phases:
  install:
    commands:
      - ls
      - unzip aws-sam-cli-linux-x86_64.zip -d sam-installation
      - sudo ./sam-installation/install
      - sam --version
  pre_build:
    commands:
      - cd SAM
      - sam package --template-file template.yml --output-template-file
    package.yml --s3-bucket "$BUCKET_LAMBDA"
  build:
    commands:
      - sam deploy --template-file package.yml --stack-name "$STACK_NAME"
    --parameter-overrides "StackName=$STACK_NAME IntegrationName=
    $INTEGRATION_NAME SourceBucket=$SOURCE_BUCKET DestinationBucket=
    $DESTINATION_BUCKET Region=$REGION AccountId=$ACCOUNT_ID" --
    capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM
```

Listing 7.28: buildspec-user-create.yaml

7.3 STEP 3 of Platform's construction

This final stage delineates the implementation of the Normalization Ingestion Pipeline. The initial segment describes the contents of the Integration Repository files, while the subsequent part elucidates the internal workflow of the ingestion pipeline.

7.3.1 Integration Repository files

The Integration Repository comprises four key files:

- "mappingField.json": This file contains the JSON configuration that the administrator loads in the "mapping_json" field within the integration module statement in the Administrator Repository.
- "addFieldStatic.json": Herein lies JSON data used by administrators for populating the "add_field_static" field within the integration module declaration in the Administrator Repository.

- "template.yml": A YAML file functioning as a blueprint for the cloud infrastructure required to establish Ingestion Pipeline normalization. CloudFormation leverages this file to orchestrate the creation of the Stack responsible for the mapping segment.
- "mappingFunction.py": A Python script comprising the code executed by the lambda function for transforming logs into the OCSF format.

These JSON files facilitate adjustments to message mappings in the OCSF format, providing a streamlined approach for automation. Modifications made within these files can significantly expedite automation processes.

7.3.1.1 template.yaml

The configuration in the "template.yaml" file delineates the structure of an AWS CloudFormation template. This template orchestrates the creation and configuration of various AWS resources.

The configuration starts by specifying the version of the AWS CloudFormation template format being used, which is '2010-09-09'. It then employs the 'AWS::Serverless-2016-10-31' transform to utilize AWS Serverless Application Model (SAM) features. The description provides an overview of the purpose of the template, which is to Create Ingestion Pipeline.

Parameters section defines input parameters required for the template. These parameters include StackName, IntegrationName, SourceBucket, DestinationBucket, Region, and AccountId, allowing customization and flexibility during deployment.

Resources section defines the AWS resources to be created and configured by the template. Notably:

1. MappingFunction:
   - Type: This resource is of type AWS::Serverless::Function, indicating it is a serverless function managed by AWS Lambda.
   - Properties:
     - Runtime: Specifies the runtime environment for the Lambda function, which in this case is Python 3.10.
     - Handler: Indicates the entry point for the Lambda function code (map-

pingFunction.handler).

  - FunctionName: Dynamically generates a name for the Lambda function using the IntegrationName parameter. This allows for unique naming based on the integration.
  - Architectures: Defines the architecture for the Lambda function, ensuring compatibility with the specified architecture.
  - Layers: Specifies additional layers to be attached to the Lambda function. In this case, it includes an AWS SDK layer for Pandas, possibly indicating the function's dependence on Pandas for data processing.
  - Timeout: Sets the maximum execution time for the Lambda function.
  - MemorySize: Defines the amount of memory allocated to the Lambda function during execution.
  - Environment: Provides environment variables to the Lambda function, including parameters and integration-specific values required for its operation.
  - Policies: Grants necessary permissions to the Lambda function through IAM policies. Here, permissions for accessing specified S3 buckets are configured.

2. IAMRole3: An IAM role used by the Lambda functions, CreateTrigger. It includes policies for S3 access, CloudWatch Logs, and Lambda permissions.
3. CustomResource: This resource type defines a custom resource that extends CloudFormation's capabilities. With "ServiceToken: !GetAtt CreateTrigger.Arn", it specifies the ARN of the Lambda function that handles the custom resource's logic. This Lambda function is invoked when the custom resource is created.
4. CreateTrigger Lambda Function:

   - It specifies that the Lambda function depends on another resource named "MappingFunction".
   - The function's properties include:

     - Handler: Specifies the entry point for the Lambda function code.
     - Runtime: Specifies the runtime environment for the function, in this case, Python 3.8.
     - Role: Specifies the IAM role used by the Lambda function. This role is referenced from another resource named "IAMRole3".

