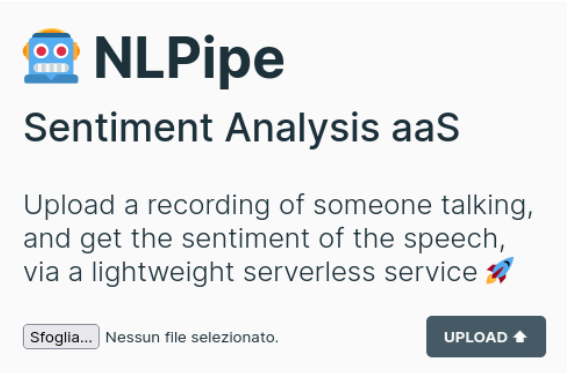
NLPipe: a scalable infrastructure to deploy NLP solvers

Edoardo Gabrielli, gabrielli.1693726@studenti.uniroma1.it
Davide Quaranta, quaranta.1715742@studenti.uniroma1.it
Daniele Solombrino, solombrino.1743111@studenti.uniroma1.it

Introduction

NLPipe is a scalable cloud infrastructure to deploy NLP task solvers on.
Specifically, this project focuses on speech recognition and sentiment analysis.
Scalability, adaptability, modularity, OS and programming language agnosticity have been taken into consideration during the entire development process, together with security and privacy.



The **requirements** of the system are:

- `R.F.1`: The user can upload audio files.
- `R.F.2`: The user can request status updates regarding the uploaded file.
- `R.F.3`: The system should extract the text of the audio file.
- `R.F.4`: The system should perform sentiment analysis of the extracted text.
- `R.F.5`: The system should store the result of `R.F.4`.

The system should also satisfy the following non-functional requirements:

- **Security**: enforce the minimum privilege principle.
- **Privacy**: user data should be anonymized.

Later in the appropriate sections, a description of how non-functional requirements have been handled will be provided.

Desired characteristics

During the design stage, the following driving principles have been taken into account:

- **Scalability**: the system should be able to automatically scale horizontally, when needed.
- **Adaptability**: the system can be easily adapted to other NLP tasks, not only sentiment analysis, with minimal effort.
- **Modularity**: to enforce adaptability, each part of the system is an independent component designed to carry out a specific task.
- **OS and language agnostic**: the choice of programming languages for a system component should not be restricted by the choice for another component. Moreover, the system should be able to run regardless of the host machine's operating system.

Design

The system is composed of multiple independent components that interact with each other in order to pass and retrieve data and to store the final result. Specifically:

- A **web interface** that allows the user to upload audio files.
- A **REST API** that handles the interaction between the web interface and the backend.
- A **pipeline** of functions, one for each step:
  - Speech to text.
  - Text sentiment analysis.
  - Logic to store final results.
- A **database** to store the result of the performed sentiment analysis.

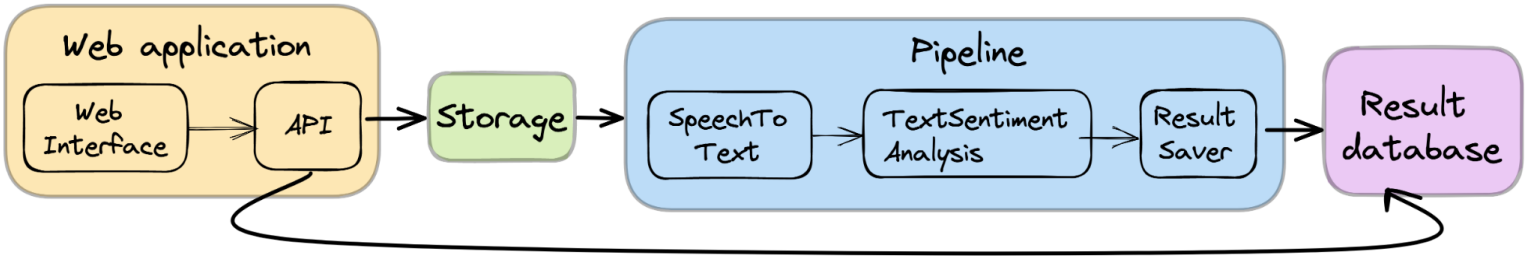


Fig. 1: high level representation of the NLPipe design

Web application: Web Interface and API

To address requirements `R.F.1` and `R.F.2`, the web application includes:

- A page containing an **upload form**.
- A page that periodically polls an API endpoint, to get information about the user's request.
- A service to expose a **REST API**, queried by the web pages and interacting with **backend** services.

Upon successful audio upload, the API generates a random **UUID** and associates it to the request, in order to uniquely identify the uploaded audio file in every following operation.

Additionally, this service interacts with backend services to store the file for processing, and to create a **database row** for that UUID.

Storage

In order to elaborate user requests, the uploaded audio files need to be:

- Temporarily stored in a location that can offer different controls for read and write permissions, to enforce the **minimum privilege** design.
- Uniquely and consistently identified.

