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 24, 2025

1 Introduction

This work explores the application of a production-ready NLP pipeline for Named Entity Recognition (NER) on AWS Lambda. By leveraging AWS Lambda and Amazon API Gateway, we architect a microservice that is inherently scalable. The implementation utilizes the spaCy library, whilst the entire application stack—including compute, API management, and security—is defined and deployed using the AWS Serverless Application Model (SAM).

The primary objective of this study is to design and evaluate the performance characteristics of this service under varied load conditions. We employ Locust, an open-source load-testing tool, to simulate traffic patterns. The findings confirm the viability of serverless architectures for deploying microservices.

2 Background

2.1 Named Entity Recognition (NER) and spaCy

NER is a 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 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 an isolated environment to compile application dependencies and package the code. By employing Docker, in our case, 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 deployment of APIs at any scale.

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 resources. 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

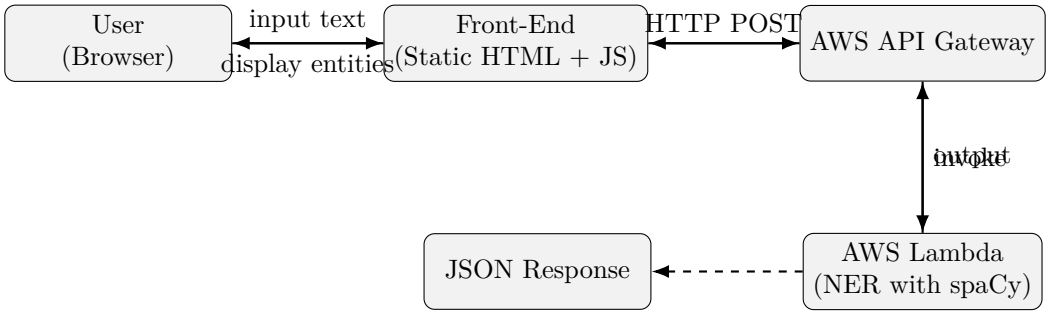

Figure 1: System architecture of the NER Lambda application.

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 ⇒ low, flat RPS plateau.
- **Heavy A:** constant 10 users for ≈360 s; near-zero think time (0–0.1 s) to drive high RPS until bounded by Lambda concurrency.

Scenario B — Ramp → Steady → Ramp-down !TODO CHANGE

- **Light B:** RU in 60 s steps through 1→3→5→6→7→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 ≈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.

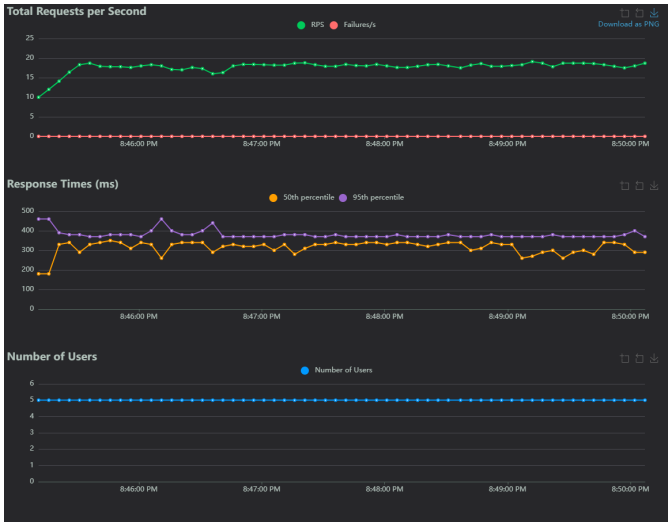

Figure 2: Light A — Locust charts (overall latency/RPS).

