Cloud Computing

Project Report

Serverless Prodigit

Luca Bonadia, 1945660
Francesco Lucianò, 1947812

Contents

List of Figures

1 Problem Addressed

Nowadays, the pandemic contest is forcing Universities to hold lectures in blended mode, both online and in class. Instead of using a consolidated platform, many academic institutions have developed proprietary software to allow their students to book seats for the lectures. Sapienza created Prodigit, a web application that lets its students reserve seats for in-class lessons attendance.

Despite its importance, during the first semester of the 2020/2021 academic year, Prodigit had a lot of availability problems. Sure enough, students were not able to book their seats at the opening of the reservations, shortening the reservation time window and causing dissatisfaction between the users.
As a consequence of this situation, the application clients had to repeat their requests during the day in order to get a positive response from the server.

Indeed, a reason behind the lack of service continuity can be attributed to the high number of students of the Sapienza University and the traffic peaks generated by the openings of new reservations.
Moving Prodigit backend to a Cloud approach would solve this problem, providing students with a more reliable service, thus increasing Sapienza's appreciation. Moreover, since the reservation system is not a system that is used non-stop by the students, moving to a Cloud oriented solution could imply a reduction of costs: sure enough, the pay-as-you-use billing appears to be the most fitting solution for such a web application.

2 Design of the Solution

The main goal of this project is to propose an infrastructure that adaptively scales in accordance with the demand, picking the best technologies which balance performances and software manageability.

Having observed that Sapienza's infrastructure does not satisfy the students' requests, this proposal solves the availability problems by leaving them to a consolidated cloud service provider which, in this case, will be AWS.

2.1 Use cases

Given the known functionalities of the reservation service, this serverless version aims at exposing the API that realizes the most frequently used operations.
In particular the APIs are divided into two main groups according to the role of its final user: the students will have the permissions to book/unbook and search the available lectures, while the teachers will be able to add/delete and modify them.
The implemented functions are:

- **Students related**
  - A **Login** functionality that, upon receiving the student identificator from the SPID authentication service, retrieves all the lectures booked by the student
  - A functionality to **Reserve a seat for a lecture**, if at least one seat is available
  - An **Unbook lecture** functionality that deletes a previously reserved seat
  - A functionality to **Search for available lectures** filtering by course, starting and ending datetime, building and classroom code
- **Teachers related**
  - A **Login** functionality that, upon receiving the teacher identificator from the SPID authentication service, retrieves all its future lectures already inserted in the database. Moreover, for each of these lectures, the system provides the building and classroom location, the number of available and reserved seats and the email addresses of every student attending the lecture (enabling the teacher to easily contact them for every communication)
  - A functionality to **Add a new lecture** by specifying every needed parameter; predictably, the system won't allow to overlap lectures on the same class
  - A functionality to **Delete an existing lecture**. This action will trigger a notification service that sends an information email to every student that had booked a seat for that lecture
  - Lastly a **Modify an existing lecture** functionality that, similarly to the delete operation, notifies the attending students for every change on the schedule

2.2 Used Services

By analyzing the requirements, in particular availability and workload heterogeneity, the project integrates, among the AWS Services, the ones that follow:

