SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF CYBERSECURITY

WattVault

CLOUD COMPUTING

Students:
Gaia Landolfo
1844821
Loris Lindozzi
2056098

Professors:
Casalicchio Emiliano

Academic Year 2023/2024

Contents

1 Introduction

WattVault is a serverless cloud application designed for the calculation and storage of electricity bills. It allows users to send their data to the application to generate a PDF report for monthly expenses, which will then be archived for later retrieval at any time. It has been designed to handle a large number of requests and to have high availability, using a dynamic architecture that automatically scales according to the workload.

2 Architecture

We have used Amazon Web Services as our cloud provider and employed a server-less architecture. The application consists in:

- An API Gateway, which routes all the requests.
- A Lambda backend, with 2 independent microservices.
- An S3 Bucket, as a storage for the PDF files.
- AWS Cloudwatch for collecting statistics and monitoring different metrics.

The schema of the architecture is shown in the figure below:

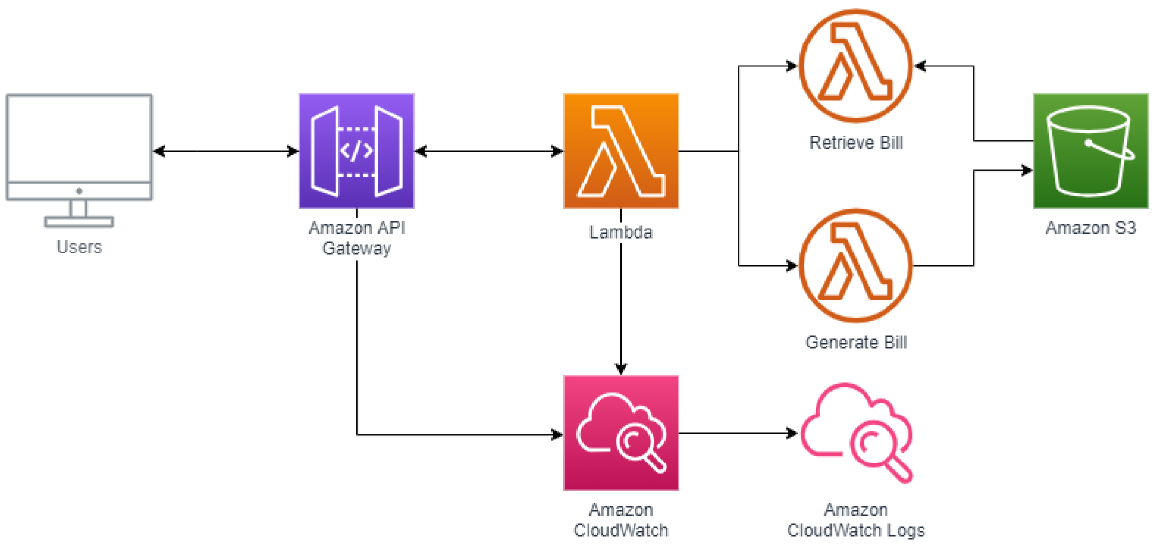


Figure 1

3 Implementation

3.1 Creating the Lambda functions

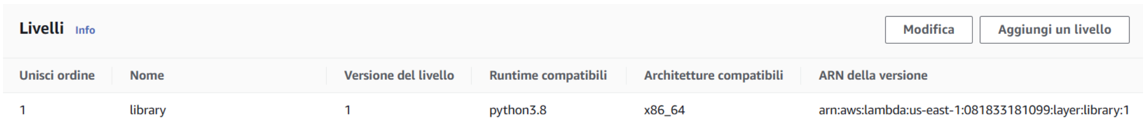
We started by creating the python lambda functions:

- PDFgenerator
- GetPDF

The PDFgenerator Lambda function is designed to process electricity bill data, generate a PDF report, and store it in an Amazon S3 bucket. This function is triggered by an API Gateway POST request and consists of 2 main part:

- generate_pdf(data): The function extracts and parses the JSON payload from the incoming request. The payload must contain user_id, name, address, billing_month, and total_consumption. Then the function uses the extracted data to generate the PDF file.
- lambda_handler(event, context): It recalls the previous function and uploads the resulted PDF to a specified Amazon S3 bucket in a structured path: electricity_bills/{user_id}/{billing_month}.pdf.
  Then, if no error occurred, it responds to the user with an HTTP 200 OK message with the link to retrieve the PDF report.

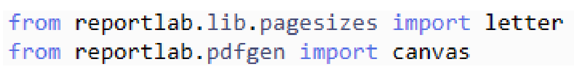
The function uses a custom library in order to generate the PDF correctly, in particular it is used reportlab. We created a new layer for the lambda function where we uploaded the zip file of the library previously downloaded.



Unisci ordine	Nome	Versione del livello	Runtime compatibili	Architetture compatibili	ARN della versione
1	library	1	python3.8	x86_64	arn:aws:lambda:us-east-1:081833181099:layer:library:1

Figure 2

In this way the PDFgenerator function is able to recognise the imported modules:



```python
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
```

Figure 3

The getPDF Lambda function is designed to retrieve a PDF report of an electricity bill from an Amazon S3 bucket based on the user's request. The function is triggered by an API Gateway GET request, which specifies the user ID and billing month in the URL path parameters.