- Timeout: Specifies the maximum execution time for the Lambda function.
- Environment: Defines environment variables accessible to the Lambda function.
- Code: This section contains the actual code that will be executed by the Lambda function. The code is provided as a ZIP archive under the "ZipFile" property. The function code is written in Python and includes imports for necessary libraries such as boto3, cfnresponse, and json. The code retrieves environment variables using the "os.environ" dictionary and defines a handler function named "handler" that processes events passed to the function by AWS Lambda. It associates an S3 bucket event trigger with the Lambda function, MappingFunction, by adding permission and configuring S3 bucket notifications.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: Create Ingestion Pipeline

Parameters:
  StackName:
    Type: String
  IntegrationName:
    Type: String
  SourceBucket:
    Type: String
  DestinationBucket:
    Type: String
  Region:
    Type: String
  AccountId:
    Type: String

Resources:
  MappingFunction:
    Type: AWS::Serverless::Function
    Properties:
      Runtime: python3.10
      Handler: mappingFunction.handler
      FunctionName: !Sub 'MappingFunction-${IntegrationName}'
      Architectures:
        - "x86_64"
      Layers:
       - "arn:aws:lambda:eu-west-1:336392948345:layer:AWSSDKPandas-
    Python310:3"
      Timeout: 900
      MemorySize: 2048
      Environment:
        Variables:
          IntegrationName: !Sub '${IntegrationName}'
          DestinationBucket: !Sub '${DestinationBucket}'
          Region: !Sub '${Region}'
          AccountId: !Sub '${AccountId}'
      Policies:
        - Version: "2012-10-17"
          Statement:
            - Effect: "Allow"
              Action:
                - "s3:*"
              Resource:
                - !Sub "arn:aws:s3:::${SourceBucket}"
                - !Sub "arn:aws:s3:::${SourceBucket}/*"
                - !Sub "arn:aws:s3:::${DestinationBucket}"
                - !Sub "arn:aws:s3:::${DestinationBucket}/*"

  CreateTrigger:
    Type: 'AWS::Lambda::Function'
    DependsOn: MappingFunction
    Properties:
      Handler: index.handler
      Runtime: python3.8
      Role: !GetAtt IAMRole3.Arn
      Timeout: 10
      Environment:
        Variables:
          IntegrationName: !Sub '${IntegrationName}'
          SourceBucket: !Sub '${SourceBucket}'
          Region: !Sub '${Region}'
          AccountId: !Sub '${AccountId}'
      Code:
        ZipFile: |
          import boto3
          import cfnresponse
          import json
      import os
      IntegrationName = os.environ['IntegrationName']
      SourceBucket = os.environ['SourceBucket']
      Region = os.environ['Region']
      AccountId = os.environ['AccountId']
      def handler(event, context):
       try:
        if event['RequestType'] in ['Create']:
         lambda_client = boto3.client('lambda')
         lambda_client.add_permission(
          FunctionName=f'MappingFunction-{IntegrationName}',
          StatementId='ID-1',
          Action='lambda:InvokeFunction',
          Principal='s3.amazonaws.com',
          SourceArn=f'arn:aws:s3:::{SourceBucket}'
         )
         s3 = boto3.client('s3')
         s3.put_bucket_notification_configuration(
          Bucket=f'{SourceBucket}',
          NotificationConfiguration= {'LambdaFunctionConfigurations':[{'LambdaFunctionArn': f'arn:aws:lambda:{Region}:{AccountId}:function:MappingFunction-{IntegrationName}', 'Events': ['s3:ObjectCreated:*']}]}
         )
        responseData = {"Success": "Good"};
        cfnresponse.send(event, context, cfnresponse.SUCCESS, responseData)
       except Exception as e:
        print(f"Error: {e}")
        cfnresponse.send(event, context, cfnresponse.FAILED, {"Error": str(e)})

  IAMRole3:
   Type: "AWS::IAM::Role"
   Properties:
    Path: "/"
    RoleName: !Sub 'custom-resource-iam-role-${IntegrationName}'
    AssumeRolePolicyDocument: "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Sid\":\"\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"lambda.amazonaws.com\"},\"Action\":\"sts:AssumeRole\"}]}"
    MaxSessionDuration: 3600
    Policies:
        - PolicyName: "Policy-s3"
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: "Allow"
                Action:
                  - "s3:*"
                Resource: 'arn:aws:s3:::*'
        - PolicyName: "Policy-logs"
          PolicyDocument:
            Statement:
              - Effect: "Allow"
                Action:
                  - "logs:CreateLogGroup"
                Resource: !Sub 'arn:aws:logs:${Region}:${AccountId}:*'
              - Effect: "Allow"
                Action:
                  - "logs:CreateLogStream"
                  - "logs:PutLogEvents"
                Resource: !Sub 'arn:aws:logs:${Region}:${AccountId}:log-
    group:/aws/lambda/*'
        - PolicyName: "Policy-lambda"
          PolicyDocument:
            Statement:
              - Effect: "Allow"
                Action:
                  - "lambda:AddPermission"
                Resource: !Sub 'arn:aws:lambda:${Region}:${AccountId}:
    function:*'

  CustomResource:
    Type: 'AWS::CloudFormation::CustomResource'
    Properties:
      ServiceToken: !GetAtt CreateTrigger.Arn
```

