Disaster Response and Coordination System

- Team 3:
  - Gino Biasioli, gino.biasioli@studenti.unitn.it
  - Ettore Miglioranza, ettore.miglioranza@studenti.unitn.it
  - Tommaso Cestari, tommaso.cestari@studenti.unitn.it

- Big Data Technologies, UniTN, 2024/2025
- Date of Submission: 10/06/2025

Abstract (Overview)

- The project is a real-time disaster response system that integrates data from meteorological data, IoT sensors, satellite images and social media messages to monitor and assess wildfire emergencies. It uses Kafka and Flink to process streaming data, Redis and MinIO for fast access and storage, Streamlit to visualize alerts, sensor readings, fire behavior and recommended actions
- **Key objectives**:
  - Create a real-time wildfire monitoring system using multiple data sources
  - Process ingested data to provide emergency agencies with valuable insights and enable better-informed decision-making during wildfire events
- **Main results achieved:**
  - An early warning system designed to predict potential disaster events weeks in advance and automatically activate the live streaming monitoring system if needed.
  - A streaming pipeline using Kafka, Flink, and Redis to process real-time sensors and alert data and MinIO and PostgreSQL to add storage and retrieval layers
  - A dashboard that dynamically displays wildfire detection status, severity scores, environmental conditions and delivers actionable recommendations for emergency responders

Problem Statement & Motivation

- Wildfires in California are growing more frequent, destructive, and deadly. Fifteen of the 20 largest wildfires in state history have occurred since 2000, and 10 of the most damaging fires to life and property have happened since 2015 (California Dept. of Fish and Wildlife).

- In January 2025 alone, wildfires across the Los Angeles area caused the deaths of at least 30 people, displaced thousands, and destroyed or damaged over 17,000 structures (World Vision). Estimated economic losses reached $30 billion (ABC News, Wells Fargo, Goldman Sachs)

- These trends underscore the urgent need for a more effective and coordinated response system that ensures emergency services have accurate and actionable information to protect lives, communities, and infrastructure.

Data exploration insights

- Satellite images and data
  - Images are retrieved from the Copernicus Sentinel-2 mission using the Sentinel Hub API
  - The values for the bands associated with each image are synthetically generated

- IoT sensors
  - The data received by the IoT sensors is synthetically generated to reflect realistic wildfire conditions, based on reference values and measurements from actual wildfire sensor deployments
  - To select the measurements for synthetic data generation, academic papers and real-world sensors were reviewed to ensure that the simulated values can reflect these measurements in actual wildfire monitoring scenarios

Data exploration insights

- Social messages
  - Generated using a set of customizable templates, each containing placeholders and interchangeable synonyms to simulate linguistic variability and mimic real-world reporting styles


```
unique_msg_id: "A1-M10_2025-06-09_12:55:11.948"
macroarea_id: "A1"
microarea_id: "M10"
text: "I'm really uneasy about what's happening out there near the north."
latitude: 38.76803219263364
longitude: -122.07637872742539
timestamp: "2025-06-09T12:55:11.948"
```