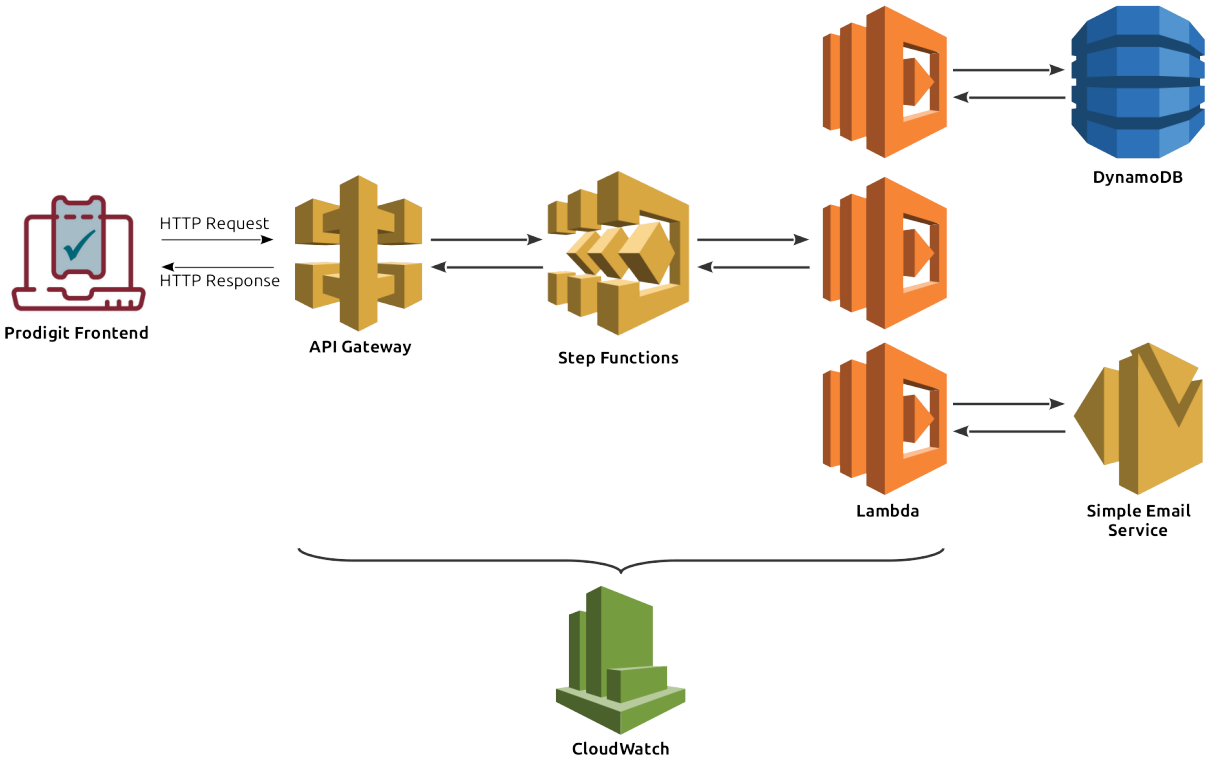

Figure 1: General scheme of the serverless Prodigit backend

2.2.1 DynamoDB

This non relational database has been picked to manage the data archiving due to its read and write fastness, its capacity of managing a large and differentiated request load, the fine grained control over resource access.
Moreover, by adding in the tables a tolerable amount of data replication, most of the functionalities does not require a table JOIN, optimizing the usage of a non relational service.

2.2.2 Lambda

The Prodigit application, due to its purposes, is characterized by long periods of low to medium load and a few request spikes (whenever a lecture becomes bookable). Thus, the pay-as-you-use billing method provided by the Lambda service perfectly fits the requirements.
Being a serverless service, the Lambdas are not tightly coupled to the physical resources, providing an exceptional horizontal scaling by creating new instances on demand.
Finally the Lambda choice is supported by the nature of our implemented functionalities that aren't CPU expensive, can be easely splitted into small sub-operations and are rather focused on read/write operations on the database.

2.2.3 Step Functions

Since to maximize the performances offered by the Lambda functions is suggested to have an higher number of simple Lambdas, rather than a few complex ones, this tool is needed to correctly orchestrate their execution.
Moreover the service greatly simplifies the developer job as the graphical view of the execution enables a greater focus on the execution logic, rather than managing the low level details of the execution chain.
Every state machine of the Serverless Prodigit backend will be created of type Express to minimize the latency and manage rapidly a large amount of workload.

2.2.4 Simple Email Service

Despite the possibility of using other non AWS-proprietary email services to notify the students, its usage is facilitated by the native integration with other AWS services. It also provides many black and white listing functionalities to have a fine grained control over which email addresses are allowed, together with a monitoring tool.

2.2.5 API Gateway

The choice of this service follows the need to expose to externals the functionalities developed, which are naturally distributed trough the REST API.
The service also provides compartmentalization, highly useful in this project to separate the services accessible by the students and the ones of the teachers.

2.2.6 Cloud Watch

To keep track of the system state the adoption of a monitoring tool was necessary.
Despite Cloud Watch offers many monitoring instruments, it was integrated in the system mainly for its detailed and partitioned logging management.