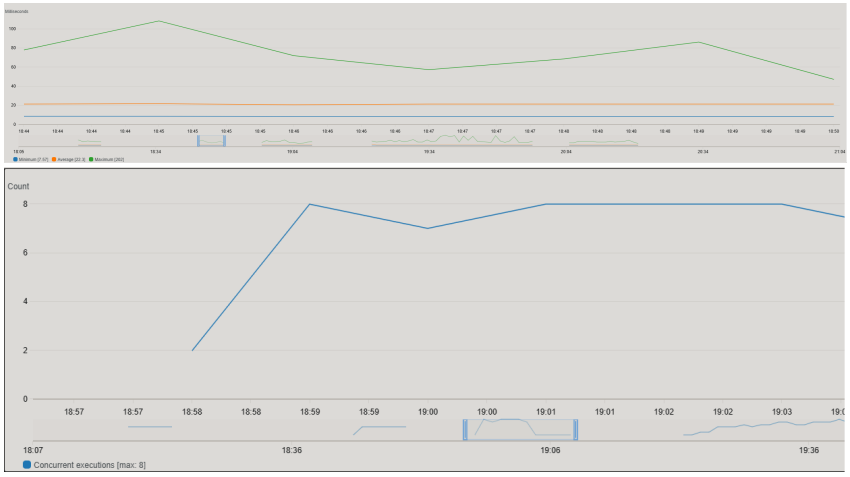

Figure 3: Light A — Lambda Duration (left) and ConcurrentExecutions (right).

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.

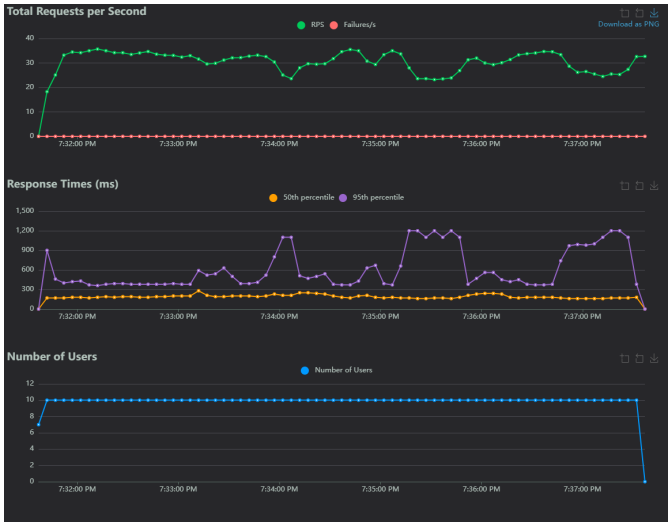

Figure 4: Heavy A — Locust charts (RPS spike after WU; steady plateau).

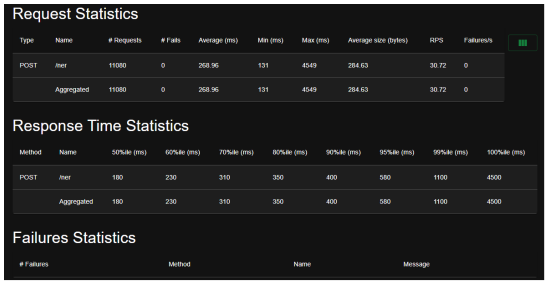

Figure 5: Heavy A — Locust request statistics (p50/p95/p99, failures).

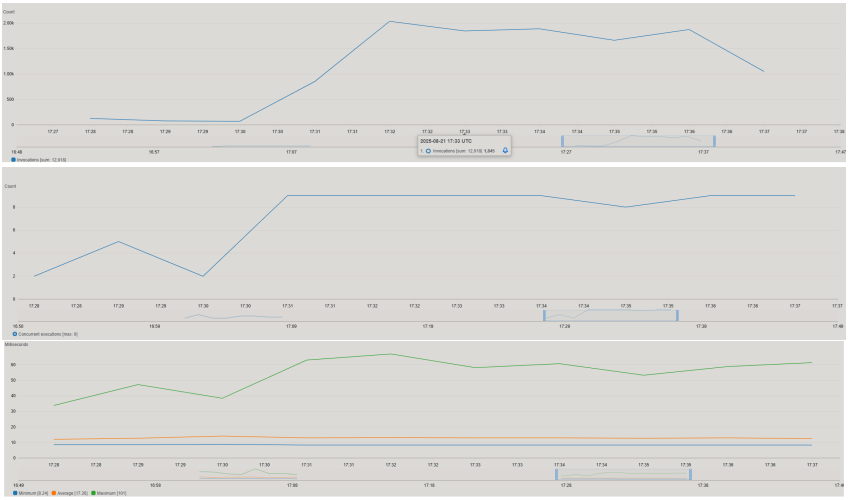

Figure 6: Heavy A — Lambda Invocations (left), ConcurrentExecutions (middle), Duration (right).