In particular it extracts the {user_id} and {billing_month} from the API Gateway request's path parameters and constructs the S3 key to locate the PDF in the S3 bucket. Then constructs an HTTP response with the PDF content.

3.2 Creating the API gateway

Next, we created an API gateway in order to handle the HTTP request from the users and invoking the corresponding Lambda functions. We created 2 routes for our gateway:

- POST Method "/generate-bill": This method handles requests to generate a PDF report for an electricity bill based on the user's input. It triggers the "generatePDF" Lambda function.
- GET Method "/retrieve-bill/{user_id}/{billing_month}": This method handles requests to retrieve a previously generated PDF report of an electricity bill. It triggers the "getPDF" Lambda function

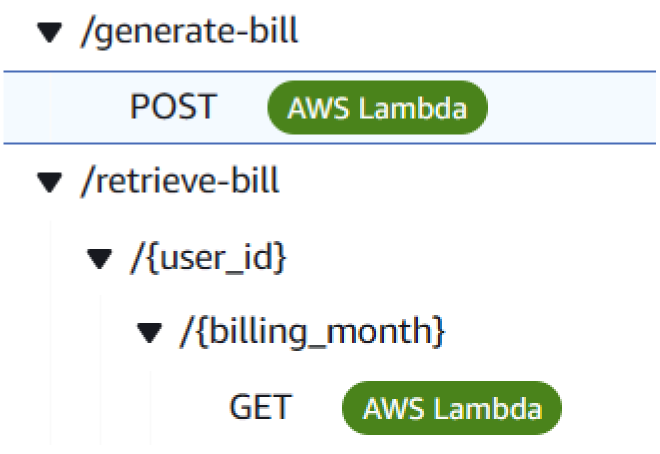


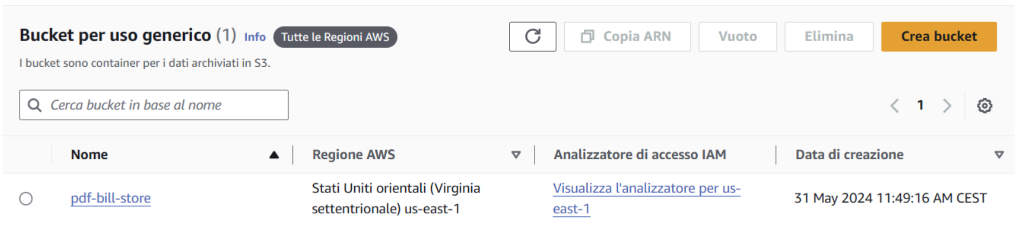
Figure 4

These methods provide a seamless interface for generating and accessing electricity bill PDFs, leveraging the scalability and reliability of AWS services.

3.3 Creating the S3 bucket

Finally, we created an S3 bucket named "pdf-bill-store" that serves as the primary storage location for the PDF files generated by the "generatePDF" Lambda function.

Each PDF file is stored in a structured directory format based on the user ID and billing month, ensuring organized and easily retrievable storage.



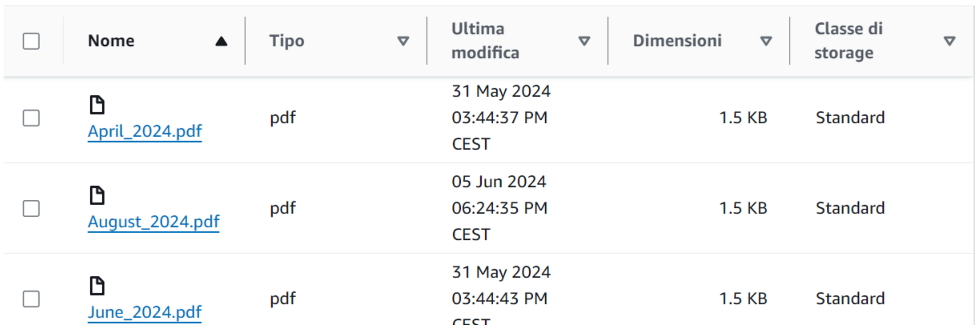
Bucket per uso generico (1) Info | Tutte le Regioni AWS

I bucket sono container per i dati archiviati in S3.