Listing 7.29: Codebuild SAM model

7.3.1.2 mappingFunction.py

The "mappingFunction.py" script differs between the "SyslogIntegrated" and "S3Integrated" integrations due to the distinct formats of the incoming messages. For "SyslogIntegrated", messages are loaded into the staging buckets of Fluentd pods in the format <data><tag.facility.severity><data(JSON)>, like this example: "2023-11-29T13:51:00+00:00 system.authpriv.notice {"host":"VRECDFE6B3505", "ident": "sudo", "message":"..."}". In contrast, for "S3Integrated", there is only a single JSON format. While the overall structure of mappingFunction.py remains mostly similar, there is an additional part specifically tailored for the "SyslogIntegrated" integration. This extra part analyzes the first two fields <data> and <tag.facility.severity>. In the following part, we describe the Syslog integration script, in order to describe also the additional part.

The Python script first initializes the AWS S3 client using "boto3.client('s3')" and then retrieves environment variables such as IntegrationName, DestinationBucket, Region, and AccountId. The code defines a Python feature called "handler" that handles events triggered by changes in an Amazon S3 bucket within a Lambda AWS feature. Extracts S3 event information such as the bucket name and object key from the event parameter. It also uses the "urllib.parse.unquote_plus()" function to decode the encoded key.

The function initializes an empty "vector_OCSF" vector, intended to store objects in OCSF format before they are loaded into the Security Lake. It then proceeds to make a request to the S3 service, providing the previously obtained bucket and key, to retrieve the object for conversion.

Utilizing "gzip.GzipFile" as a context manager, the code efficiently unzips the content of the file received from the S3 bucket. Subsequently, a for loop iterates through each line of this unzipped file, representing one log entry per line.

The code below divides a line into a list using the tab character as the delimiter. The resulting list is stored in the variable s. A new empty dictionary called "new_ocsf" is then created to store the processed information for this line. The first item in the list, namely the date and time in the ISO format, is extracted. This code calculates the data path for the key in the buckets of Security Lake. It does so by converting the date and time string to a datetime object and then formats the extracted date to "YYYYMMDD" format. The Unix timestamp date is then converted to milliseconds. The processed time is stored in the "new_ocsf" dictionary. The second field, <tag.facility.severity>, is analyzed to determine the severity level. The code examines the third part of this field and, depending on the value found, assigns the right value to the OCSF attribute "severity_id" and its associated attribute "severity". These fields are then stored in the dictionary.

The code then loads the JSON file data "addFieldStatic.json" and adds it to the "new_ocsf" dictionary.

The code takes the element at position 2 from the list, saves it in a variable, "input_string". Then loads the data from the JSON file "mappingField.json" and calls the function "process_json()" passing as arguments the loaded JSON, the dictionary "new_ocsf" and the JSON string "input_string".

The "process_json()" function, driven by the mapping parameters specified in the "mappingField.json" file, orchestrates the analysis and mapping of the log "input_string" into corresponding fields within the OCSF. These mappings are stored within a dictionary.

Once a line's mapping is completed, the dictionary is appended to the "vector_OCSF". Upon reaching a threshold of 75,000 elements in "vector_OCSF", or after analyzing all rows in the file, a Pandas DataFrame is constructed using the data from "vector_OCSF". Subsequently, this data frame is converted into Parquet format and stored within a byte buffer. The Parquet file's name is dynamically generated utilizing key components. A path within the S3 bucket is then formed, incorporating details such as integration name, region, account ID, and event date. Finally, the Parquet file is uploaded to the designated S3 bucket using the previously created path.

Fuction "process_json(mappingField, new_ocsf, input_string)"

The function "process_json(...)" uses another function called "extract_value(json_string, key)". The "extract_value(json_string, key)" function takes a JSON string and a key as input. The JSON string is loaded into a Python object using JSON.loads() method. The "json_data.get(key)" method is used to retrieve the value associated with the provided key. If the key is not found in the JSON string, the function returns None.

The "process_json(mappingField, new_ocsf, input_string)" function is the main function for processing. It takes three arguments: "mappingField", which is a dictionary containing the mappings, "new_ocsf", which is a dictionary used to store the results, and "input_string", which is the input JSON string.

Two variables, "key_with_point" and "json_key", are initialized in the function.

To perform the task, begin by iterating through the keys and values of the "mappingField" dictionary.

- If the key is in the format "key1_OSCF.Key2_OSCF.Key3_OSCF...", save "key1_OSCF" in the "key_with_point" variable, and the second piece of "Key2 _OSCF.Key3_OSCF..." with the value associated in the "json_key" dictionary. If the iteration after the key starts with "key1_OSCF", the json_key dictionary continues to be filled. However, if it is different, create a dictionary variable "json_new = {}" to save new fields within the object. Then, recursively call the "process_json(json_key, json_new, input_string)", passing as the first parameter the dictionary "json_key" and the dictionary "json_new" to fill.
- If the key and value are in the format "key1_OSCF" and "key_oldFormat", call the function "extract_value(. . .)" with "input_string" and the value "key_oldFormat" as arguments. This function returns the value to be saved. Save this value in the "new_ocsf" dictionary using the key "key1_OSCF".
- If the key and value are in the format "key1_OSCF" and {"key_oldFormat": {"value_1_oldFormat": "value_1_OCSF", ... }}, the code will call the function "extract_value(. . .)" by passing input_string and the value "key_oldFormat" as arguments. This function returns the value to the old format. To convert it to the OCSF value, the "extract_value(. . .)" function is invoked again by passing the JSON {"value_1_oldFormat": "value_1_OCSF", ... } along with the value in the old "value_n_oldFormat" format as arguments. This function returns the value to be saved. The value is then saved in the "new_ocsf" dictionary using the key "key1_OSCF".
- If the key and value are in the format "key1_OSCF" and {"key_oldFormat": "regular expression"}, the code will call the function "extract_value(. . .)" by passing "input_string" and the value "key_oldFormat" as arguments. This function returns the full value. Inside this value, the code extracts the relative value indicated by the regular expression. If this value is present, it is saved in the "new_ocsf" dictionary using the key "key1_OSCF
- If the key and value are in the format "key1_OSCF" and {"key_oldFormat":{"regular expression":{value_1_oldFormat":"value_1_OCSF", ... }}}, the code will call the function "extract_value(. . .)" by passing "input_string" and the value "key_oldFormat" as arguments. This function returns the full value. Inside this value, the code extracts the relative value indicated by the regular expression. If this value is present, the "extract_value(. . .)" function is invoked again by passing the JSON {"value_1_oldFormat": "value_1_OCSF", ... } along with the relative value as arguments. This function returns the value to be saved. The value is then saved in the "new_ocsf" dictionary using the key "key1_OSCF".