3 Implementation of the Solution

3.1 Tables Structure

To implement the persistence on DynamoDB, the data has been summarized into five different tables, described (in DynamoDB JSON syntax) as follows:

- **Studenti**: identified by the student matricola, holds the email and the list of lectures booked by the student

```
{   "matricola": { "N" },
    "email":     { "S" },
    "lezioni":   { "NS" } }
```

- **Docenti**: stores the teacher name, surname, email address (as, differently from the students, are publicly known) and a list of teached courses

```
{   "id_docente": { "N" },
    "nome":       { "S" },
    "cognome":    { "S" },
    "id_docente": { "N" },
    "corsi":      { "L":
        { "M": {"id_corso": { "N" }, "nome_corso": { "S" } } }
    } }
```

- **Edifici**: contains the physical information about the Sapienza's classrooms

```
{   "codice_edificio": { "S" },
    "indirizzo":       { "S" },
    "aule":            { "L":
        { "M": { "codice_aula": { "S" }, "posti_totali": { "N" } } }
    } }
```

- **Esami**: holds both the exam data (main and side teacher, name, cfu) and a set of its associated lectures

```
{   "id_esame":      { "N" },
    "cfu":           { "N" },
    "collaboratori": { "S"  },
    "docente":       { "S" },
    "id_referente":  { "N" },
    "lezioni":       { "NS" },
    "nome_esame":    { "S" } }
```

- **Lezioni**: contains the data of a specific lecture as the start and end timestamp, in which classroom it will take place, course id, the remaining seats and the email set of attending students

```
{   "id_lezione":      { "N" },
    "id_corso":        { "N" },
    "codice_aula":     { "S" },
    "codice_edificio": { "S" },
    "nome_esame":      { "S" },
    "inizio":          { "N" },
    "fine":            { "N" },
    "posti_liberi":    { "N" },
    "posti_totali":    { "N" },
    "studenti":        { "SS" } }
```

An analysis during the stress test phase, showed that many interactions on the lectures table require the course id (a student searching a lecture to book, a teacher looking for its lectures). For this reason the table is provided with a secondary index, the course id itself, to help DynamoDB to better partition the table content.

