CLOUD PICTURE PLATFORM

MEI - ACAP (subgroup 11.3)

Authors:

Albert Bausili
Bernat Borràs
Àlex Ollé
Noa Yu Ventura

17/12/2025

Contents

1 Scope

1.1 Introduction

We live in a world saturated with digital media, making us struggle to maintain a centralized and secure repository for our ever-growing personal image libraries. Finding specific memories has become a daunting task, as traditional storage methods lack the intelligence to retrieve photos based on their actual content or context. Additionally, it is common to face a disconnect between our private collections and the wider web, lacking the tools to easily discover similar imagery or visually related images online.

Cloud Picture Platform (referred to as CloudPP from now on) bridges this gap by delivering a sleek, intelligent platform powered by **cutting-edge technologies** that transforms how we manage and explore our images. Our application offers advanced features such as semantic search and automated text recognition, ensuring that every image—whether searched by visual label or embedded text—is just a simple query away. By integrating high-performance cloud infrastructure with intuitive design, CloudPP not only solves the chaos of storage but empowers users to unlock the full potential of their digital memories.

This report centers specially around the infrastructure of the application, its deployment and the workflow we have established for the compliance with the 12 factor methodology. This is because this subject is about cloud infrastructure, not programming. For more information about the application itself read the documentation in the README.md of our repository or go to http://tiny.cc/demo_ACAP to see a demo of its functionalities. The link to our **project repository** is: `https://github.com/CCBDA-UPC/2026_1-Project-11_03_A`

2 Planning

2.1 Milestones

We were planning to have a milestone for each week. Each milestone is due a certain date and when the due date arrives the issues corresponding to a milestone should be finished. In case an issue is not finished in time a milestone branch is created to have a "snapshot" of the develop branch at that moment. As shown in the table below we can see each of the milestones we have followed.

Milestone	**Start Date**	**End Date**
0.1.0	November 20th	November 27th
0.2.0	November 28th	December 4th
0.3.0	December 5th	December 12th
1.0.0 (Release)	-	December 17th

The following sections (project tasks and Gantt diagram) are from the final state of the project, meaning that they reflect the development of the project and not strictly their planning. This is because due to our lack of knowledge for starting from scratch, we considered it was not worth estimating (exactly) how long each task would take, it was more efficient adding them to the milestone and distribute them equally. Then we would try to meet the milestone's deadline as much as possible.

2.2 Project Tasks

As a **disclaimer**, the minimum granularity we are using is number of hours as integers, so some tasks that took less than 1h are shown as 1h. Tasks that lasted 1h are also shown as 1h. Tasks that lasted more than 1h and a half are shown as 2h, etc. Consequently, tasks below 1h and a half are shown as 1h, etc.

Some issue numbers might be missing (such as #1 or #43), this is due to two factors:

- GitHub shares the numbers for both PRs and issues. For example, issues #1 and #55 do not appear because they are PRs.
- Some issues were closed (won't do) or were already done in another PR (they were simple issues), so they won't be showed in this table.

Table 1: Project Development Tasks, Milestones, and Assignments

Project Schedule						
ID	**Task Name**	**Description**	**Mile.**	**Hrs**	**Assignee**	**Type**
M01	Project Coordination	Twice a week sync meetings, milestones planning and teacher interviews	-	15	All Team	Plan
M02	Architecture Design	Initial system architecture design sessions	-	6	All Team	Plan
M03	First Draft	First draft development	-	3	All Team	Plan
#4	Add Basic Structure	Create directory structure of the project	0.1.0	1	Albert	Backend
#5	FrontendLogin	Add login capabilities to the react + shadcn frontend	0.1.0	3	Àlex	Frontend
#6	BackendLogin	Add necessary functionality to support login to the platform	0.1.0	3	Bernat	Backend
#7	ImageIngestion-Lamba	Create the image ingestion lambda	0.1.0	2	Albert	Infra
#8	ImageRekognition-Lambda	Create a lambda for extracting the contents of an image and indexing them in a DB	0.1.0	2	Noa	Infra
#13	Backend Cognito	Replace current login system for Cognito	0.2.0	1	Bernat	Backend
#15	Backend Functionalities	Add Create and Read fundamental operations for images	0.2.0	2	Bernat	Backend
#16	Lambda OpenSearch	Switch RDS to OpenSearch Serverless for AWS Rekognition Lambda	0.2.0	1	Noa	Infra.
#17	SearchLambda	Implement the search lambda to search for photos	0.2.0	2	Albert	Backend
#18	Frontend Image Upload	Implement image upload capabilities	0.2.0	1	Àlex	Frontend
#19	GetImages-Lambda	Add a lambda for getting the images for frontend display with pagination	0.2.0	1	Albert	Backend
#20	ImplementAPI-Gateway	Implement and use API gatweay to interact with the frontend	0.2.0	1	Bernat	Infra.
#21	ImageCaching	Implement image caching and pagination in an Amazon ElastiCache	0.3.0	3	Bernat	Backend
#22	SimilarImage-Lambda	Implement search for similar image	0.3.0	4	Albert	Infra
#23	Implement Image Grid	Implement Image Grid with pagination support	0.3.0	5	Àlex	Frontend
#24	DeleteImages-Lambda	Implement the Image deletion lambda	0.3.0	2	Noa	Backend

Continued on next page

Table 1 – continued from previous page

ID	Task Name	Description	Mile.	Hrs	Assignee	Type
#35	FixDeployment-Bugs	Fix bugs and add CI/CD	0.3.0	9	Albert	Maint.+ CI/CD
#37	Backend File Limit	Add File Limit to Upload Image	0.3.0	1	Bernat	Backend
#38	Backend Auth	Add Authentication to Endpoint Upload Image	0.3.0	1	Bernat	Backend
#44	Search Image	Implement image search bar	0.3.0	1	Àlex	Frontend
#45	Suggest Similar Images	Implement suggest similar images feature	0.3.0	3	Àlex	Frontend
#46	Tests Naming Issue	Change Naming for Local Tests from Integration to Unit	0.3.0	1	Albert	Maint.
#54	Production CD	Allow permanent option manual action trigger in CD	0.3.0	4	Noa	CI/CD
#58	Dependency-Groups	Make sure that ALL pyproject.toml use dependency groups	0.3.0	1	Bernat	Maint.
#64	SolveMerge-Conflicts	Solve merge conflicts dev -¿ main	0.3.0	1	Albert	Maint.
SI	Solve Integration Issues	Debug integration between frontend and backend	1.0.0	8	Albert	Integration
CR	Code Review	Reviewing Pull Requests: look for good code practices, security vulnerabilities, review unneeded configurations/comments, etc.	-	4	Bernat	-
			-	8	Albert	-
			-	6	Noa	-
			-	0	Àlex	-
PR	Project Report	Writing report and reviewing the document	-	12	Bernat	-
			-	8	Albert	-
			-	30	Noa	-
			-	6	Àlex	-
PP	Presentation Prep	Preparing slides and rehearsing presentation	-	8	Bernat	-
			-	6	Albert	-
			-	6	Noa	-
			-	3	Àlex	-

2.3 Gantt Diagram

To provide a clearer visual representation of the project's evolution and the timeline of the tasks described above, the following Gantt chart is provided in the next page.

Tasks in red indicate that they were completed outside the corresponding milestone, tasks in black indicate that they were completed within the milestone's date.