Cerca bucket in base al nome

Nome	Regione AWS	Analizzatore di accesso IAM	Data di creazione
pdf-bill-store	Stati Uniti orientali (Virginia settentrionale) us-east-1	Visualizza l'analizzatore per us-east-1	31 May 2024 11:49:16 AM CEST

Figure 5



Nome	Tipo	Ultima modifica	Dimensioni	Classe di storage
April_2024.pdf	pdf	31 May 2024 03:44:37 PM CEST	1.5 KB	Standard
August_2024.pdf	pdf	05 Jun 2024 06:24:35 PM CEST	1.5 KB	Standard
June_2024.pdf	pdf	31 May 2024 03:44:43 PM	1.5 KB	Standard

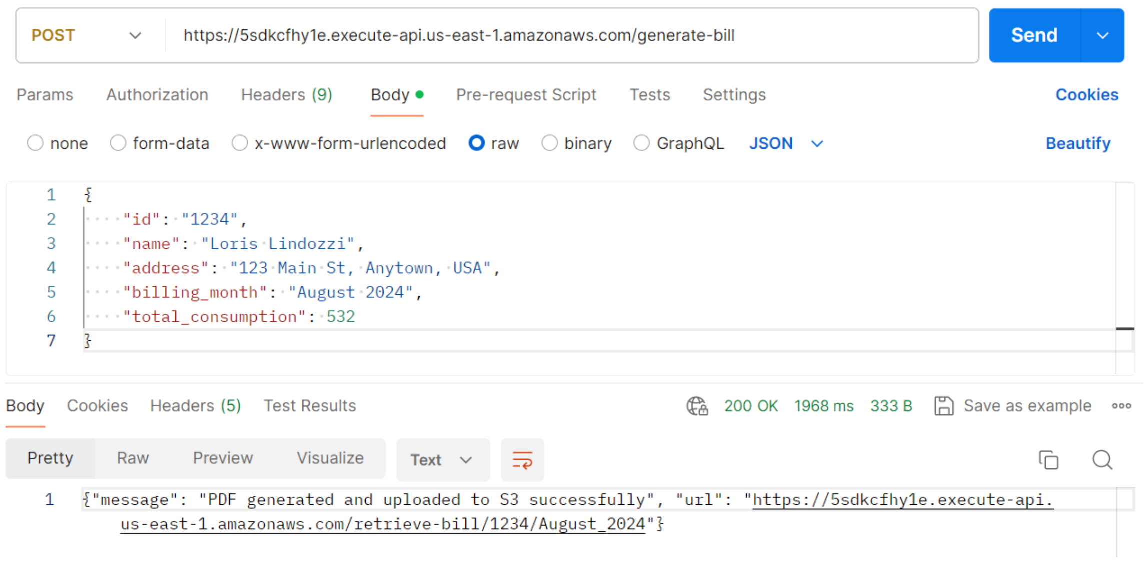
Figure 6

We choose this settings because S3 provides robust security features such as access control policies, encryption, and logging, ensuring that the PDF files are stored securely. The Lambda functions can be configured with appropriate IAM roles and permissions to ensure they have the necessary access to the S3 bucket. Also, S3 is highly scalable, meaning it can handle the storage of a large number of PDF files without any performance degradation. This scalability ensures that as the number of users and PDF files grows, the storage solution remains reliable and efficient.

4 Testing

4.1 Introduction

First of all, we tested the entire infrastructure using the Postman application to verify the correct functioning of both the generation and subsequent retrieval of the PDF file. To do this, we created test calls that respectively invoked the two methods exposed by the API gateway.

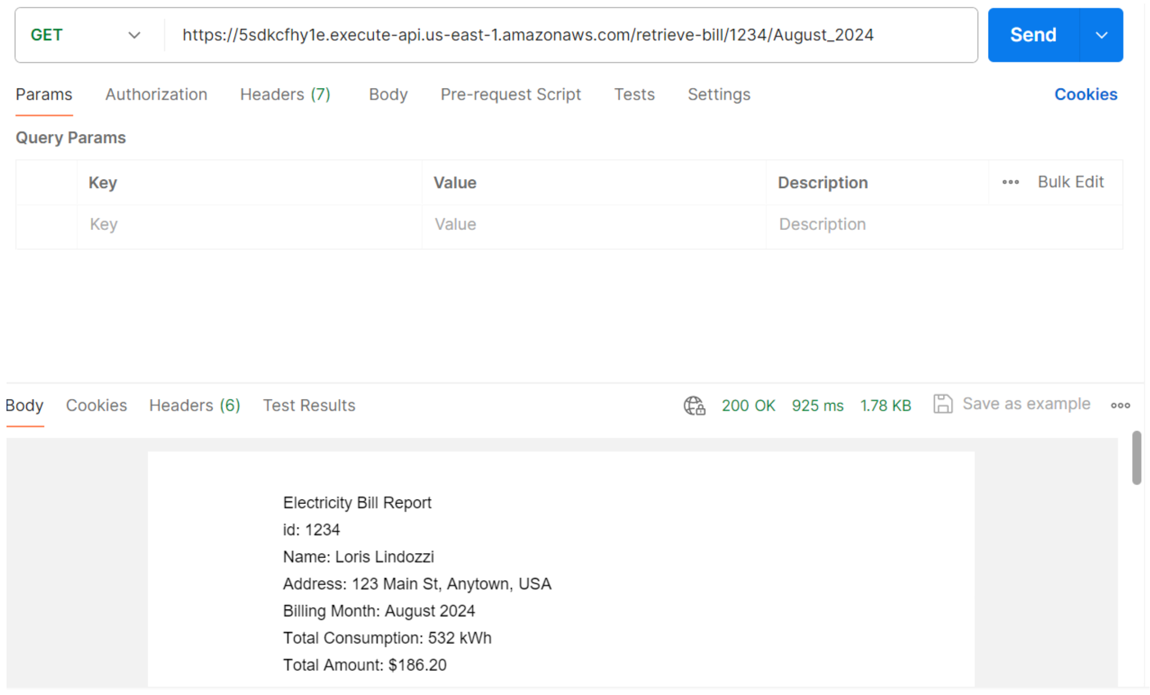


```
https://5sdkcfhy1e.execute-api.us-east-1.amazonaws.com/generate-bill
```

```
{
    "id": "1234",
    "name": "Loris Lindozzi",
    "address": "123 Main St, Anytown, USA",
    "billing_month": "August 2024",
    "total_consumption": 532
}
```

```
{"message": "PDF generated and uploaded to S3 successfully", "url": "https://5sdkcfhy1e.execute-api.
    us-east-1.amazonaws.com/retrieve-bill/1234/August_2024"}
```