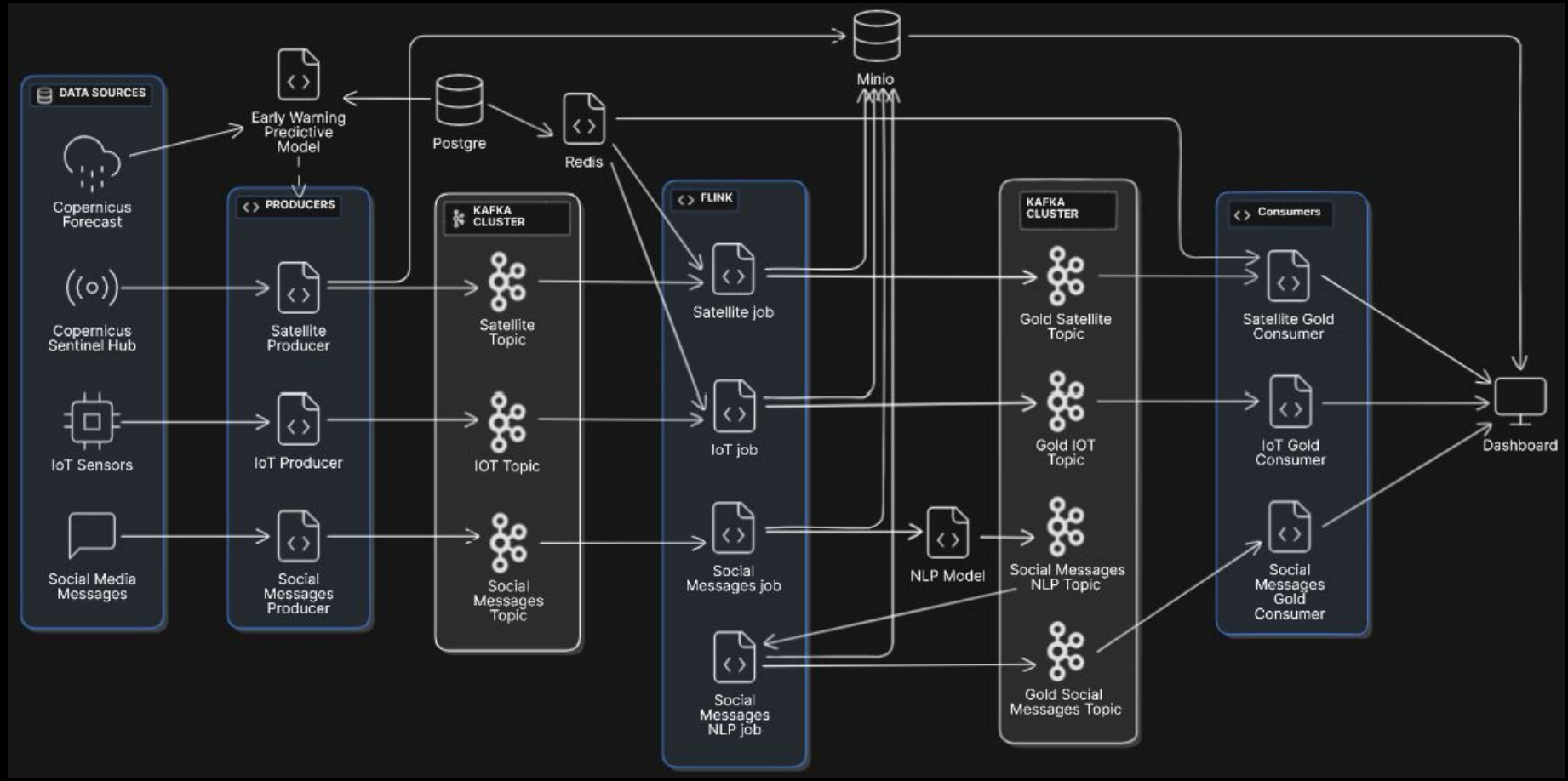

- Meteorological data
  - Two datasets (fire detection historic data and meteorological historic data) downloaded as CSVs from the Copernicus Climate Data Store
  - Merged and processed to have a unique dataset used to train the predictive model

Fire Detection Data	Meteorological Data
- Location: lat, lon	• Location: lat, lon
- Time: acq_date, acq_time	• Time: time, valid_time
- Fire properties: brightness, frp, bright_t31	• Wind: u10, v10, u100, v100 • Temperature: t2m, d2m, sst, stl1
- Detection: scan, track, confidence, satellite, instrument, version	• Pressure: msl, sp • Moisture: swvl1, cvh • Forecast: number, step, surface, depthBelowLandLayer
- Context: daynight, type	