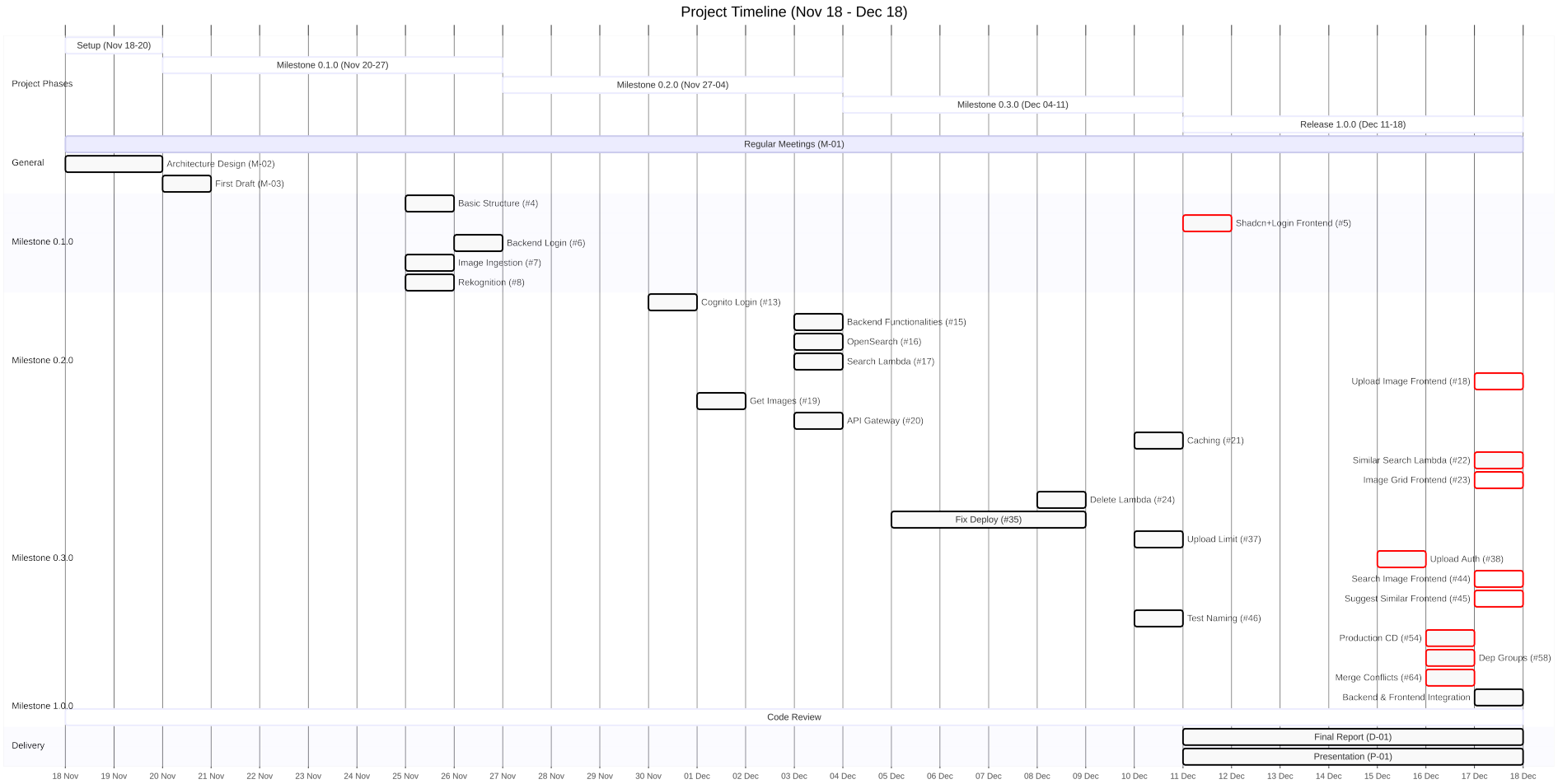

Figure 1: Project task Gantt chart, covering the period from November 18 to December 17

3 Design

The image on the next page shows the intial design of the architecture we had in mind to develop for this project.

It implements a hybrid cloud architecture that balances the control of a long-running containerized service with the scalability of serverless functions. The core logic resides in a centralized FastAPI backend running on EC2, which acts as the primary entry point and orchestrator for user requests. Instead of burdening this web server with heavy processing tasks, the system offloads specific operations—such as image ingestion, processing, and deletion—to ephemeral AWS Lambda functions. This ensures the API remains responsive while heavy compute tasks scale independently on demand.

The entire infrastructure follows a GitOps workflow managed by GitHub Actions. Every deployment treats the infrastructure as code (CloudFormation), building immutable Docker artifacts for the backend and syncing static assets for the frontend to S3/CloudFront.

Component Layer	**Technology & Role**
Orchestration	**EC2 (Docker):** Hosts the FastAPI application; serves as the "brain" that delegates tasks.
Compute Workers	**Lambda:** Handles specific asynchronous tasks like `ImageIngestion`, `ImageProcessing`, and `ImageDeletion`.
Data Stores	**S3:** Primary storage for raw images and frontend static files. **OpenSearch:** Dedicated search engine for indexing image metadata. **Redis:** In-memory cache for session management and quick data retrieval.
Security	**Cognito:** Managed authentication for users. **VPC:** Isolates the EC2, Redis, and OpenSearch instances in private subnets for security.

Table 2: CloudPP Architecture Summary

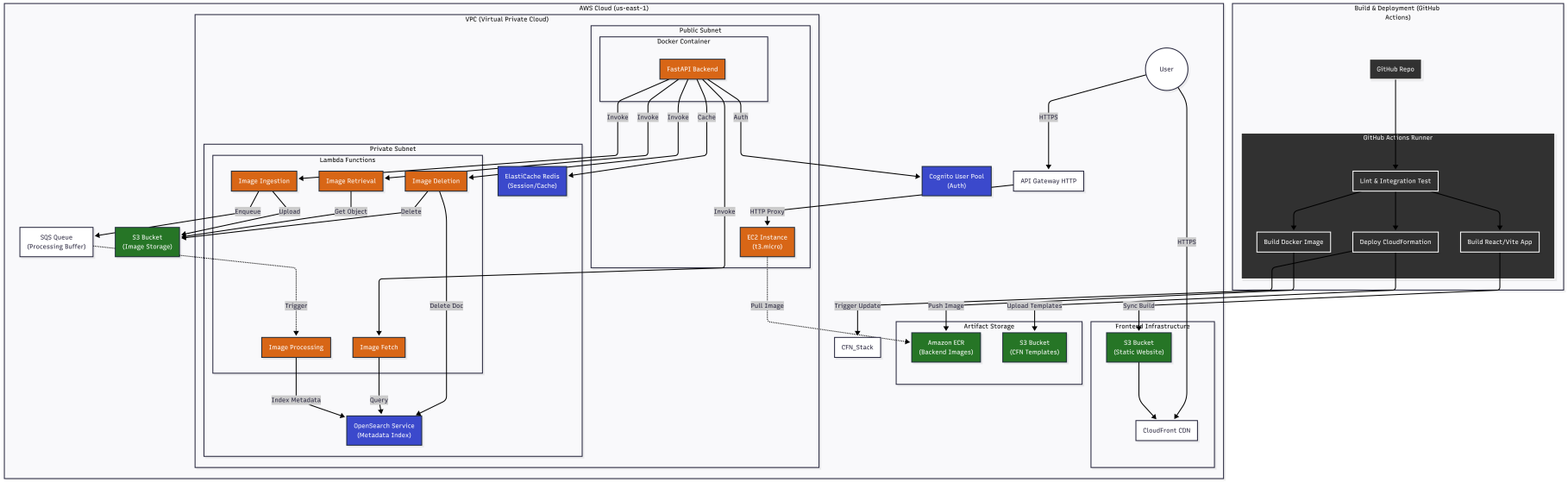

Figure 2: Initial architecture of the infrastructure in our planning

4 Final Design

The initial design laid the foundation for a hybrid cloud system that leverages both containerized services and serverless functions to strike a balance between control and scalability. However, over the course of the project we have refined and optimized its architecture to ensure maximum efficiency, performance, and maintainability. In this section, we outline the final design of the architecture of our application as shown in figure .

