Serverless Named Entity Recognition on AWS Lambda

Riccardo Pitzanti - 1947877 pitzanti.1947877@studenti.uniroma1.it
Federico Iannini - 1931748 iannini.1931748@studenti.uniroma1.it
Sapienza Università di Roma

August 22, 2025

1 Introduction

The proliferation of unstructured textual data across digital platforms has precipitated a growing demand for efficient and scalable information extraction techniques. Named Entity Recognition (NER), a fundamental subtask within Natural Language Processing (NLP), serves as a critical enabling technology for this purpose, tasked with identifying and classifying atomic elements in text into predefined categories such as persons, organizations, and locations. The operationalization of such NLP models, however, traditionally necessitates managing server infrastructure, which introduces complexity, overhead, and potential underutilization of resources.

This work explores the application of a serverless computing paradigm to address these challenges, deploying a production-ready NLP pipeline for NER on AWS Lambda. By leveraging AWS Lambda and Amazon API Gateway, we architect a microservice that is inherently scalable, cost-efficient, and devoid of server management concerns. The implementation utilizes the spaCy library, a robust industrial-strength framework for NLP in Python, ensuring high-quality inference capabilities. The entire application stack—including compute, API management, and security—is defined and deployed programmatically using the AWS Serverless Application Model (SAM), embodying Infrastructure-as-Code (IaC) principles to guarantee reproducibility and reliability.

The primary objective of this study is to design, implement, and rigorously evaluate the performance characteristics of this serverless NER service under varied load conditions. We employ Locust, an open-source load-testing tool, to simulate traffic patterns ranging from benign to stressful, measuring key client-side and provider-side metrics. Our analysis focuses on quantifying latency, throughput, stability, and the impact of vertical scaling (memory allocation) within the constraints of an educational AWS environment. The findings demonstrate the viability of serverless architectures for deploying lightweight, performant, and cost-effective NLP microservices.

2 Background

2.1 Named Entity Recognition (NER) and spaCy

Named Entity Recognition (NER) is a natural language processing (NLP) task focused on identifying and categorizing information elements, known as entities, within unstructured text. These entities are typically classified into predefined categories such as persons (PERSON), organizations (ORG), and geopolitical entities (GPE). The spaCy library in Python provides a production-ready framework for implementing NLP pipelines, offering statistical models for tasks like NER.

2.2 AWS SAM & Containerized Build with Docker

The AWS Serverless Application Model (SAM) is an open-source framework that extends AWS CloudFormation to provide a simplified syntax for defining serverless resources. It is a form of Infrastructure as Code (IaC) specifically designed to express the functions, APIs, permissions, and events that compose a serverless application. A containerized build process involves using a consistent, isolated environment to compile application dependencies and package the code. By employing Docker, SAM ensures that the build process is executed within a containerized environment that replicates the AWS Lambda runtime. This guarantees consistency between the development build and the deployment target, thereby mitigating compatibility issues.

2.3 AWS Lambda and HTTP API

AWS Lambda is a serverless, event-driven compute service that allows for the execution of code in response to triggers without requiring the management of underlying servers. It automatically scales with incoming request volume and utilizes a fine-grained cost model based on actual compute consumption. The Amazon API Gateway is a fully managed service that simplifies the creation, publication, and maintenance of secure APIs at any scale. It acts as a front-door for applications to access data and business logic from backend services, providing essential features like HTTP endpoint management, request routing, and authorization.

2.4 Observability with CloudWatch

Observability is a system property that describes how well internal states can be understood from external outputs, primarily through the collection and analysis of logs, metrics, and traces. Amazon CloudWatch is a monitoring and observability service that provides a unified view of AWS resource health, application performance, and operational trends. It automatically collects performance metrics from services like AWS Lambda.

2.5 Load Testing with Locust

Locust is an open-source load testing tool that allows developers to define user behavior with Python code and simulate various concurrent users to assess a system's performance under stress. Its distributed and scalable nature makes it suitable for testing the limits of web services and APIs. It provides a real-time web-based user interface to visualize performance indicators such as requests per second, response times, and the number of failing requests as the test is running.

3 System Design and Implementation

3.1 Architecture overview

- **Client** sends `POST /ner` with a short text body.
- **API Gateway (HTTP API)** forwards requests to Lambda.
- **Lambda** parses input, runs spaCy NER (warm model), and returns spans.
- **CloudWatch** records metrics and logs for observability.

API contract.

```
POST /ner
Content-Type: application/json
{"text": "Alan Turing was born on June 23, 1912, in London, England."}
```

```
200 OK
{"entities": [
    {'text': 'Alan Turing', 'label': 'PERSON', 'start': 0, 'end': 11},
    {'text': 'June 23, 1912', 'label': 'DATE', 'start': 24, 'end': 37},
    {'text': 'London', 'label': 'GPE', 'start': 41, 'end': 47},
    {'text': 'England', 'label': 'GPE', 'start': 49, 'end': 56}]
}
```

4 Methodology and Implementation

4.1 Inference Core Module (`src/ner.py`)

This module constitutes the computational core responsible for the Named Entity Recognition (NER) task. It utilizes the spaCy library, a framework for natural language processing (NLP) in Python. The pre-trained statistical model (`en_core_web_sm`) is loaded into a global constant at module import time. The primary function, `extract_entities(text: str)`, processes an input string through the spaCy pipeline. It returns a list of dictionaries, each containing the extracted entity's surface form (`text`), its ontological class (`label`), and the character-level indices (`start`, `end`) denoting its span within the original text.

4.2 Lambda Handler Function (`src/handler.py`)

This module implements the AWS Lambda function handler, which serves as the entry point for requests proxied by Amazon API Gateway. Its primary role is to manage the HTTP request-response cycle. The handler first invokes a helper function, `_parse_body`, to normalize the incoming event structure. This function abstracts away the differences between the event payload delivered by API Gateway (where the HTTP request body is passed as a JSON-encoded string) and the event object used during local testing (which may be a Python dictionary). The handler then performs input validation, checking for the presence and type of the required `'text'` field. Invalid requests result in a `400 Bad Request` response. Validated text is passed to the inference core, and the resulting entities are serialized into a JSON object returned within a `200 OK` response, complete with appropriate HTTP headers for content type.