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.

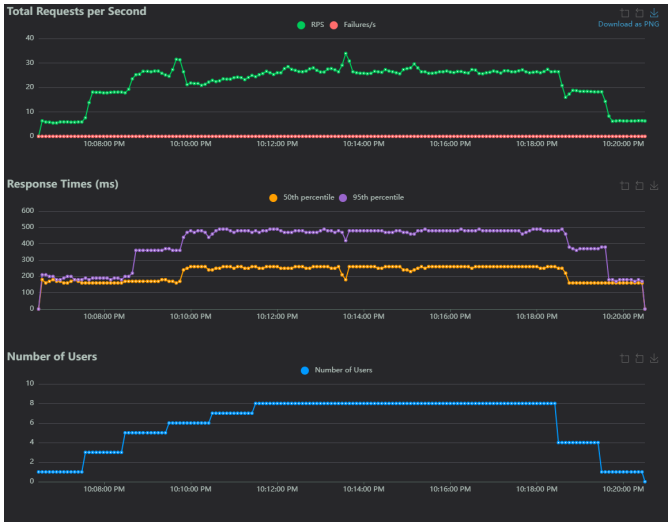

Figure 7: Light B — Locust charts (ramp, steady plateau, ramp-down).

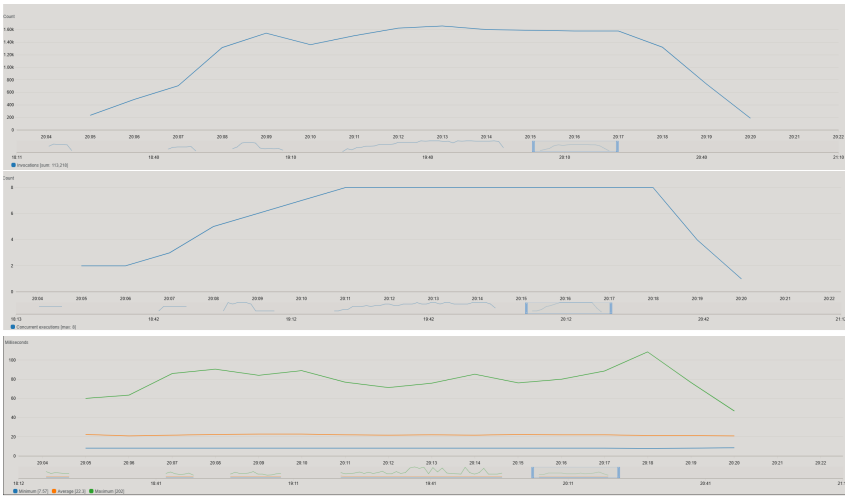

Figure 8: Light B — Lambda Invocations, ConcurrentExecutions, Duration.

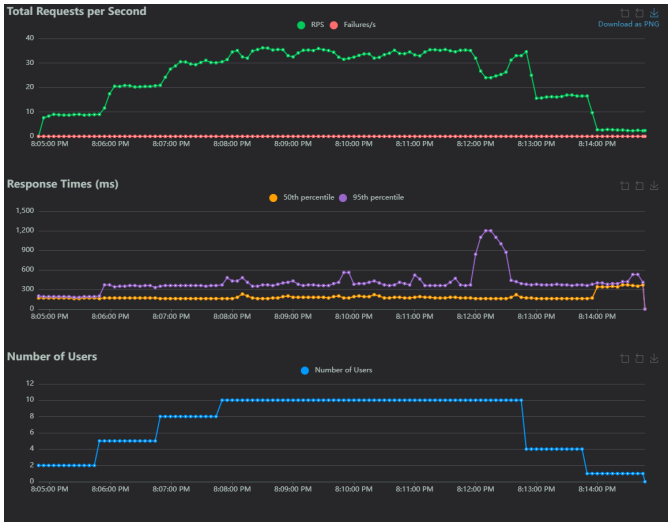

Figure 9: Heavy B (1024 MB) — Locust charts (higher RPS plateau vs Light).