Once the iteration is complete, the function will return and the "new_ocsf" dictionary will be complete.

7.3.2 Ingestion Pipeline

The platform integration architecture uses S3 buckets to store messages awaiting conversion to OCSF format. Messages from different sources are deposited into these buckets either directly by S3-integrated vendors or applications with custom integrations or via Fluentd Pods for Syslog-integrated sources. Lambda functions, scripted in Python, are pivotal in handling integrations. Triggered by the uploading of objects into specific S3 buckets, these functions meticulously analyze messages, transforming them into the standardized OCSF format. Once a predefined threshold of converted logs is reached, the Lambda function orchestrates the creation of a DataFrame, which is then converted into the Parquet format. Finally, the Parquet-formatted data is uploaded into the Security Lake with a structured key format for efficient organization, including metadata such as integration name, region, account ID, and event date.

Chapter 8

Platform's Validation

In this chapter, we analyze the maximum traffic ingestion capacity measured in messages per second of the platform, evaluating possible bottlenecks and studying possible optimizations. This platform is created within an AWS account, subject to constraints such as the maximum use of EC-2's vCPUs, the limit of parallel execution of Lambda functions, and the maximum frequency of requests to S3 buckets. Let's examine in particular the integration of Linux events in Syslog format from external sources in Security Lake.

8.1 Environment where the platform is created

The platform is created within an AWS account, that account has several limits that affect the functionality of our platform. First, there is a cap on total CPU allocation for EC2 instances, fixed at 64 vCPUs. This means that the sum of vCPUs on all nodes cannot exceed 64.

Secondly, there is a constraint on the performance of Lambda functions. We are allowed a maximum of 50 simultaneous runs of Lambda. When this limit is reached, subsequent Lambda invocations are queued, leading to limitation. Lambda will attempt to run these events in the queue twice, with delays between attempts. However, if the event fails all attempts or stays in the queue too long, Lambda discards it. The throttle phenomenon indicates the maximum ingestion capacity of the platform.

In addition, for buckets S3, we are limited to 3500 PUT requests per second for the same prefix. In addition, files uploaded in the Parquet format Security Lake must be less than 50 MB. To ensure that Lambda remains within the limit of 3500 PUT requests per second, we aim to aggregate multiple messages in the same Parquet file without exceeding the 50MB size limit.

8.2 Linux events' integration (Syslog)

The integration of the Linux events in Syslog format in Security Lake was done using the platform with the catching part having nodes with instance type “t3.2xlarge”, (8vcpu and 32 of GiB of memory). The nodes are 3 and then scale up to a maximum of 7 when the average total CPU consumption exceeds 50%.

The module created to configure the integration in the "main.tf" file of the Admin Repository is represented in the following code section.

Listing 8.1: Module of integration Linux Syslog into Admin Repository

```
module "integrationLinuxSyslog" {
    type = "SyslogIntegrate"
    integration_name = "integrationLinuxSyslog"
    source = "./modules/SyslogIntegrated"
    bucket_destination_ocsf = "aws-security-data-lake-eu-west-1-kohgjkhjbvg"
    mapping_json = {
        "category_name2":"System␣Activity",
        "category_id":1,
        "class_name":"Process␣Activity",
        "class_uid":1007,
        "metadata":{"version":"v1.0.0","profiles":["host"], "product":{"name":"VM"}}}
    }
    add_field_static = {
        "actor.process.cmd_line":"ident",
        "actor.user.name":"host",
        "device.host":"host",
        "process.cmd_line":{"message" : "COMMAND=(.+)"},
        "process.lineage":{"message" : "COMMAND=([^␣]+)"},
        "process.xattributes.pwd":{"message" : "PWD=([^␣]+)"},
        "process.user.name":{"message" : "USER=([^␣]+)"},
        "process.user.type_id":{"message":{"USER=([^␣]+)":{"user":1, "root":2, "system":3, "None":0}}},
        "process.user.type":{"message":{"USER=([^␣]+)":{"user":"User", "root":"Admin", "system":"System", "None":"Unknown"}}},
        "activity_id":{"message":{"COMMAND=":{"COMMAND=":1, "None":0}}},
        "activity":{"message":{"COMMAND=":{"COMMAND=":"Launch", "None":"Unknown"}}},
        "raw_data":"message"
    }
    port_expose_service = 5140
    min_replicas = 9
    max_replicas = 200
    pod_limits_cpu = "1000m"
    pod_limits_memory = "1024Mi"
    pod_requests_cpu = "1000m"
    pod_requests_cpu = "1024Mi"
    average_CPU_usage_pods = 50
    IP_source = "123.23.1.21"
    buffer_size = 256k
    flush_interval = 1m
}
```