4.3 Build Process

The build is executed using the AWS SAM CLI command `sam build -use-container`. This process constructs the deployment package inside a Docker container that emulates the Amazon Linux environment of AWS Lambda.

4.4 Infrastructure Provisioning (SAM Template)

The cloud infrastructure is defined declaratively using the AWS Serverless Application Model (SAM), an extension of AWS CloudFormation. The template, `template.yaml`, specifies a minimal and functional stack:

- A single AWS Lambda function resource with its runtime (`python3.9`), allocated memory (512 MB), and timeout (15 seconds) defined in the `Globals` section.
- An Amazon API Gateway HTTP API resource, which provisions a managed HTTPS endpoint. This API is configured with a single route (`POST /ner`) that integrates directly with the Lambda function.

- A pre-existing IAM role (`LabRole`) is referenced for execution permissions, a constraint of the AWS Academy Learner Lab environment. In a standard AWS account, SAM would typically generate a minimal role with necessary permissions automatically.

This Infrastructure-as-Code (IaC) approach guarantees that the entire application stack is versioned, reproducible, and deployable with a single command.

4.5 Front-End

The front-end component of the application is deliberately designed as a minimal web interface to facilitate testing and demonstration of the deployed Named Entity Recognition (NER) service. It is implemented as a static `HTML` page enriched with basic `JavaScript` logic. The interface provides a text area where the user can enter arbitrary input and a form field for specifying the API Gateway endpoint generated during the deployment process. Upon submission, the client-side script performs an asynchronous `HTTP POST` request to the NER API, transmitting the input text in JSON format.

The server's response, which consists of the extracted named entities together with their associated labels, is rendered directly in the browser in a structured JSON format. This design ensures transparency of the system's behavior and allows the user to validate the correctness of the back-end inference in real time.

For ease of distribution and reproducibility, the static front-end is packaged within a lightweight Docker container based on the `nginx` web server. This container exposes the page on a local port, enabling researchers and practitioners to access the interface through a standard web browser without the need for additional configuration or build tools. Such a containerized approach guarantees consistency across different environments, while maintaining the simplicity appropriate for an internal testing utility rather than a production-grade user interface.

5 Performance Evaluation and Testing

5.1 Goals and approach

The objective is to evaluate the responsiveness, scalability, and stability of the NER microservice deployed on AWS Lambda behind an HTTP API. Each experiment uses a structured workload with *Warm-up (WU)*, *Ramp-up (RU)*, *Steady (S)*, and *Ramp-down (RD)* phases. We collect both *user-oriented* (client) and *system-oriented* (provider) metrics and align the CloudWatch time windows with the Locust test window (WU excluded).

5.2 System under test (SUT) and constraints !TODO

- **Function:** AWS Lambda (Python 3.10), NER inference (ONNXRuntime), Region `us-east-1`.
- **Memory:** Baseline 512 MB; Heavy variants at 1024 MB. We also re-ran Scenario B Heavy at 512 MB to assess vertical scaling (512 $\rightarrow$ 1024 MB).
- **API:** AWS HTTP API (API Gateway) fronting the Lambda function.
- **Client:** Locust on a developer workstation; single generator node.
- **Lab limits:** Learner Lab Lambda concurrency $\approx$ 10; budget is pay-per-request, so short, repeated runs are preferred.

5.3 Common settings across all scenarios !TODO

- **Warm-up (WU):** 60 s at 2 users (spawn rate 2/s) to trigger cold starts. Immediately after WU the client statistics are reset so that RU/S/RD are isolated in the reported numbers.
- **Payload mix:** Short and long English sentences with named entities (persons, organizations, locations, dates). Requests are `POST /ner` with a small JSON body.
- **Light vs Heavy:** Each scenario has a *Light* profile (lower RPS) and a *Heavy* profile (higher RPS). Heavy runs use near-zero client think time to approach the Lambda concurrency ceiling; Light runs use more realistic user pacing.
- **Timing windows:** For each run we record absolute UTC start/end times; CloudWatch queries use the same window (WU excluded) to export provider metrics.

5.4 Workload scenarios !TODO motivate

We use three canonical shapes; durations below refer to the *main window* (after WU and stats reset). "Users" are active virtual users; RPS emerges from users, client think time, and function latency.

Scenario A — Bursty

- **Light A:** constant 5 users for 300 s; long think time (5–10 s) to emulate intermittent clicks $\Rightarrow$ low, flat RPS plateau.
- **Heavy A:** constant 10 users for $\approx$360 s; near-zero think time (0–0.1 s) to drive high RPS until bounded by Lambda concurrency.

Scenario B — Ramp $\rightarrow$ Steady $\rightarrow$ Ramp-down !TODO CHANGE

- **Light B:** RU in 60 s steps through 1$\rightarrow$3$\rightarrow$5$\rightarrow$6$\rightarrow$7$\rightarrow$8 users, then S at 8 users for 240 s, then RD 4 users (60 s) and 1 user (60 s). Think time short (0.2–1.0 s).
- **Heavy B:** Faster RU to 10 users, S at 10 users for $\approx$240 s, then RD; near-zero think time (0–0.1 s) for high RPS.
- **Vertical scaling:** We executed Heavy B twice with identical shape, once at 512 MB and once at 1024 MB, to quantify the impact of memory on latency (p95/p99) and steady-state throughput.

Scenario C — Spike

- **Light C:** pre-spike 30 s at 1 user, spike to 8 users for 240 s, then 60 s at 2 users; short think time (0.2–1.0 s).
- **Heavy C:** pre-spike 30 s at 2 users, spike to 10 users for 240 s, then 60 s at 2 users; near-zero think time (0–0.1 s).

5.5 Metrics collected

Client (user-oriented, Locust):

- Per-run CSV/HTML with request counts, failure rate, response-time distribution (p50/p95/p99), and effective RPS during RU/S/RD (WU excluded).