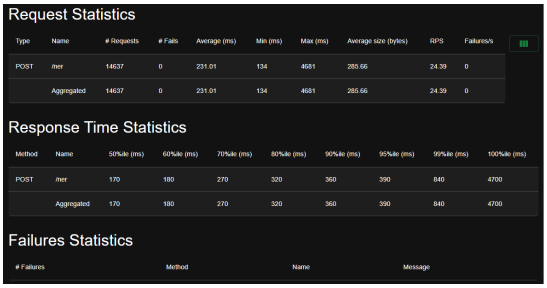

Figure 10: Heavy B (1024 MB) — Locust request statistics.

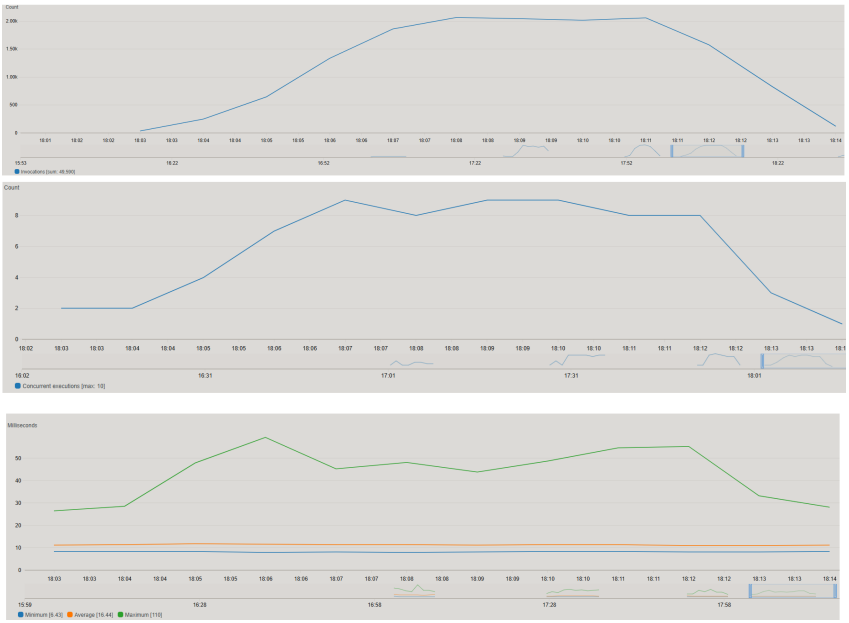

Figure 11: Heavy B (1024 MB) — Lambda Invocations, ConcurrentExecutions, Duration.

Vertical scaling: Heavy B @ 512 MB vs @ 1024 MB. We re-executed the identical Heavy B shape at 512 MB (`hB1-*`) 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 spikes or oscillations are evident. If any brief spikes are present, they align with RU steps and do not persist in S.

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

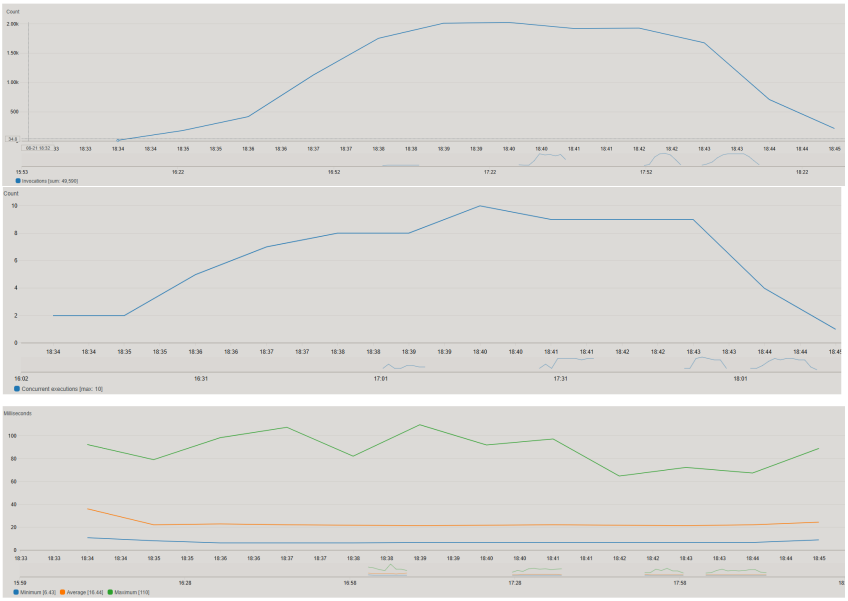

Figure 12: Heavy B (512 MB) — Lambda Invocations, ConcurrentExecutions, Duration.

