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 25, 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. The choice of NER as the target workload, rather than other AI models is motivated by its balance of practical relevance and clear performance indicators. NER is a core component of many real-world applications where extracting structured information from unstructured text is essential. Application fields include news and document analytics, compliance monitoring, biomedical literature mining, customer support automation, and conversational agents.

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

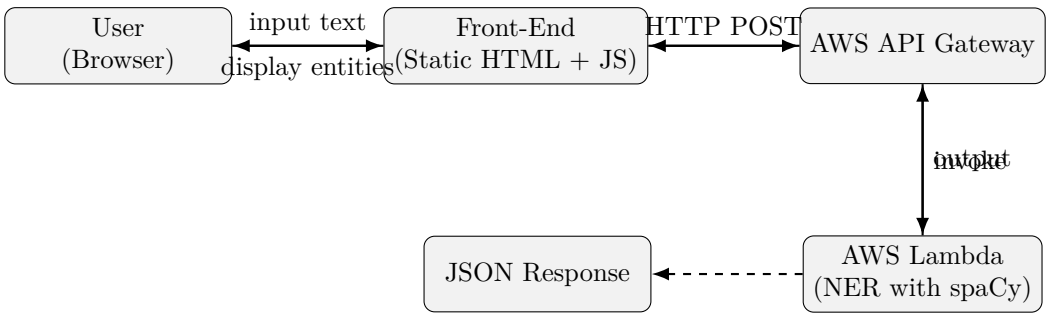

Figure 1: System architecture of the NER Lambda application.

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 Module (`src/ner.py`)

This module is responsible for the NER task: it utilizes the spaCy library, a framework for NLP in Python. The pre-trained 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.

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. 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.

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 NER service. It is implemented as a static `HTML` page with basic `JavaScript` logic.

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. 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 Common settings across all scenarios

The experiments share a set of common settings. First, a Warm-up (WU) phase of 60 s at 2 users (spawn rate 2/s) is executed in order to trigger cold starts. Immediately after the warm-up phase the client statistics are reset so that the run-up, steady, and run-down periods are isolated in the reported numbers. The payload mix consists of short and long English sentences containing named entities such as persons, organizations, locations, and dates; requests are performed via POST /ner with a small JSON body. Each scenario is executed in both a *Light* and a *Heavy* profile: the Light profile uses lower request rates per second (RPS) with more realistic user pacing, while the Heavy profile uses higher RPS and near-zero client think time to approach the Lambda concurrency ceiling.

5.3 Workload scenarios

We use three canonical workload shapes as a methodological basis to stress different operational aspects of the system under test, as depicted in Table 1. Durations reported below always refer to the *main window* (after the warm-up and statistics reset). "Users" denote active virtual users, while the effective request rate (RPS) emerges from the interplay of user count, client think time, and function latency. The selection of these three shapes is motivated by the need to emulate diverse yet representative access patterns. The *Bursty* scenario models sustained but intermittent interaction, suitable for capturing cold start recovery and concurrency stabilization. The *Ramp–Steady–Ramp-down* scenario represents a gradual increase in load, followed by a sustained plateau and subsequent decrease, which allows evaluation of scaling dynamics. The *Spike* scenario emulates sudden surges of demand, providing insights into system responsiveness under rapid load fluctuations.

Scenario	**Profile**	**Description**
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.
B — Ramp → Steady → Ramp-down	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	Heavy B executed 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.
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).

Table 1: Definition of workload scenarios (Light and Heavy profiles).

5.4 Extra: Stress test scenario

The decision to conduct a stress test, even with the known constraint of the AWS Learner Lab's resource limits, was a calculated one aimed at challenging the boundaries of the serverless architecture. While the used account was eventually suspended for exceeding its concurrency quota (10 concurrent executions), the outcome itself was a valuable data point. Crucially, the client-side metrics collected by Locust prior to the suspension were preserved. However, CloudWatch metrics were not available after the run, due to the account's suspension. The stress test scenario was therefore executed with a shortened 3-minute window to probe headroom while staying within budget.

Table 2: Test composition for the stress test run.

Stage	Duration	Active users	Spawn behavior	Per-user think time
Warm-up (WU)	30 s	2	2 users/s (WU only)	near-zero
Main window	180 s	40 (steady plateau)	rapid climb after WU	≈0–0.1 s

6 Results and discussion

The primary objective of this evaluation is to assess whether the system under study maintains its service quality under stress conditions. Before examining the results, it is useful to clarify how the reported charts and metrics should be read and interpreted. The Locust charts primarily display the evolution of requests per second (RPS) together with latency percentiles (p50, p95, p99). The RPS curve reflects the effective throughput generated by the virtual users, while the percentile lines capture the distribution of response times. When both curves settle into a stable plateau, this indicates that the system has reached a steady operating state. Complementing these charts, the Locust request statistics provide aggregated values of latency percentiles, request counts, and possible failures, thereby allowing the tails of the latency distribution and the presence or absence of execution errors to be assessed at a glance. On the provider side, the AWS Lambda metrics add further explanatory depth. The *Invocations* series tracks the cumulative number of requests served, where a linear slope corresponds to constant throughput. The *ConcurrentExecutions* metric shows the number of function instances active at a given moment, thus making concurrency ceilings and the elasticity of scaling behavior visible. The *Duration* series measures execution time per invocation, with narrow and stable bands signifying predictable performance and wider fluctuations revealing potential variability.

Our reading of the experiments is two-sided and evidence-led. First, we cross-check throughput by comparing Locust RPS with the slope of CloudWatch *Invocations*; the two match in shape and level during steady windows, confirming that client-side and provider-side views are consistent. Second, we validate elasticity by verifying that *ConcurrentExecutions* rises and falls with the arrival rate (bursts, ramps, and spikes) and returns to baseline after load drops, indicating healthy scale-out/scale-in. Third, we bound latency using Locust percentiles: p50 captures the typical user experience, while p95/p99 quantify tail behavior; brief transitions (ramp steps and spikes) are assessed separately from steady windows. Finally, we scan for stress signals: failures in Locust, and visual cues of throttling or saturation in CloudWatch (e.g., a flat concurrency ceiling or diverging RPS vs. invocations).

Across all shapes, the service sustains steady throughput with low median latency and bounded tails, without recorded request failures. In the heavy profile, Locust reports RPS $\in [23.06, 30.72]$ with scenario medians p50 = 170–180 ms and p95 = 390–580 ms (p99 = 840–1100 ms); the highest observed steady-state throughput is 30.72 req/s in Heavy A (bursty). In the light profile, throughput spans RPS $\in [17.90, 22.01]$ with p50 = 170–330 ms, p95 = 370–480 ms, and p99 $\leq$ 500 ms. Max latencies of 4–5 s appear as rare outliers that coincide with abrupt

fan-out (spikes/step-ups), a pattern consistent with cold starts rather than sustained saturation. CloudWatch corroborates these patterns: *ConcurrentExecutions* traces mirror the input shapes (bursts, ramps, spikes), *Invocations* grow linearly during steady windows (constant effective throughput), and *Duration* bands remain tight once warm. No evidence of a hard concurrency ceiling or plateaued throughput is visible in any scenario, which supports the conclusion that the application scales elastically within the tested load envelope.

