

Cloud Native Technical Landscape

APPENDIX D

Document History

Version	Date	Names	Comments
1.0	10/02/2023	3.3	First draft
1.1	12/07/2023	3.3	Minor adjustments
1.2	06/01/2025	3.3	List of Project related tools added

List of Annexes

#	Annex
1	<i>Azure Naming Conventions for EMSA projects.docx</i>

Contents

1	Cloud-Native Approach.....	4
1.1	Design Guidelines	4
1.2	Cloud Architecture	7
2	Cloud architecture components	10
2.1	Application-level architecture	10
2.1.1	Building Blocks	10
2.2	Infrastructure Architecture.....	20
2.2.1	Cloud / On-prem access management	20
2.2.2	Hybrid Cloud Networking	21
2.2.3	Integrated Monitoring	24
2.2.4	Business Continuity / Disaster Recovery	25
2.2.5	Multicloud Kubernetes	26
2.2.6	Infrastructure as Code	28
2.2.7	CI/CD and automation	29
2.2.8	Governance.....	32
2.2.9	Data analytics.....	33
2.3	Data catalogue	34
2.4	Cloud architecture technology landscape matrix.....	35
2.5	Project related tools	36
1	Cloud-Native Approach.....	3
1.1	Design Guidelines	3
1.2	Cloud Architecture	6
2	Cloud architecture components	9
2.1	Application level architecture	9
2.1.1	Building Blocks	9
2.2	Infrastructure Architecture.....	19
2.2.1	Cloud / On-prem access management	19
2.2.2	Hybrid Cloud Networking	20
2.2.3	Integrated Monitoring	23
2.2.4	Business Continuity / Disaster Recovery	24
2.2.5	Multicloud Kubernetes	25
2.2.6	Infrastructure as Code	27
2.2.7	CI/CD and automation	28
2.2.8	Governance.....	31
2.2.9	Data analytics.....	32

2.3	Data catalogue	33
2.4	Cloud architecture technology landscape matrix.....	34
2.5	Project related tools	35

1 Cloud-Native Approach

1.1 Design Guidelines

Applications running in cloud environments are not necessarily Cloud Native *per se*. Scenarios like lift and shift migrations may use cloud platforms, yet not take advantage of cloud ecosystems' benefits. In contrast, applications running on-premises may be classified as Cloud Native if they meet certain properties or requirements. So, it is not the location that defines "Cloud Nativeness", but the readiness of the application to adopt more evolved environments.

Modern PaaS and Serverless platforms enable applications to take advantage of cloud managed services. In contrast to this, more traditional and not so flexible platforms may limit applications design to a non-cloud readiness state. Again, although more frequently found on Cloud Providers, Cloud Platforms can be also developed and/or operated on- premises.

There may be multiple interpretations on what properties define a system as Cloud Native, however all of these interpretations agree that the core aspects of "Cloud Nativeness" are speed and agility. The following sections summarize the key properties regarding the project's team vision on the topic.

A widely accepted method for constructing cloud-based applications is the Twelve-Factor application. It describes a set of principles and practices that developers follow to construct applications optimized for modern cloud environments. Systems built upon these principles can deploy and scale rapidly and add features to react quickly to market changes.

Factor	Explanation
1 - Code Base	A single code base for each microservice, stored in its own repository. Tracked with version control, it can deploy to multiple environments (QA, Staging, Production).
2 – Dependencies	Explicitly declare and isolate dependencies. Regardless of what platform your application is running on, use the dependency manager included with your language or framework. How you install operating system or platform dependencies depends on the platform: <ul style="list-style-type: none">• In noncontainerized environments, use a configuration management tool (Chef, Puppet, Ansible) to install system dependencies.• In a containerized environment, do this in the Dockerfile
3 - Configurations	Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code. The same deployment can propagate across environments with the correct configuration applied.

4 - Backing Services	Ancillary resources (data stores, caches, message brokers) should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable.
5 - Build, Release, Run	Each release must enforce a strict separation across the build, release, and run stages. Each should be tagged with a unique ID and support the ability to roll back. Modern CI/CD systems help fulfill this principle.
6 – Processes	Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store.
7 - Port Binding	Each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices.
8 – Concurrency	When capacity needs to increase, scale out services horizontally across multiple identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available. Develop the application to be concurrent making scaling out in cloud environments seamless.
9 – Disposability	Service instances should be disposable. Favour fast startup to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement.
10 - Dev/Prod Parity	Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.
11 – Logging	Treat logs generated by microservices as event streams. Process them with an event aggregator. Propagate log data to data-mining/log management tools like Azure Monitor or Splunk and eventually to long-term archival.
12 - Admin Processes	Run administrative/management tasks, such as data cleanup or computing analytics, as one-off processes. Use independent tools to invoke these tasks from the production environment, but separately from the application.

Additionally, at EMSA application should also follow these 3 additional principles/Factors

Factor	Explanation
13 - API First	Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service.

14 – Telemetry	On a workstation, you have deep visibility into your application and its behaviour. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data.
15 - Authentication/ Authorization	Implement identity from the start. Consider RBAC (role-based access control) features available in public clouds.

EMSA projects shall follow as close as possible the overall Cloud Native principles described above and Architecture guidelines described in the next chapters; this document depicts the preferred architectural scenario for EMSA applications as well as the preferential components/products to support EMSA Maritime Applications implementations.

This document is not closed; it can be changed by adding new elements not initially foreseen and/or adding alternatives to the proposed components/products.

In the same line, exceptions to the preferred elements are possible; EMSA is open to discuss deviations from the preferred architecture and preferred components/products presented in this document, if dully justified. The adoption of any deviation and/or alternative must be proposed in advance by the project teams, dully justified (why, advantages, disadvantages,), discussed with EMSA competent groups (APCG, Cloud Centre of Excellence) and always subject to the final and formal EMSA's agreement.

1.2 Cloud Architecture

The diagram below presents the high-level components view of the EMSA cloud native reference architecture:

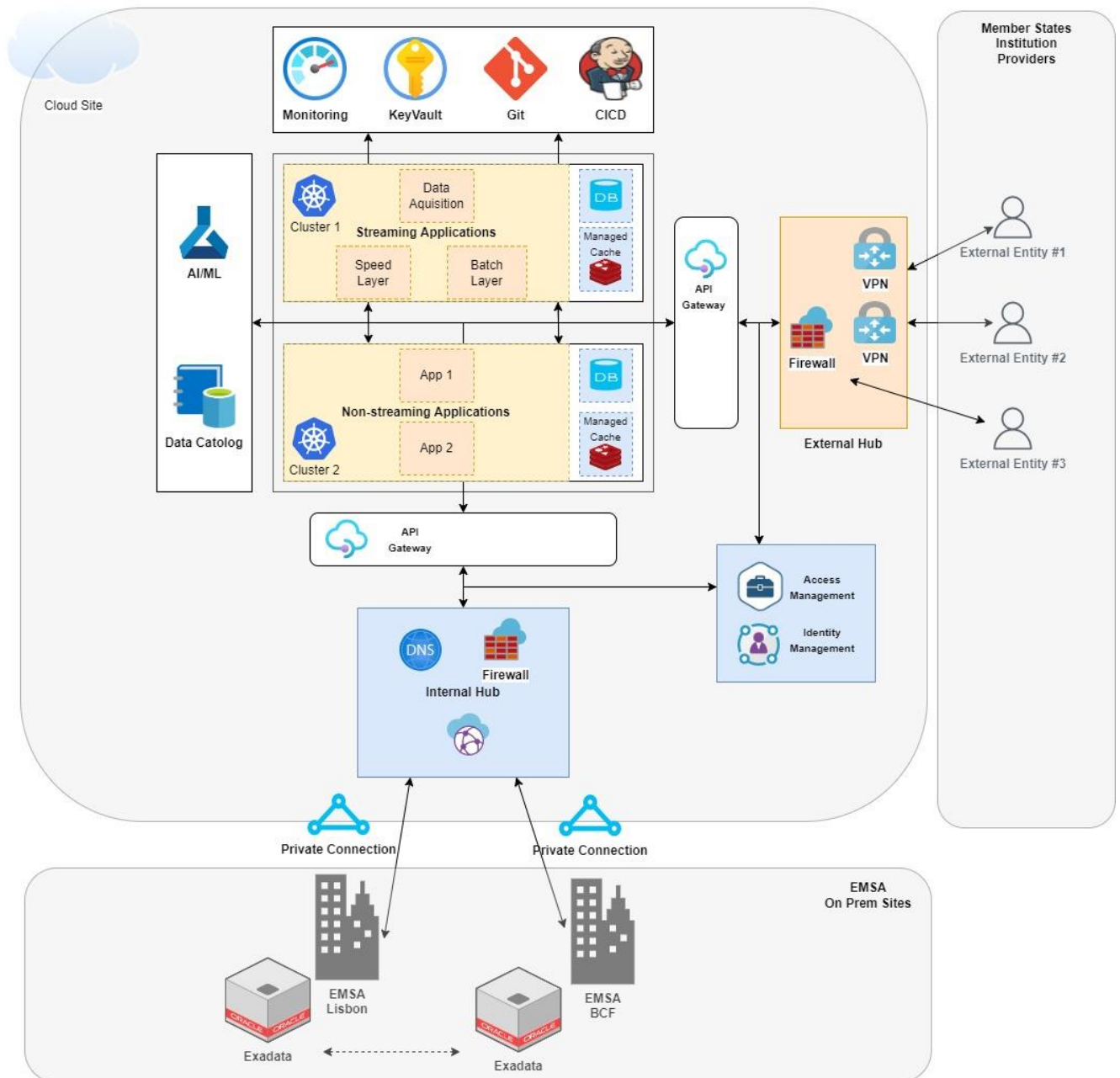


Image 1 – Reference Architecture

The image is based on three site blocks:

- EMSA on-premises**, represents an actual physical site containing all the existing systems that will prevail on-premises after a cloud migration movement. Those systems will be composed by equipment's and data, that, for reasons like legal requirements, cloud costs, application requirements and migration effort will not (yet) be migrated to run in a public cloud, but still require access to or by other services deployed in the cloud. Depending on the requirements, the site connection to the cloud may be private or use a public connection

- **Member states / Institutions / Providers**, represents the sites, that act like a private consumer/producer of EMSA data, and should not be seen as a public access.
- **Cloud**, represents the cloud provider(s) where all systems that are part of the cloud journey, or are “born” as cloud native, will reside.

