Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Cloud Computing for Big Data Analytics
Course Project – Final Report

InmoDecision: A Data-Driven Advisor for Real Estate Investment with Social Sentiment Analysis

Students:
Gabriel Guerra
Alexis Vendrix
Joan Rodríguez
Nashly González

Professor:
Angel Toribio

May 25, 2025

Table of Contents

1. Introduction....3
2. What does the project finally do?....3
2.1 Functionalities....4
3. Scope Changes....5
4. System Architecture and Justifications....7
4.1. AWS Cloud Services....7
4.1.1. Amazon Elastic Container Registry (ECR)....7
4.1.2. Amazon Simple Storage Service (S3)....8
4.1.3. Amazon Virtual Private Cloud (VPC)....9
4.1.5. AWS Fargate....11
4.1.6. Amazon CloudWatch Logs....11
4.1.7. AWS Identity and Access Management (IAM)....12
4.1.8. AWS CloudFormation....13
4.1.9. AWS Elastic Beanstalk....13
4.2. Open Data Sources....15
4.3. Programming Languages, Frameworks, Libraries and Additional Tools....15
5. Use of the Twelve-Factor Methodology....18
6. Development Methodology....19
6.1. Task Breakdown....19
6.2. Gantt Chart....20
7. Problems and Solutions....21
8. Time Investment and Deviations....22
9. Conclusions and Future Work....22
10. References....22

1. Introduction

InmoDecision is a data-driven web application designed to assist individuals in making smarter real estate investment decisions. By combining open housing market data with real-time sentiment analysis from social media platforms, the tool provides a more complete view of property opportunities—not only based on prices and location, but also on public opinion.

The project focuses on building a functional prototype using Streamlit, integrated with datasets from platforms such as Open Data Barcelona, Idealista, and sentiment data from sources like Bluesky. The geographic scope is limited to a specific city to ensure feasibility within the academic timeline.

Born from the need to simplify the property decision-making process, InmoDecision aims to support first-time buyers, retail investors, and families through an accessible, intuitive platform. Ultimately, the project demonstrates how open data and AI techniques can be combined in the cloud to deliver socially valuable solutions.

2. What does the project finally do?

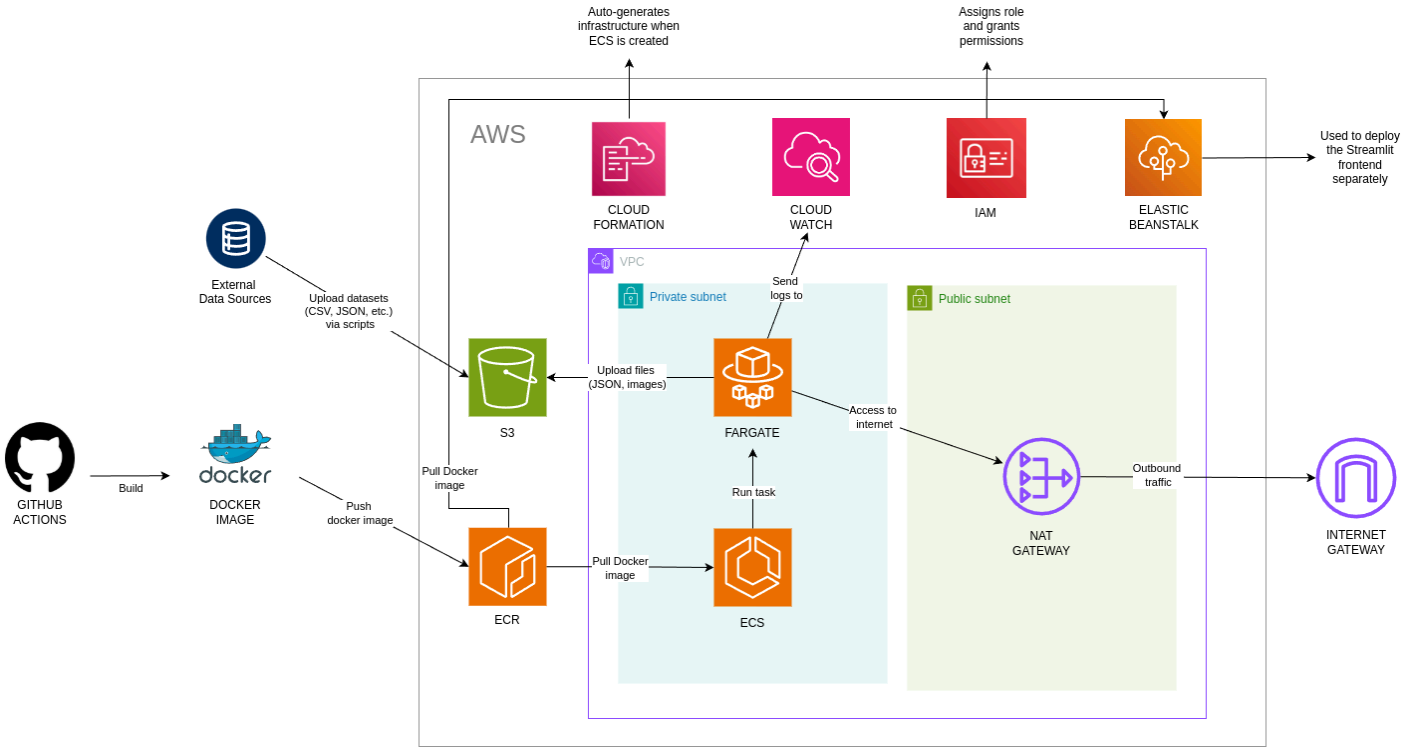

Figure 1. Final Cloud Architecture: Real-Time and Batch Processing with AWS Services

The final project, InmoDecision, is a full-stack, cloud-native application that integrates open data, social media sentiment, and data science techniques to assist users in making informed real estate investment decisions in Barcelona. It leverages the power of AWS cloud services to ingest, process, store, and analyze both static and streaming datasets, while also offering a frontend interface built with Streamlit.

At the core of the system are two primary workflows: the ingestion and processing of external datasets and the real-time sentiment analysis of social media content. Datasets from public sources such as Open Data Barcelona, Idealista, and others were uploaded to Amazon S3 using Python scripts. These include structured, semi-structured, and unstructured data in formats like CSV and JSON. Once in the cloud, the data was used for modeling, visualization, and inference.

For real-time processing, we implemented a data streaming pipeline using Amazon ECS with AWS Fargate. A containerized worker application built in Python collects posts from the Bluesky platform, performs sentiment analysis using natural language processing (NLP), and saves results as JSON files in S3. This worker runs as an ECS task using a Docker image built via GitHub Actions and pushed to Amazon ECR. Fargate executes the task in a private subnet, and thanks to a NAT Gateway, it has outbound internet access to fetch external data while keeping the container isolated from direct exposure.

To manage and monitor the application, CloudWatch Logs collects task execution logs for debugging and observability, and IAM roles grant secure access between ECS, ECR, S3, and other services. The infrastructure — including VPCs, subnets, roles, and ECS clusters — was provisioned automatically via AWS CloudFormation, which is triggered when setting up ECS services through the AWS Console.

On the frontend, the Streamlit application was containerized separately using Docker and deployed on AWS Elastic Beanstalk, allowing us to decouple the web interface from the backend processing services. The dashboard allows users to explore market trends and sentiment insights interactively.

This architecture ensures scalability, fault isolation, and modularity, in alignment with cloud-native principles. The application demonstrates how open data and cloud computing can be combined to build socially valuable solutions with minimal operational overhead.

2.1 Functionalities

- Price predictor: Based on the collected listing, a price prediction model was created to predict prices based on a property's features. This predictor is use in our application to tell if a given property is above or under the market price.A price prediction model was developed using collected listing data. This model predicts property prices based on their

features and is integrated into our application to determine if a property's price is above or below market value.

- Real estate investment plan: In our application, while selecting a property and filling some key financial data, we are showing future cash flows to help the user to determine if an investment is profitable or not.Our real estate investment application helps users assess property profitability by projecting future cash flows based on selected properties and key financial inputs. This feature aids in determining investment viability.