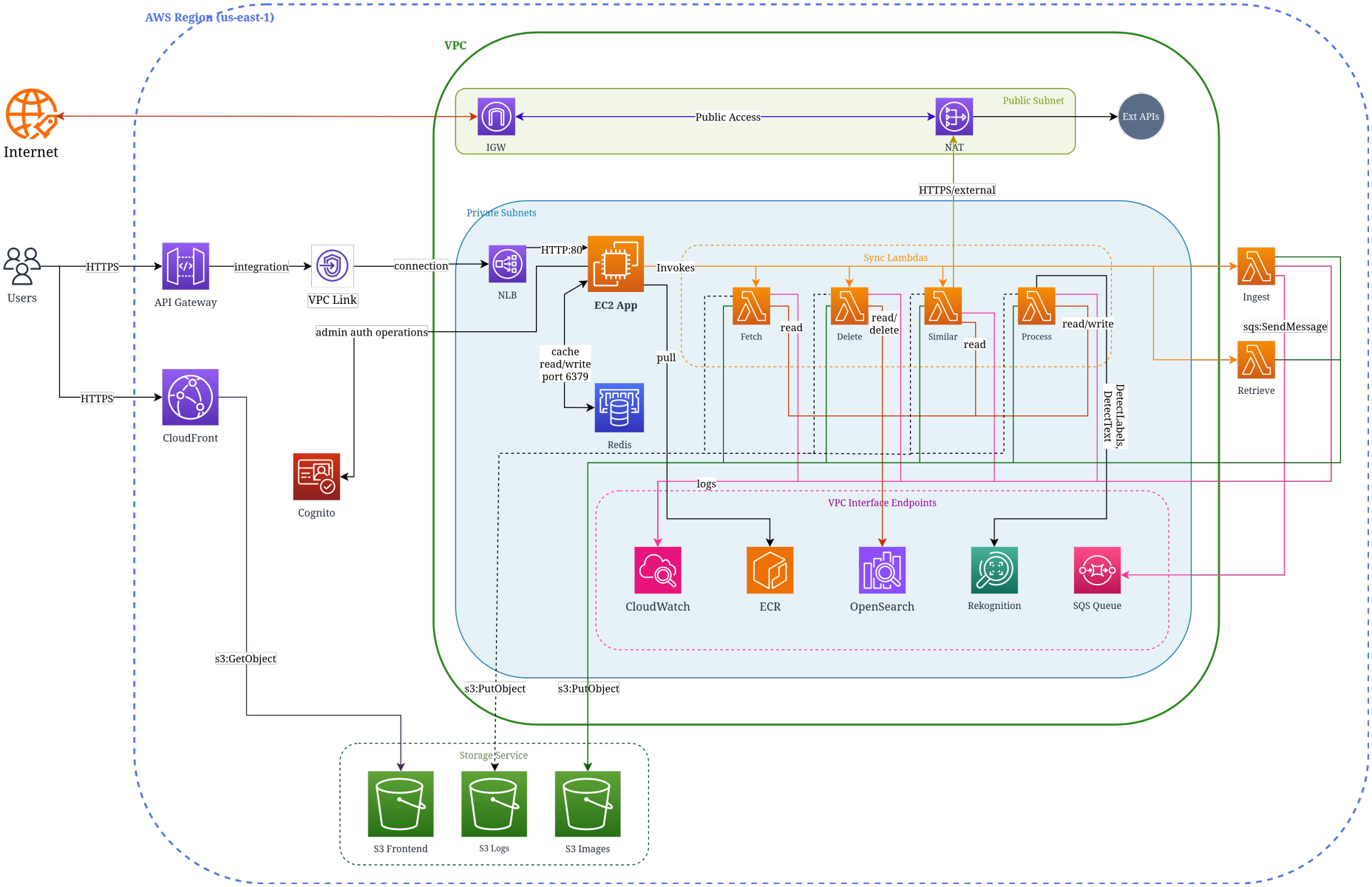

Figure 3: Final architecture design of our application and its infrastructure

4.1 Architecture Overview

CloudPP is a cloud-based photo analytics and management platform deployed entirely on AWS. The project follows a **monorepo structure** with three main components: a Python FastAPI backend, a React/TypeScript frontend, and serverless AWS Lambda functions. Infrastructure is defined as code using **AWS CloudFormation** with a nested stack architecture that enables modular, maintainable deployments.

4.2 Application Design

The codebase is organized into four primary directories:

- `apps/backend`: A FastAPI application containerized with Docker, handling authentication and orchestrating Lambda invocations.
- `apps/frontend`: A React application built with Vite and TanStack Router.
- `apps/lambdas`: Six independent Lambda functions implementing an event-driven image processing pipeline.
- `infrastructure`: CloudFormation templates defining all AWS resources.

This monorepo approach simplifies dependency management, enables code sharing via `apps/common` (e.g., shared Logger utility), and streamlines CI/CD.

4.3 Infrastructure Design

4.3.1 Nested CloudFormation Stack Architecture

The `main.yaml` template orchestrates **12 nested stacks**, each responsible for a specific concern. This separation follows the **single responsibility principle**: stacks can be updated independently, and outputs are shared via `!GetAtt` references creating clear dependency chains.

4.3.2 VPC Infrastructure

The VPC implements a **hub-and-spoke model** with strict network isolation:

Network Topology:

- **VPC CIDR**: `10.0.0.0/16` with DNS hostnames and DNS support enabled.
- **Private Subnet 1**: `10.0.1.0/24` in Availability Zone A — hosts Lambda functions and EC2.
- **Private Subnet 2**: `10.0.2.0/24` in Availability Zone B — provides multi-AZ redundancy.
- **Public Subnet**: `10.0.10.0/24` — hosts only the NAT Gateway for outbound internet.

Stack	Purpose
`VpcStack`	VPC, subnets, NAT Gateway, Internet Gateway, VPC endpoints
`LoggingStack`	Shared S3 bucket for application logs
`OpenSearchStack`	OpenSearch Serverless collection with encryption/network policies
`ElastiCacheStack`	Redis cluster for pagination caching
`ImageIngestionStack`	S3 bucket, SQS queue, ingestion Lambda
`ImageProcessingStack`	Rekognition-powered metadata extraction Lambda
`ImageRetrievalStack`	S3 listing Lambda
`ImageFetchStack`	OpenSearch query Lambda
`ImageDeletionStack`	S3 and OpenSearch deletion Lambda
`SimilarImageStack`	External image search Lambda
`OpenSearchAccessStack`	IAM data access policies for OpenSearch
`BackendStack`	EC2, NLB, API Gateway, Cognito

Table 3: CloudFormation nested stacks and their responsibilities

Routing Configuration:

- **Public Route Table**: Routes `0.0.0.0/0` to the Internet Gateway for NAT Gateway egress.
- **Private Route Table**: Routes `0.0.0.0/0` to the NAT Gateway, enabling private resources to reach the internet without direct exposure.