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. 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.

7 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 and an heavier stress test. An additional stress test scenario was executed with a shortened 3-minute plan. 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 (stress test). Future work could focus on refining both the scalability experiments and the methodology for measurement. On the experimental side, one direction could be to extend the range of load profiles beyond the canonical shapes used in this work or to explore other workload patterns. On the methodological side, we could investigate the impact of different client-side configurations on the observed behavior, such as varying think times or concurrency levels.

8 Charts and statistics

In the next pages, we present the charts and statistics for each scenario. In order, we present the following: light tests (A, B, C), heavy tests (A, B, B with lower memory, C), and the stress test (A). The test naming and the letters used to identify the scenarios are defined in the `Workload scenarios` subsection.

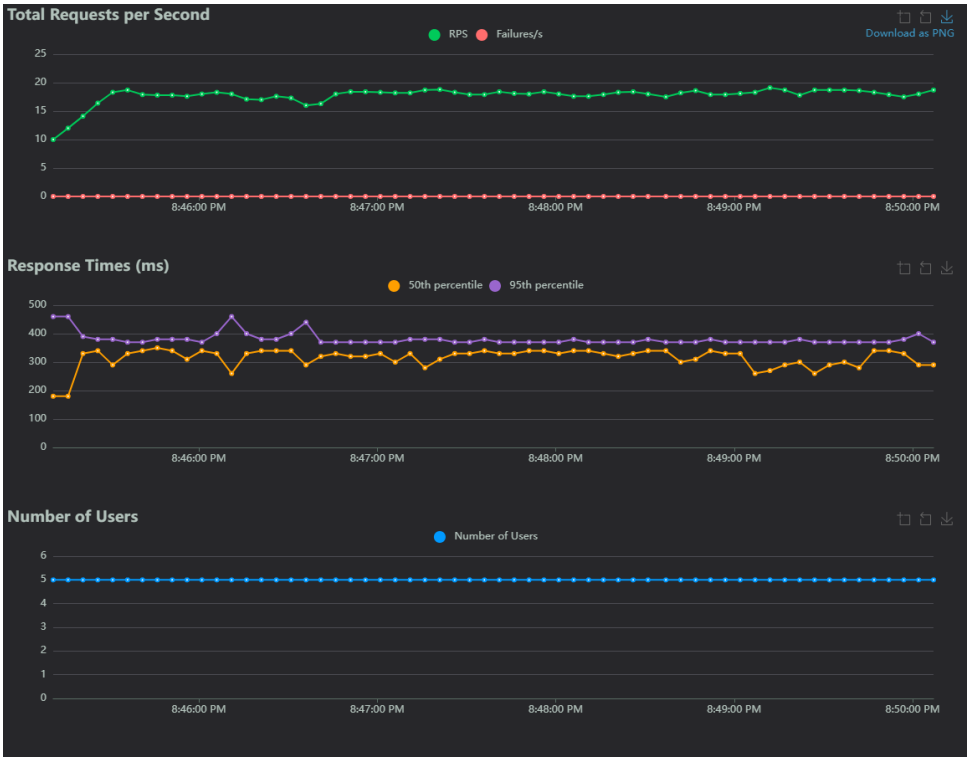

Figure 2: Light A — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	5380	0	278.23	134	4749	283.32	17.9	0
	Aggregated	5380	0	278.23	134	4749	283.32	17.9	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	330	340	350	360	370	380	450	4700
	Aggregated	330	340	350	360	370	380	450	4700

Failures Statistics

# Failures	Method	Name	Message

Figure 3: Light A — Locust request statistics.

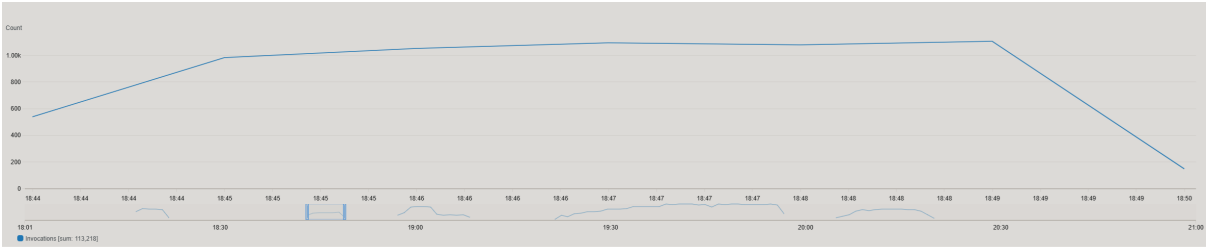

Figure 4: Light A — Lambda Invocations.

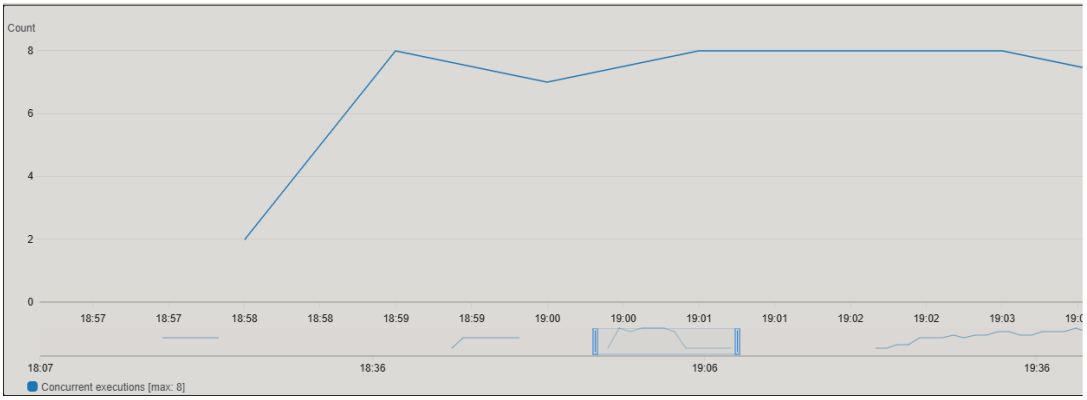

Figure 5: Light A — Lambda ConcurrentExecutions.

