SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

TicketCloud

CLOUD COMPUTING

Professor:
Emiliano Casalicchio

Students:
Andrea Di Marco
Jemuel Espiritu Fuentes
Michele Granatiero
{dimarco.1835169, fuentes.1803530, granatiero.1812623}@studenti.uniroma1.it

---

Academic Year 2022/2023

Contents

1 Introduction

For our Cloud Computing course project we designed and implemented a ticketing web application using technologies from different cloud providers such us *Amazon*, *OpenAI* and *Auth0*.

TicketCloud, with the resources provided by *Sapienza University*, leverages various services of *AWS* to build and host a robust application readily available to users. This is a key aspect for a service of this caliber since the expected traffic can be unpredictable, with uneven spikes and booking requests. Priority was also given to an accommodating user experience thanks to the use of Angular with Material design.

To make the best use of our resources we populated our website with content given by state of the art AI text and image generators such as *ChatGPT*, *Dalle2* and *NightCafe* respectively for the description of the fictional events and their cover images.

We then performed a number of different performance tests on the architecture such as load tests, endurance tests, stress tests and spike tests by using *JMeter*. We finally collected client side and server side data with the tools provided by *JMeter* and *AWS CloudWatch*.

2 Implementation

The implementation of our system is based on a multi-tier **serverless** architecture developed following the best practices advised by *AWS Well Architected* to the best of what the resourced given to us by *Sapienza University* allowed. We circumvented the aforementioned limitations by making use of cloud services outside of the AWS ecosystem such as *Auth0*.

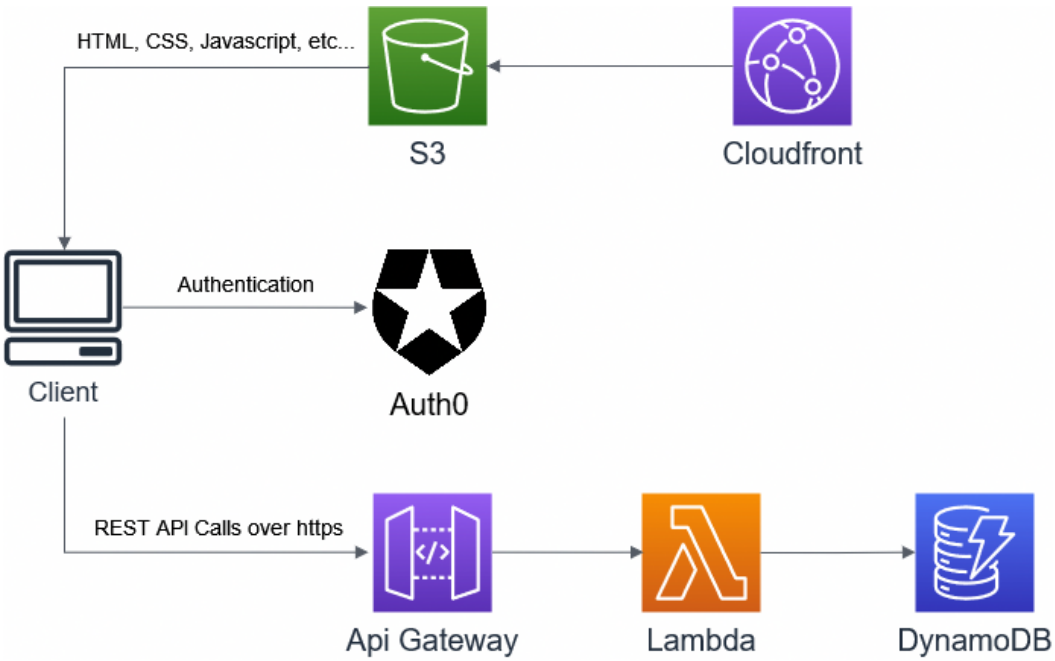

Figure 1: The architecture of TicketCloud

2.1 Presentation Tier

TicketCloud is built on the *Angular* framework and it is hosted on the AWS infrastructure using a combination of *AWS S3* and *AWS Cloudfront* as Content Delivery Network.

2.1.1 Authentication

TicketCloud allows users to register and be recognized by implementing the authentication service provided by *Auth0*. The original plan was to use *AWS Cognito* as advised by *AWS Well Architected* but the *AWS Academy Learner Lab* account provided by *Sapienza University* wouldn't allow it.

Auth0 takes care of the users' registration process by implementing connections with other platforms such as *Google*, *Facebook*, *GitHub* and more. Our users can also create brand new accounts that will then be required to verify by responding to an email sent to the provided address.

Auth0 also allows fast login for users that have previously been authenticated and till have the cookies of our website in their browser.

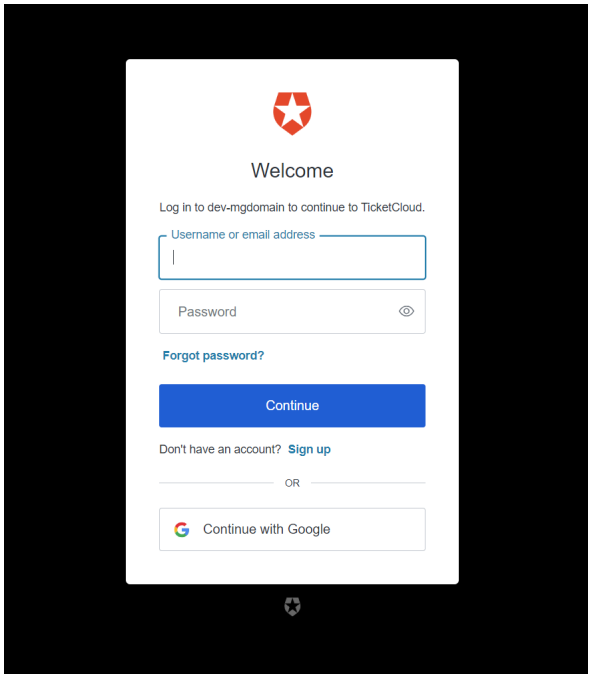

Figure 2: Authentication of website

2.1.2 Events

The landing page of TicketCloud serves the purpose of informing our users of all the available events. Said events are divided by the following categories:

- **Music**
- **Show**
- **Other**
- **Sport**

On top of the site we implemented an header containing the button to return to the dashboard and other useful functionalities such as the ability to search an event by beginning to type its name into a search bar and go to its respective page. The header also contains the authentication button thanks to which registration and log in, which make use of Auth0, can be carried out.

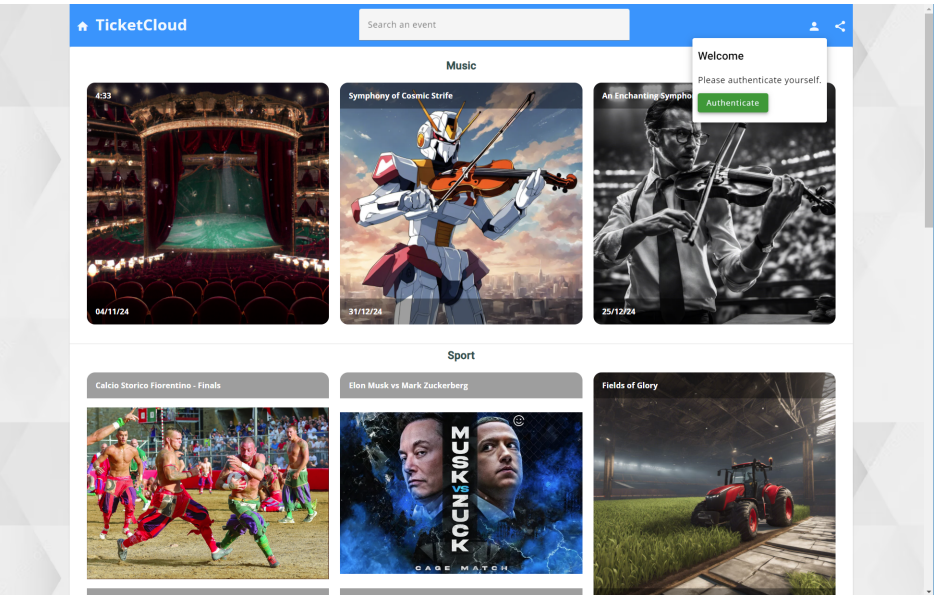

Figure 3: Dashboard of website

2.1.3 Tickets

Lastly we implemented the **event page** which contains every important information regarding the selected event. A user can read the description of the event, the date of the event and the number of available tickets.

Another important feature is the **purchase button**, which allows an **authenticated user** to book tickets for the selected event. This is composed of button and a menu that showcases the number of tickets the user wants. This button is disabled for guest users viewing the page.

After completing the procedure the user receives feedback from the system thanks to **snackbar** component.

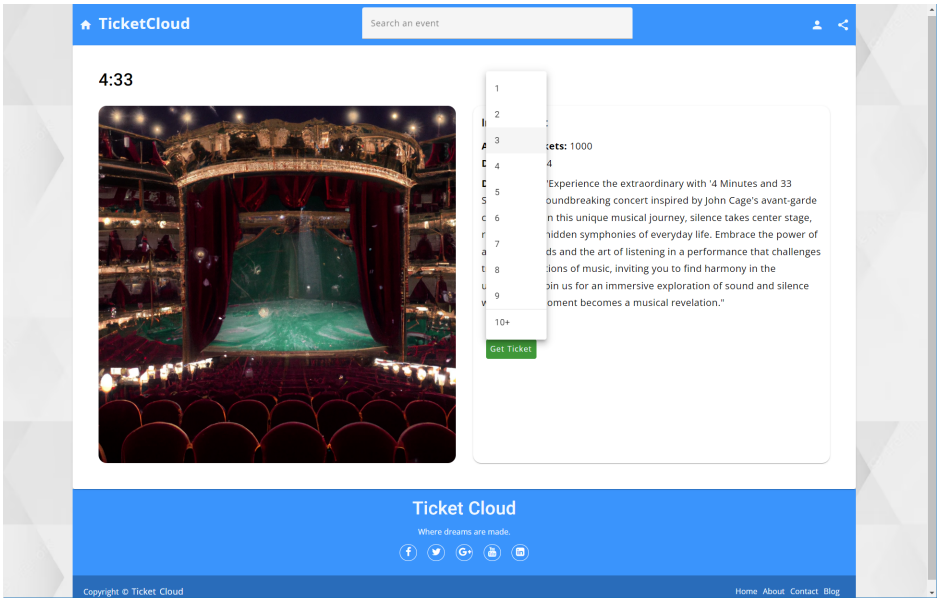

Figure 4: Event page and purchase button

2.2 Logic Tier

We decided to host the resources of the application on *AWS Simple Storage Service (S3)* and distribute it by using *AWS CloudFront*, the primary Content Delivery Network of *AWS* that also allows for SSL encryption.