Endpoint	Type	Purpose
S3	Gateway	S3 access without NAT charges
SQS	Interface	Queue access from VPC-bound Lambdas
Rekognition	Interface	Image analysis API calls
CloudWatch Logs	Interface	Lambda logging
SSM/SSM Messages/EC2 Messages	Interface	Session Manager debugging
ECR API/ECR DKR	Interface	Docker image pulls without internet
OpenSearch Serverless	Interface	Search API access

Table 4: VPC Endpoints configuration

VPC Endpoints (11 total):

Security Groups:

- **LambdaSecurityGroup**: Allows inbound HTTPS (443) from VPC CIDR, permits all outbound traffic. Shared across Lambda functions and VPC endpoints.

- **BackendSecurityGroup**: Allows inbound HTTP (80) from anywhere for API access.

Design Rationale: The decision to place **Lambdas inside the VPC** was driven by OpenSearch Serverless requirements—it only accepts connections via VPC endpoints for enhanced data isolation. The trade-off is increased cold start times (2-5s), but security and compliance benefits outweigh this cost.

4.3.3 OpenSearch Serverless

OpenSearch Serverless provides two critical capabilities that drove the migration from RDS:

1. **Full-text search**: Native support for text queries across image labels and OCR-extracted text without custom indexing logic.
2. **Vector database**: Ability to store and query embedding vectors for similarity search, enabling future ML-powered image matching.

Collection Configuration:

- **Type**: `SEARCH` — optimized for full-text search workloads.
- **Name**: Dynamically generated as `cpp-${EnvironmentName}` (e.g., `cpp-dev`, `cpp-prod`).

Security Policies:

- **Encryption Policy**: Uses AWS-owned keys for encryption at rest.
- **Network Policy**: Restricts access to the VPC endpoint when deployed in VPC mode; falls back to public access for development.

Data Access Policy: Grants fine-grained permissions to Lambda IAM roles. Only the four Lambda roles (`ImageProcessing`, `ImageFetch`, `ImageDeletion`, `SimilarImage`) have access—the backend cannot directly query OpenSearch.

4.3.4 ElastiCache Redis

Redis caches image listing results to reduce Lambda invocations:

- **Engine**: Redis 7.1
- **Node Type**: `cache.t3.micro` (cost-optimized for development)
- **Nodes**: Single node (no replication for cost savings)
- **Security**: Accessible only from VPC CIDR on port 6379

The backend stores paginated image lists with TTL, avoiding repeated `image_retrieval` Lambda calls for identical queries.

4.3.5 Backend Infrastructure

Compute:

- **EC2 Instance**: Amazon Linux 2023 (`t3.micro`), deployed in private subnet.
- **Docker Runtime**: The FastAPI app is containerized and pulled from ECR at startup.
- **UserData Script**: Installs Docker, logs into ECR, pulls image, and runs container with environment variables.

Load Balancing:

- **Network Load Balancer (NLB)**: Internal NLB distributes traffic to EC2 target group on port 80.
- **VPC Link**: Connects API Gateway to the private NLB.

API Gateway (HTTP API):

- **Type**: `HTTP_PROXY` integration via VPC Link.
- **CORS**: Permissive (*) for all origins, methods, and headers.
- **Auto-deploy**: Changes deploy automatically via `$default` stage.

Authentication (Cognito):

- **User Pool**: Email-based signup without auto-verification (admin confirms in tests).
- **Client**: `USER_PASSWORD_AUTH` and `ADMIN_NO_SRP_AUTH` flows enabled.

IAM Permissions: The EC2 role can invoke all five Lambda functions, perform Cognito admin operations, and signal CloudFormation for deployment status.

4.3.6 Lambda Functions Infrastructure

Each Lambda follows a consistent pattern:

VPC Configuration: Four Lambdas require VPC access for OpenSearch. They use the `AWSLambdaVPCAccessExecutionRole` managed policy and are attached to both private subnets with the shared security group.

SQS Event Source Mapping: The `image_processing` Lambda is triggered by SQS with:

- **Batch Size**: 10 messages
- **Visibility Timeout**: 330 seconds (exceeds Lambda timeout of 300s to prevent duplicate processing)

Lambda	Trigger	VPC	Key Permissions
`image_ingestion`	Synchronous	No	S3 PutObject, SQS SendMessage
`image_processing`	SQS	Yes	S3 GetObject, Rekognition, OpenSearch
`image_retrieval`	Synchronous	No	S3 ListBucket
`image_fetch`	Synchronous	Yes	OpenSearch query
`image_deletion`	Synchronous	Yes	S3 DeleteObject, OpenSearch delete
`similar_image`	Synchronous	Yes	OpenSearch query, external API calls

Table 5: Lambda functions configuration

4.4 Image Processing Pipeline

The system implements an **event-driven architecture**:

1. **Ingestion**: `image_ingestion` receives base64-encoded images, stores them in S3, and publishes an SQS message.
2. **Processing**: `image_processing` is triggered by SQS, calls Rekognition for `DetectLabels` and `DetectText`, then indexes metadata into OpenSearch.
3. **Retrieval**: `image_retrieval` lists images from S3; `image_fetch` queries OpenSearch for text-based search.
4. **Deletion**: `image_deletion` removes images from both S3 and OpenSearch atomically.
5. **Similarity**: `similar_image` retrieves an image's Rekognition labels from OpenSearch and makes **external API calls to open data sources** (Wikimedia Commons, Library of Congress) to find visually similar images based on those labels.

This decoupled pipeline allows each stage to scale independently and fail gracefully.

4.5 Testing Strategy

4.5.1 Unit Tests

Each Lambda and the backend have dedicated unit tests that run without AWS credentials by mocking external services. CI executes these on every PR to `develop`.

4.5.2 Integration Tests

Full end-to-end integration tests deploy a complete CloudFormation stack per PR, execute tests against real AWS resources, and tear down the stack afterward. The `deploy` label triggers this workflow.

4.6 CI/CD Workflows

4.6.1 PR Pipeline (`deploy.yml`)

- **Trigger**: PRs to `main` or `develop` with `deploy` label.
- **Jobs**: Lint → Unit Test → Deploy Stack → Integration Test → Cleanup.
- **Concurrency**: One deployment per PR using `concurrency.group`.
- **Stack Naming**: `cloudpp-pr-{number}-{branch}` ensures isolation.

4.6.2 Production (`prod-manual.yml`)

- **Trigger**: Manual `workflow_dispatch` with "prod" confirmation.
- **Stack**: Persistent `cloudpp-prod` stack.
- **Features**: Includes destroy action for complete teardown.

4.7 Key Architectural Decisions

Throughout the project's evolution, several significant architectural changes were made to improve performance, reduce operational complexity, and enable new capabilities. The following decisions shaped the current architecture:

4.7.1 Migration from RDS to OpenSearch Serverless

The initial architecture used Amazon RDS (Aurora) to store image metadata. This approach required:

- Custom SQL queries for text search with `LIKE` operators.
- Manual index management for performance optimization.
- Operational overhead for database scaling and backups.

OpenSearch Serverless was adopted because it provides both **full-text search** (native inverted indexes for efficient text queries) and **vector database** capabilities (for storing and querying embedding vectors). This dual capability enables sophisticated image-to-image similarity searches using ML embeddings, which would require significant custom development with RDS. Additionally, OpenSearch Serverless is fully managed, eliminating database administration overhead.

4.7.2 VPC-Bound Lambda Execution

Required for OpenSearch Serverless VPC endpoint access. Accepts cold start penalty (2-5s) for enhanced security and compliance with data isolation requirements.

4.7.3 Nested CloudFormation Stacks

Enables team parallelism—different engineers can own different stacks. Each stack can be updated independently without affecting unrelated resources.