Provider (system-oriented, CloudWatch / Lambda):

- *Duration* (we use p95), *Invocations*, *ConcurrentExecutions*.
- Metrics are exported at 60 s resolution using the exact Locust main-window timestamps.

5.6 Test variants covered in this study !TODO table

- **A, B, C (Light):** realistic pacing, lower RPS.
- **A, B, C (Heavy @** 1024 MB**):** near-zero think time to stress concurrency and throughput.
- **Vertical scaling (B Heavy,** 512 MB **vs** 1024 MB**):** identical shape and pacing, only the Lambda memory differs.

6 Results and discussion !TODO enlarge images

In this section we report the outcomes of the experiments defined in the previous section. For each scenario (A, B, C) we present both the *Light* (lower RPS) and *Heavy* (higher RPS) variants.

6.1 Scenario A (Bursty)

Light A (intermittent pacing). With long think times (5–10 s) and a constant small user pool, the overall throughput remains intentionally low and flat. This profile emulates intermittent human activity rather than throughput saturation. Lambda metrics show a correspondingly low and steady invocation rate and a shallow concurrent-executions curve.

Heavy A (high-RPS bursty). Switching to near-zero client think time with 10 users converts A into a sustained high-throughput plateau. In Locust, the RPS rises sharply after WU. On the provider side, *ConcurrentExecutions* plateaus near the Learner Lab ceiling (about 9–10), which is expected. *Invocations* grow linearly over time during the steady phase, and *Duration* maintains a tight band once the runtime is warm.

6.2 Scenario B (Ramp → Steady → Ramp-down)

Light B (baseline shape). The Locust charts show the stair-step RU, a clear steady plateau, and a controlled RD. Latency percentiles track this shape with mild variations, as expected under moderate load. Lambda *Invocations* and *ConcurrentExecutions* rise predictably during RU and flatten during the steady window.

Heavy B @ 1024 MB **(high-RPS ramp/steady).** With near-zero think time and 10 users, Heavy B reaches a higher RPS plateau than Light B while preserving the RU/S/RD structure. Locust statistics show tightened latency percentiles during the steady phase, indicating that the function remains comfortably within its latency budget under sustained load. Lambda *ConcurrentExecutions* plateaus close to the concurrency limit; *Duration* stabilizes to a narrow band after RU; *Invocations* show the expected linear accumulation over the steady window.

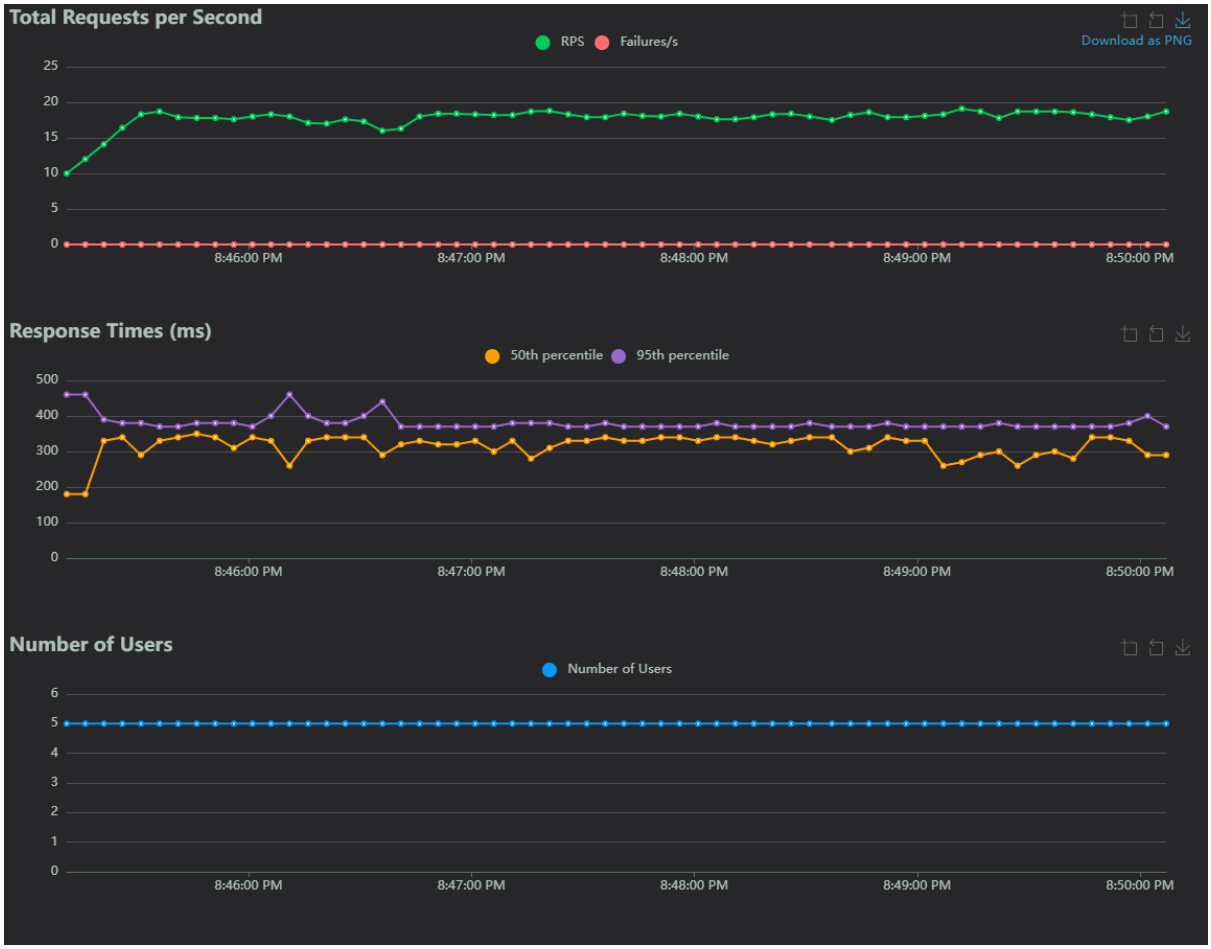

Figure 1: Light A — Locust charts (overall latency/RPS).