To create and deploy our **RESTful APIs** we chose to use *AWS API Gateway* and linked the endpoints to our *Lambda* function thanks to Lambda Proxy Integration. We then added Cross-origin resource sharing (**CORS**) as an additional layer of security.

Our backend uses four *Lambda* functions to interact with the *DynamoDB* database:

- **getEvents**: queries the Event table and returns all the rows.
- **getEventById**: queries the Event table and returns the event with the given **id**.
- **getAvailableTickets**: queries the Event table and returns the number of available tickets for a given event
- **createTicket**: creates a specified number of entries in the Ticket table given an event **id**, a user email and the number of tickets to book while assigning a unique identifier to each entry

2.3 Data Tier

We chose to use *AWS DynamoDB* to store the data requested and created by our application. A ticketing service can be extremely heavy on its database usage as the number of users, events and tickets can be fairly high. *DynamoDB* allows us to have

a fast, flexible and reliable database for our usage that is also highly scalable. We decided to use 2 tables to store our data:

- Event
  - **id** : number
  - available_tickets : number
  - capacity : number
  - description : string
  - date : string
  - name : string
  - url : string
- Ticket
  - **id** : string
  - id_event : number (Global Secondary Index)
  - user_email : string

A relevant aspect of the Ticket table is the way the single ticket's id is created. The *Lambda* function responsible for the ticket creation uses the timestamp and a *Universally Unique Identifier* generator to create a unique id across all tickets. This allows the system to have consistent data.

Another relevant aspect of the Ticket table is the use of a *Global Secondary Index* (GSI), which allows the table to define a second key to use for a query. This is an important optimization since we can search for every ticket for a given event without otherwise scanning the entire Ticket table, allowing us to **consume less resources**.

On the user side, *Auth0* takes care of the user data in its database. The informations are stored in the following table, accessible with the *Auth0 API*:

- User
  - **email**
  - username
  - password
  - connections

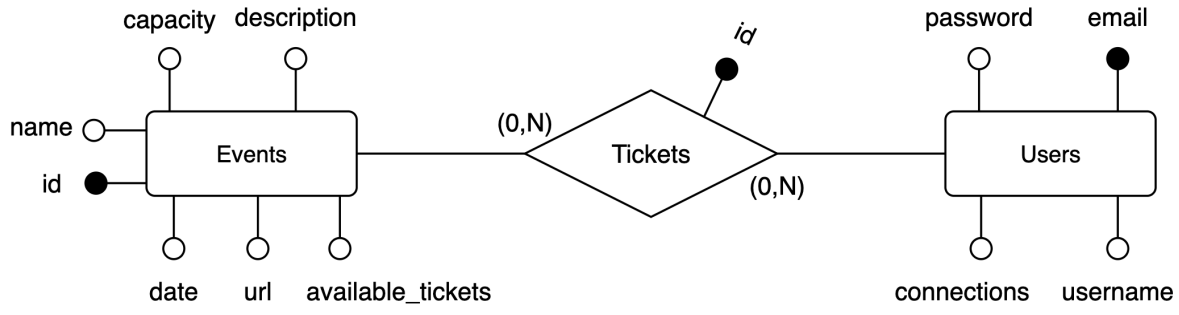

Figure 5: DB Entity-Relationship diagram.

3 Testing

In this section we show the results of our testing activities. The tool utilized is **JMeter**, an open source application for testing an analyzing the behaviour of a service. Every test is organized as follows:

1. **Warm up**: an initial stage that ensures consistency of the system
2. **Ramp up**: a stage in which the load is increased in a controlled manner
3. **Steady**: the main stage in which the load is constant representing the expected traffic
4. **Warm Down**: the final stage in which the load is gradually decreased to a minimum

Every phase is characterized by a duration and the amount of load which differ based on the test type. The duration of each phase was: 150 seconds, 600 seconds, 900 seconds and 150 seconds respectively and 30 minutes in total. The endurance test has double the time for each phase.

Figure 6: Expected user count through time.

3.1 Load Testing

We performed load testing tested the system with three different loads:

1. **Low Load** with 100 concurrent users.
2. **Average Load** with 500 concurrent users.
3. **Heavy Load** with 1000 concurrent users.

3.1.1 Low Load

As expected most tests performed quite well with a load of just 100 concurrent users. An higher average response time than other websites can be expected since *AWS* hosted TicketCloud on the *North Virginia* region, as required by the *Academy Lab* account, while our tests took place in Italy.

On the server side we can clearly spot the areas given by the different phases of users described above. Worthy of note is how the *Lambda* function presents a spike in the error rate during the Ramp-up phase probably due to provisioning issues.

Calling the *Auth0* APIs we can also see they behave as expected, by having higher error rates when *JMeter* bots are trying to authenticate, as *Auth0* recognizes the session key as not authentic, and very low error rates when requesting the login page. Response times were lower than one second throughout the test except for the booking APIs during the **ramp up** phase.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
GET Home	812	23914	15283	66195	85452	115362	210	192528	21,67%	27,7/min	91,16	1,65
GET Events	792	13180	1421	43262	63292	84160	157	84189	2,53%	27,1/min	5,57	0,07
GET img 10	743	8436	3159	21044	21049	21058	121	21065	30,42%	25,5/min	76,43	0,05
GET img 11	740	3722	21	21035	21044	21056	12	21089	13,65%	25,6/min	37,04	0,04
GET img 04	742	5618	9	21106	36117	78166	4	84162	0,67%	25,5/min	58,62	0,08
GET img 07	742	6482	8	24077	43103	84131	4	84204	1,08%	25,8/min	55,22	0,08
GET img 01	743	4169	76	21037	21045	21057	24	21089	16,02%	25,8/min	547,19	0,04
GET img 02	742	3827	31	21032	21043	21053	13	21060	13,61%	25,7/min	100,39	0,04
GET img 05	741	5999	12	22070	42125	84149	5	84189	1,62%	25,8/min	92,66	0,08
GET img 12	741	11480	3169	42060	42077	42092	127	42115	11,20%	26,0/min	57,45	0,10
GET img 09	737	6070	8	22072	42091	84120	4	84161	1,09%	25,8/min	56,04	0,08
GET img 03	733	3989	54	21036	21045	21053	17	21083	14,87%	25,6/min	255,52	0,04
GET img 06	740	11654	3075	42064	42080	42098	49	42109	13,11%	25,9/min	23,81	0,07
GET img 08	741	4095	79	21038	21044	21056	23	21079	14,84%	26,2/min	523,80	0,04
TOTAL	10489	8180	166	24064	42077	84127	4	192528	11,20%	5,9/sec	1922,71	2,46

Figure 7: Summary of low load test landing

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET Availability (1)	2955	1344	0	57504	3761,05	0,00%	1,7/sec	0,88	0,30	541,1
POST Tickets (1.2)	2935	1458	0	45187	3818,79	6,81%	1,7/sec	0,88	0,46	547,5
GET Availability (2)	2851	803	0	35422	2700,93	0,00%	1,6/sec	0,86	0,29	541,3
POST Tickets (2.2)	2834	865	0	36464	2864,18	17,64%	1,6/sec	0,86	0,45	543,5
GET Availability (3)	2810	210	0	1230	106,50	0,00%	1,6/sec	0,85	0,29	542,4
POST Tickets (3.2)	2800	290	0	3204	127,63	53,57%	1,6/sec	0,83	0,45	530,2
TOTAL	17185	836	0	57504	2744,73	12,80%	9,7/sec	5,11	2,20	541,1

Figure 8: Summary of low load test booking

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Auth0/authorize	5199	6123	0	42106	10095,26	26,18%	2,9/sec	7,67	2,90	2682,7
Auth0/authorize	5169	475	0	42095	2464,19	24,40%	2,9/sec	7,67	2,98	2680,1
Auth0/ulp/react-components/1.78.14	5149	4739	0	21077	7457,06	13,73%	2,9/sec	155,01	0,92	54457,5
Auth0/styleguide/components/1.0.8/	5125	858	0	21066	3627,14	2,40%	2,9/sec	6,62	1,07	2334,8
Auth0/ulp/react-components/1.78.14	5106	154	0	21066	1603,63	0,51%	2,9/sec	3,63	1,35	1284,2
TOTAL	25748	2481	0	42106	6492,32	13,51%	14,5/sec	179,55	9,18	12689,4