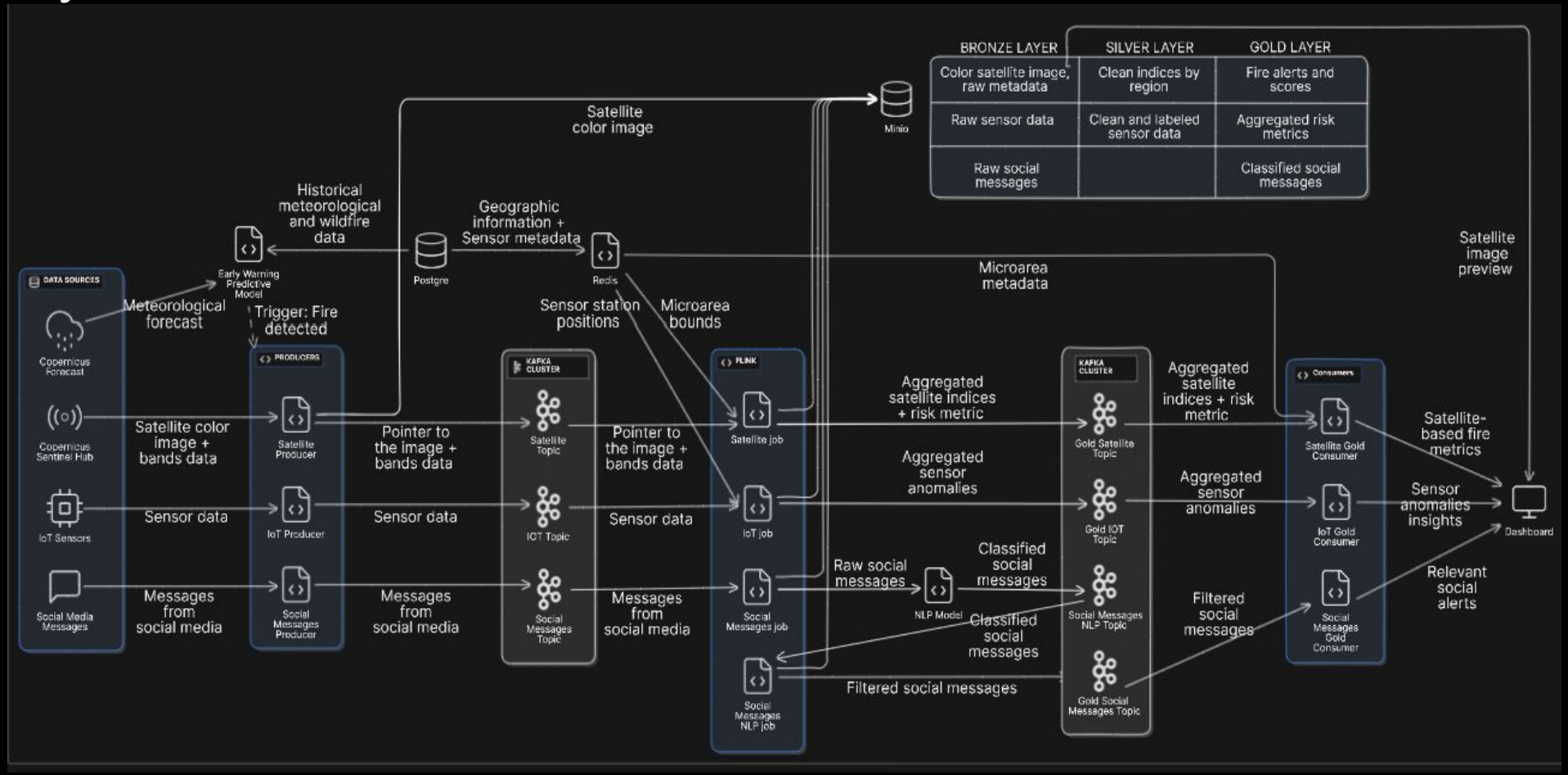
Volumes per microarea, standard streaming would involve from 50 to 500/1000 micro areas:

Data Source	Data Format	Kafka /Flink	Memory	Frequency	Volume per minute	Volume per hour
Satellite Images	.Jpeg	X	~15 Mb (Compression to ~700 Kb for writing)	1Jpeg/min	~15 Mb	~1 Gb
Satellite raw data	.Json	✓	~ 5 Kb	1Json/min	~ 5 Kb	~300Kb
Social raw msg	.Json	✓	~ 300 Bytes	1Json/second - 100Jsons/second	~ 18 Kb/ ~18 Mb	~ 1 Mb / ~ 1 Gb
IoT raw data	.Json	✓	~ 250 Bytes	~80Jsons/5 seconds	~240 Kb	~ 14 Mb
Satellite gold data	.parquet	✓	~ 46,6 Kb	1parquet/min	~ 46,6 Kb	~ 3 Mb
Social gold data	.parquet	✓	~ 8,81 Kb	1parquet/second - 100 parquets/second	~ 530 Kb /~52Mb	~ 31 Mb /~3Gb
IoT gold data	.parquet	✓	~ 36,6 Kb	1parquet/min	~ 36,6 Kb	~ 2 Mb

System Architecture



System Architecture /2



Technologies & Justifications

Component	Technologies	Justification
Message broker	Kafka, Zookeeper	Kafka enables efficient and decoupled communication between highly heterogeneous data sources (satellite, sensors, social media). Zookeeper ensures cluster coordination and reliability, essential for maintaining a robust pipeline under load.
Stream processing engine	Apache Flink	Chosen for its ability to process continuous streams from multiple sources with low latency. Its windowing and state management features allow real-time aggregation, filtering, and classification, making it ideal for event-driven emergency monitoring.
Caching In-memory	Redis	Provides rapid access to dynamic values such as real-time alert levels or temporary aggregates. It reduces pressure on the database and ensures fast retrieval for components like the dashboard and Flink.
Historical database	PostgreSQL	Stores enriched and aggregated historical data (e.g., past fire events, meteorological conditions), supporting both structured querying and spatial analysis for deeper insights and model training.
Object storage	MinIO	Used to store large, heterogeneous, and unstructured data like raw satellite images and gold-layer outputs. Its S3 compatibility and lightweight deployment make it perfect for a containerized, local big-data environment.

Technologies & Justifications

Component	Technologies	Justification
Predictive Model	Random forest	A robust and interpretable ML model for predicting fire risk using multiple numerical weather variables. Efficient to train and test in limited time, well-suited for structured data and supports rapid iteration and retraining.
NLP microservice- Messages Classification	FastAPI, Scikit-learn, transformers	The NLP microservice was deployed using the RESTful FastAPI package to allow us to expose prediction endpoints for downstream consumers in real-time. Scikit-learn supports fast prototyping of lightweight classifiers, while Transformers allow leveraging semantic understanding of social media messages. Together, they handle both structured and contextual features across noisy, real-time data.
Containerization	Docker compose	Guarantees reproducibility and portability of the entire multi-service architecture, enabling fast local deployments, testing, and collaboration across different development setups.
Dashboard	Streamlit	Allows the rapid creation of interactive dashboards that combine map visualizations, real-time social alerts, and statistics, essential for monitoring diverse regions and data types.