8.3 Test 0

In the initial test, we utilized the configuration outlined in the preceding chapter. The scalability of the system was examined, in particular in relation to the incoming flow of messages per second. Initially, in the catching part, the system uses 9 pods, a number that remained consistent until the message flow reached 28,000 messages per second. As the message flow increased beyond this threshold, the number of pods dynamically scaled up, reaching its maximum allocation when the incoming flow surpassed 260,000 messages per second.

Similarly, the number of nodes in the system remained constant until the message flow reached 100,000 messages per second. Beyond this point, the system dynamically scaled up the number of nodes, again capping at the maximum when the incoming flow exceeded 260,000 messages per second.

Graphs 8.1 and 8.2 depict the relationship between incoming message flow and the corresponding resource allocation in two different aspects of the system. In both graphs, the left vertical axis represents the rate of incoming messages per second, while the horizontal axis denotes the passage of time. On the right vertical axis, the resource allocation is illustrated, with Graph 8.1 showing the number of pods and Graph 8.2 displaying the number of nodes. This visualization helps us understand how the system dynamically scales its resources in response to varying message loads over time.

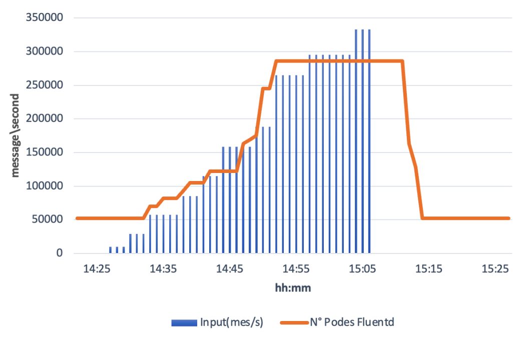


Figure 8.1: Results Test-0: pods.

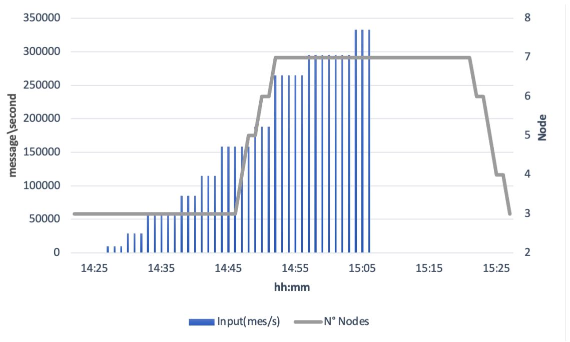


Figure 8.2: Results Test-0: nodes

In the Normalization ingestion pipeline, we observe a direct correlation between incoming traffic and the number of lambda invocations and concurrent executions. Once the incoming traffic surpasses 260,000 messages per second, Lambda becomes saturated, resulting in queued invocation requests and increased throttling instances.

Three graphs, numbered 8.3, 8.4, and 8.5, intricately illustrate the behavior of our normalization pipeline, focusing notably on the constraint of parallel executions. Within each graph, the blue line signifies the influx of data into the platform, correlating with the left ordinate axis. In the initial graph (8.3), the green line portrays the frequency of Lambda function invocations. Moving to the subsequent depiction (8.4), the yellow line showcases the concurrent execution of Lambda functions, highlighting the operational capacity of our infrastructure. Finally, in the third graph (8.5), the red line delineates instances of throttling, identifying operational bottlenecks that may impede performance. These visualizations offer a comprehensive understanding of our normalization pipeline's dynamics, shedding light on both its strengths and limitations in managing parallel executions.