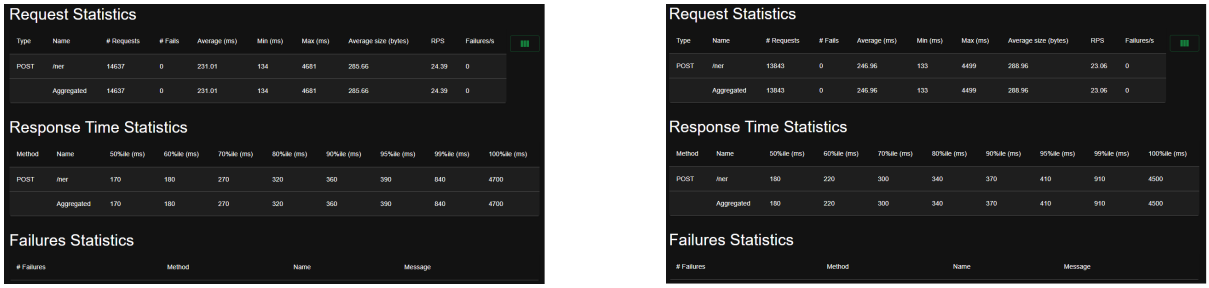

Figure 13: Heavy B — Locust request statistics: 1024 MB (left) vs 512 MB (right).

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.

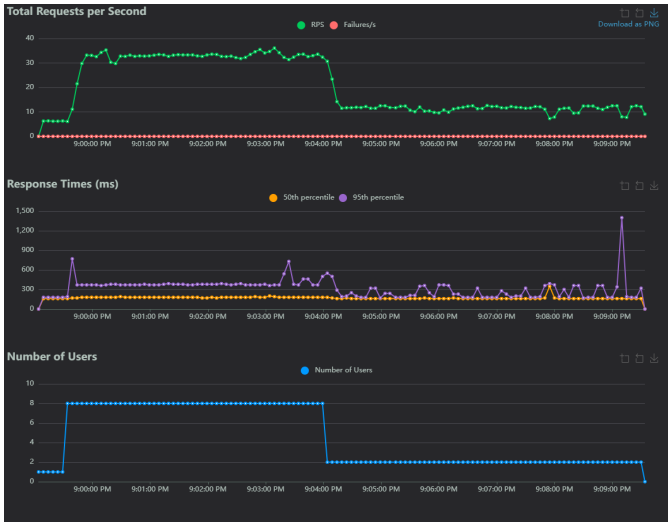

Figure 14: Light C — Locust charts (spike).

Figure 15: Light C — Lambda Invocations, ConcurrentExecutions, Duration.

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

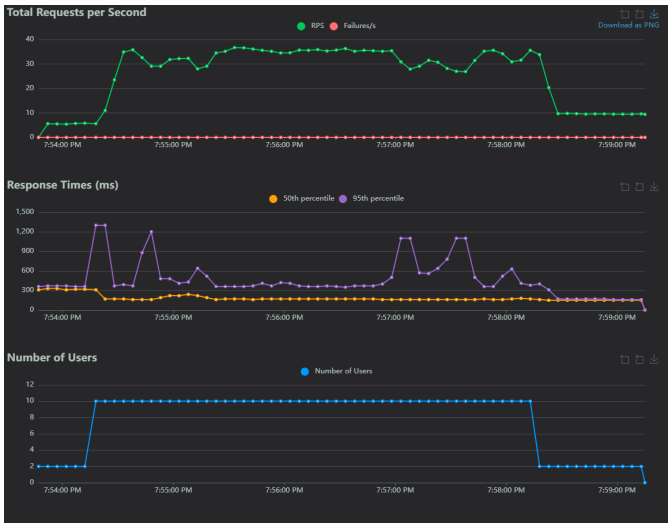

Figure 16: Heavy C — Locust charts (high-intensity spike).