3.2 Students API

3.2.1 Retrieve Login data

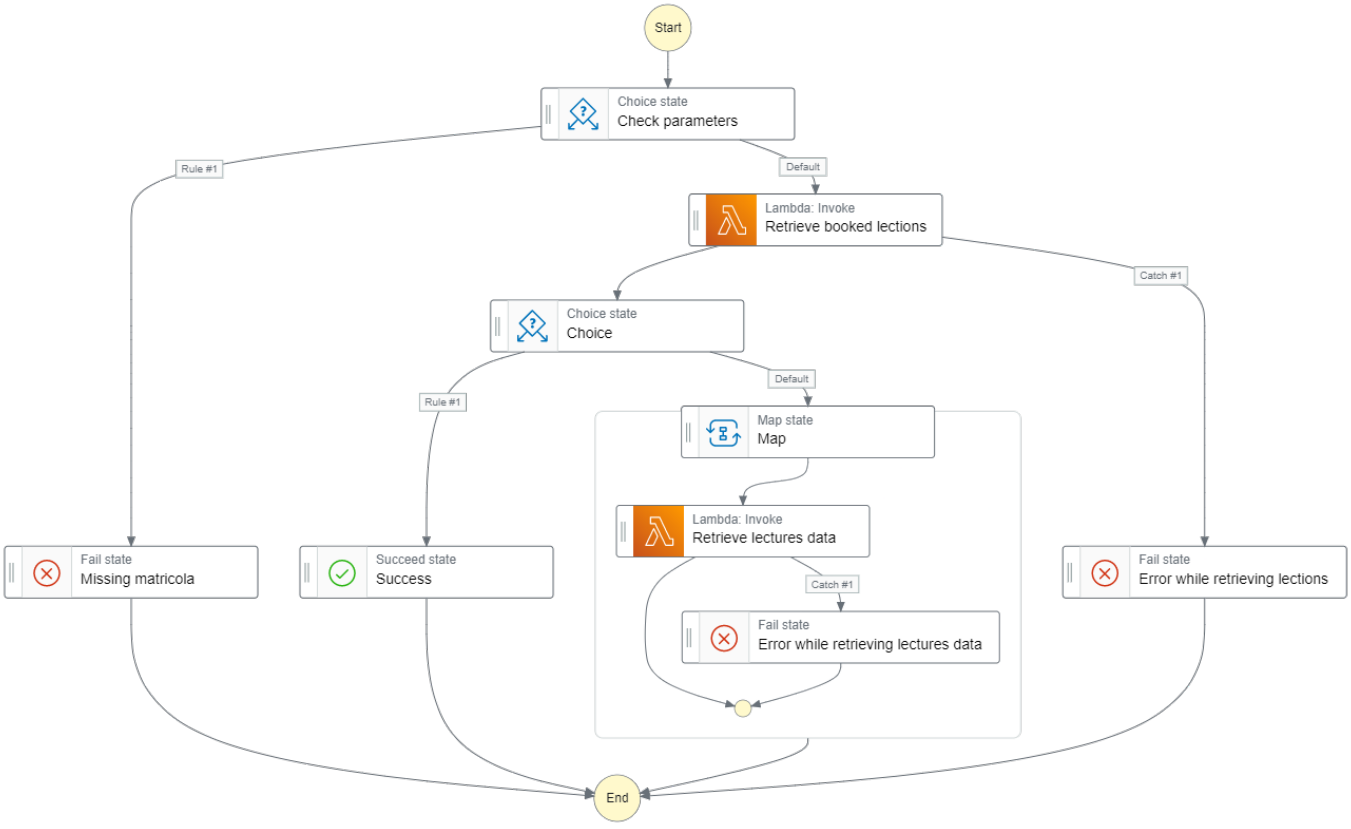

Figure 2: State Machine that, after a successful login, retrieves the booked lectures data

This is the first state machine invoked whenever a student uses the Prodigit service. As the SPID service manages the student personal data, upon logging in the state machine

task is to simply retrieve the lectures booked by the user. After a simple check on the presence and correctness of the student matricola (which is the only input needed), the step machine proceeds by retrieving every booked lecture data. By using a `Map` construct to iterate over every booked lecture, it's possible to make AWS parallelize the retrieval execution.
To enforce every student privacy the result does not contain the list of students email attending the lecture.