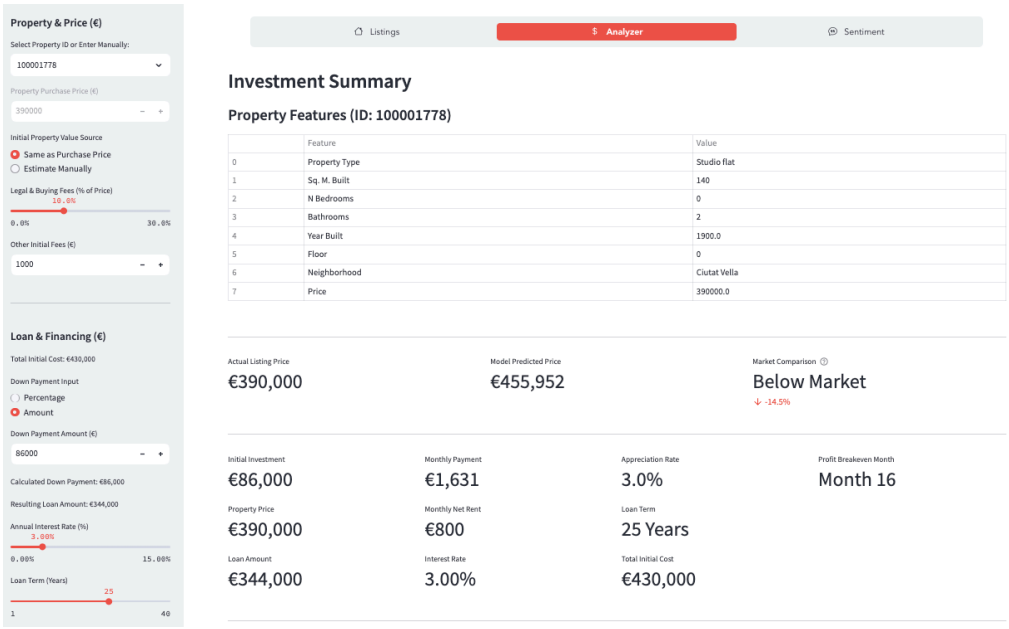

Figure 2. Streamlit App - Investment dashboard

- Real estate property listings explorer: An interactive map of the city of Barcelona is displayed, showing the locations of a set of properties. Users can apply custom filters such as price, number of rooms, bathrooms, and other parameters. The interface also allows users to visualize statistical information, such as price distributions and the top neighborhoods by number of listings.

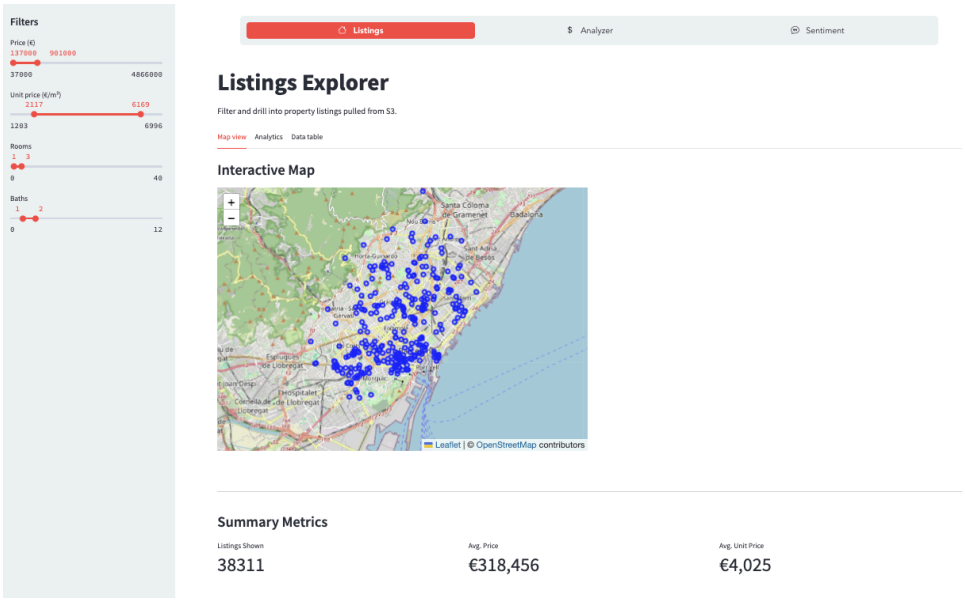

Figure 3. Streamlit App - Listing analysis

- Sentiment Analysis: Analyzing recent Bluesky posts from real estate accounts, this report aims to determine the current sentiment surrounding the real estate market. By examining posts originating from real estate related accounts, this study seeks to ascertain the prevailing sentiment regarding the current state and future outlook of the real estate market. The findings of this analysis will offer valuable insights into the collective perceptions and attitudes of real estate professionals and potentially related stakeholders actively participating in online discussions.

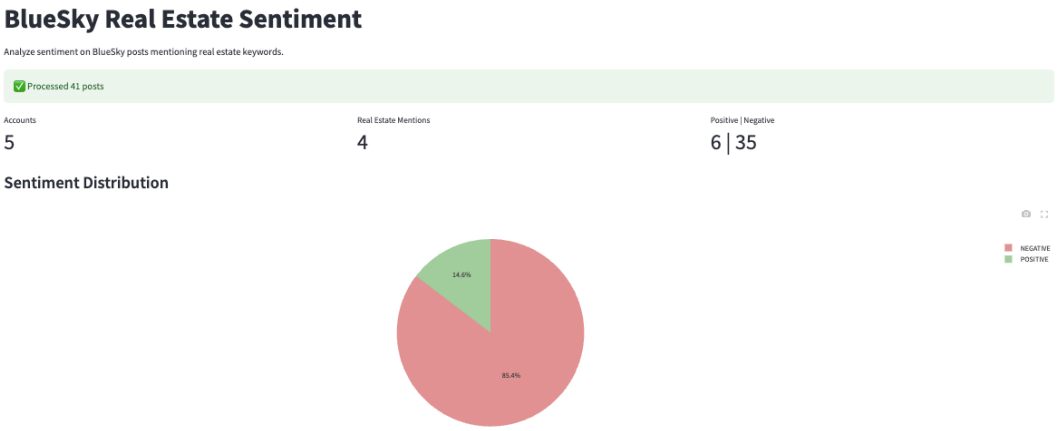

Figure 3. Streamlit App - Sentiment analysis

3. Scope Changes

At the beginning of the project, our architectural vision was broad but unclear. The initial draft mentioned EC2 for hosting the frontend and included the possibility of using Kafka for data ingestion, but lacked a defined cloud deployment strategy and technical details. Our first architecture diagram reflected a general concept with many open questions around orchestration, data handling, and infrastructure setup.

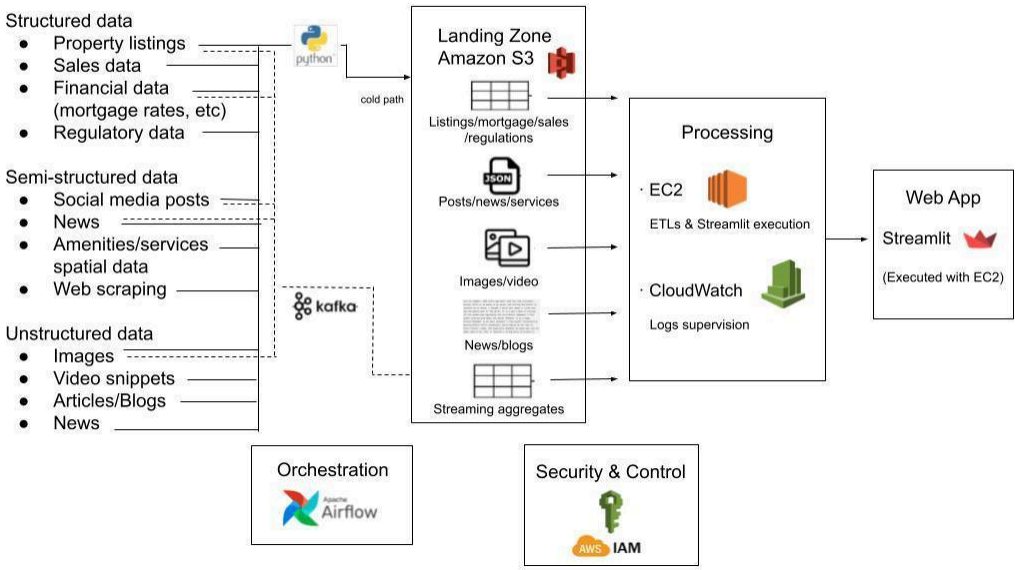

Figure 4. Initial System Vision: Data Ingestion and Processing Concept

As the project evolved, we conducted extensive research, experimentation, and redesign. In the final version, AWS Elastic Beanstalk replaced EC2 for deploying the Streamlit dashboard, simplifying scaling and deployment. Kafka was discarded in favor of a custom real-time ingestion pipeline using the Bluesky WebSocket API, running in a Docker container deployed on AWS ECS with Fargate.