Figure 9: Summary of low load test login

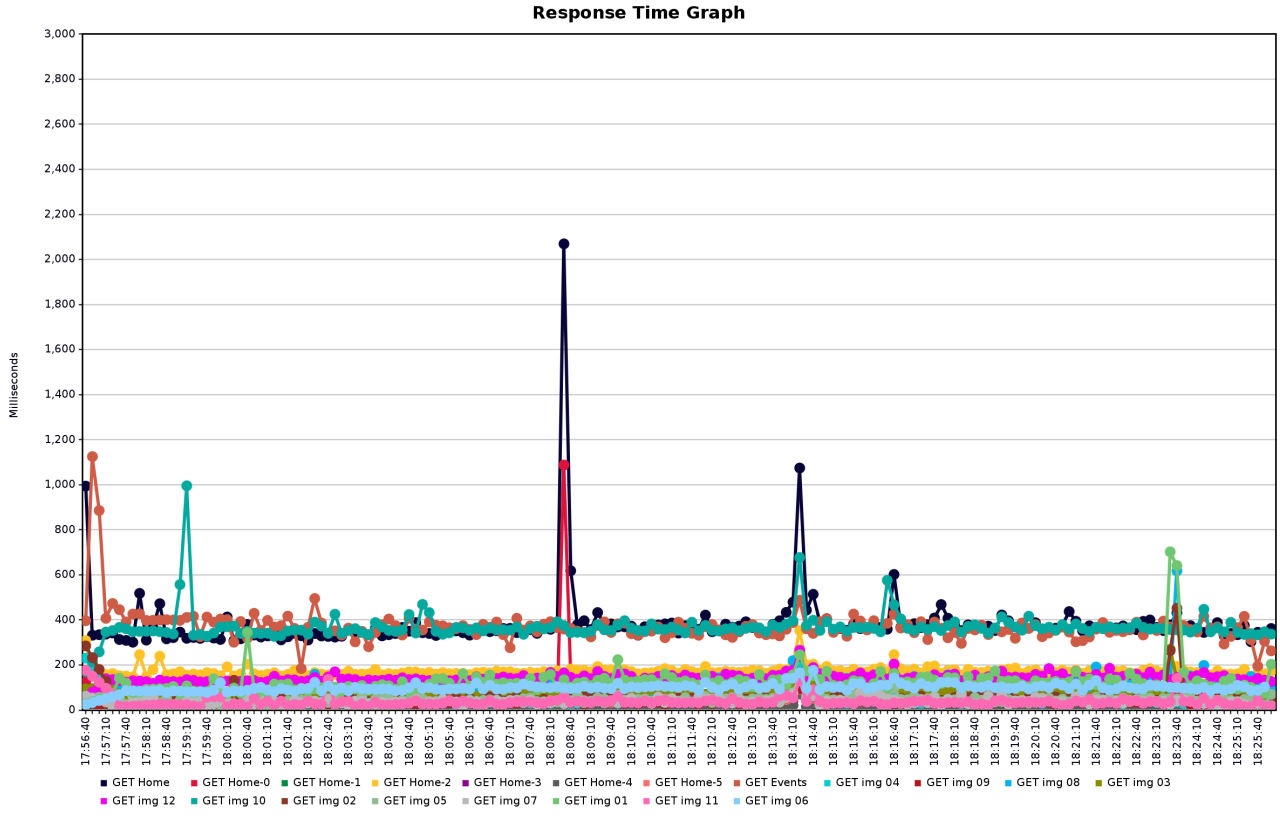

Figure 10: Response time graph of low load test landing

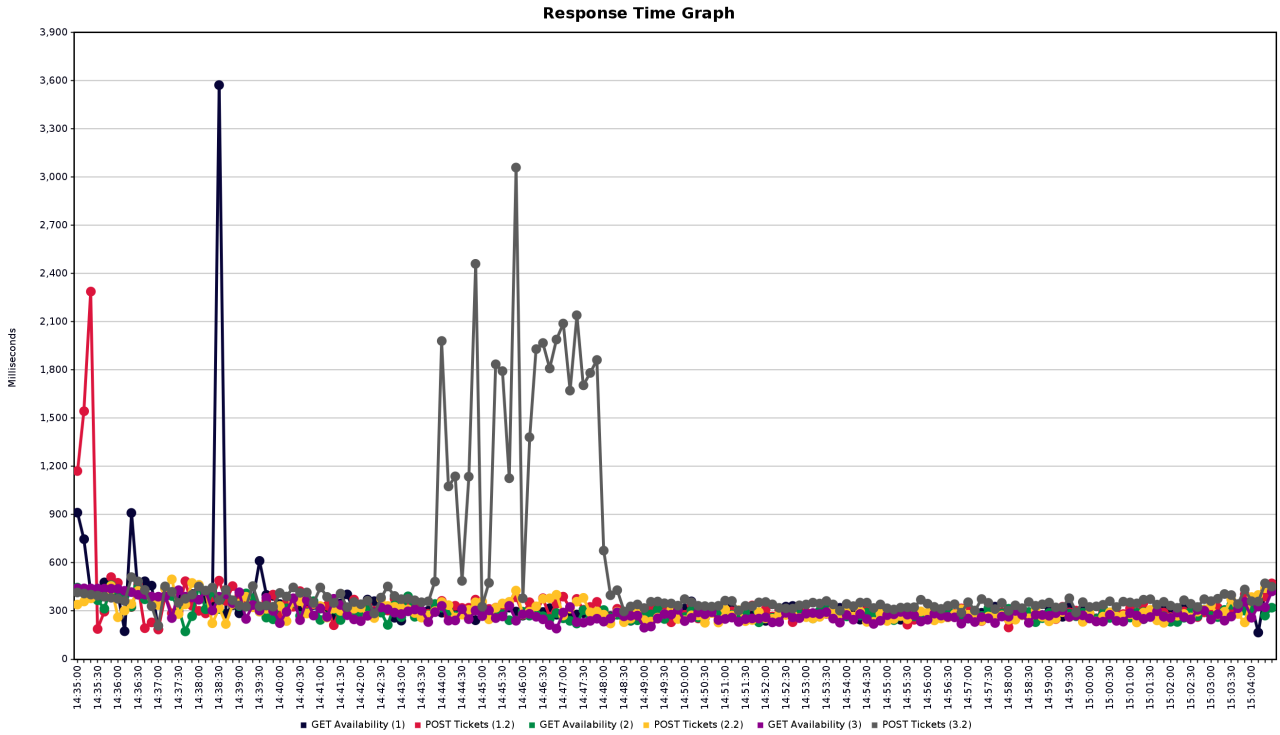

Figure 11: Response time graph of low load test booking

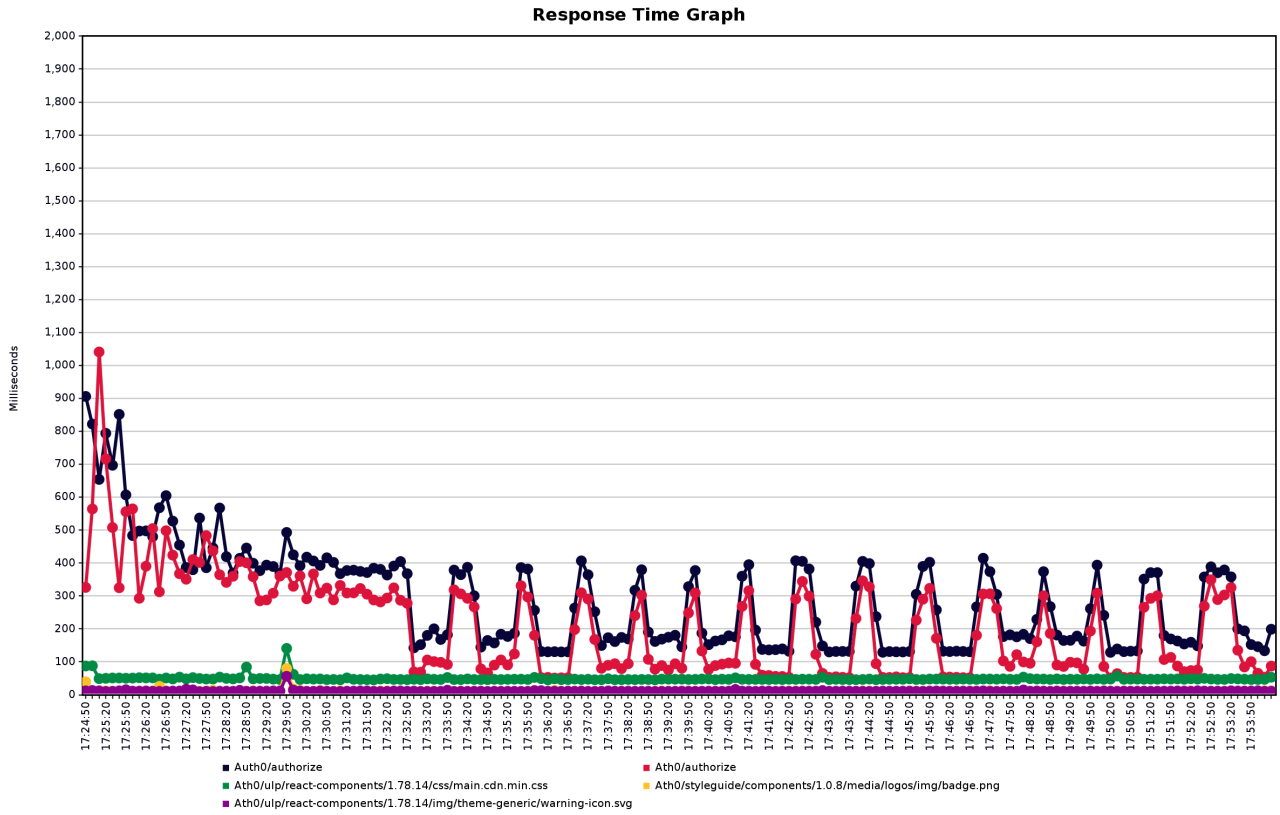

Figure 12: Response time graph of low load test login

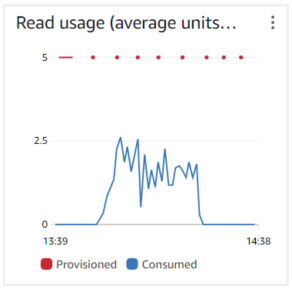

Figure 13: Read requests on Events

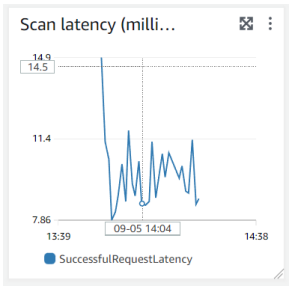

Figure 14: Latency on Events

Figure 15: Write requests on Tickets

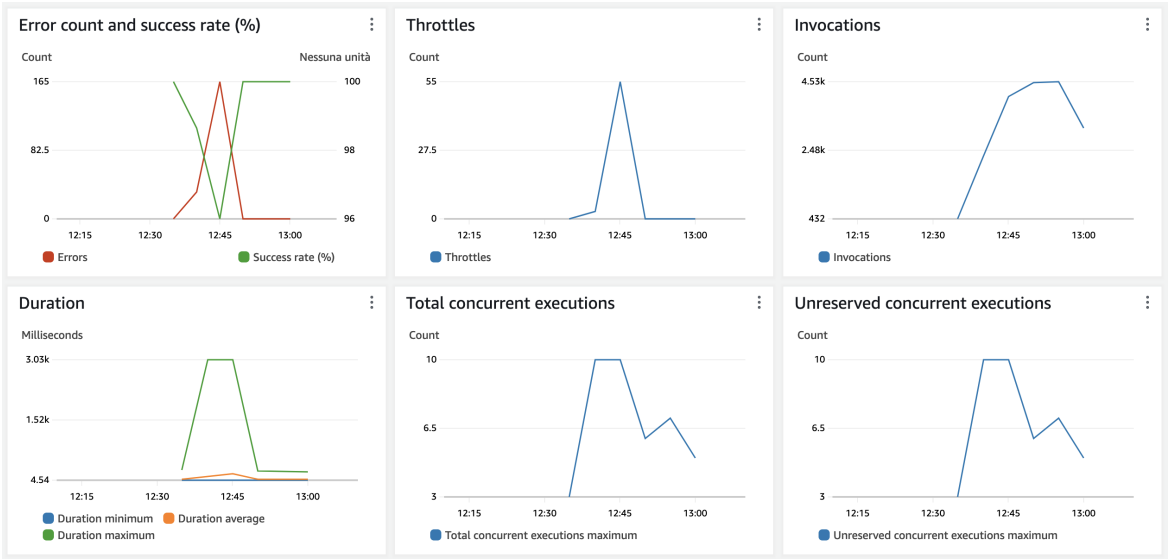

Figure 16: AWS lambda summary on low load

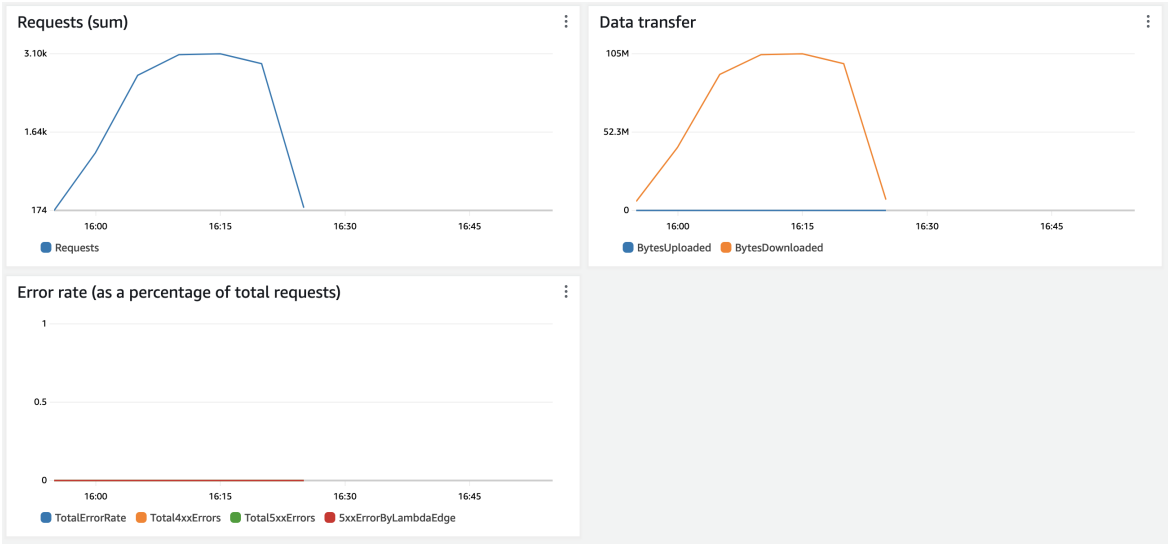

Figure 17: AWS Cloudfront on low load

3.1.2 Average Load

With a sustained load of 500 concurrent users we can see a significant increase in the response time during the **ramp up** phase for the booking procedure that it is then stabled by the provisioning of more resources by *AWS* infrastructure. The landing procedure on the other hand struggled to handle the amount of concurrent users requesting the homepage. We can also see how the increase in load did not result in greater latency for *DynamoDB* operations although it did result in higher error rates for the Lambda functions, especially during the **ramp up** and **steady** phases. This was probably due to *AWS* not provisioning resources fast enough for the Academy Lab account.

Auth0's reaction on the other hand while being always below one second of latency it is highly unstable with frequent spikes in response time. It is worth noting that every simulated users waits a randomly chosen amount of time between one and five seconds before looping back, this means that most users will wait around three seconds and then send their requests at the same time. We believe that this is the cause of said spikes.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received...	Sent KB/...
GET Events	9077	1993	219	7953	13496	21396	0	25040	64.96%	5.1/sec	31.45	0.27
GET Home	9126	2091	324	7522	11858	18959	0	36273	83.71%	5.1/sec	442.21	7.06
