D5.1 - Demonstration of big-data framework for situation awareness on fire danger index

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 101037247

Project Acronym	SILVANUS
Grant Agreement number	101037247 (H2020-LC-GD-2020-3)
Project Full Title	Integrated Technological and Information Platform for Wildfire Management
Funding Scheme	IA – Innovation action

DELIVERABLE INFORMATION

Deliverable Number:	D5.1
Deliverable Name:	Demonstration of big-data framework for situation awareness on fire danger index
Dissemination level:	Public
Type of Document:	Demonstrator
Contractual date of delivery:	30/06/2023 (M21)
Date of submission:	01/08/2023
Deliverable Leader:	CMCC
Status:	Final
Version number:	0.5
WPLeader/ TaskLeader:	DELL/DELL
Keywords	Big Data Analytics Framework
Abstract	The deliverable will showcase the technical integration of big-data framework elements for the development of situational awareness on the threat of forest fire.

Deliverable Leader:	CMCC
Lead Author(s)	Marco Mancini
Reviewers	

Document History			
Version	**Date**	**Contributor(s)**	**Description**
V0.1	26.04.2023	Marco Mancini (CMCC)	Proposal of ToC
V0.2	11.05.2023	Jose Ramon Martinez (ATOS), Maria Maslioukova (CTL), Ciro Caterino (EAI), Mustafa AlBado (DELL)	Individual section contribution
V0.3	23.05.2023	Aris Bonanos (EXUS), Nohora Sanchez (VTG)	Section contribution and initial editorial review
V0.4	28.06.2023	Geordios Diles (EXUS), Mustafa AlBado (DELL), Oikonomou Panagiotis (UTH), Georgia Christodoulou (CTL), Nohora Sanchez (VTG)	Revision of sections, including executive summary and introduction, and conclusion and references.
V0.5	30.06.2023	Marco Mancini (CMCC)	Final review of the deliverable and released for internal review

List of Contributors

Partner	Author(s)
CMCC	Marco Mancini
CTL	Georgia Christodoulou, Konstantinos Avgerinakis, Maria I. Maslioukova
EXUS	Aris Bonanos, George Diles
DELL	Mustafa Al-Bado, Matthew Keating, Deirbhile Healy
UTH	Oikonomou Panagiotis
ATOS	Jose-Ramon Martinez-Salio
VTG	Nohora Sanchez, Tomas Piatrik, Krishna Chandramouli

List of beneficiaries

No	Partner Name	Short name	Country
1	UNIVERSITA TELEMATICA PEGASO	PEGASO	Italy
2	ZANASI ALESSANDRO SRL	Z&P	Italy
3	NETCOMPANY-INTRASOFT SA	INTRA	Luxembourg
4	THALES	TRT	France
5	FINCONS SPA	FINC	Italy
6	ATOS IT SOLUTIONS AND SERVICES IBERIA SL	ATOS IT	Spain
6.1	ATOS SPAIN SA	ATOS SA	Spain
7	EMC INFORMATION SYSTEMS INTERNATIONAL	DELL	Ireland
8	SOFTWARE IMAGINATION & VISION SRL	SIMAVI	Romania
9	CNET CENTRE FOR NEW ENERGY TECHNOLOGIES SA	EDP	Portugal
10	ADP VALOR SERVICOS AMBIENTAIS SA	ADP	Portugal
11	TERRAPRIMA - SERVICOS AMBIENTAIS SOCIEDADE UNIPESSOAL LDA	TP	Portugal
12	3MON, s. r. o.	3MON	Slovakia
13	CATALINK LIMITED	CTL	Cyprus
14	SYNTHESIS CENTER FOR RESEARCH AND EDUCATION LIMITED	SYNC	Cyprus
15	EXPERT SYSTEM SPA	EAI	Italy
16	ITTI SP ZOO	ITTI	Poland
17	Venaka Treleaf GbR	VTG	Germany
18	MASSIVE DYNAMIC SWEDEN AB	MDS	Sweden
19	FONDAZIONE CENTRO EURO-MEDITERRANEOSUI CAMBIAMENTI CLIMATICI	CMCC F	Italy
20	EXUS SOFTWARE MONOPROSOPI ETAIRIA PERIORISMENIS EVTHINIS	EXUS	Greece
21	RINIGARD DOO ZA USLUGE	RINI	Croatia
22	Micro Digital d.o.o.	MD	Croatia
23	POLITECHNIKA WARSZAWSKA	WUT	Poland
24	HOEGSKOLAN I BORAS	HB	Sweden
25	GEOPONIKO PANEPISTIMION ATHINON	AUA	Greece
26	ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS	CERTH	Greece
27	PANEPISTIMIO THESSALIAS	UTH	Greece

No	Partner Name	Short name	Country
28	ASSOCIACAO DO INSTITUTO SUPERIOR TECNICO PARA A INVESTIGACAO E DESENVOLVIMENTO	IST	Portugal
29	VELEUCILISTE VELIKA GORICA	UASVG	Croatia
30	USTAV INFORMATIKY, SLOVENSKA AKADEMIA VIED	UISAV	Slovakia
31	POMPIERS DE L'URGENCE INTERNATIONALE	PUI	France
32	THE MAIN SCHOOL OF FIRE SERVICE	SGPS	Poland
33	ASSET - Agenzia regionale Strategica per lo Sviluppo Ecosostenibile del Territorio	ASSET	Italy
34	LETS ITALIA srls	LETS	Italy
35	Parco Naturale Regionale di Tepilora	PNRT	Italy
36	FUNDATIA PENTRU SMURD	SMURD	Romania
37	Romanian Forestry Association - ASFOR	ASFOR	Romania
38	KENTRO MELETON ASFALEIAS	KEMEA	Greece
39	ELLINIKI OMADA DIASOSIS SOMATEIO	HRT	Greece
40	ARISTOTELIO PANEPISTIMIO THESSALONIKIS	AHEPA	Greece
41	Ospedale Israelitico	OIR	Italy
42	PERIFEREIA STEREAS ELLADAS	PSTE	Greece
43	HASICSKY ZACHRANNY SBOR MORAVSKOSLEZSKEHO KRAJE	FRB MSR	Czechia
44	Hrvatska vatrogasna zajednica	HVZ	Croatia
45	TECHNICKA UNIVERZITA VO ZVOLENE	TUZVO	Slovakia
46	Obcianske zdruzenie Plamen Badin	PLAMEN	Slovakia
47	Yayasan AMIKOM Yogyakarta	AMIKOM	Indonesia
48	COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION	CSIRO	Australia
50	FUNDACAO COORDENACAO DE PROJETOS PESQUISAS E ESTUDOS TECNOLOGICOS COPPETEC	COPPETEC	Brazil

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACRONYM	DESCRIPTION
BCE	Binary cross entropy
CNN	Convolutional Neural Network
FWI	Fire Weather Index
LSTM	Long short-term memory
TPH	Transformer Prediction Heads
YOLO	You only look once
ML	Machine Learning
WP	Workpackage
SAL	Storage Abstraction Layer
DL	Deep-learning
CNN	Convolutional neural network
JSON	JavaScript Object Notation

EXECUTIVE SUMMARY

The objective of the deliverable is to outline in detail the activities carried out in workpackage (WP5) of the SILVANUS project titled “Response coordination to contain the spread of wildfires”. The WP consists of consists of five (5) tasks, out of which the deliverable outlines the activities carried out in T5.1 to T5.3. The purpose of the deliverable is to report on the demonstration of SILVANUS big-data framework that has been implemented which will serve and act as the backbone for delivering data handling capacity for all the modules of the project to be developed and integrated. In this regard, the deliverable provides a detailed description of the different software components that have been developed which are aligned to the needs of the project as validated by the stakeholders and reported in D2.1.

In addition to the report on the demonstration of various software components implemented for data handling, the deliverable also reports on the scientific innovation of algorithmic development for the detection of fire and smoke components. The use of different machine learning approaches and deep-learning models have been elaborated in detail. Also, the deliverable reports on the machine learning lifecycle management methodology that has been adopted for enabling dynamic deployment of the trained models and to enable training of the deep-learning models as new data emerges.

Each scientific development reported in the deliverable also includes a validation framework that is used to evaluate the performance of the algorithms developed within SILVANUS.

1. Introduction

SILVANUS project development has been systematically categorised to address all the three phases of integrated wildfire management ranging from Phase A: Prevention and preparedness to Phase C: Restoration and adaptation. While the Phase A and C activities are being addressed in WP2, WP3 and WP6 and WP7 respectively, the focus on WP4 and WP5 have been placed on delivering effective and efficient technology solutions for the fire fighters in tackling the detection and response coordination to wildfire incidents. To this end, the activities of WP5 are aimed at building a big-data framework that will provide the necessary software services deployed in the relevant cloud environment for enabling key insights to be drawn from the different heterogenous data sources that have been deployed on the field.

The goal of Phase B activity on detection and response is to design and deliver an ICT platform that offers advanced capabilities for the first responders (fire fighters, medical response team, public administration authorities, civil protection agencies, forest management personnel, and other relevant stakeholders). The project development will demonstrate the ability of fire detection services that are being deployed in the forest and enable early-stage detection of fire through the concept extraction of smoke. Additionally, the project also has carried out research activity on the modelling of forest fire spread considering the impact of climate and weather services and to enable coordination of a response with the deployment of forward command centre to subdue and contain forest fire effectively and efficiently. The research and technology innovation in the project will include knowledge gathered from Earth Observation data sources, granular predictive models of weather and climate conditions, use of autonomous systems to obtain the insights on the spread of fire. The deliverable will summarise all the relevant activities carried out in WP5 addressing the challenges as outlined.

The structure of the deliverable is as follows. In Section 2, an overview of the SILVANUS Big-data framework has been presented, which has been introduced in D8.1, that was submitted in M12. The contents of the big-data framework section also include references to the activities carried out in WP4, with special demonstration of the Apache NiFi data ingestion platform and services. One of the core components of the big-data framework relates to the storage abstraction layer (SAL), which has been described in detail within Section 2. The scientific results on the development of data driven approaches for fire detection is reported in Section 3, which includes summarised two complementary approaches that has been adopted within SILVANUS. In Section 4, the data driven approach for modelling the fire spread has been presented in detail, whose performance has been compared against the gold standard of FLAMAP solution that has been adopted in the literature. Subsequently, in Section 5, the use of satellite data processing capabilities developed within SILVANUS has been reported. In Section 6, the data-driven approach for fire danger risk prediction has been presented. Section 6 describes the data set release on fire detection, followed by the integration services outlined for demonstration in Section 7. The integration of health sensors and knowledge components is also reported in Section 7. Section 8 presents the conclusion of the deliverable and outlines the future work to be carried out during the project lifecycle.

2. SILVANUS Big Data Framework

The Big Data framework is introduced in Deliverable 8.1 as a core component that encompasses a collection of components responsible for the ingestion, storage, transformation, and processing of data, collectively forming the foundation of SILVANUS, enabling the creation of various user products. The Big Data framework is a unified ensemble of components and APIs designed to empower an implementation in delivering the SILVANUS user products, with the flexibility to be deployed on various suitable platforms. This deliverable focuses on providing implementation details regarding the Storage Abstraction Layer (SAL) and introducing the ML lifecycle management service within the Big Data framework. The architecture diagram, depicted in Figure 1, illustrates the framework of a Big Data system. It showcases the interconnectedness between the various components, the framework itself, and the user products spanning different work packages (WPs)

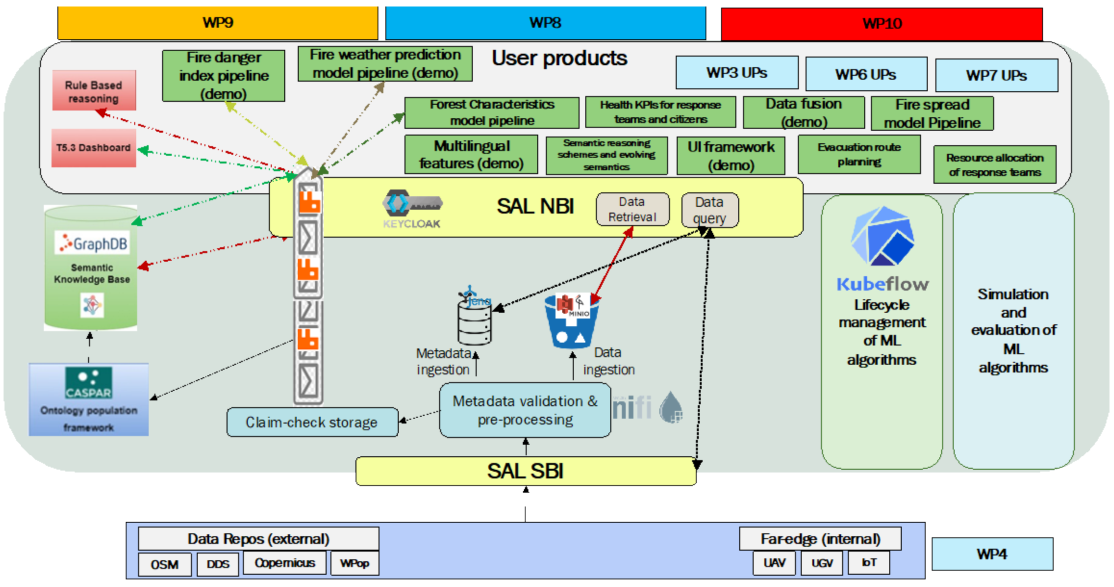

WP9
WP8
WP10
User products
Rule Based reasoning
Fire danger index pipeline (demo)
Fire weather prediction model pipeline (demo)
WP3 UPs
WP6 UPs
WP7 UPs
Forest Characteristics model pipeline
Health KPIs for response teams and citizens
Data fusion (demo)
Fire spread model Pipeline
T5.3 Dashboard
Multilingual features (demo)
Semantic reasoning schemes and evolving semantics
UI framework (demo)
Evacuation route planning
Resource allocation of response teams
KEYCLOAK
SAL NBI
Data Retrieval
Data query
GraphDB
Semantic Knowledge Base
Kubeflow
Lifecycle management of ML algorithms
Simulation and evaluation of ML algorithms
jena
MINIO
Metadata ingestion
Data ingestion
CASPAR
Ontology population framework
Claim-check storage
Metadata validation & pre-processing
nifi
SAL SBI
Data Repos (external)
OSM
DDS
Copernicus
WPop
Far-edge (internal)
UAV
UGV
IoT
WP4

Figure 1: SILVANUS Big-data framework

2.1. Data sources integration

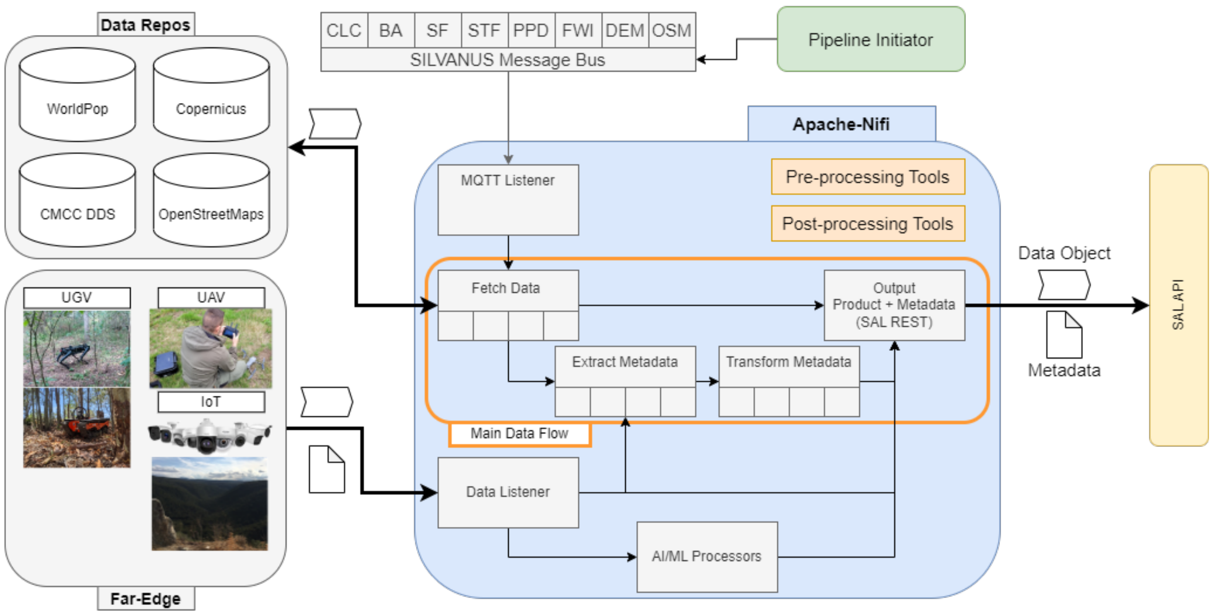


Figure 2 Data Ingestion Pipeline architecture

Within the Big Data Framework data sources are ingested into the storage layer via the Data Ingestion Pipeline, a tightly integrated component for managing the ingestion, annotation, and pre-processing of data from a variety of providers**.** A brief summary of this component, available datasets, and related tooling is described below, with further detail available within the demonstration report D4.1.

Figure 2 Data Ingestion Pipeline architecture depicts the flow of data objects through the Data Ingestion Pipeline, as data is ingested and moves through the corresponding Nifi pipeline. The format, size, ingestion frequency and structure of this data can vary depending on the provider. The specific implementation of each pipeline can vary; however, the flow and high-level operations remain consistent. Metadata extraction of data objects at ingestion time is one of these key processes that takes place at this stage, as this supports the data annotation and later retrieval of data objects from the Big-data framework.

The Data Ingestion Pipeline directly communicates via a REST API with the SILVANUS Storage Abstraction Layer (SAL), an abstraction component of the Big-data Framework and underlying object storage solution. Exposed by the SAL is a single abstract POST HTTP endpoint that is responsible for ingesting all data sources from the Data Ingestion Pipeline. The output HTTP message from the Ingestion Pipeline follows a consistent format containing:

- **HTTP Body – Object data**: A single data object encoded as the HTTP message body
- **Headers – Object metadata, Ingestion Pipeline Attributes:** Extracted object metadata is attached to the same HTTP request, following the SILVANUS Object Metadata specification and format. Metadata generated by the specific ingestion pipeline is also attached, these are the Apache Nifi flow file attributes generated from the pipeline processors, including the initiating queue, dataset ingestion parameters etc. This allows further enrichment of metadata provided alongside data objects to be leveraged in the SAL.

The ingestion of individual datasets is managed by a series of corresponding data pipelines. As presented in Deliverable D8.1, we implement a communication solution based on asynchronies message queue - the SILVANUS Message Bus. As part of the MVP and demonstration of the Big Data Framework, we implement a series of initial ingestion queues which map directly to a corresponding data pipeline.

As each ingestion queue corresponds to an individual data product, each pipeline implements a corresponding MQTT listener, which awaits new messages published and in turn initiates the ingestion of a data product. Messages are published two an ingestion queue via primary mechanisms.

1. **Static Product Ingestion:** This category defines datasets which have statically defined parameters for ingestion. Parameters for ingestion refer to attributes such as the geospatial area, ingestion frequency, data format or temporal range. For some data sources we need to only consider a statically defined set of parameters (and therefore ingestion messages) which can be automated and ingested based on these requirements.
2. **Custom Product ingestion:** Custom dataset parameters are a requirement of some datasets, specifically the user products that leverage these datasets within AI/ML training / inference and visualization dashboards. An example of this requirement is presented in Section 0.

In order to ingest these custom datasets a user product can publish a message to the relevant message queue, containing the specific ingestion and pre-processing parameters that are required. In the following table we present the data sources ingested via the Data Ingestion Pipeline and available for use within the SAL for the SILVANUS MVP demonstrations. It reflects the current status and pre-processing capabilities within the Data Ingestion Pipeline for demonstration purposes and can be expanded as new data source providers are integrated and additional pre/post-processing techniques become available.

Table 1 - Status and Pre-processing Capabilities Within the Data Ingestion Pipeline

Queue Name	Dataset	Parameters	Output
ingest.dem	Digital Elevation Model	pilot: [*pilot_string] type: [dem, asp, slp]	Tiff
ingest.osm	OpenStreetMaps Road / Rail	Pilot: [*pilot_string] type: [road, rail] resolution: [*Int] bbox: [*GeoJSON_coords]	GeoJSON, NetCDF
ingest.sentinel-ndvi	Sentinel-2/3 + NDVI	resolution: [*Int] footprint: [*GeoJSON_bbox] cloud: [*Int]	SAFE, Tiff
ingest.lst	Land Surface Temperature	type: - [H, DC, TCI]	NetCDF
ingest.ba	Burned Area	version: - [V1, V3]	NetCDF
ingest.pop	Population Density	pop_year: - [*YYYY] country: - [*ISO 3166 code]	CSV, Tiff
ingest.stf	Short-term Forecast	prod_date: - [*YYYY_MM_DD] b_north: - [*float]	NetCDF

		b_south: - [*float] b_west: - [*float] b_east: -[*float]	
Ingest.clc	Corine Landcover	year: - [*YYYY] b_north: - [*float] b_south: - [*float] b_west: - [*float] b_east: -[*float]	Tiff

The implemented architecture supports the SILVANUS Platform with two key properties:

- **Loose coupling of individual components** – supporting cases where a connection link between two components may be lost temporarily.
- **Common communication framework** – supporting key functionalities such as request of custom datasets from user products and notification of dataset ingestion status and availability to relevant user products.

2.2. Storage Abstraction Layer

The Storage Abstraction Layer (SAL) serves as an intermediary between data sources, user products, and the object store within the SILVANUS system. Its primary function is to abstract the object store, offering two key advantages. Firstly, it enables flexibility in managing data at rest, allowing for efficient data management practices. Secondly, it decouples data from user products in a multi-source, multi-client environment, providing support for security, policy, privacy, and business constraints. By utilizing the SAL, the SILVANUS system achieves enhanced control and adaptability in handling data across various components.