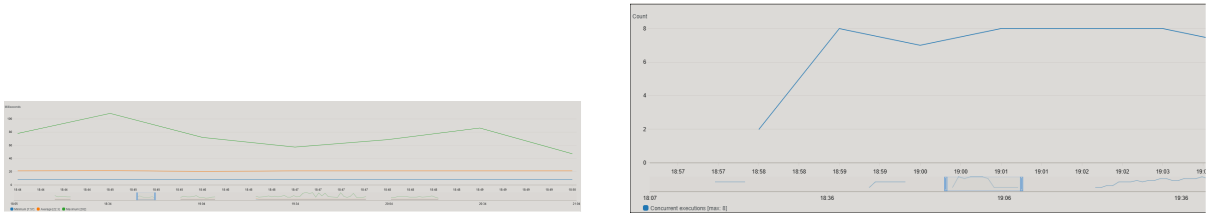

Figure 2: Light A — Lambda Duration (left) and ConcurrentExecutions (right).

Vertical scaling: Heavy B @ 512 MB vs @ 1024 MB. We re-executed the identical Heavy B shape at 512 MB (`hBl-*`) to isolate the impact of memory. The comparison is clearest during the steady window:

- **Latency effect:** The 1024 MB configuration exhibits lower and tighter p95/p99 latency bands than 512 MB, consistent with the expectation that more memory (and CPU share) shortens inference time. This is visible in the Lambda *Duration* time series and in the Locust percentiles.
- **Throughput effect:** Because the account-level concurrency limit remains the same, higher memory primarily improves *per-invocation* speed rather than concurrency. As a result, the steady-state RPS at 1024 MB is higher than at 512 MB for the same user pacing, and the system reaches the plateau faster during RU.
- **Stability:** Both memory settings remain stable during the steady window; no persistent

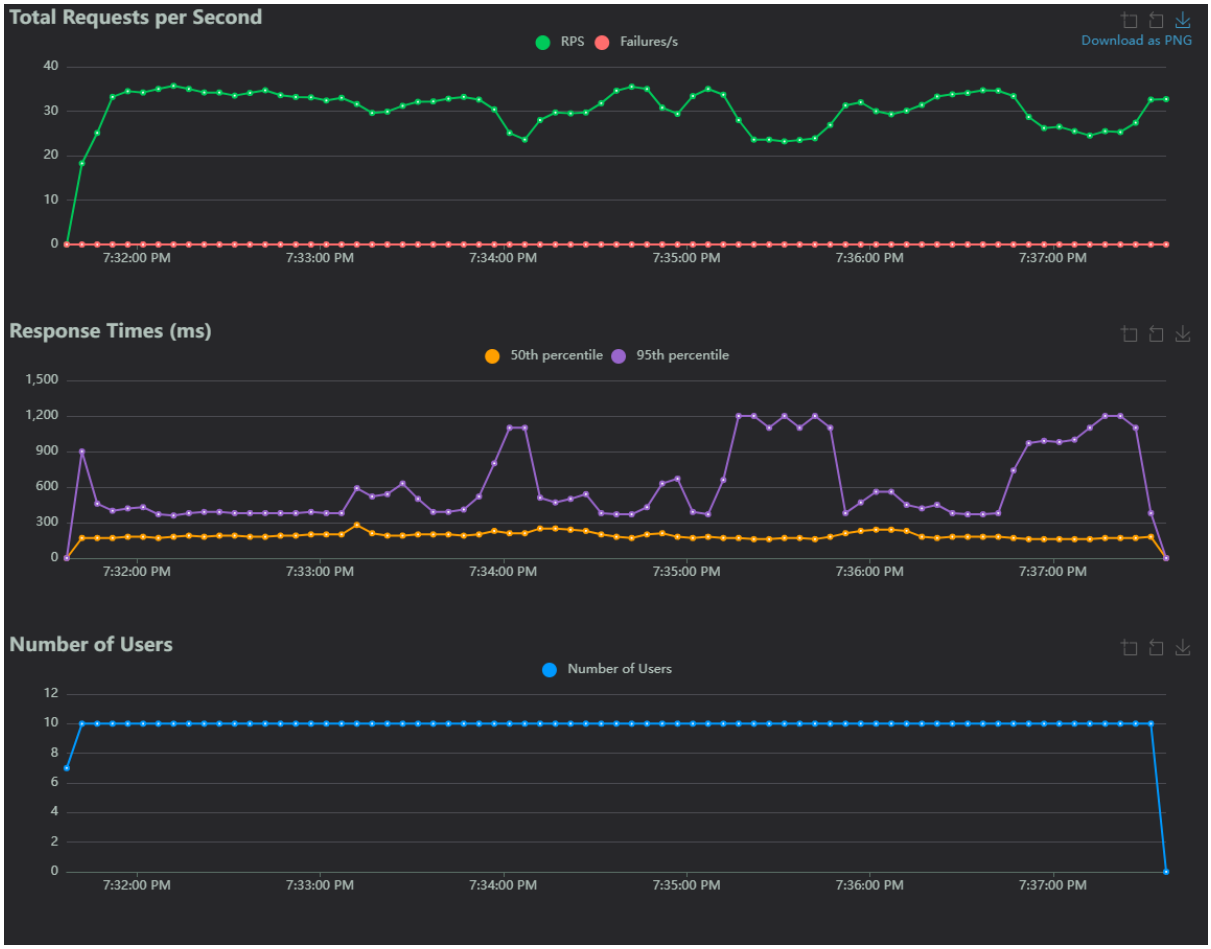

Figure 3: Heavy A — Locust charts (RPS spike after WU; steady plateau).

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	11080	0	268.96	131	4549	284.63	30.72	0
	Aggregated	11080	0	268.96	131	4549	284.63	30.72	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/ner	180	230	310	350	400	580	1100	4500
	Aggregated	180	230	310	350	400	580	1100	4500

Failures Statistics

# Failures	Method	Name	Message

Figure 4: Heavy A — Locust request statistics (p50/p95/p99, failures).

spikes or oscillations are evident. If any brief spikes are present, they align with RU steps and do not persist in S.