GET Home-0	7650	144	49	300	391	1801	10	10427	0.00%	4.3/sec	30.09	1.97
GET Home-1	7650	573	72	450	4487	11026	0	15339	19.56%	4.3/sec	10.76	1.63
GET Home-2	7650	1839	208	6132	12023	17910	0	35345	58.97%	4.3/sec	38.07	2.48
GET Home-3	3825	54	28	130	144	334	8	1597	0.00%	2.1/sec	3.01	1.02
GET Home-4	3825	61	36	135	149	328	9	1199	0.00%	2.1/sec	25.83	1.02
GET Home-5	3825	276	202	589	756	1304	26	2620	0.00%	2.1/sec	358.75	1.01
GET img 01	8815	1716	235	6048	9784	16742	0	25019	42.05%	5.0/sec	4378.22	0.36
GET img 02	8834	964	79	1845	6590	15220	0	24384	42.85%	5.0/sec	780.40	0.35
GET img 03	8833	1196	128	3563	7066	15155	0	24929	43.04%	5.0/sec	2003.91	0.35
GET img 04	8826	437	44	549	1437	11220	0	19955	37.63%	5.0/sec	439.33	0.63
GET img 05	8812	437	58	613	1098	10499	0	19837	37.19%	5.0/sec	692.47	0.63
GET img 06	8819	546	76	653	3792	9592	0	15108	58.07%	5.0/sec	141.37	0.42
GET img 07	8829	463	45	573	1567	11490	0	19962	37.20%	5.0/sec	410.91	0.63
GET img 08	8819	1621	222	5610	9424	16434	0	25034	42.25%	5.0/sec	4055.93	0.36
GET img 09	8819	471	42	557	1670	11985	0	20031	38.12%	5.0/sec	414.50	0.63
GET img 10	8819	1108	334	3224	6628	13087	0	16358	59.56%	5.0/sec	532.61	0.35
GET img 11	8814	895	56	1060	7057	15363	0	24420	42.60%	5.0/sec	290.72	0.36
GET img 12	8813	1297	124	3739	8061	20017	0	30062	61.78%	5.0/sec	293.31	0.51
TOTAL	158480	988	77	2378	6657	14934	0	36273	42.52%	88.3/sec	15165.59	21.88

Figure 18: Summary of average load test landing

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received...	Sent KB/...
GET Availability (1)	16475	330	170	394	551	3198	148	5454	72.24%	9.2/sec	4.74	1.66
GET Availability (2)	15877	304	163	386	463	3178	128	5480	72.41%	8.9/sec	4.60	1.61
GET Availability (3)	15703	294	144	367	452	3155	126	3430	72.31%	8.9/sec	4.60	1.61
POST Tickets (1.2)	16383	397	171	422	1757	3400	147	3814	74.39%	9.2/sec	4.75	2.58
POST Tickets (2.2)	15782	395	165	413	3144	3394	128	3839	74.49%	8.9/sec	4.63	2.52
POST Tickets (3.2)	15623	465	147	569	3151	3399	126	3919	77.11%	8.9/sec	4.59	2.50
TOTAL	95843	364	165	396	774	3382	126	5480	73.81%	53.5/sec	27.61	12.34