Pipeline: Speech2Text, TextSentimentAnalysis, ResultSaver

The pipeline can be seen as a layered design. Generally, each layer:

- Gets the output of the previous layer and uses it as its input.
- Uses the obtained input to perform a specific task, in order to comply with requirements `R.F.3`, `R.F.4` and `R.F.5` (speech-to-text, text sentiment analysis and data storage).
- Sends its output to the next layer.

Specifically:

- **Speech2Text** gets the audio file from the storage space, converts it into text and sends the transcript to TextSentimentAnalysis.
- **TextSentimentAnalysis** performs a sentiment analysis on the received text and sends the computed sentiment to ResultSaver;
- **ResultSaver** stores the result in the results database.

These tasks are **lightweight** and **loosely coupled**: a **serverless** approach can be used.

Database

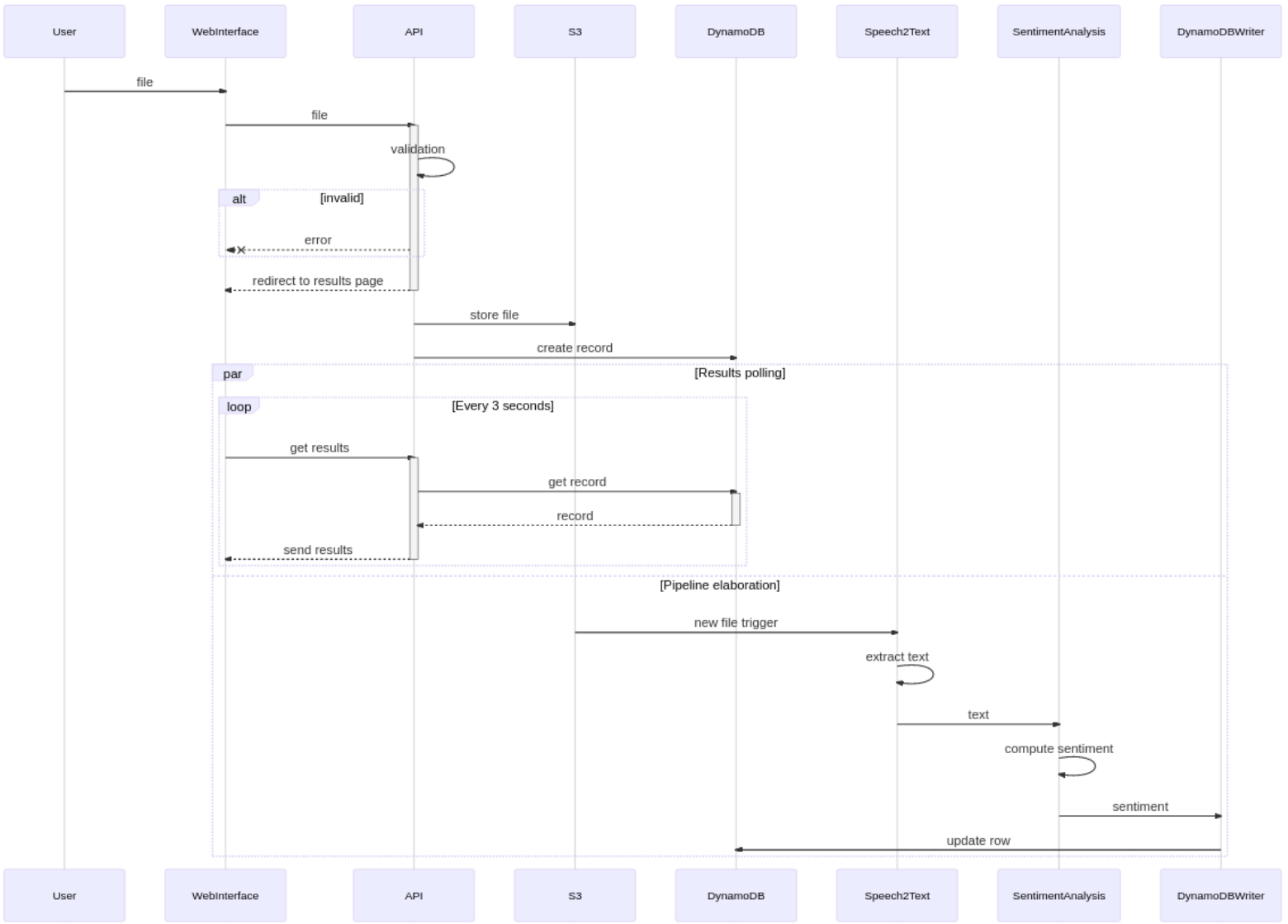
A read and write database is needed. As per **minimum privilege** design:

- The API needs write access to the DB, in order to create the record upon user upload. The API also needs read privileges, in order to understand whether a computation has been completed or not, and to present status updates.
- The DB writing function needs a write access, in order to actually be able to write data in the database.

A **NoSQL** database can be used, since entities, relationships, constraints and ACID properties are not required.

