Cloud Computing Project Report

Francesco Fazzari
1935070
Sapienza University of Rome
fazzari.1935070@studenti.uniroma1.it

Dario Loi
1940849
Sapienza University of Rome
loi.1940849@studenti.uniroma1.it

CONTENTS

I. INTRODUCTION

Today, with the advancement of large language models that are increasingly able to mimic human emotions to provide more natural feedback to users, an AI model capable of recognizing emotions from a sentence can be used to analyze emotional tone in areas such as customer service, mental health, education, system responsiveness and many other fields.

Our projects deploys a sentiment analysis classifier through AWS lambda, delivering real-time performance on CPU-only infrastructure through the use of techniques such as model compilation and quantization.

We provide an extensive benchmark of both user-oriented and system-oriented metrics. We utilize the locust[3] testing framework to simulate realistic user loads in different configurations that allow us to show the vertical and horizontal scaling capabilities of our cloud-based solution.

II. BACKGROUND

A. AI Model

To perform sentiment analysis on arbitrary input text, we leverage the model zoo provided by Hugging Face transformers[6], specifically, we start with a finetuned version of MobileBERT[5, 4] which classifies sentences in four classes of emotions: `happy`, `angry`, `sad`, and `other`. The "other" labels serves as a default option so that the model is not forced to confidently give an incorrect answer.

The service API can easily be extended to cover the classification of a wider array of human emotions by manually finetuning a more general model, however we consider this to be out of the scope of our project since our main aim is to analyze the performance of the inference system when deployed on cloud infrastructure, not to develop a state-of-the-art sentiment analysis model

B. Quantization

Deploying Large Language Models (LLMs) on the cloud using only the CPU-based infrastructure at our disposal is quite the challenge. The challenges are twofold:

1) The pytorch[1] framework which is used as a runtime for AI models is quite cumbersome as a dependency, and exceeds the size limits offered by AWS lambda and AWS lambda layers
2) The language model's parameters are in the order of hundred of megabytes, this also violates the aforementioned constraint. In addition, this forces us to requisition a larger amount of resources to perform inference, increasing the costs of our infrastructure

We solve this by converting our models to the open ONNX standard, meaning that we only need the much lighter `onnxruntime`[2] library as a dependency. During the conversion process, we perform dynamic quantization of the `float32` model parameters down to `uint8`, we then export the model to the ONNX-compatible ORT format.

This allows us to reduce the model size by a factor of ≈ 3.68, from 81.5 MiB to 22.1 MiB.

C. AWS Lambda and Lambda Layers

AWS Lambda is a serverless computing service offered by Amazon Web Services (AWS) that enables running code without provisioning or managing servers. It automatically scales and executes code in response to events, making it ideal for building event-driven and real-time applications. Lambda supports multiple programming languages and provides pay-as-you-go pricing based on the actual execution time and resource usage.

We choose to employ AWS Lambda as our core infrastructure due to its simplicity and autoscaling capabilities. In order to

deploy our application efficiently, we separated the runtime and model weights from the dependencies, which have been uploaded as a separate Python 3.12 layer.

D. CloudWatch

Amazon CloudWatch is a monitoring and observability service provided by AWS. It enables real-time monitoring of AWS cloud resources and applications, helping developers and system administrators track resource utilization, application performance, and operational health. CloudWatch is set up to automatically log relevant metrics for any AWS Lambda instance, we leverage this data during our experiments to show reports on system-oriented metrics under various load configurations. We also make use of CloudWatch's automatic monitoring capabilities to enhance the reliability of our cloud application, as explained in the next section.

E. Alarms

In order to monitor our cloud infrastructure and be able to respond to criticalities quickly and effectively, we create a set of CloudWatch alarms, as shown in table I, furthermore, we use Amazon SNS (Simple Notification Service) to send an email alert whenever one of the alarms is triggered, so that availability of the service can be restored as soon as possible.

TABLE I
AWS LAMBDA ALARM PREDICATES

Alarm Name	Trigger (5-minute average)
TooManyConcurrentLambda	ConcurrentExecutions $>$ 500
LambdaThrottles	Throttles $>$ 0
LambdaErrors	Errors $>$ 0

III. APPLICATION

Our implementation consists of a serverless application that utilizes AWS Lambda functions to perform inference on input sentences using the MobileBERT[5] model.