2.2.1. Object store

The object store serves as the central repository within SILVANUS for storing both raw and processed data. Given that a significant portion of the ingested raw data in SILVANUS is unstructured, it is more efficient to store it in a unified object store rather than employing multiple databases for implementing the object store in SILVANUS, the MinIO object store is utilized and managed through the standard S3 storage API. This combination ensures seamless compatibility and efficient data management within the SILVANUS ecosystem. In the SILVANUS platform, the Apache NiFi PutS3Object processor is used to store the data in the MinIO object store.

MinIO is an open-source object storage system that is designed to be simple, scalable, and cloud-native. It allows you to store and retrieve large amounts of unstructured data, such as documents, images, videos, and other types of files. MinIO is built on the concept of object storage, where data is stored as objects rather than in a hierarchical file structure. Each object is assigned a unique identifier and is stored with its associated metadata. This approach allows for efficient and flexible storage of data, as objects can be accessed and manipulated independently. One of the key features of MinIO is its high scalability. It is designed to scale horizontally by distributing data across multiple servers, allowing you to expand storage capacity as your data grows.

2.2.2. Data and metadata ingestion

The interactions among the SAL for the three ingestion methods - external, internal, and user products - are depicted in Figure 3. A distinct SAL interface is required for each ingestion method to facilitate communication. During this process, the date object and its metadata are coupled and sent to the SAL by the Data Ingestion Pipeline (DIP) over endpoint. The SAL implementation then processes the input based on its origin, validates the metadata, and confirms that there are no data duplicates. The data objects are stored and/or forwarded to the user products via the message bus, and a metadata entry is added to the metadata index.

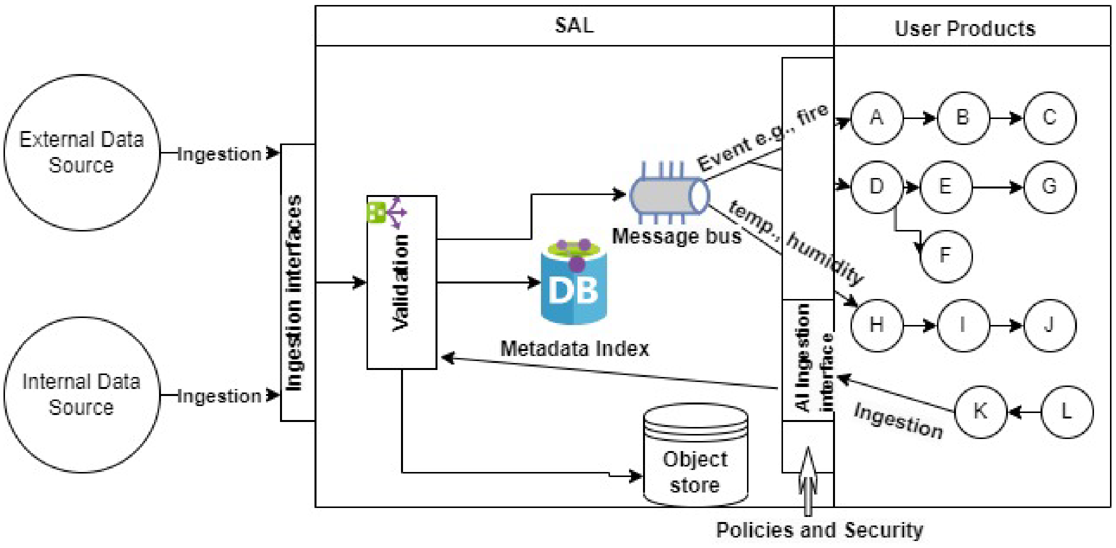


Figure 3: The interaction between SAL and other components in SILVANUS platform

Figure 4 illustrates the implementation of the validation steps using Apache NiFi. The validation process focuses on the mandatory fields specified in Table 7 of Deliverable 8.1.

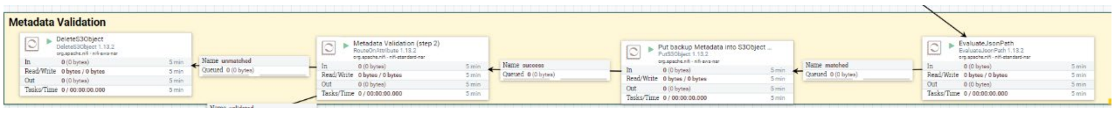


Figure 4: Metadata validation

The SILVANUS SAL metadata index relies on the Knowledge Graph technology, which is recommended to be implemented without blank nodes and duplicates to optimize search efficiency. To achieve this, three duplication check steps are employed. The first step involves a data duplication check, where the unique ID of the object data provided by the data source is utilized. The second step utilizes the metadata Format field, which comprises subfields such as type, resolution, and event. Lastly, the Spatial field describes the spatial characteristics of the data object, including coordination and pilot. These duplication check steps are executed using NiFi processors and JenaDB. Figure 5 shows the workflow of data and metadata duplication checks. Figure 6 showcases the implementation of the Data and Metadata duplication check process using Apache NiFi.

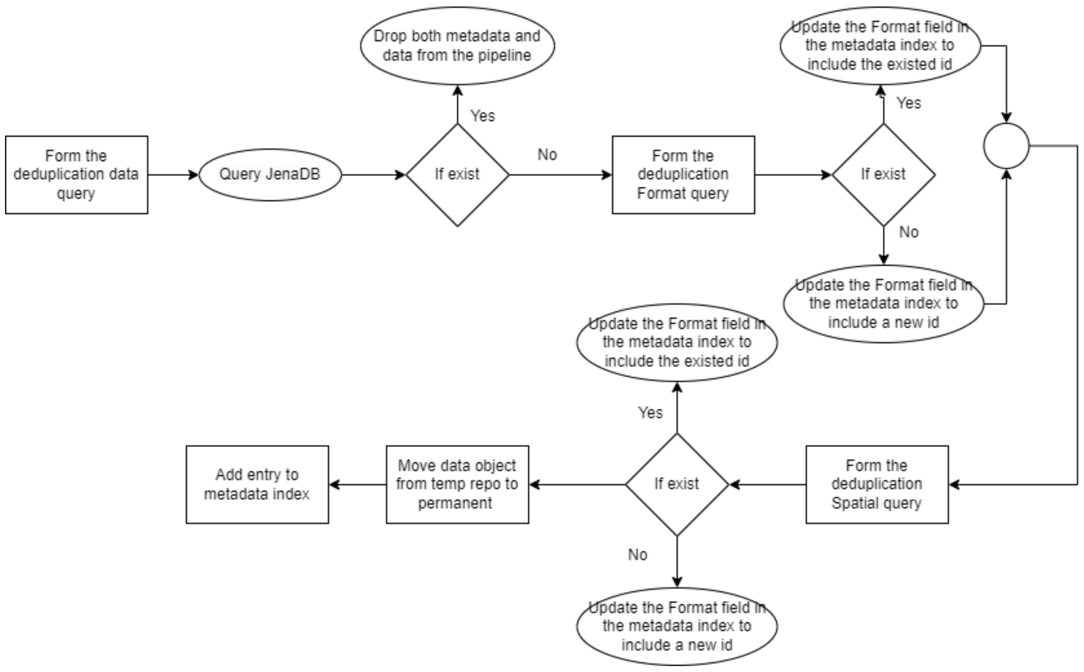

Form the deduplication data query
Query JenaDB
If exist
Yes
Drop both metadata and data from the pipeline
No
Form the deduplication Format query
If exist
Yes
Update the Format field in the metadata index to include the existed id
No
Update the Format field in the metadata index to include a new id
Form the deduplication Spatial query
If exist
Yes
Update the Format field in the metadata index to include the existed id
No
Update the Format field in the metadata index to include a new id
Move data object from temp repo to permanent
Add entry to metadata index

Figure 5: The flowchart of data and metadata deduplication

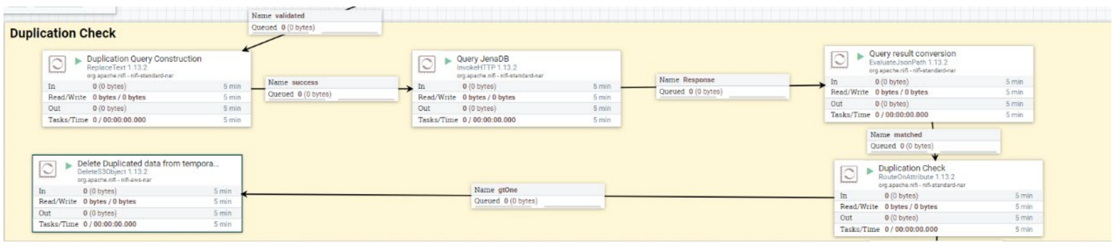

Duplication Check
Name validated
Queued 0 (0 bytes)
Duplication Query Construction
ReplaceText 1.13.2
org.apache.nifi - nifi-standard-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min
Name success
Queued 0 (0 bytes)
Query JenaDB
InvokeHTTP 1.13.2
org.apache.nifi - nifi-standard-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min
Name Response
Queued 0 (0 bytes)
Query result conversion
EvaluateJsonPath 1.13.2
org.apache.nifi - nifi-standard-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min
Name matched
Queued 0 (0 bytes)
Duplication Check
RouteOnAttribute 1.13.2
org.apache.nifi - nifi-standard-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min
Name gtOne
Queued 0 (0 bytes)
Delete Duplicated data from tempora...
DeleteS3Object 1.13.2
org.apache.nifi - nifi-aws-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min

Figure 6: Data and Metadata duplication check

Data objects in SILVANUS can sometimes be quite large, up to 5GB. To optimize performance and minimize memory usage, these objects are primarily stored on disk in the temporary directory and are only loaded into memory as needed. Conversely, since metadata messages are lightweight, they are kept in memory. Data objects are only stored when they pass metadata and data duplication checks. Once stored, the data objects are saved in the Object storage using the Apache NiFi PutS3Object processor. If user products do not directly request the data objects, they are deleted from the temporary directory. However, if the data objects are passed to the message bus, they remain in the temporary directory until their expiration date.

Once the data has been stored in the object storage, the results of the "Data and metadata duplication check" process are transformed into Triples format before being included as an entry in the knowledge-graph-based metadata index.

2.2.3. Data retrieval

Data retrieval is a fundamental capability facilitated by the SAL, which considers multiple factors for efficient data access. These factors encompass the real-time or near real-time velocity at which data is generated, processed, and analyzed, the size of the data, the effectiveness of consuming large datasets within user product pipelines, as well as the ability to efficiently query and retrieve both internal and external datasets. The SAL provides a range of interfaces to support the SILVANUS services. These interfaces enable the services to efficiently query, retrieve, and store objects and datasets, including their associated metadata. The subsequent subsections outline the specifics of three methodologies employed within the SILVANUS platform for data retrieval purposes.

2.2.3.1. Message queue

The purpose of this interface is to deliver event-based messages to consumers on the platform. These messages can originate from various sources such as IoT far-edge sensors or user products running on the SILVANUS cloud. When the message creator or publisher is the data source itself, the message consumer directly receives the data. However, there is a limitation in terms of the size of the published data object. It should be smaller than the specified threshold, known as the Claim Check Pattern threshold (ccp_threshold). If the data object exceeds this threshold, the appropriate solution is to utilize the Claim Check Pattern for delivering messages to consumers.

RabbitMQ plays a crucial role in providing the message queue service within the SILVANUS SAL. It is an open-source message broker software designed to facilitate the distribution and processing of messages between different applications or components. RabbitMQ implements the Advanced Message Queuing Protocol (AMQP), a widely adopted standard for messaging middleware.

In order to maintain access control to the published data, a dedicated virtual host (vhost) is assigned to each data source, ensuring separate environments with distinct read and write permissions. Data consumers are required to obtain approval from the data source before accessing a specific queue, and they are granted read-only permission to retrieve the data.

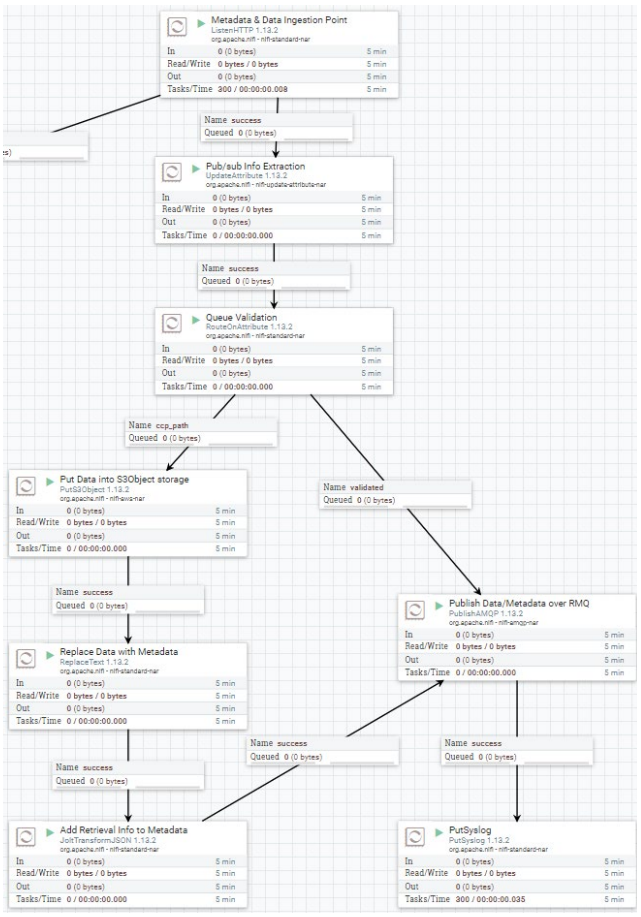
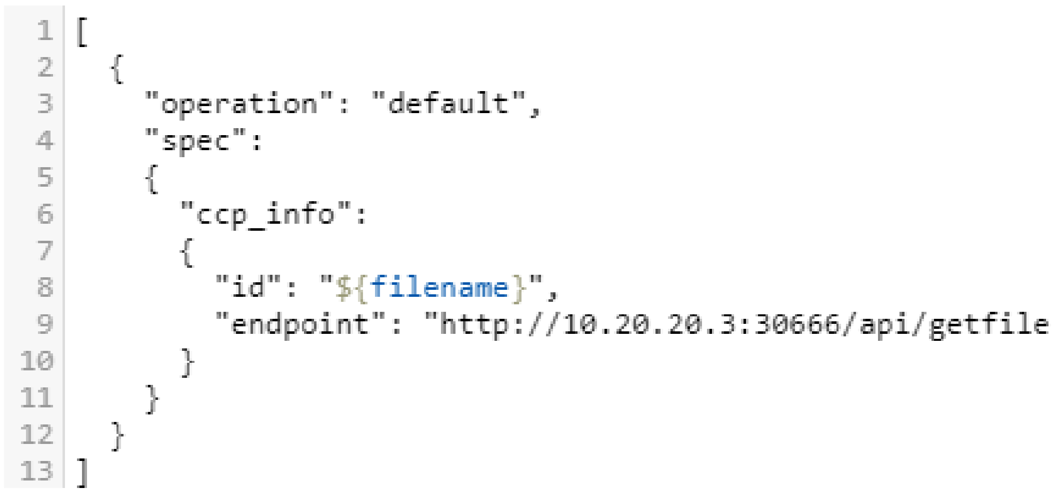

Metadata & Data Ingestion Point
ListenHTTP 1.13.2
org.apache.nifi - nifi-standard-nar
Tasks/Time 300 / 00:00:00.008
Name success
Pub/sub Info Extraction
UpdateAttribute 1.13.2
org.apache.nifi - nifi-update-attribute-nar
Name success
Queue Validation
RouteOnAttribute 1.13.2
org.apache.nifi - nifi-standard-nar
Name ccp_path
Name validated
Put Data into S3Object storage
PutS3Object 1.13.2
org.apache.nifi - nifi-aws-nar
Name success
Replace Data with Metadata
ReplaceText 1.13.2
org.apache.nifi - nifi-standard-nar
Name success
Add Retrieval Info to Metadata
JoltTransformJSON 1.13.2
org.apache.nifi - nifi-standard-nar
Name success
Publish Data/Metadata over RMQ
PublishAMQP 1.13.2
org.apache.nifi - nifi-amqp-nar
Name success
PutSyslog
PutSyslog 1.13.2
org.apache.nifi - nifi-standard-nar
Tasks/Time 300 / 00:00:00.035
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Queued 0 (0 bytes)

Figure 7: Apache NiFi implementation for the SAL message queue and CCP

2.2.3.2. Claim Check Pattern

Due to the substantial size of some of the datasets handled by the SILVANUS system, which can reach up to approximately 5GB, it is not advisable to directly pass such large files as part of the event messages between services. To address this, the Claim Check design pattern is employed. The SILVANUS SAL makes decisions based on this pattern's ccp_threshold and the pub/sub queue policy. It saves the data objects in a temporary data repository while updating the metadata with relevant details for retrieving the stored data objects using the "Add retrieval Info to Metadata" processor depicted in Figure 7. As shown in Figure 8, the updated field format enables consumers to retrieve the data utilizing the CCP solution efficiently.



```
[
  {
    "operation": "default",
    "spec":
    {
      "ccp_info":
      {
        "id": "${filename}",
        "endpoint": "http://10.20.20.3:30666/api/getfile
      }
    }
  }
]
```

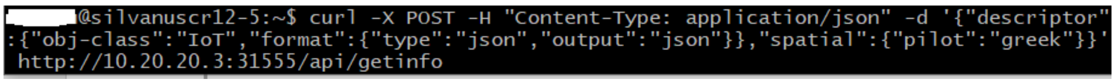
Figure 8: The format of added filed to the metadata to enable CCP using NiFi JoltTransformJSON Processor

2.2.3.3. Metadata index query

The SILVANUS SAL offers a metadata interface (i.e., the query interface in Figure 9) that enables SILVANUS services to search the Metadata Index and locate the specific data objects they need to fulfil their respective functions. This interface also provides the capability to obtain additional metadata related to the requested object(s). When querying the Metadata Index (Steps 1-4 in Figure 9), the response comprises a list of metadata entries that satisfy the specified query constraints. Among the available fields within the metadata, the 'id' field is particularly relevant, as it can be utilized to retrieve the corresponding data object through the data retrieval interface (Steps 5-8 in Figure 9).

Figure 9: The workflow for the data retrieval solution

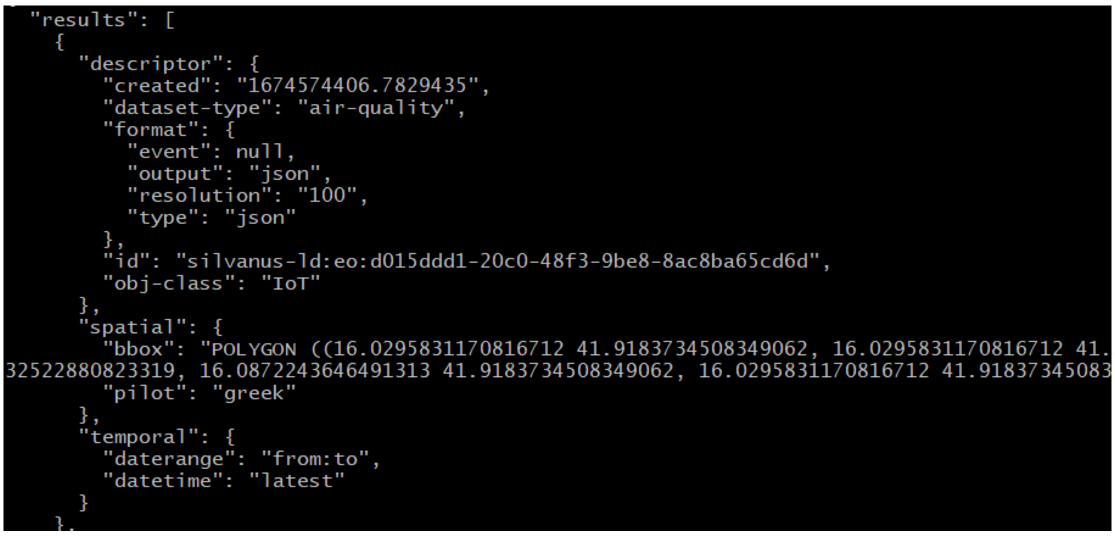
In Figure 10, an example of the query format for a user product is displayed, representing step 1 as depicted in Figure 9. It is noteworthy that the query format shares similarities with the metadata input, both in terms of the format itself and the fields utilized within the query. This alignment in format and fields allows for consistency and ease of use between the user product query and the associated metadata input.



```
[redacted]@silvanuscr12-5:~$ curl -X POST -H "Content-Type: application/json" -d '{"descriptor"
:{"obj-class":"IoT","format":{"type":"json","output":"json"}},"spatial":{"pilot":"greek"}}'
 http://10.20.20.3:31555/api/getinfo
```

Figure 10: Query format

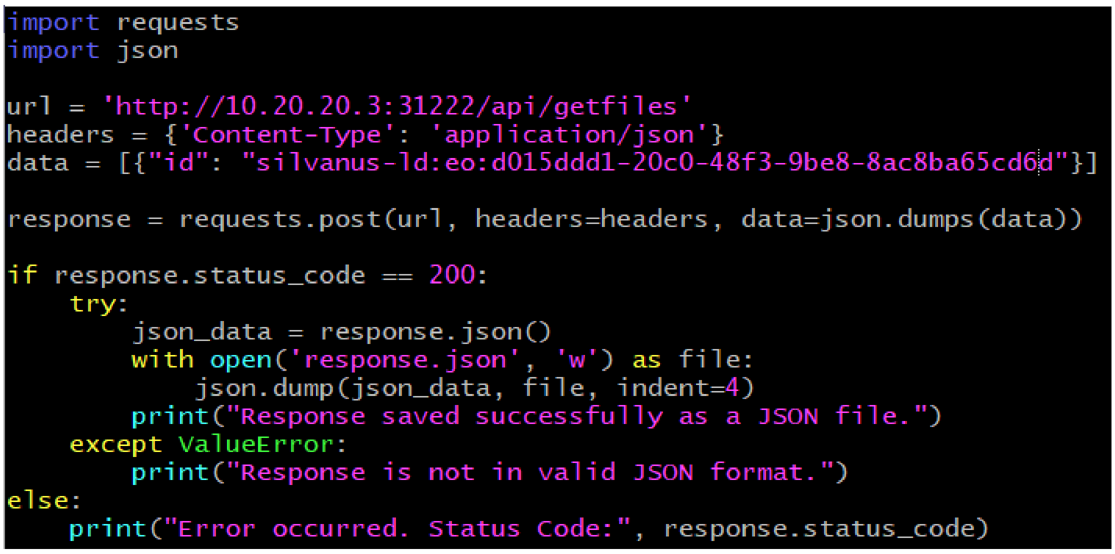
Figure 11 displays a portion of the response corresponding to the query depicted in Figure 10. It provides an excerpt of the response that was generated as a result of executing the query, presenting relevant information or data related to the query criteria.



```
  "results": [
    {
      "descriptor": {
        "created": "1674574406.7829435",
        "dataset-type": "air-quality",
        "format": {
          "event": null,
          "output": "json",
          "resolution": "100",
          "type": "json"
        },
        "id": "silvanus-ld:eo:d015ddd1-20c0-48f3-9be8-8ac8ba65cd6d",
        "obj-class": "IoT"
      },
      "spatial": {
        "bbox": "POLYGON ((16.0295831170816712 41.9183734508349062, 16.0295831170816712 41.
32522880823319, 16.0872243646491313 41.9183734508349062, 16.0295831170816712 41.91837345083
        "pilot": "greek"
      },
      "temporal": {
        "daterange": "from:to",
        "datetime": "latest"
      }
    },
```