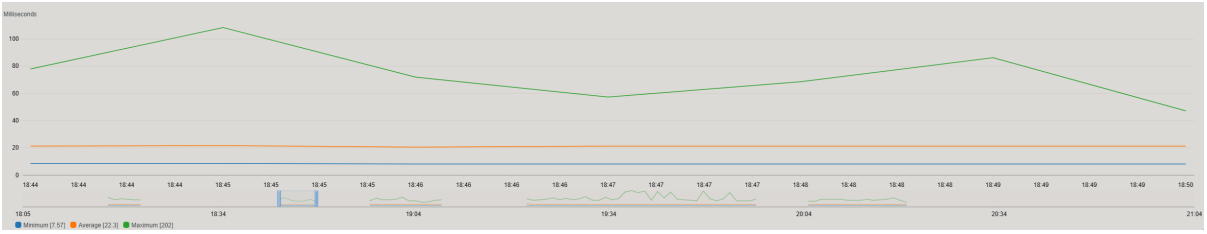

Figure 6: Light A — Lambda Duration per invocation.

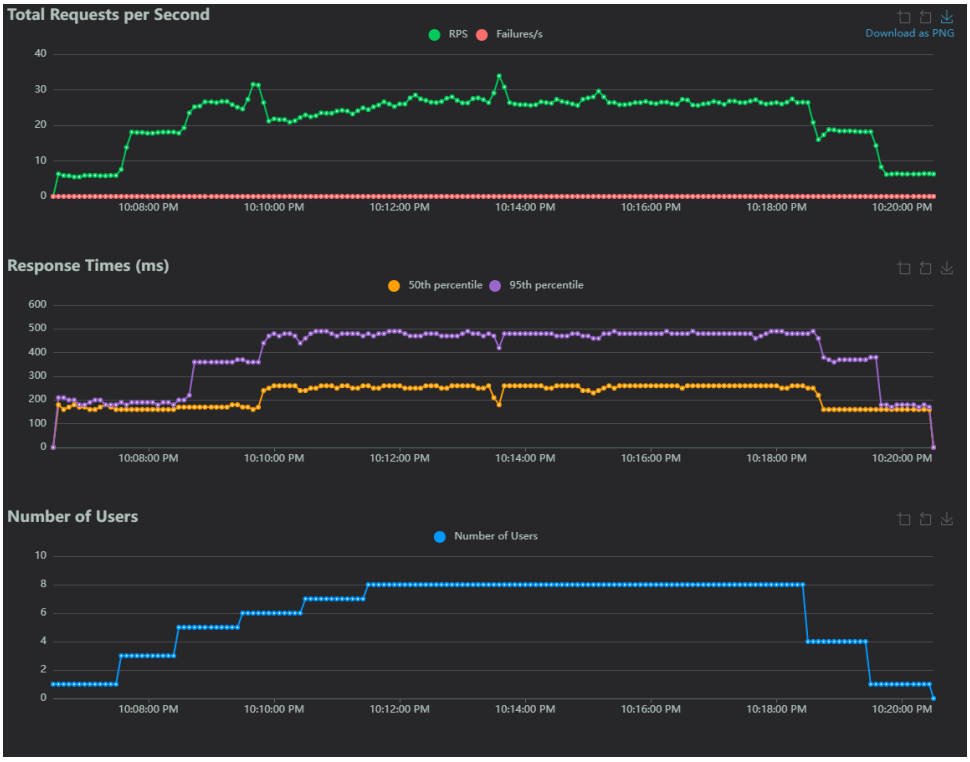

Figure 7: Light B — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	18513	0	268.51	133	4261	287.21	22.01	0
	Aggregated	18513	0	268.51	133	4261	287.21	22.01	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	240	260	270	380	460	480	500	4300
	Aggregated	240	260	270	380	460	480	500	4300

Failures Statistics

# Failures	Method	Name	Message

Figure 8: Light B — Locust request statistics.

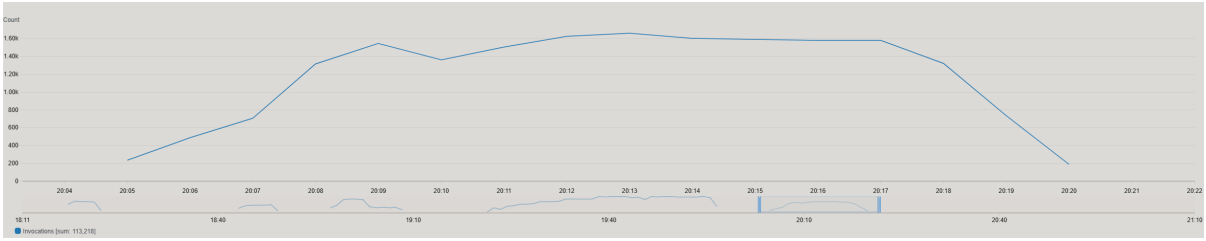

Figure 9: Light B — Lambda Invocations.

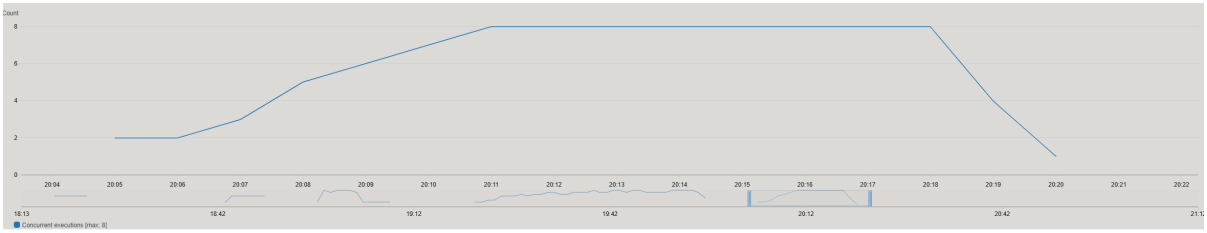

Figure 10: Light B — Lambda ConcurrentExecutions.

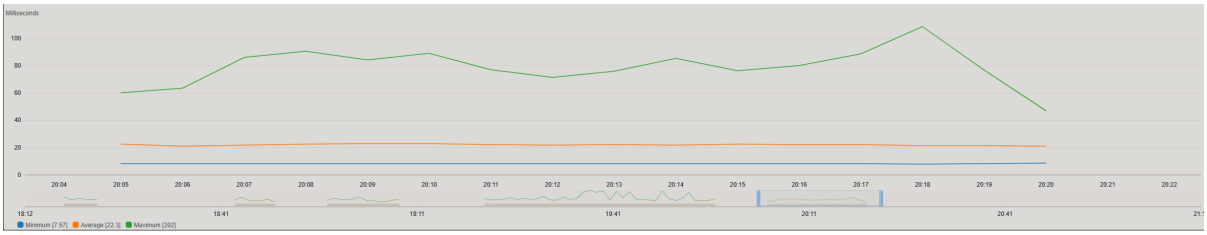

Figure 11: Light B — Lambda Duration per invocation.