Unfortunately, we didn't get access to the Idealista API and decided to create a data streaming process for property listing with python, Kafka and Airflow. A script is generating listings every few minutes and another script is listening to Kafka to transfer that information to S3. This entire process is contained in a Docker image and executed outside the cloud. With more time, we would have included this entirely in AWS.

One of the most significant shifts was how we handled external data sources. Initially, we considered building ingestion pipelines for each dataset, but due to time constraints and practical considerations, we opted to manually clean, structure, and upload public datasets (e.g., from Open Data Barcelona, Idealista, INE) directly to Amazon S3 using scripts or notebooks. These sources now complement the streaming data and are integrated into the dashboard's visualizations.

We also added:

- CI/CD pipeline using GitHub Actions for automated image build and deployment
- CloudFormation stacks (auto-generated) to define and provision infrastructure
- Secure IAM roles and a custom VPC with private/public subnets
- Logging and observability via Amazon CloudWatch

Added features:

- Streamlit deployed via Elastic Beanstalk
- ECS + Fargate architecture with private subnet and NAT Gateway
- Real-time Bluesky ingestion container with S3 output
- Manual and scripted ingestion of multiple external datasets directly into S3
- CI/CD with GitHub Actions
- IAM roles and CloudWatch logging for security and observability

Unimplemented or adjusted features:

- Kafka ingestion outside the cloud due to complexity
- EC2 hosting replaced by Elastic Beanstalk
- Dynamic frontend features like user login or favorites were not implemented

In summary, we moved from a conceptual, partially defined scope to a complete, functional, cloud-native system. The final implementation balances automation, scalability, and simplicity while integrating both real-time and static data sources.

4. System Architecture and Justifications

Our architecture combines key AWS services to support a secure and scalable data processing pipeline. We used a custom VPC with ECS and Fargate for running containerized tasks, managed through CloudFormation and monitored via CloudWatch. The frontend applications, built with Streamlit, were deployed using Elastic Beanstalk. Additionally, all structured, semi-structured, and unstructured data was uploaded to S3 for centralized storage. This setup ensures automation, efficiency, and cost optimization using AWS Free Tier resources.

4.1. AWS Cloud Services

To implement this architecture, we used a variety of AWS services, each fulfilling a specific role within the system. These include ECR for storing Docker images, ECS and Fargate for running containers, S3 for data storage, IAM for security, CloudWatch for logging, and CloudFormation for infrastructure provisioning. The following subsections describe each service, how we configured it, and the rationale behind our decisions.

4.1.1. Amazon Elastic Container Registry (ECR)

We used Amazon Elastic Container Registry (ECR) to store our Docker container images securely and privately. After building the Docker image from the GitHub repository using GitHub Actions, the image was pushed to a private ECR repository. The ECR registry acted as the central artifact store for our application deployments. We automated the login, build, tagging, and push steps using a GitHub workflow defined in a YAML file. This way, any code change that was pushed to the repository automatically triggered the build and upload of a new image version to ECR. The latest version of the image was always pulled from ECR during ECS task executions.

We chose ECR because it integrates natively with AWS services like ECS, Fargate and EB, supports IAM authentication, and fits perfectly into a CI/CD workflow using GitHub Actions. Using ECR allowed us to avoid managing separate image hosting services like Docker Hub and benefit from better security, performance, and permissions control within the AWS environment.

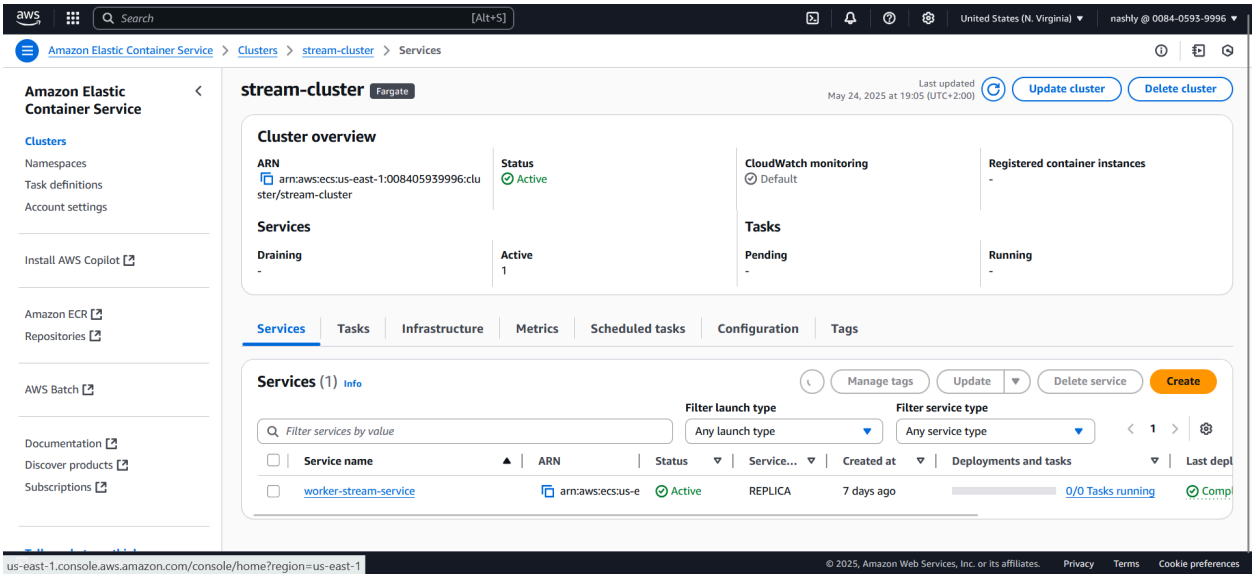

Figure 5. Docker image stored in Amazon ECR used for ECS deployment

4.1.2. Amazon Simple Storage Service (S3)

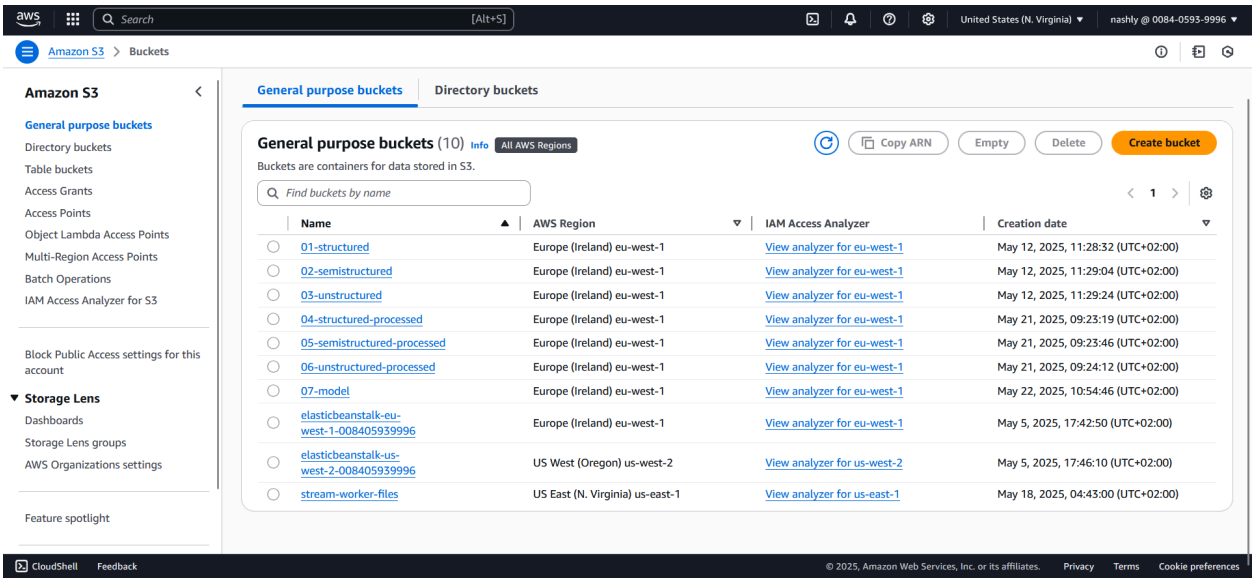

Figure 6. Data outputs and folders structure of external data sources inside Amazon S3

Amazon S3 was used as the central object storage solution for data ingestion and processed outputs. Multiple S3 buckets were created to support different categories of data, including structured, semi-structured, and unstructured data. These buckets were used to store datasets from various external sources, such as the Barcelona City Council, INE (Instituto Nacional de Estadística), and Idealista, each organized according to its data type and purpose. In addition to these, a dedicated bucket named stream-worker-files was created, where the application saves

processed JSON files and media files (e.g., images). The ECS container had appropriate IAM permissions to write directly to this bucket. Data was organized into folders such as images_bluesky/ and posts_bluesky/ for clarity and traceability. A dedicated bucket named 07-model was also created, and was used to store a serialized machine learning model in joblib format.