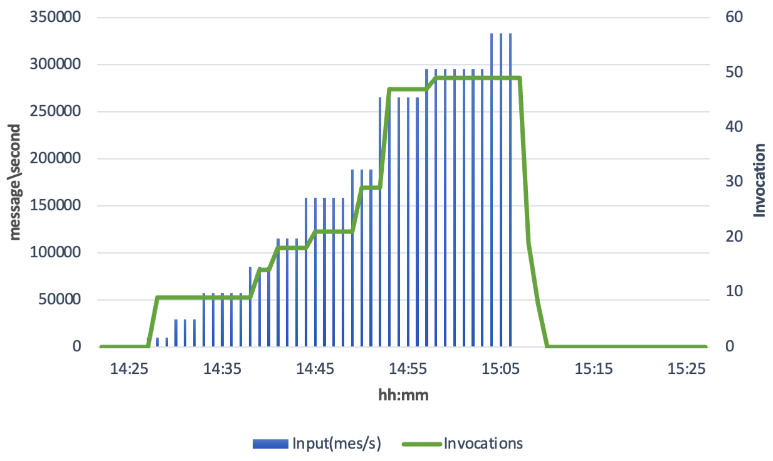


Figure 8.3: Results Test-0: invocations

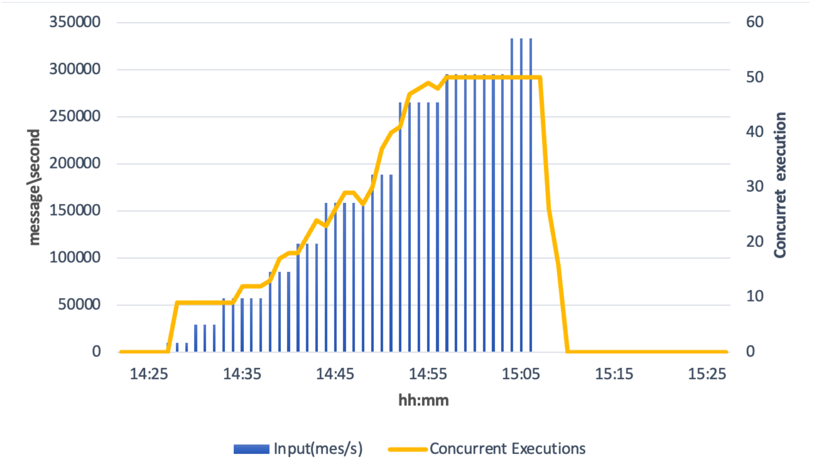


Figure 8.4: Results Test-0: concurrent execution

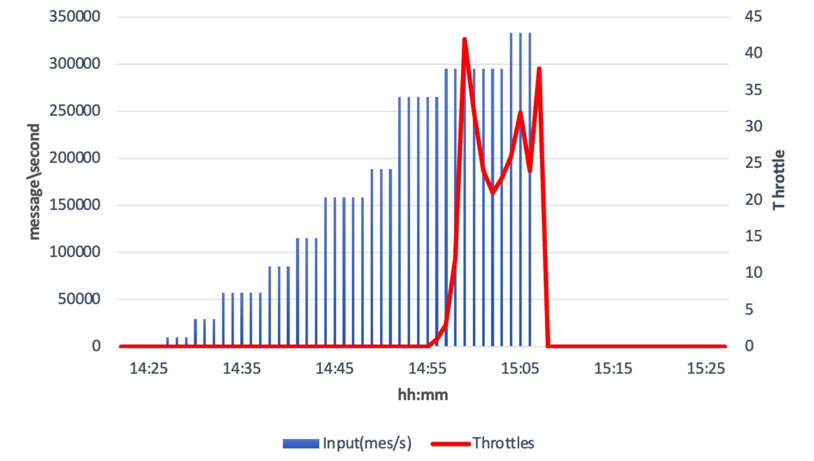


Figure 8.5: Results Test-0: throttles

With this test configuration, the platform demonstrates its capability to process 260,000 messages per second. This achievement stands out significantly, surpassing the EPS (events per second) of a large SIEM by threefold and exceeding the C-SOC (Central Directorate of Criminal Police, Italy [24]) by an impressive 34 times.

8.4 Optimization of lambda

To optimize ingestion capacity, it is crucial to control the invocation frequency of the Lambda function. This frequency directly correlates with the number of data loads initiated by Pods accessing staging buckets. One approach to reduce loads is by extending the buffering time within the Pods. This adjustment prompts the Pods to handle larger files but with reduced frequency, effectively decreasing the overall load on the system.

Another change to decrease loads is to reduce the number of pods. Fewer pods with a larger cpu that absorb more data.

Our research focuses on analyzing two key factors: the buffering time (flush_t interval) within Fluentd Pods and the vCPU size assigned to each Pod. By adjusting these parameters, we aim to gauge their impact on the platform's ability to handle traffic ingestion effectively. This investigation will provide valuable insights for refining optimization strategies to enhance platform performance.

8.4.1 Test: flush_interval's variations

In these tests, we maintain a constant value of "1000m" for the variables "pod_limits_cpu" and "pod_requests_cpu". Our focus is then directed towards assessing the platform's capacity to ingest incoming traffic by varying the buffer's time, known as "flush_interval", within the pods. The tests we conducted varied the "flush_interval" parameter across different time intervals: 1s, 5s, 10s, 30s, 60s, 90s, 120s, 180s, and 240s.

The graph presented in Figure 3.4 illustrates the relationship between ingestion capacity (messages per second) and the flow rate (measured in seconds). Initially, the curve exhibits a sharp incline from 1s to 10s, indicating rapid growth in ingestion capacity. Subsequently, from 10s to 120s, the curve demonstrates a more gradual ascent, suggesting a milder increase in ingestion capacity over time. Beyond 120s, the growth rate of the curve diminishes progressively, eventually converging towards an ingestion capacity of approximately 310,000 messages per second.