A brief description of each one of the components in the previous image is presented below:

Kubernetes

For the management and deployment of the various applications packaged into containers there's a Kubernetes platform, as it allows to take advantage of advanced deployment strategies, scalability and high availability.

To support to the applications in scope of the Cloud Roadmap analysis, two clusters of Kubernetes are defined:

- One that will host applications and components that deal with the streaming of messages/events.
- A second cluster to host the remaining applications that fall outside of the streaming scope.

The various maritime applications will be divided into two architectural patterns: Lambda and Layered.

- The lambda architecture will be followed by the applications that deal with the streaming of events/messages.
- The layered architecture will be followed by the other maritime applications.

Shared Streaming

To handle the high volume of data of the several applications, a shared component for the streaming of events and messages is included.

Composable Apps

The applications will adopt a microservices architecture, i.e., each functional requirement will be met by a specific service. These services can then be shared by the various applications that use those functionalities.

API Gateway/Manager

To integrate with external systems, an API gateway/manager shall be used with the ability to authorize and authenticate all requests.

Service Mesh

Integration between internal microservices/services shall follow the Service Mesh concept

Monitoring

Monitorization of the infrastructure as well as of the applications themselves is done by services dedicated for that purpose and is represented by the monitoring component. Microservices will either push metrics to the monitoring components or will provide an API endpoint where this information can be collected.

Key vault

Handles any sensitive application specific data, such as passwords, connection strings, certificates, etc.

Git

This component is responsible for source code repository and version control.

CI/CD

Responsible for the automated continuous integration and deployment of the applications and infrastructure.

IAM

Provides Identity Management services to manage user accounts and Access Management services to authenticate and authorize users and systems across all maritime applications as well as Single Sign-On capabilities.

Databases

Handle data persistency, storing and supplying data for / to all the other layers. These services include Relational and Non-relational databases and **distributed caches**. The above image presents both cloud and on-premises (Oracle Exadata) data sources.

Data Catalog

Gathers metadata and automates the build of the data catalogue repository. These facilitate data discovery, data lineage analysis and data governance.

AI/ML services

Includes artificial intelligence and machine learning services capable of detecting patterns on data and offer a set of predictive features to support future challenges EMSA wants to address.

Cloud Storage

Includes cloud file/blob storage services for file/binary persistency, archive historical data, etc., capable of being used as hot and/or cold storage, providing high availability access.

2 Cloud architecture components

2.1 Application-level architecture

The high-level architecture diagram shown below, specifies the necessary building blocks to support the applications. This diagram also represents and differentiates between the horizontal components, that are maintained by EMSA (namely every component that is outside the scope of the application domain including API Manager and API Gateway) and those components that fall into the application's development purview.

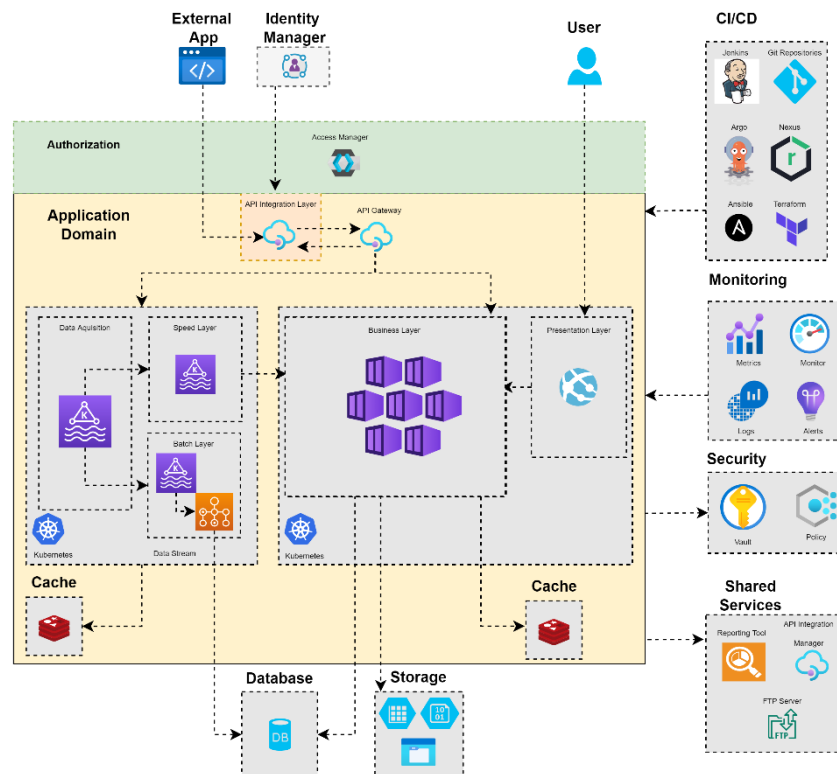


Image 2 – General Application Architecture

Although not shown in the above image, it would be possible to also deploy additional Linux Virtual Machines but it shall always be taken as an exception that must be justified in detail. The decision to deploy and use VMs shall be taken as exceptional, evaluated in a case-by-case basis, must be fully justified and always approved by EMSA.

2.1.1 Building Blocks

2.1.1.1 Streaming (Message/Event Broker)

Based on the need to analyse a high volume of data, a Lambda Architecture should be followed. This allows processing high volumes of data in two latencies: real-time and batch.

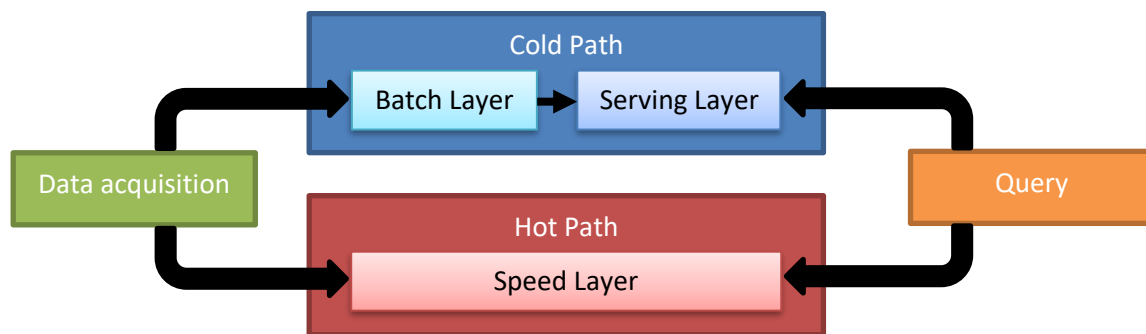
The main goal of following this approach is to achieve a generic, scalable, and fault-tolerant big data processing architecture.

When working with very large data sets, it can take a long time to run the sort of queries that clients need. These queries can't be performed in real time, and often require algorithms such as MapReduce that operate

in parallel across the entire data set. The results are then stored separately from the raw data and used for querying.

One drawback to this approach is that it introduces latency — if processing takes a few hours, a query may return results that are several hours old. Ideally, you would like to get some results in real time (perhaps with some loss of accuracy) and combine these results with the results from the batch analytics.

The **lambda architecture** addresses this problem by creating two paths for data flow. All data coming into the system goes through both the Hot and Cold Paths.



1. **Batch Layer (Cold path):** This component saves all data coming into the system as batch views in preparation for indexing. The input data is saved in a model that looks like a series of changes/updates that were made to a system of record. The data is treated as immutable and append-only to ensure a trusted historical record of all incoming data.
2. **Serving Layer (Cold path):** This layer incrementally indexes the latest batch views to make it queryable by end users. This layer can also reindex all data to fix a coding bug or to create different indexes for different use cases. The key requirement in the serving layer is that the processing is done in an extremely parallelized way to minimize the time to index the data set. While an indexing job is run, newly arriving data will be queued up for indexing in the next indexing job.
3. **Speed Layer (Hot path)**
This layer complements the serving layer by indexing the most recently added data not yet fully indexed by the serving layer. This includes the data that the serving layer is currently indexing as well as new data that arrived after the current indexing job started. Since there is an expected lag between the time the latest data was added to the system and the time the latest data is available for querying (due to the time it takes to perform the batch indexing work), it is up to the speed layer to index the latest data to narrow this gap.

Message/Event Broker

Event data will be pushed to an Event Broker. This will ensure the availability and scalability needed to process the large volume of data necessary for the applications.

The following figure shows a generic event processing flow:

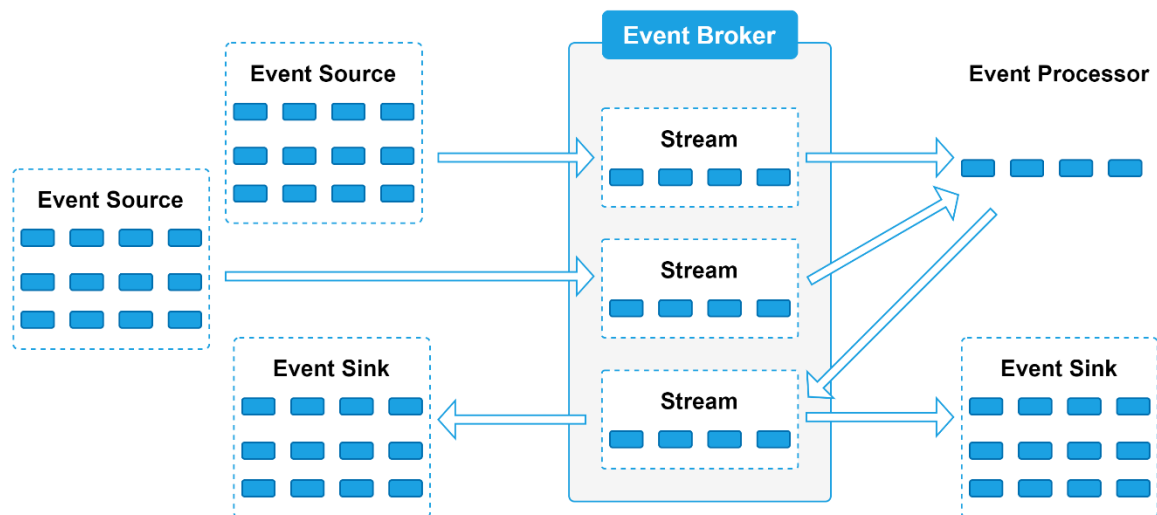


Image 3 – Generic event processing flow

The event broker can also improve decoupling between a service and the systems that depend on it.