Figure 19: Summary of average load test booking

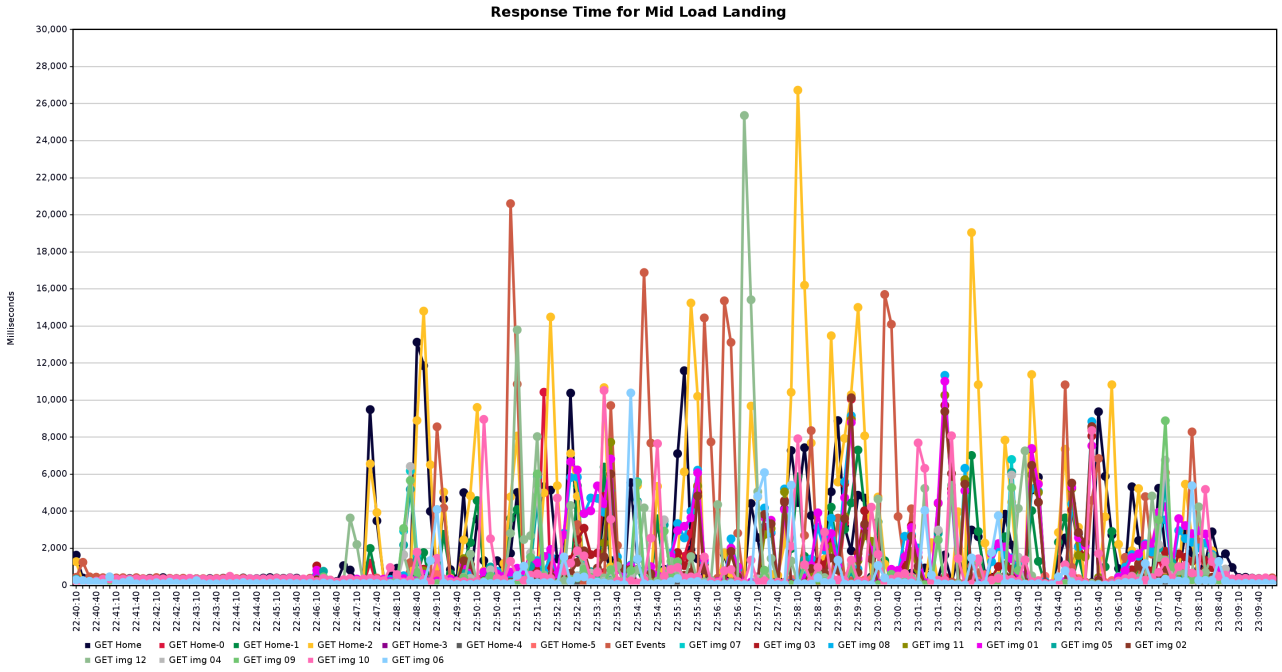

Figure 20: Response time graph of average load test landing

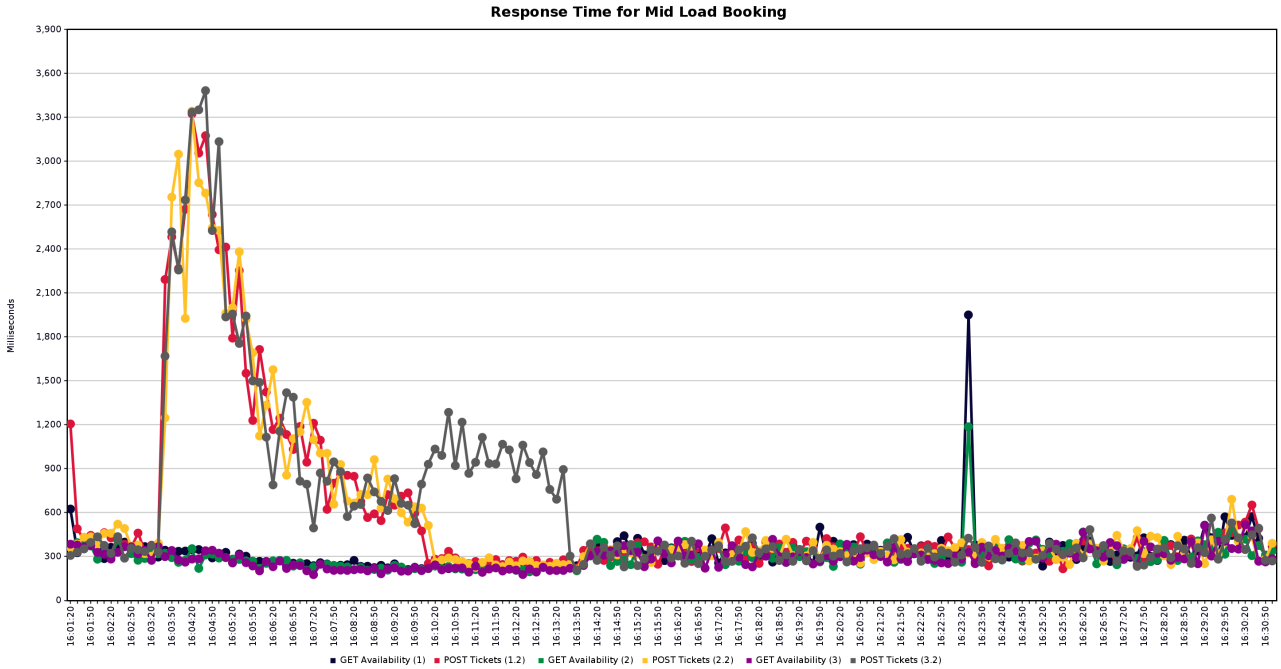

Figure 21: Response time graph of average load test booking

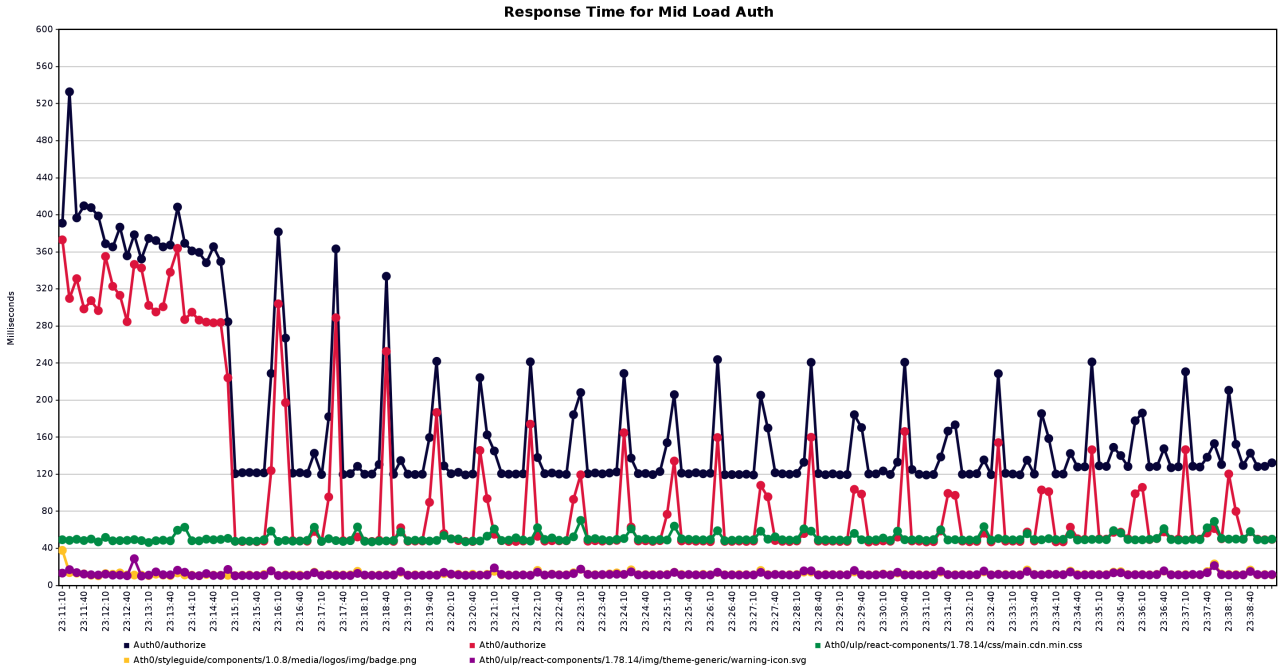

Figure 22: Response time graph of low load test login

Figure 23: Summary land AWS Dynamo events

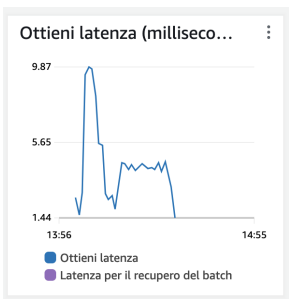

Figure 24: Summary land AWS Dynamo events

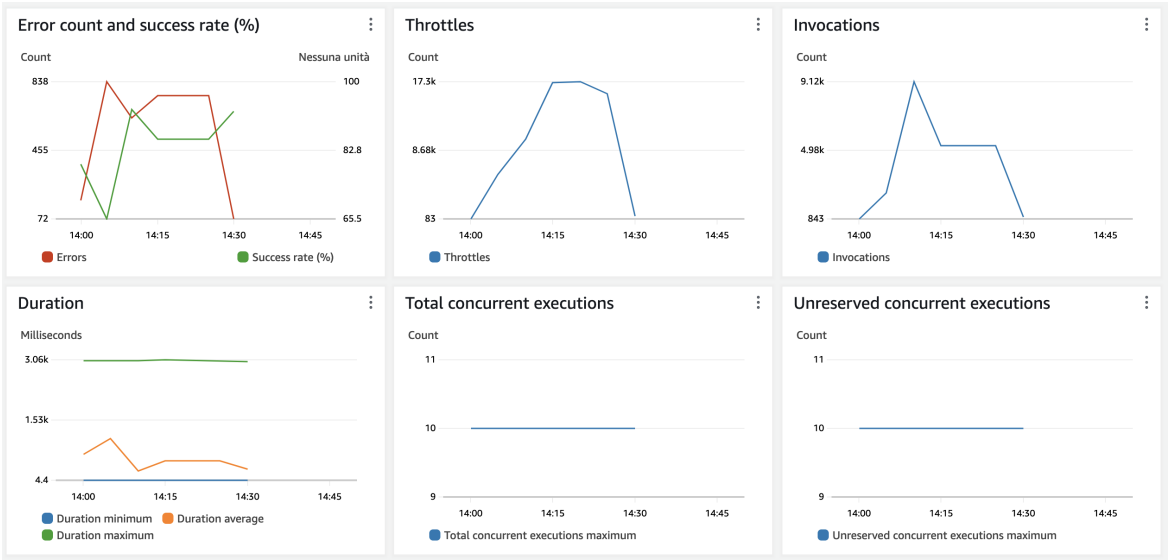

Figure 25: Summary book AWS lambda

3.1.3 Heavy Load

With a sustained load of 1000 concurrent users we can see that the landing on the site follows the same structure as the previous tests although the error rates are significantly higher. The booking requests are uneven throughout the entire test and the same can be said for the *Auth0* response times that follow the same pattern as the average load test but with even higher spikes up to three seconds.

AWS data is not dissimilar from the average load test results.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB...	Sent KB/sec
GET Availabi...	32919	338	156	402	2361	4428	0	5509	87.96%	18.4/sec	31.33	1.76
GET Availabi...	31800	325	153	386	1776	4177	0	5528	85.68%	17.9/sec	25.32	2.07
GET Availabi...	31412	309	142	375	747	3396	0	5485	82.12%	17.8/sec	17.91	2.60
GET Events	15171	2383	449	9670	12884	18747	0	27867	63.69%	4.3/sec	26.89	0.24
GET Home	15226	4604	936	14607	19609	30267	0	60613	87.43%	4.3/sec	351.41	5.48
GET Home-0	12114	436	159	557	1366	6771	10	15544	0.00%	3.4/sec	23.86	1.57
GET Home-1	12114	1002	126	2263	7434	14897	0	25021	23.60%	3.4/sec	8.79	1.22
GET Home-2	12114	4049	470	14401	18647	24697	0	58848	66.15%	3.4/sec	31.00	1.79
GET Home-3	6057	108	89	194	218	694	8	3219	0.00%	1.7/sec	2.34	0.81
GET Home-4	6057	153	126	235	421	1089	10	2960	0.00%	1.7/sec	20.39	0.81
GET Home-5	6057	679	562	1557	1864	2469	28	3251	0.00%	1.7/sec	284.71	0.80
GET img 01	14930	4447	2005	12872	15682	20432	0	28550	37.79%	4.2/sec	3982.70	0.33
GET img 02	14918	1261	538	3105	5219	10928	0	20179	38.73%	4.2/sec	699.84	0.32
GET img 03	14927	2235	1074	6432	8352	12386	0	20043	38.25%	4.2/sec	1823.40	0.32
GET img 04	14895	992	169	1525	7161	14098	0	25537	39.87%	4.2/sec	354.36	0.51
GET img 05	14899	1070	227	1719	7351	14352	0	25540	39.47%	4.2/sec	561.75	0.51
GET img 06	14901	1133	127	3591	7223	14206	0	20015	56.23%	4.2/sec	122.75	0.36
GET img 07	14930	976	166	1440	6944	14104	0	25378	39.58%	4.2/sec	335.37	0.52
GET img 08	14927	4168	1847	12140	14659	19504	0	29609	37.72%	4.2/sec	3689.29	0.32
GET img 09	14903	1010	170	1635	7053	14358	0	25564	39.12%	4.2/sec	340.08	0.52
GET img 10	14907	1818	600	5206	8137	12891	0	129540	52.92%	4.2/sec	518.24	0.34
GET img 11	14907	925	299	1589	4984	11083	0	20043	38.97%	4.2/sec	260.32	0.32
GET img 12	14910	1864	237	6189	10030	17878	0	24553	59.45%	4.2/sec	260.48	0.46
POST Ticket...	32759	394	156	512	3166	4690	0	10541	89.01%	18.3/sec	31.43	2.70
POST Ticket...	31613	382	154	442	3043	4480	0	10101	86.62%	17.8/sec	25.72	3.18
POST Ticket...	31220	392	144	433	3144	3600	0	10052	85.05%	17.8/sec	18.19	3.99
TOTAL	455587	1259	158	3583	7873	15822	0	129540	60.62%	4.2/sec	451.34	0.84

Figure 26: Summary of heavy load test landing

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received...	Sent KB/...
GET Availability (1)	32919	338	156	402	2361	4428	0	5509	87.96%	18.4/sec	31.33	1.76
GET Availability (2)	31800	325	153	386	1776	4177	0	5528	85.68%	17.9/sec	25.32	2.07
GET Availability (3)	31412	309	142	375	747	3396	0	5485	82.12%	17.8/sec	17.91	2.60
POST Tickets (1.2)	32759	394	156	512	3166	4690	0	10541	89.01%	18.3/sec	31.43	2.70
POST Tickets (2.2)	31613	382	154	442	3043	4480	0	10101	86.62%	17.8/sec	25.72	3.18
POST Tickets (3.2)	31220	392	144	433	3144	3600	0	10052	85.05%	17.8/sec	18.19	3.99
TOTAL	191723	357	154	396	2475	4260	0	10541	86.11%	106.9/sec	148.55	16.12