The application is a simple REST API that exposes a POST endpoint, an example of a valid request is shown in fig. 1. The REST API answers with the corresponding prediction, together with a probability distribution across all of the labels.

The model, dependency layers and code are stored in an S3 bucket, this is necessary since Lambda deployments without layers and S3 buckets can only consist of less than 50.0 MiB of data. Using this additional storage increases the maximum

```
{
  "sentence": "I am so excited about tomorrow"
}
```

Fig. 1. An example of a JSON request body sent to the `/sentimentAnalysisONNX` endpoint. The payload contains a single key, `sentence`, with the text to be analyzed.

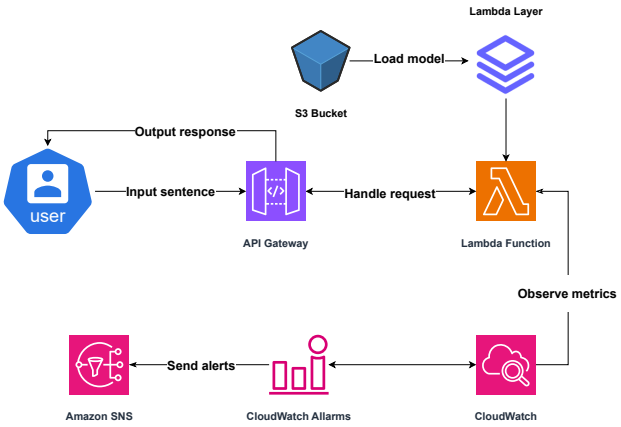


Fig. 2. Architecture diagram.

size to 250.0 MiB and allows us to deploy our inference code to AWS Lambda.

We utilize an API Gateway to expose this service as an HTTP API to the outside world, only allowing POST and OPTIONS requests for CORS compliance.

As explained thoroughly in sections II-D and II-E, we employ CloudWatch for active monitoring of our infrastructure as a whole.

A diagram of our cloud architecture is shown in fig. 2.

The resource configuration of our Lambda is shown in table II, the model runs in under 256 MiB of RAM, but we requisition 512 MiB since that also increases the CPU power at out disposal, decreasing response time.

TABLE II
LAMBDA CONFIGURATION

RAM	Ephemeral Storage	Timeout
512 MiB	512 MiB	30 s

A. Frontend

Together with the API, we also develop a simple frontend that allows a user to interact with the model in a natural way. This also allows for a qualitative evaluation of the API's response time, as one can gauge whether the request is handled in a timely manner in an easier way if the request is issued through a familiar user experience which is comparable to those of other language model services.

The interface is provided as a single static website that leverages tailwind CSS for a simple and clean aesthetic.

IV. PERFORMANCE EVALUATION

A. Locust

Locust is an open-source load testing framework designed for assessing the performance and scalability of web applications by simulating user behavior. It allows developers to define user scenarios and simulate concurrent users to measure how the application performs under load. Locust provides real-time

monitoring of key performance metrics, such as response times, request rates, and error rates, making it a valuable tool for identifying bottlenecks and optimizing application performance. We use Locust to simulate realistic user behavior and measure the performance of our Lambda function under different load conditions. We take great care to configure proper ramp-up periods and rate limiting for each of our simulations, to ensure that our tests are not perceived as Denial of Service attempts by AWS.

B. Test Configurations

As shown in table III, we envision three different test scenarios to showcase the behavior of our application across different possible realistic situation, the *bursty* test models a sparse request pattern, and aims to show the capability of our infrastructure to handle intermittent requests. The *medium* and *heavy* tests test the lambda's capability to handle more significant volumes of traffic, for these workloads we are particularly interested in our infrastructure's horizontal scaling capabilities. Each of these tests samples one sentence from a set of ten sentences, with length varying from 19 characters to 187 characters, this is to try and simulate a realistic distribution of the data. The model can handle phrases as long as 128 tokens, which can be thought of as corresponding to roughly 128 words.