Figure 7



```
https://5sdkcfhy1e.execute-api.us-east-1.amazonaws.com/retrieve-bill/1234/August_2024
```

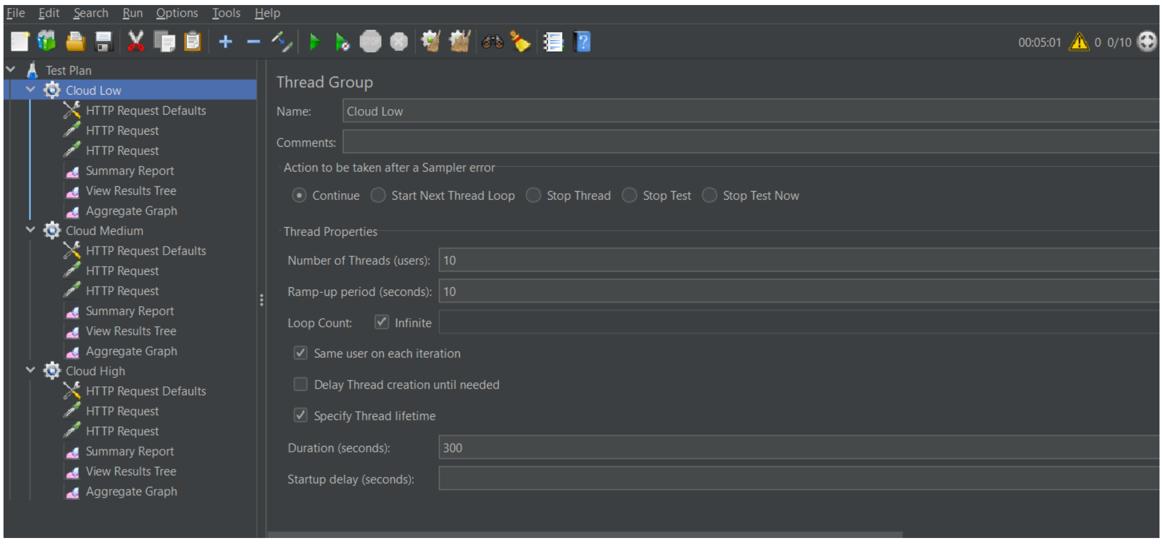
Electricity Bill Report
id: 1234
Name: Loris Lindozzi
Address: 123 Main St, Anytown, USA
Billing Month: August 2024
Total Consumption: 532 kWh
Total Amount: $186.20

Figure 8

4.2 Test setting

After verifying the correct functioning, we focused on testing the performance of the infrastructure by conducting stress tests. For this purpose, we used AWS CloudWatch as a monitoring tool and Apache JMeter as a performance testing toolkit. In JMeter we have created a Test Plan with 3 different phases:

- Low phase: 5 minutes simulating 10 users (threads)
- Medium phase: 10 minutes simulating 50 users (threads)
- High phase: 15 minutes simulating 200 users (threads)



Test Plan
- Cloud Low
  - HTTP Request Defaults
  - HTTP Request
  - HTTP Request
  - Summary Report
  - View Results Tree
  - Aggregate Graph
- Cloud Medium
  - HTTP Request Defaults
  - HTTP Request
  - HTTP Request
  - Summary Report
  - View Results Tree
  - Aggregate Graph
- Cloud High
  - HTTP Request Defaults
  - HTTP Request
  - HTTP Request
  - Summary Report
  - View Results Tree
  - Aggregate Graph

Thread Group

Name: Cloud Low

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 10

Ramp-up period (seconds): 10

Loop Count: [✓ Infinite]

[✓ Same user on each iteration]

Delay Thread creation until needed

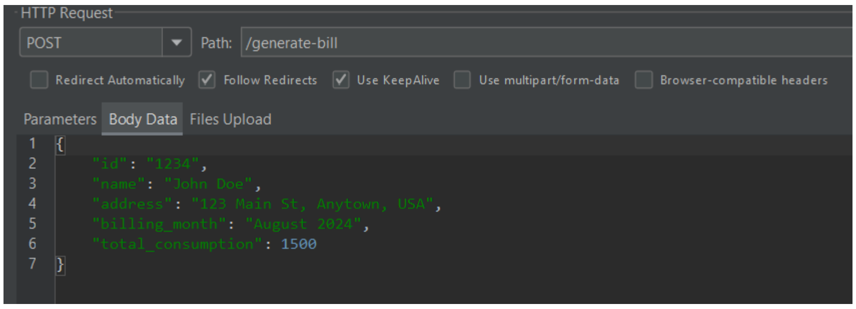
[✓ Specify Thread lifetime]

Duration (seconds): 300

Startup delay (seconds):

Figure 9

In each phase, the test was conducted by calling the API Gateway at the address "https://5sdkcfhy1e.execute-api.us-east-1.amazonaws.com" simultaneously on both of the 2 methods "generate-bill" and "retrieve-bill".