4.7.4 Ephemeral PR Environments

Integration tests run against real infrastructure, catching issues that mocks would miss. Each PR gets an isolated stack that is automatically destroyed after testing.

4.7.5 Open Data Integration

The `similar_image` Lambda integrates with external open data sources (Wikimedia Commons APIs, Library of Congress APIs) to find visually similar images. Rather than hosting and maintaining our own copy of these open datasets—which would require significant storage infrastructure, data synchronization pipelines, and ongoing maintenance—we opted to consume their public APIs directly. This approach was particularly well-suited to the project's scope and timeline, allowing us to deliver the similarity search feature efficiently while leveraging the datasets' native search capabilities. The trade-off is external API dependency and rate limits, but the reduced infrastructure complexity made this the pragmatic choice.

5 Technologies and Open Datasets

5.1 Development Technologies

To kick off the project, we selected a forward-looking stack built on emerging standards. The backend uses FastAPI managed by uv, while the frontend leverages Bun and TanStack Start as high-performance alternatives to traditional frameworks.

To ensure robustness, we integrated a modern local CI workflow including Ruff (linting/formatting), Mypy (static typing), and Pre-commit hooks. The technologies used are shown in the table below.

Table 6: Software Development Technologies

Technology	**Domain**	**Alternatives**	**Justification**
FastAPI	Backend Framework	Django, Flask	Chosen for high-performance async support and automatic OpenAPI documentation.
uv	Package Manager	Pip, Poetry	Significantly faster dependency resolution (written in Rust), optimizing CI/CD build times.
Docker	Containerization	Podman, Containerd	Ensures environment parity between local development and the EC2 production instance.
Boto3	AWS SDK	AWS CLI	Exposes a comprehensive object-oriented API for AWS services, facilitating programmatic infrastructure control and automation.
Pydantic	Data Validation	Marshmallow	Native integration with Python type hints reduces boilerplate and ensures strictly typed data exchange.
Redis	In-memory Data Store	Memcached	Delivers sub-millisecond latency via in-memory storage and supports complex data structures for high-performance caching.
Ruff	Linter	Flake8, Pylint	Replaces multiple tools with a single, fast Rust-based tool, simplifying the pre-commit pipeline.

Technology	Domain	Alternatives	Justification
mypy	Static Type Checker	Pyright	Performs static analysis to enforce type consistency, detecting potential errors early to prevent runtime failures.
Pre-commit	Git Hooks	Husky, Lefthook	Enforces code quality standards (Ruff, Mypy) locally before code is pushed.
TanStack Start	Full-stack Framework	Next.js	Leverages TanStack Router for end-to-end type safety and offers server functions for seamless full-stack development.
Bun	Frontend Runtime	(Partial) Vite, npm, pnpm, yarn	Superior speed in installing packages and building the React application compared to Node.js.

5.2 AWS Services

The infrastructure manages a decoupled image processing workflow using independent Lambda functions orchestrated by SQS.

Data and security rely on OpenSearch (metadata), ElastiCache (pagination), and Cognito (identity). The backend is hosted via EC2 and API Gateway, while the frontend is delivered through S3 and CloudFront. Below can be found a summary of the AWS services we have integrated in our project.

Table 7: AWS Cloud Infrastructure Services

AWS Service	Function	Alternatives	Justification
EC2	Backend Compute	Fargate, App Runner	Provides full control over the Docker environment and OS for the FastAPI backend.
Lambda	Event Processing	AWS Batch	Handles spiky workloads like image processing; costs incurred only during execution.
S3	Storage	EFS, Glacier	Industry standard for durable, cost-effective storage of user images and logs.
CloudFront	CDN	Akamai, Cloudflare	Delivers low-latency, globally distributed content for static and dynamic assets, enhancing performance and scalability for a global audience.
VPC	Networking	None (Core Service)	Provides critical network isolation for the backend, databases, and private subnets.
SQS	Messaging	SNS, EventBridge	Decouples ingestion from processing to ensure reliable task execution and retries.
OpenSearch	Search Engine	DynamoDB, Aurora	Handles complex search queries on image metadata more efficiently than standard databases.

AWS Service	Function	Alternatives	Justification
ElastiCache	Caching	Memcached	Reduces database latency and supports complex data structures for pagination.
Cognito	Authentication	Auth0, Custom DB	Managed service for secure user sign-up/in and token generation.
API Gateway	API Entry Point	Load Balancer	Offloads rate limiting, routing, and traffic management from the application code.
CloudWatch	Monitoring & Logging	Datadog, Prometheus	Centralizes application logs, metrics, and alerts for better operational insights and monitoring.
Systems Manager	Management	SSH Keys (Bastion)	Allows secure shell access to EC2 instances without opening SSH ports (Port 22).
CloudFormation	IaC	Terraform, CDK	Native integration ensures consistent infrastructure state and simplified deployment.
IAM	Identity & Access Mgmt	Active Directory, Okta	Centralized identity management and fine-grained access control for AWS resources.

5.3 Open Data Sets

A very important feature of our application is the suggestion of images that are similar to the one the user is looking at. As seen in table 1 we have a serverless lambda to implement this functionality. This component operates by retrieving the semantic metadata of the source image, specifically usage labels previously extracted by AWS Rekognition and indexed in OpenSearch. It prioritizes the highest-confidence labels to generate context-aware search keywords, which are then used to query external open data repositories—specifically Wikimedia Commons and the Library of Congress through API. This process allows us to enrich the user experience by dynamically aggregating and presenting semantically related content availability in the public domain.

In this project, we utilize two primary open data sources for image retrieval:

- **Wikimedia Commons:** this is a media file repository that makes public domain and freely-licensed educational media content available to everyone. It is hosted by the Wikimedia Foundation and contains over 90 million community-contributed images and media files.
- **Library of Congress:** this is the research library that officially serves the United States Congress. It provides open access to extensive digital collections, including a vast archive of public domain U.S. Government historical images.

Dataset Name	**Brief Description**	**Organization**	**API Call (Base URL)**
Wikimedia Commons	A centralized repository of over 90 million freely usable media files and public domain content.	Wikimedia Foundation	`https://commons.wikimedia.org/w/api.php`
Library of Congress	The research library of the U.S. Congress providing access to extensive digital collections and historical images.	Library of Congress	`https://www.loc.gov/photos/`

Table 8: Summary of Open Datasets Used

6 Development Workflow

To develop this project we have followed a set of rules, agreements and workflows to maintain consistency and organize the work. A brief explanation is provided in this section.

6.1 Workflow Followed to Develop the Project

- **Monorepo Setup:** in order to have less organizational work we decided to have only one repository for the frontend, the backend and the infrastructure. This allowed us to have all the issues in the same place and all the code easily accessible to everyone.
- **GitOps:** we have CI/CD in GitHub Actions in order to test each PR we do to develop and main. By running the integration tests and actually deploying the infrastructure to AWS we can make sure that it is working when the pipeline returns successful.
- **GitHub Secrets:** we used GitHub secrets to store important information such as our AWS account's credentials. They are needed to be able to run the integration tests in the CI/CD pipeline.
- **GitHub Issues:** we create issues as soon as we see a necessity such a missing feature, a misconfiguration, something not working as expected, etc. Issues are closed once the feature/fix has been merged to develop.
- **Pull Request merging:** before merging a PR to develop or to main it needs to be reviewed by at least 1 other person, and that other person must approve. If they request changes those must be resolved.
- **Git Branches:** one branch for each GitHub issue has been created, so for each feature or fix we had to do a new branch was created and upon finish we would create the PR.
- **Version Management:** each week we had a "sprint" called milestone in GitHub. We had 3 different milestones: 0.1.0, 0.2.0 and 0.3.0. The branches called the same as each of the milestones are like a snapshot of the develop branch when each of the milestones ended.