Multiple event brokers can be deployed as a distributed cluster to ensure elasticity, scalability, and fault-tolerance during operations. This should be decided by analysing the criticality of each application, as an example, if an application is defined as critical and its failure could cause the entire system to stop then it's a prime candidate to have its own broker and mirror maker as a redundancy. Otherwise, non-critical applications should consume events/messages directly from the main streaming producer.

Event brokers collaborate on receiving and durably storing Events (write operations) as well as serving events (read operations) into Event Streams from one or many clients in parallel. Clients that produce events are called Event Sources and are decoupled and isolated, through the brokers, from clients that consume the events, which are called Event Sinks.

Typically, the technical architecture follows the design of "dumb brokers, smart clients". Here, the broker intentionally limits its client-facing functionality to achieve the best performance and scalability. This means that additional work must be performed by the broker's clients. For example, unlike in traditional messaging brokers, it is the responsibility of an event sink (a consumer) to track its individual progress of reading and processing from an event stream.

Tech Stack:

- Apache Kafka + Strimzi (open-source solution)

Strimzi is an open-source project that provides a simplified way to run Apache Kafka on Kubernetes. It leverages Kubernetes Operator pattern, providing three operators, to address the whole lifecycle from creating, managing, and monitoring Kafka clusters to providing features like topic and user handling:

- **Cluster Operator:** Responsible for deploying and managing Apache Kafka clusters. Can also deploy the Topic Operator and User Operator at the same time, as part of Entity Operator configuration.
- **Topic Operator:** Responsible for managing Kafka topics.
- **User Operator:** Responsible for managing Kafka users.
- **Kafka Connect:** Connects external systems to the Kafka cluster.
- **Kafka Mirror Maker:** Replicates data between Kafka clusters.
- **Kafka Bridge:** Acts as a bridge between different protocols and Kafka clusters.

2.1.1.2 Business Layer

Microservices

Applications shall follow the Microservices architecture model as much as possible. This allows for better control of functional responsibilities per service which in turn should make it easier and faster to change individual services, allows for the sharing of services between applications and to manage possible resource usage bottlenecks.

Microservices should be containerized, this will facilitate several of the Twelve-Factor application guidelines we mentioned earlier.

Microservices can interact with each other following both orchestration and choreography patterns. To further benefit from the containerized services we suggest the use of a Kubernetes platform, improving the deployment models, scalability and availability as already mentioned.

Tech Stack:

- **Azure Kubernetes Service (AKS)**
AKS is a fully managed Kubernetes to deploy and manage containerized applications, this managed service on Azure also offers an integrated continuous integration and continuous delivery (CI/CD) experience, and enterprise-grade security and governance.
- **Serverless Frameworks**

For running serverless code there are several open-source frameworks that can be used, but it is suggested to choose one that also integrates with the monitoring tools (see the dedicated chapter). We share a summary analysis of some viable options:

OpenFaaS

- Integrates with Prometheus
- Useful metrics available out of the box
- Able to call functions asynchronously
- Requires commercial capabilities to
 - Integrate with Kafka & AWS SQS
 - scale to zero for idle functions and
 - retry failed invocations.

KNative

- Provides specific tools to build code into containers
- Can be deployed on any cluster using Kubernetes own resource YAML
- Its provisioning can be resource heavy
- Integration with Prometheus
- Allows for scale to zero functionality
- Support for eventing and http serving
- Supported by RedHat and Google

Fission

- Provides Prometheus integration
- Offers autoscaling for functions based on CPU usage
- Allows for scale to zero scale functionality

Service Discovery

Since microservices architecture best practices imply that every module and responsibility should be isolated in a single service and to lighten the hassle of maintaining configurations throughout all applications, a service discovery model shall be implemented. Three components integrate this module:

- Service providers should register themselves with the service registry whenever they enter the system.
- Service registry keeps the network locations of the service providers up to date and thus it needs to have high availability ensured.
- Service consumer gets the network location of a specific service provider from the service registry and can then connect to it.

Tech Stack:

- There is no need to instantiate specific technologies, implementation should be based on native Kubernetes service discovery module.

Distributed Cache

Every application shall have its own segregated cache, this allows for less maintenance through the applications ecosystems. Using caches for highly requested data (and others highly used elements) is a way to achieve rigorous latency requirements needed. Additionally, distributed caches will ensure that the microservices themselves can be kept stateless, allowing for better scalability as well as disposability of containers.

All critical data should still be stored in a persistent format.

Tech Stack:

- Azure cache for Redis.
Azure Cache for Redis provides a fully managed in-memory data store based on the Redis software. Redis enables high-performance and scalable architectures where low-latency and high-throughput data storage are critical points. It's able to process large volumes of application requests by keeping frequently accessed data in the server memory, which can be written to and read from quickly.

2.1.1.3 *Presentation Layer*

UI

Maritime Applications User Interfaces shall use JavaScript libraries and frameworks such as React or Angular. This will allow to align them with UI and UX design best practices.

Applications that will require integration of various UIs of the other maritime applications shall follow an architecture based on the micro-frontends model. This type of architecture allows to decouple the responsibility of integrating the different UIs from a specific team and it also helps to standardize modules and packages shared by the different applications. However, if a specific application UI is not very complex in its functionalities an argument can be made to follow a different architecture than micro-frontends. This of course, needs to be evaluated on a case-by-case basis. The various UI components should also be containerized and served inside Kubernetes like the rest of the applications' services. This will allow, amongst other things, for a more precise management of load balancing clients' requests and of resources shared by applications.

Tech Stack:

- JavaScript libraries
Development of the UIs shall use JavaScript based libraries ReactJS, Vue or Angular. This will ensure that the skills and know-how needed to maintain applications will be kept at a manageable level.
- Apache Server
When needed, Apache Web Server can be used as the proxy/reverse proxy system; as it is already used in EMSA, it will take advantage of the existing know-how.
- At EMSA the solution chosen to implement the Micro Frontend (MFE) pattern is based on Module Federation, by Webpack. This solution provides a great way to integrate remote parts of code in the project as native components.
Module Federation is a functionality that allows Webpack latest versions (Webpack 5 and higher) to integrate components from remote origins to the host application, on execution time.
One of the main features of module federation is that some dependencies between the main application(host) and the MFE associated with them can be shared, granting a great performance, reducing the overhead of the page loading, and providing a great time-to-ready, granting a great user experience in general.
- Single Page Application: A single-page application, SPA, is an app that works inside a browser and does not require page reloading during its usage. As it can provide a better end-user experience, it should be taken into consideration in the design of EMSA applications/systems.
- Containers / Kubernetes
By containerizing the UI component and in conjunction with the Kubernetes platform this component becomes highly scalable and in keeping with a high availability model.

Nowadays, several of the Maritime Applications UI's are embedded in the Liferay Portal as Portlet components. It shall be noted that the current Liferay Portal is going to be gradually discontinued and therefore new implementation shall be designed taking this fact into consideration.

2.1.1.4 IAM of maritime applications

IAM

There are IAM three components on-prem: Identity Provider, Identity Manager and LDAP. These three components will handle user access control and user management to maritime applications.

The Identity Provider handles all authentications for the maritime applications and implements OAuth2 and OpenID Connect.

The Identity Manager, IdM-V2, handles account management lifecycle (creation of new accounts, modification of existent accounts, enable/disable). In addition, it implements all provisioning processes that pushes information to Maritime Applications that requires to receive accounts information. Usage of this "push" mechanism shall be reduced and avoid for new applications and gradually replaced by calls to IdM-V2 userInfo Web Services

The LDAP, is the main account directory (the user database)

Tech Stack:

- Identity Provider: KeyCloak

Keycloak allows to add authentication and secure services with minimum effort, providing user federation, strong authentication, user management and fine-grained authorization. This is also the component that will implement the standards of OAuth2 and OpenID Connect.

- Identity Manager: IdM-V2
IdM-V2 is based on Oracle Identity Manager and is the tool used to handle identity management for the architecture. It can be integrated with cloud providers resources through the use of specific connectors. This is the tool already used in EMSA for user provisioning.
- LDAP: openLDAP
Account/users database.

2.1.1.5 Security

Key Vault

For handling any sensitive application specific data, such as passwords, connection strings, certificates, etc. the use of a key vault is mandatory. Furthermore, the use of a key vault per application domain will allow for a better segregation of sensitive data.

Tech Stack:

- Azure Key Vault
This service is a cloud service used for securely storing and accessing secrets. A secret should be anything that we want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys.
Key Vault service supports two types of containers: vaults and managed hardware security module (HSM) pools. Vaults support storing software and HSM-backed keys, secrets, and certificates. Managed HSM pools only support HSM-backed keys.

Security by Design

All the development of new applications and/or changes on existing EMSA maritime applications should follow the best practices of a security by design approach. This means that applications should have built in, since its genesis, a security mentality that follows principles and guidelines such as:

- Avoiding serialization, specially of sensitive data. If needed class-specific methods should be written to handle serialization and ensure that sensitive data is not exposed to the serialization stream. Furthermore, deserialization of unvalidated and untrusted input should be strictly avoided.
- Zero trust principles should be followed, meaning that all access to the applications should be authenticated, authorized, and continuously validated.
- Never expose passwords or critical data when storing or sending from the browser to the backend using instead one-way cypher to store and transmit this type of data
- Make sure to check and sanitize against external input errors or attacks, like for example SQL Injection, and use tools like prepared statements,
- Never reveal implementation information like stack traces on error messages, among other best practices.