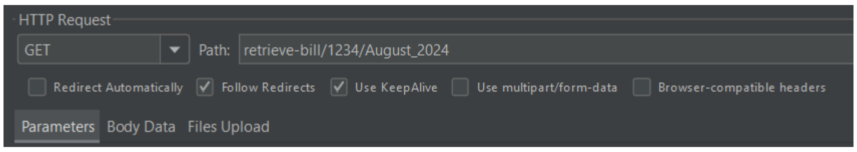

HTTP Request

POST Path: /generate-bill

Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use multipart/form-data Browser-compatible headers

```
{
    "id": "1234",
    "name": "John Doe",
    "address": "123 Main St, Anytown, USA",
    "billing_month": "August 2024",
    "total_consumption": 1500
}
```

Figure 10


HTTP Request

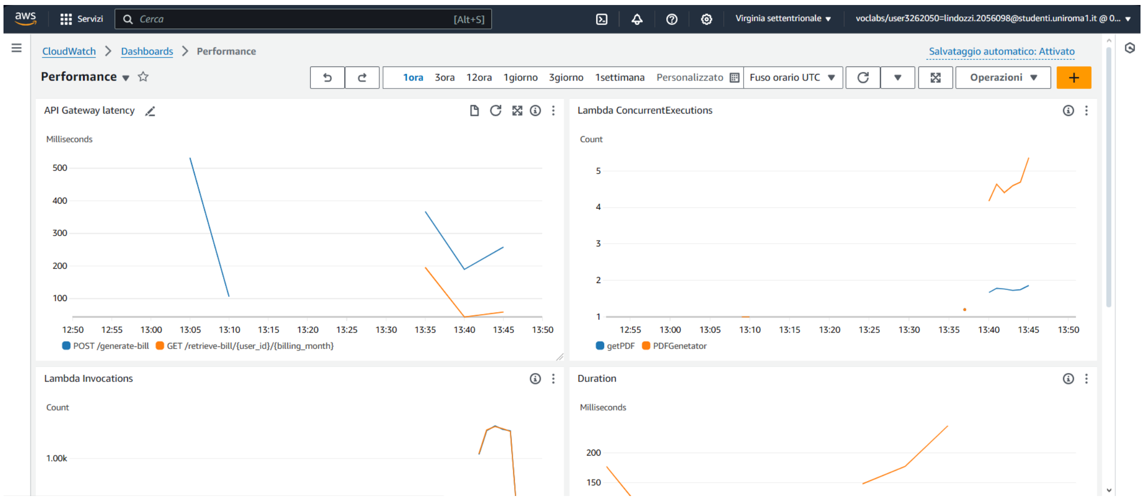
GET Path: retrieve-bill/1234/August_2024

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters Body Data Files Upload

Figure 11

To better track all the results of our tests, we created a dashboard in AWS CloudWatch, setting up graphs to provide real-time feedback with all the parameter of interest like API gateway latency, Lambda concurrent executions, duration, number of invocations and error count.



CloudWatch > Dashboards > Performance
Performance
1ora 3ora 12ora 1giorno 3giorno 1settimana Personalizzato
Fuso orario UTC
Operazioni
Salvataggio automatico: Attivato
API Gateway latency
Milliseconds
POST /generate-bill
GET /retrieve-bill/{user_id}/{billing_month}
Lambda ConcurrentExecutions
Count
getPDF
PDFGenetator
Lambda Invocations
Count
Duration
Milliseconds

Figure 12

4.3 Performance Result

At the end of the test we can analyze the performance and the system behavior. First of all we can see the Lambda invocation:

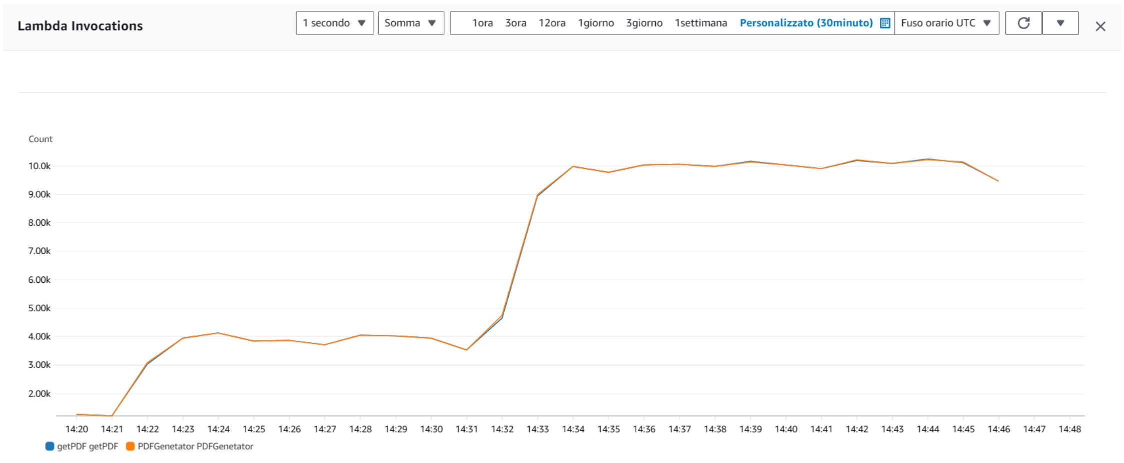


Figure 13

As expected this reflects the behaviour of the 3 phases of the test with a gradual increment of the total invocation. Next we can check the API Gateway performance, in particular the latency metric:

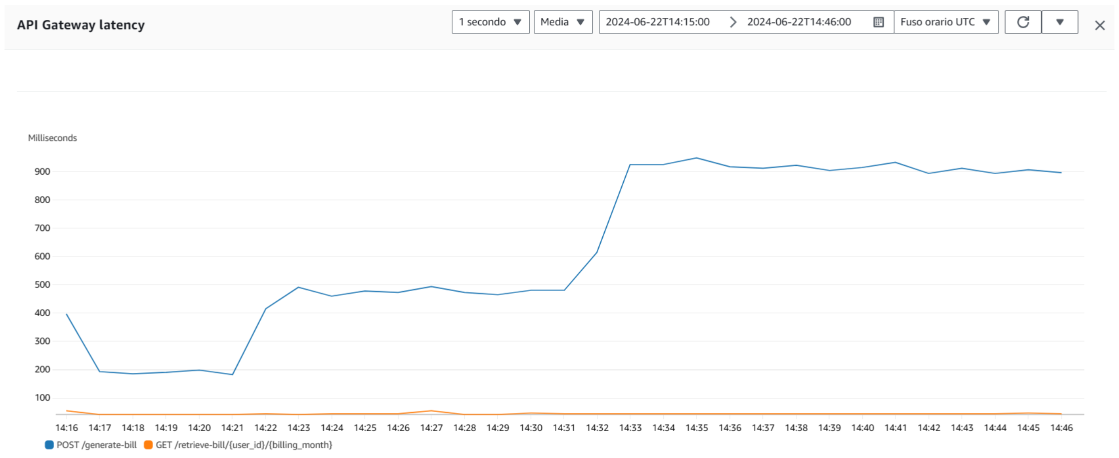


Figure 14

We can notice how the latency for the ”generate-bill” method increases with the numbers of requests, instead the ”retrieve-bill” method remains stable. This highlights how the first function is more computationally expensive than the other.

Than we can observe the lambda concurrent execution in order to test the scalability of our infrastructure:

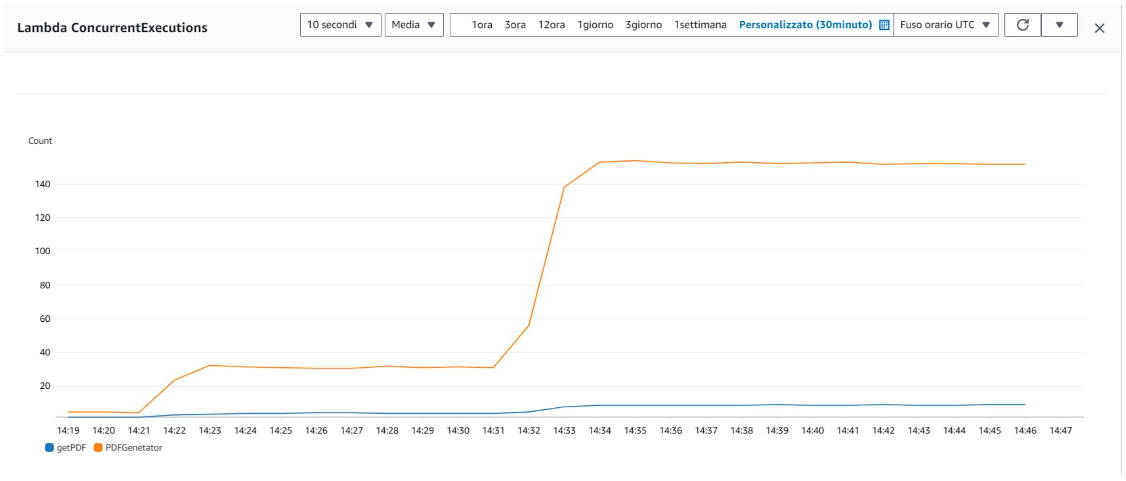


Figure 15

The ”generate-bill” function trend reflects the 3 phases of the test, with a peak of 150 concurrent executions during the third phase. Instead the ”retrieve-bill” has a much lower increment with a pick of 10 concurrent execution.