S3 is cost-effective, highly durable, and well-suited for unstructured data storage. It was the ideal solution for our use case, as we needed a simple way to store structured and unstructured outputs from our real-time data stream worker and from various external data providers. Alternative storage options like EFS or RDS were discarded due to complexity or cost.

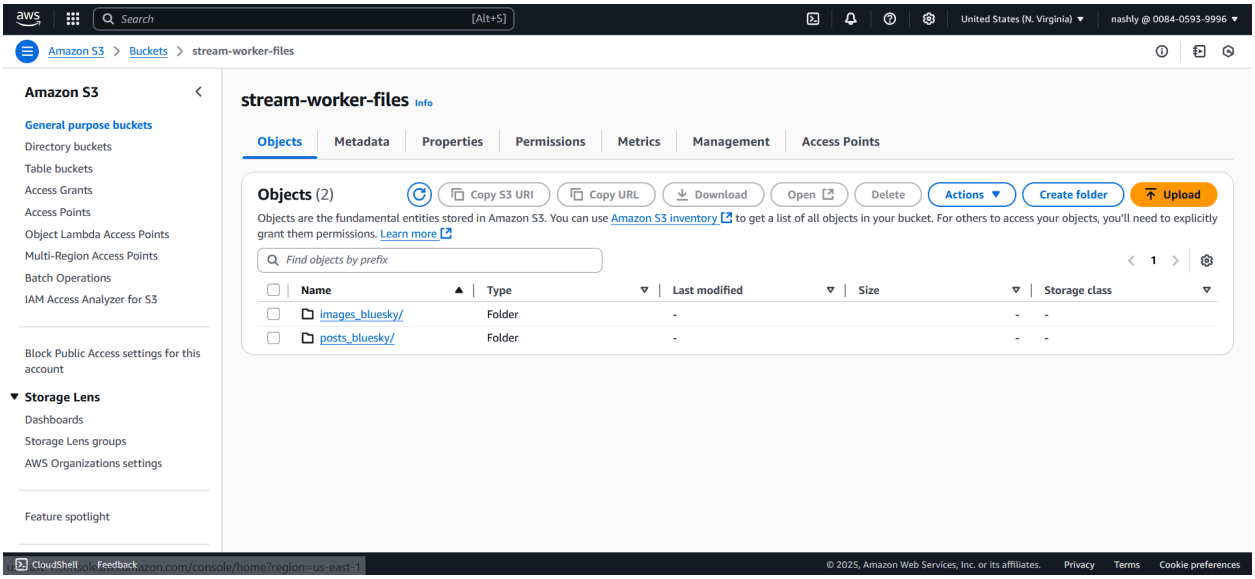

Figure 7. Data outputs and folders structure of Streaming BlueSky inside Amazon S3

4.1.3. Amazon Virtual Private Cloud (VPC)

We set up a custom VPC to logically isolate and secure our ECS services. Within the VPC, we defined both private and public subnets. ECS and Fargate were launched in the private subnet to improve security. The public subnet contained a NAT Gateway, allowing containers in the private subnet to access the internet (for example, to download Python dependencies) without being directly exposed.

Creating a custom VPC with segregated subnets allowed us to control the network flow and apply security best practices. This architecture is also scalable and reusable. We avoided deploying into the default VPC to maintain stricter control and adhere to production-grade cloud networking patterns.

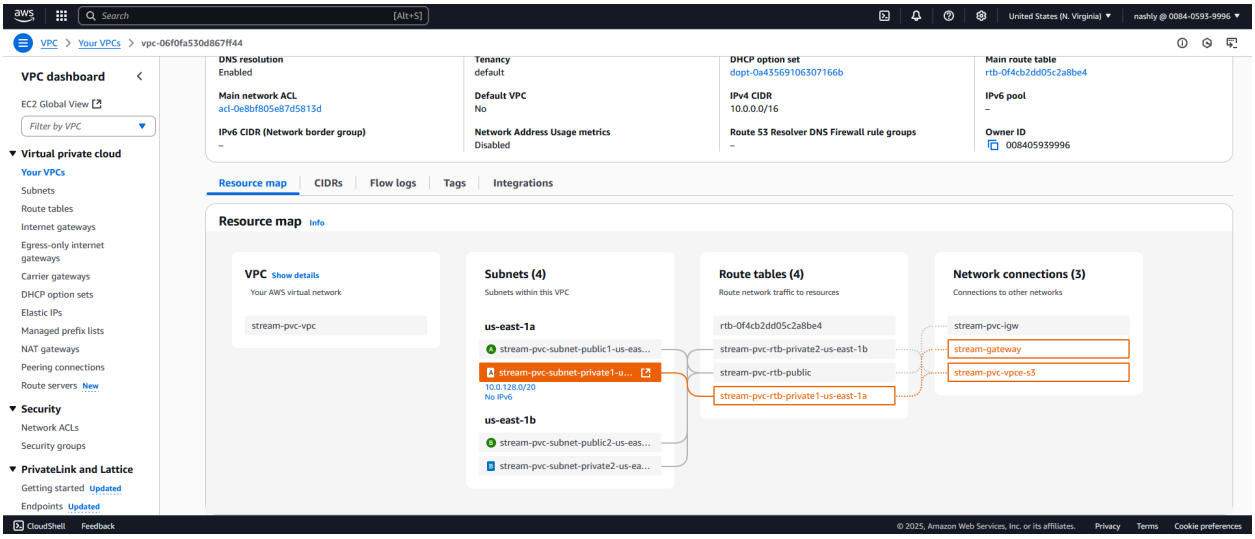

Figure 8. Custom VPC Configuration for ECS Deployment – stream-pvc-vpc

4.1.4. Amazon Elastic Container Service (ECS)

Amazon ECS served as our container orchestration platform. We created a cluster named stream-cluster where we registered a task definition called worker-stream. This task was responsible for processing real-time social media posts. With every change to the task definition or Docker image, a new version was registered, and the ECS service was updated to deploy the latest version of the container.

We opted for ECS because it provides a managed and scalable way to run containers, with deep integration with other AWS services. Compared to Kubernetes or self-managed EC2 instances, ECS allowed us to focus on the application rather than infrastructure management, reducing operational overhead.