Figure 11: An example of query results

Figure 12 demonstrates a sample Python code that downloads a file with the id 'silvanus-ld:eo:d015ddd1-20c0-48f3-9be8-8ac8ba65cd6d' from SAL.



```python
import requests
import json

url = 'http://10.20.20.3:31222/api/getfiles'
headers = {'Content-Type': 'application/json'}
data = [{"id": "silvanus-ld:eo:d015ddd1-20c0-48f3-9be8-8ac8ba65cd6d"}]

response = requests.post(url, headers=headers, data=json.dumps(data))

if response.status_code == 200:
    try:
        json_data = response.json()
        with open('response.json', 'w') as file:
            json.dump(json_data, file, indent=4)
        print("Response saved successfully as a JSON file.")
    except ValueError:
        print("Response is not in valid JSON format.")
else:
    print("Error occurred. Status Code:", response.status_code)
```

Figure 12: File download request example

2.3. ML lifecycle management

2.3.1. Introduction: what ML lifecycle

The machine learning lifecycle management supports the creation and the execution of machine learning experiments, from the data acquisition to the model serving. The entire cycle is composed by different steps, which can be summarized as follows:

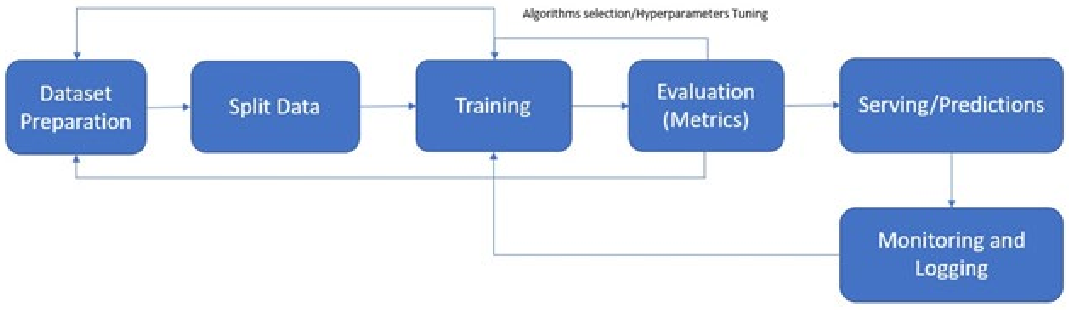


Figure 13 – ML lifecycle

1. **Data preparation**: it's the first and most delicate phase, because it needs the knowledge of the data and features, and needs a deep study of the data composition; depending on this awareness, data are prepared through preprocessing activities, in order to make them compliant with the problem that must be solved with, and with machine learning algorithms to experiment.
2. **Split data**: data is split into different sets for being processed by machine learning algorithms.
3. **Training and validation model**: it can be considered as a mini-cycle of subtasks, in which many experiments can be performed with different algorithms and different parameters configurations, each of them is evaluated through performance metrics, such as, precision, recall, f-score, and so on.
4. **Serving**: the trained model can be exposed for processing elaboration requests about predictions.
5. **Monitoring and logging**: the final step consists of monitoring the executions of the served model.

Many tools and frameworks are available to address the purposes of this fundamental part of the machine learning processes. A deep scouting of the principals has been provided, in order to select the best solution for supporting the machine learning activities within the Silvanus project.

2.3.2. Principal ML lifecycle management frameworks

Although there is a very large number of frameworks for managing ML lifecycle, the scouting has been concentrated on the most popular and famous, in order to select the best solution for the SILVANUS project. The following list is the result of this activity, where, for each framework, advantages and disadvantages are summarized; for deeper descriptions the links to the official sites are provided.

Neptune.ai (https://neptune.ai/)

Advantages:

- Works with any ML framework
- Keep track of any metadata type of a single experiment (from hyperparameters, to plots, to source code and more)
- Distributed computing (log from multiple machines to the same run)
- Store and download models
- Keep track of dataset versions

Disadvantages:

- It can't serve the model
- It’s not free. It can be pricey (150 for teams, 600 for organizations)

Amazon Sagemaker (https://aws.amazon.com/pm/sagemaker/)

Advantages:

- SageMaker pipelines look almost identical to Kubeflow’s but it looks like their definitions require lots more detail and do little to simplify deployment.

Disadvantages:

- The main reason not to use it, however, is because it does not allow portability, while Kubeflow allows the user to keep the entire application portable between cloud providers, as it can run anywhere that Kubernetes is supported

Prefect.io (https://www.prefect.io/)

Advantages:

- Manages workflows as flows of execution.
- Provides Open-Source for personal usage, priced for organization/cloud.
- Provides PY SDK, cloud deploying, UI.
- Highly used for serving.
- Can be free/open source for personal usages.
- Widely corporate usages and sponsoring.

Disadvantages:

- Priced for organizations.

Google Vertex AI (https://cloud.google.com/vertex-ai/)

Advantages:

- Uses Kubeflow’s software under the hood but with the infrastructure managed by Google
- Works with pipelines
- Provides serving

- Probably cheaper as you only pay for the computing power as you're using it
- Allows to avoid the negatives of Kubeflow while keeping the possibility of migrating to another cloud

Disadvantages:

- It's not free. It can be pricey

Kubeflow (https://www.kubeflow.org/)

Advantages:

- Every step of the ML lifecycle can be orchestrated with Kubeflow Pipelines, which are controllable from a simple UI
- There are integrated notebook servers for quick experimentation and easy access to the cluster's resources
- It's a flexible and extensible framework due to it relying on Kubernetes to manage all code execution
- Possibility to deploy a cluster autoscaler, enabling the cluster to scale to heavier operations with no added complexity
- Each pipeline step is isolated in its own container, which improves the developer experience as opposed to a monolithic solution where all the phased are bundled together, like airflow
- Although even in Airflow there's the docker operator which allows you to execute a command inside a Docker container
- Another benefit over Airflow however is the responsiveness of the UI with status changes update in real time.
- The reusability of components is also a big benefit. There are many contributions in this area by the community.
- Free, voluntary contributes can be sent.

Disadvantages:

- As a negative, in order to exploit the easy access to the full compute power of the cluster from the notebooks, every script needs to be converted manually into a Kubeflow component to get it running in a pipeline. This means writing a Dockerfile, building a container, creating a component spec, and adding a custom container spec to make sure it runs on the right nodes and requests enough memory.

Conclusions

The winning option seems to be Kubeflow because it scales better and offers a better developing experience. It's simpler than SageMaker and more portable, and also it has access to SageMaker jobs via Kubeflow components anyway. An option to use it through Google's Vertex AI could be considered as well as it offers possibly better management of the Kubeflow infrastructure. A good solution could be also Prefect, since it has both Python sdk and UI, but the license is not free for cloud solutions.

2.3.3. Kubeflow cloud environment

The Kubeflow solution has been deployed with the support of the official cloud environment provider partner (INTRASOFT). It is on top of Kubernetes deploy and exploits it for creating PODs for each component or virtual server (Notebooks, Pipelines, etc…). It supports multi-account management, by assigning to each account a dedicated namespace in which the analyst can operate. For SILVANUS project the namespace "lifecycleml" has been created and provided; the namespace manager can grant access to other accounts.

The home dashboard is accessible to the following link:

https://kubeflow.platform.silvanus-project.eu/

And it appears as follows:

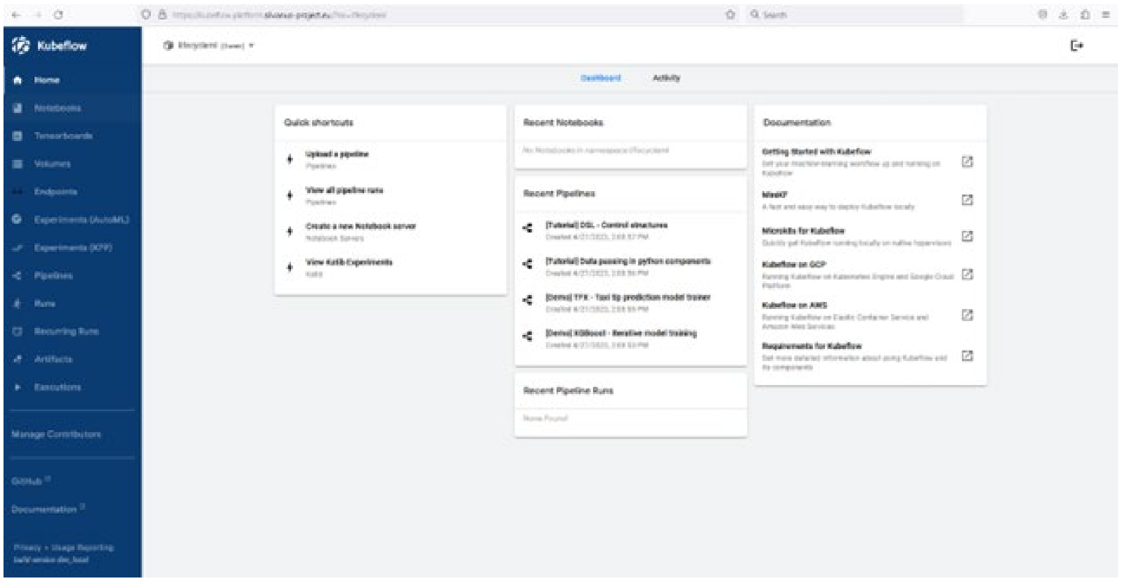
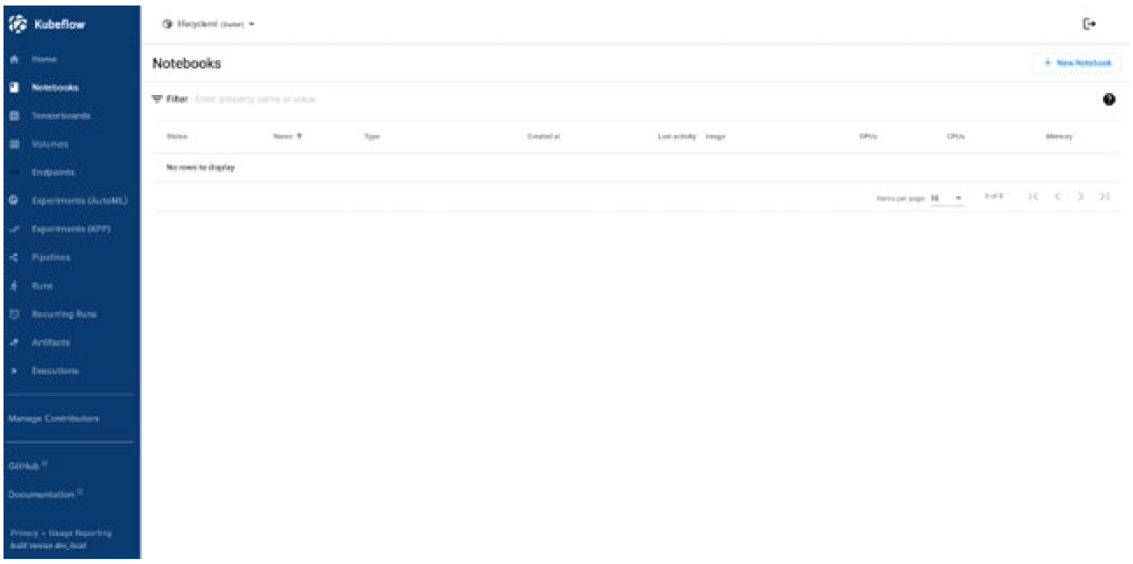


Figure 14 – Kubeflow Dashboard

On the left bar there are listed all deployed components. This distribution appears as complete for all of the official components that Kubeflow environment supports.

Most important components, for the purposes of SILVANUS project, are "Notebooks", "Pipelines" and "KServe" (that shows its features through "Endpoints" section).

Notebooks component appears as follows:

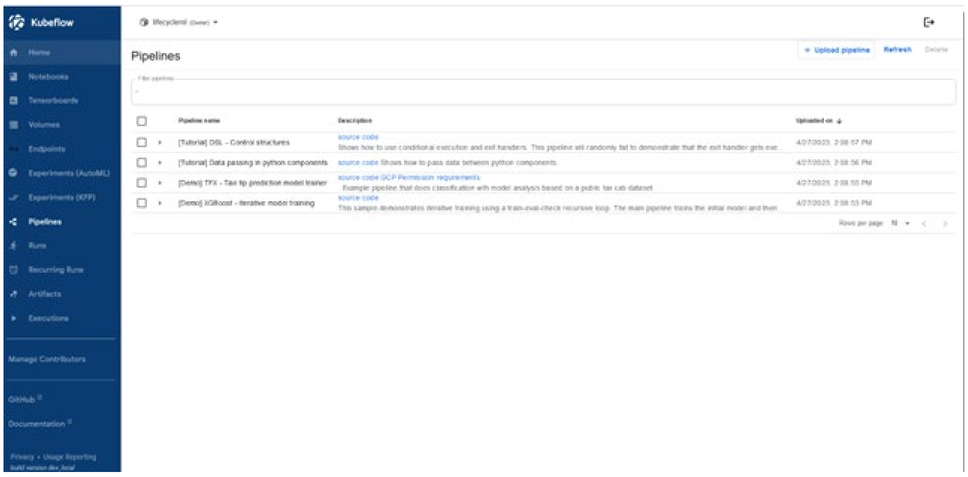


Status	Name	Type	Created at	Last activity	Image	GPUs	CPUs	Memory
No rows to display								

Figure 15 – Kubeflow Notebooks

When creating a new Notebook, a virtual environment has been set. This results in a new notebook server, of different types (Jupyter, VS code, RStudio, etc…), each of the ones are created from a pre-existing Docker image. Other important settings to select are the HW features (cores, memory, etc…) and the volume to be associated with the Docker container that starts the notebook server.

The Pipelines section allows to upload the ML pipelines developed through the SDks that Kubeflow provides (i.e. Python SDK): the developers implement the steps of the pipelines through the SDK and upload them into the environment. Each pipeline step is a Docker container.



Pipeline name	Description	Uploaded on
[Tutorial] DSL - Control structures	source code Shows how to use conditional execution and exit handlers. This pipeline will randomly fail to demonstrate that the exit handler gets exe...	4/27/2023, 2:58:57 PM
[Tutorial] Data passing in python components	source code Shows how to pass data between python components.	4/27/2023, 2:58:56 PM
[Demo] TFX - Taxi tip prediction model trainer	source code GCP Permission requirements Example pipeline that does classification with model analysis based on a public tax cab dataset.	4/27/2023, 2:58:55 PM
[Demo] XGBoost - Iterative model training	source code This sample demonstrates iterative training using a train-eval-check recursive loop. The main pipeline trains the initial model and then...	4/27/2023, 2:58:53 PM

Figure 16 – Kubeflow Pipelines

The usage of Pipelines is very important for playing experiments and tuning hyperparameters of ML algorithms, and for combining different components (steps) in order to train as the best model as possible.

Running experiments are listed in section below:

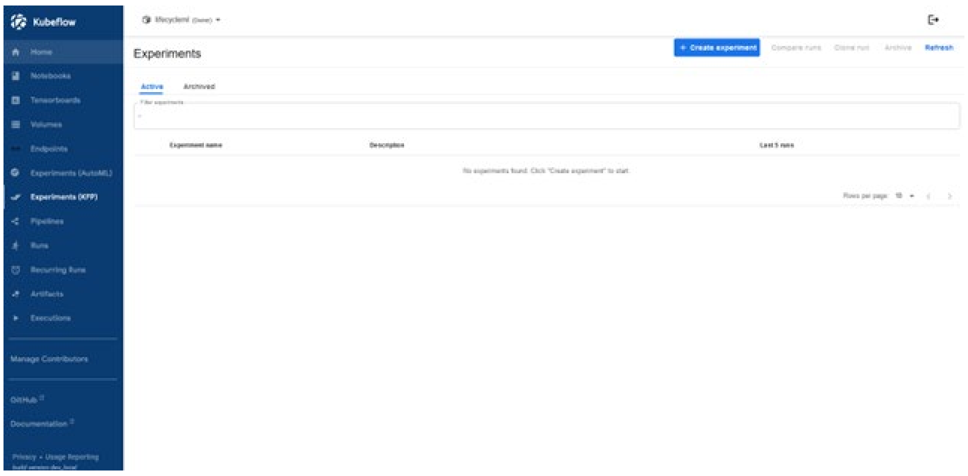


Figure 17 – Kubeflow Experiments

Through the KServe component it is possible to publish the trained model as endpoint, in order to exploit it for predictions.

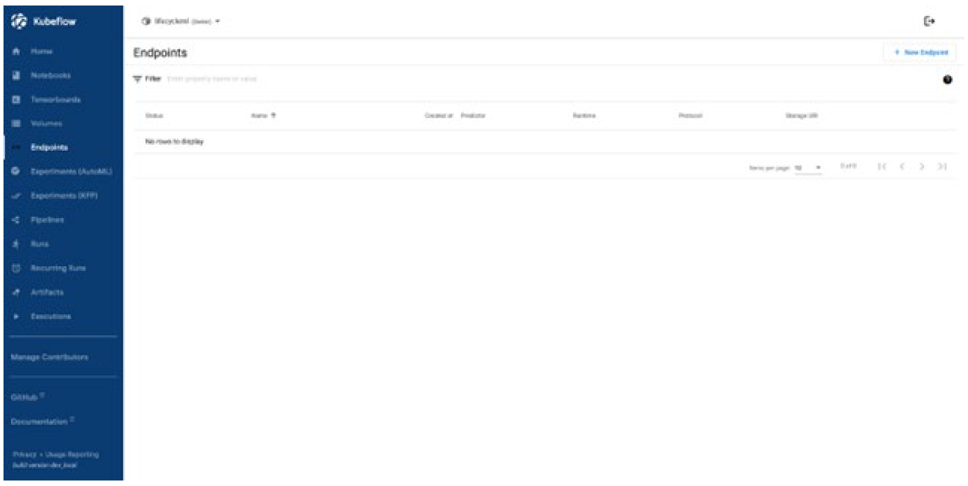


Figure 18 – Kubeflow Endpoints

The described environment shows the current state of the readiness of cloud deploying of the ML lifecycle.

2.4. System cloud-native readiness

The SAL has been successfully deployed and seamlessly integrated into the SILVANUS cloud platform. All its components are designed for cloud-native environments and can be accessed through IP-cluster or NodePort Kubernetes services. Below is a list of the SAL services along with their corresponding SILVANUS cloud endpoints.

Service name	Endpoint
SAL data and metadata ingestion pipeline	http://AnyHostIP:30516/metadata/ingest
SAL message queue service	(RabbitMQ) AnyHostIP:30672
Claim Check Pattern retrieval agent	http://AnyHostIP:30666/api/getfile
Metadata schema	http://AnyHostIP:30288/schema.json
Query interface	http://AnyHostIP:31555/api/getinfo
Retrieval interface	http://AnyHostIP:31222/api/getfiles

3. Data-driven approaches for fire detection

For the purposes of SILVANUS, ATOS and CTL have developed fire and smoke detection algorithms to run on high and low edge devices, respectively. The primary purpose of these algorithms was to detect fire near or on the edge, but a version of them will also be deployed in the SILVANUS framework for checking other incoming imagery data. Specifically, images arriving from UGVs and UAVs to NiFi, will pass through the additional step of the AI/ML Processors to check if they contain fire/smoke. The need for this additional step, is to equip the framework with another powerful fire/smoke detection tool, for cases it cannot happen on the edge, and avoid missing any fire events.

While both ATOS and CTL, have the same end goal (detection of fire/smoke), different, but complementary, approaches have been followed because of slightly different needs and requirements. Nevertheless, using the two approaches makes the detection tool more powerful, as they complement each other. In the following sections, the developed approaches will be presented along with example results.

3.1. ML Approaches

3.1.1. Fire and smoke detection using Deep-learning

Atos has built a detector of fire and smoke that is able to detect these elements using real color images as input. This detector can be run in “near” real time.

Atos approach has been based on the use of Yolov5[1] as base detector. This approach has been afterwards extended to other Yolo related products like TPH-Yolov5[2] and Yolov8[3]. The base idea has been to use the capabilities of Yolo architecture to have a detector able to detect fire and smoke in near real time footage.

The initial approach has been creating a tagging our own data set for fire and smoke, for this task we have gathered over 30000 images of different sources, and we have annotated the fire and the smoke in a semi-automatic way (see Figure 1 for an example). After this annotation task, we trained the Yolov5l (“large model”) model with these images using docker and Kubernetes with our cluster of machines.