Also the average duration of each request and the error count can be analyzed:

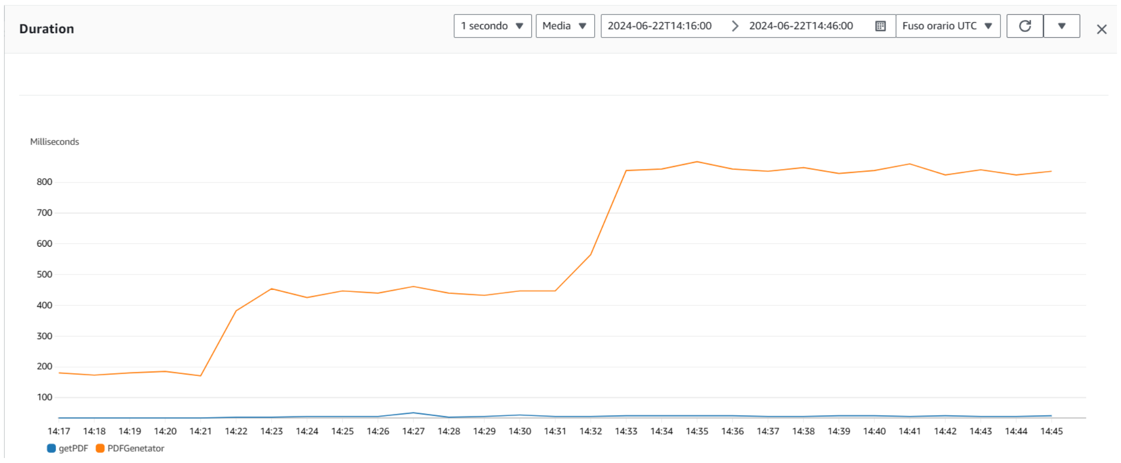


Figure 16

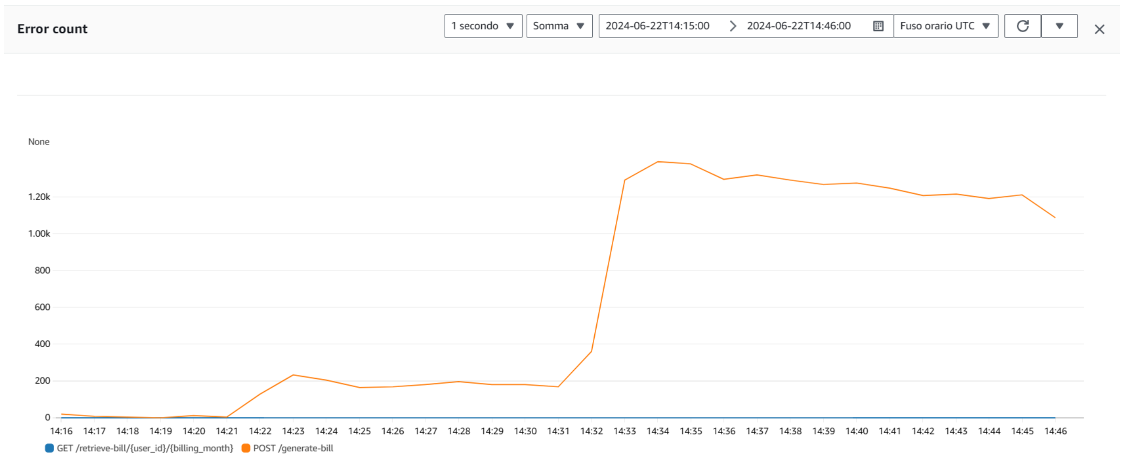
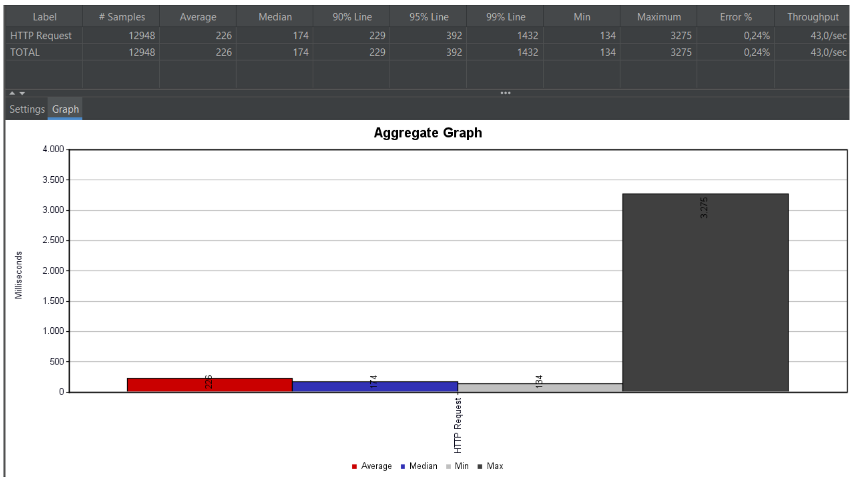