As designed on the architecture, the security of future applications is considered since the beginning and the choice of the components like Key Vaults (to store secrets, keys, etc.), API Management (to decouple applications services) and the integration with the Identity Provider will enable the authorization natively on the proposed architecture.

Periodic security assessments should also be in place. The various maritime applications should be tested to validate possible security risks. Again, following best practices, some examples of assessments are vulnerabilities testing, penetration testing, IT audits and IT risk assessment. These will help to map possible vulnerabilities of the system and their severity, specific targets that can be exploited like domain rights or data that could be stolen, they also help to assess compliance levels and the likelihood as well as impact of potential attacks. Some cloud providers also have services that can produce reports of security compliance and automated response, for the Azure example, Microsoft Defender for Cloud and Azure Sentinel.

As cloud provider-agnostic solutions, Qualys and Rapid7 InsightVM are examples of security suites and tools that can provide asset management (endpoints, cloud, servers, virtualized infrastructure, IoT, etc), updated vulnerability management and scanning, risk assessments and automated patching, while providing statistics, vendor lifecycle information and remediation instructions over multiple, heterogeneous and hybrid environments.

2.1.1.6 Monitoring

Monitoring is performed by a group of tools that can be transversal to all application domains. The collected data can be processed and sent to appropriate databases inside the monitor environment where they can be queried and shown on monitoring dashboards. The gathering of logs, metrics and traces can be managed by specific agents or by exposing specific APIs.

Tech Stack:

- **Azure Monitor**
This service allows for a precise analysis of availability, performance, and usage of the applications. It also as tools incorporated for distributed application metrics, log and traces analysis and alerts.
- **Azure Application Insights**
Application Insights should be used to monitor Azure cloud services apps for availability, performance, failures, and usage by combining data from Application Insights SDKs with Azure Diagnostics data.
- **Azure Arc and Agents**
With these tools it is possible to manage on-prem and cloud infrastructures. This way there is only a single location that monitors the whole infrastructure. The agents allow for the collection of needed information while Arc allows to manage and monitor the infrastructure.
- **Prometheus+Grafana and Elastic Stack**
To considering a more agnostic approach regarding metrics, monitoring and logging centralization. The above Azure solutions allows to have a centralized dashboard to cloud and on-prem infrastructure monitoring but may result in an increase in costs and imply the migration of the existing monitoring rules to another platform.

These solutions could be integrated to most of the on-prem assets, as well as cloud providers platforms (IaaS, Kubernetes platforms, managed databases, etc.) with the objective of centralizing all monitoring, alerting and automated response.

Please note that the decision regarding the usage of Azure components versus the agnostic “Prometheus + Grafana” shall be discussed and agreed on a project-by-project basis.

2.1.1.7 Shared Services

A shared services is any service that is horizontal to the applications, and therefore is not dedicated to specific one. Usage of the Shared Services is mandatory.

These services are not under the scope of the Maritime Applications; they are independent projects providing services to the Maritime Applications. Therefore, if a specific Maritime Application foresees the need for changes, they must be clearly discussed and agreed between both project teams. The major Shared Services

are listed below (Infrastructure services e.g., Firewalls, etc., although horizontal components, are not presented):

Portal

EMSA's Portal has multiple features, but currently its usage at EMSA is quite reduced. In the future architecture, EMSA's Portal will continue to serve as the entry point for single Sign-on operations. However, the usage of the Portal based on Liferay COTS will be gradually discontinued and this will be replaced by some other, still to be selected implementation.

Databases

They provide relational and non-relational data storage services to be used by each of the data layers of the applications, though considered as shared services.

Tech Stack:

- PostgreSQL
Managed PostgreSQL database service should be preferred to implement Relational Database services. As an example, Azure specific Managed PostgreSQL service offers several deployment options ("single server" deployment that provides database services on a single availability zone, including security, backups, patching and a 99,99% availability SLA, "Flexible server" deployment, offering high availability DB across multiple zones). In addition, PostGIS is a PostgreSQL extension that adds support for geographic objects, allowing location queries to be run in SQL. In the current context of EMSA's application landscape, this feature is of high importance due to the nature of the data handled at EMSA. This PostGIS extension is widely available on the major cloud providers offering managed PostgreSQL database services.

Kubernetes

As already mentioned, two Kubernetes clusters to give support to the applications are foreseen in the architecture.

Rancher

Rancher shall be used to centralize the deployment, authentication, access management and observability of multiple Kubernetes clusters.

API Management

For the APIs management we propose a two-tier pattern.

- For exposing any APIs to public and external networks the use of an API Integration Layer is advised to ensure authentication and authorization for all requests. This layer, by also being integrated with the Access Management component, will authenticate and validate requests, for example user roles should be validated at this tier.
- Communications between applications and/or services should be mediated by a second tier through an API Gateway which will manage all requests made between necessary applications. This way there is no need for managing the connections to each application's APIs during developments and/or updates to any applicational services.

Authentication and authorization of the requests will be ensured with the integration between the API Gateway and the IAM component. The gateway should integrate with the underlying Kubernetes service

discovery API. To manage the integration between services we suggest what already exists in EMSA, namely RedHat Fuse. This will be referenced as the API Integration Manager.

Tech Stack:

- RedHat Fuse
EMSA uses RedHat Fuse as the integration layer between applications / services.
- Azure Api Manager
This service allows to manage and secure all the APIs across cloud and on-prem and enables optimization of traffic flow, while increasing security and compliance requirements with functions like filtering IP addresses, limiting call rate, authentication. It is also compatible with both OAuth2.0 and OpenID Connect
- Azure Application Gateway.
This managed load balancing service that can perform a layer-7 routing and SSL termination.

IAM

The identity and Access Management components shall provide IAM services to all applications.

Message / Event broker

EMSA's maritime applications requiring a message broker / event stream shall use a horizontally managed service. The applications will require to follow naming guidelines and quotas according to best practices for multitenancy Kafka deployments.

Tech stack:

- Apache Kafka

FTP server

Given the requirements of EMSA for the sharing of and access control to files through FTP, EMSA's internal solution based on proFTP shall provide the FTP services to all applications.

Tech Stack:

- proFTP (base on redhat server deployed on-prem)

ENC and Geoserver

EMSA makes use of proprietary Electronic Nautical Chart data that are distributed using OGC Web Map Service (WMS) implemented on GeoServer to be consumed on the UI's. Both components, individually or integrated together, will continue to supply such services in the future architecture, are considered shared components.

Tech Stack:

- Geoserver server (by geoserver.org)
- CMAP SDK and map Files (by c-map.com)

2.1.1.8 Storage

Databases

For the data layer, the use of databases that take advantage of managed services should be privileged. This allows for distributed access to data, to address issues of availability and it shifts the responsibility to the service provider to handle backups either locally or with georedundancy. When it comes to the segregation of the data layers for each application there are two options, either one managed instance (database server) per

application which will allow for a more precise allocation of resources, availability and scalability requirements; or a horizontal shared managed instance (database server) with each database segregated through access policies for each application, this allows for the reusability and sharing of resources between the data layers of the various applications.

Blob Storage

When it comes to persistent unstructured data, blob storage shall be used. This allows for distributed access to files, to address issues of availability, it shifts the responsibility to the service provider to handle backups either locally or with georedundancy.

Tech Stack:

- Azure Blob Storage - the typical tiers are the following:
 - Hot → High storage cost / low access cost
 - Cool → Low storage cost / high access cost
 - Archive → lowest storage cost / highest access cost

The tier should be chosen accordingly with functional and technical needs of each application or specific process.

2.1.1.9 CI/CD

CI/CD

Centralized CI/CD technologies and automation principles are mandatory for all applications.

Tech Stack:

- Jenkins
To leverage what EMSA already uses today for the automation of integration and deployment of maritime applications, a central Jenkins instance to handle CI/CD pipelines of the various maritime applications shall be used.
- Azure Kubernetes Service
This service is better suited if the chosen path is to go for an instantiation of Jenkins inside of the Kubernetes platform. If there is a need to scale CI/CD pipelines or segregate them, by environments for example, it's easier to configure new pods to be run with Kubernetes.
- Nexus
For the storage of artifacts, Nexus shall be used. It supports the most common artifacts formats and is what EMSA already uses today.
- Terraform
For the creation of infrastructure components (IaC) we recommend Terraform scripts.
- Ansible
For the orchestration, configuration management and deployment of the applications we suggest the use of Ansible, more details in chapter [2.2.62-2.6](#).

2.2 Infrastructure Architecture

2.2.1 Cloud / On-prem access management

Azure AD will provide access to all resources under EMSA cloud subscriptions. The Azure AD will be managed and controlled centrally by EMSA.

The integration with EMSA's on-premise Active Directory installation will be part of the Cloud Landing Zone setup.

2.2.2 Hybrid Cloud Networking

Networking is a fundamental IT infrastructure capability that provide the technology to allow the communication between components.

In a cloud-based infrastructure, that communication can occur in a public cloud context, or in a hybrid way, where the communication flows between the public and private infrastructures, allowing a seamless integration between the traditional on-premises network technologies, with the cloud-based ones.

2.2.2.1 Network Topology

Hub and Spoke is a reference network topology, commonly used in cloud environments. The main concept is the existence of a hub network that acts as a central connectivity point, and from which isolated networks (spokes) can consume common resources.

This kind of approach allows for cost savings and components isolation.

Typical components that are shared are:

- Firewall
- DNS
- NTP
- Directory Services
- VPN Gateways
- Public IP

The main purpose of a spoke is to isolate a part of the components (like a specific application or an environment). As described, the spokes will use the shared services from the hub, to avoid components duplication, like having multiple DNS servers or VPN connections.

A cloud design that uses 2 Hub's is the architectural option implemented for EMSA. One should be used to centralize all communications from EMSA datacentres, and another to centralize communications from member states and data providers.