Implementation

The development first happened locally, in order to better test the single components and their interactions (depicted in `Fig. 2`). The system has been designed and implemented as **cloud-native** for AWS; as a result, it can be easily deployed there.



`Fig. 2: NLPipe sequence diagram`

Web application: Interface and API

The service is divided into the following independent parts, with separated concerns:

- A **front-facing** web application written in plain HTML, CSS, JavaScript, composed by:
    - An upload page `/` with a `POST` form to `/api/upload`
    - A results page `/results.html`
- A **back-end** written in **Go**, that:
    - Exposes a **REST API** composed by the endpoints:
        - `POST /api/upload`

        - `GET /api/results/:uuid`
    - Interacts with:
        - Amazon **S3** to store the uploaded file.
        - Amazon **DynamoDB** to create the row for the file/request.

Upon a successful upload, the back-end creates a **UUID** that is used to uniquely identify the request/file and the API redirects the user to `/results.html?uuid=x`, where `x` is the UUID that has just been generated.
All bad requests are blocked and **errors are notified** to the user.

Docker has been used to facilitate the development, also by ensuring that each developer has the same environment. The **`docker-compose.yml`** file describes the required services, which are **s3mock**[1] and **dynamodb-local**[2], respectively needed to simulate the S3 API and to have a self-hostable version of DynamoDB.

The **Dockerfile** is thoroughly described in the deployment chapter.

Storage

To store the audio files, an **Object storage** like Amazon S3 is appropriate.
Files are renamed with the **UUID** that is generated by the API, upon successful upload.
As introduced before, to develop locally it was used s3mock.

Pipeline

All the steps are written in **Python 3.** Required external files, tools and libraries have been imported using Python 3 specific functions.

Speech2Text and TextSentimentAnalysis use **pre-trained NLP Deep Learning models**.

Speech2Text

Speech2Text model is **provided by Google** and supports 125 languages with automatic language recognition, noisy audios, punctuation and different kinds of audio formats.

Input audio must be encoded in Base64, before being fed to the model.
The model (and thus the program) **returns a string with the audio transcription**.

TextSentimentAnalysis

TextSentimentAnalysis adopts a **Naive Bayes Classifier** (> 90% accuracy): 80% of input data has been used in training, the remaining 20% for testing. Model and input data used have been **provided by the NLTK library**.

---

[1] https://hub.docker.com/r/adobe/s3mock
[2] https://hub.docker.com/r/amazon/dynamodb-local

After tokenization, features are extracted from the input string, in order to query the classifier.
The model **returns a string** that can be "**`positive`**" or "**`negative`**", corresponding to the result of the classification.

ResultSaver

ResultSaver creates a **JSON** string, which will be used to store the result of the entire computation in a database; the string contains two pieces of information: the name of the input file audio (**UUID**) and its computed **sentiment**.

Intra-pipeline communication

Components inside the pipeline need a form of communication: in this local development stage, all the functions are called from a monolithic controller program, which saves returned data and passes it to the following component.

Database

For each UUID, the database should store the status and (if finished), the sentiment.

Relations are not needed, neither ACID transactions or integrity constraints, so a NoSQL database like DynamoDB is appropriate. Moreover, the design of DynamoDB (optimistic replication, row-level atomicity) allows massive horizontal scaling and high performance.

A typical item used in the project is:

```
{
      uuid String
      status String: <processing|finished|error>
      sentiment String: <positive|negative>
}
```

As introduced before, to develop locally it was used dynamodb-local.

Deployment on AWS

The system is **cloud-native for AWS**, so it fully uses the services offered by the cloud provider. The architecture can be synthesized as shown in the following diagram:

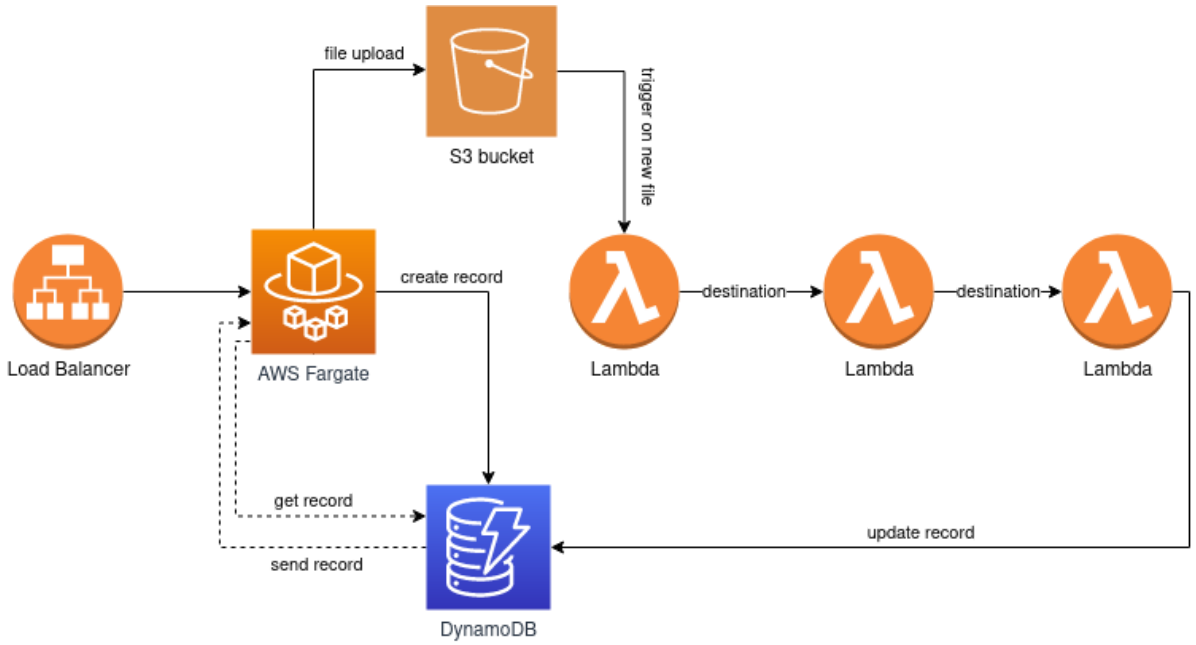


Fig. 2: NLPipe AWS architecture and interactions

IAM Roles: Web app, Storage, Pipeline and Results Database

IAM Roles manage various aspects of AWS product usage, by imposing policies that must be followed. NLPipe IAM Roles use the “least privilege principle” design and in addition to the default ones, other IAM policies imposed in NLPipe are:

NLPipe Component	**Policies**
Fargate	• Write into DynamoDB table `nlpipe`. • Read from DynamoDB table `nlpipe`. • Create new files in S3 bucket `uploads.nlpipe`. • Write on CloudWatch `/ecs/nlpipe-app`.
Speech2Text	• Read from S3 bucket `uploads.nlpipe`. • Write on CloudWatch `/lambda/speech2text`.
TextSentimentAnalysis	• Write on CloudWatch `/lambda/textsentimentanalysis`.
ResultSaver	• Write on CloudWatch `/lambda/resultsaver`. • Write to DynamoDB table `nlpipe`.

AWS Fargate: Web application (Interface and API)

Fargate is a managed service to deploy **containers** in a **serverless** fashion. Since the web application is built as a **Docker** image, it is sufficient to push it in an **Amazon ECS** (Elastic Container Service) repository, and run a *Task* described by a *Task Definition*.

To accomplish what said above, it is necessary to:

1. Create an ECS repository.
2. Build the Docker image of the web application.
3. Tag the built image with the repo URL.
4. Push the image.

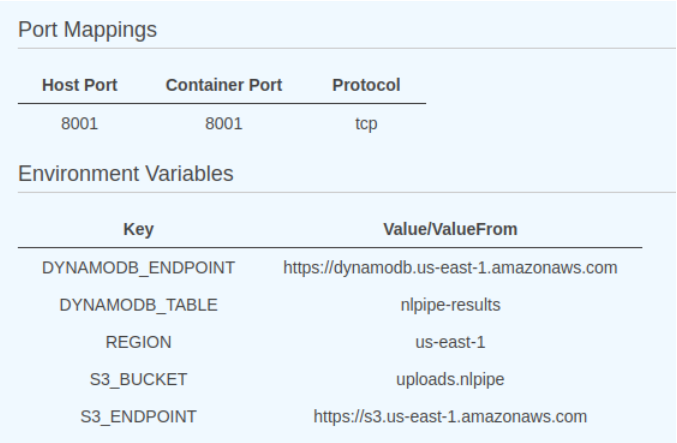
The building and pushing process can be easily automated via a simple Bash script. As introduced in the previous chapter, the **Dockerfile** is:

```
## STAGE 1
FROM golang:1.15 as builder

RUN apt update && apt install -y --no-install-recommends ca-certificates

ENV GO111MODULE=on \
    CGO_ENABLED=0 \
    GOOS=linux \
    GOARCH=amd64

WORKDIR /app
COPY . .
RUN go mod download

RUN go build

## STAGE 2
FROM scratch
COPY --from=builder /app/nlpipe /app/
COPY --from=builder /app/html /app/html
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/ca-certificates.crt

EXPOSE 8001

WORKDIR /app
ENTRYPOINT ["./nlpipe"]
```

The **multi-stage build**[3] technique is used to keep the image size as minimal as possible, and to avoid having unnecessary files in it.
As an example, the latest `nlpipe` image is 16MB, and only contains the built Go binary and the files needed for the front-end; in other words, the files specified in the `COPY` instructions in the second stage.

The pushed image is used to create a **Task Definition**, which contains settings like name, execution role, privileges, network mode, volumes, allocated resources.