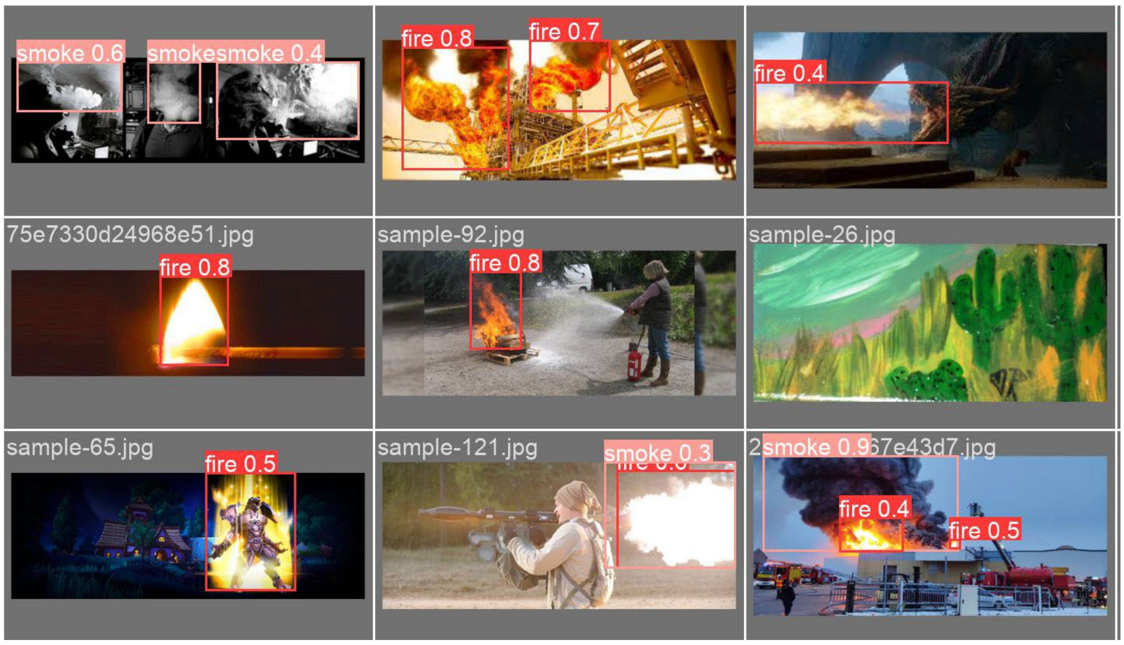


Figure 19 - Fire and smoke annotated data (sample)

The result of our training is a state-of-the-art trainer able to detect fire and smoke in near real time footage, as demonstrated in Athens General Assembly (see Figure 2 for an image of a demonstration video). Resulting model and program has been dockerized for its posterior use.

[1] https://github.com/ultralytics/yolov5

[2] https://github.com/cv516Buaa/tph-yolov5

[3] https://github.com/ultralytics/ultralytics

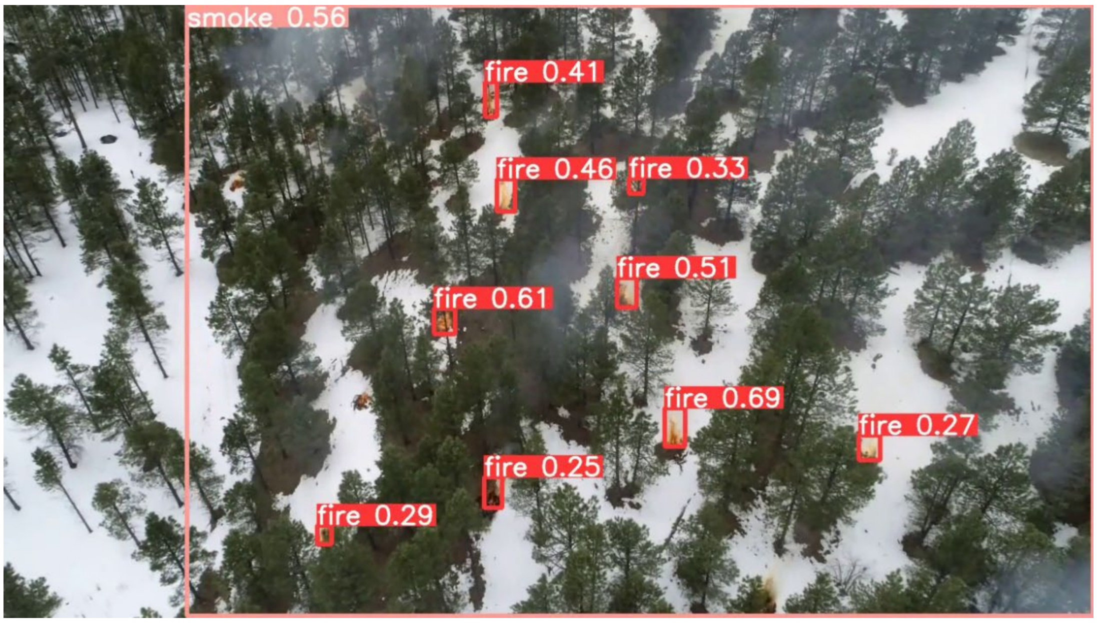


Figure 20 – Fire and smoke detection using real time video (video extract)

This detector is already very good at the task assigned, but we tried to explore possible improvements of it in two directions:

- Improve detection in images as seen from a drone; that is, very small details in images.
- Improve detection precision and latency time.

For the first approach, we explored the use of TPH-Yolov5. TPH-Yolov5 is an open-source project that uses TPH (Transformer Prediction Heads) to improve detection of different scale objects (in our case small objects) (see Figure 3 for an example of detection using TPH-Yolov5)

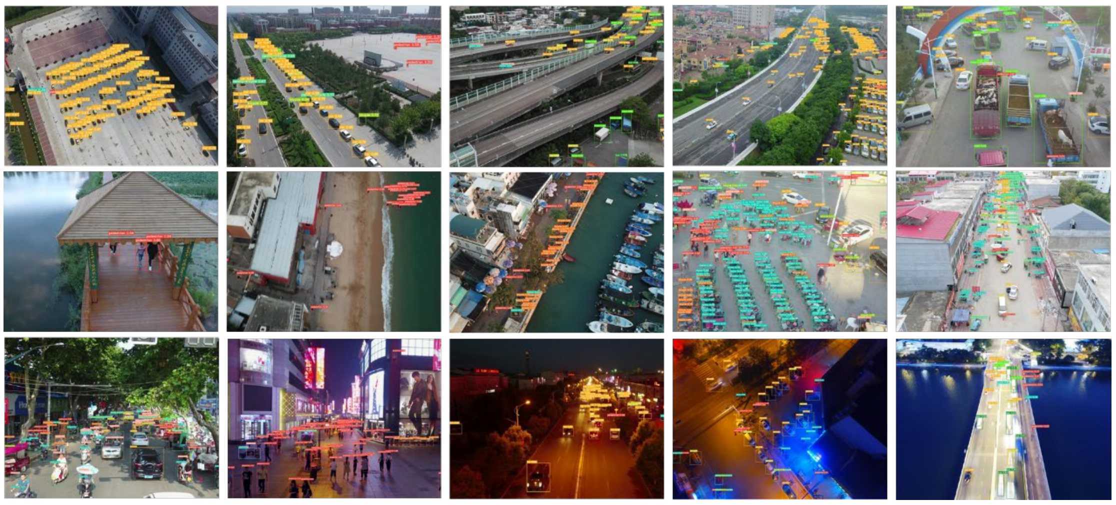


Figure 21 - TPH Yolo detection examples (taken from TPH yolo github page)

Our next step was to adapt our dataset and train with this new architecture using our cluster of machines. Resulting model and program has been also dockerized for its posterior use.

The result of this training was worst in terms of detection precision. Additionally, it did not improve significatively the detection from the drone footage nor was good enough for the use of satellite images.

Second approach has been to use state-of-the-art detection algorithm (Yolov8 at the time of the experiment) to re-train and improve our fire and smoke detector.

For this approach, we used YoloV8 with the Yolov8l model (“large” model) as base and we trained it with the same dataset (you can see some metrics in Figure 4 for a training of 100 epochs).

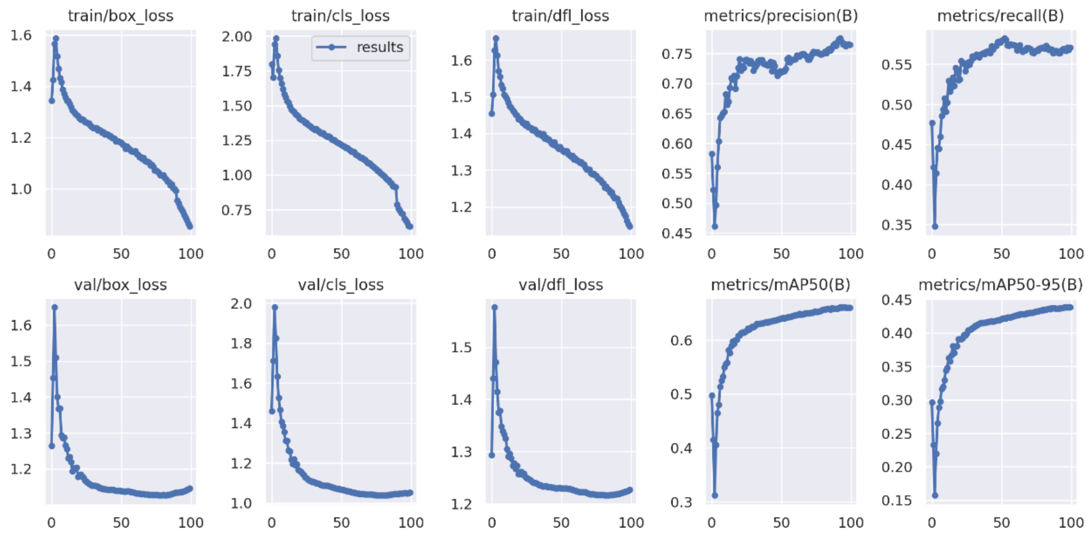


Figure 22 - Yolov8 based training results for 100 epochs

Again, the resulting model was dockerized for future use.

The results of Yolov8 training were slightly (less than 1%) better than the results of Yolov5.

As a possible future direction (not only for SILVANUS, but for our commercial products) we are planning to extend the dataset by using synthetic images (either images with fire added by “inpainting” techniques or pure synthetic images) (see Figure 5)

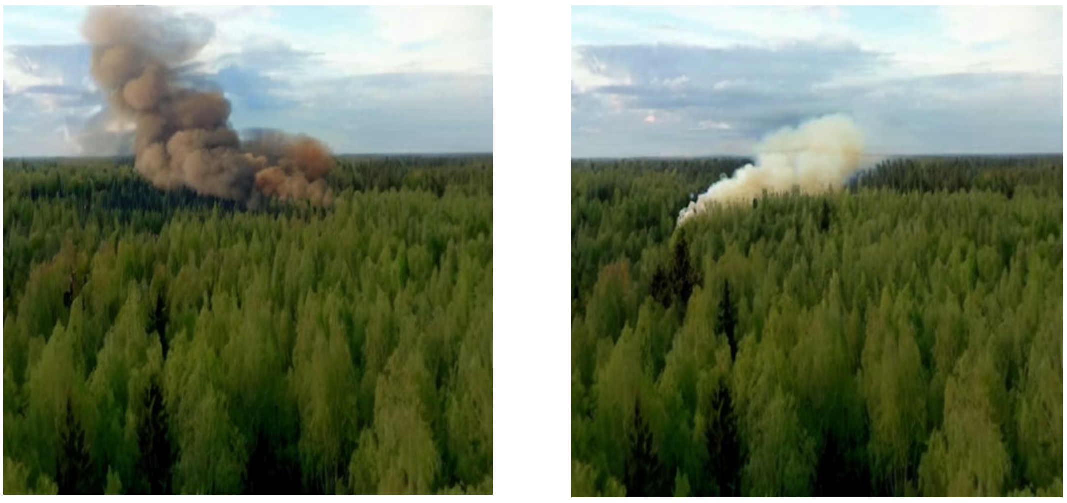


Figure 23 - Two synthetic images from same source with fire and smoke added

3.1.2. Fire and smoke detection using Statistical machine learning

4.1.2.1. Data collection and pre-processing

Before starting the development of the fire/smoke detection and localisation models, CTL collected images from various sources to create the dataset needed for their training. An overview of the dataset curation process can be found in sections 5.2.1.1 through 5.2.1.3 in deliverable D4.2 Demonstration of social media analytics for localising the origin of wildfire ignition. An initial dataset was created to start the training/testing of the algorithms, but it is continuously updated from new sources (e.g., images collected from pilot sites) to enhance it and increase algorithm robustness. The current version, contains ~21K images depicting fire, smoke, both or none. Furthermore, fire/smoke-like objects (e.g., lanterns and clouds, respectively) were also included to minimise false positives (i.e., incorrect classifications as fire/smoke). Lastly, for the localisation models a subset of the images illustrating fire/smoke underwent a manual annotation process for the identification of the areas of interest (within the image) to use as ground truth during training.

4.1.2.2. ML Approaches, ML training and validation

Lightweight variants of the selected detection algorithms were deployed on the CTL's IoT Edge devices, as described in section 4.1.2. through 4.5 in deliverable D4.1 - Demonstration of data collection, aggregation of Earth Observations, weather/climate models and in-situ environmental sensors for forest fire risk/threat assessment. For the case of the AI/ML Processor though, that will be stationed in SILVANUS cloud, we do not have so strict power and processing limitations, therefore the original (non-reduced) versions of the models will be used. Additionally, to the detection models, the cloud will be employed with the localisation models as well, to help identify the fire/smoke regions within the image. Examples of the detection and localisation algorithms are shown in Figure X and Figure XII(a), respectively.

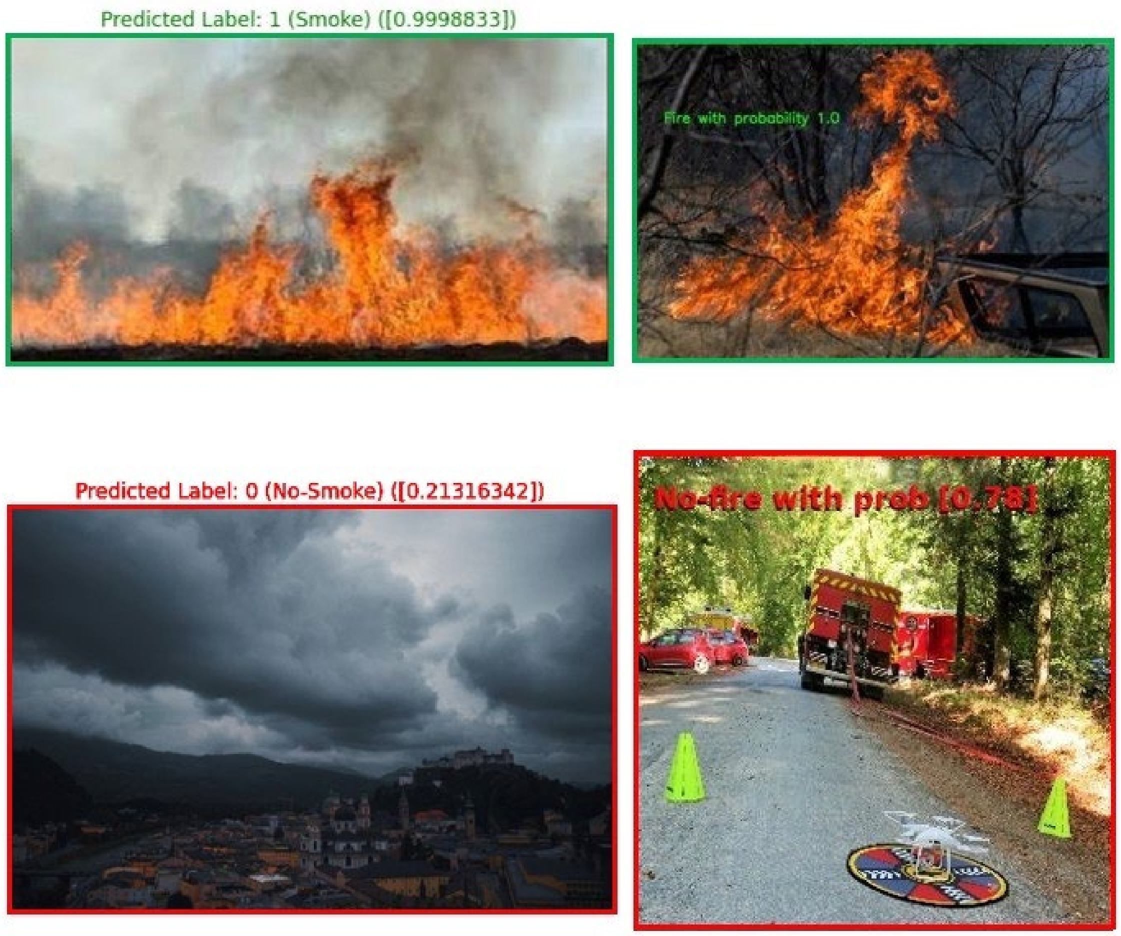

Predicted Label: 1 (Smoke) ([0.9998833])
Fire with probability 1.0
Predicted Label: 0 (No-Smoke) ([0.21316342])
No-fire with prob [0.78]

Figure X - Smoke (left) and fire (right) detection model results

Several ML models are being tested, to determine the most efficient and effective ones for the detection tasks. Some of these models being: ShuffleNet[4], MobileNetV3[5] and Xception[6]. More specifically, we are using two different networks – one for the fire and another for the smoke detection, to ensure we achieve the peak performance of each detector because of the discrepancies of the two phenomena (e.g., colour, movement, etc.). For the training of the detectors, we fine-tune the aforementioned networks, to exploit existing knowledge, by adjusting some of their weights to fit our data.

Material on the training of the fire detection model has already been presented in D4.2, therefore here we will give emphasis to the current results of the smoke detector.

[4] *Ma N., Zhang X., Zheng H. T., Sun J., "ShuffleNetV2: Practical Guidelines for Efficient CNN Architecture Design", Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 116-131*

[5] Howard, Andrew G. et al. "Searching for MobileNetV3." *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019): 1314-1324.

[6] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 1800-1807, doi: 10.1109/CVPR.2017.195.

Regarding the smoke detection task, we have made significant progress in our experiments, which involved testing and experimenting with various deep learning classification models due to the nature of our image data and the construction of a sizable dataset (consists of only several thousands of images). Due to the many available network architectures we aimed to restrict the search space by considering the need for lightweight models that could also be easily deployed in an edge computing environment and would not easily overfit, given our datasets size. Therefore, we mostly focused on experimenting with networks that fulfilled specific requirements and restrictions.

Our primary criterion was that the models should be lightweight, ensuring that the time per inference step on a CPU was less than 100 ms and their total size was under 75 MB, in order to include only relatively small architectures with not so many parameters. To meet these specifications, we carefully selected architectures that met these criteria, ultimately opting for the most sophisticated model within the chosen family. The model architectures that we trained, fine-tuned, and evaluated include: EfficientNetB2V2, DenseNet169, NASNetMobile, MobileNetV3Large, and MobileNetV3Small. By systematically exploring these models, we aimed to identify the most effective solution for smoke detection, balancing accuracy and efficiency for real-world deployment. During our experiments, for all the models tested in our smoke detection experiments, we applied transfer learning by utilizing pre-trained versions of the aforementioned networks. These pre-trained models were based on the ImageNethttps://www.image-net.org/[7]dataset.

To adapt the models to our specific smoke detection task, we made modifications to their architecture. We retained only the backbone of the pre-trained networks, removing their original head part (fully connected layers) of the networks. In place of the removed head, we incorporated our own fully connected layers at the end of the architecture. By utilizing the pre-trained backbone as a feature extractor, the model leveraged the valuable learned information from the previously trained task. Consequently, this approach allowed the models to effectively learn to distinguish smoke in the given images.

Furthermore, we experimented with various fine-tuning and transfer-learning settings. For instance, we tested freezing the backbone base (feature extractor) entirely or just initializing it with the pre-trained weights and continuing the learning process. After careful evaluation, we found that the optimal setting involved freezing the backbone's weights during the initial epochs and then unfreezing the entire network while using a smaller learning rate. This approach prevented the complete erasure of the original features learned from the pre-trained task and ensured better adaptation to our smoke detection objective. Overall, the combination of transfer learning using pre-trained networks, adapting the architecture, and fine-tuning the models allowed us to make substantial progress in our smoke detection experiments.

Based on the results of the above-mentioned experiments, the model which was found to be the most suitable and had the best performance was the MobileNetV3Large, which roughly consists of 3.3 million parameters (including the additional fully-connected layers). The particular model scored F1-score=0.88 on unseen test data examples of the output of the specific detection algorithm are shown in Figure Z. Finally, it is crucial to mention that the algorithm was also tested on image data which were collected during the pilot in Croatia. Contentedly the model accurately detected the observed smoke in the images presented.