Figure 27: Summary of heavy load test booking

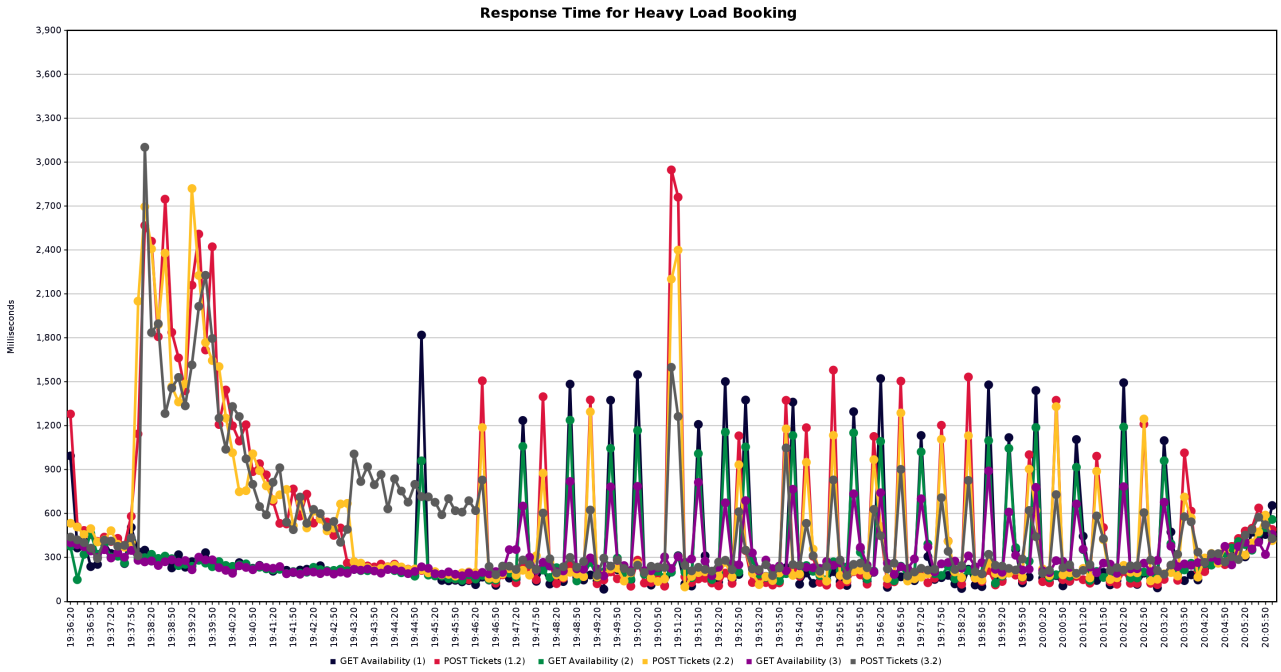

Figure 28: Response time graph of heavy load test booking

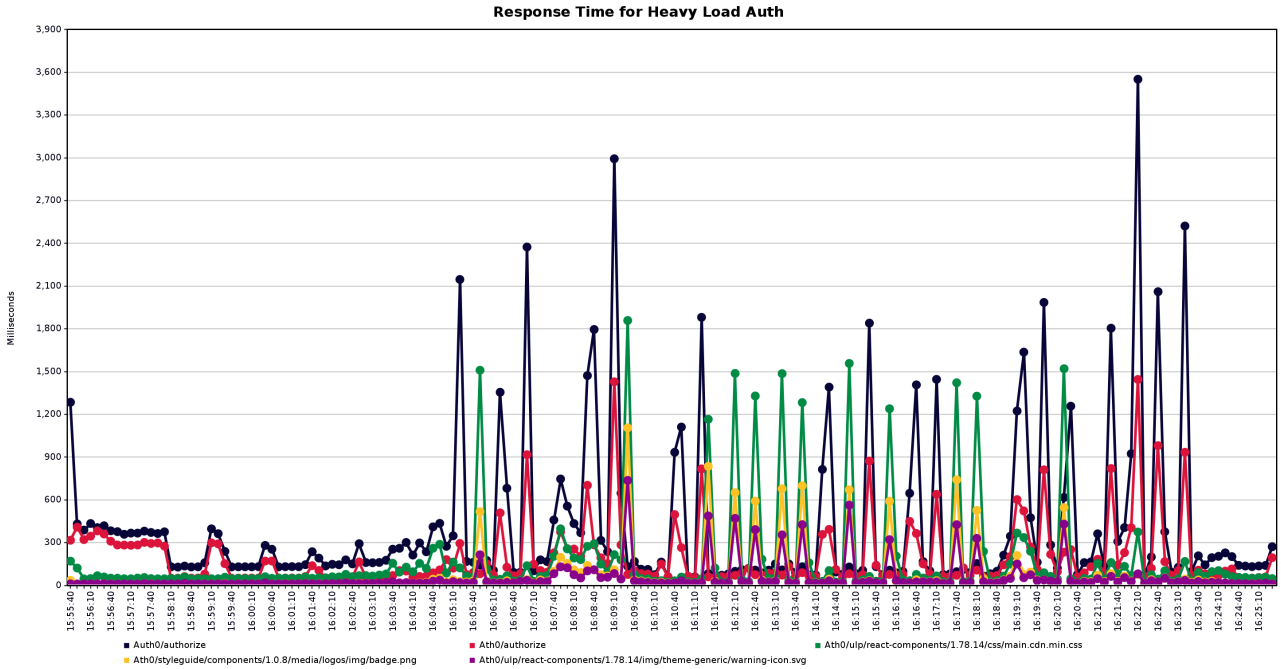

Figure 29: Response time graph of heavy load test login

3.2 Stress Testing

With a load of 5000 concurrent users response times are uniformly high and error rates are also higher as it is to be expected with a load this high for a small project. The purpose of this test were to learn the absolute limits our project could sustain.

AWS CloudFront managed to redirect landing calls efficiently and without fail even with such a stressful load. *AWS Lambda* returned a higher rate of failures again given to the concurrency limitations imposed by the *Academy Lab* account.