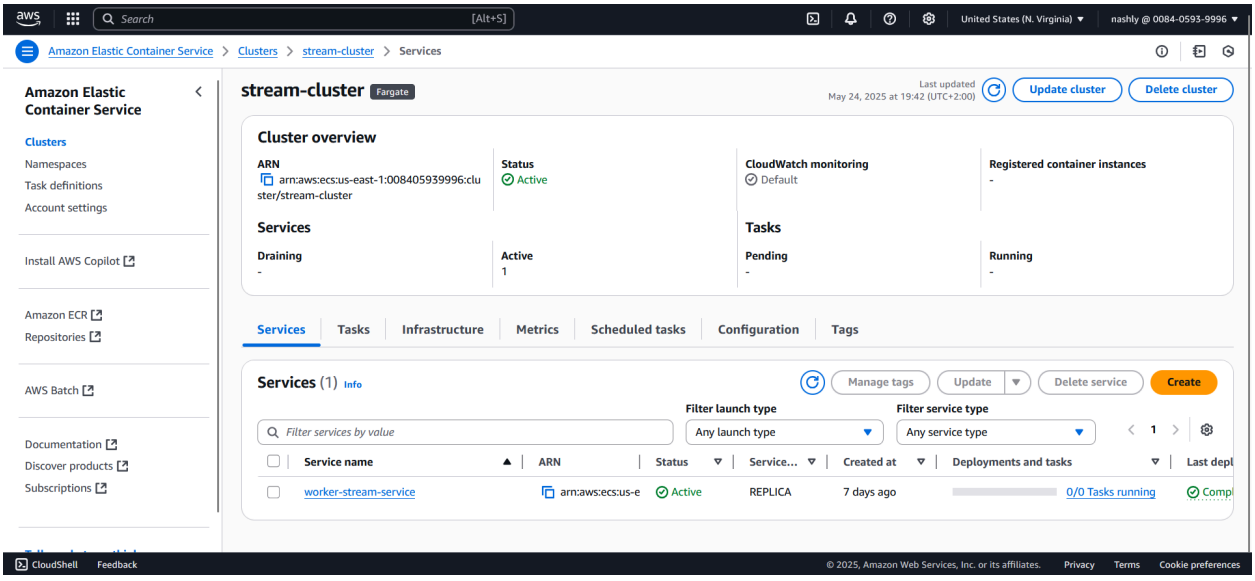

Figure 9. ECS Cluster and service configuration for stream processing

4.1.5. AWS Fargate

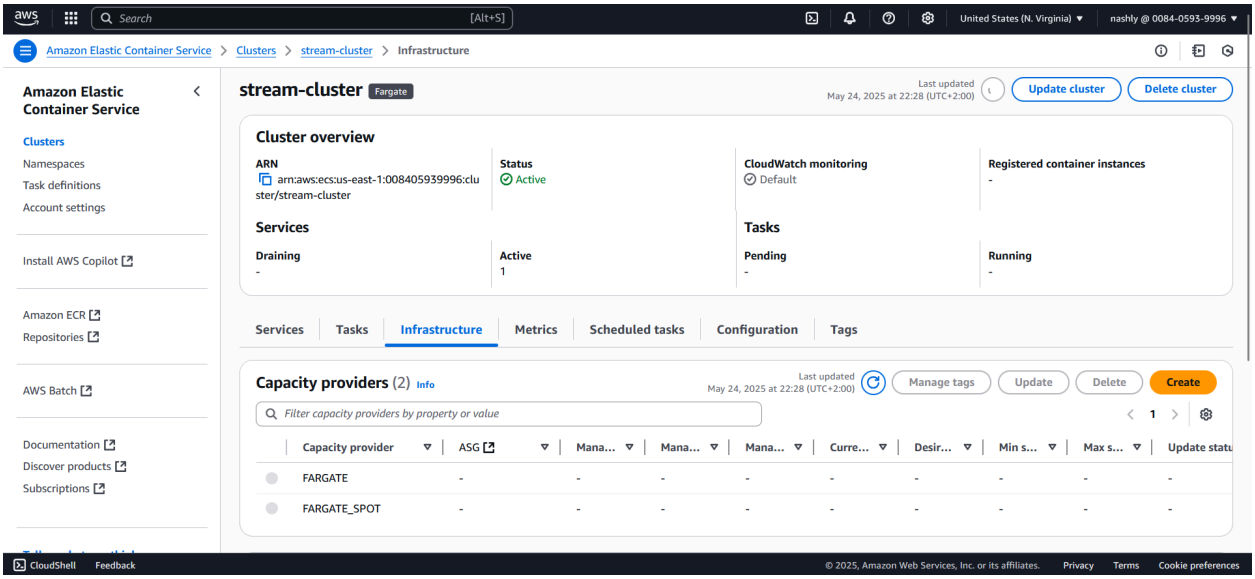

Figure 10. ECS Task launched using Fargate for serverless execution

Fargate was used to launch and run our containers without provisioning any EC2 instances. Each time ECS needed to run a task, it delegated to Fargate, which abstracted away the underlying compute resources. We specified CPU and memory limits, environment variables, secrets from Parameter Store, and log configuration in the task definition.

Fargate is the ideal choice for serverless container hosting. It allowed us to run containers without managing any servers or clusters, which simplified operations significantly. Choosing Fargate over EC2 reduced complexity and matched our needs for auto-scaling and pay-per-use cost efficiency.

4.1.6. Amazon CloudWatch Logs

We used CloudWatch Logs to monitor and debug our ECS container executions. Each task streamed logs to a specific log group and stream, as defined in the task definition. We could see real-time events such as data being saved locally or uploaded to S3, helping us trace the workflow and validate proper execution.

CloudWatch Logs provided us with centralized observability of all services. It was a natural fit due to its integration with ECS, and it allowed us to debug and monitor without installing external logging systems. This greatly accelerated our troubleshooting process and compliance with the requirement of using logs extensively.

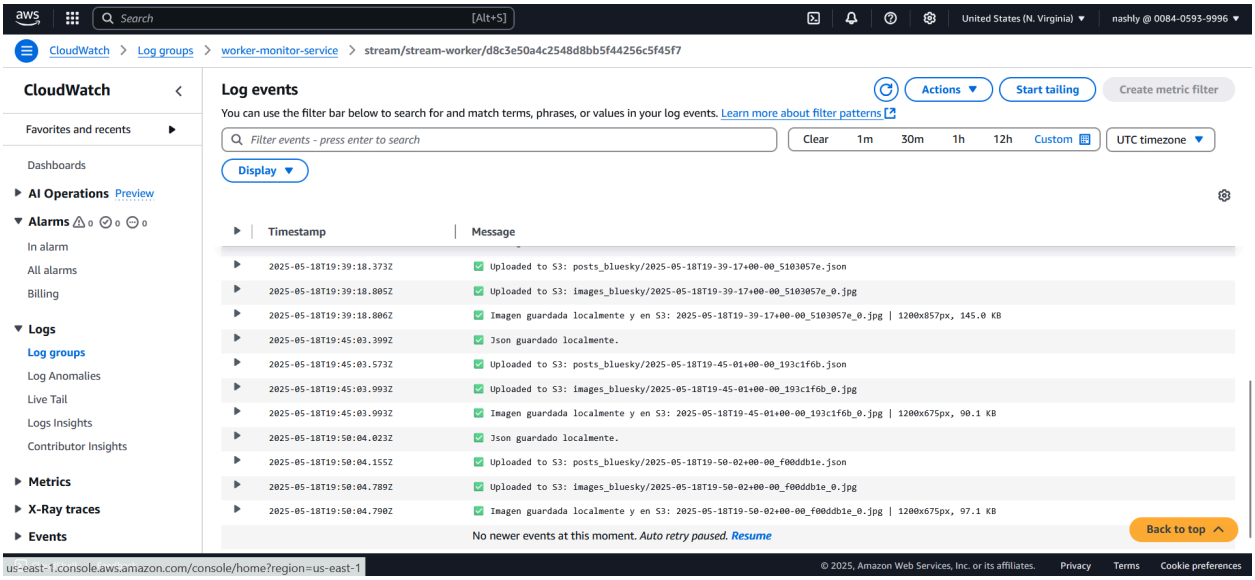

Figure 11. Real-time log output from ECS task inside CloudWatch Logs

4.1.7. AWS Identity and Access Management (IAM)

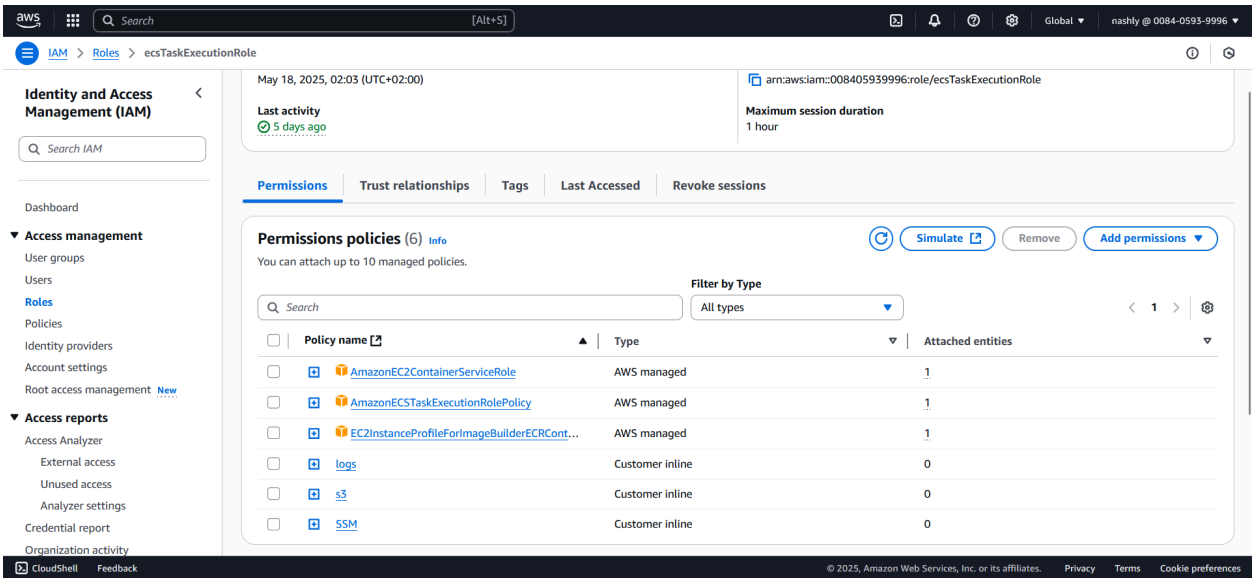

Figure 12. IAM roles and permissions attached to ECS tasks

We created and attached two main roles: ecsTaskExecutionRole and EC2ContainerServiceRole. These roles granted permissions to access services such as S3, CloudWatch Logs, ECR, and Systems Manager (SSM). Permissions were finely scoped to actions like writing logs, downloading Docker layers, and accessing secrets securely.