[7] https://www.image-net.org/

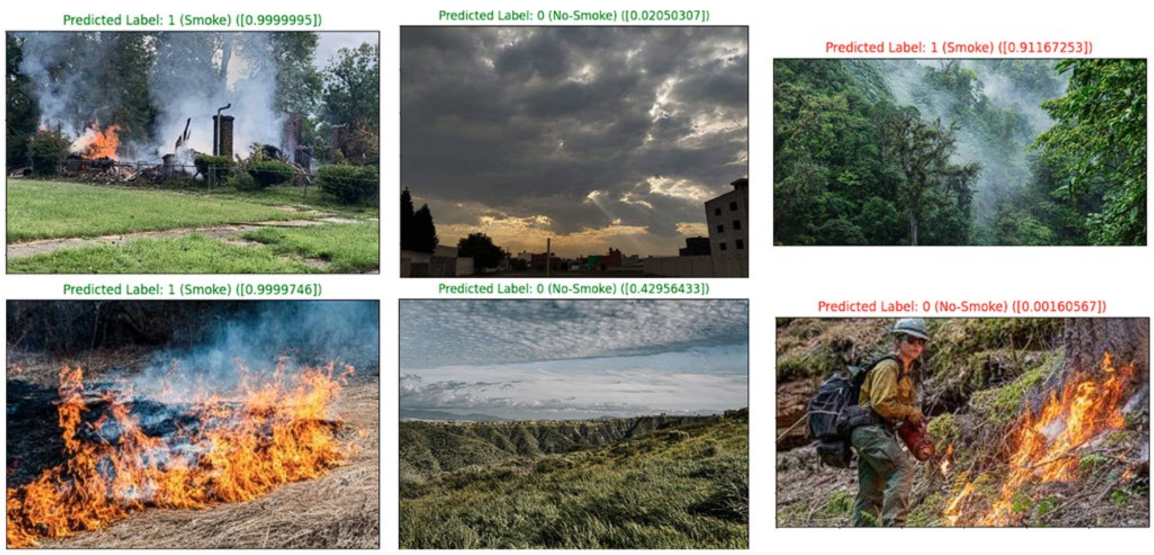

Predicted Label: 1 (Smoke) ([0.9999995])
Predicted Label: 0 (No-Smoke) ([0.02050307])
Predicted Label: 1 (Smoke) ([0.91167253])
Predicted Label: 1 (Smoke) ([0.9999746])
Predicted Label: 0 (No-Smoke) ([0.42956433])
Predicted Label: 0 (No-Smoke) ([0.00160567])

Figure Z - Smoke Detection Results of fine-tuned MobileNetV3Large model

Regarding the localisation networks, at the moment we have results only for the case of fire, which as described in previous deliverables, we are following a superpixel approach (see example in Fig. XII-a). Adopting a superpixel segmentation algorithm, instead of the conventional bounding boxes, has the advantage of detecting smaller and more irregular areas of the image (instead of rectangle areas) – making the detection of small fire sources even within dense forestry more accurate. Additionally, to the training of the model, we focused on the post-processing of its results to smooth-out the predicted fire area and reduce any false positives. In Figure XII(b) we demonstrate the fire localisation results on a sample image, before and after the superpixel localisation post-processing.

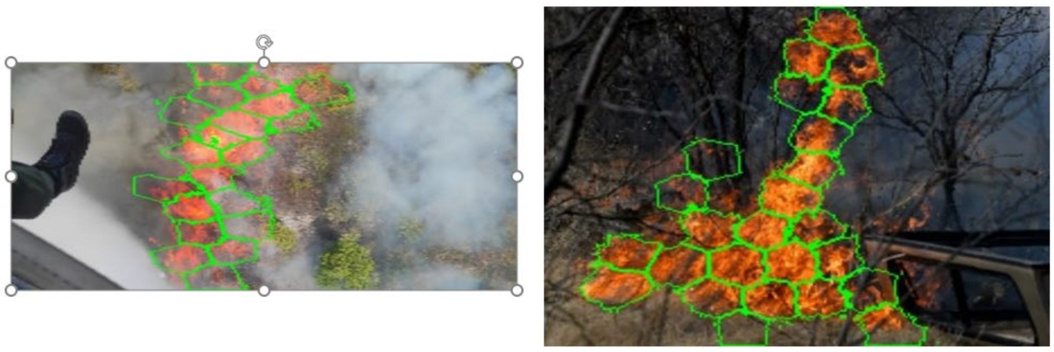


Figure XII (a) - Fire localisation baseline algorithm results on sample images

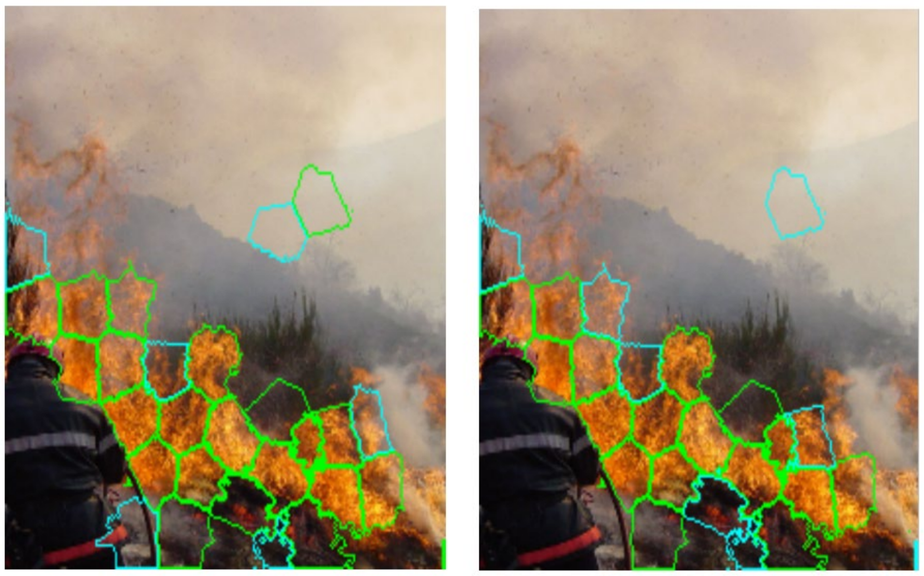


Figure XII (b) - Fire localisation on sample image before post-processing (left) and fire localisation after post-processing results (right).

For the case of smoke localisation, unfortunately the aforesaid method does not yield good results because smoke's color and density significantly vary, depending on the material that is being burned, making smoke localisation slightly more challenging task. Besides this smoke, by nature, looks a lot like other physical phenomena (e.g., clouds, mist, etc.). This resemblance (see Figure Y) gets even more enhanced when you focus on smaller areas of the image, as done in the case of superpixels. So far, for smoke localisation, we have experimented with the employment of object detection models, like YoloV5[8], to precisely locate the smoke within a picture or video frames. However, labelled data that include the exact bounding area that the smoke is located in sample images is required to obtain a model which behaves accurately. At the time being, the labelled samples in our disposal are only a few hundreds of images, therefore, we are actively exploring the possibility of expanding our dataset to construct more accurate object detection models for this enhancement. Currently, our best version of YoloV5 achieved mAP%0.5 = 0.56 on the smoke localisation task.

Furthermore, we plan to experiment with spatio-temporal algorithms, for the analysis of video frames or sequential images. By incorporating the temporal dimension in our algorithms, we aim to achieve higher robustness and generalization accuracy, even in the most challenging scenarios (smoke covering fire, fire far away from the camera etc.).

As this is a work in process, the final models for both the fire/smoke detection and localisation tasks will be presented in deliverable D4.5 - Report on SILVANUS advanced detection capabilities.

[8] https://pytorch.org/hub/ultralytics_yolov5/

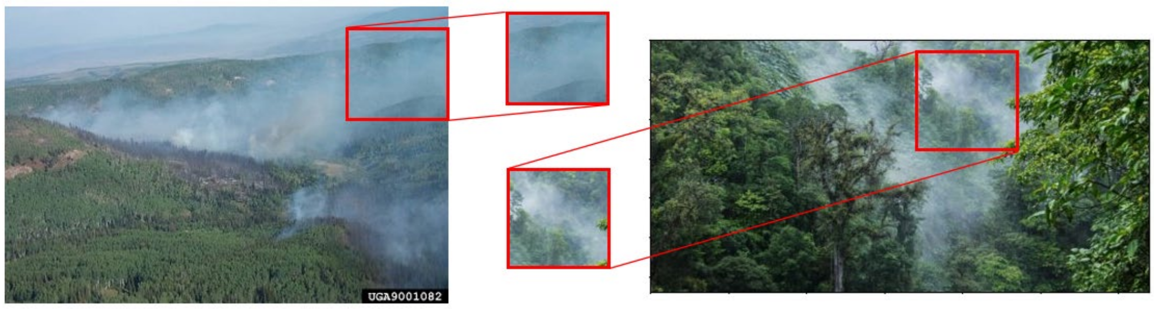


Figure Y - Comparison of image subareas with smoke (left) and mist (right).

Lastly, it is worth mentioning that all models will be dockerised and installed on SILVANUS cloud so that NiFi (and potentially other services) can use them to process new images.

Regarding the localisation networks,

4. Data-driven approaches for fire spread

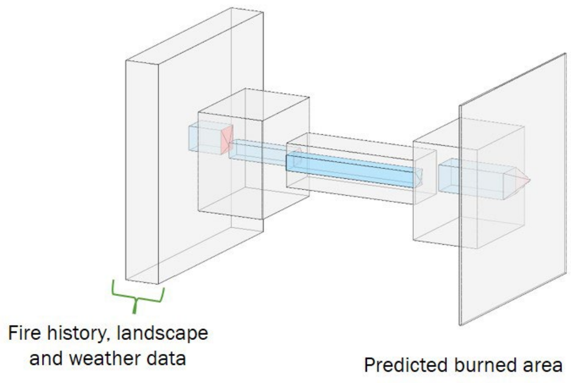
4.1. ML Approaches

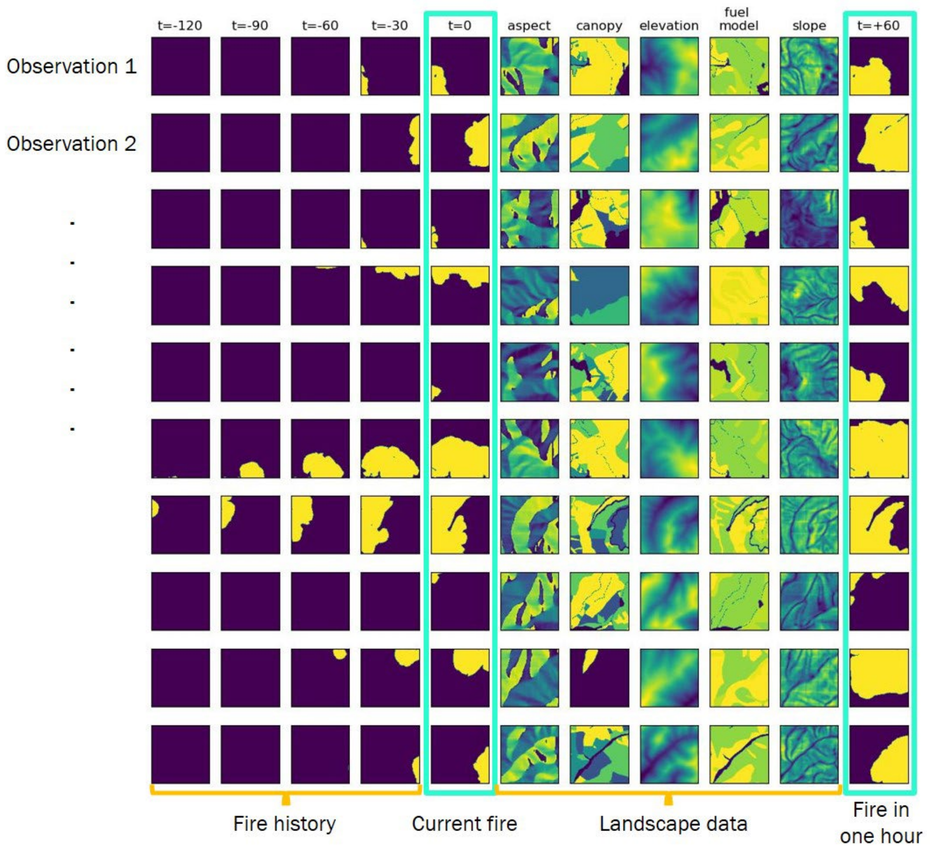
EXUS has developed an initial model that predicts the spread of fire over the next period, given information about the current position of the fire, the topographical features of the landscape, fuel availability and characteristics, and the weather conditions in the area.

Given the spatial structure of the data, a convolutional neural network (CNN) wastrained to make this prediction. All input parameters and features were converted into images, and each channel in the input "image" is a different feature of the current scenario. This includes: the current and historic position of the fire (using up to two hours of historic data); topographical information including the aspect, elevation, slope, canopy cover and fuel model of the area; and weather data including information on temperature, humidity, cloud cover, wind direction and wind speed. The output is an image with a single channel which predicts where the fire will be in the future (using a value of 0 or 1 to represent which pixels are not yet burned or burned, respectively).

The current specifications of the models are as follows. The size of the area that the images represent is one square kilometer (e.g., 1x1 km), and each pixel has a resolution of 10m by 10m. The model currently predicts the position of the fire one hour into the future. Both the area represented by the image, and the future time in which the prediction takes place can easily be changed to train a new model, depending on the needs of the end users.

The CNN is built using PyTorch. It has 17 channels in the input image (these layers are described above) and a single channel in the output image. There are three hidden layers in the CNN, of 32, 64 and 32 channels respectively.



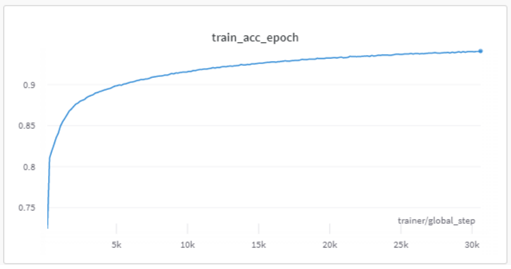


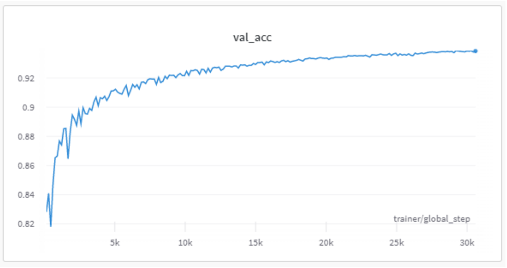
4.2. ML Training and Validation

The model above is trained using 4,875 images, for a total of 100 epochs. A binary cross entropy (BCE) loss function is used, given that the problem is formulated as a classification problem (each pixel must be classified as burning or not burning). During training, the model was evaluated on 696 validation images, and the BCE loss was calculated at the end of each epoch. The model was trained using a batch size of 32 images. After training was completed, the model was tested on 1,392 entirely unseen images.

4.3. ML Evaluation and Results

The final model had a training accuracy of 94.1%, a validation accuracy of 93.9%, and a test accuracy of 93.1% (showing little evidence of overfitting). The final training BCE loss of the model was 0.362. The results were visually inspected (see image above) to assess how reasonable the predictions were (e.g., did the burned area remain burned, were the magnitude and direction of the new burned areas comparable to the actual burned area?). A cut off of 0.5 was used to classify pixels as burned or not burned. This cutoff can be modified to make the model more conservative or aggressive, depending on the needs of the end user.





5. Satellite supporting tools

5.1 ML Approaches

Atos has created two supporting tools for helping in the analysis of images taken from satellite: Segmentation module and Super-resolution module.

Segmentation module has used as base the GeoSeg repository[9]. This module was state-of-the-art at the moment and is using a semantic segmentation toolbox based on PyTorch that makes use of advanced Vision Transformers for remote sensing (e.g., satellite) image.

We retrained this repository with our own dataset of satellite images using our hardware cluster for training. The resulting module was dockerized.

You can see some results in next figures. Figure 24 shows the original image and Figure 25 the segmented version.

Next actions for our side will be to compare the result of the segmentation against the results offered by Copernicus Land Monitoring Service[10].

[9] https://github.com/WangLibo1995/GeoSeg

[10] https://land.copernicus.eu/imagery-in-situ/eu-dem/eu-dem-v1.1

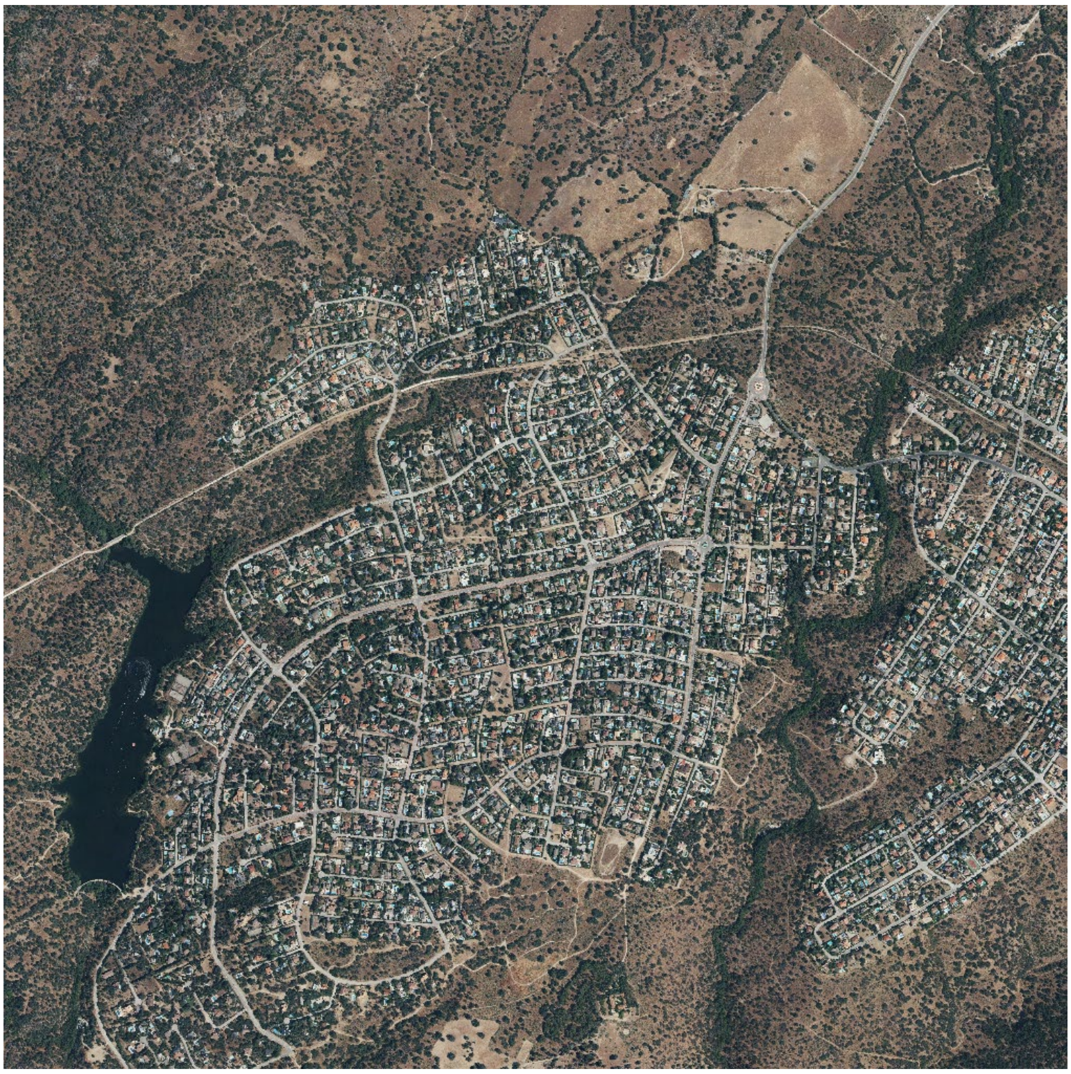


Figure 24 - Original satellite image

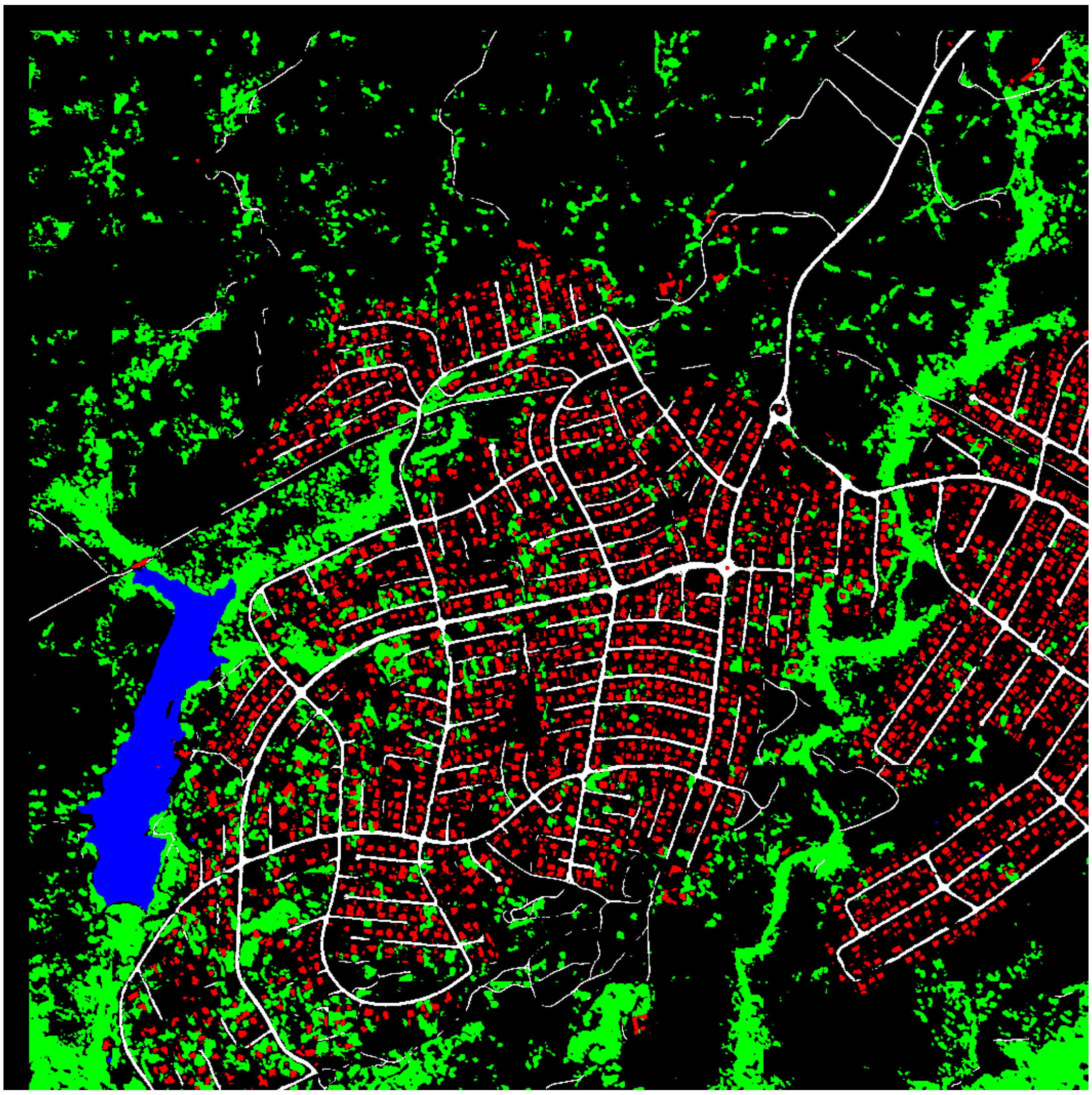


Figure 25 - Segmented image

Superresolution module was created as a complementary tool. The objective was to improve the satellite images in cases where the quality was not adequate for a proper segmentation.