[3] https://docs.docker.com/develop/develop-images/multistage-build/



Port Mappings

Host Port	Container Port	Protocol
8001	8001	tcp

Environment Variables

Key	Value/ValueFrom
DYNAMODB_ENDPOINT	https://dynamodb.us-east-1.amazonaws.com
DYNAMODB_TABLE	nlpipe-results
REGION	us-east-1
S3_BUCKET	uploads.nlpipe
S3_ENDPOINT	https://s3.us-east-1.amazonaws.com

Inside a Task Definition there is the **Container Definition**, which contains further settings like container name, image, port mapping, environment variables, etc.
Upon defining settings, a Task can be run.

Amazon S3: Storage

An S3 bucket is used to store audio files uploaded by users. To enforce security requirements, the bucket is **not publicly accessible**, IAM policies regulate its access and only the uploaded audio files are stored.
To save on archiving costs, the S3 bucket's **retention policy** is set to one day.

When a new file is uploaded in the bucket, an **AWS Event** automatically triggers the execution of Speech2Text and passes to it the name of the newly added file, which corresponds to the UUID set by the API.

AWS Lambda: Pipeline (Speech2Text, TextSentimentAnalysis and ResultSaver)

AWS Lambda is a managed service to execute code in a serverless way, perfectly blending in with the characteristics of pipeline components. A new Lambda is automatically launched for every new request and scaling is entirely managed by AWS.

Every call to an AWS Lambda function results in the creation of an execution **environment** and a **context**, solely dedicated to the specific call.
Environment includes a Unix File System, context is a dictionary of input arguments.

In Lambdas, only maximum **execution time** and maximum **RAM** can be customized. AWS will pick the number of vCPUs and network bandwidth according to allocated RAM.

A Lambda function can be automatically triggered from other AWS services, including AWS S3 (via **AWS Events**) and AWS Lambda (using **AWS Destinations**).

External **files**, **tools** and **libraries** can be put in **AWS Layers**: attaching the Layer to the Lambda allows the latter to reference the entire Layer content.

Intra-pipe **communication** between Lambdas is **automatically managed**, by means of placing data in the callee's context, for future retrieval.
In S3 triggers, information about the AWS Bucket is passed, whilst with AWS Destinations everything returned by "caller" Lambda is serialized.

Now to the Pipeline deployment characteristics.

Speech2Text

Whenever a new audio file is uploaded in the **AWS S3 bucket**, an **AWS Event** automatically starts `Speech2Text`, which retrieves the audio, stores it in its File System and performs the task, as per design and implementation chapters.

The NLP model is placed in and loaded from a **Layer**, linked to `Speech2Text` Lambda.

TextSentimentAnalysis

Upon `Speech2Text` completion, an **AWS Destination** automatically invokes TextSentimentAnalysis Lambda (data passing already described in paragraph intro).

The NLTK library and Naive Bayes Classifier model are placed in and loaded from separate, dedicated **Layers**, linked to `TextSentimentAnalysis` Lambda.

ResultSaver

After `TextSentimentAnalysis` successful termination, another **AWS Destination** autonomously calls ResultSaver Lambda (data passing described in paragraph intro).

Using AWS SDK, a connection with **DynamoDB** is established and, if successful, a **JSON** containing the formatted result is enqueued for writing.

Amazon DynamoDB: NoSQL database

DynamoDB is an high-performance key-value/document database, used to store requests status and text sentiment analysis results.

Table creation is a one step process. Writing uses **JSON** strings: for each key, DynamoDB creates a column in the table (if not present) and puts the value in the column.

Load Balancing and Autoscaling of the Web application

To make the **web application** (on AWS Fargate) **scale horizontally**, it is necessary to create a load balancer and to define an autoscaling policy.

ECS has an integrated way to do this:

1. Create a **Service** related to the Task Definition and to the Cluster.
2. Configure the Service's VPC and security groups settings.
3. Create an Application Load Balancer, listening on HTTP.
4. Attach created ALB to Service, map port 80 to 8001 (exposed by container)
5. Configure the Service Auto Scaling as desired.

Autoscaling policy

To easily test the scaling capabilities of the project, the following lowered-down thresholds have been set, as recommended in class notes and lectures.
Additional configurations have been taken into consideration, in an empirical study.

Scaling direction	CPU Usage (AVG)	For	Action	Cooldown
In	<= 10%	>= 1 min	remove 1 instance	1 min
Out	>= 25%	>= 1 min	add 1 instance	1 min

Test & Validation

Hypothesis to be proved

Tests in this chapter had the goal of validating three different assumptions, formulated from theoretical studies presented throughout the semester-long class:

1. The **cloud** enables easier **scalability,** opposed to a monolithic architecture.
2. In AWS **Lambdas**, allocated **RAM** and **execution time** are inversely correlated, since AWS allocates vCPUs according to allocated RAM and more RAM reduces swapping needs and speeds up external files loading.
3. Bad capacity planning has a bad impact on company monthly bills.

Conducted tests

Tests have been conducted using custom-made **Bash** scripts. Input data comes from Kaggle and has been pre-processed as needed.

The following table synthesizes all the conducted scaling tests.