IAM ensured our application adhered to the principle of least privilege. We used managed policies where applicable to simplify role setup and avoid misconfigurations.

This approach was essential to securely grant the ECS tasks access to necessary AWS resources without overexposing them.

4.1.8. AWS CloudFormation

We used AWS CloudFormation to provision the ECS cluster and ECS service via templates generated from the AWS console. These templates defined infrastructure as code and allowed reproducible creation of environments.

CloudFormation allowed us to follow Infrastructure as Code (IaC) practices. This was beneficial for traceability, team collaboration, and disaster recovery. If we ever needed to recreate or scale our environment, we could do so deterministically using the same templates.

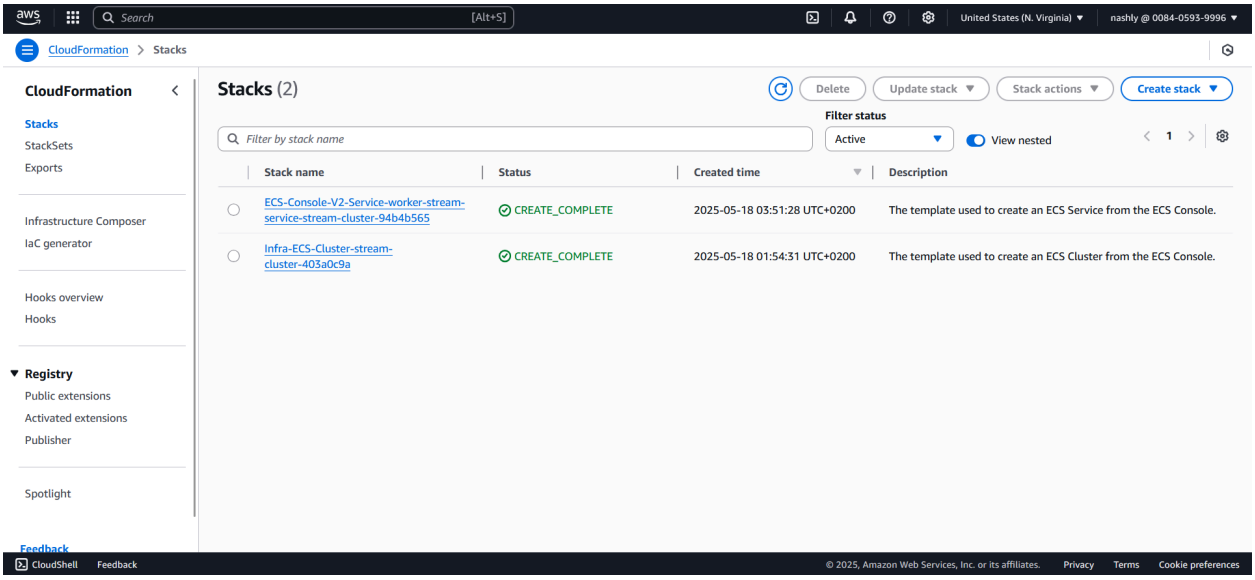

Figure 13. X. Auto-generated CloudFormation stack provisioning ECS and networking resources

4.1.9. AWS Elastic Beanstalk

AWS Elastic Beanstalk is a PaaS service that allows the creation and deployment of applications in a defined set of AWS services, such as EC2, S3, or SNS. It simplifies the process of deployment by abstracting the management of virtual machines and other components such as load balancers. It supports a wide range of programming languages, including Python. Given its ease of use, native integration with the AWS ecosystem, and built-in support for Python, we chose AWS Elastic Beanstalk to deploy our Streamlit application.

The application was containerized using Docker and stored in Amazon Elastic Container Registry (ECR), enabling consistent, portable deployments across environments.

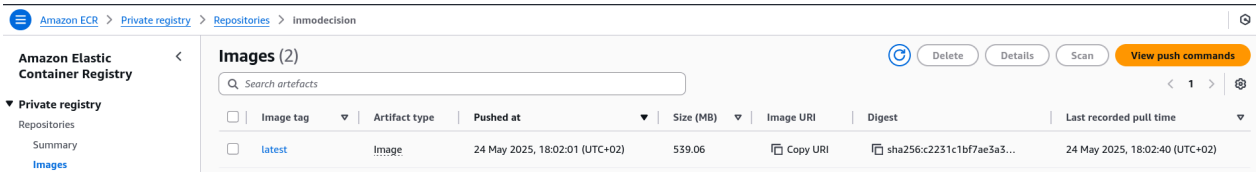

Figure 14. Dockerized Streamlit application, stored in AWS ECR.

To meet the resource demands of the Streamlit app, we opted for a t3.small EC2 instance instead of the default t3.micro, which Elastic Beanstalk typically assigns to free-tier environments. The t3.micro instance, offering only 1 vCPU and 1 GiB of RAM, was insufficient for our use case: the application experienced extremely slow startup times, followed by crashes during execution. These issues were primarily due to memory constraints, as the Streamlit app required more RAM to load models, handle data, and render the user interface. Upgrading to a t3.small instance, which provides 2 GiB of RAM and improved performance, resolved these stability and performance issues.

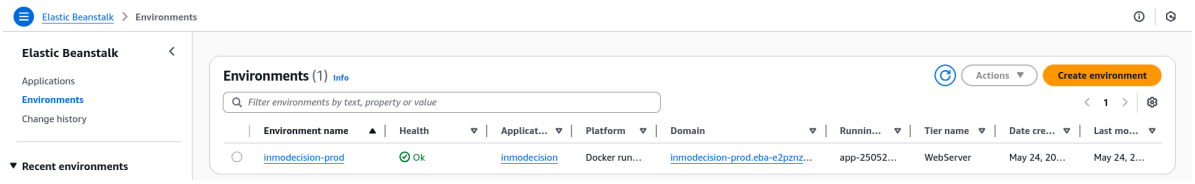

Figure 15. Running AWS EB environment for the Streamlit application.

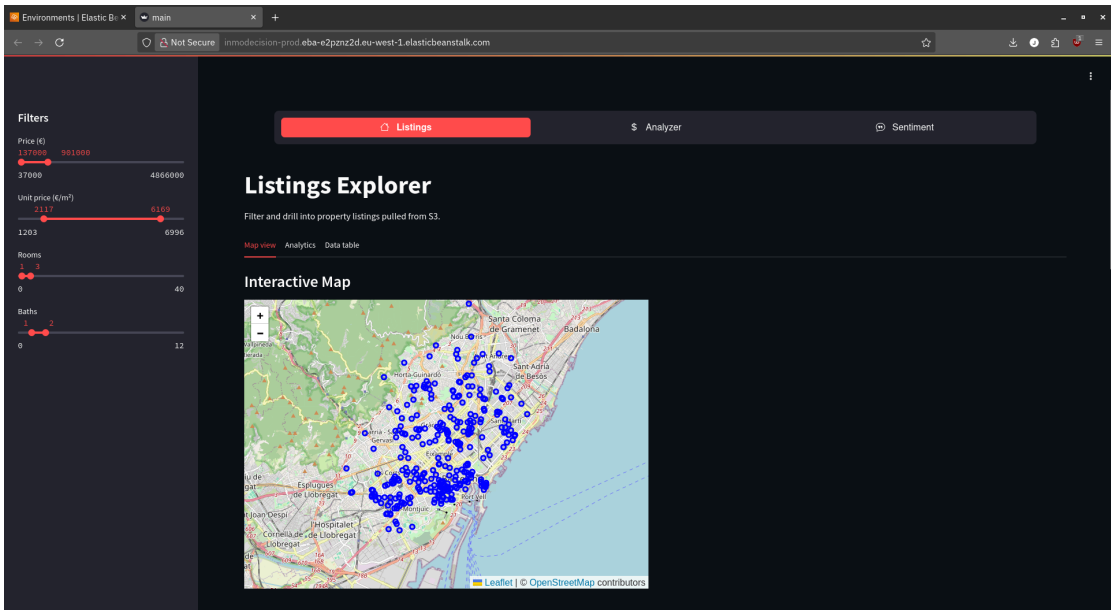

Figure 16. Deployed Streamlit application with AWS EB.