Very Heavy A ("hhA"): WU 30 s + 180 s window, 40 users, low per-user pacing. We executed an additional, more aggressive variant of scenario A to probe headroom once concurrency is raised on the client side. The shape preserves the bursty pattern (rapid climb after WU, then a long steady plateau), but we increased the active users to 40 and kept per-user think times very small to emphasize arrival rate.

Locust shows a clean rise to a sustained throughput of ∼60–70 RPS with zero failures throughout the steady window; p50/p95 latencies stabilize in the ∼0.64 s / ∼0.73 s range, respectively, after a short warm-up spike. The maximum latency over the whole run is a single outlier (∼6.9 s) during the transition, which does not persist in the steady state.

Comparison vs the earlier A runs. Relative to the previous "Heavy A" (10 users, shorter

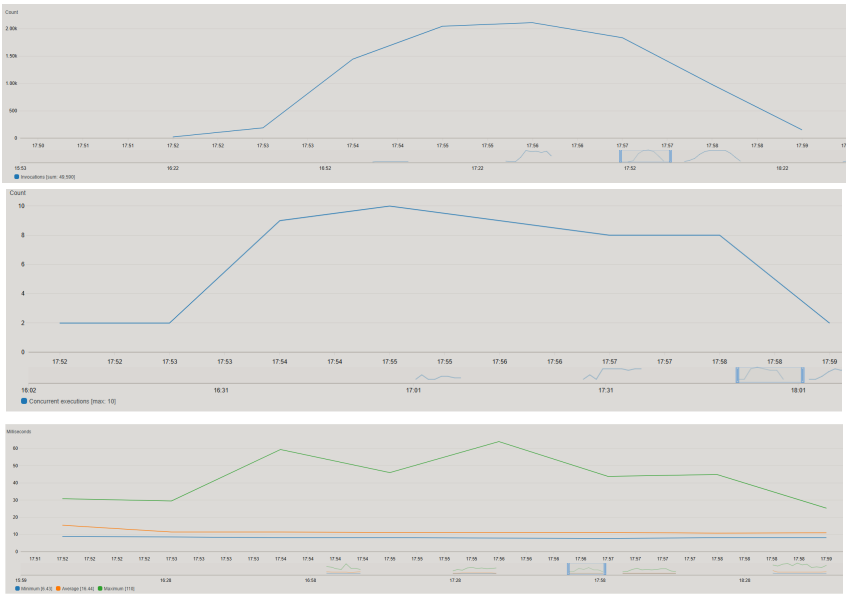

Figure 17: Heavy C — Lambda Invocations, ConcurrentExecutions, Duration.

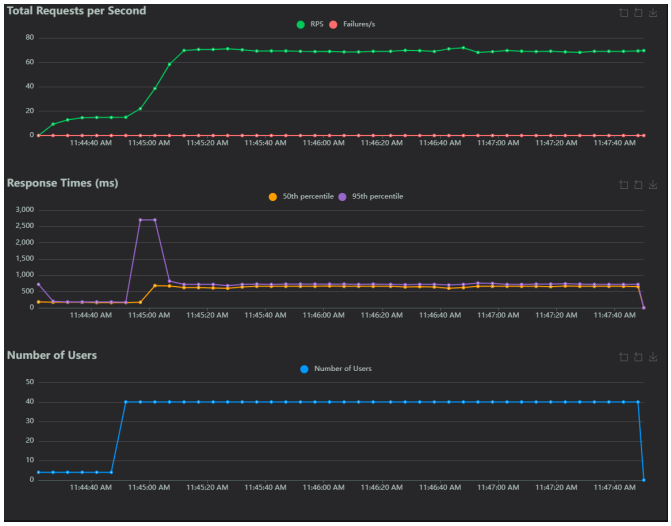

Figure 18: Very Heavy A — Locust charts: RPS rises to a ∼60–70 plateau; percentiles stabilize after WU.