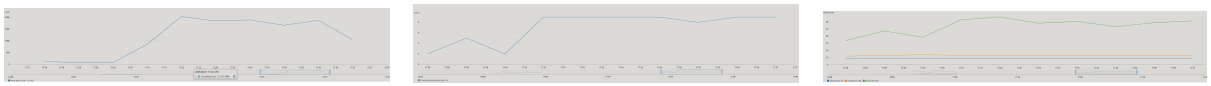

Figure 5: Heavy A — Lambda Invocations (left), ConcurrentExecutions (middle), Duration (right).

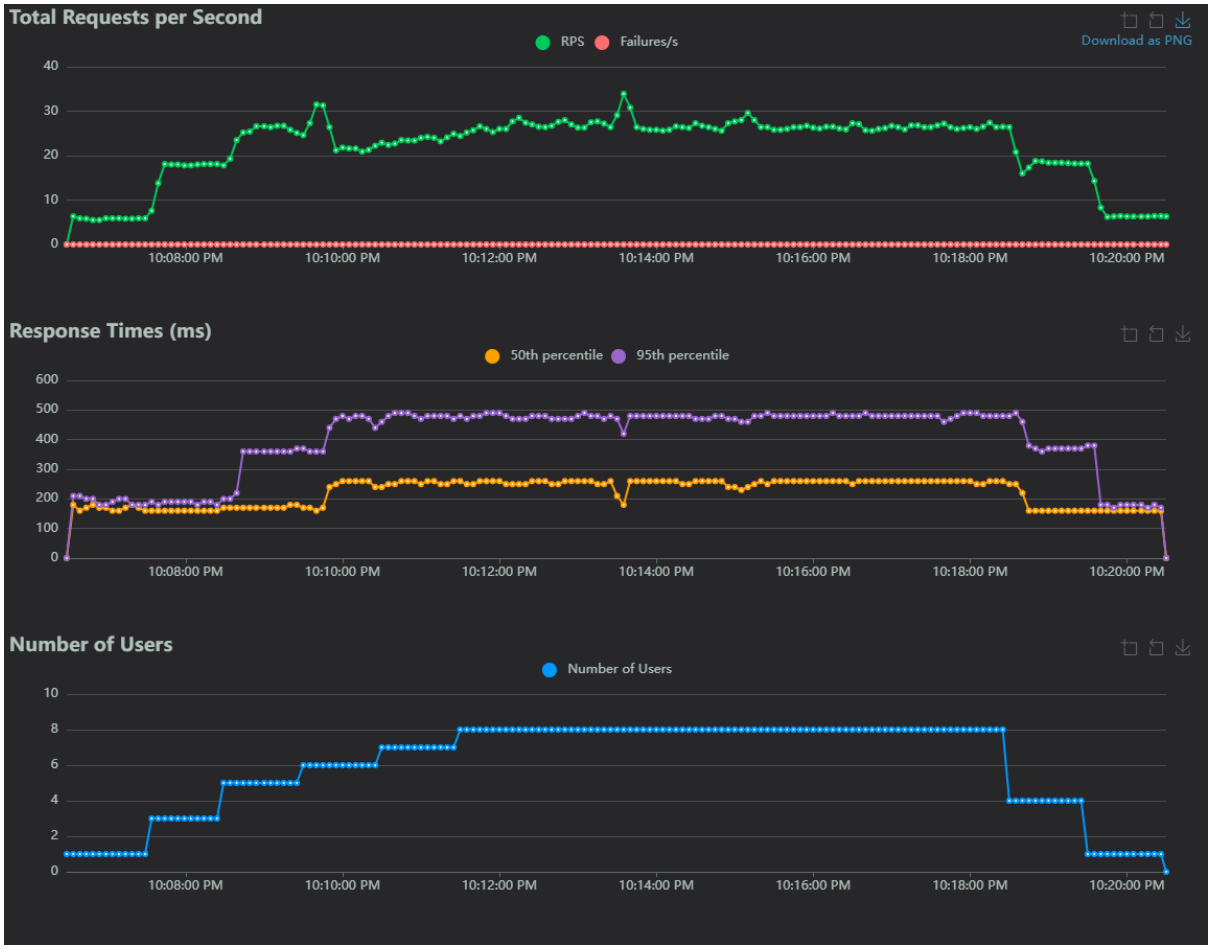

Figure 6: Light B — Locust charts (ramp, steady plateau, ramp-down).

Figure 7: Light B — Lambda Invocations, ConcurrentExecutions, Duration.

6.3 Scenario C (Spike)

Light C (moderate spike). After a short pre-spike, the user count jumps to the steady segment and then drops. Locust shows the expected RPS surge and recovery, with latency percentiles temporarily stretching at the spike boundary and then stabilizing. Lambda *ConcurrentExecutions* mirrors the spike profile; *Invocations* reflect the burst in requests; *Duration* remains in a narrow band during the steady segment.

Heavy C (high-intensity spike @ 1024 MB). With near-zero think time and 10 users, the spike immediately drives the system to the concurrency limit. Locust shows a sharp RPS jump with stable percentiles in the middle of the spike; Lambda confirms the concurrency plateau and a consistent *Duration* band. The swift stabilization after the spike indicates that