4.2. Open Data Sources

4.2.1. Government Open Data

Spain's Instituto Nacional de Estadística (INE), Open Data Barcelona and the European Central Bank Data Portal for property price indexes, demographics, and economic indicators.

4.2.2. Idealista API

For live property listings, pricing, and region-level property attributes. Due to the unavailability of a live property listings API and limitations encountered with web scraping, the ingestion of property listing data was simulated for this project. There is no real-time streaming pipeline in place, instead, we sourced publicly available static datasets, which were manually uploaded and used within the application to provide realistic and consistent data for exploration and analysis.

4.2.3. Streaming BlueSky Posts

To monitor real-time conversations related to real estate, we built a Python application that connects to the Bluesky API using WebSockets. It filters relevant posts, performs basic sentiment analysis, and stores the results—text, metadata, and images—in Amazon S3. The service runs in a container on AWS Fargate, enabling automated and scalable data collection without managing servers.

4.3. Programming Languages, Frameworks, Libraries and Additional Tools

To ensure a smooth and scalable cloud deployment, we will leverage a wide range of tools and services, all centered around Python, Docker, and Amazon Web Services (AWS). Each service is selected to match specific needs — from infrastructure provisioning to application hosting, data processing, and system observability.

4.3.1. GitHub and GitHub Actions

We used GitHub to host our project code and manage version control. GitHub Actions was configured to automate the CI/CD pipeline: on every push to the main branch, a workflow built the Docker image, pushed it to Amazon ECR, and triggered a deployment to ECS. This automation ensured that our application was always up to date without manual intervention.

Using GitHub Actions streamlined our development process and reduced deployment errors. It allowed us to maintain a consistent build and release cycle with minimal effort, fully integrating with AWS services like ECR and ECS through secure credentials and role-based access.

4.3.2. Docker and Docker Compose for service orchestration

We used Docker to containerize all application components, ensuring consistency across local and cloud environments. ECS services were deployed using one Docker image, while a separate image was created to deploy the Streamlit dashboard via Elastic Beanstalk. Docker simplified deployment and testing, enabling seamless builds through GitHub Actions and flexible deployment across ECS and Elastic Beanstalk with minimal adjustments.

4.3.3. Python for all components (ETL, API, and dashboard)

Python is the main development language for our entire stack. It enables seamless integration with AWS services via Boto3, provides rich libraries for data processing) and is fully supported across services like Lambda, EC2, and Elastic Beanstalk. Pandas, NumPy for data manipulation.

4.3.4. Kafka-python for Data ingestion

In this setup, Apache Kafka acts as a middle man. Our scripts generating synthetic data is a producer (Idealista listings and interest rate), sending data to Kafka topics (data streams). The other script is a consumer, reading data from these topics. Kafka decouples the producer from the consumer, meaning they don't directly interact. This provides buffering, so data isn't lost if the consumer is slow or down. It ensures reliable, real-time data flow between your scripts. Kafka also enables scalability for handling large data volumes. Essentially, it manages the data pipeline between data generation and consumption.

4.3.5. Airflow

Apache Airflow is an open-source platform designed for orchestrating complex computational workflows and data pipelines. It enables users to define, schedule, and monitor sequences of tasks programmatically, primarily using Python. These workflows, represented as Directed Acyclic Graphs (DAGs), allow for clear dependency management between tasks. Airflow provides a robust user interface for visualizing workflow status, managing task execution, and troubleshooting issues. Airflow is used in our project to automate the generation and transfer of data to S3

4.3.6. Streamlit

Streamlit is a modern Python framework designed for building interactive web applications for data science and machine learning. Streamlit allows us to transform Python scripts into user-friendly dashboards and tools with very little effort, all without needing to write HTML, CSS, or JavaScript.

In our project, Streamlit will serve as the main investment analysis dashboard where users can:

- Search and filter real estate listings by location, price, or property type
- Visualize market trends using charts and interactive widgets
- Simulate investment scenarios with input sliders and calculators
- Explore sentiment indicators from news and social media
- Compare mortgage scenarios and rental yields

5. Use of the Twelve-Factor Methodology

Throughout the development of the InmoDecision system, we aimed to align with the principles of the Twelve-Factor methodology as much as the project scope and academic constraints allowed.

- Codebase: The entire project is stored in a single GitHub repository, enabling version control, collaboration, and integration with CI/CD workflows via GitHub Actions.
- Dependencies: All dependencies are explicitly declared in a requirements.txt file, and containerized using Docker, ensuring consistency across environments.
- Config: Configuration values (like API keys and AWS credentials) are injected via environment variables and securely managed through AWS Parameter Store.
- Backing services: S3, ECR, CloudWatch, and RDS-equivalent functionalities are treated as attached resources and are replaceable and decoupled from the core logic.
- Build, release, run: We defined separate stages using GitHub Actions to build the Docker image, push to ECR, and deploy the container to ECS using Fargate.
- Processes: The application is stateless and runs as an isolated ECS task, allowing it to scale and restart without data loss.
- Port binding: The Streamlit application is deployed through AWS Elastic Beanstalk and binds to a port exposed by the platform, enabling external access via HTTP.
- Concurrency: Although concurrency wasn't heavily implemented, Fargate's architecture allows horizontal scaling of tasks if required.
- Disposability: Tasks are designed to be short-lived and restartable without dependency on prior state, supporting quick scaling and fault tolerance.
- Dev/prod parity: We maintained minimal differences between development and production by using the same Docker image across environments.
- Logs: Application output is captured by Amazon CloudWatch Logs, making logs streamable and centralized.
- Admin processes: While limited by project scope, diagnostic and testing scripts could be executed manually in isolated containers if needed.

Although this was an academic project with time constraints, we were intentional in applying the Twelve-Factor principles to promote good cloud-native design. The use of containers, serverless infrastructure (Fargate and Beanstalk), environment-based configuration, and centralized logging allowed us to build a modular, portable, and manageable system. These practices would facilitate scaling and maintainability if the system were to be extended in a production environment.

6. Development Methodology

The global management of the tasks will follow an agile Kanban methodology with regular stand up meetings to follow and discuss our tasks' board.

○ Task Breakdown

Task	Estimated Hours	Team Members Involved	Estimated Dates
Architectural Design	20	Gabriel, Alexis	April 1 – April 5
Research and Documentation	20	Nashly, Joan	April 1 – April 7
Data Collection and Processing	25	Joan, Gabriel	April 6 – April 14
Backend Development	25	Joan, Gabriel	April 8 – April 20
Frontend Development	15	Nashly, Alexis	April 15 – April 25
Deployment and Testing	20	Gabriel, Nashly	April 22 – May 5
Documentation Writing	15	All	May 1 – May 10
Meetings and Coordination	20	All	Ongoing (April 1 – May 22)

- **Gantt Chart**

CCBDA - Project

InmoDecision

Group Members:
Gabriel, Alexis, Nashly and Joan

Project start: **sáb, 3/15/2025**

Display week: **4**

TASK	ASSIGNED TO	PROGRESS	START	END
Design, Architecture and Data Collection				
Architectural Design	Gabriel, Alexis	- %	4/1/25	4/5/25
Research and Documentation	Nashly, Joan	- %	4/1/25	4/7/25
Data Collection and Processing	Joan, Gabriel	- %	4/6/25	4/14/25
Development				
Backend Development	Joan, Gabriel	- %	4/8/25	4/20/25
Frontend Development	Nashly, Alexis	- %	4/15/25	4/25/25
Testing and Validation				
Deployment and Testing	Gabriel, Nashly	- %	4/22/25	5/5/25
Documentation Writing	All	- %	5/1/25	5/10/25
Management and Coordination				
Meetings and Coordination	All	- %	4/1/25	5/22/25