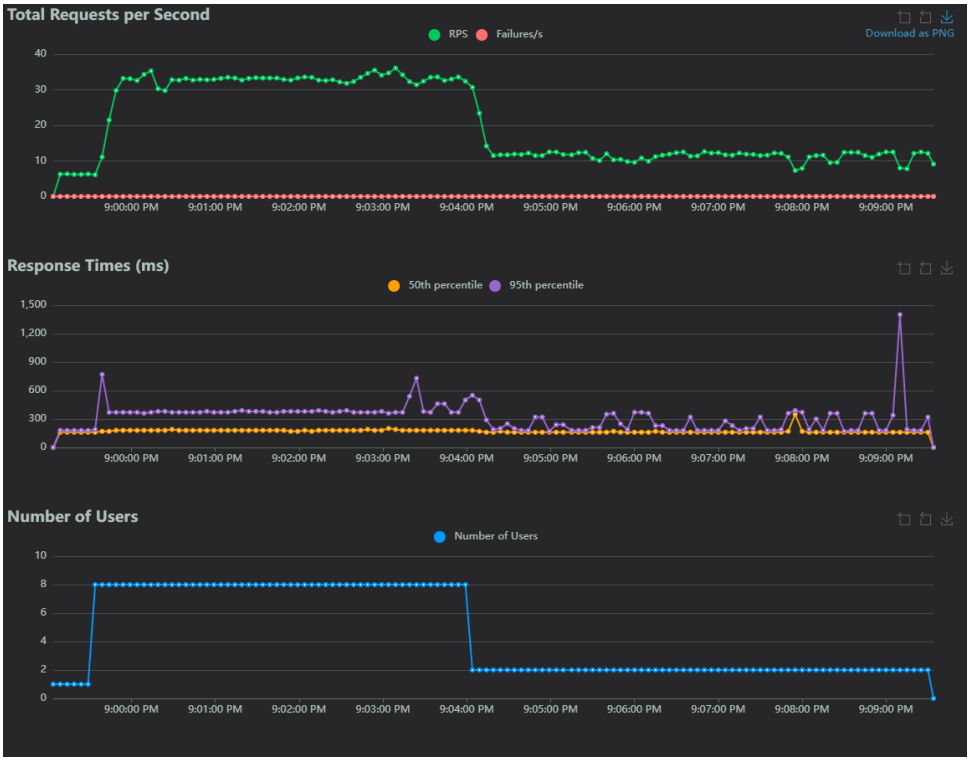

Figure 12: Light C — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	12755	0	223.12	134	5308	287.51	20.22	0
	Aggregated	12755	0	223.12	134	5308	287.51	20.22	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	230	340	360	370	500	5300
	Aggregated	170	180	230	340	360	370	500	5300

Failures Statistics

# Failures	Method	Name	Message

Figure 13: Light C — Locust request statistics.

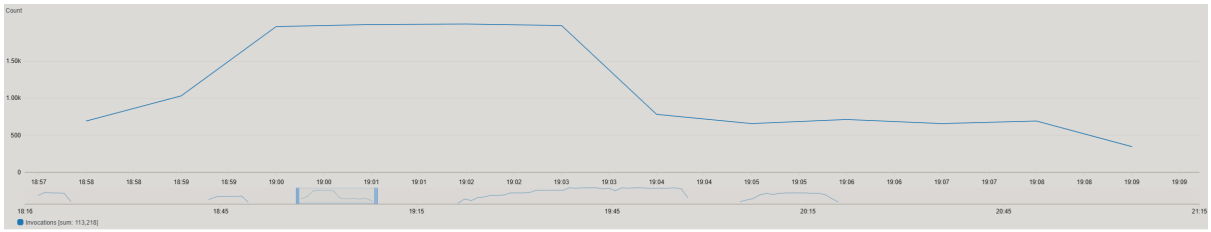

Figure 14: Light C — Lambda Invocations.

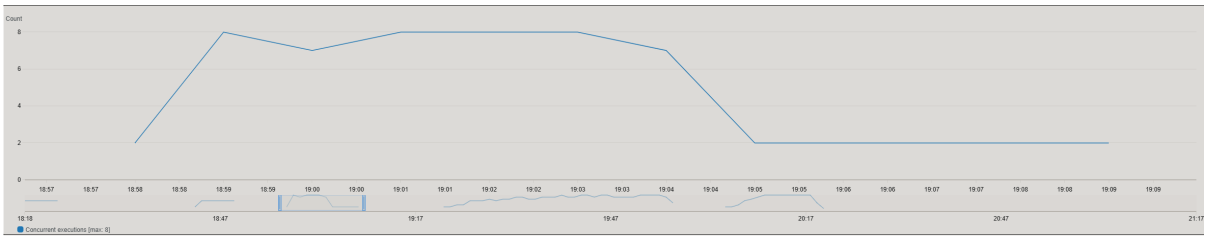

Figure 15: Light C — Lambda ConcurrentExecutions.

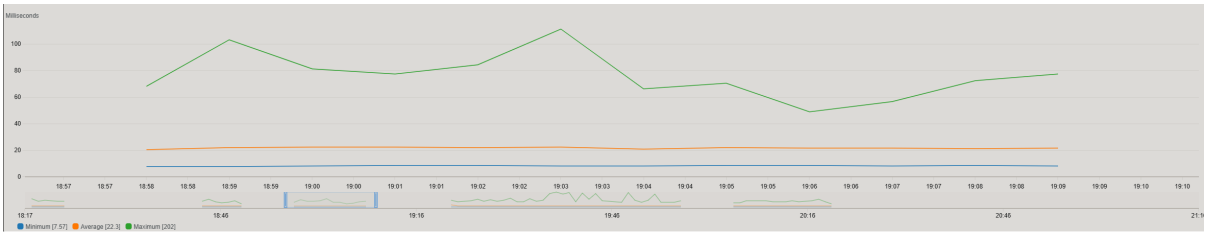

Figure 16: Light C — Lambda Duration per invocation.