This is not a traditional hub-spoke approach. The base idea is that member states are not part of EMSA infrastructure, but provide or consume data directly to/from these systems, so, it seems appropriate to have a specific hub where connections from member states are concentrated, allowing segregation from the EMSA on-prem network.

All the connections between hub and spokes or between spokes, is provided by Virtual Network Peering, that allows communication between different virtual networks. This communication traffic is routed by the cloud provider private network, and never passes by public networks.

The image below represents an approach to EMSA cloud network architecture, based on the architecture described above.

The following diagram does not represent all environments/spokes and subnets. It is an example that only represent a subset of Production applications and the Pre-Production spoke.

Additionally, it also represents a database specific spoke.

The on-premises location identified in the diagram, are only representing the connection between the EMSA DCs and the cloud.

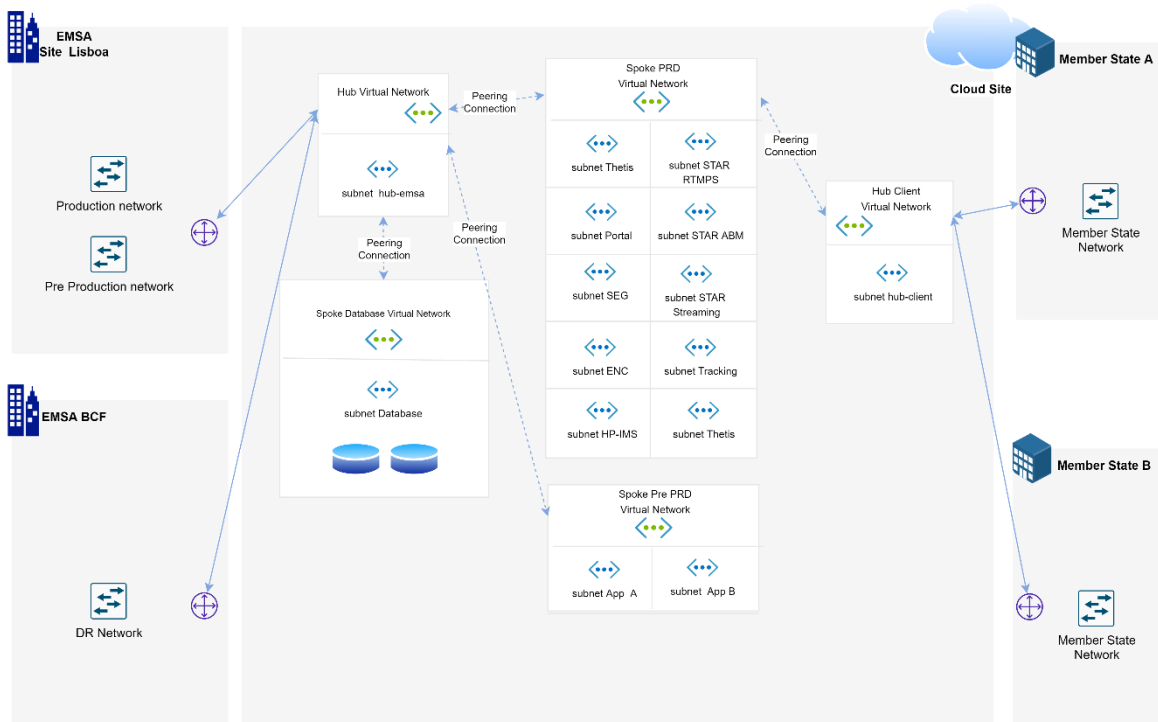


Image 4 Cloud network architecture

Network Hub-Spoke Topology Guidelines

- All common resources should reside in the Hub virtual network
- All common resources, related to member states connections, should reside in a “secondary” hub virtual network
- Each spoke represents an Environment
- Database resources will have its own spoke
- Spoke virtual network will contain resources related to application infrastructures
- Each Spoke virtual network, will contain multiple subnets
- Each subnet will represent an application network, and act as a boundary to the other applications in the same virtual network
- Communication hub <-> spoke or spoke <-> spoke will be performed by using virtual network peering

2.2.2.2 Hybrid network connection

To EMSA, the maintenance of a hybrid approach to cloud is an important aspect of any proposed architecture, as there is the need of maintain some components on-premises, therefore we will start by identifying the communication entry points to the cloud portion of EMSA infrastructure.

Public access *aka* Internet access

2.2.2.3 Hybrid Cloud Security

The presented architecture is considering Firewall before Application Gateway.

This kind of approach is used to minimize the number of Application gateways needed to support.

The following rules are considered best practices to a Firewall before Application Gateway positioning situation:

- Inbound HTTPS traffic, should be sent to the Firewall and then to Application Gateway
- Outbound traffic (HTTPS or HTTP), should be sent to the Firewall, bypassing the Application Gateway

Digital Certificate strategy

In order to support HTTPS communications, TLS/SSL certificates must be used. Entrust TLS/SSL is a digital certificate tool that can provide certificate management, issuing, renewal and reporting. Regarding Azure, Entrust TLS/SSL integrates with Azure Key Vault, enabling storage and native integration with the suggested platforms. Additionally, it is also possible to integrate Entrust TLS/SSL features with non-Azure infrastructure, relying on automation modules and exposed APIs.

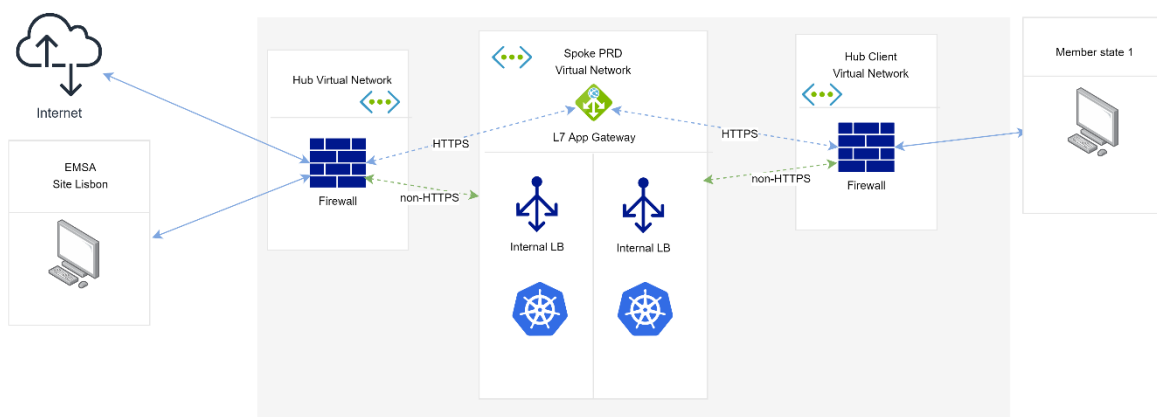


Image 6 – Infrastructure Security Architecture

The need to maintain all communication flow through firewalls. have two main reasons:

- If needed, having a mechanism that allows to block some kind of communication.
- Even if is not blocked, having the possibility to log the communication traffic.

2.2.3 Integrated Monitoring

Observability plays a crucial role when implementing Cloud-Native applications and platforms. One of the biggest challenges regarding monitoring is to establish a centralized strategy to collect, process, store and infer from monitoring data.

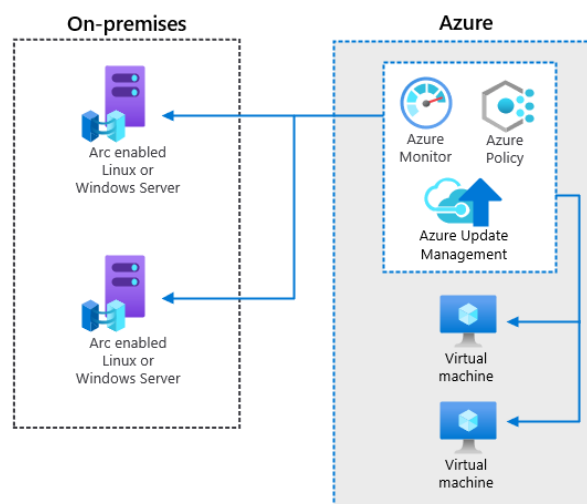
Platforms and applications deployed on Azure can integrate with the Azure Monitoring framework. This includes collection of:

- Platforms Health Alerts
- Platforms Security Events
- Infrastructure Logs
- Infrastructure Metrics
- Application Custom Logs
- Application Tracing

By keeping this data in a centralized environment, automated querying and alerting can be implemented. This may include automated resolution upon receiving infrastructure operational alerts, application outages and SOC incidents.

The collection of monitoring data can be achieved by enabling extensions and addons on the Azure Platform. Additionally, EMSA currently runs most of its infrastructure on-premises, Azure Arc can be enabled to monitor on-premises (or additional clouds) environments in the same way as Azure resources. By installing the Azure Arc Agent, these "hybrid" servers are treated as Azure Resources, enabling the homogeneous Azure framework that connects governance, policies, automation and ultimately monitoring.

The following diagram illustrates this integration:



In this proposal EMSA Splunk log monitoring solution will be replaced by Azure Log Manager, for applications deployed in Azure and for those deployed in-house or in a different public cloud. Whilst it is important to have all logs available in a single location, to be able to correlate logs over different deployment locations, we still need to evaluate the costs associated (due to the big amount of logs that EMSA generates) to this as well as how the integration with the SOC – that is currently heavily based on the Splunk platform – could be supported in this scenario.

2.2.4 Business Continuity / Disaster Recovery

The migration of EMSA's applications to the cloud needs to include Business Continuity / Disaster Recovery (BC/DR) mechanisms.

Although infrequent, services outages may happen. This may include faulty VMs and AKS nodes, corrupted storage hardware, datacentre network instability, region-wide DNS resolution problems etc. If regional

redundancy of services can't mitigate the fault, inter-geographical BC/DR strategies should be ready to provide continuous service.