Superresolution was created based on StableDiffusion 2.0 (using the so-called StableDiffusionUpscalePipeline)[11]. Resulting module has been dockerized.

This module was tested with satellite images that later were segmented using the segmentation tool for checking and comparing results (Figure 8)

[11] https://huggingface.co/docs/diffusers/v0.16.0/en/api/pipelines/stable_diffusion/upscale#diffusers.StableDiffusionUpscalePipeline

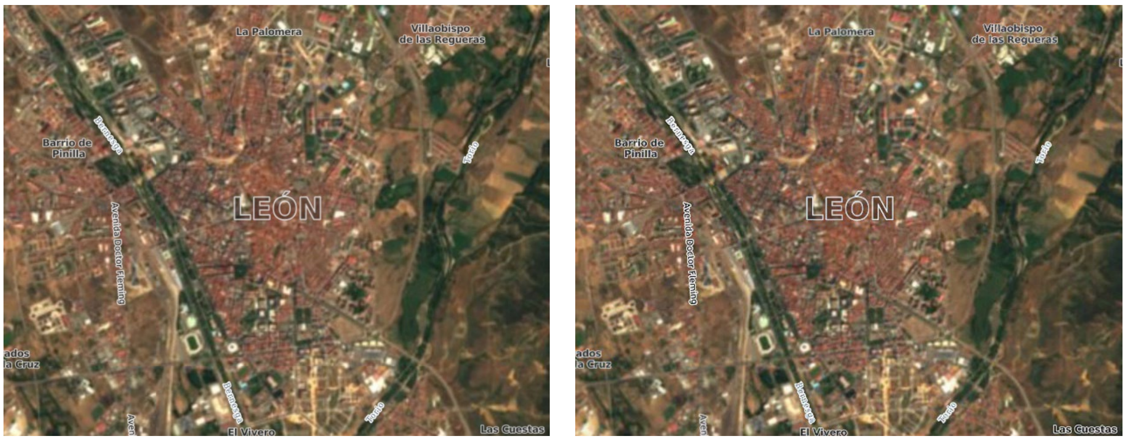


Figure 26 - Low resolution image (from Copernicus site) and super-resolution result

5. Data-driven approaches for fire danger risk prediction

The danger and risk of fire depends on several factors, and the correlation between these factors can be spatial, temporal or spatio-temporal. For instance, the occurrence of fire primarily depends on the availability of burnable mass, which in turn depends on the weather not only on the day of the fire but also on the weather of the days preceding the fire event. On the other hand, spatial correlation between neighboring areas is important to predict the probability of the fire to grow into neighboring areas e.g., the topography, wind speed etc. Including only the spatial or temporal features neglects their correlation which is important to predict the joint probability that a fire will occur and further grow into a large. The joint probability of fire is important in order to properly allocate resources to more severely affected areas.

In order to evaluate data-driven approaches and compare them with empirical approaches for fire danger risk we followed the work by [1] in which they study the use case of forest fires in the region of Greece.

5.1. ML Approaches

Keeping in mind the spatial, temporal and spatio-temporal nature of forest fire, we test three different neural networks. Spatial correlation between the fire events and the features are best explored using a Convolutional Neural Network (CNN) [2]. Temporal correlation instead is best explored using Long Short-Term Memory (LSTM) [3] as it is capable of taking time series data as input. To explore correlations between the spatial and temporal features, we adopt a ConvLSTM [4].

Input Features

The input of all three networks consists of the following features that are important for the prediction of fire: 1) **daily weather forecast** which includes the maximum and minimum temperature, and components of wind speed together with max precipitation, 2) **Local vegetation** includes Leaf Area Index (LAI), Fraction of Photosynthetically Active Radiation (Fpar), Normalized Difference Vegetation Index (NDVI), and Enhanced Vegetation Index (EVI),

3) Other satellite variables such as **day and night land surface temperature**, 4) factors related to **human activity** are modeled using maps of road density, population density, and 5) **topography variables** include elevation, aspect and slope. The network inputs are mapped to the historical burned pixel on the next day. The weather forecast maps are taken on the same date, while the satellite features are taken from the previous. All the input features are harmonized to the same resolution.

Network configuration

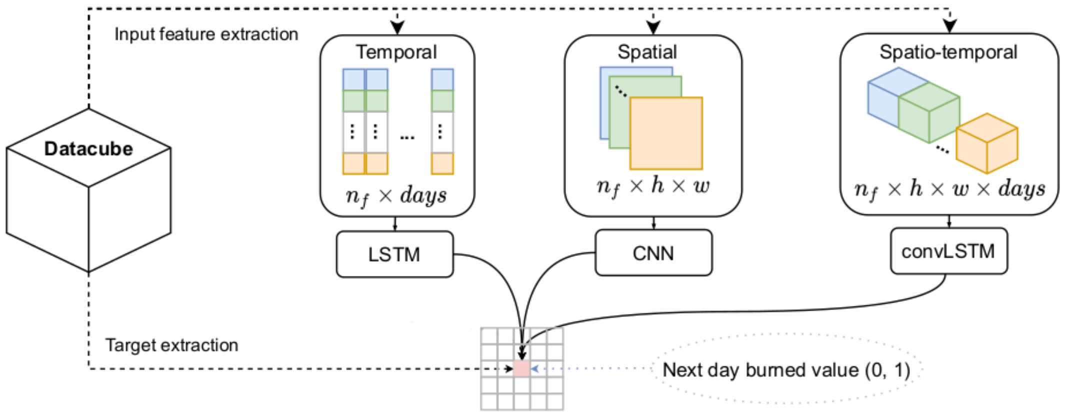


Figure 27 – ML Models considered for predicting next daily fire danger.

The three different networks (CNN, LSTM and ConvLSTM) considered and the input are schematically shown in Figure 27 (adapted from [1]). Each of the input features of the network is mapped to the burned value of the pixel: 1 if the pixel is burned on the day and 0 if it is not burned.

For CNN, the input consists of 25 x 25 pixel images in 18 channels (13 dynamic features, and 5 static features listed in the previous Section Input features) i.e. a single input has the shape of 1 x 18 x 25 x 25. The image is centered around the location of the burned pixel. For LSTM, the input consists of time series data spanning 10 days for each pixel in the dataset for all the 18 input features (10 x 18 x 1 x 1). For ConvLSTM, the input is a time series consisting of 25 x 25 images in 18 channels (10 x 18 x 25 x 25).

In CNN, we perform the convolution using 16 filters with kernels of size 3 x 3, convolved with padding of 1 and stride of 1. Following the convolution layer, a 2 x 2 max pooling is performed after which the flattened output is passed through linear layers separated by a dropout layer of p = 0.5 and then finally passed to a 2-class logsoftmax layer. The learning rate and weight decay are set to 0.0004, and 0.03 respectively.

In LSTM, we use one LSTM layer with 64 neurons the output of which passes through two linear layers of 64 and 32 neurons, respectively, where the neurons are separated by a dropout layer of p = 0.5. The output of the linear layers is then passed to a 2-class logsoftmax layer. The learning rate and weight decay are set as 0.001, and 0.01 respectively.

For ConvLSTM, the convolution part of the network follows the configuration of a CNN as having 16 filters with 3 x 3 size kernel whereas the LSTM part consists of a single LSTM layer. For ConvLSTM, weight decay and learning rate are set at 0.03, and 0.0001 respectively.

5.2. ML Training and Validation

The training and validation of the networks is performed using the dataset by Prapas et al. [5], where they study the use case of predicting Fire danger in Greece. The dataset contains historical data for the input features and the information of the burned pixel, harmonized to a uniform resolution of 1 km x 1 km. Furthermore, the dataset spans from 2009 to 2020.

We use the data from 2009 to 2018 for the training dataset and 2019 for the validation dataset. We choose to follow this procedure of splitting the training and validation data instead of random splitting in order to avoid overestimating the performance of the network. Prediction of fire danger is a forecasting problem and must be validated with a dataset from the future or hindcast in the past.

The dataset for training is prepared in the following way: for each year in the dataset, we collect all pixels where fire occurred and extract the features in the format required by each of the networks described in Network configuration.

In a problem like predicting fire danger, the negative sampling (sampling of No Fire events) is very important because the dataset is imbalanced - there are many more negative events than positive events (Fire events). In order to avoid the network from learning very trivial mapping care must be taken when drawing the negative samples (for example, picking negative samples from areas which are at little to no fire risk). We therefore randomly sample twice as many negative samples as positive samples, taking care that, 1) for any given year, pixels which have already been included in the positive sample at any day are not included in the negative sample, 2) we use CLC classes to select negative samples from regions which are susceptible to fire risk (for example, sea or water bodies cannot be at risk of forest fire), and 3) we sample negative events from the months which present high fire risk.

The features extracted from the procedure mentioned above are presented to the network in batches of 128 samples/batch and trained for 50 epochs. We use a cross-entropy loss function with Adam optimizer. To further balance the dataset, in computing the loss function, we weigh the class of Fire events more than the No Fire events.

The network performance is measured using the AUROC, F1, F2, Precision and Recall scores; the F2 score of the validation dataset during the training is used to select the best model which is used for the inferencing phase. The weights given to the class of Fire events in the computation of the loss function are optimized to achieve a balance between the Precision and Recall score (with more weight towards the recall score).

5.3. ML Evaluation and Results

After selecting the best models of each type (CNN, LSTM and ConvLSTM) following the procedure described in the previous section, in this section we report the results from the best model and comparing the results with the Fire Weather Index, which is has been a standard since several decades [6].

The FWI is considered an accurate empirical model for predicting the occurrence of forest fires. Keeping this in mind, we compare the fire danger map of Greece from our network with the FWI map for a given day.

In the following table the validation score corresponding to the best models chosen during the training are reported.

Table 2 – Validation performance score for the three different networks (CNN, LSTM, ConvLSTM)

Network/Score	**Precision**	**Recall**	**AUROC**	**F1**
CNN	0.48	0.82	0.69	0.60
LSTM	0.52	0.80	0.71	0.63
ConvLSTM	0.64	0.87	0.81	0.74

While these scores, indeed, represent the statistical performance of the network for the validation dataset, it is only part of the picture. Therefore, in the following figure for example we compare the fire danger prediction of the CNN network for a given day with the FWI of the same day.

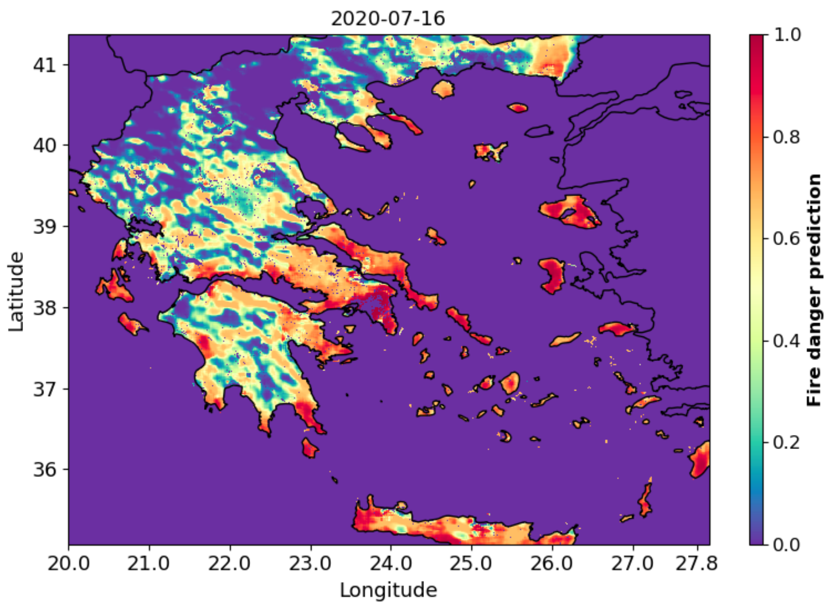


Figure 28 – Fire Danger Prediction based on CNN for July 16th 2020 for Greek region.

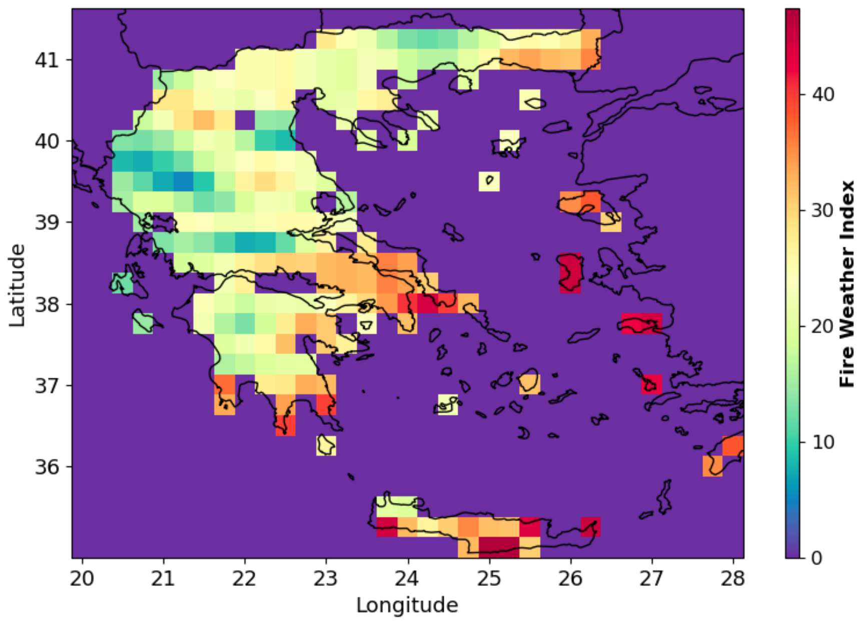


Figure 29 – Fire Weather Index from https://cds.climate.copernicus.eu/cdsapp#!/dataset/cems-fire-historical for July 16th 2020 for Greek region.

Comparing the two approaches for the Greek region, one can appreciate the similar patterns of the two maps in general. In addition, however, the prediction map from CNN shows finer details in the occurrence of fire due to higher resolution, and the additional information provided during the training of the network in the form of the feature described in Section Input features. Compared to the FWI, the additional information, in particular those concerning human activity, helps the network in better assessing the fire danger in specific areas compared to the FWI.

5.4. Next Steps

In the second part of the project, we will focus on the train/validation and testing of CNN, LSTM and ConvLSTM models and assessing their performance in predicting fire danger in the peak fire season in the SILVANUS pilot sites (Gargano, Portugal) using heterogeneous datasets (EO Satellite data and derived products – e.g., NDVI, FAPAR, high-resolution weather forecasting models).

5.5. References

[1] Prapas, Ioannis, et al. “Deep Learning Methods for Daily Wildfire Danger Forecasting.” arXiv, 2021. arXiv, https://arxiv.org/abs/2111.02736.

[2] Yann, Lecun, et al. “GradientBased Learning Applied to Document Recognition.” Proceedings of the IEEE, vol. 86, no. 12, 1998, pp. 2278-2324.

[3] Sepp, Hochreiter, and Jurgen Schmidhuber. “Long short-term memory.” Neural Computation, vol. 9, no. 8, 1997, pp. 1735-1780.

[4] Shi, Xingjian, et al. “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting.” arXiv, 2015. arXiv, https://arxiv.org/abs/1506.04214.

[5] Turner, J.A., and B.D. Lawson. “Weather in the Canadian Forest Fire Danger Rating System. A user guide to national standards and practices.” Pacific Forestry Centre, 1978.

[6] Prapas, Ioannis, et al. A Datacube for the analysis of wildfires in Greece. 2021. Zenodo, https://zenodo.org/record/4943354#.ZIMjFNJBwUT. Accessed 9 June 2023.

6. Fire and smoke dataset

Atos has created for Silvanus a fully operational dataset of fire and smoke in 50forest areas using a drone view. All the images are tagged. The photos (25903) have been synthetically generated. The process for the creation and tagging of the dataset has been:

- Creation of the "prompts" for the synthetic images: In this phase, we created 141 prompts of forest fires as seen from drones. To generalize, we also included different types of terrain, vegetation, time of day, altitude, season and visibility, including also nighttime. We included different types of trees, meadows, shrubs, and undergrowth.
- With this "prompts" we launched the process of creation of synthetic data, creating, in different batches, up to 100 photorealistic images of each prompt. The rate of creation was of 1 image per 11 seconds using Nvdia 3080 graphical card. In this phase we created roughly 30000 images. This phase took more than one week. (see Figure 27 and Figure 28 for some examples)
- After finishing this process, we passed the resulting images through an artificial intelligent "interrogator" based on CLIP to filter images with some defect or artefact. We discarded all images that were not realistic enough (e.g. fire over water) or not valid for our propose (e.g. fire in buildings) between other trials. In this phase we did not eliminate images without fire and smoke. This process took less than 8 hours discarding over 2000 images.
- Finally, we passed the resulting images though Atos detector of fire and smoke modified to:
  - Create the labelling using coco-dataset style.
  - Discard all images that does not have, at least, one fire and/or smoke detection on them.
- The final set, prepared for training, included 25903 images and their labels. The set was prepared using the Coco dataset files structure.

The dataset has been uploaded to : https://venakatreleaf.sharepoint.com/:f:/r/sites/silvanus-ga/Shared Documents/General/Datasets?csf=1&web=1&e=u8CHA1 as a ZIP file (uploading as non-zip probed to be too prone to failure)

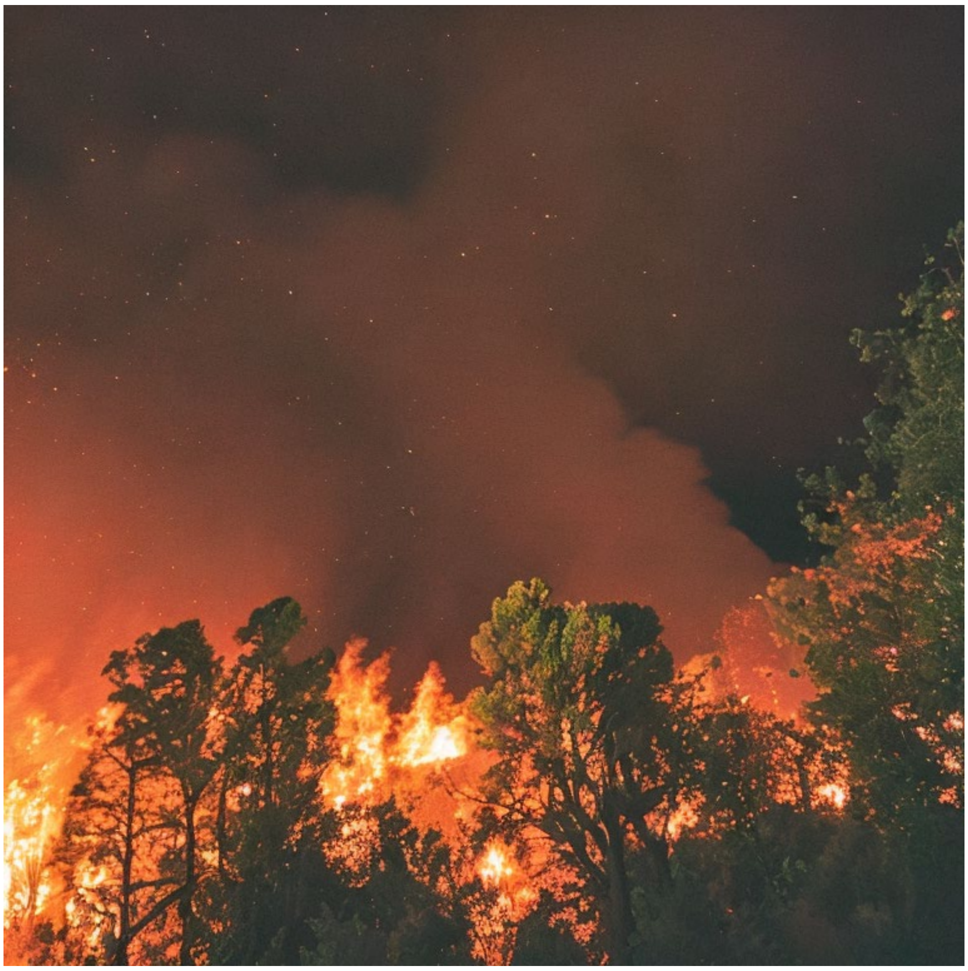


Figure 30 - Fire and smoke (synt generated image)

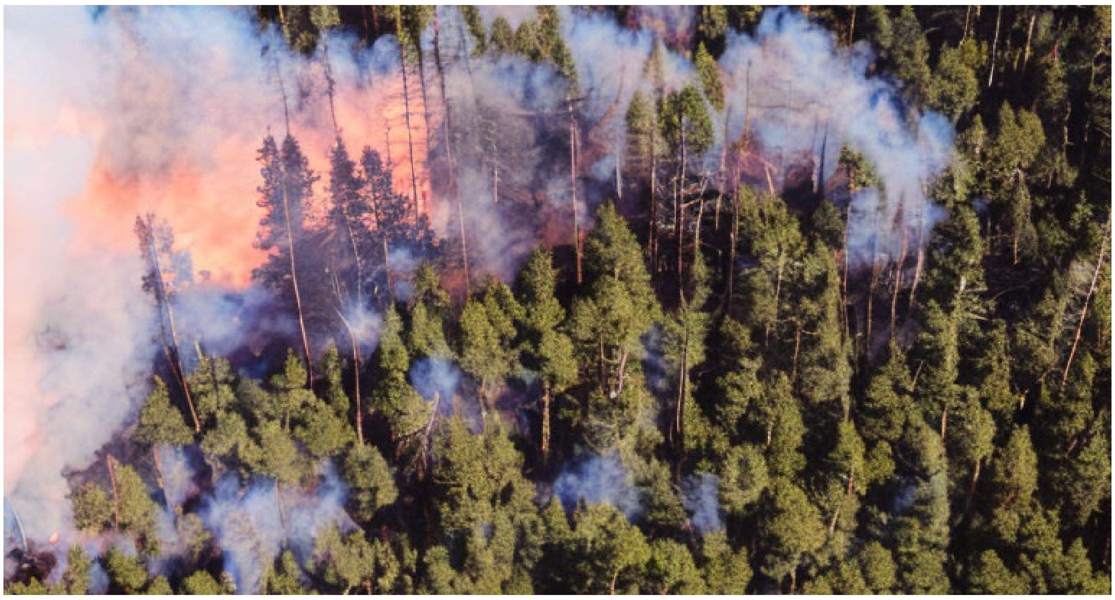


Figure 31 - Fire and smoke (synt generated example)

6.1.1. Structure of the dataset:

The dataset is included in the folder called "fire" compressed into a file called “fire.zip”. Inside the “fire” folder you can find:

- fire.yaml: is a file, contains the description of the classes. There are only two classes in our dataset, “0” for fire and “1” for smoke
- images: is a folder including all images. Inside, you can find:
    - "train" folder with all images for training (20722 images files). This is roughly the 80% of the dataset
    - "val" folder with all images for validation (5181 images files). This is approx. 20% of the dataset.
- labels: is a folder with all labels. Inside you can find:
    - "train" folder with all labels for training (20722 labels files). This is roughly the 80% of the dataset. The names of the labels files are the same as the image file names with extension "txt"
    - "val" folder with all images for validation (5181 labels files). This is approx. 20% of the dataset. The names of the labels files are the same as the image file names with extension "txt"
    - The format of the labels follows COCO tagging format:

      ClassID x_center y_center width height

      Note that all the numbers are normalized between 0 and 1 using the width and height of the image
      ClassID is the same found in the *yaml* file; 0 for fire and 1 for smoke

6.1.2. Ultralytics Hub proof of concept

As a proof of concept and test of the dataset already created, we trained a small subset of it using Ultralytics HUB[12] in order to obtain a detector able to be run in an Android self-phone.