6.2 IaC Compliance

We followed the IaC design methodology, as shown below:

- **Declarative Resource Definition:** The entire AWS infrastructure (including EC2 instances, Cognito User Pools, and API Gateways) is defined in declarative CloudFormation templates (e.g., `backend.yaml`, `main.yaml`), ensuring the desired state is codified rather than manually configured.
- **Automated Deployment Pipeline:** The `deploy.yml` GitHub Actions workflow automatically executes `aws cloudformation deploy` commands upon PR creation or update that contain the "deploy" tag (to prevent unauthorized use), ensuring that infrastructure changes are applied consistently and without human error during the CI/CD process.
- **Modular Architecture:** The infrastructure is modularized using a master stack (`main.yaml`) that references nested stacks (e.g., `VpcStack`, `BackendStack`, `ElastiCacheStack`), which promotes code reuse and maintainability characteristic of mature IaC setups.
- **Immutable Artifact Integration:** The deployment process builds immutable Docker images tagged with the commit SHA and pushes them to ECR before updating the CloudFormation stack to reference these specific tags, ensuring strict configuration management between code and infrastructure.
- **Version Control:** The `README.md` explicitly lists an `infrastructure` directory as part of the project structure, confirming that infrastructure definitions are stored and versioned in the git repository alongside the application source code. It also allowed us to somewhat distribute the load of work across the weeks.

6.3 Possible Problems

The workflow we have followed helped us avoid many problems that could have made the development slower or face more difficulties. Below some can be found for not following the method IaC and our workflow:

- **Configuration Drift:** Without the strict definitions in backend.yaml or main.yaml, manual changes made in the AWS Console (e.g., tweaking security groups or memory limits) would lead to environments (Dev vs. Main) becoming inconsistent over time, causing "it works on my machine" bugs.
- **Slow & Error-Prone Deployments:** The automated pipeline in deploy.yml handles complex tasks like creating repositories, pushing Docker images, and updating stacks. Doing this manually for every release would be tedious and highly susceptible to human error (e.g., forgetting to update a Lambda environment variable).

- **Lack of Disaster Recovery:** If the stack were deleted or a region failed, rebuilding the complex web of dependencies—EC2, Redis, OpenSearch, and multiple Lambdas—would be a manual, time-consuming nightmare without the "blueprint" provided by the templates.
- **Security Vulnerabilities:** Manual configuration often leads to oversight. The templates enforce specific security policies, such as the BackendSecurityGroup allowing only HTTP access or IAM roles with least-privilege policies like InvokeLambdaPolicy. Manual setups might accidentally leave ports open or grant overly broad permissions.
- **Issues and PR Management:** it would have been more difficult o maintain 3 different repositories, one for frontend, one for backend and one for infrastructure. Instead, we kept it as a monorepo.
- **PR Reviews:** if at least one person didn't review the PRs everything that would be pushed to develop would not work and would have security vulnerabilities. It was good practice to review the code as in many occassions it proved to be useful to find grave misconfiguration issues.
- **Branch Management:** we had to make sure that we could work in parallel as it would make our work much easier. This is why we created different branches for each of the features. It also helps keep consistency of our tests when new code is added.

7 Validation

7.1 Unit Tests

We have added unit tests to our backend and lambdas to make sure they work correctly before pushing them to the origin remote branch. The way to execute these tests appears on the README.md using the uv command, which can be found in the repository of the project. It is also executed in the CI/CD pipeline in case the developer forgets to run them in local.

However, unit tests are not the most reliable way to check if the code is working correctly, specially for something so sophisticated as deploying infrastructure. This is why we have added integration tests.

7.2 Integration Tests

The integration tests in this project are run in the CI/CD pipeline we have in GitHub Actions. A developer creates a PR and before merging the integration tests are run to make sure the code with the modifications works correctly.

The way in which these integration tests check that everything is working has its own workflow. Below the steps are explained and in table 4 the workflow can be visually appreciated.

- Runs unit tests (in case the developer forgot to run them locally) and performs a static analysis of the code to make sure the code quality standards are met.
- The docker image is built with its corresponding tag and pushed to Amazon ECR.
- A new isolated environment is created and the Cloudformation stack is launched including the VPC, Opensearch cluster, lambda functions, etc.
- Then we use SSM to start the EC2 instance for the backend. We also use this stage to create Redis endpoints and cognito IDs.
- After it is deployed, we check that this live infrastructure is working as expected by running these integration tests against it.

- Finally all services and data are cleaned up forcefully, emptying S3 buckets, the CloudFormation stack and the ECR repository.

If the pipeline fails means that one of more of the tests failed, meaning the developer needs to keep trying until it is successful. Once the pipeline returns successful then the developer asks for a review from at least 1 other person. If that person approves the PR then it can be merged, otherwise changes need to be applied before merging.

This workflow and good practice to run integration tests ensure a stable and working environment even for development.

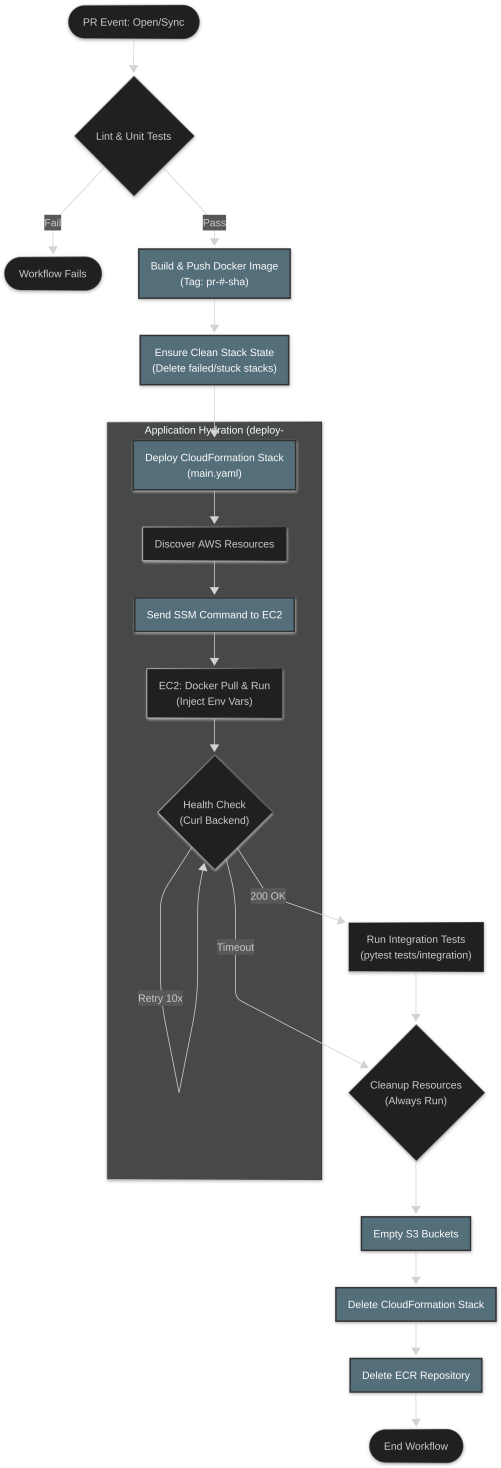

Figure 4: CI/CD pipeline workflow for running integration tests

7.3 12 Factor Methodology

We consider the application is highly compliant with the 12-factor methodology, with minor deviations in how concurrency is managed in the current deployment configuration. A table with the evaluation of each of these principles can be found below.