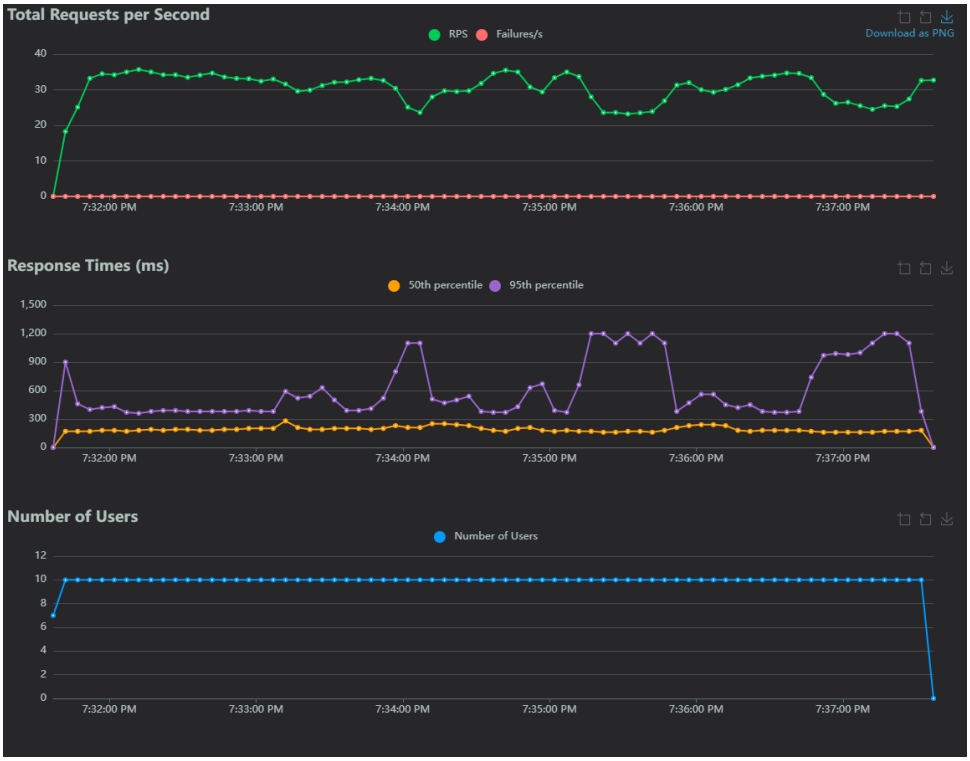

Figure 17: Heavy A — Locust RPS and latency percentiles.

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 18: Heavy A — Locust request statistics.

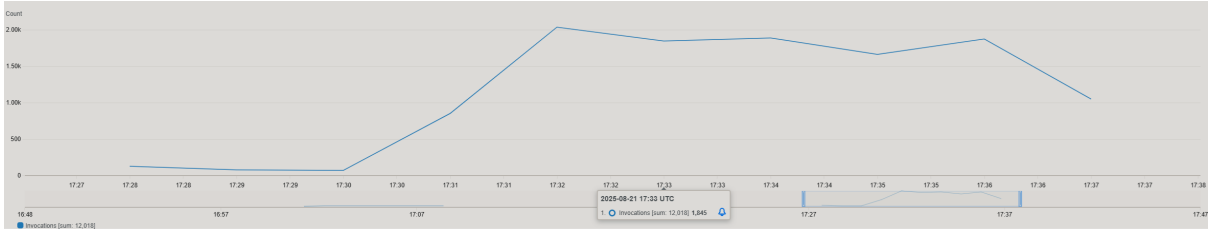

Figure 19: Heavy A — Lambda Invocations (provider-side throughput).

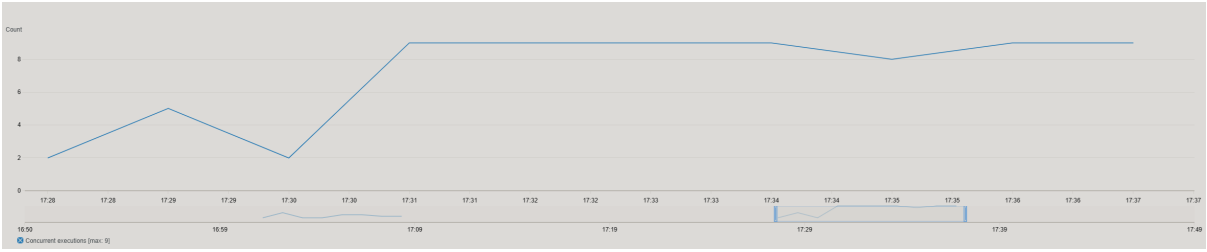

Figure 20: Heavy A — Lambda ConcurrentExecutions (elastic scale-out/in).

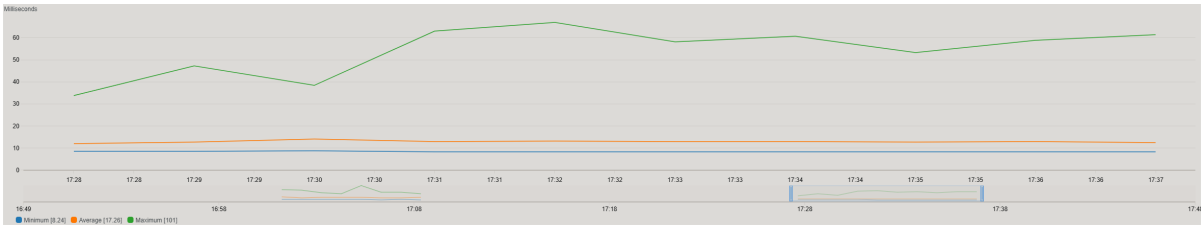

Figure 21: Heavy A — Lambda Duration per invocation.

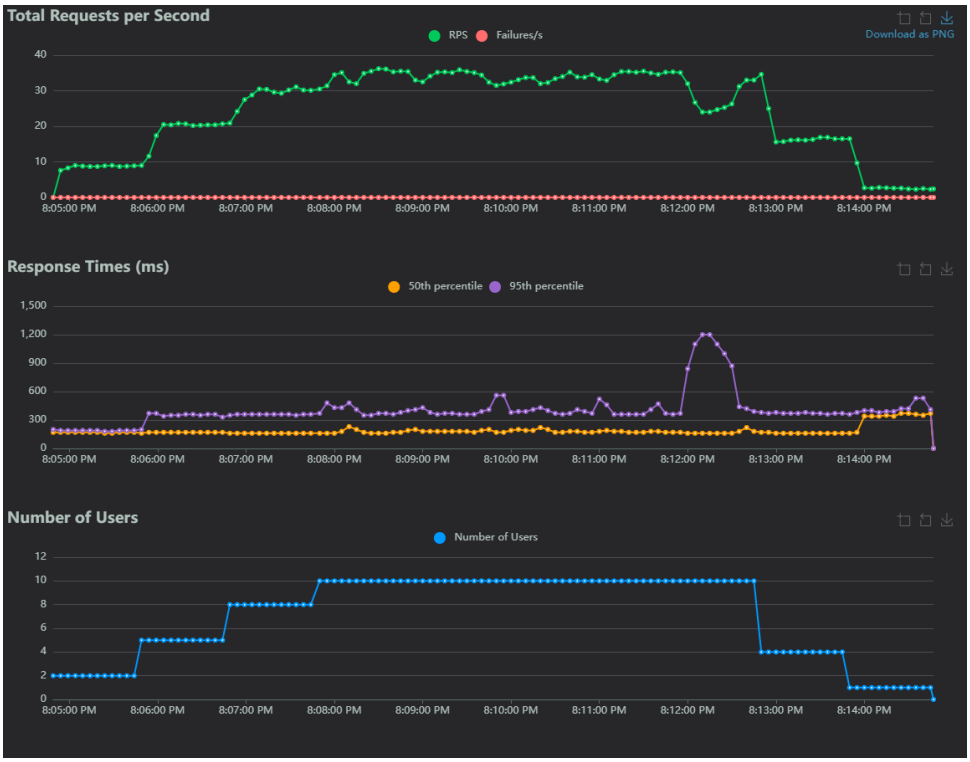

Figure 22: Heavy B — Locust RPS and latency percentiles.