3.2.2 Book Lecture

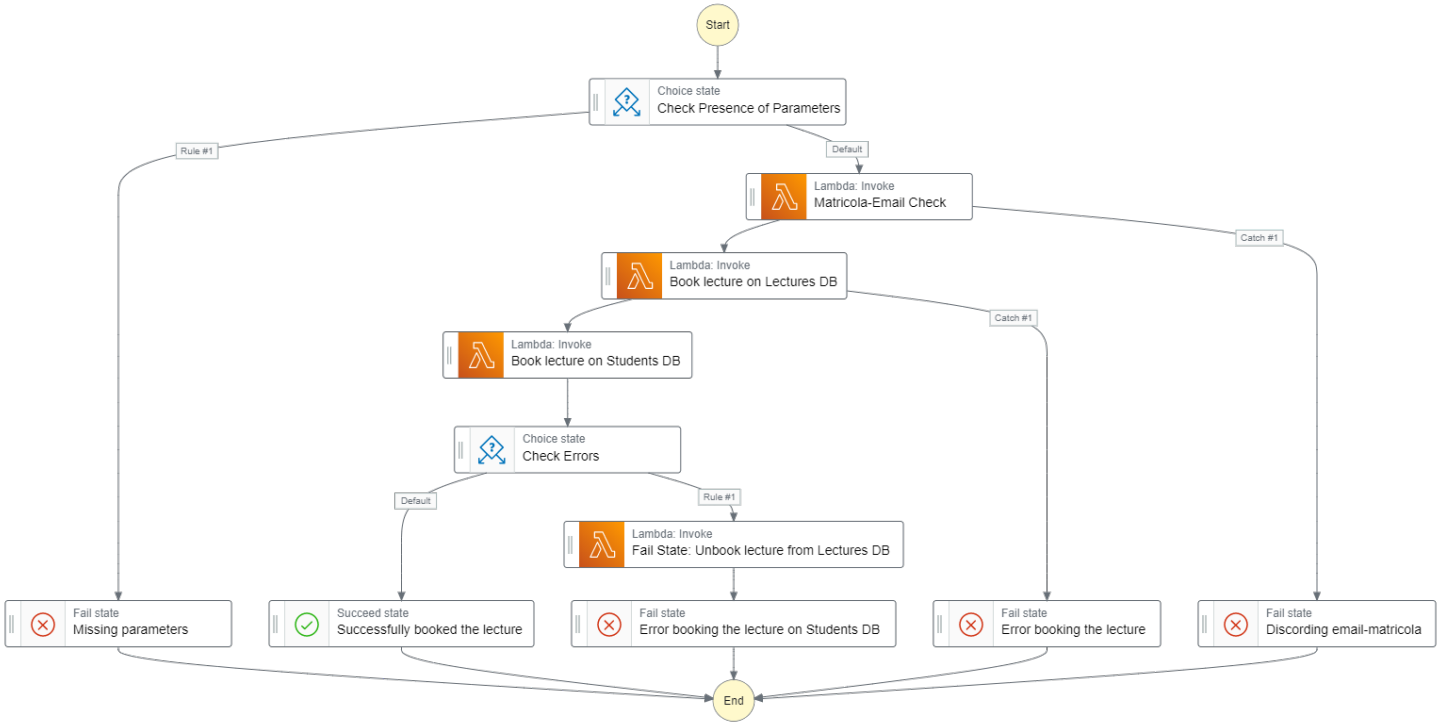

Figure 3: State Machine to allow a student to book lectures

After the parameters check, the state machine continue the execution by actually reserving the seat for the student. As data replication is needed to correctly make use of non-relational databases advantages, the lecture booking is divided into two separated steps.

In the first one the seat is reserved on the lectures table, decreasing by one the number of available seats and adding the students email to the set that contains the ones of its classmates.
In the second phase the lecture id is added to the the set of lectures booked by the student. If in this last operation an error occurs, the step function tries to re-establish a non conflicting state for the database, by reverting the first half of the booking process.