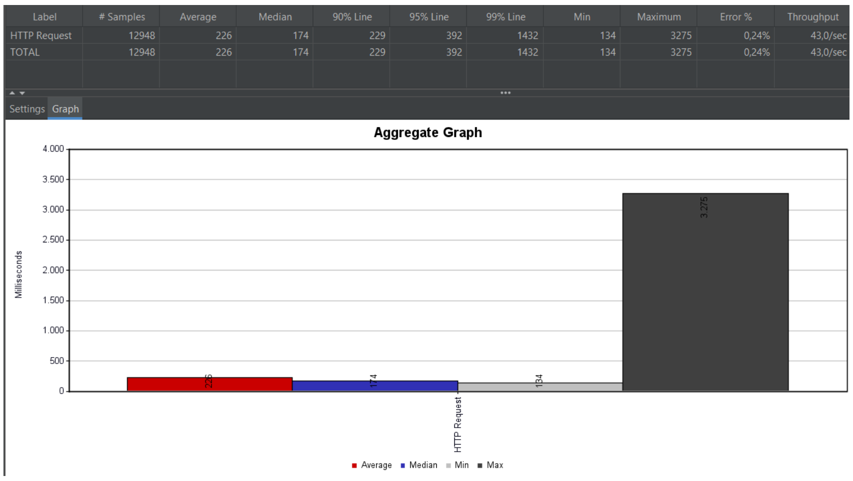
Figure 17

As the other metrics, the "generate-bill" function is the only one that suffered from the stress test.
Finally we report the statistics extracted from the JMeter tool for each test.



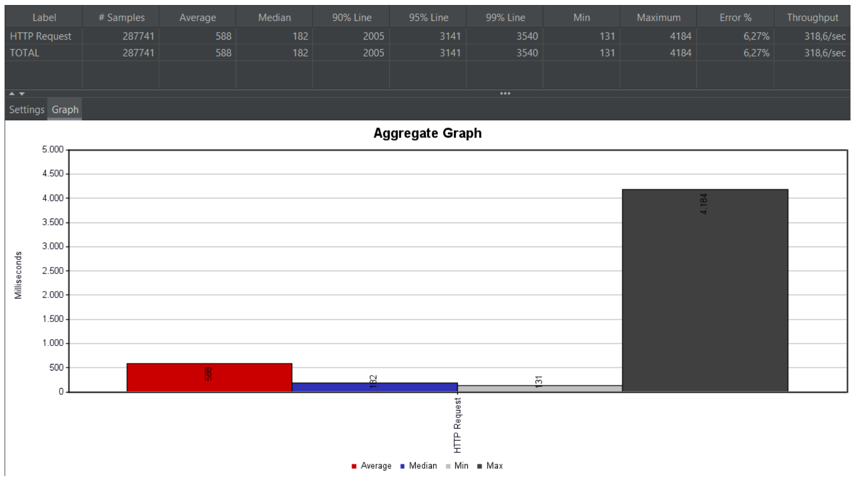
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	12948	226	174	229	392	1432	134	3275	0,24%	43,0/sec
TOTAL	12948	226	174	229	392	1432	134	3275	0,24%	43,0/sec

Figure 18



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	12948	226	174	229	392	1432	134	3275	0,24%	43,0/sec
TOTAL	12948	226	174	229	392	1432	134	3275	0,24%	43,0/sec

Figure 19



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	287741	588	182	2005	3141	3540	131	4184	6,27%	318,6/sec
TOTAL	287741	588	182	2005	3141	3540	131	4184	6,27%	318,6/sec

Figure 20

Some interesting data that we can retrieve is that, although the average request duration ranges from 220 milliseconds to 600 milliseconds depending on the test, the maximum peak is almost stable, ranging from 3300 to 4100 milliseconds.

5 Further Works

This project successfully demonstrates the generation and retrieval of electricity bill PDFs using AWS Lambda, API Gateway, and S3. However, there are several areas where the system can be enhanced and expanded to provide additional features, improve performance, and increase user satisfaction.

- Integrating AWS Cognito for managing user authentication and authorization will ensure that only authenticated users can generate and retrieve their electricity bills. This integration can provide a more secure and user-friendly authentication process.
- The PDF reports can be made more informative by including additional details such as itemized billing information, usage graphs, previous balance, and payment history. Customizing the PDF layout and design to improve readability and presentation will enhance user experience.
- Performance can be optimized by implementing caching mechanisms. Caching frequently accessed data or responses can reduce latency and improve the overall performance of the system. Additionally, optimizing the Lambda function execution and reducing cold start times will enhance the user experience by providing faster response times.

6 Conclusion

This project demonstrates the ease of implementing a serverless architecture in AWS and its ability to efficiently handle highly variable workloads while minimizing costs. By leveraging AWS Lambda, the system efficiently generates PDF reports based on user input, storing them securely in Amazon S3. The integration with API Gateway facilitates easy interaction with the Lambda functions, providing a seamless interface for users to generate and retrieve their electricity bills. Additionally, it is highly convenient not to worry about scaling, which typically demands much more effort with auto-scaling groups or Kubernetes in a traditional server-based architecture.