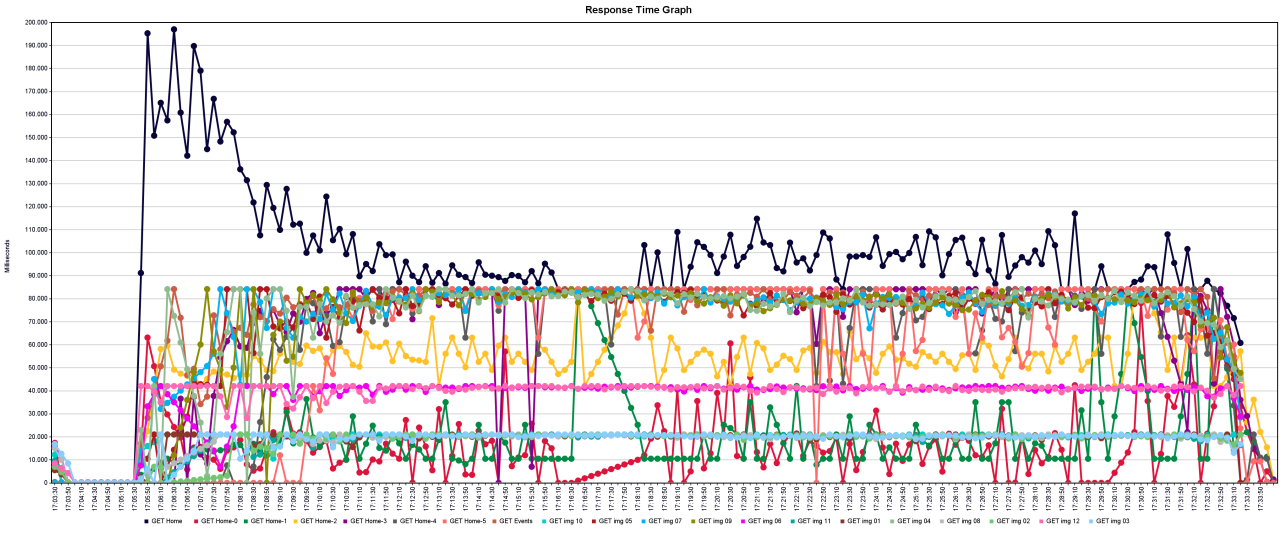

Figure 30: Response time graph of stress load test landing

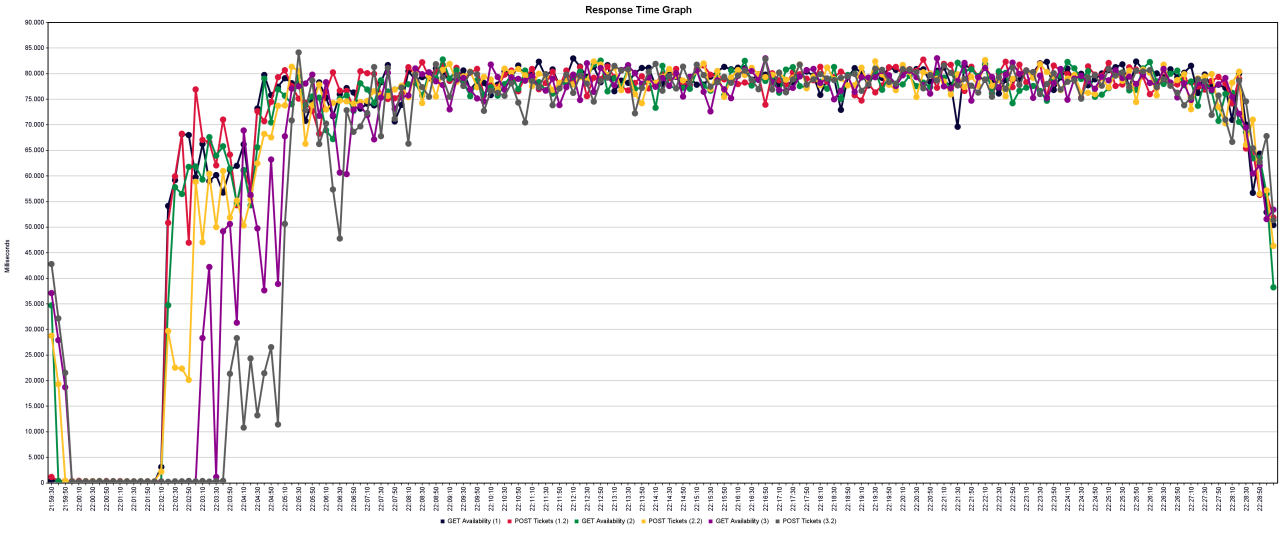

Figure 31: Response time graph of stress load test booking

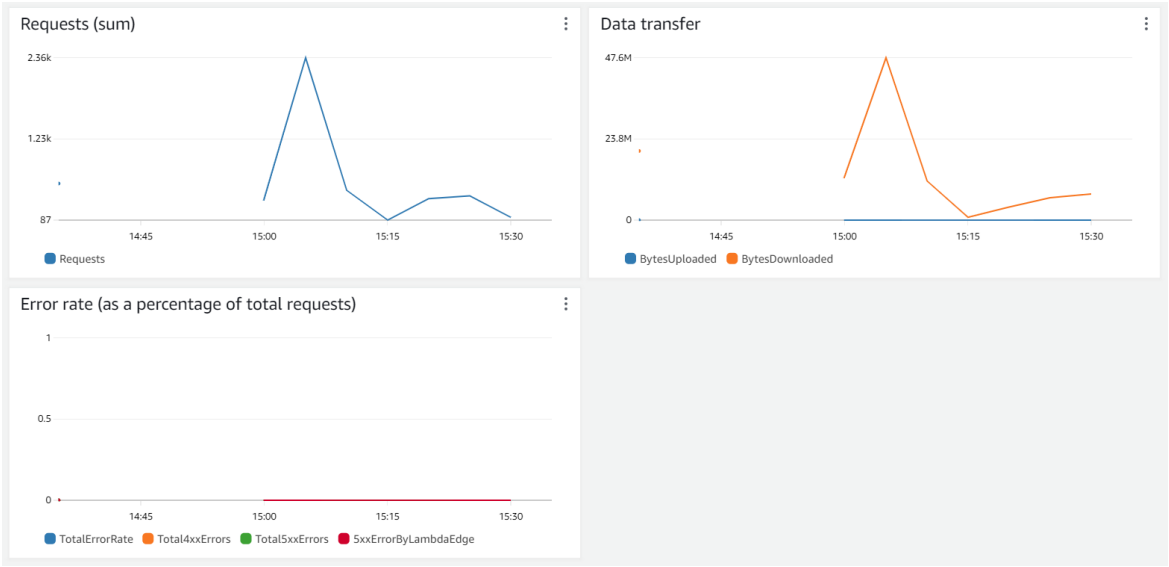

Figure 32: Summary stress AWS Cloudfront

3.3 Endurance Testing

For this test the system had to endure the expected load of 500 concurrent users for a full hour. Results show error rates and response time not too dissimilar from the average load test, with an increase during the ramp up phase and a subsequent decrease and stabilization for the other phases.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/s...	Sent KB/sec
GET Availabilit...	32772	231	161	379	387	436	147	3412	74.77%	9.2/sec	4.72	1.65
POST Tickets ...	32663	492	162	699	3174	3466	146	4168	82.21%	9.1/sec	4.72	2.57
GET Availabilit...	31706	224	157	374	381	416	127	3412	75.02%	8.9/sec	4.60	1.61
POST Tickets ...	31615	495	160	710	3170	3451	127	4483	82.66%	8.9/sec	4.59	2.50
GET Availabilit...	31540	216	138	357	364	408	127	3417	75.20%	8.9/sec	4.59	1.61
POST Tickets ...	31441	521	161	871	3164	3438	127	3744	83.93%	8.9/sec	4.59	2.50
TOTAL	191737	363	158	388	795	3386	127	4483	78.95%	53.4/sec	27.58	12.34

Figure 33: Summary of endurance load test booking

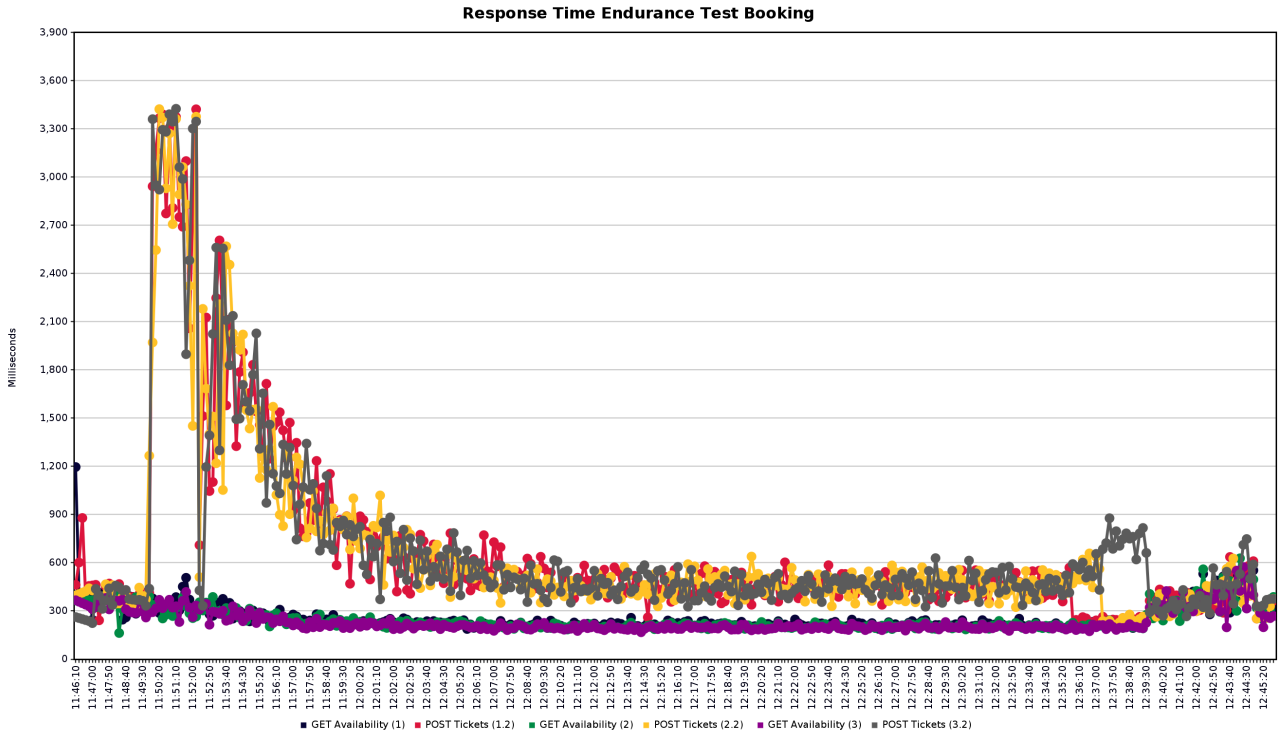

Figure 34: Response time graph of endurance load test booking

3.4 Spike Testing

For this last test the system had to endure a load of 1000 concurrent users for 10 minutes, as to be expected the *AWS* infrastructure wasn't able to allocate enough resources in such a short amount of time leading to very high response times and error rates higher than success rates for *AWS Lambda*.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received...	Sent KB/...
GET Availability (1)	2476	413	160	430	2828	4767	1	5080	76.86%	17.2/sec	25.92	1.90
GET Availability (2)	2232	328	156	383	1370	4528	1	5290	80.65%	15.8/sec	20.19	1.98
GET Availability (3)	1882	273	139	363	379	3901	1	5225	87.94%	14.7/sec	13.87	2.20
GET Events	1063	902	2	2585	7229	15646	0	20317	90.97%	7.2/sec	27.38	0.10
GET Home	1198	2334	3	12196	18416	27528	0	41825	74.04%	7.3/sec	492.37	8.58
GET Home-0	776	546	37	608	4657	10252	10	11021	0.00%	5.8/sec	40.94	2.69
GET Home-1	776	138	99	171	539	1871	0	2288	5.15%	5.8/sec	12.81	2.63
GET Home-2	775	790	213	3304	4831	5495	0	6808	19.48%	5.7/sec	43.39	4.69
GET Home-3	387	66	12	148	170	1320	8	1888	0.00%	2.9/sec	3.96	1.36
GET Home-4	387	121	15	441	820	1270	10	1425	0.00%	2.9/sec	34.41	1.36
GET Home-5	387	1274	45	2430	4002	27232	26	29988	0.00%	2.4/sec	399.42	1.12
GET img 01	768	5200	2	17522	19560	33615	0	44136	90.62%	5.3/sec	767.52	0.06
GET img 02	811	4101	4	15817	18038	19959	0	30893	84.71%	5.5/sec	238.69	0.10
GET img 03	790	4761	165	16375	18264	20009	0	29055	79.75%	5.5/sec	791.45	0.14
GET img 04	816	2143	3	8709	11600	17786	0	20036	91.05%	5.6/sec	82.56	0.10
GET img 05	807	2473	3	9936	11811	16843	0	19660	93.18%	5.6/sec	96.67	0.08
GET img 06	806	662	2	1094	3336	13239	0	15024	78.91%	5.4/sec	83.31	0.22
GET img 07	818	2414	2	9943	12746	17893	0	24705	92.67%	5.6/sec	66.38	0.08
GET img 08	766	4924	1	17561	19514	31024	0	39367	88.38%	5.4/sec	888.15	0.08
GET img 09	811	2347	3	9157	12012	18203	0	25322	93.96%	5.4/sec	56.69	0.07
GET img 10	834	2237	1	8142	10194	15706	0	27248	83.45%	5.7/sec	256.06	0.16
GET img 11	806	3387	4	14603	17435	19877	0	20138	81.02%	6.1/sec	126.68	0.14
GET img 12	802	1975	2	7187	8981	14840	0	23000	86.78%	5.7/sec	124.66	0.20
Land/authorize (a...	1106	1061	1	1161	10367	14349	0	16169	81.92%	8.3/sec	22.36	1.53
POST Tickets (1.2)	2309	305	157	482	847	4257	1	5559	81.85%	16.2/sec	23.60	2.86
POST Tickets (2.2)	2062	523	156	2737	3376	4365	1	5862	92.39%	15.2/sec	18.84	3.05
POST Tickets (3.2)	1705	514	138	2553	3366	4083	1	5077	98.89%	13.3/sec	11.79	3.19
TOTAL	29156	1398	133	4322	10269	18765	0	44136	76.06%	2.6/sec	63.21	0.52

Figure 35: Summary of spike load test landing

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received...	Sent KB/...
GET Availability (1)	2476	413	160	430	2828	4767	1	5080	76.86%	17.2/sec	25.92	1.90
GET Availability (2)	2232	328	156	383	1370	4528	1	5290	80.65%	15.8/sec	20.19	1.98
GET Availability (3)	1882	273	139	363	379	3901	1	5225	87.94%	14.7/sec	13.87	2.20
POST Tickets (1.2)	2309	305	157	482	847	4257	1	5559	81.85%	16.2/sec	23.60	2.86
POST Tickets (2.2)	2062	523	156	2737	3376	4365	1	5862	92.39%	15.2/sec	18.84	3.05
POST Tickets (3.2)	1705	514	138	2553	3366	4083	1	5077	98.89%	13.3/sec	11.79	3.19
TOTAL	12666	389	156	448	3054	4467	1	5862	85.58%	85.3/sec	106.33	13.91