3.2.3 Unbook Lecture

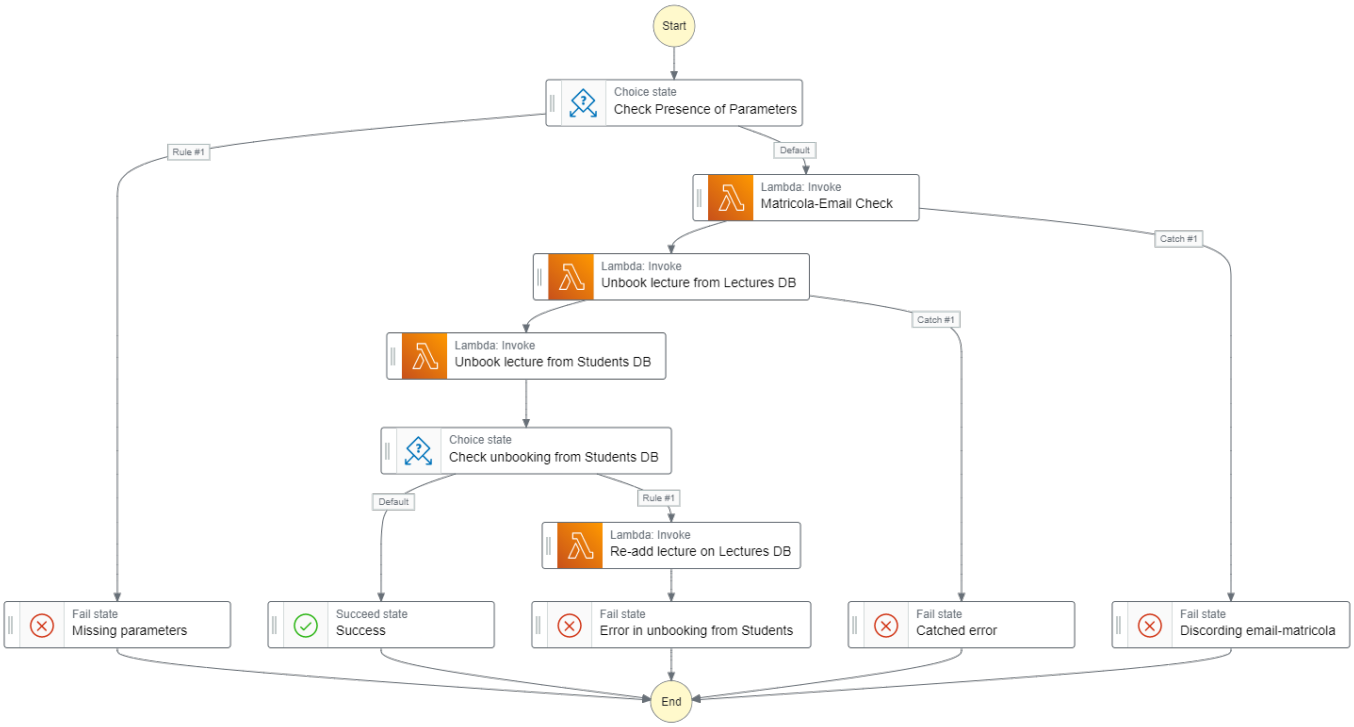

Figure 4: State Machine to allow a student to book lectures

The workflow of this state machine specularly follows the last one.
As before, after the parameter check, the system tries to remove the reserved seat from both tables. If an error is encountered the step function tries to achieve the old consistent state.

3.2.4 Search Lectures

Differently from the other functionalities, the operation of searching a lecture can be implemented using a single Lambda function, which checks the course id presence and correctness of the given parameters, after that it queries the database for any non-full lecture that satisfies the conditions provided by the user.

3.3 Teachers API

3.3.1 Retrieve Login data

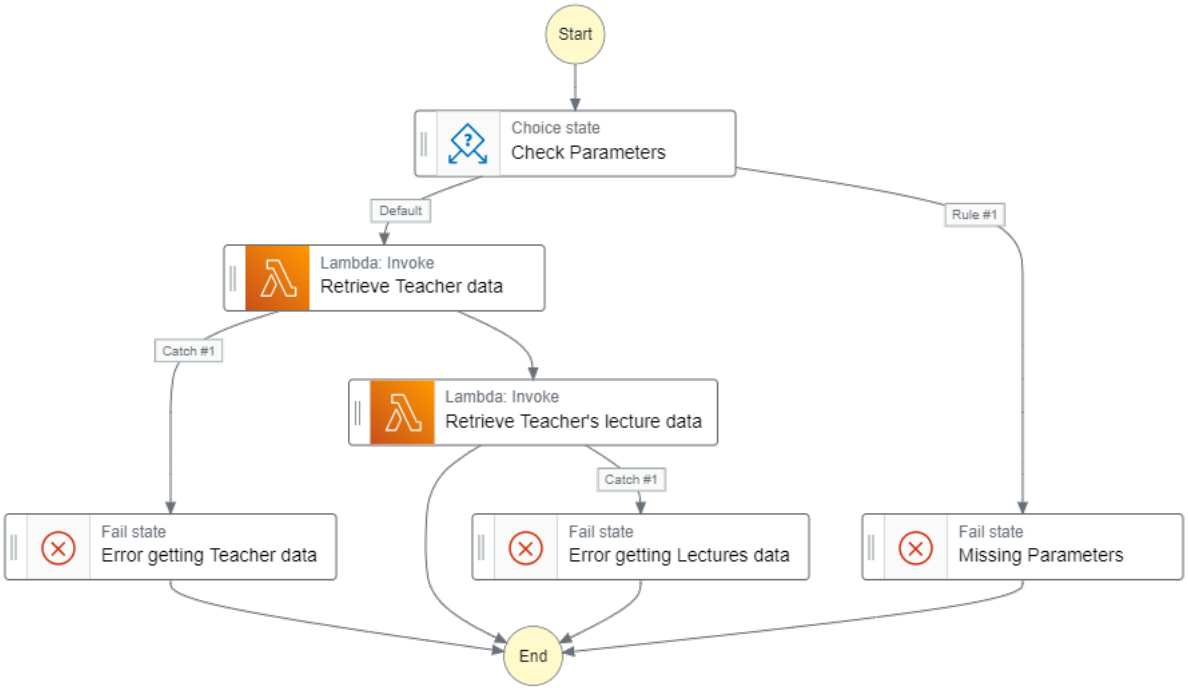

Figure 5: State Machine that, after a successful login, retrieves a teacher's data

After logging, every teacher has retrieved from the database the data of each of its course and the data of all its upcoming lectures.
The set of lectures is presented as a single array to provide the frontend developers the freedom of deciding the best order for visualization (by date, splitted by course, etc..).