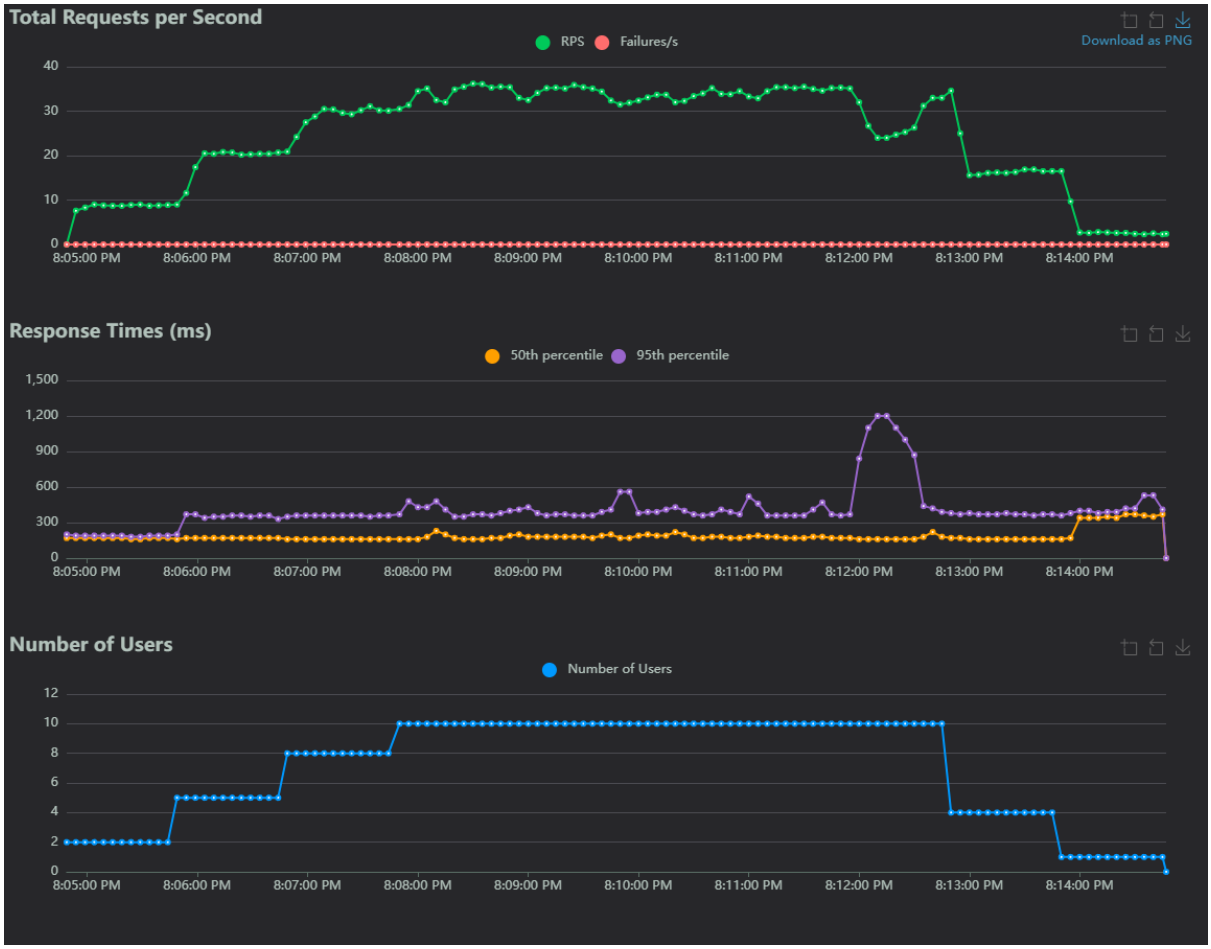

Figure 8: Heavy B (1024 MB) — Locust charts (higher RPS plateau vs Light).

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	14637	0	231.01	134	4681	285.66	24.39	0
	Aggregated	14637	0	231.01	134	4681	285.66	24.39	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/ner	170	180	270	320	360	390	840	4700
	Aggregated	170	180	270	320	360	390	840	4700

Failures Statistics

# Failures	Method	Name	Message

Figure 9: Heavy B (1024 MB) — Locust request statistics.

the cold-start penalty has been amortized during WU and the system sustains the intensity without degradation.

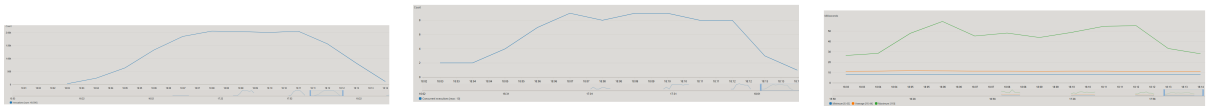

Figure 10: Heavy B (1024 MB) — Lambda Invocations, ConcurrentExecutions, Duration.

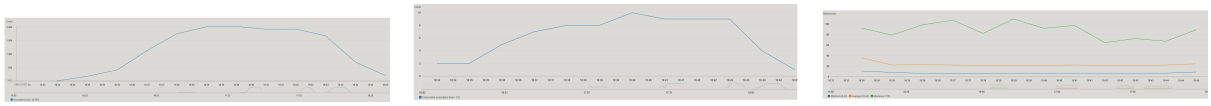

Figure 11: Heavy B (512 MB) — Lambda Invocations, ConcurrentExecutions, Duration.

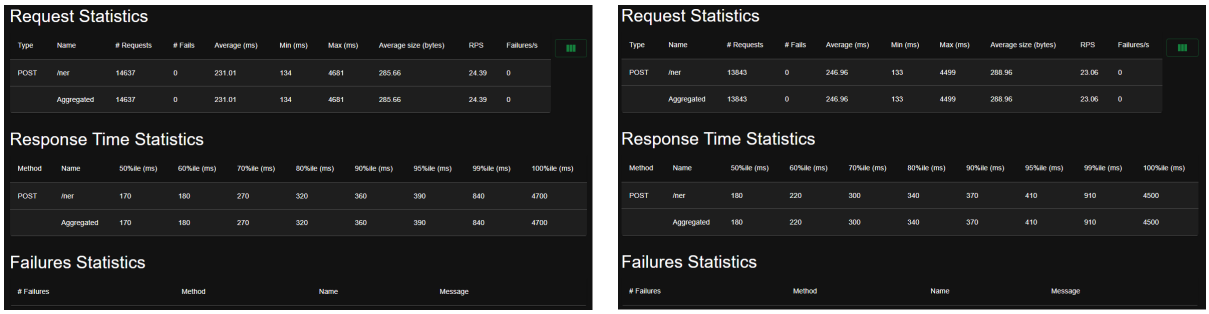

Figure 12: Heavy B — Locust request statistics: 1024 MB (left) vs 512 MB (right).