We made two small tests:

- First, we uploaded 2000 images for training and 1000 for validation in a private view
- Then, we trained the images (using google Colab) including:
    - Yolov5 during 100 epochs
    - Yolov8 during 100 epochs
- Result was then migrated to Android compatible format and put inside Ultralytics App in our personal account (using a private view).

The resulting models, although limited, are good enough to detect fire and smoke using an Android self-phone (Figure 29).

[12] https://ultralytics.com/hub

This POC proves the quality of the created dataset. Certainly, including more fine tuning of hyperparameters, the use of more images and more epochs of training will yield a better result.

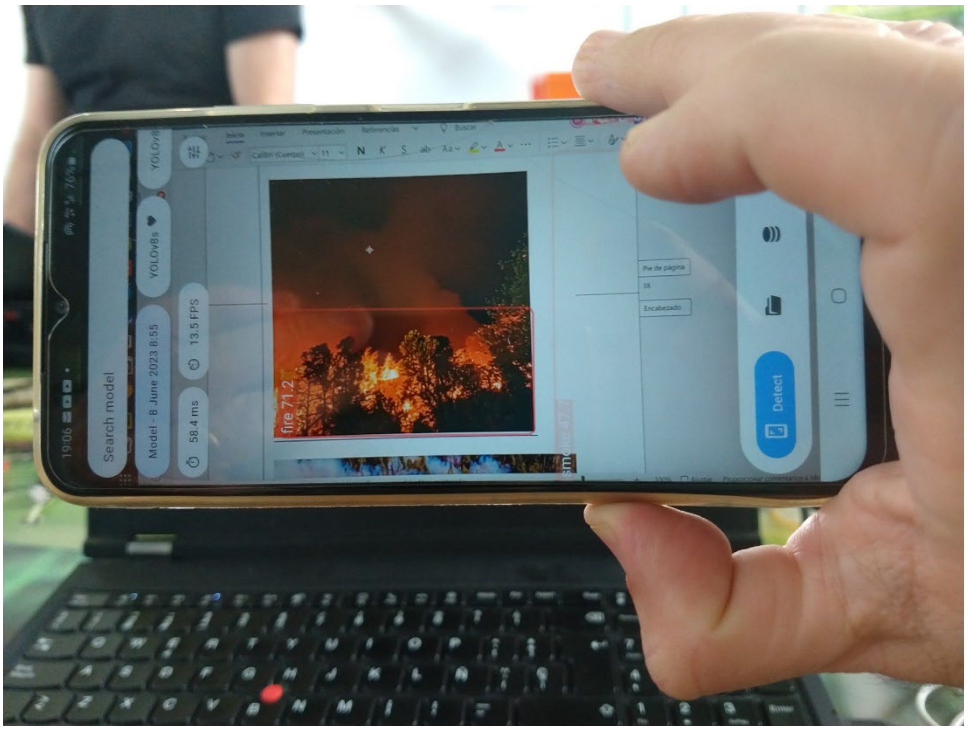


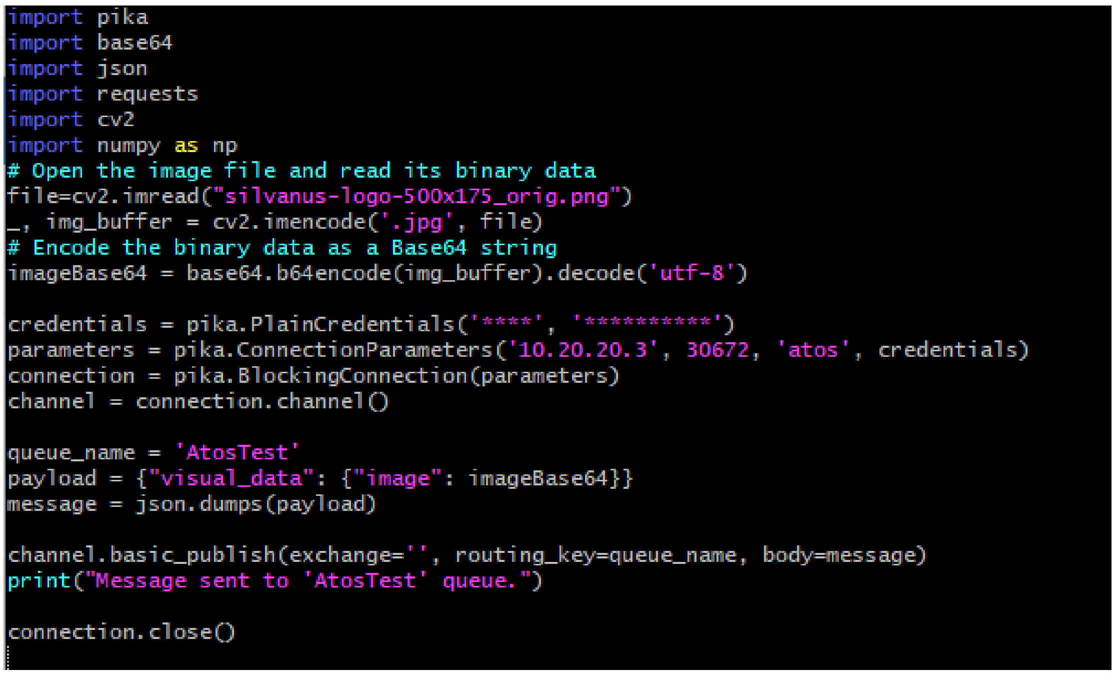
Figure 32 - Fire and smoke self-phone training POC

7. Integration within SILVANUS Platform [DELL]

7.1. Fire and smoke detection from Atos

Atos fire and smoke detector will be executed in the pipeline of image detection running in the Edge device provided by Dell.

- The detector will be connected to Rabbit MQ subscribed to the JSON message containing the picture to analyze. Whenever a new image arrives, the system will read it, decode it (from base64 format) and analyze it looking for fire and smoke. Currently, the Fire and smoke detection service is tested and validated over “AtosTest” queue. Displayed in Figure 30 is a sample Python code snippet that illustrates how to utilize the SAL RabbitMQ message bus for requesting the Fire and Smoke Detection service.



```python
import pika
import base64
import json
import requests
import cv2
import numpy as np
# Open the image file and read its binary data
file=cv2.imread("silvanus-logo-500x175_orig.png")
_, img_buffer = cv2.imencode('.jpg', file)
# Encode the binary data as a Base64 string
imageBase64 = base64.b64encode(img_buffer).decode('utf-8')

credentials = pika.PlainCredentials('****', '**********')
parameters = pika.ConnectionParameters('10.20.20.3', 30672, 'atos', credentials)
connection = pika.BlockingConnection(parameters)
channel = connection.channel()

queue_name = 'AtosTest'
payload = {"visual_data": {"image": imageBase64}}
message = json.dumps(payload)

channel.basic_publish(exchange='', routing_key=queue_name, body=message)
print("Message sent to 'AtosTest' queue.")

connection.close()
```

Figure 33: Python script to request picture analytics from the Fire and smoke detection service

- As an output, the system will send a JSON message to Rabbit MQ containing the detection data (image with detection, boxes and confidence of the detection). The image with the detections (see Figure 30) will be sent inside the JSON using base64 encoding.

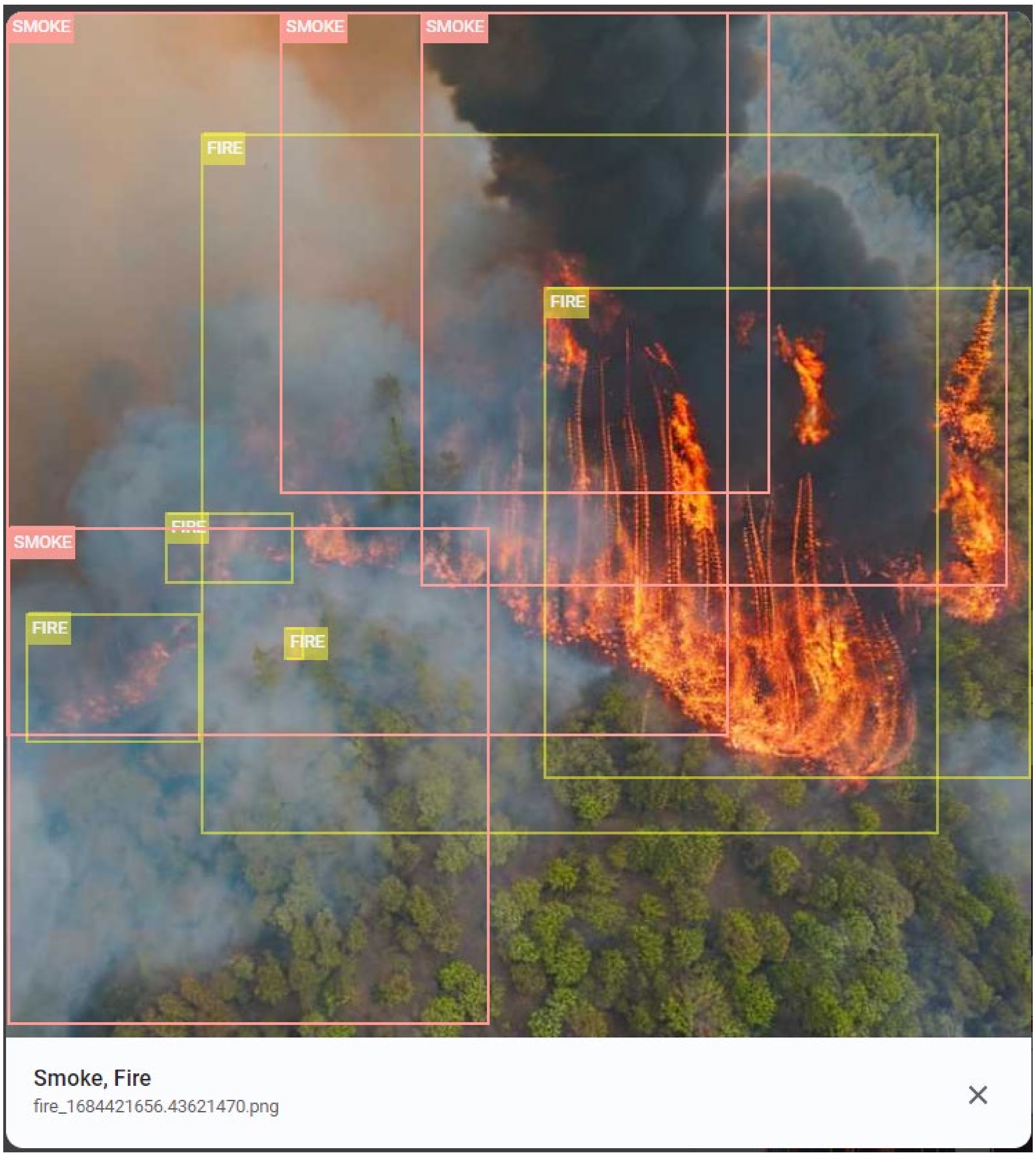


Figure 34 - Fire and smoke detector output (Atos)

7.2. Health KPIs for response teams and citizens from UTH

This subsection provides an overview of the files adopted to annotate sensor readings that monitor the effects of a fire in a specific area. In this documentation, we provide details about two json files, i.e., data.json & metadata.json as well as for the accompanying Python code which is responsible to produce/manage them. These files are used in a data processing workflow involving the submission of files to a specific URL (http://192.168.168.4:9004/) using a multipart POST request where the collected data can be consumed by other components.

The data processing workflow consists of the following steps:

1. Collect emissions data from Raspberry Pi sensors.
2. Store collected data in a Mongo DB.
3. Analyze data and calculate the Air Quality Index (AQI).

4. Create the respected data.json and meta-data.json files.
5. Submit the latter files to the SILVANUS Cloud.

The discussed JSON files represent data and meta-data information related to environmental sensors and air quality measurements, as presented in Table 2.

Table 3: Air quality data

data.json	V1.0 15/06/2023	This file contains a structured representation of data, specifically related to air quality sensor readings	Health impact monitoring
meta-data.json	V1.0 15/06/2023	This file contains metadata information associated with the data.json file	Health impact monitoring

7.2.1. Air quality data

The first JSON file (data.json) captures sensor data related to various air quality parameters and provides location and timestamp information for each measurement.

The structure, as well as the content of each field, are summarized as follows:

- **uuid**: A unique identifier for the data entry.
- **sensors**_type: An array that lists the types of sensors used. In this case, it includes PM (Particulate Matter) measurements, sulfur dioxide, carbon monoxide, nitrogen dioxide, and ozone.
- **timestamp**: The date and time when the data was recorded, in ISO 8601 format.
- **location**: An array containing location information where the measurements were taken. It includes a placename and geometry (latitude and longitude coordinates) within a Point object.
- **area**: An object that specifies the area of influence for the sensor data. It includes the **radius** value and the **unit** of measurement.
- **sensor_id**: The identifier of the sensor device used to collect the data.
- **sensory_data**: An object that provides the measured values for each sensor type. Each sensor type has an associated sub-object with a value and a unit.
- **AQI**: The Air Quality Index (AQI) value is calculated based on the collected data. In this case, the AQI is categorized as "Extremely Poor."

An example of the data.json is presented below:

```
{
   "uuid": "uth-123123-lkasjd82-askjd91230asd",
  "sensors_type": [
    "PM1.0",
    "PM2.5",
    "PM10.0",
    "sulfur dioxide",
    "carbon monoxide",
    "nitrogen dioxide",
    "ozone"
  ],
  "timestamp": "2023-04-23T11:29:36.372+00:00",
  "location": [
    {
      "placename": "somewhere",
      "geometry": {
        "type": "Point",
        "coordinates": [
          {
            "lat": 35.151688,
            "lon": 33.350244
          }
        ]
      }
    }
  ],
  "area": {
    "radius:": 2,
    "unit": "meter"
  },
  "sensor_id": "raspberry_1",
  "sensory_data": {
    "PM1.0": {
      "value": 228.0,
      "unit": " micrograms per cubic meter"
    },
    "PM2.5": {
      "value": 230.0,
      "unit": " micrograms per cubic meter"
    },
    "PM10.0": {
      "value": 527.0,
      "unit": " micrograms per cubic meter"
    },
    "sulfur dioxide": {
      "value": 228.0,
      "unit": "ppm"
    },
    "carbon monoxide": {
      "value": 1.0,
      "unit": "ppm"
    },
    "nitrogen dioxide": {
      "value": 0.2,
      "unit": "ppm"
    },
    "ozone": {
      "value": 0.2,
      "unit": "ppm"
    }
  },
  "AQI": "Extremely Poor"
}
```

7.2.2. Air quality metadata

The meta-data.json file contains metadata information about the data.json file including details about the data format, spatial and temporal aspects, data lineage, and associated tags. The structure, as well as the content of each field, are summarized as follows:

- descriptor: Contains information about the JSON file's descriptor.
- uuid: The unique identifier associated with the data in the data.json file.
- obj-class: The object class, which in this case is "IoT" (Internet of Things).
- format: Describes the format details.
- type: The type of format used, which is JSON in this case.
- resolution: The resolution value associated with the data.
- output: The output format used, which is also JSON.
- access: Describes the access level or permission setting for the data, set to "default" in this case.
- dataset-type: Specifies the type of dataset, specifically "air-quality".
- created: The timestamp or date when the data was created, provided in Unix timestamp format.

● spatial: Contains spatial information about the data.

● bbox: Represents a bounding box polygon defined by a set of coordinates, forming a closed shape.

● coordinates: Contains an array of latitude and longitude coordinates that define the bounding box.

● pilot: Indicates the pilot project associated with the data, specified as "greek".

● temporal: Contains temporal information about the data.

● datetime: Specifies the temporal reference, set to "latest" indicating the most recent data point.

● daterange: Indicates the availability of a date range for the data, specified as "from:to".

● lineage: Describes the lineage of the data object.

● source: Specifies the source of the data. In this case, it is an empty array indicating no specific sources.

● processing: Indicates the processing stage of the data, set to "raw" implying that the data is in its original, unprocessed form.

● tag: Contains additional tags or labels associated with the data.

● device: Specifies the device used to collect the data, in this case, a "raspberry pi".

● sensors: Lists the types of sensors used, matching the data.json file's "sensors_type" field.

● AQI: Indicates whether the Air Quality Index (AQI) is available, set to true.

An example of the meta-data.json is presented below:

```json
{
  "descriptor": {
    "uuid": "uth-123123-lkasjd82-askjd91230asd",
    "obj-class": "IoT",
    "format": {
      "type": "json",
      "resolution": "100",
      "output": "json"
    },
    "access": "default",
    "dataset-type": "air-quality",
    "created": "1674574406.7829435"
  },
  "spatial": {
    "bbox": "POLYGON ((16.0295831170816712 41.9183734508349062, 16.0295831170816712 41.8832522880823319, 16.0872243646491313 41.8832522880823319, 16.0872243646491313 41.9183734508349062, 16.0295831170816712 41.9183734508349062))",
    "coordinates": [
      {
        "lat": 41.918373450834906,
        "lon": 16.02958311708167
      },
      {
        "lat": 41.88325228808233,
        "lon": 16.02958311708167
      },
      {
        "lat": 41.88325228808233,
        "lon": 16.08722436464913
      },
      {
        "lat": 41.918373450834906,
        "lon": 16.08722436464913
      }
    ],
    "pilot": "greek"
  },
  "temporal": {
    "datetime": "latest",
    "daterange": "from:to"
  },
  "lineage": {
    "source": "[]",
    "processing": "raw"
  },
  "tag": {
    "device": "raspberry pi",
    "sensors": [
      "PM1.0",
      "PM2.5",
      "PM10.0",
      "sulfur dioxide",
      "carbon monoxide",
      "nitrogen dioxide",
        "ozone"
    ],
    "AQI": true
  }
}
```

7.2.3. Python code

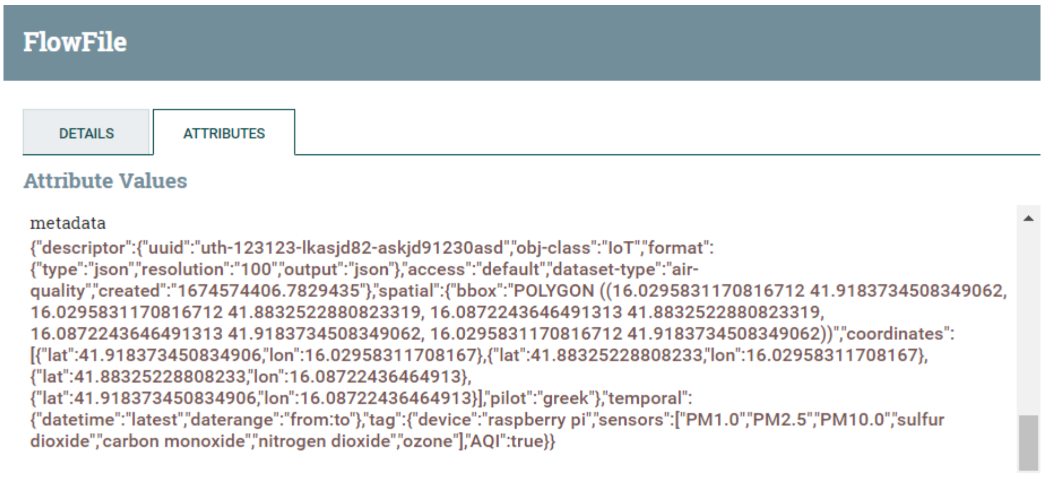
The following Python code performs a POST request to a specified URL (**http://192.168.168.4:9004/**) with attached files as multipart form data. The code demonstrates how to send a POST request with files attached and retrieves and prints information from the response, including the status code and the parsed JSON content.

```python
import requests
import json
headers = {
    # 'Content-Type': 'multipart/form-data',
}

files = {
    'data': open('data.json', 'rb'),
    'metadata': open('metadata.json', 'rb'),
}

response = requests.post('http://192.168.168.4:9004/', headers=headers, files=files)

print(response.status_code)
print(response.json())
```

- **headers**: A dictionary variable that may contain additional headers to be sent with the request. In this case, no additional headers are set as the client can set it and add the boundary value.
- **files**: A dictionary variable containing the files to be attached to the request. It includes two files, **'data.json'** and **'metadata.json'**, opened in read-binary mode (**'rb'**).
- **response**: A variable that stores the response obtained from sending the POST request. The **requests.post()** function is used to perform the request, passing the URL, headers, and files as parameters.
- **print(response.status_code)**: Prints the HTTP status code received in the response. In this case, 200 is printed.
- **print(response.json())**: Prints the response content parsed as JSON.