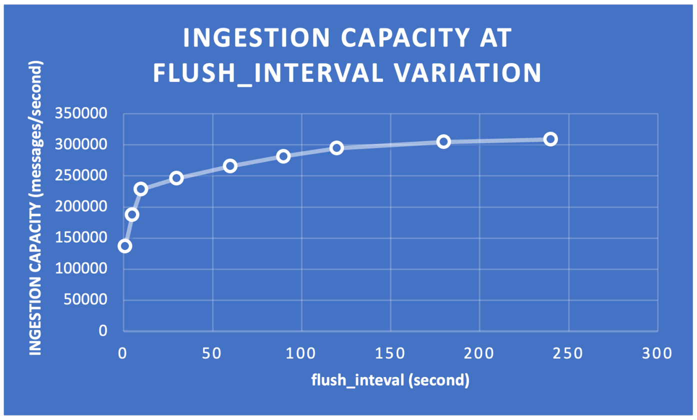


Figure 8.6: Graph of ingestion capacity at flush_interval variation

8.4.2 Test: Pod CPU variations

The tests maintained a constant 60-second flush buffer while adjusting the "pod_requests_cpu" and "pod_limits_cpu" fields. Pod CPU resources were incrementally increased from 0.25 up to 1.5, with a deliberate decision not to exceed 1.5 CPUs to maintain adequate redundancy. This adjustment resulted in fewer pods within the EKS cluster but with enhanced computational power.

The aim of this CPU increment was to assess the platform's ingestion capacity concerning pod vCPU variations. Graph 8.4 illustrates a notable growth trend, particularly pronounced from 0.25 to 0.5 CPU, followed by continued growth from 0.5 to 1 CPU. However, the growth trajectory starts to diminish beyond 1 CPU, particularly evident from 1 to 1.5 CPUs.

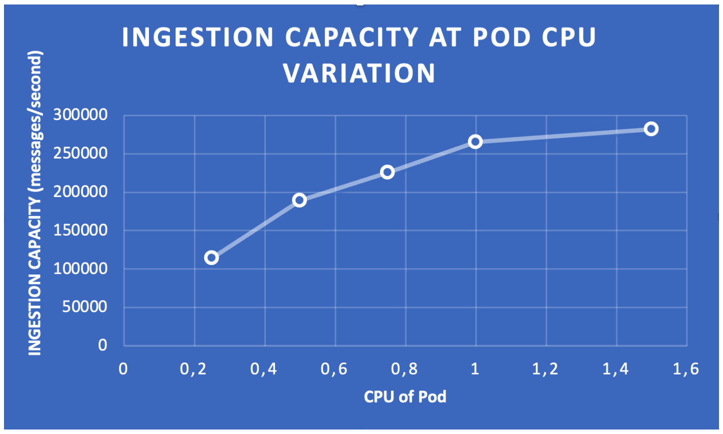


Figure 8.7: Graph of ingestion capacity at pod CPU variation

8.4.3 Test: unifying the results of previous studies

We combined the results of previous studies to create an optimized platform. We chose a flush interval of 120 seconds because increasing the value does not improve the gain significantly. We also used 1.5 CPU pods, as it provided the greatest gain without decreasing redundancy too much.

Figures 8.8, 8.9, and 8.10 describe the behavior of the ingestion pipeline within the platform concerning the influx of incoming messages. The y-axis represents the input flow in messages per second, while the x-axis denotes time. In all graphs, the blue line signifies the flow entering the platform. In the first figure, the green line represents the invocation count of the lambda function, while in the second figure, the yellow line indicates the number of concurrent executions. In the third graph, the red line depicts the occurrences of throttling.

The platform demonstrated its capability to efficiently ingest traffic up to 375,000 messages per second. However, as illustrated in the figures, when the message flow surpasses 400,000 messages per second, the parallel executions of the lambda function reach the maximum of 50, resulting in the first instances of throttling.

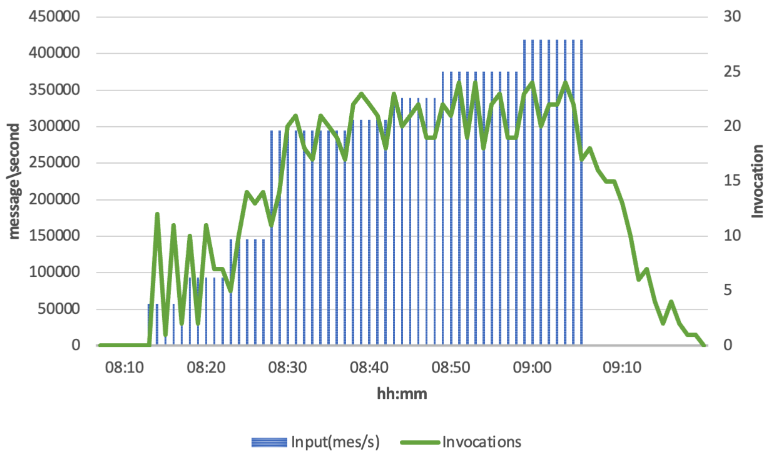


Figure 8.8: Results of the optimized test: invocations.