Implementation & Code Repository

- https://github.com/BDT-Team3/DRCS_1.0.git
- Docker Images: python:3.10-slim, python:3.9, postgres:14, redis:7-alpine, confluentinc/cp-zookeeper:7.5.0, confluentinc/cp-kafka:7.5.0, minio/minio:latest, flink:1.17-scala_2.12.
- DRCS is a real-time emergency management platform that ingests and processes multi-source data — including satellite images, IoT sensors, social media, and weather forecasts — to generate early warnings and visualize disaster impact zones. It supports rapid, data-driven decisions via an interactive dashboard.
- Key Modules & Pipelines:
  - Data Ingestion: img_producer/, sens_producer/, msg_producer/ → Kafka topics.
  - Real-Time Processing (Flink Jobs): satellite_flink_job/, iot_flink_job/, social_flink_job/.
  - Early Warning: early_warning_model/ forecasts risk levels at startup.
  - Dashboard: dashboard/ visualizes sensor alerts, satellite NDVI, social signals.
  - Storage: Postgres for geographic informations, MinIO as a multi-layer Data Lakehouse (bronze, silver, gold).
  - Setup & Orchestration: setup_orchestrator/ processes GeoJSON and micro areas.
- Getting started
  - Clone repo: git clone https://github.com/BDT-Team3/DRCS_1.0.git.
  - Add Sentinel Hub credentials in config.toml.
  - Download required JARs for Flink Kafka integration
  - (If on Apple Silicon): export DOCKER_DEFAULT_PLATFORM=linux/amd64.
  - Start system: bash ./entrypoint.sh
  - Dashboard: http://localhost:8501, MinIO UI: http://127.0.0.1:9001 (user/pass: minioadmin)

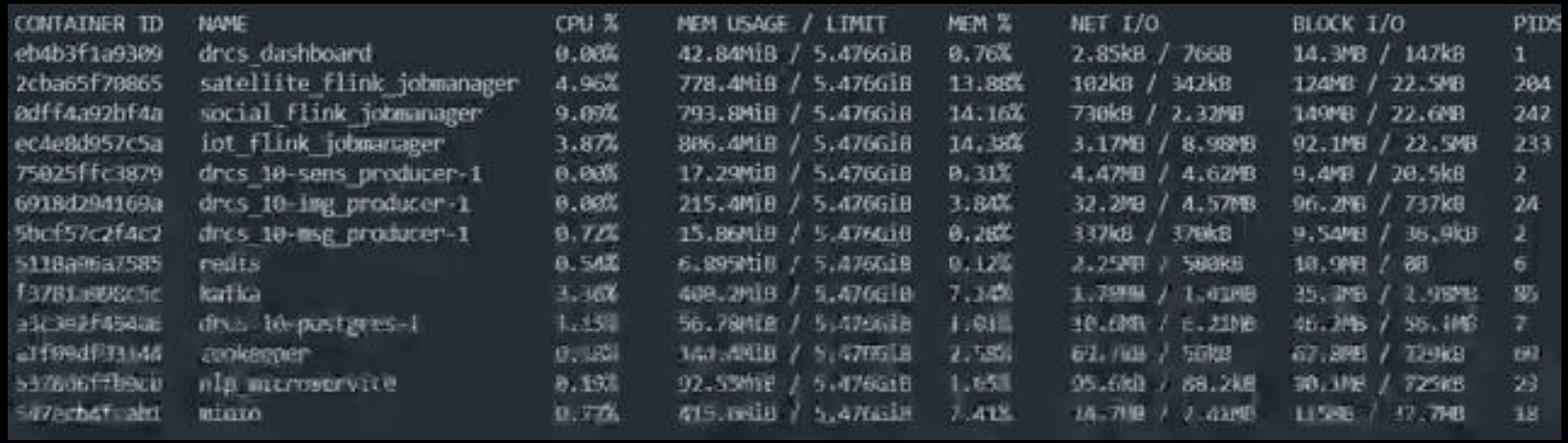
Results & Performance

- Processing latency between event timestamp (when the raw data payload is fetched/generated) vs response timestamp (when the final gold payload is sinked to minIo and dashboard):