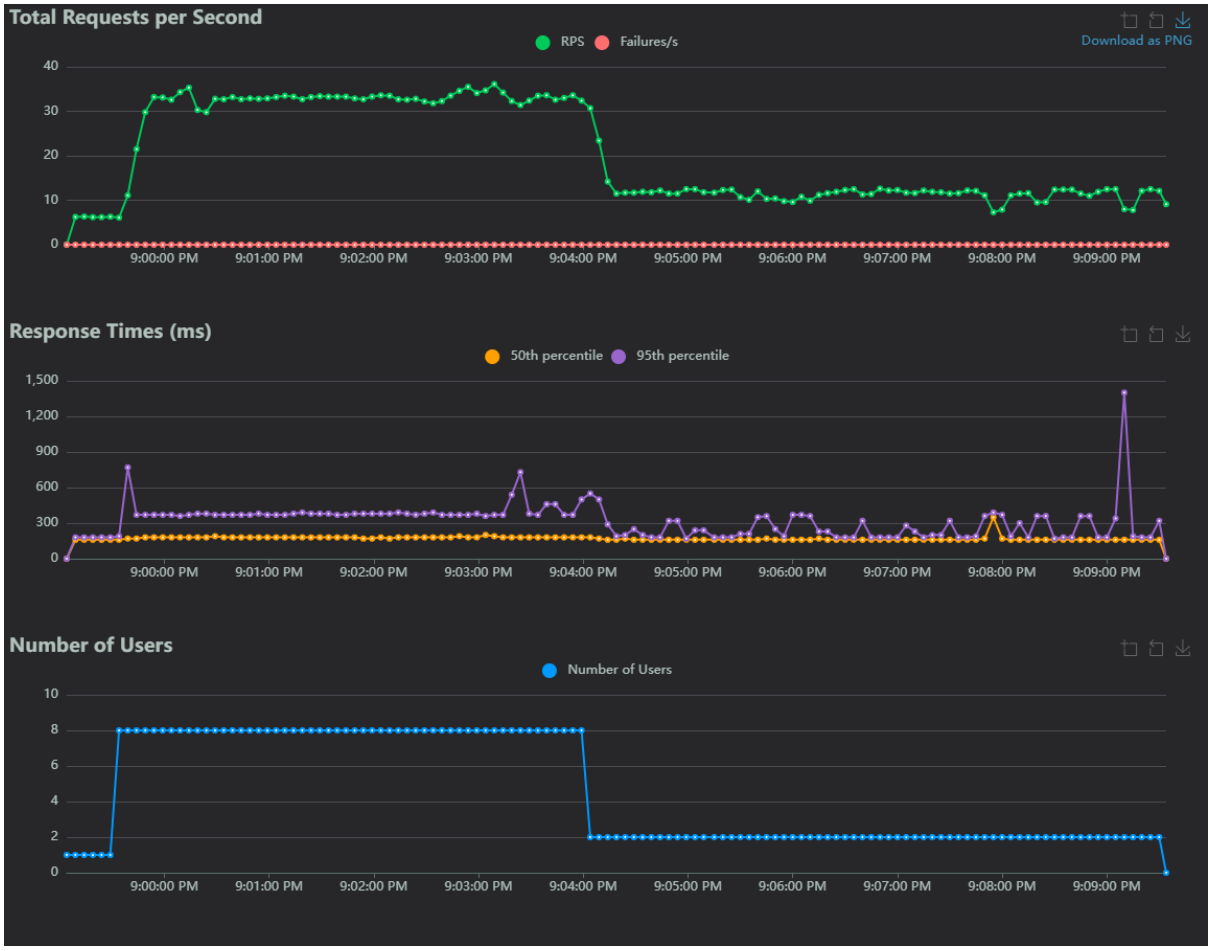

Figure 13: Light C — Locust charts (spike).

Figure 14: Light C — Lambda Invocations, ConcurrentExecutions, Duration.