7.2.4. Data and metadata ingestion to the SAL

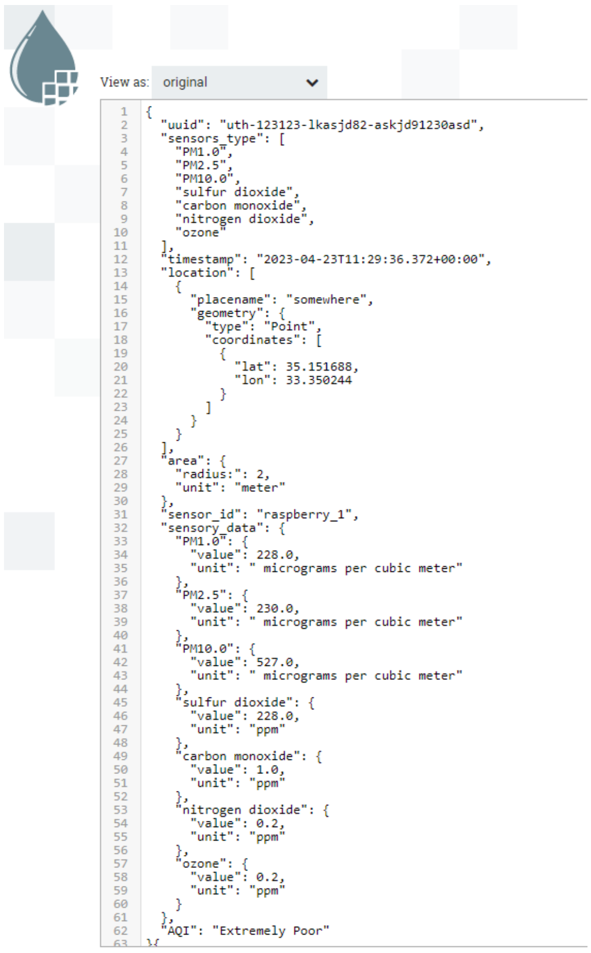
Figure 31 and Figure 32 show evidence of receiving both metadata and data successfully by the SAL.



metadata

{"descriptor":{"uuid":"uth-123123-lkasjd82-askjd91230asd","obj-class":"IoT","format":{"type":"json","resolution":"100","output":"json"},"access":"default","dataset-type":"air-quality","created":"1674574406.7829435"},"spatial":{"bbox":"POLYGON ((16.0295831170816712 41.9183734508349062, 16.0295831170816712 41.8832522880823319, 16.0872243646491313 41.8832522880823319, 16.0872243646491313 41.9183734508349062, 16.0295831170816712 41.9183734508349062))","coordinates":[{"lat":41.918373450834906,"lon":16.02958311708167},{"lat":41.88325228808233,"lon":16.02958311708167},{"lat":41.88325228808233,"lon":16.08722436464913},{"lat":41.918373450834906,"lon":16.08722436464913}],"pilot":"greek"},"temporal":{"datetime":"latest","daterange":"from:to"},"tag":{"device":"raspberry pi","sensors":["PM1.0","PM2.5","PM10.0","sulfur dioxide","carbon monoxide","nitrogen dioxide","ozone"],"AQI":true}}

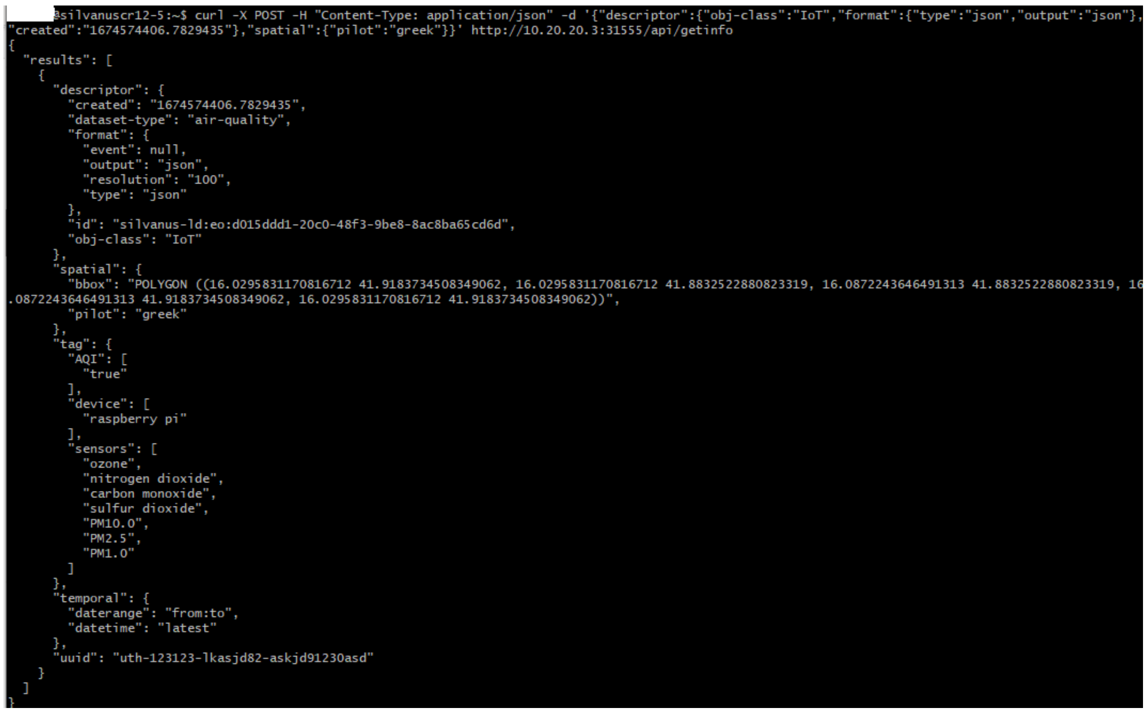
Figure 35: Screenshot shows that the metadata is successfully received by the SAL



```
{
  "uuid": "uth-123123-lkasjd82-askjd91230asd",
  "sensors_type": [
    "PM1.0",
    "PM2.5",
    "PM10.0",
    "sulfur dioxide",
    "carbon monoxide",
    "nitrogen dioxide",
    "ozone"
  ],
  "timestamp": "2023-04-23T11:29:36.372+00:00",
  "location": [
    {
      "placename": "somewhere",
      "geometry": {
        "type": "Point",
        "coordinates": [
          {
            "lat": 35.151688,
            "lon": 33.350244
          }
        ]
      }
    }
  ],
  "area": {
    "radius:": 2,
    "unit": "meter"
  },
  "sensor_id": "raspberry_1",
  "sensory_data": {
    "PM1.0": {
      "value": 228.0,
      "unit": " micrograms per cubic meter"
    },
    "PM2.5": {
      "value": 230.0,
      "unit": " micrograms per cubic meter"
    },
    "PM10.0": {
      "value": 527.0,
      "unit": " micrograms per cubic meter"
    },
    "sulfur dioxide": {
      "value": 228.0,
      "unit": "ppm"
    },
    "carbon monoxide": {
      "value": 1.0,
      "unit": "ppm"
    },
    "nitrogen dioxide": {
      "value": 0.2,
      "unit": "ppm"
    },
    "ozone": {
      "value": 0.2,
      "unit": "ppm"
    }
  },
  "AQI": "Extremely Poor"
}{
```

Figure 36: Screenshot shows that the data is successfully received by the SAL

In Figure 33, a query is depicted, directed towards the SAL, utilizing certain fields from the 'meta-data.json' file. The query's objective is to find records within the SAL that correspond to the query parameters. The results of the query display a list of matching records in the SAL. In this particular instance, a single match is found, identified by metadata containing the UUID 'uth-123123-lkasjd82-askjd91230asd'.



```
[redacted]@silvanuscr12-5:~$ curl -X POST -H "Content-Type: application/json" -d '{"descriptor":{"obj-class":"IoT","format":{"type":"json","output":"json"},
"created":"1674574406.7829435"},"spatial":{"pilot":"greek"}}' http://10.20.20.3:31555/api/getinfo
{
  "results": [
    {
      "descriptor": {
        "created": "1674574406.7829435",
        "dataset-type": "air-quality",
        "format": {
          "event": null,
          "output": "json",
          "resolution": "100",
          "type": "json"
        },
        "id": "silvanus-ld:eo:d015ddd1-20c0-48f3-9be8-8ac8ba65cd6d",
        "obj-class": "IoT"
      },
      "spatial": {
        "bbox": "POLYGON ((16.0295831170816712 41.9183734508349062, 16.0295831170816712 41.8832522880823319, 16.0872243646491313 41.8832522880823319, 16
.0872243646491313 41.9183734508349062, 16.0295831170816712 41.9183734508349062))",
        "pilot": "greek"
      },
      "tag": {
        "AQI": [
          "true"
        ],
        "device": [
          "raspberry pi"
        ],
        "sensors": [
          "ozone",
          "nitrogen dioxide",
          "carbon monoxide",
          "sulfur dioxide",
          "PM10.0",
          "PM2.5",
          "PM1.0"
        ]
      },
      "temporal": {
        "daterange": "from:to",
        "datetime": "latest"
      },
      "uuid": "uth-123123-lkasjd82-askjd91230asd"
    }
  ]
}
```

Figure 37: Submit a query for the Query interface and receives a result proved that the data is stored in the SAL

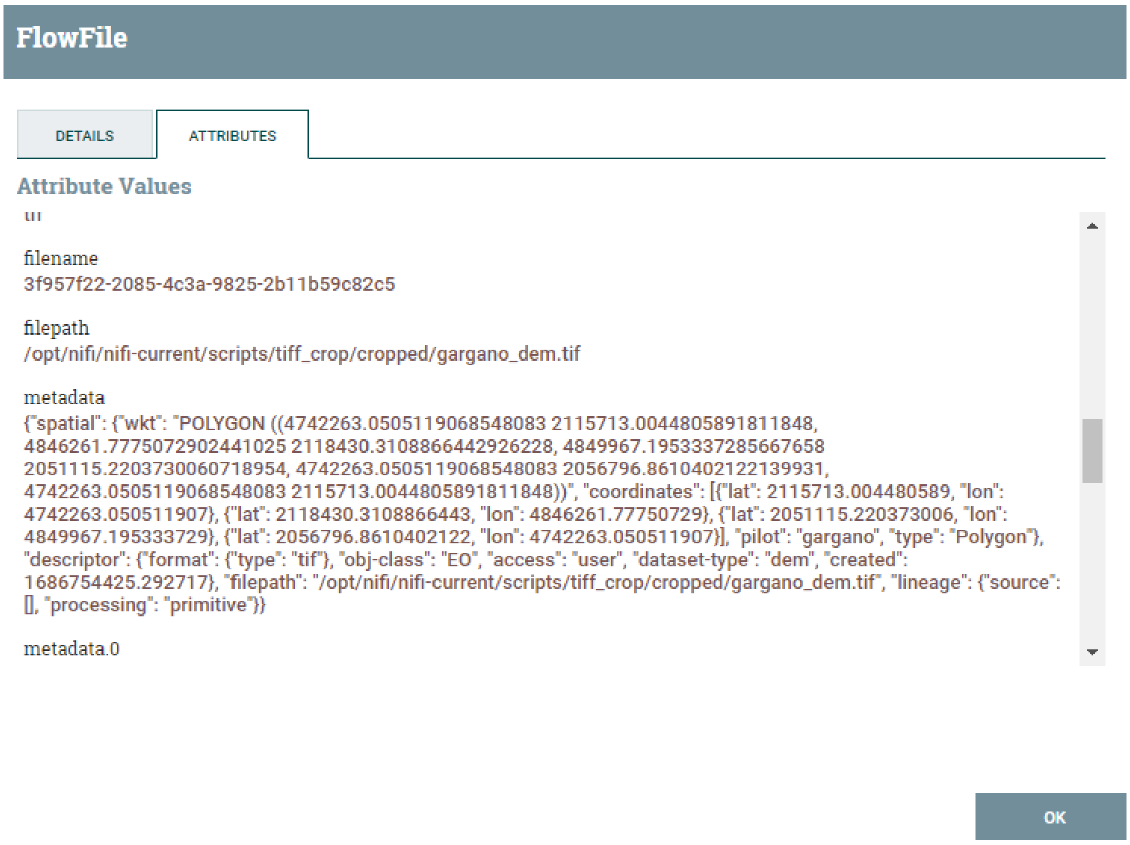
7.3. Fire Spread Model from EXUS

EXUS's Fire Spread Model service (FSM) will be deployed and run in SILVANUS cloud. Its integration with the rest of SILVANUS' components will take place through the SAL and the RabbitMQ broker. SAL with be both the data ingestion and data output node FSM. This means that on the one hand, all required input (e.g. metadata, area slope, spreading barriers etc.) will be fetched from the SAL by utilizing its exposed REST endpoints. On the other hand, FSM's output will be also stored in SAL, again through REST requests.

FSM also uses RabbitMQ broker for the notifications. Specifically, it follows the claim-check pattern described in D8.1 and Section 2.2.3.2, as the pipeline decided by the consortium, mainly for the exchange of large-size data files, which is the case for the FSM input/output data. During the data ingestion, FSM listens to dedicated RabbitMQ queues for changes in the data files within the SAL, and upon such notification, it fetches the corresponding data through the process described above. During the data output, upon FSM storing it in the SAL, it publishes the required notifications to dedicating queues for the three SILVANUS components that consume FSM's output know that there is a new output and where to find it.

The progress so far regarding the integration includes the definition of the interfaces (data format, pipelining process, exchange triggering mechanism etc.) among the FSM and the components that interconnects with, within SILVANUS cloud, as well as the partial implementation and testing of these interfaces. The partial implementation includes the data ingestion part, that is the development of the RabbitMQ subscriber listening to the notifications and the fetching of the data itself from the SAL. Next steps include the implementation of data output, i.e. making a POST request to SAL and publishing the action to the dedicated queues in RabbitMQ.

An illustrative example showcasing the integration between the SAL and FSM is depicted in the following figures. Figure 35portrays essential details, including the filename (e.g., "3f957f22-2085-4c3a-9825-2b11b59c82c5") and metadata, pertaining to the ingested data from the Gargano pilot. Moving on to Figure 36(a), it exhibits a notification message transmitted via the RabbitMQ message bus. Notably, this message incorporates relevant information, specifically under the "ccp_info" field, to facilitate the retrieval of the ingested data. Lastly, Figure 36(b) demonstrates the visualized data received during the testing phase of data ingestion from the SAL into the FSM.



uri

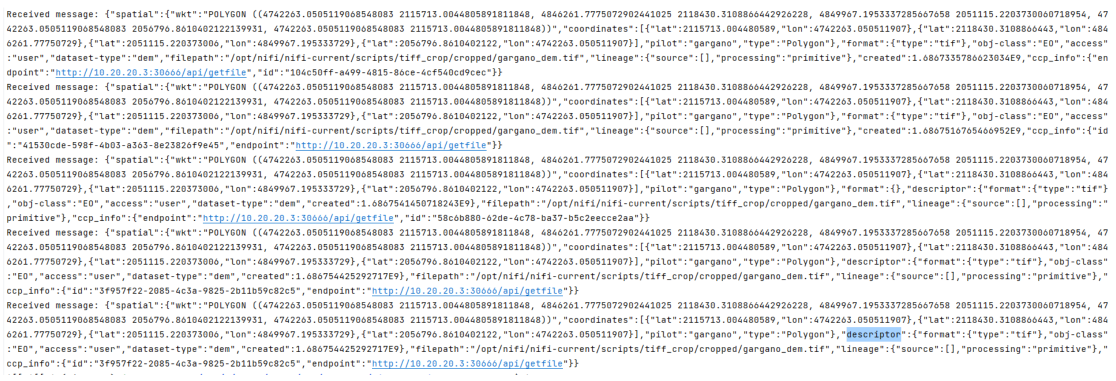
filename
3f957f22-2085-4c3a-9825-2b11b59c82c5

filepath
/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif

metadata
{"spatial": {"wkt": "POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.220373006071 8954, 4742263.0505119068548083 2056796.861040212213 9931, 4742263.0505119068548083 2115713.0044805891811848))", "coordinates": [{"lat": 2115713.004480589, "lon": 4742263.050511907}, {"lat": 2118430.3108866443, "lon": 4846261.77750729}, {"lat": 2051115.220373006, "lon": 4849967.195333729}, {"lat": 2056796.8610402122, "lon": 4742263.050511907}], "pilot": "gargano", "type": "Polygon"}, "descriptor": {"format": {"type": "tif"}, "obj-class": "EO", "access": "user", "dataset-type": "dem", "created": 1686754425.292717}, "filepath": "/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif", "lineage": {"source": [], "processing": "primitive"}}

metadata.0

Figure 38: An example of ingested data to the SAL



```
Received message: {"spatial":{"wkt":"POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.2203730060718954, 47
42263.0505119068548083 2056796.8610402122139931, 4742263.0505119068548083 2115713.0044805891811848))","coordinates":[{"lat":2115713.004480589,"lon":4742263.050511907},{"lat":2118430.3108866443,"lon":484
6261.77750729},{"lat":2051115.220373006,"lon":4849967.195333729},{"lat":2056796.8610402122,"lon":4742263.050511907}],"pilot":"gargano","type":"Polygon"},"format":{"type":"tif"},"obj-class":"EO","access"
:"user","dataset-type":"dem","filepath":"/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif","lineage":{"source":[],"processing":"primitive"},"created":1.6867335786623034E9,"ccp_info":{"en
dpoint":"http://10.20.20.3:30666/api/getfile","id":"104c50ff-a499-4815-86ce-4cf540cd9cec"}}
Received message: {"spatial":{"wkt":"POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.2203730060718954, 47
42263.0505119068548083 2056796.8610402122139931, 4742263.0505119068548083 2115713.0044805891811848))","coordinates":[{"lat":2115713.004480589,"lon":4742263.050511907},{"lat":2118430.3108866443,"lon":484
6261.77750729},{"lat":2051115.220373006,"lon":4849967.195333729},{"lat":2056796.8610402122,"lon":4742263.050511907}],"pilot":"gargano","type":"Polygon"},"format":{"type":"tif"},"obj-class":"EO","access"
:"user","dataset-type":"dem","filepath":"/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif","lineage":{"source":[],"processing":"primitive"},"created":1.6867516765466952E9,"ccp_info":{"id
":"41530cde-598f-4b03-a363-8e23826f9e45","endpoint":"http://10.20.20.3:30666/api/getfile"}}
Received message: {"spatial":{"wkt":"POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.2203730060718954, 47
42263.0505119068548083 2056796.8610402122139931, 4742263.0505119068548083 2115713.0044805891811848))","coordinates":[{"lat":2115713.004480589,"lon":4742263.050511907},{"lat":2118430.3108866443,"lon":484
6261.77750729},{"lat":2051115.220373006,"lon":4849967.195333729},{"lat":2056796.8610402122,"lon":4742263.050511907}],"pilot":"gargano","type":"Polygon"},"format":{},"descriptor":{"format":{"type":"tif"}
,"obj-class":"EO","access":"user","dataset-type":"dem","created":1.6867541450718243E9},"filepath":"/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif","lineage":{"source":[],"processing":"
primitive"},"ccp_info":{"endpoint":"http://10.20.20.3:30666/api/getfile","id":"58c6b880-62de-4c78-ba37-b5c2eecce2aa"}}
Received message: {"spatial":{"wkt":"POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.2203730060718954, 47
42263.0505119068548083 2056796.8610402122139931, 4742263.0505119068548083 2115713.0044805891811848))","coordinates":[{"lat":2115713.004480589,"lon":4742263.050511907},{"lat":2118430.3108866443,"lon":484
6261.77750729},{"lat":2051115.220373006,"lon":4849967.195333729},{"lat":2056796.8610402122,"lon":4742263.050511907}],"pilot":"gargano","type":"Polygon"},"descriptor":{"format":{"type":"tif"},"obj-class"
:"EO","access":"user","dataset-type":"dem","created":1.686754425292717E9},"filepath":"/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif","lineage":{"source":[],"processing":"primitive"},"
ccp_info":{"id":"3f957f22-2085-4c3a-9825-2b11b59c82c5","endpoint":"http://10.20.20.3:30666/api/getfile"}}
Received message: {"spatial":{"wkt":"POLYGON ((4742263.0505119068548083 2115713.0044805891811848, 4846261.775072902441025 2118430.3108866442926228, 4849967.1953337285667658 2051115.2203730060718954, 47
42263.0505119068548083 2056796.8610402122139931, 4742263.0505119068548083 2115713.0044805891811848))","coordinates":[{"lat":2115713.004480589,"lon":4742263.050511907},{"lat":2118430.3108866443,"lon":484
6261.77750729},{"lat":2051115.220373006,"lon":4849967.195333729},{"lat":2056796.8610402122,"lon":4742263.050511907}],"pilot":"gargano","type":"Polygon"},"descriptor":{"format":{"type":"tif"},"obj-class"
:"EO","access":"user","dataset-type":"dem","created":1.686754425292717E9},"filepath":"/opt/nifi/nifi-current/scripts/tiff_crop/cropped/gargano_dem.tif","lineage":{"source":[],"processing":"primitive"},"
ccp_info":{"id":"3f957f22-2085-4c3a-9825-2b11b59c82c5","endpoint":"http://10.20.20.3:30666/api/getfile"}}
```

(a)

(b)

Figure 39. (a) Logs of successful notification receival by RabbitMQ subscriber and (b) visualized data received during the testing of data ingestion from the SAL into the FSM

8. Conclusion

The deliverable has summarised the overall contribution of WP5 activities towards enriching the operational capabilities of SILVANUS project. The demonstrations of the different algorithms and scientific development has been successfully presented during the technical meeting and the feedback from external experts have been gathered.

The activities of WP5 will continue until the end of the project duration and thus, the activities reported in the deliverable will continue to mature with the availability of new datasets and improvements to the algorithms. Subsequent research results will be presented in the next deliverable.