Data Type	**Event Timestamp**	**Response Timestamp** **(taken during sink operations)**	**Latency (ms)**	**Notes**
Satellite Image	2025-06-08T12:36:45.665	2025-06-08T12:36:48.908	3243 (OK)	Image stream processing delay
Sensor aggregated measurements	1749385855769 (ms) (**latest timestamp among any measurements fetched and aggregated within window interval)	1749385873510 (ms)	17741 (very high latency, **bottleneck**) **Further analysis required	IoT stream processing delay
Social Message	2025-06-08T12:22:04.500	2025-06-08T12:22:10.056	5556 (Acceptable? given NLP/FastAPI inference)	Social msg stream processing and nlp classification delay

Results & Performance

- Docker stats on 12 Gb RAM - 12 Core CPU:



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
eb4b3f1a9309	drcs_dashboard	0.00%	42.84MiB / 5.476GiB	0.76%	2.85kB / 766B	14.3MB / 147kB	1
2cba65f70865	satellite_flink_jobmanager	4.96%	778.4MiB / 5.476GiB	13.88%	102kB / 342kB	124MB / 22.5MB	204
0dff4a92bf4a	social_flink_jobmanager	9.09%	793.8MiB / 5.476GiB	14.16%	730kB / 2.32MB	149MB / 22.6MB	242
ec4e8d957c5a	iot_flink_jobmanager	3.87%	806.4MiB / 5.476GiB	14.38%	3.17MB / 8.98MB	92.1MB / 22.5MB	233
75025ffc3879	drcs_10-sens_producer-1	0.00%	17.29MiB / 5.476GiB	0.31%	4.47MB / 4.62MB	9.4MB / 20.5kB	2
6918d294169a	drcs_10-img_producer-1	0.00%	215.4MiB / 5.476GiB	3.84%	32.2MB / 4.57MB	96.2MB / 737kB	24
5bcf57c2f4c2	drcs_10-msg_producer-1	0.72%	15.86MiB / 5.476GiB	0.28%	337kB / 376kB	9.54MB / 36.9kB	2
5118a96a7585	redis	0.54%	6.895MiB / 5.476GiB	0.12%	2.25MB / 580kB	10.9MB / 0B	6
f3781a8d8c5c	kafka	3.36%	408.2MiB / 5.476GiB	7.34%	1.79MB / 1.41MB	35.3MB / 2.98MB	95
a1c3e2f454ac	drcs_10-postgres-1	1.15%	56.78MiB / 5.476GiB	1.01%	10.6MB / 6.21MB	46.2MB / 56.1MB	7
a1f09df31a44	zookeeper	0.12%	144.4MiB / 5.476GiB	2.58%	67.7kB / 56kB	67.2MB / 229kB	60
b378d6ffbacb	nlp_microservice	0.13%	92.53MiB / 5.476GiB	1.65%	95.6kB / 88.2kB	90.1MB / 725kB	23
5d7ecb6fcabd	minio	0.77%	415.6MiB / 5.476GiB	7.41%	14.7MB / 7.41MB	11.5MB / 47.7MB	18

Unit	Meaning	Size in Bytes
MiB	Mebibyte	1,048,576 bytes (2^{20})

Snapshot min 0:01 evaluation:

Container	MEM Usage	% of Limit	Notes
iot_flink_jobmanager	806.4 MiB	14.38%	Highest usage; stateful stream tasks (e.g., aggregations, timers)
social_flink_jobmanager	793.8 MiB	14.16%	Also high; NLP + state may increase memory footprint
satellite_flink_jobmanager	778.4 MiB	13.88%	Also high; satellite image processing and NDVI calcs likely contribute
nlp_microservice	92.55 MiB	1.65%	Lightweight; likely idle when not called
drcs_dashboard	42.84 MiB	0.76%	Very low — efficient Streamlit frontend
producers	15–215 MiB	<4%	All low, as expected from synthetic data generation
minio	415.8 MiB	7.42%	Acceptable; writing/serving images and parquet data
kafka	400.2 MiB	7.14%	In line with expected use for a broker under moderate load
postgres	56.78 MiB	1.01%	Lightweight; used for metadata/logs
redis	6.89 MiB	0.12%	Very lightweight; good sign for orchestrator state caching