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 23: Heavy B — Locust request statistics.

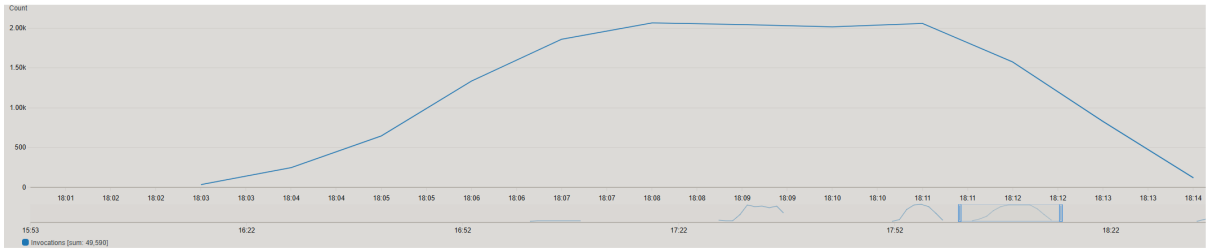

Figure 24: Heavy B — Lambda Invocations (provider-side throughput).

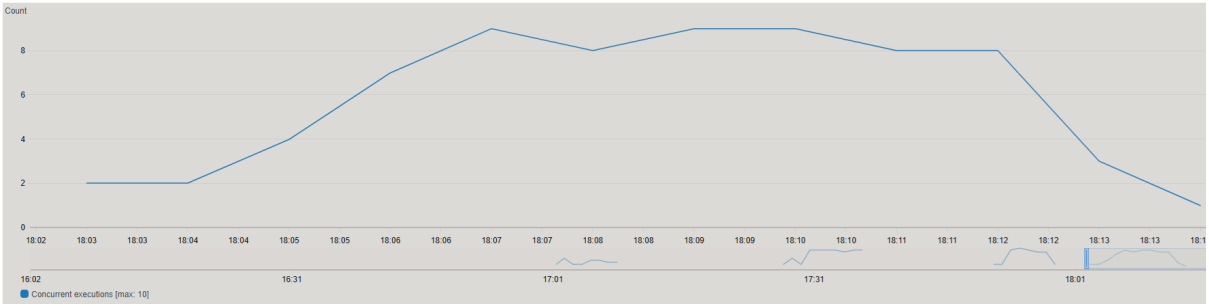

Figure 25: Heavy B — Lambda ConcurrentExecutions (elastic scale-out/in).

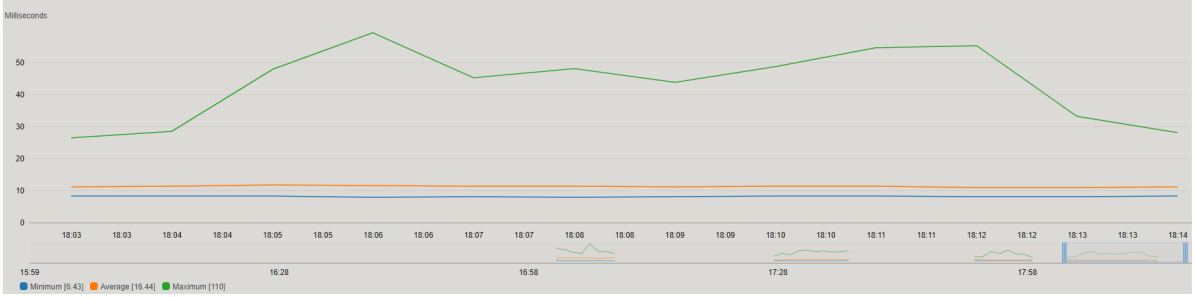

Figure 26: Heavy B — Lambda Duration per invocation.

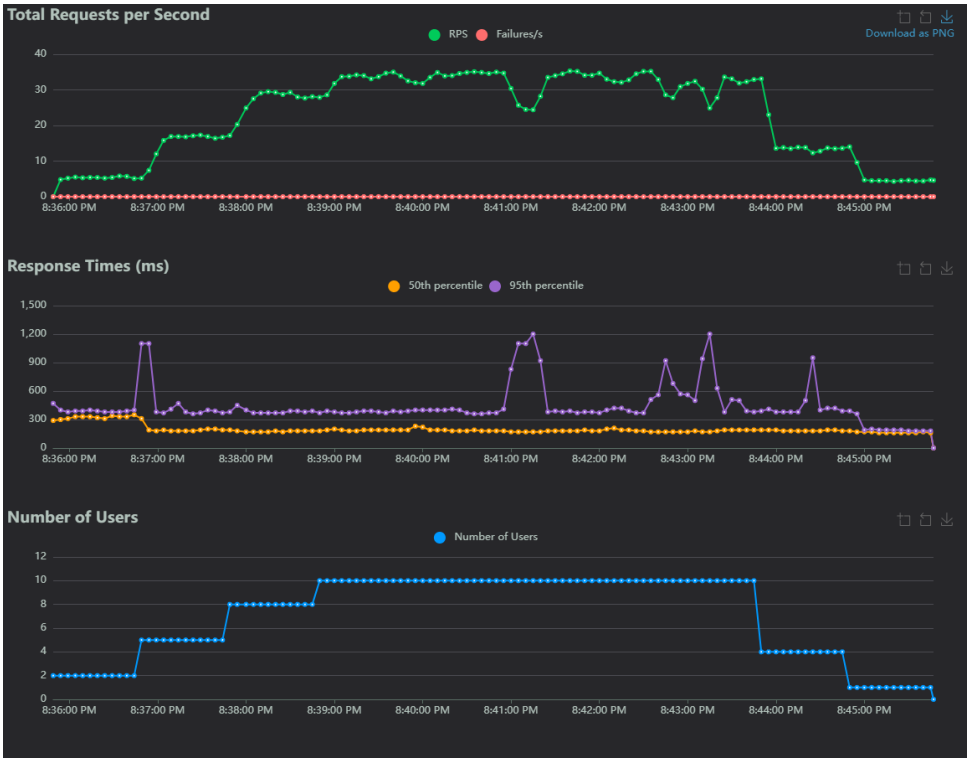

Figure 27: Heavy B (low memory) — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	13843	0	246.96	133	4499	288.96	23.06	0
	Aggregated	13843	0	246.96	133	4499	288.96	23.06	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	220	300	340	370	410	910	4500
	Aggregated	180	220	300	340	370	410	910	4500

Failures Statistics

# Failures	Method	Name	Message

Figure 28: Heavy B (low memory) — Locust request statistics.