Azure provides a set of BC/DR functionalities on the Azure Site Recovery framework for sites and infrastructure like recovery plans, automated synchronization and failover drills (failover to a test environment in order to assess synchronization, networking and availability responses). Additionally, depending on the service nature, different strategies could also be applied:

- **Infrastructure and Applications:** Relying on IaC modules, instantiated infrastructure and applications can be stored as templates and configuration files (Terraform) in a code repo (GitLab) and application repositories (package repos, container registries), ready to be redeployed at any moment to a different region. Leveraging automation, this process can be started following an outage, manually or by automated triggering.
- **Data:** Services like Azure Storage Accounts have synchronization, replication, failover and DNS redirection mechanisms that could be configured. Depending on the specific resource and requirements, geo replication could be configured.
Using a Managed PostgreSQL database service, offers failover mechanisms and/or very high SLA regarding for Relational Database instances. As an example, Azure Managed PostgreSQL service offers several deployment options serving different failover levels:
 - “single server” deployment that provides database services on a single availability zone, including security, backups, patching and a 99,99% availability SLA.
 - “Flexible server” deployment, offering high availability DB across multiple zones.

2.2.5 Multicloud Kubernetes

Containerization is the process of creating software packages, with the code and the required dependencies, in single lightweight “containers” that run independently of the underlying infrastructure.

Due to the benefits like portability, efficiency and agility, containers have been used as one of the approaches to cloud based applications.

With the rapidly growth of containerized applications, one of the biggest challenges was the management and orchestration of multiple container instances. As solution to that challenge, the most popular system used is Kubernetes (K8s)

“...is an open-source system for automating deployment, scaling, and management of containerized applications.”
(<https://kubernetes.io/>)

K8s can be used and implemented in a “traditional” way, using the unmodified, upstream open-source components (aka as vanilla Kubernetes), but the needed know-how and the effort to implement and maintain such implementation is too high.

A more common approach is to typically distributions, that create enterprise ready K8s solutions, that are ready to deploy (with all the needed ecosystem resources), and typically offer an GUI to facilitate the management and operation.

Today, most of the cloud providers, have is one K8s distribution, with the most common being:

- Microsoft Azure Kubernetes Services (AKS)
- Amazon Elastic Kubernetes Services (EKS)
- Google Kubernetes Engine (GKE)

But they all have in common, that they must be managed inside is one cloud context.

In a hybrid and multicloud approach, having a reduced number of management consoles, and other operational tools, is a “must have”, so, is purposed the use of tools that provide a single pane of glass to multiple clusters across any infrastructure (cloud or on-premises), in a way that create a Kubernetes-as-a-service experience.

One of those tools is Rancher. Rancher provides an Enterprise grade solution to centrally manage any CNCF-certified Kubernetes distribution.

Rancher is a tool to centralize the deployment, authentication, access management and observability of multiple K8s clusters, compatible with distributions across multiple infrastructures, environments and providers.

The following diagram represents an approach of an multicloud K8s architecture, managed by Rancher, in EMSA context.

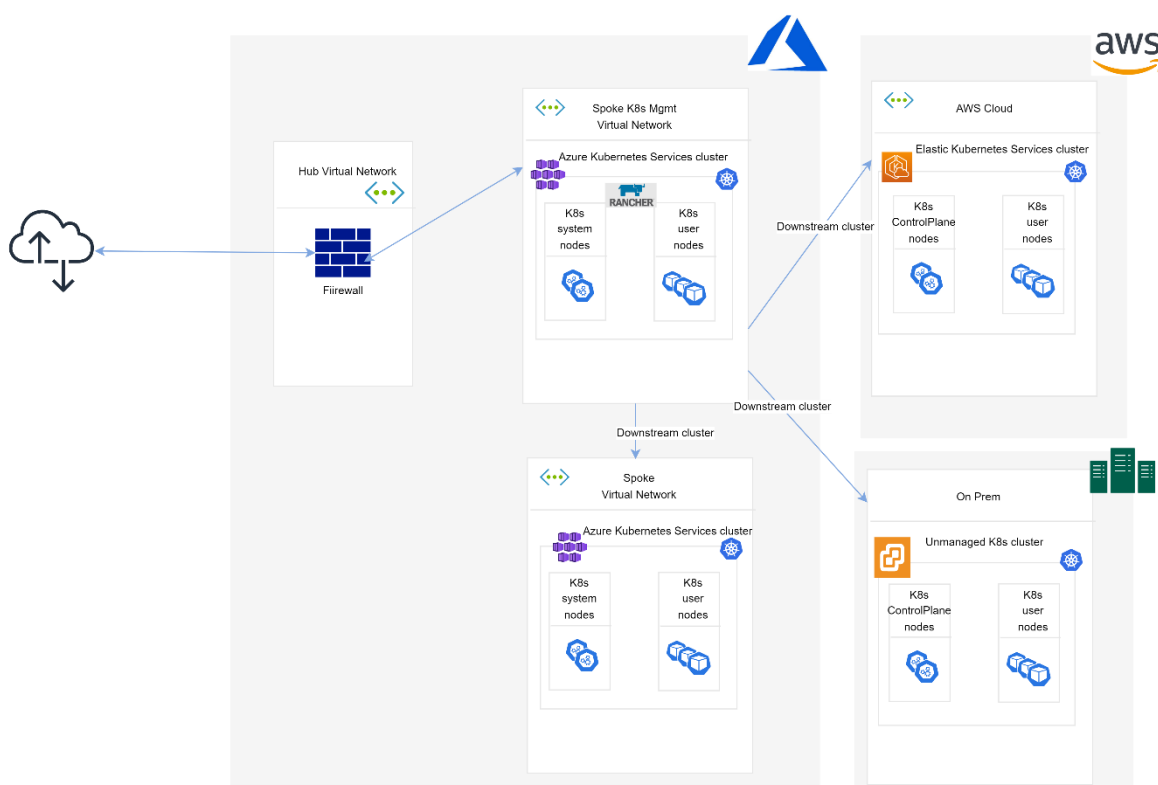


Image 7 – Rancher - Multicloud Kubernetes Architecture

In a high-level context, Rancher has two major “blocks”:

- The Rancher management product
- The downstream clusters (the clusters where the user application will be deployed)

To achieve high availability, the Rancher management can be deployed in any existing Kubernetes cluster. In the presented scenario, is deployed in a AKS cluster, but if needed, can be deployed on-premises (using for example, a vSphere virtualization infrastructure to provide compute resources) or in another cloud provider.

After that, Rancher will be able to deploy and manage the downstream clusters. In the presented scenario, an AKS, an EKS and an on-premises vSphere-based implementation are being used as downstream clusters.

2.2.6 Infrastructure as Code

Infrastructure as Code (IaC) is an alternative process of deploying and configuring infrastructure datacentre components, such as (but not limited to) virtual machines and networks, instead of using the interactive and manual configurations.

This “code” is written in a declarative way, in configuration files. After it is created, it is interpreted by the IaC tool, that will deploy the infrastructure with the defined specifications.

The main advantage of using an IaC approach is to make things faster and automate repeatable tasks, eliminating manual processes.

To reach its full potential, the following guidelines should be implemented:

- Source code is stored in a Git repository, and used as *single source of truth*
- IaC source code will be versioned, as application code
- Changes to the infrastructure, are implemented in the source and not on the target
- The combined utilization with CD tools will be privileged
- Best practices, as code reviewing, should be implemented

Each Cloud Provider has its own native tools to implement IaC, like Azure Resource Manager (ARM templates) and AWS CloudFormation, but they are limited to its own cloud context.

To achieve multicloud portability, a more cloud-agnostic tool needs to be used. EMSA has adopted:

- Terraform
- Ansible

Note that not all the referred tools are a pure IaC tools, but all of them can be used with that purpose. Ansible are more configuration management tools, so the combined use with terraform can be achieved with great results.

The following diagram represents an IaC architecture based on Terraform and GitLab.

In the scenario, is used a GitLab deployment in Azure. GitLab component aggregate the git repositories, the pipeline workflow and the Terraform engine. After that, the deployment happens in three different clouds: Azure, AWS and on-prem private cloud.

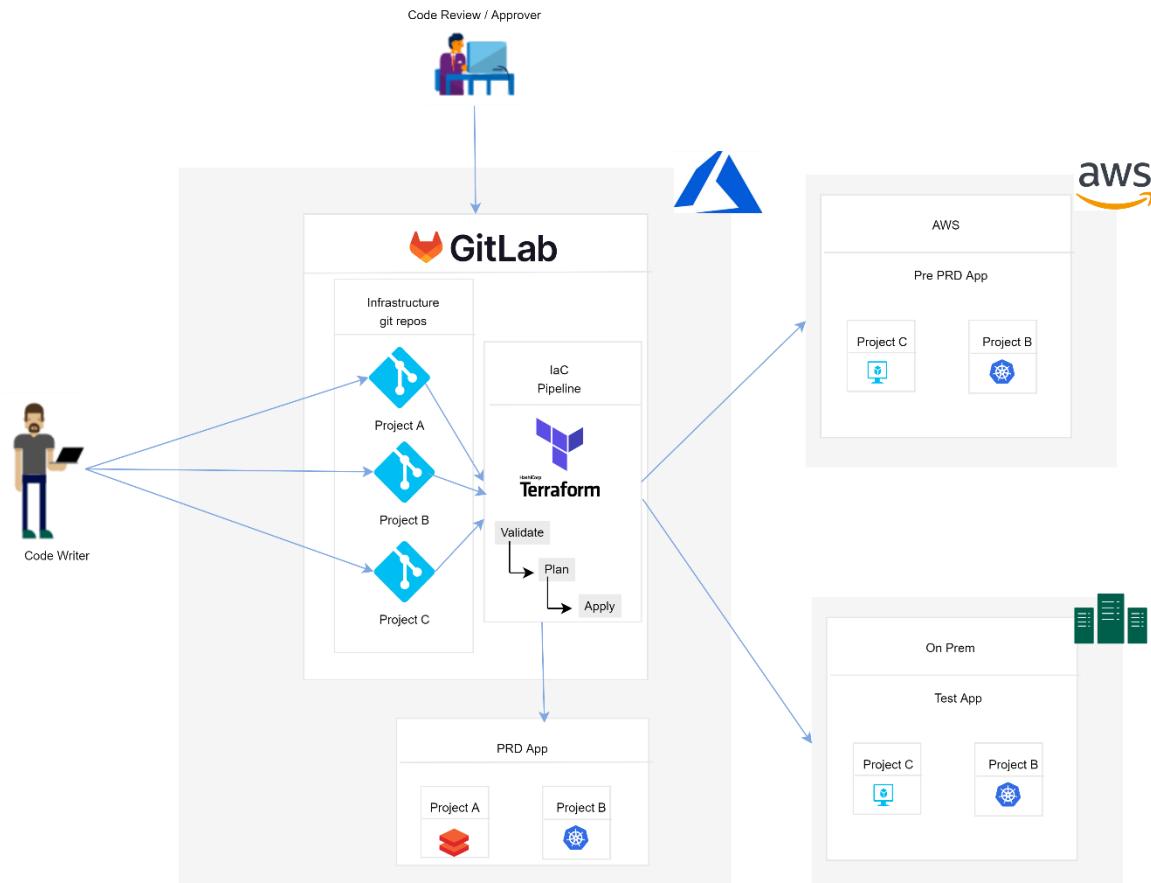


Image 8 – Gitlab/Terraform Workflow

Tech Stack:

- Terraform / Ansible

2.2.7 CI/CD and automation

Continuous Integration and Continuous Delivery is adopted by EMSA for all projects lifecycle. In CI/CD, pipelines are a core concept; pipelines are a compilation of procedures that allow us to transform repeatable tasks in predictable and repeatable automated actions.