TABLE III
LOAD TESTING CONFIGURATIONS

Scenario	Users	Total Ramp-up	Duration	Request Delay (s)
Bursty	3	3 s	2 min	[5, 10]
Medium	50	25 s	2 min	[3, 5]
Heavy	100	50 s	1 min	[1, 3]

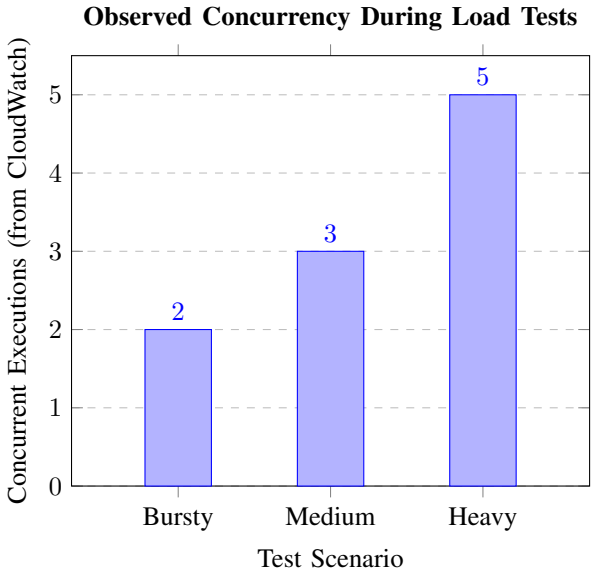
V. RESULTS

For each of the three test configuration, we report the metrics as summarized by locust in figs. 3 to 5. We also report quantitative results in table IV, here our aggregate response time statistics are relative to the the end of our tests, therefore our percentiles have already stabilized with respect to the cold start period, the presence of which is indicated by the high values reported in the Max column.

For all of our scenarios, no failures were reported, and none of the alarms described in table I are ever triggered.

A. Concurrency and Scaling

To better understand the scaling behaviors that occur during ramp up times as reported in figs. 4 and 5, we show the maximum concurrent executions reached during every test as reported by CloudWatch in section V-A. The low numbers show that our system is able to cope with all tests in an excellent manner, probably due to the fact that the requests are slightly interleaved thanks to each user waiting a bit before issuing the next request. Our system can process thousands of requests with less than a dozen concurrent instances, all the while maintaining 100% availability.



B. Vertical Scaling

To further investigate the possibility of scaling our infrastructure vertically, we run the *Heavy* test configuration with double the amount of provisioned RAM increasing it from 512 MiB to 1024 MiB, which should also result in our lambda being assigned more compute resources. This test configuration shows no significant changes in response time metrics. By pinging the API we estimate that the network delay is of around 100 ms, meaning that the actual processing time spent to evaluate each request when the Lambda is warm is of around 20 s. CloudWatch metrics confirms our intuition, reporting execution times in the order of seconds.

This suggests that individual requests do not need more execution resources and are already being handled in a close-to-optimal way, provided that no cold-start occurs and that the automatic load balancing provided by AWS copes with any eventual burst of requests.

We still show the graph relative to this test in fig. 6 as we believe it to show interesting behavior with respect to vertical scaling.

VI. CONCLUSION

We have shown that, despite challenges in terms of model size and computational cost, it is possible to deploy an LLM to effectively perform downstream tasks on arbitrary input text. Our proposed architecture is simple yet scalable, and has proven to be fault-free and scalable to realistic traffic (of non-malicious nature). Through our benchmarks we have identified that the current bottlenecks lay in network overhead which might be reduced by deploying our serverless infrastructure closer to the end-user, something that is not available to us given our current resources (but easy to implement). We have also shown our service's capacity to automatically scale to meet user demands effortlessly, both horizontally and vertically.

Fig. 3. Graphs for the Bursty Load Test, following the cold start period, response times quickly lower and stay steady, the fluctuations in the requests-per-second graph are due to user rate limiting, and not to be attributed to problems in the backend.