Tested component	Requests	Type of testing	Delay between calls	Tools
Web Server	500	Sequential	500 ms	Bash scripts Postman Newman
Speech2Text	up to 2560	Parallel	not used	Bash scripts AWS CLI
TextSentimentAnalysis				

All of the above tests have been replicated multiple times, in order to have a more precise empirical process.

Local tests focused on gathering baseline benchmarks to match against AWS run times and were stored and processed locally.

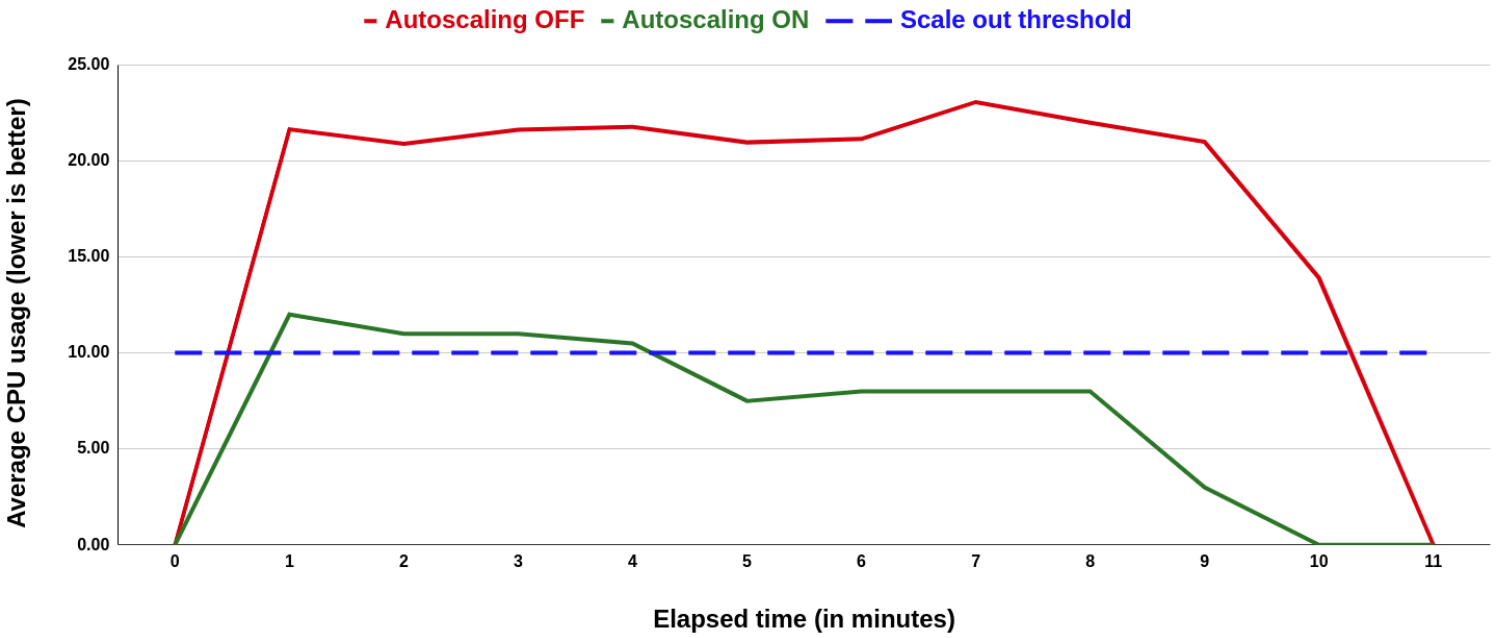
For the AWS calls, **CloudWatch** has been used to gather execution times and useful statistics.

Test results

Scalability

`Chart 1` compares average CPU utilization on Fargate, with and without autoscaling.

Specifically, without autoscaling, CPU usage peaks and stays high, going down only when the computation actually ends.
On the contrary, turning autoscaling on, after CPU usage peaks, autoscaling kicks in, bringing the average usage below the desired threshold.



`Chart 1: Web server, CPU usage, autoscaling on vs. off`

`Chart 2` shows a clear **correlation between number of parallel calls and run times** in **monolithic** contexts, resulting in execution times growing together with the number of parallel calls. On the contrary, on **AWS run times stay always the same**.