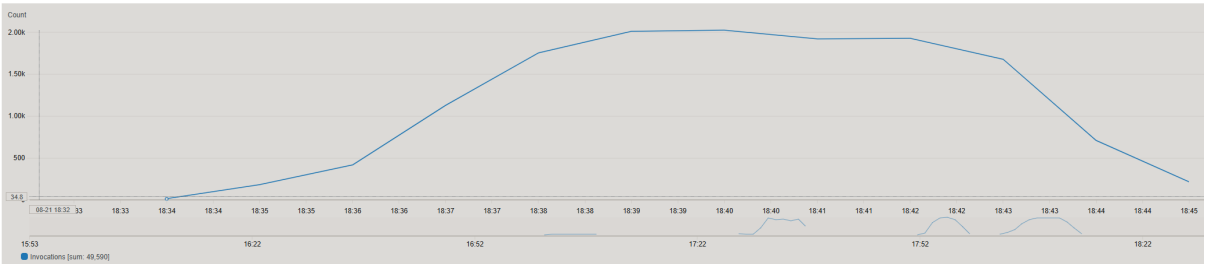

Figure 29: Heavy B (low memory) — Lambda Invocations.

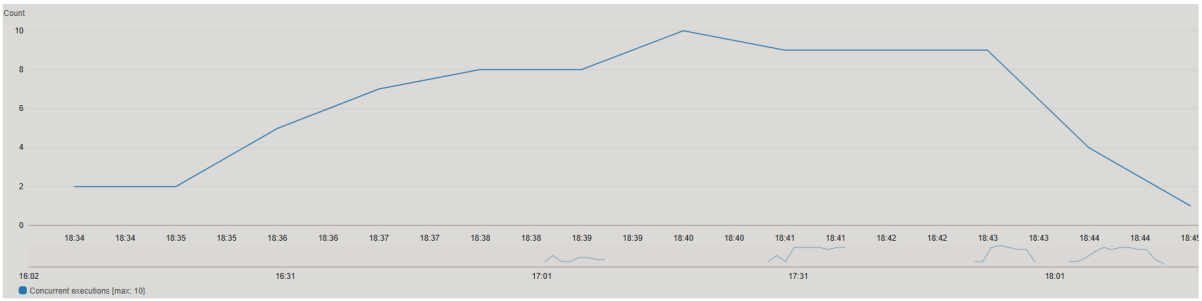

Figure 30: Heavy B (low memory) — Lambda ConcurrentExecutions.

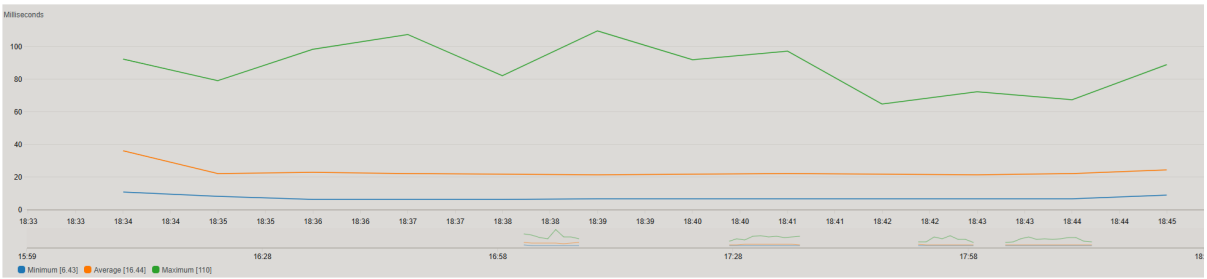

Figure 31: Heavy B (low memory) — Lambda Duration per invocation.

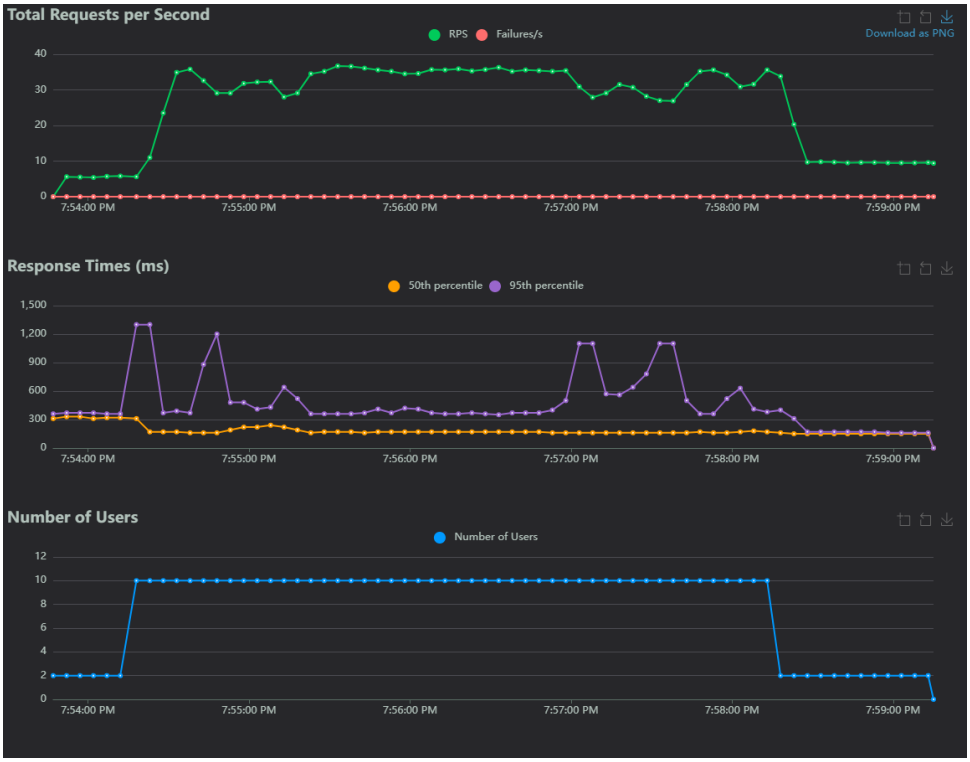

Figure 32: Heavy C — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	8637	0	241.81	135	5090	288.61	26.13	0
	Aggregated	8637	0	241.81	135	5090	288.61	26.13	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	200	280	330	370	480	980	5100
	Aggregated	170	200	280	330	370	480	980	5100

Failures Statistics

# Failures	Method	Name	Message

Figure 33: Heavy C — Locust request statistics.