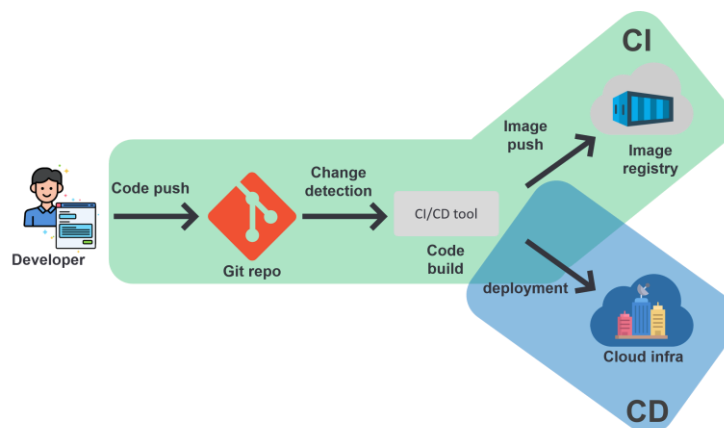


Image 9 – CI/CD pipeline flow

In an application lifecycle, we have the following steps to CI/CD pipeline integration

1. **Code push** – Where a new code is pushed to an application Git (EMSA used GitLab)
2. **Image build** – The CI/CD tool detect changes in the application Git and start the application image build pipeline
3. **Deployment** – After the application image build, the CI/CD tool apply the build in the infrastructure as code (IaC)

In this way, is possible to execute a full application life cycle, with a single code push to a Git repository.

However, if there is a manual intervention on the infrastructure, it is possible to provoke a configuration drift which may remain unnoticed, or, in worst case, may result in a broken application. To correct that it will be necessary to manually run the CD pipeline.



Image 10 – Configuration drift in manual infrastructure changes

The **GitOps** approach, complements the traditional application lifecycle, where, in addition to the addressed issues, previously referred, help us to maintain a versioned IaC, in a Git repository, like an application code.

Since GitOps is a methodology, it also uses dedicated tools to achieve the IaC deployment, where is focus is the deployment part (CD).

In this scenario, a push/pull deployment topology is adopted, where a dedicated CD tool, ArgoCD.

ArgoCD is constantly comparing the IaC stored in the Git, with the real infrastructure, detecting any configuration drifts, and applying remediation, automatically, if needed.

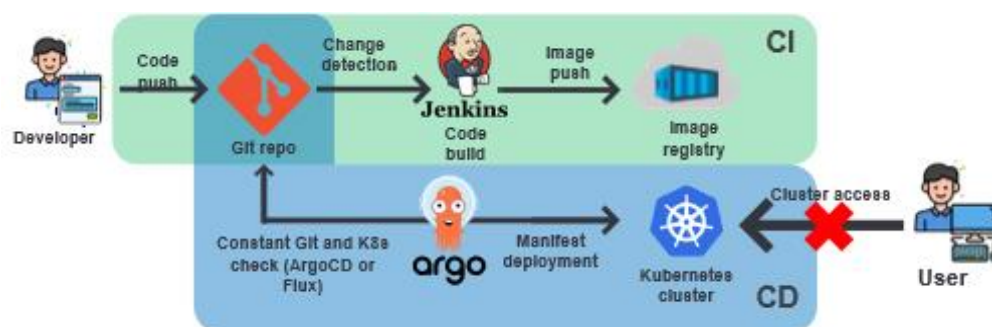


Image 11 – ArgoCD GitOps flow

In this approach, the Git is used as Single Source of Truth for application code and infrastructure code.

In an application lifecycle with GitOps, the following steps exit to an CI pipeline integration with a dedicated CD tool:

1. **Code push** – Where a new code is pushed to an application Git
2. **Image build** – The CI/CD tool detect changes in the application Git and start the application image build pipeline
3. **IaC update** – The CI tool change de IaC with the performed application changes (only in a multi-repository architecture)
4. **Deployment** – The CD tool detect the changes in the IaC code and apply the changes.

Using this kind of approach, in a Kubernetes context, tools like ArgoCD can exist in Kubernetes clusters, but not limited to his own cluster. In this case, ArgoCD will extract the IaC from the configured Git, and deploy in any configured target Kubernetes cluster.

Note: ArgoCD is a CD tool, dedicated to Kubernetes.

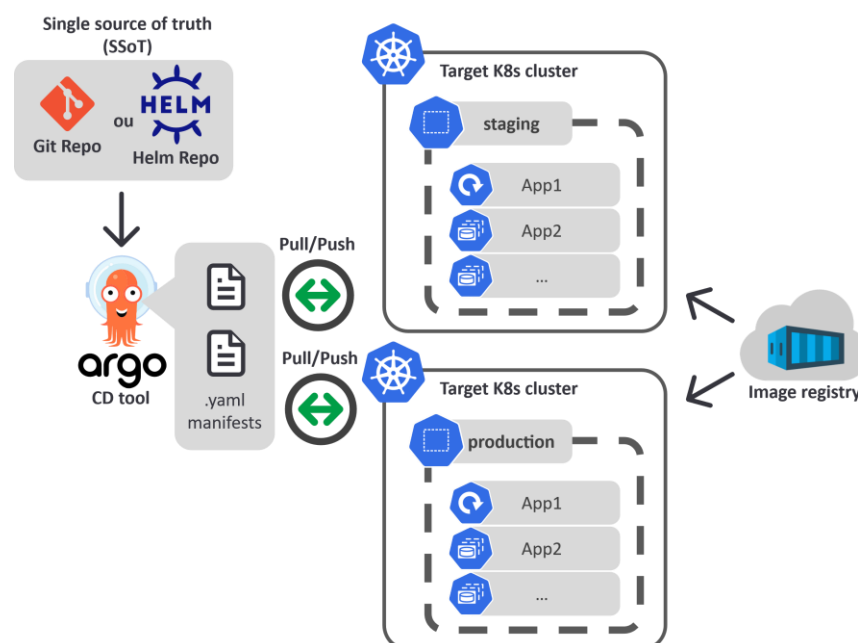


Image 12 – ArgoCD architecture

This approach allows to implement a series of improvements, in the CD processes, such as:

- Remove the need to perform manual accesses to the infrastructure, reducing the configuration drift probability, that can take a while until they are detected:
- Self-healing capabilities, in situations that configuration drifts are provoked by external factors
- Promote the use of Infrastructure as Code, allowing versioning and incremental evolution, with the consequently accountability of its management
- Increment the portability of the infrastructure
- Access to graphical consoles that helps in the infrastructure visualization and platform abstraction

Simplifies the CD process to large or small application lifecycles scales.

Tech Stack:

- Terraform / Ansible
- GitLab

- ArgoCD

2.2.8 Governance

Governance provides mechanisms and processes to maintain control over cloud resources.

2.2.8.1 Subscriptions

Subscriptions are the starting point of a cloud infrastructure creation.

They are a logic unit, where all the resources are created, and are associated to a unique billing. Therefore, the creation of multiple subscriptions is typically associated with billing requirements, and not with Operational requirements.

EMSA's environments and applications should be segregated at Azure subscription-level. In the following diagram, a mix between different agencies and applications can be achieved with the use of management groups.