Figure 36: Summary of spike load test booking

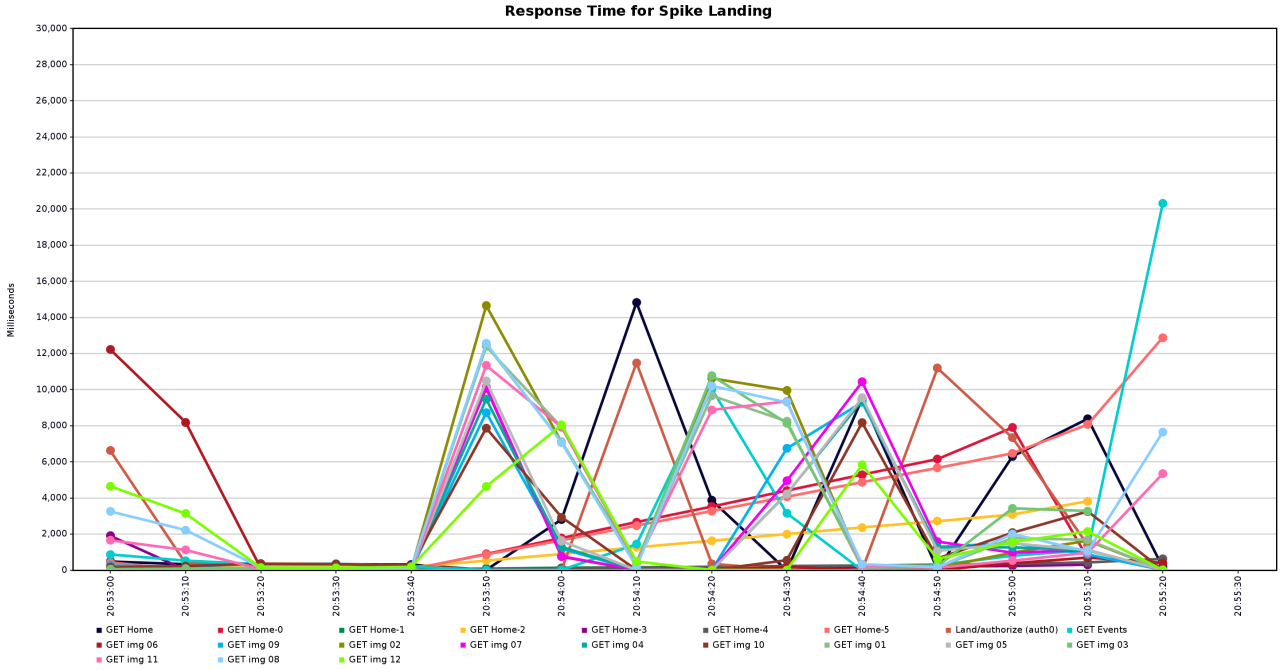

Figure 37: Response time graph of spike load test landing

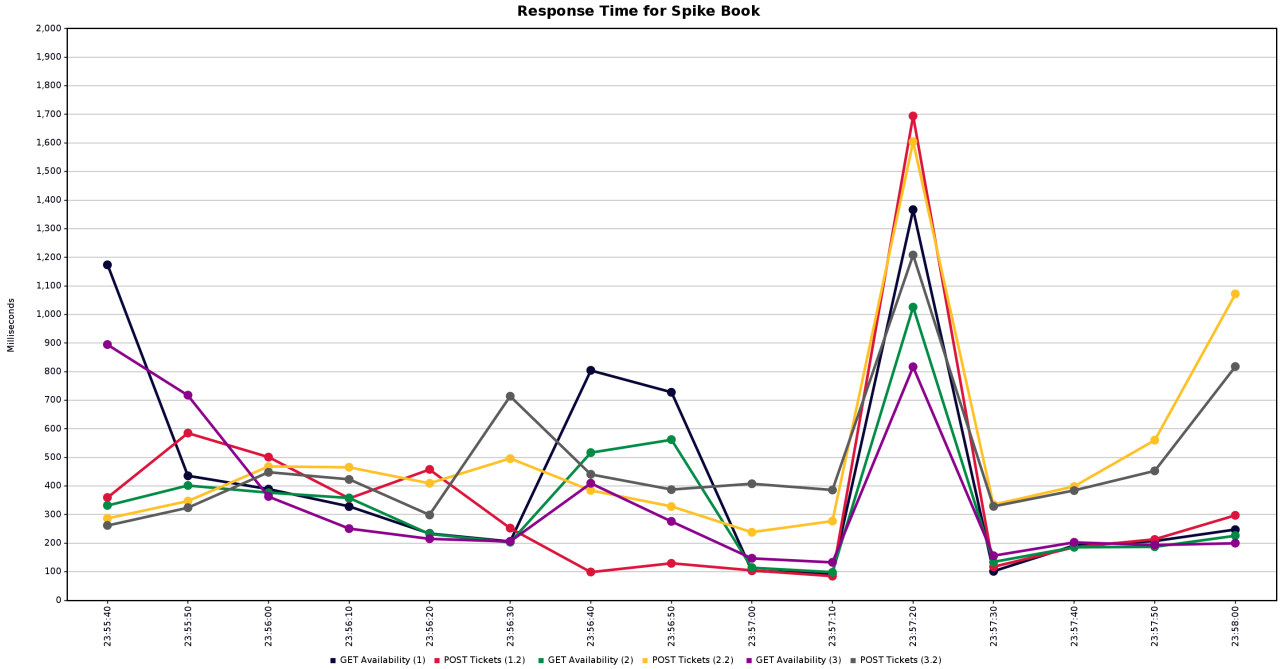

Figure 38: Response time graph of spike load test booking

Figure 39: Summary spike Dynamo tickets

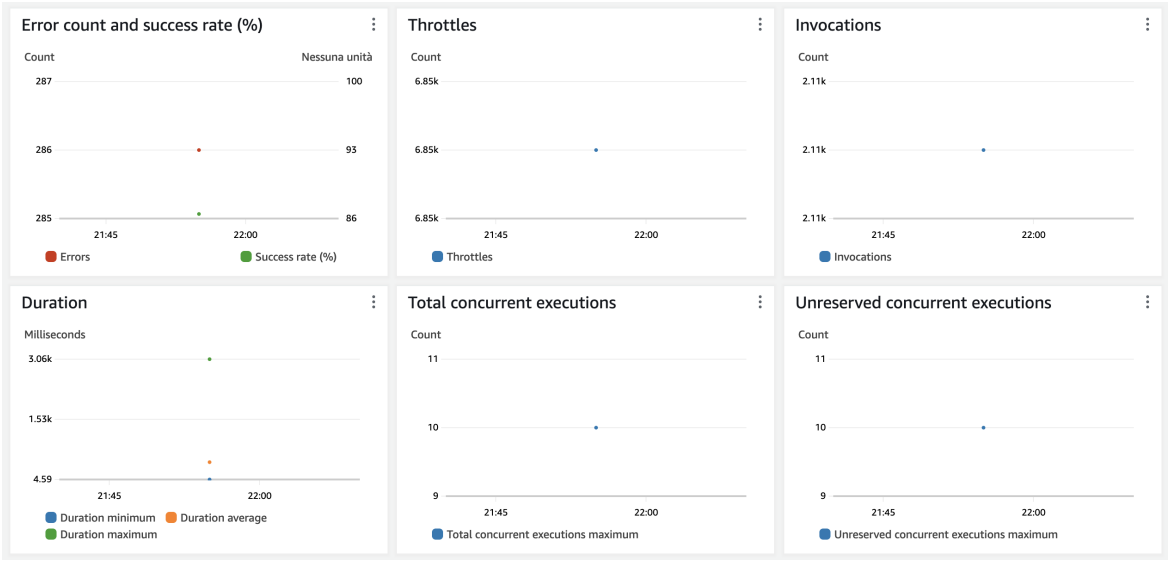

Figure 40: Summary spike AWS lambda

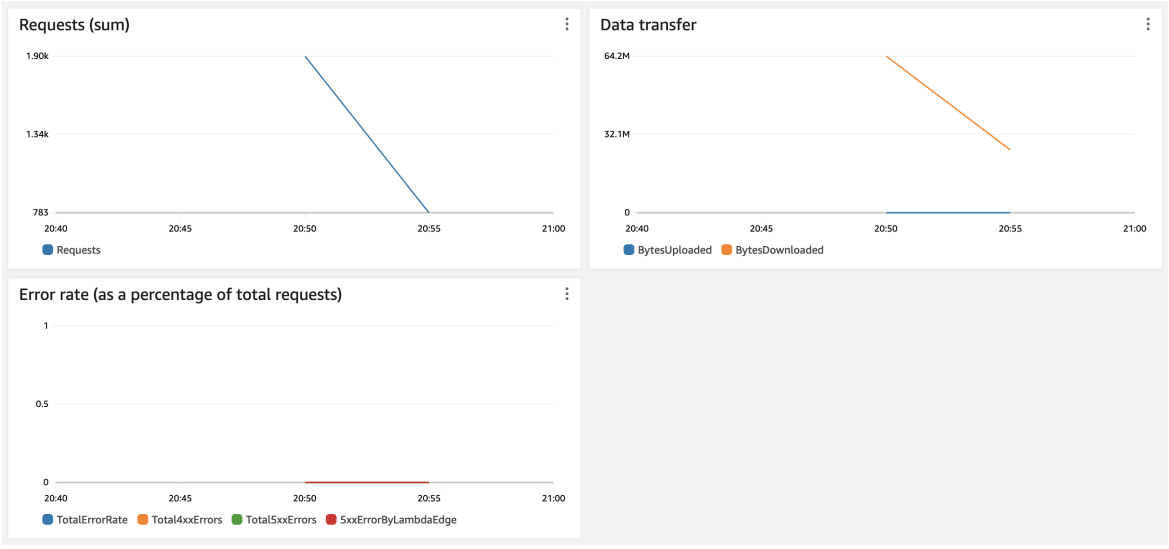

Figure 41: Summary spike AWS Cloudfront

4 Future developments

In this section we will propose some possible additions and activities to perform in the future in order to create a better platform for the user base.

First of all the limitations deriving from our *AWS Academy Lab* accounts meant that some useful services weren't available for use. Transitioning from a combo of *AWS S3* and *AWS Cloudfront* to a much simpler and more maintainable *AWS Amplify* solution would be an immediate option.
A second change in the architecture would be utilizing *AWS Cognito* instead of the external *Auth0* service for the authentication process. *Cognito* presents very useful features such as integration with other AWS services such as *Amplify*, *AWS IAM* and *AWS API Gateway*. In addition the cost of Auth0 is greater.

Improving the event page would certainly be a priority and adding a functioning payment service would be the first step. A check against malicious users is also required to offer a truly professional ticketing service.

At the moment we have not developed a comprehensive user page but our data tier has the necessary information to showcase past purchases which could be useful. Additionally a recommendation system based on previous purchases is a normal service in today's applications and one should definitely be implemented in the future.

Regarding our architecture *DynamoDB* was our immediate choice for storing every information generated by our system and we believe it should be interesting to compare the performance relative to a similar system but with a traditional relational database such as *Amazon RDS*.
A similar activity could also be performed regarding the backend component. At the moment the infrastructure is completely serverless but it could be useful to analyze the behaviour of a serverful counterpart using services such as *AWS EC2* and *AWS ELB*.

Additional testing activities should also be considered such as testing for a longer period of time than 60 minutes and collecting a larger number of samples. Using AWS to create a distributed load testing platform could be a good option. Automating application testing would help with the future scaling of the system as a whole and it would be easily achieved by using services such as *EC2*, *Lambda* and *Amplify*.