Container	Net I/O	Notes
IoT_flink_jobmanagers	~3.17 MB / 8.98 MB (Iot),	Both ingestion and responses from and to Kafka
Sat_ flink_jobmanagers	~100Kb/300Kb	Low throughput ratio, payload (pointer to image + metadata) processed each minute. Ingestion and sink to kafka.
social_flink_jobmanager	730 KB / 2.32 MB	High output-to-input ratio — NLP results back to Kafka
img_producer	32.2 MB / 4.57 MB	Downloads satellite images (input-heavy)
minio	14.7 MB / 2.41 MB	Accepting writes (bronze layer) + some dashboard reads
Others	Mostly under 2 MB total	Expected for idle/light containers

Container	PID Count (Process IDs)	Notes
flink_jobmanagers	~233–242	Normal for multi-threaded streaming jobs
kafka	85	Typical broker threads, stable
nlp_microservice	69	FastAPI + worker pool (e.g., uvicorn + asyncio)
Others	<10	Mostly idle or simple task containers

Lessons Learned

The challenges encountered are listed in order of difficulty and time investment:

Technical Stack Familiarization
The biggest challenge was interacting with a complex tech stack, as none of us had prior experience with technologies like Kafka, Flink, or MinIO. Implementing the system required extensive study and countless hours spent debugging Java errors, resolving incompatibilities between components, and researching the most appropriate setup. This phase demanded deep technical perseverance.

Integration with Real-World Data Sources
The second major challenge involved working with real-world scientific data from platforms like Copernicus Data Space and Sentinel Hub API. Fetching data in a streaming fashion required handling complex geospatial formats and understanding how to programmatically interface with these systems. This task was both technically and conceptually demanding.

Problem Definition and Data Modeling
Another key challenge was defining the problem itself: choosing the right data, modeling it appropriately, and selecting a realistic study case that would still allow for general-purpose system design. While grounding the system in a real-life scenario (e.g., California wildfires) was helpful at first, we quickly encountered inconsistencies between our design goals and what the data, APIs, and available hardware actually allowed us to implement. This led to a continuous cycle of revising our modeling approach and refining earlier design decisions.

Synthetic Data Generation
Since much of the needed data was either unavailable or outdated, we had to design a system for generating synthetic data that could realistically simulate a disaster event. This was especially difficult because the fake data needed to be scientifically coherent, geographically plausible, and operationally usable across multiple modules. Maintaining consistency across all simulated data sources added an extra layer of complexity.

Team Coordination
Ensuring effective collaboration within the team was also a challenge. We needed to distribute the workload in a way that allowed everyone to contribute independently, while still staying aligned with the global vision. That said, we feel we succeeded here—each member left a distinct and invaluable mark on the project, contributing irreplaceable solutions to the various problems we faced.

Balancing Ambition with Feasibility
Perhaps the hardest lesson was accepting that, due to time and resource constraints, we often had to compromise between what was ideal and what was achievable. We regularly had to scale back or adjust our plans, hoping that our implemented solutions were “good enough” to meet the system’s goals without sacrificing stability or clarity.