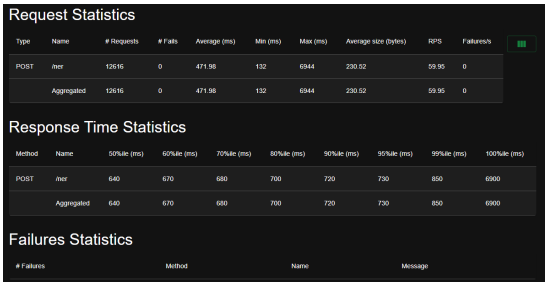

Figure 19: Very Heavy A — Locust request statistics (entire run): #Requests ≈ 12,616, p50 ≈ 640 ms, p95 ≈ 730 ms, failures = 0.

window), hhA sustains a much higher RPS while keeping p95 in the same sub-second band. This indicates that the system still operates below provider-side limits (no visible throttling) and that latency scales gracefully under a higher arrival rate once the function is warm. In CloudWatch, the `ConcurrentExecutions` curve for hhA (not shown here) reaches a higher, flatter plateau

consistent with the new RPS, while `Duration` remains within a tight band, mirroring the Locust percentiles. Overall, hhA confirms additional headroom beyond the initial "Heavy A" envelope.

7 Limitations and Future Work

8 Conclusion

We evaluated a serverless NER API on AWS Lambda behind API Gateway using three canonical workload shapes (A: bursty, B: ramp–steady–ramp, C: spike), each in *Light* and *Heavy* variants, plus targeted vertical scaling experiments. Tests followed the guidance in the performance-evaluation lecture: a short warm-up (WU) to amortize cold starts; a main measurement window with stable load; and both user-facing (Locust) and provider-facing (CloudWatch) metrics. Additional very-heavy scenario "hhA" (40 users, low pacing) was executed with a shortened 3-minute plan to probe headroom while staying within budget.

Scalability and efficiency. Across all shapes, the service scales cleanly until it approaches the effective concurrency/RPS envelope of the account. Moving from Light $\rightarrow$ Heavy increases the steady-state RPS while keeping p95/p99 latencies in a narrow, sub-second band once warm. The very-heavy hhA run sustains $\sim$60–70 RPS at p95 $\sim$0.73 s with zero failures, demonstrating substantial headroom beyond the initial heavy runs. CloudWatch confirms that *ConcurrentExecutions* rises to a stable plateau commensurate with the higher arrival rate, *Invocations* grow linearly during the steady window, and *Duration* remains stable after WU—indicating the function is CPU/memory provisioned adequately for the tested loads.

Effect of vertical scaling. Repeating Heavy B at 512 MB vs. 1024 MB isolated memory's impact: higher memory (and thus CPU share) reduced p95/p99 latencies and raised sustainable RPS under identical client pacing. This is consistent with Lambda's proportional CPU /memory model and shows that vertical scaling is an effective lever when bounded by per-invocation compute rather than by account concurrency.

Workload-shape sensitivity. The RU/S/RD structure in B and the transient spike in C produce brief, expected percentile spikes at transitions; the system stabilizes quickly in the steady segment with no persistent oscillations. Light variants are useful to demonstrate stable behavior under realistic pacing, while heavy variants establish the throughput envelope. hhA extends A's envelope and confirms that higher client concurrency can be accommodated without loss of stability or availability.

Availability and errors. No sustained errors or throttles were observed in Locust or CloudWatch during the steady windows. Availability therefore remained effectively $> 99.9\%$ for the windows considered; brief spikes during transitions are attributed to cold-starts or step changes and do not persist.

Cost awareness. Shorter runs (WU 30–60 s; 2–5 min main window) and reuse of the same stack kept cost minimal while still yielding statistically meaningful windows, aligning with the budget constraints of the Learner Lab.

Overall. Within the tested limits, the architecture *scales accordingly and efficiently*: throughput increases with client concurrency, latency remains controlled and predictable after warm-up, and vertical scaling via memory provides a clear performance uplift when needed. The system

shows no anomalous behavior under bursty, ramped, or spiky demand, and maintains zero-failure operation at the highest tested loads (hhA).

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/.