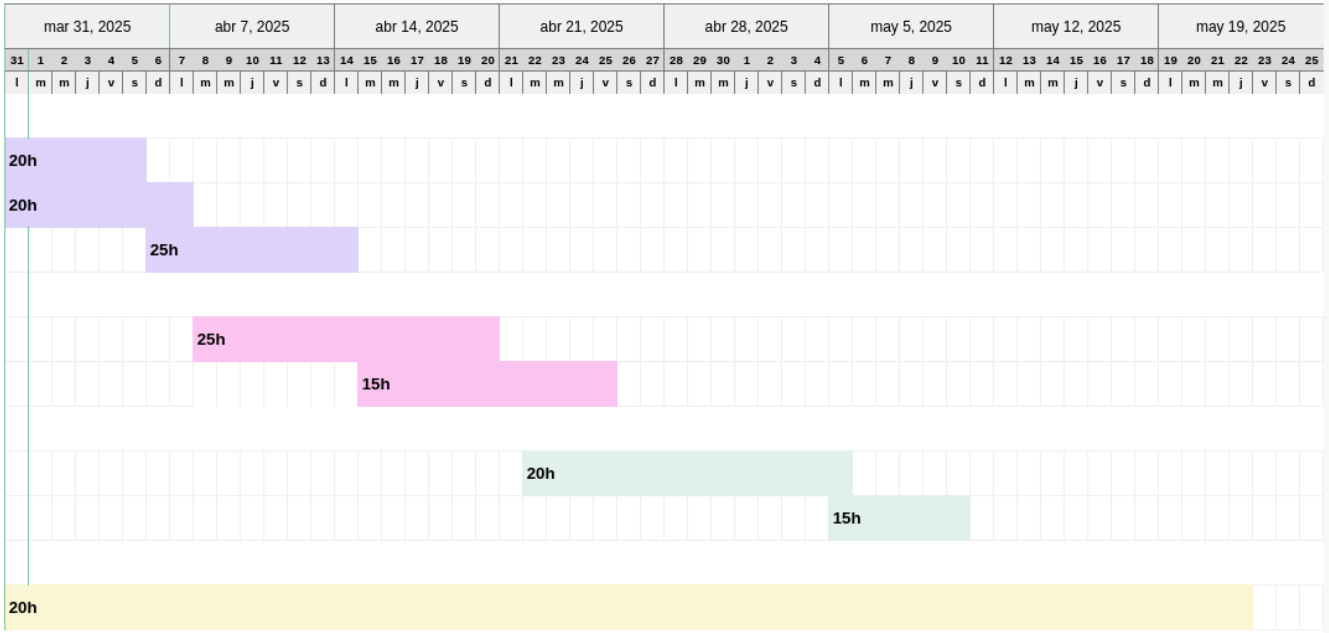

Figure 17. Gantt chart

7. Problems and Solutions

Throughout the development of the InmoDecision system, we encountered several technical and architectural challenges. The most significant issues revolved around permissions, networking, cost optimization, and cloud deployment configurations.

One of the first major problems arose when deploying the ECS task. The containerized worker application, which was responsible for ingesting data from the Bluesky API and saving it to S3, failed to function properly due to a lack of network connectivity. After debugging, we realized the ECS task was deployed inside a private subnet without any route to the internet. To solve this, we had to provision a NAT Gateway in a public subnet and then update the route table of the private subnet to allow outbound traffic through the NAT. This enabled the container to access the internet for pulling dependencies and streaming data. Additionally, to avoid unnecessary charges, we learned to stop the NAT Gateway by setting the ECS service's desired task count to zero when not in use.

Another recurring problem was related to IAM permissions. Initially, the ECS task did not have the necessary roles to interact with services such as Amazon S3, CloudWatch Logs, or Parameter Store. This resulted in application failures that were not immediately obvious. We resolved this by attaching the correct IAM roles (ecsTaskExecutionRole and a custom task role) and manually adjusting their inline policies to grant specific permissions needed for our services to function securely and effectively.

We also faced some infrastructure setup confusion while working with AWS CloudFormation. Since some stacks were auto-generated by the AWS console (such as when creating ECS clusters or services), understanding their role and structure took time. After investigation, we realized these CloudFormation stacks defined our infrastructure as code and enabled us to recreate or update it consistently.

On the frontend side, we initially considered deploying Streamlit via EC2. However, this introduced unnecessary complexity. We switched to Elastic Beanstalk, which allowed us to deploy the containerized dashboard with minimal setup and built-in scaling, saving both time and operational overhead. During deployment, we also encountered performance issues with the default instance type, which were resolved by selecting a more suitable configuration to meet the application's resource needs.

Lastly, due to time constraints, we manually ingested many of the static datasets (from sources like Open Data Barcelona and Idealista) into S3, instead of building dedicated ingestion pipelines. This manual approach, while not fully automated, was practical and allowed us to proceed with data modeling and visualization.

These challenges were resolved through iterative testing, peer collaboration, and extensive AWS documentation consultation. The solutions we implemented not only addressed immediate blockers but also improved the maintainability, scalability, and cost-efficiency of our overall architecture.

8. Conclusions and Future Work

The InmoDecision project successfully developed a functional, cloud web application that serves as a data-driven advisor for real estate investment. By integrating market data with real-time social sentiment analysis from platforms like Bluesky, the tool provides a more complete view of property opportunities, going beyond just prices and location to include public opinion. The system uses a range of AWS cloud services, including ECR, ECS with Fargate, S3, IAM, CloudWatch, and Elastic Beanstalk, demonstrating our understanding of what was seen during classes and labs.

Key achievements include:

- **Robust Cloud Architecture:** A well-defined architecture utilizing AWS services ensures scalability, fault isolation, and modularity, adhering to cloud-native principles.
- **Real-time Capabilities:** Implementation of a data streaming pipeline for real-time sentiment analysis of social media content using containerized worker applications on AWS Fargate.
- **Interactive Frontend:** A user-friendly Streamlit dashboard deployed on Elastic Beanstalk, allowing interactive exploration of market trends, sentiment insights, and investment simulations.
- **CI/CD Implementation:** Automation of the build and deployment process using GitHub Actions, pushing Docker images to ECR and deploying to ECS.
- **Adherence to Best Practices:** Application of Twelve-Factor Methodology principles for good cloud-native design, promoting modularity, portability, and manageability.

Future Work

While our application provides a strong foundation, several areas offer potential for future enhancements and full-scale production readiness:

- **Full Integration of Idealista Listing Fetching in the Cloud:** Currently, the ingestion of Idealista property listing data is simulated and processed outside the cloud due to API limitations and time constraints. A significant future step would be to build a robust, automated, real-time streaming pipeline for Idealista listings directly within AWS. This

could involve exploring alternative data ingestion methods, such as direct API access (if feasible) or a dedicated web scraping solution, fully integrated with Kafka on AWS (e.g., MSK) and Airflow for orchestration, ensuring continuous and real-time updates to the property database in S3.

- **Enhanced Sentiment Analysis:** Improving the sophistication of the sentiment analysis model, potentially by incorporating more advanced NLP techniques, handling sarcasm or nuanced language, and integrating sentiment data from a wider array of social media platforms beyond Bluesky.
- **Integration with Additional Data Sources:** Incorporating more diverse open data sources to provide even richer context for investment decisions.
- **Advanced Analytics and Machine Learning Models:** Developing more complex predictive models, such as rental yield forecasts or neighborhood growth predictions, and exploring reinforcement learning for optimal investment strategies. For our model, we applied a simple linear regression.
- **Cost Optimization and Monitoring beyond Free Tier:** As the project scales beyond academic use, implementing more rigorous cost monitoring and optimization strategies, and potentially exploring reserved instances or savings plans for sustained workloads.

9. References

- Open Data Barcelona: https://opendata-ajuntament.barcelona.cat/
- Datos.gob.es (Spain Open Data Portal): https://datos.gob.es
- Idealista API: https://developers.idealista.com/
- Hugging Face (Sentiment Analysis Models): https://huggingface.co
- Streamlit (Web App Framework): https://streamlit.io
- Amazon EC2 (Compute Hosting): https://aws.amazon.com/ec2/
- Amazon S3 (Object Storage): https://aws.amazon.com/s3/
- Amazon DynamoDB (NoSQL Database): https://aws.amazon.com/dynamodb/
- Amazon CloudWatch (Monitoring and Logging): https://aws.amazon.com/cloudwatch/
- AWS IAM (Access Management): https://aws.amazon.com/iam/
- AWS SNS (Notification Service): https://aws.amazon.com/sns/
- AWS Free Tier Overview: https://aws.amazon.com/free/