Limitations

- **Sentinel Hub API Rate Limits**
  - **Current Setup**: Fetches satellite imagery every minute per microarea
  - **Scaling Problem**: Sentinel Hub APIs have strict rate limits and quota restrictions, adding multiple micro areas would quickly exceed API quotas
- **Single-Node Kafka Architecture**
  - **Current Setup**: Single Kafka broker handling all topics -> Network I/O saturation
  - Message lag, dropped messages, broker crashes
- **High-Risk Bottleneck in NLP HTTP Integration**
  - **Current Setup**: Synchronous HTTP calls to FastAPI NLP microservice for social media processing
  - **Scaling Problem**: Flink tasks **block on synchronous HTTP calls**, leading to queued tasks under load, using FastAPI's async capabilities doesn't help if Flink itself waits for responses, windowing (batching) helps mitigate, but doesn't eliminate the **core blocking issue**.
- **NLP model is a placeholder:** Given the limited number of training sentences and TF-IDF's inability to capture contextual or semantic meaning, it was adopted as a lightweight placeholder during prototyping, with plans to replace it with a more generalizable deep learning model in the future.
- **Flink Resource Constraints**
  - **Current Setup**: Limited task managers for Flink cluster
  - **Scaling Problem**: Memory pressure from larger state sizes and increased throughput, low parallelism set due to resources constraints
  - **Impact**: Processing latency increases, potential job failures under extreme load
- **Lack of Auto-Scaling**
  - **Problem**: Static container deployment via docker-compose
  - **Scaling Issues**: Manual intervention required for adding resources
- **Iot Flink Job latency bottleneck:** Currently experiencing very high latency in iot computations of the gold data sinked to the dashboard with just one microarea, this could grow exponentially with more region introduced.
- **Timestamp timezone issue:** We used UTC timestamps, which were not synchronized with California time. Attempting to change this caused several formatting issues, so due to time constraints, we decided not to update it.

Future Work

Architecture & Stream Processing: 1) Replace custom NLP microservice with Flink-native sentiment analysis or improve the already present decoupling via Kafka: send NLP requests/results through Kafka topics to fully separate Flink from the HTTP service. 2) Use proper `addSink()` methods instead of `map()` in Flink jobs. 3) Enable dynamic scaling based on the number of monitored micro areas. 4) Improving window management/watermark strategy and solving critical bottleneck latency recorded in Iot (17 seconds delta between response and event ts). 5) Changing aggregated computations in window processing function to stateful operations.

Simulation Enhancements: 1) Improve wildfire simulation on satellite images (e.g., heatmap overlays). 2) Generate more realistic sensor and social media data.

Geospatial Management: Move hard coded geospatial data from `data_templates.py` to a centralized PostgreSQL

Early Warning System: 1) Integrate Copernicus Weather APIs (forecast and historical). 2) store predictions for retraining; expose model confidence. 3) implement MLOps (Champion–Challenger) for both fire and NLP models.

Container Orchestration: 1) Dynamically generate Docker Compose based on the number of micro areas. 2) Enable dynamic scaling of Kafka, Flink, and dashboard instances accordingly.

Dashboard & Monitoring: 1) Integrate early warning scores into the dashboard. 2) Add monitoring for Sentinel Hub usage and system health. 3) Change current dashboard with dynamic creation of multiple instances according to micro areas or handle multi micro area processing with one dashboard instance.

Data Management: 1) Use Delta Lake or Apache Iceberg with partitioning and time-travel. 2) Understand how to use flink connectors to delta table formats, hdfs for S3/MinIO and apache Iceberg with Hive metastore catalog.

References & Acknowledgments

- List of papers, datasets, tools used (cite main references)
  - IoT sensors - papers and real-world sensors
    - De Rango A, Furnari L, Cortale F, Senatore A, Mendicino G. Wildfire Early Warning System Based on a Smart CO2 Sensors Network. Sensors. 2025; 25(7):2012. https://doi.org/10.3390/s25072012
      Sensor : Milesight EM500-CO2
    - Barkjohn, K. K., Holder, A. L., Frederick, S. G., & Clements, A. L. (2022). Correction and Accuracy of PurpleAir PM2.5 Measurements for Extreme Wildfire Smoke. Sensors, 22(24), 9669. https://doi.org/10.3390/s22249669
      Sensor : PurpleAir PA-II
    - Martínez-de Dios JR, Merino L, Caballero F, Ollero A**.** Automatic Forest-Fire Measuring Using Ground Stations and Unmanned Aerial Systems. *Sensors*. 2011; 11(6):6328–6353. https://doi.org/10.3390/s110606328
      Sensor : FLIR ThermoCAM P20

- Acknowledge any mentors, contributors, or external sources that helped