Table 9: 12-Factor App Compliance Assessment

Factor	Status	Reasoning	Evidence
1. Codebase	**Yes**	One codebase tracked in revision control, many deploys.	The project uses a single Git repository (monorepo). `deploy.yml` manages deployments to different environments (dev, main) from this single source.
2. Dependencies	**Yes**	Dependencies are explicitly declared and isolated.	Dependencies are locked using `uv sync` and declared in `pyproject.toml`. The Dockerfile copies these specifically.
3. Config	**Yes**	Store config in the environment.	Configuration variables (e.g., `REDIS_HOST`, AWS regions) are injected as Environment Variables into the Docker container via `backend.yaml`.
4. Backing Services	**Yes**	Treat backing services as attached resources.	Redis, OpenSearch, and S3 are defined as loose dependencies in `main.yaml` and connected via endpoints passed at runtime.
5. Build, Release, Run	**Yes**	Strictly separate build and run stages.	The pipeline has a distinct "Build" phase (Docker image with unique SHA tag) and "Deploy" phase (CloudFormation update).
6. Processes	**Yes**	Execute the app as one or more stateless processes.	The backend runs as a stateless Docker container. State is offloaded to S3, OpenSearch, or Redis.
7. Port Binding	**Yes**	Export services via port binding.	The `Dockerfile` explicitly exposes port 8000 and the application listens on this port.

Factor	Status	Reasoning	Evidence
8. Concurrency	**Partial**	Scale out via the process model.	**Design:** Stateless and scalable. **Implementation:** Deploys a single `AWS::EC2::Instance` instead of an Auto Scaling Group.
9. Disposability	**Yes**	Maximize robustness with fast startup.	Uses `python:3.11-slim` for lightweight footprint. User Data scripts handle initialization on boot.
10. Dev/Prod Parity	**Yes**	Keep dev, staging, and prod similar.	Developers use `uv sync` locally, matching the Docker build. The same templates are used for dev and main.
11. Logs	**Yes**	Treat logs as event streams.	`logging.yaml` defines a central bucket for logs. Backend likely logs to stdout.
12. Admin Processes	**Yes**	Run admin tasks as one-off processes.	Admin tasks (e.g., infrastructure setup) are handled via CloudFormation or Lambdas, not manual SSH.

7.4 User Requirements Coverage

All user requirements have been met, as we are our own stakeholders. We set a scope for this project and we have respected it. The application has all the features we wanted it to have and is working as expected.

8 Challenges

8.1 Limited Computation Time on GitHub

While testing all the infrastructure was being correctly deployed and the code was working on the 12th of December we encountered a limit. GitHub free plan only allows for a very limited computation time for pipelines, so we could no longer test our code. Our application takes quite a while to be completely deployed, being Opensearch the bottleneck. This is not something we can handle, it is just how much Amazon Opensearch takes to be up and running.

To fix it we tried to fork the repository but it did not work, as this feature is disabled in the repository that was shared with us. What we did was create our own with the same code and run the pipelines with our own GitHub accounts. This is the reason why there is a big gap between 2 different days in testing in the pipelines.

8.2 Limited computation on Overleaf

The tool we are using to write this report is Overleaf. It is a latex editor that allows you to write content without having to worry about the format because you can set it once and it is applied everywhere. It is super useful for big projects in which a lot of people work or the document is so long you can not keep the format consistent everywhere. The problem we encountered is that the free plan was too little to compile the document, so we had to upgrade the subscription to the next tier by entering our credit card's details and paying for the next tier.

8.3 Limited time to develop and deploy

We have had very little time to design, plan, develop and test such a big project. We defined and limited the scope to the minimum, but it still took a lot of work because of how easy it is to misconfigure a setting and not having the deployment work. It would have been better if we had more than 1 month, because we still needed the approval from the lab teacher.

Having limited time also made it more difficult to make this document. There were issues being closed, ongoing and PRs being merged at the same time of the making of this document, very close to the release date. Meaning we had to modify different sections of this document at the moment the source code was updated. We believe it is not due to lack of planning, but the failure of following the planning through.

8.4 Free AWS Account being too Restricted

We wanted to add Opensearch as a full text search and vector database indexer for our application. Its main role would be that when a person searches for an image its metadata that was saved in Opensearch would allow this image to be found. However, we saw that the AWS free account did not have this service included. Since all the design and the idea was already thought out, we had to enter our credit card's details.

We could have moved on with another idea and not having to use this service, but it had already been discussed with the lab teacher and he liked the idea. In addition, if we manage it correctly we should not have spent a single dime from our bank accounts.

8.5 AWS Opensearch's High Provisioning Time

We saw that our deployment took very long every time consistently. After trying and retrying and debugging we found out that it was not because of our code or slow deployment, but because Amazon Opensearch takes **at least** 15-30 minutes to initialize from scratch. To prevent high resource consumption we turn off the Opensearch service each time we do not need it, but at the cost of it taking very long to initialize.

After having wasted days on trying a fix for this we gave up and accepted it will take this long to deploy everything from scratch.

8.6 Mono Repository Maintainability

It was mentioned previously that having a mono repo was intended to reduce coordination and management work for the frontend, backend and infrastructure. But having a mono repo also had its disadvantages. We had many merge conflicts in our code and the README had to be updated every single time because they were big changes.

The directory structure was intended for it to be a mono repo at the beginning but due to many design decisions later it ended up being different from the initial purpose.

8.7 IAM Permission Errors

We had a ton of permission errors at the beginning because we were trying to limit it to the fullest. Since we were not going everywhere, we decided to first have a wroking version with allowing all and then little by little do different tests removing certain permissions for each of the lambdas, until it worked with only the permissions that each component and service requires.

8.8 Learning New Technologies from Scratch

A significant challenge we encountered was that many of the technologies we used were unfamiliar to the team. From AWS services like CloudFormation and Opensearch to handling complex deployment pipelines on GitHub, we had to learn how to use these tools from the ground up. This not only added to our learning curve but also significantly slowed down the development process.

While we were able to get things working in the end, the time spent figuring out configurations, documentation, and troubleshooting issues that arose from our inexperience meant that tasks that might otherwise be straightforward took much longer to complete. This delay was compounded by the fact that we had very little time to begin with, making it even more critical to get up to speed quickly.

9 Total Costs

By having most of the functionalities in serverless lambdas we can minimize the cost of the infrastructure. This way the backend handles very basic functionalities such as caching. We also optimized the usage of resources by turning all the deployment down (including Opensearch which takes around 30-45 minutes to be started every time) when the pipeline was finished running the tests in develop. Moreover, by restricting the permissions for each service in AWS we can ensure that no one has been able to utilize our resources without permission.

The total cost of this project for December can be found in the figure below. In the chart we can also have an estimate value of November's cost.