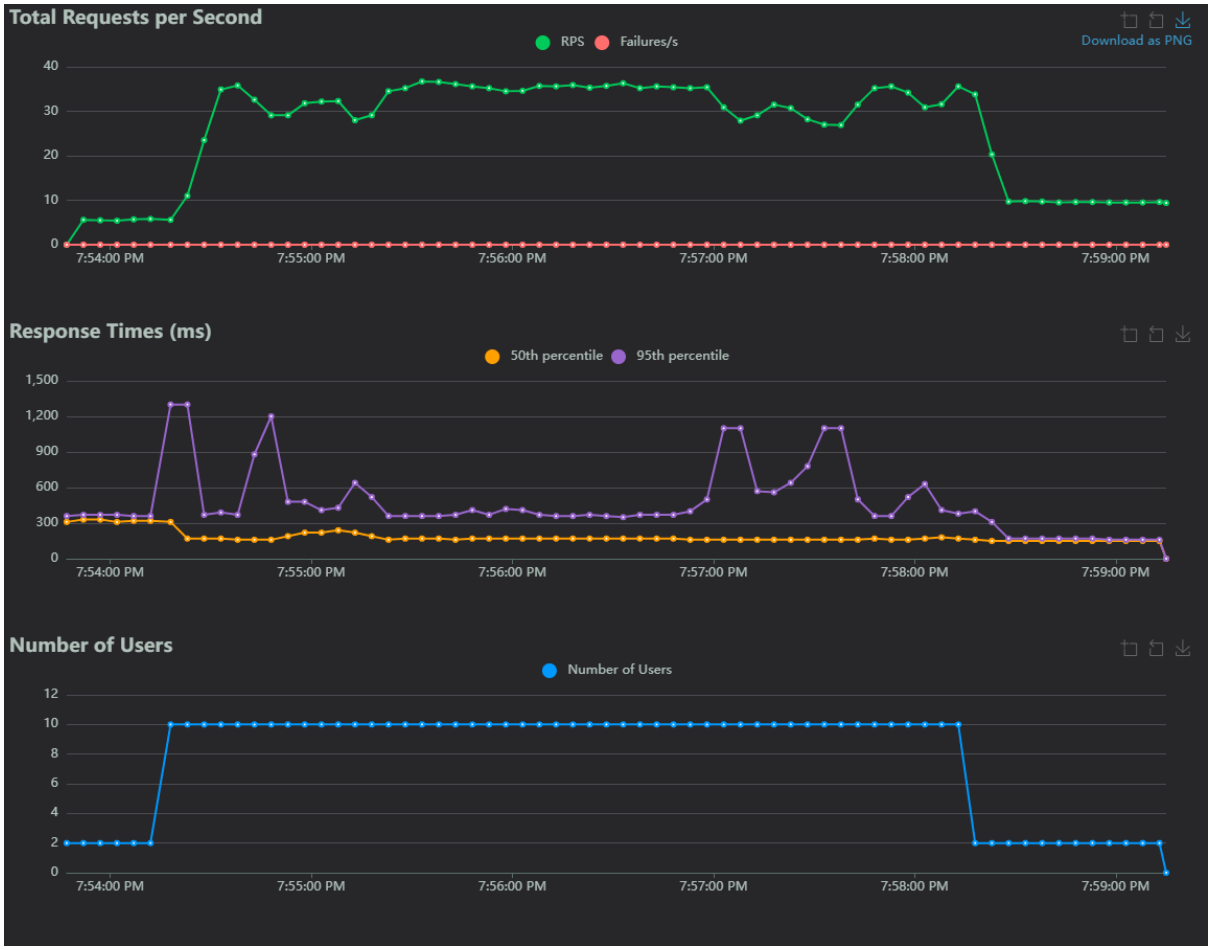

Figure 15: Heavy C — Locust charts (high-intensity spike).

Figure 16: Heavy C — Lambda Invocations, ConcurrentExecutions, Duration.

6.4 Cross-scenario observations

Light vs Heavy. Across A/B/C, Heavy runs exhibit (i) higher plateau RPS in Locust and (ii) a *ConcurrentExecutions* plateau close to the account cap (9–10). Latency percentiles are generally tighter once warmed and stay within the expected budget during S. Light runs remain below saturation and are useful to show stability under realistic pacing.

Cold starts and warm-up. The initial minute (WU) avoids contaminating the reported Locust metrics with cold-start spikes. On Lambda graphs, any residual *InitDuration*/*Duration* spikes appear near the WU boundary, then disappear in the main window.

Availability. No sustained failures are visible in the Locust tables for the provided snapshots. On the provider side, the absence of persistent anomalies in *Duration* and *ConcurrentExecutions* suggests that throttling was either absent or negligible in the main windows. (If point spikes exist at RU transitions, they are short-lived and expected.) The final report will quantify availability from the exported metrics as:

$$\text{Availability} \approx 1 - \frac{\sum(\text{Errors} + \text{Throttles})}{\sum(\text{Invocations})}.$$

6.5 Summary of findings (to be quantified with CSVs)

- Heavy variants reach a concurrency-limited plateau with stable p95/p99 latencies; Light variants show low, steady load with minimal variance.
- Vertical scaling (Heavy B, 512 MB $\rightarrow$ 1024 MB) reduces latency percentiles and increases achievable throughput for the same user pacing, while the concurrency plateau remains unchanged (as expected).
- Under all tested shapes, the service maintains stable behavior during the Steady window, with quick recovery after spike/ramp transitions.

7 Limitations and Future Work

8 Conclusion

This study successfully designed, implemented, and evaluated a fully serverless microservice for Named Entity Recognition, demonstrating the effective application of AWS Lambda for deploying natural language processing workloads. The architecture, built upon AWS Lambda, API Gateway, and the spaCy library, proved capable of delivering low-latency inference while abstracting away all server management concerns. The use of the AWS Serverless Application Model (SAM) ensured that the infrastructure was defined as code, making the deployment process reproducible, version-controlled, and automatable.

The comprehensive performance evaluation, conducted using Locust across multiple workload scenarios, yielded key insights. The system exhibited stable and predictable behavior under both light and heavy load profiles, consistently operating within its performance envelope. The experiments confirmed that the service efficiently scales with incoming request volume, as governed by the AWS Lambda concurrency model. Furthermore, the vertical scaling assessment revealed that increasing allocated memory directly improved inference latency and overall throughput, highlighting a straightforward mechanism for performance tuning. The service maintained high availability throughout testing, with errors and throttling being negligible during steady-state operations.

In conclusion, this project serves as a validated blueprint for deploying efficient, cost-effective, and scalable NLP applications in a serverless environment. The architectural patterns and methodologies employed—from containerized builds to automated deployment and systematic load testing—are directly transferable to other similar computational tasks. The results affirm that serverless computing is a powerful paradigm for event-driven, bursty workloads like NLP microservices, offering a compelling combination of operational simplicity, automatic scaling, and a fine-grained cost model. Future work could explore the integration of larger models, more complex NLP pipelines, and optimization strategies for further reducing cold start latency.

Reproducibility & Repository Notes

The repository is organized for "clone $\rightarrow$ run" on Windows:

1. Install Python 3.10+, Docker Desktop, AWS CLI v2, AWS SAM CLI.
2. `python -m venv .venv; .  .\.venv\Scripts\Activate.ps1`
3. `pip install -r requirements.txt`
4. `sam build -use-container`
5. `sam deploy -guided -profile learnerlab`
6. Use `scripts/smoke.ps1` to POST `/ner`.
7. Tear down with `scripts/delete.ps1`.

Ethical, Cost, and Budget Considerations

We designed for negligible standing cost and minimal environmental impact: no always-on compute, small artifacts, and conservative logs. Load tests remain moderate to avoid unintended denial-of-service behavior and unnecessary spend.

References

[1] ExplosionAI. *spaCy*. https://spacy.io/.

[2] ExplosionAI. *NER in spaCy*. https://spacy.io/api/entityrecognizer.

[3] AWS. *AWS Serverless Application Model (SAM)*. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/.

[4] AWS. *AWS Lambda Developer Guide*. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[5] AWS. *Amazon API Gateway HTTP APIs*. https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api.html.

[6] AWS. *Amazon CloudWatch Metrics for Lambda*. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html.

[7] Locust. *A modern load testing framework*. https://locust.io/.