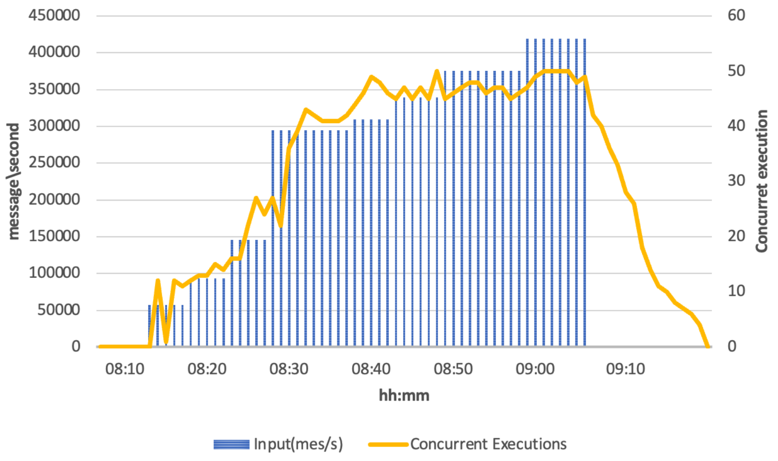


Figure 8.9: Results of the optimized test: concurrent executions.

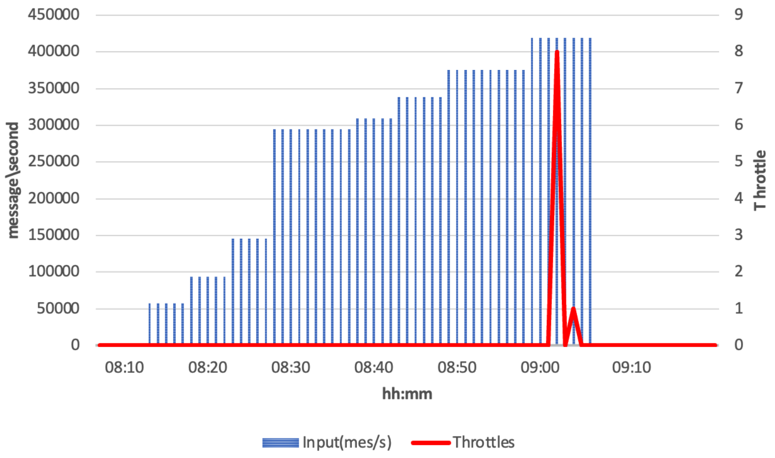


Figure 8.10: Results of the optimized test: throttles.

Chapter 9

Conclusions

The developed platform enhances Security Lake functionality by seamlessly integrating external third-party sources. Specifically, it can incorporate three types of sources: S3 integrated Vendors, Applications with custom S3 integration, and Syslog integrated Vendors.

This platform streamlines and expedites integration through a semi-automated process: integrators utilize pre-configured Terraform modules for each integration, detailing the integration method and the mapping from the original format to OSCF. Leveraging this information, the platform autonomously constructs an infrastructure capable of collecting external source logs, transforming and mapping them to the OSCF format, and efficiently uploading them into the Security Lake. Developed using the Terraform language, the platform is easily extensible, requiring minimal additional code to accommodate new integration types.

Notably, the platform is capable of scaling in relation to the incoming message flow that it has to enter into the Security Lake. Following optimization, it can ingest up to 375,000 messages per second, a performance level five times higher than that of large SIEMs and fifty times greater than the C-SOC of the Central Directorate of Criminal Police (ITALY).

OCSF is an expanding and increasingly used Schema Framework. In the future it could also be extended in a wider context and not only that related to Cybersecurity.

Bibliographic references

1. https://www.splunk.com/en_us/blog/security/overcome-cybersecurity-challenges-to-improve-digital-resilience.html.

2. https://schema.ocsf.io/1.0.0/base_event?extensions=.

3. https://github.com/ocsf/ocsf-docs/blob/main/Understanding%20OCSF.md.

4. https://schema.ocsf.io/1.1.0/classes/scheduled_job_activity?extensions=.

5. https://github.com/ocsf/ocsf-schema/blob/main/extensions.md.

6. https://geekflare.com/devops-tools-ansible-and-terraform//.

7. https://www.qovery.com/blog/what-is-kubernetes/.

8. https://kubernetes.io/docs/concepts/overview/components/.

9. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

10. https://kubernetes.io/docs/concepts/services-networking/service/.

11. https://aws.amazon.com/getting-started/aws-networking-essentials/?nc1=h_ls.

12. https://docs.aws.amazon.com/it_it/vpc/latest/userguide/what-is-amazon-vpc.html.

13. https://docs.aws.amazon.com/it_it/vpc/latest/privatelink/privatelink-access-aws-services.html.

14. https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html.

15. https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome-introducing.html.

16. https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html.

17. https://aws.amazon.com/eks//.

18. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-howdoesitwork.html.

19. https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html.

20. https://docs.aws.amazon.com/it_it/lambda/latest/dg/invocation-sync.html.

21. https://docs.aws.amazon.com/it_it/lambda/latest/dg/invocation-async.html.

22. https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/subnet.

23. https://docs.fluentd.org/input/syslog.

24. https://www.cybersecitalia.it/polizia-di-stato-cosi-funziona-il-cyber-security-operations-center-c-soc-di-roma-a-tutela-delle-banche-dati-delle-forze-di-polizia/13082/.