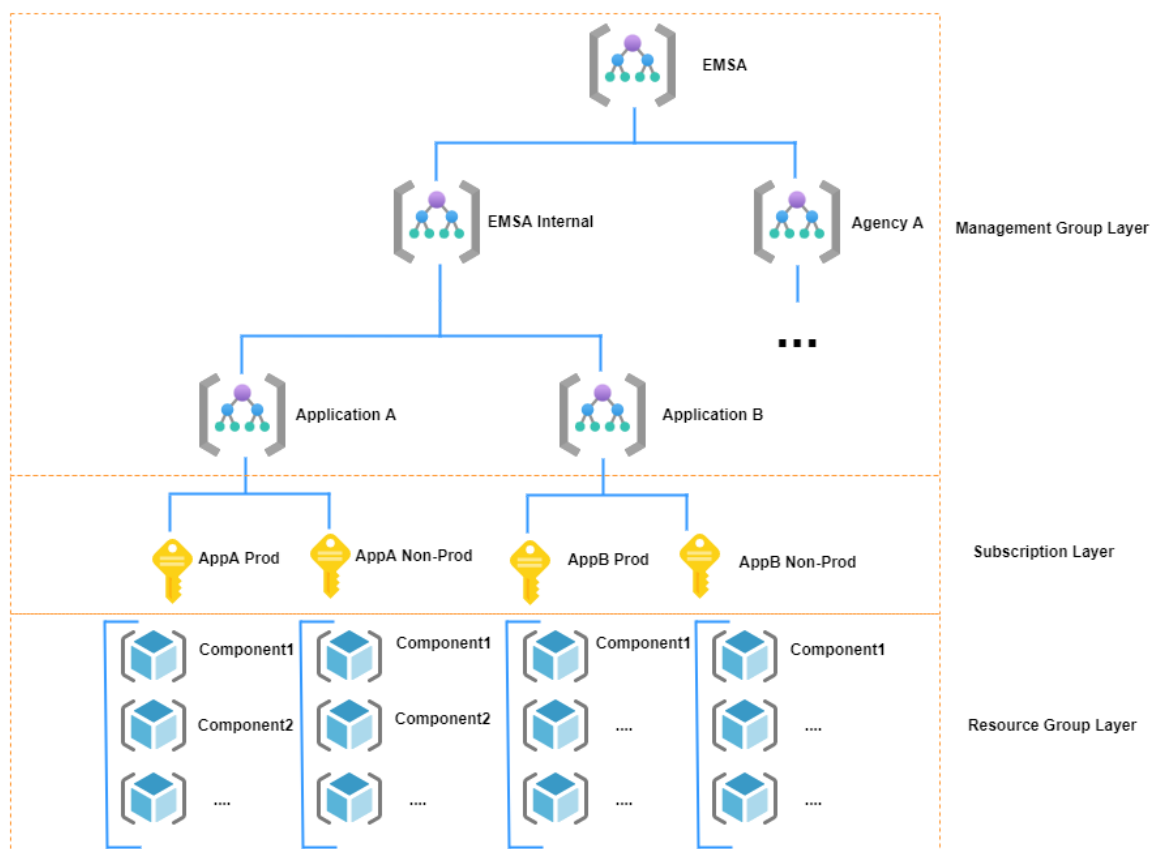


Image 13 – Subscriptions

In this approach, each “funder” (in which EMSA is the most relevant) should have their own set of applications which ultimately have a set of subscriptions per environment.

2.2.8.2 Naming convention

Naming convention is the definition of name structure standards used in cloud objects.

Cloud objects tend to “explode” in number, so, a well-defined structure is fundamental for good resource management; it helps to identify the object type, the region an object belongs to, or the kind of workload.

The naming convention shall be applied to all cloud objects, from subscriptions, to resources.

Resource names, have scopes where the name must be unique:

- Resource level → name must be unique in the resource context. Example: subnet names must be unique in virtual network context
- Resource Group level → name must be unique in a resource group. Example. Virtual machine names must be unique inside a specific resource group
- Global → name must be unique across all Azure. Example: Storage account

A good example of naming convention is the general one proposed by Microsoft¹. However, EMSA has a more specific convention already being used in several EMSA cloud Projects running on Azure that can be found in annex **“Azure Naming Conventions for EMSA projects.docx”**. Following these conventions is mandatory for all projects.

2.2.8.3 Tagging

A tag is defined as a key: value pair. It can be applied to any cloud resource, and is used, to help to:

- resources categorization for activities like automation
- cost monitoring
- billing filter
- logging
- object history

Like naming, tagging should be enforced by policies, and defined in IaC configuration files. Typically, these are divided in 2 groups: Business and Operational

Business	Operational
Cost Center	Project
Criticality	Owner
Department	Environment
	Creation date

Table 1 –Tagging – key's

The table above is an example of commonly used tags, but tags should be adapted for the most important keys relevant to the organization.

2.2.9 Data analytics

EMSA already uses Databricks as its most important platform for Data Analytics. This will continue to be the default choice, also for Cloud Native developments hosted in cloud.

¹<https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-naming>

2.3 Data catalogue

A modern data catalogue solution constitutes a key aspect in implementing an efficient data Governance strategy. Data catalogue services are included in the proposed cloud services framework. These data catalogue services will need to be capable of integrating with other technologies, both on cloud and on-premises data sources, in order to gather the metadata to build the data catalogue repository. Also, data catalogue services offer metadata management features, targeted to discover data and support data governance, by offering web interfaces for users to query and view an inventory of assets and create/manage a business glossary.

As these types of services gather metadata, they will need to offer the following Key Capabilities:

- Harvest technical metadata from a range of data sources, as wide as possible, and provide access to these metadata using public or private IPs.
- Create and manage a common enterprise vocabulary with a business glossary. Build a hierarchy of categories, subcategories and terms.
- Offer the users the possibility to enrich the gathered technical metadata with annotations, comments and/or free-form tags, linking data entities and attributes to the business terms.
- Search features to explore data assets and allow users to browse the data catalogue.
- Possibility to automate the data harvesting jobs.
- Offer integration of the data catalogue with other applications using REST APIs and SDKs.

Data catalogue services tend to be offered as serverless managed services, meaning EMSA does not need to provision any kind of resources (compute, storage, etc) to use the services, although this option depends on each cloud provider's particular offering.

According to the database technologies in use at EMSA and the future cloud databases use cases, the data catalogue service should support the following technologies in the process of metadata gathering:

- Oracle standalone Database and Exadata DB Systems
- Object Storage (although this depends on each cloud provider offering)
- PostgreSQL
- Microsoft SQL Server and Microsoft Azure SQL Database
- Apache Kafka
- XML files (.xml, .xsd)
- Avro files (.avro, .avro.gz)
- Excel files (.xls, .xlsx)
- Simple JSON files (.json, .json.gz)

The choice of such a tool depends on the chosen cloud provider, as the major providers all offer data catalogue solutions integrated "out of the box" with the vast majority of their other services and with extended features to link to in-house sources to gather metadata. Microsoft Purview Governance Portal is a good example of this, as it offers complete integration with Azure services and wide connectivity with other technologies ².

For a cloud agnostic alternative, we recommend the use of Apache Atlas, on which Microsoft Purview is based, as the solution for metadata management and governance of data.

² For a complete list of supported technologies please visit <https://docs.microsoft.com/en-in/azure/data-catalog/data-catalog-dsr>

2.4 Cloud architecture technology landscape matrix

	Cloud Agnostic	Azure specific
Relational Database	PostgreSQL supported on a Managed services (PaaS)	PostgreSQL supported on a Managed services (PaaS)
NoSQL Database	MongoDB supported on a Managed services (PaaS)	Azure CosmosDB for MongoDB
AI/ML Services	No specific technology recommended, only desired service features	
Data Catalogue services	Apache Atlas	Microsoft Purview Governance Portal
FTP Server	No specific technology recommended, only desired service features.	
Streaming & Messaging	Apache Kafka + Strimzi	
CI/CD	Jenkins, Argo, GitLab, Ansible, Terraform, HELM, Nexus	
Orchestration & Management	Kubernetes, Suse Rancher	Suse Rancher, Azure Kubernetes Service
Serverless	OpenFaas, KNative, Fission	
API Integration Layer	Red Hat Fuse, Axway API Gateway	Red Hat Fuse , Azure API Manager, Azure Application Gateway
Service Mesh	Istio	
Platform		Azure Cloud
IAM	LDAP, Keycloak (OpenID Connect, OAuth2) , AD, Oracle Identity Management	
Cache	Redis	Azure Cache for Redis
Microservices	Java (SpringBoot, Quarkus), OpenJDK, GO, Node.JS,	
User Interface	React, Angular, Vue	
Web Server	Apache HTTP Server	
Geographic Information Systems	GeoServer, CMAP	
Private Connection		Azure Express Route, Azure Site-to-Site VPN
Container Registry	Nexus Repository	
Secrets Vault		Azure KeyVault
Backup and Recovery	CommVault	Azure Backup and Azure Site Recovery
Monitoring	Prometheus + Grafana, Elastic Stack	Azure Monitor
Linux VMs	RedHat	

Table 2 – Cloud architecture technology options landscape matrix

2.5 Project related tools

AREA	TOOL	Available
Team Collaboration and Document Management Tools	EMSA Teams	YES
	EMSA Confluence	YES
Requirements Registry	EMSA JIRA	YES
Issue Tracking System	EMSA JIRA	YES
CI/CD	EMSA Jenkins	YES
	ArgoCD	YES ³
	Nexus	YES
	SonarQube	YES
	gitLab	YES
	Ansible	YES
	Flyway	YES
	Puppet	YES
Tests Management	EMSA JIRA + Xray	YES
Automatic Testing Tools	ReadyAPI	YES
	JMeter	YES
	Cucumber with Cypress	YES
	Selenium	YES
Monitoring Tool	Azure Insight	YES ⁴
	Azure Monitor	YES ⁵
	Prometheus	NO
	Grafana	NO
	Nagios	YES
Log tracking tool	Splunk	YES
	Or other similar tools	

³ YES, after Landing Zone availability

⁴ YES, after Landing Zone availability

⁵ YES, after Landing Zone availability

Information Security Risk Assessment methodology	ITSRM2 (DIGIT)	--
Secure development best practices	OWASP Top Ten Or equivalent industry standards	--
Security assessment Tools	Checkmarx Veracode Or equivalent tool, or combination of tools that allow automated source code AND dependencies' security assessment.	NO NO
Configuration files security assessment tools	GoGuard Or equivalent	NO
Vulnerability assessment Tools	Acunetix (Web vulnerability assessment) Nessus (Network vulnerability assessment) Or similar full automated web and network vulnerability assessment tools	NO
Security Monitoring Tools	Vulnerability scanners like Detectify or Nexus or OpenVAS or equivalent; EDRs on Endpoints and servers like WD-ATP or equivalent; Static and Dynamic Code Analysis tools like Checkmarx, Veracode, SonarQube or equivalent tool; PenTest tools; Log analysis and correlation based on IoC and Yara rules on tools like Splunk or Sentinel or equivalent; Artificial Intelligence anomaly detection with tools like DarkTrace or WD-ATP or equivalent;	NO NO YES ⁶ NO YES ⁷

⁶ YES, only for SonarQube

⁷ YES, only for Splunk

	IPS and IDS like Snort or Cisco SourceFire or equivalent; WAF like F5 ASM or Barracuda or equivalent etc...	NO
		NO
		NO