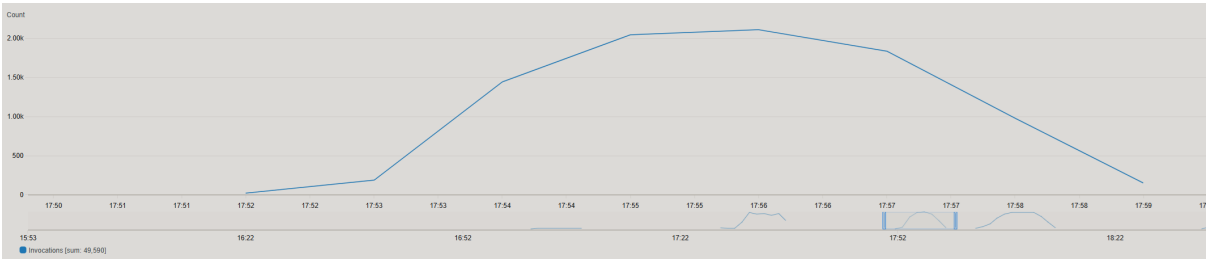

Figure 34: Heavy C — Lambda Invocations.

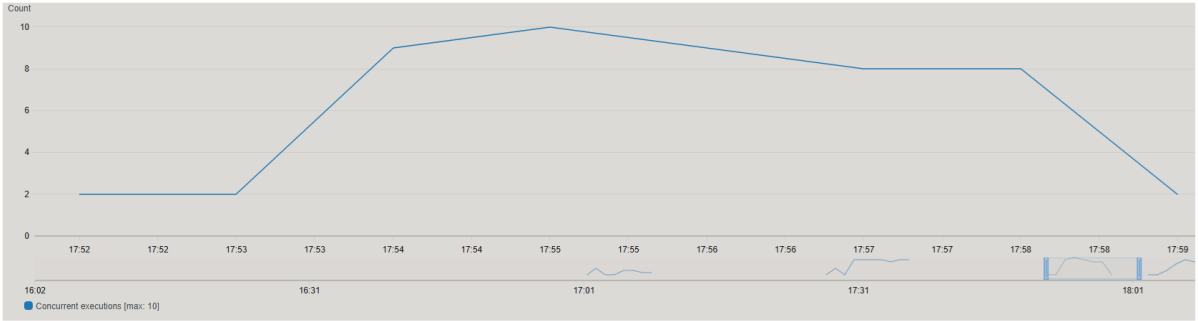

Figure 35: Heavy C — Lambda ConcurrentExecutions.

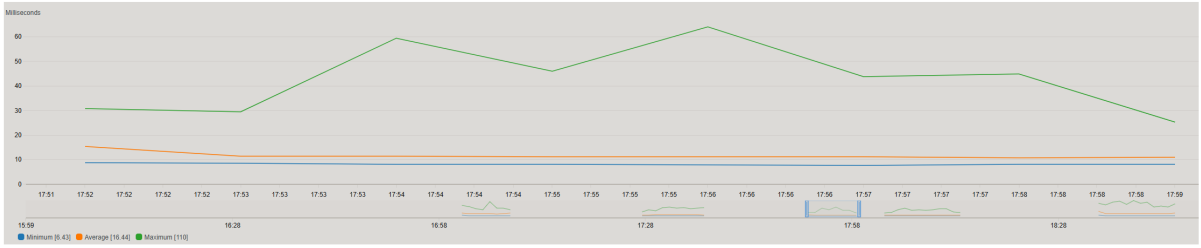

Figure 36: Heavy C — Lambda Duration per invocation.

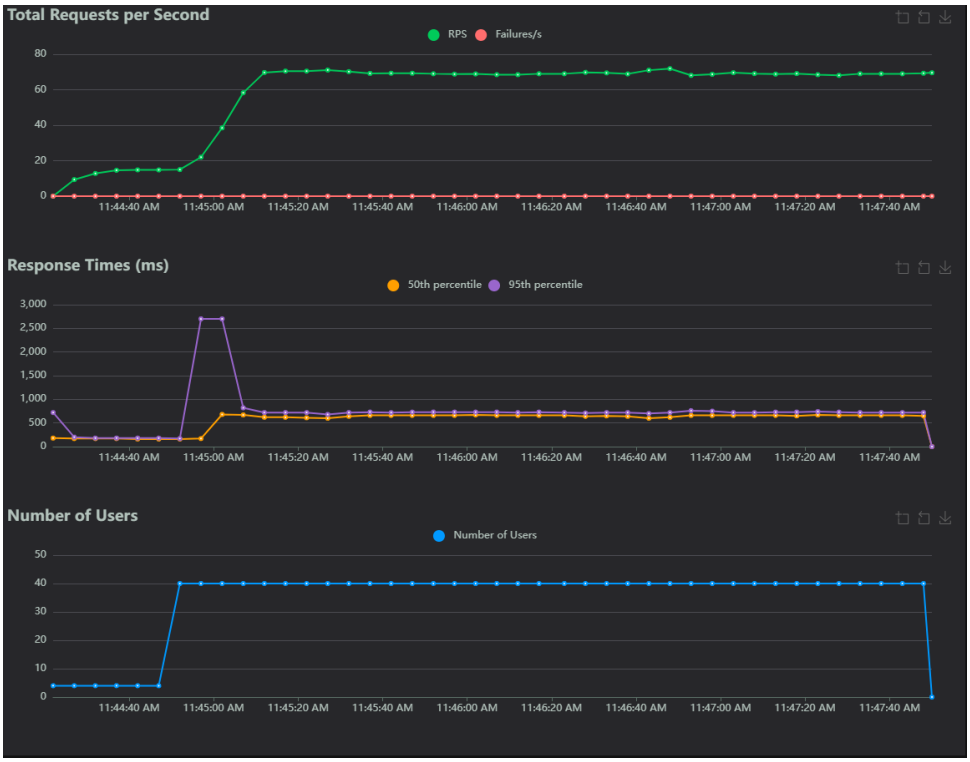

Figure 37: Stress Test — Locust RPS and latency percentiles.

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/ner	12616	0	471.98	132	6944	230.52	59.95	0
	Aggregated	12616	0	471.98	132	6944	230.52	59.95	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	640	670	680	700	720	730	850	6900
	Aggregated	640	670	680	700	720	730	850	6900

Failures Statistics

# Failures	Method	Name	Message

Figure 38: Stress Test — Locust request statistics.

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/.