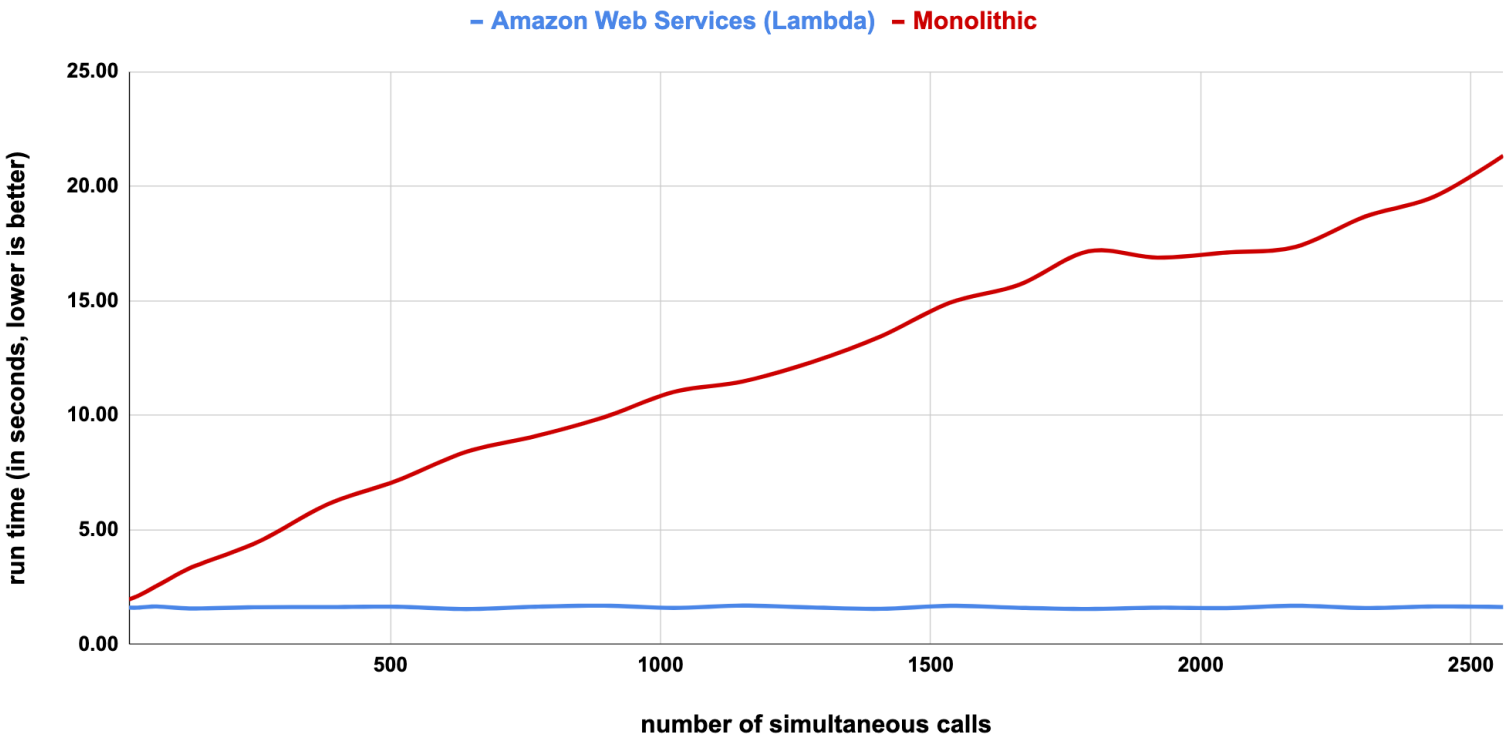


Chart 2: Speech2Text, Serverless (Lambda) vs. Monolithic architecture, number of parallel calls/execution times correlation.

Lambdas: Maximum RAM vs. run times

Charts 3 shows a clear **inverse correlation between maximum allocated RAM and run times in AWS Lambdas**: the more the RAM grows, the smaller execution times get.

The result is very interesting, especially since pipeline tasks always take approximately the same amount of RAM usage.
The phenomenon is probably related to how AWS provisions resources: the more memory is added, the more vCPU cores (and networking) are allocated.