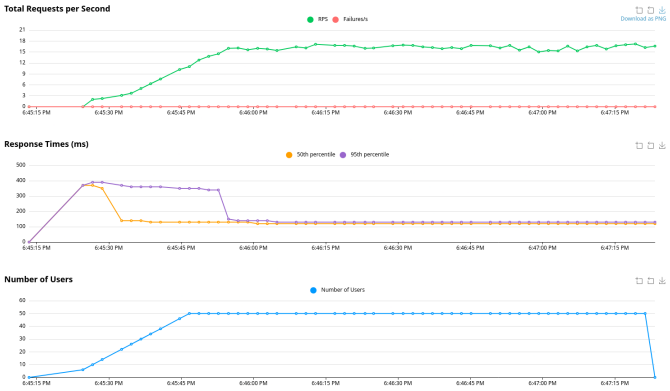


Fig. 4. Graphs for the Medium Load Test. The linear correlation between requests-per-second satisfied (shown at the top) and the amount of active users (shown at the bottom) indicates successful scaling by our Lambda infrastructure. We also see a response time that stabilizes to an acceptable amount of latency, however, the 99-th percentile response time stays high during the warm-up period of our users, this suggests that some of our new users are being assigned to new instances that must have a new cold start period, and that therefore our Lambda infrastructure is actively scaling out.

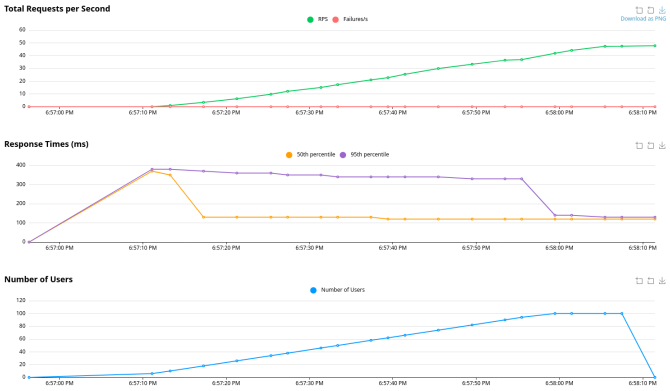


Fig. 5. Graphs for the Heavy Load Test. The data shown further confirms the hypothesis that the ramp-up user period corresponds to an active scale-out behavior of our lambda infrastructure, as most of our time is spent ramping up the amount of users sending requests to the service, we also experience an heightened 99-th percentile response time during this period.

TABLE IV
AGGREGATED PERFORMANCE STATISTICS PER LOAD TEST

Test Scenario	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Median (ms)	95%ile (ms)	99%ile (ms)
Bursty	51	0	144.75	119	362	130	340	360
Medium	1767	0	130.77	112	1520	120	140	350
Heavy	1722	0	136.35	111	389	120	330	360

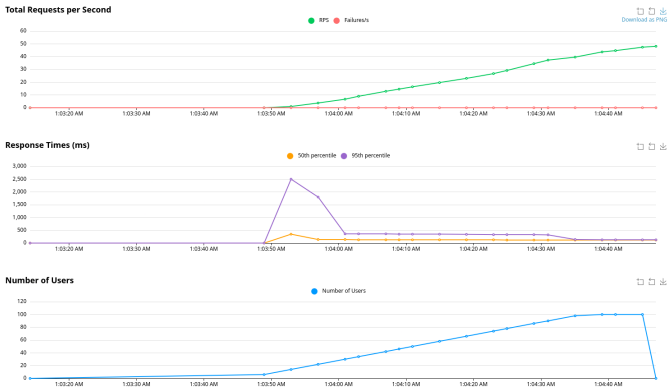


Fig. 6. Graphs for the Heavy Load Test with Lambda RAM scaled to 1024 MiB. this test was executed some time after the others and we have incurred a heavy cold-start penalty, therefore we notice three phases of response time 99-th percentile, the first peak corresponds to the lambda's cold start period, the second plateau occurs during the whole ramp up period, finally, the response time settles down to the minimum when the ramp-up period terminates.

REFERENCES

[1] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://docs.pytorch.org/assets/pytorch2-2.pdf.

[2] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[3] Jonatan Heyman, Lars Holmberg, et al. *Locust - A modern load testing framework*. https://github.com/locustio/locust. Version 2.28.1. 2024. URL: https://github.com/locustio/locust.

[4] lordtt13. *emo-mobilebert*. https://huggingface.co/lordtt13/emo-mobilebert. 2022.

[5] Zhiqing Sun et al. *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices*. 2020. arXiv: 2004.02984 [cs.CL]. URL: https://arxiv.org/abs/2004.02984.

[6] Thomas Wolf et al. "Transformers: State-of-the-Art Natural Language Processing". In: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.