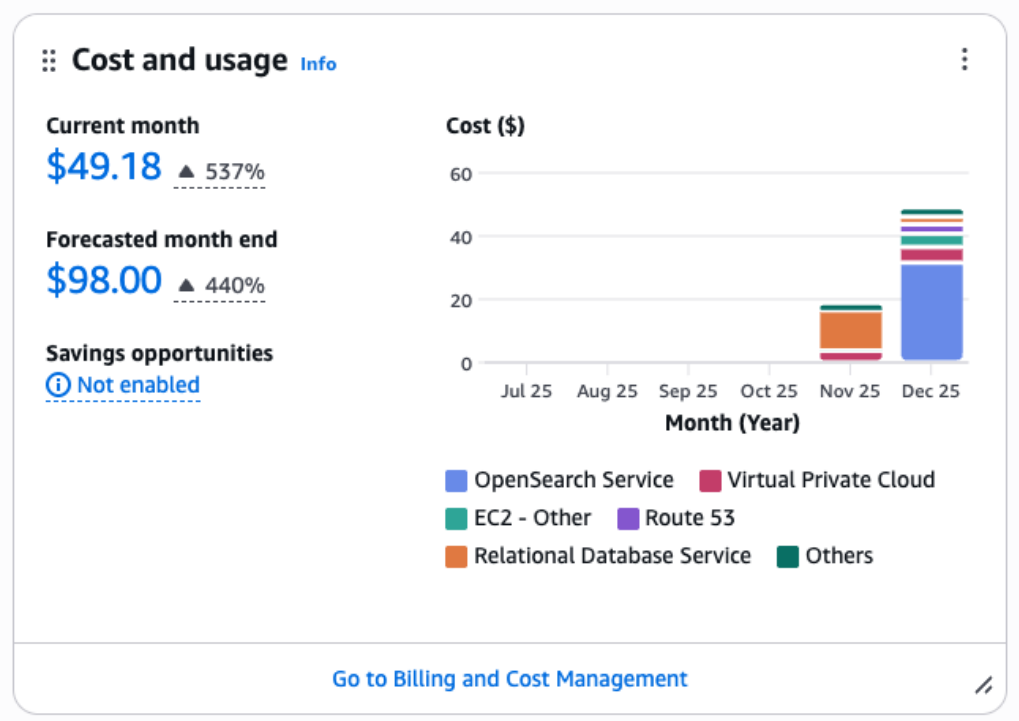

Figure 5: Visual appreciation of the cost for December

In the next figure we can see the cost for both months, meaning the cost of the whole project. We have not consumed more dollars than the free ones given to us thanks to the free educational AWS account.

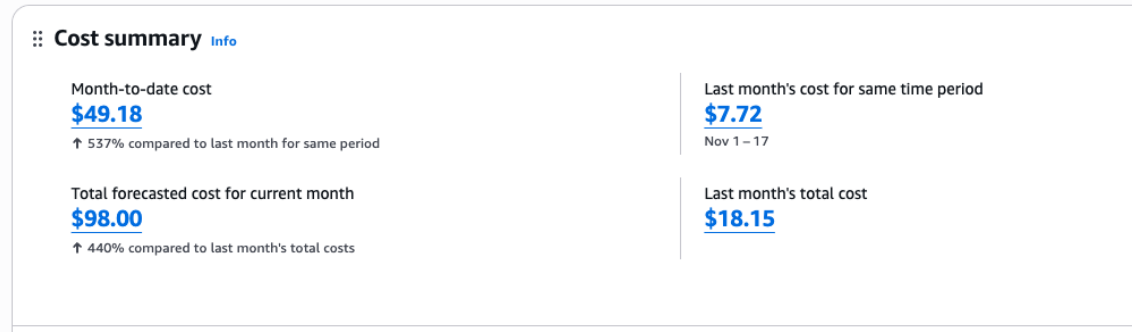

Figure 6: Screenshot of the cost for both November and December

10 Future Considerations

10.1 Improvement Vectors

Below there is a list of things that we think that could be improved in our project. They are either things that we wanted to do but did not have the time to, or more complex things that required more time to do than the one we had.

- **Job Queueing:** we thought it would be nice to implement a system that managed all the jobs, making our application completely asynchronous. Currently we need to wait for the lambda to be created and up, but by having jobs management we could leave the job system to manage the lambda's execution, decoupling the request from the execution.
- **EKS for High Availability:** by adding EKS we could have automatic horizontal autoscaling and a very reliable orchestrator, but it complicates the infrastructure a ton and we did not have to include this.
- **Lifecycle of Logs:** right now we do not have a way to delete old logs or store them in a less expensive storage option such as Glacier. It would be a nice improvement to the application. But we decided to not add it because for now the application is in the development state, and as such each deployment is independent of each other and they all start from scratch, meaning logs will not pile up in the S3 bucket.
- **Mobile version for Easy Access:** if we had the time we could've developed a mobile version that allows to do all this picture management from your phone and by having one account share it across all devices and saving the configuration for your account.
- **Add Image Filtering by Metadata:** we have a way to search for an image by entering text, but maybe you want to search by using filters such as resolution 4k, location: Barcelona, etc. Having direct filters would help Opensearch a lot because these are not user inputs but predefined deterministic searching options.

10.2 Viability

We think that our application is highly maintainable due to how well organized the repository is, how well documented all the features and deployments are, and how much we stick to the principle IaC. The only futrue limitation we foresee is the lack of polishment. Due to having very little time to do it, we focused more on the technical aspects of the project and not in the user's experience when using the application. The user interface and some useful features might be missing because of this.

11 Conclusion

Through the development of this project, CloudPP, we have gained deep knowledge into the advantages and limits of cloud-native architecture, navigating the vast ecosystem of AWS services to build a robust and scalable platform. We discovered that mindful planning and regular architectural synchronization were just as critical as the implementation itself, allowing us to successfully orchestrate complex interactions between serverless functions, containerized services, and managed data stores.

Moving forward, we will follow and have in mind the Infrastructure as Code (IaC) and automated CI/CD practices mastered during this project, as they proved to be fundamental in preventing configuration drift and ensuring reliable deployments. The lessons learned regarding the 12-Factor App methodology and modular design will serve as a definitive blueprint for our future engineering projects, guiding us to build software that is not only functional but maintainable and resilient.

12 References

Amazon Web Services, Inc. "Amazon EKS Autoscaling." *Amazon EKS User Guide*, https://docs.aws.amazon.com/eks/latest/userguide/autoscaling.html. Accessed 15 Dec. 2025.

Amazon Web Services, Inc. "Amazon Rekognition - Automate Image and Video Analysis with Machine Learning." *AWS*, 2025, https://aws.amazon.com/rekognition/. Accessed 20 Nov. 2025.

Amazon Web Services, Inc. "Searching Data in Amazon OpenSearch Service." *Amazon OpenSearch Service Developer Guide*, https://docs.aws.amazon.com/opensearch-service/latest/developerguide/search.html. Accessed 28 Nov. 2025.

Amazon Web Services, Inc. "Serverless Image Handler." *AWS Solutions Library*, https://aws.amazon.com/solutions/implementations/serverless-image-handler/. Accessed 22 Nov. 2025.

Amazon Web Services, Inc. "Web Application Hosting in the AWS Cloud." *AWS Whitepapers*, https://docs.aws.amazon.com/whitepapers/latest/web-application-hosting/web-application-hosting.html. Accessed 1 Dec. 2025.

Astral Software Inc. "Ruff." *Astral Docs*, https://docs.astral.sh/ruff/. Accessed 27 Nov. 2025.

Astral Software Inc. "uv." *GitHub*, https://github.com/astral-sh/uv. Accessed 17 Dec. 2025.

Linsley, Tanner. "TanStack Start." *TanStack*, https://tanstack.com/start/latest. Accessed 26 Nov. 2025.

Oven. "Bun." *Bun*, https://bun.sh/. Accessed 26 Nov. 2025.

Ramirez, Sebastián. "FastAPI." *FastAPI*, https://fastapi.tiangolo.com/. Accessed 3 Dec. 2025.

Redis Ltd. "Redis." *Redis*, https://redis.io/. Accessed 1 Dec. 2025.