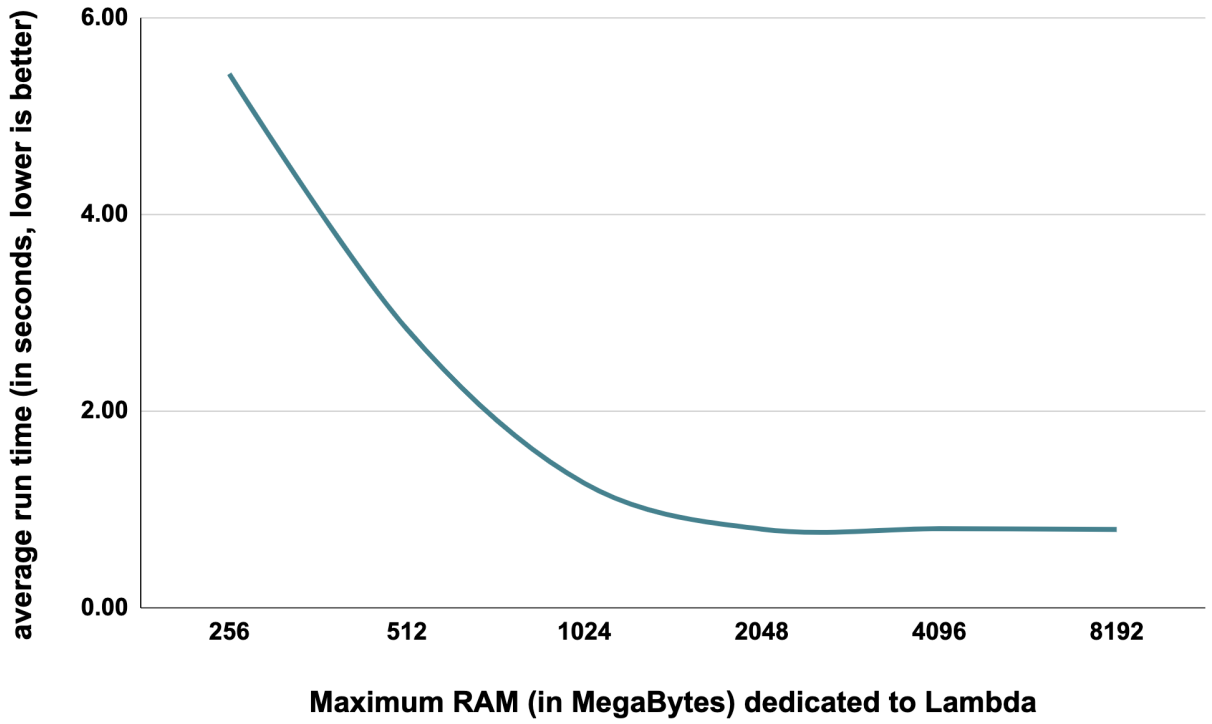


Chart 3: TextSentimentAnalysis, RAM/run times inverse correlation

Economical impact of bad capacity planning

Chart 4 shows the economical impact of bad capacity planning.
After a certain point, more RAM doesn't lead to any speedup, while billing costs continue to increase: the company wastes money, paying much more for no tangible improvement.

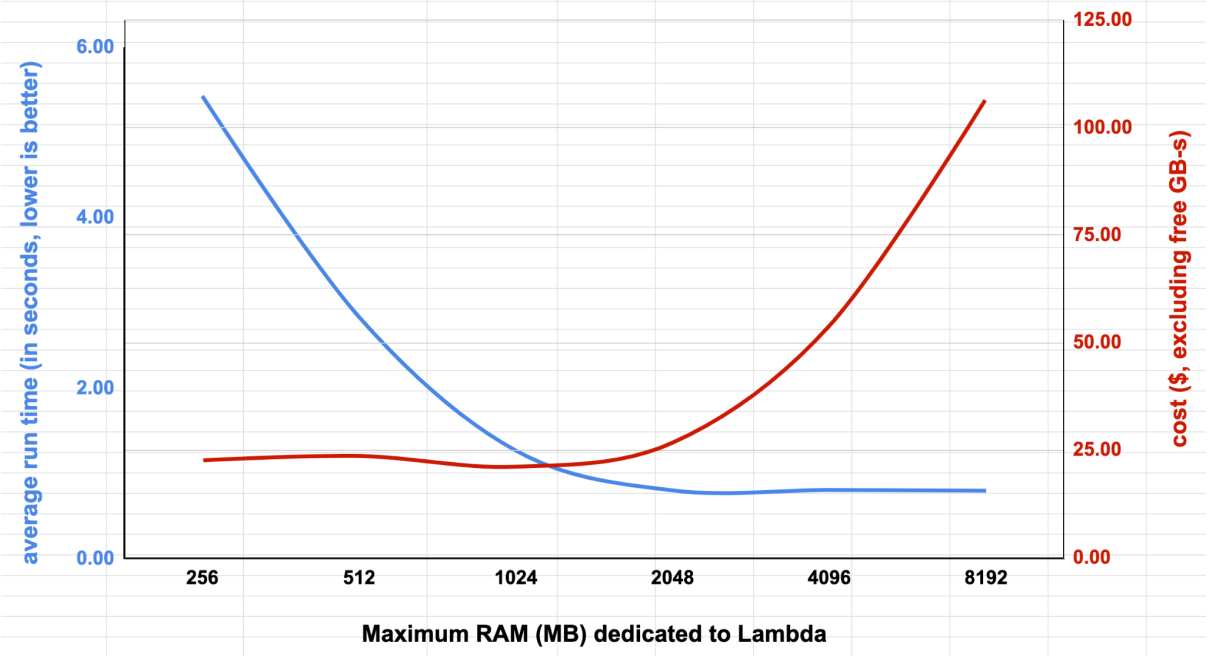


Chart 4: TextSentimentAnalysis, RAM/billed cost correlation

Project distribution

The project is **open source** and publicly available on GitHub[4].
Anyone can fork the project and create their own NLPipe.

Distributed code includes local and AWS versions of NLPipe components, external libraries and tools needed and utility scripts.

Utility scripts help in different ways: input preprocessing, local and AWS benchmark execution, local benchmark statistics, AWS benchmark invocation preparation.

AWS code, together with the provided external libraries and tools, can be used to perform a manual deployment of the infrastructure.

Local code is useful to test tasks logic, behaviour and accuracy.
Additionally, paired with benchmarking and input preprocessing scripts, it can be used to compute baseline performances to match against AWS performances.

We truly hope this will help the international NLP community.

[4] https://github.com/fortym2/NLPipe