3.3.2 Add Lecture

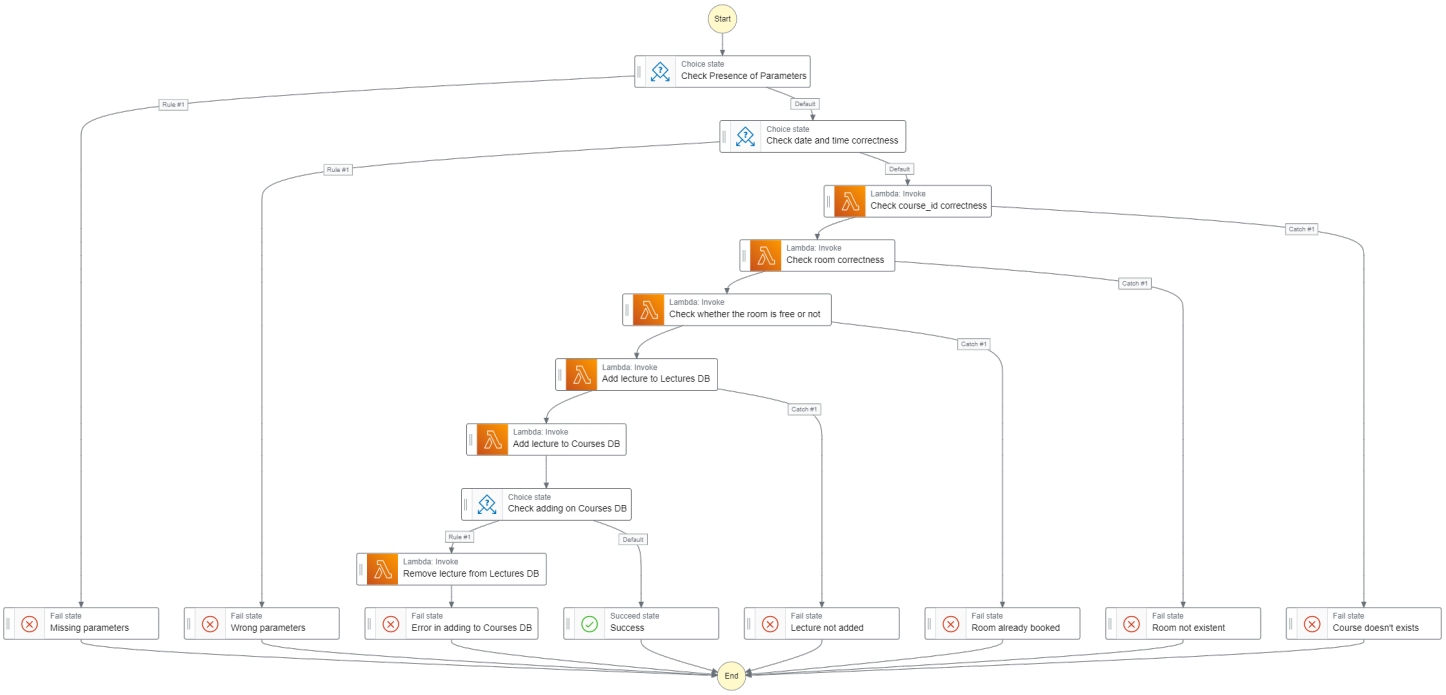

Figure 6: State Machine lets a teacher create a new lecture

With this step function a teacher can specify the parameters of a new lecture that the system will create. Due to the parameters quantity and complexity, the function checks the correctness of each of them and stops the execution as soon as possible while providing a specific error message.
As the previous state machines, a lecture is considered to exist only if a replication of its id is present on its corresponding course entry. For this reason a fallback lambda is used to maintain consistency.

3.3.3 Delete Lecture

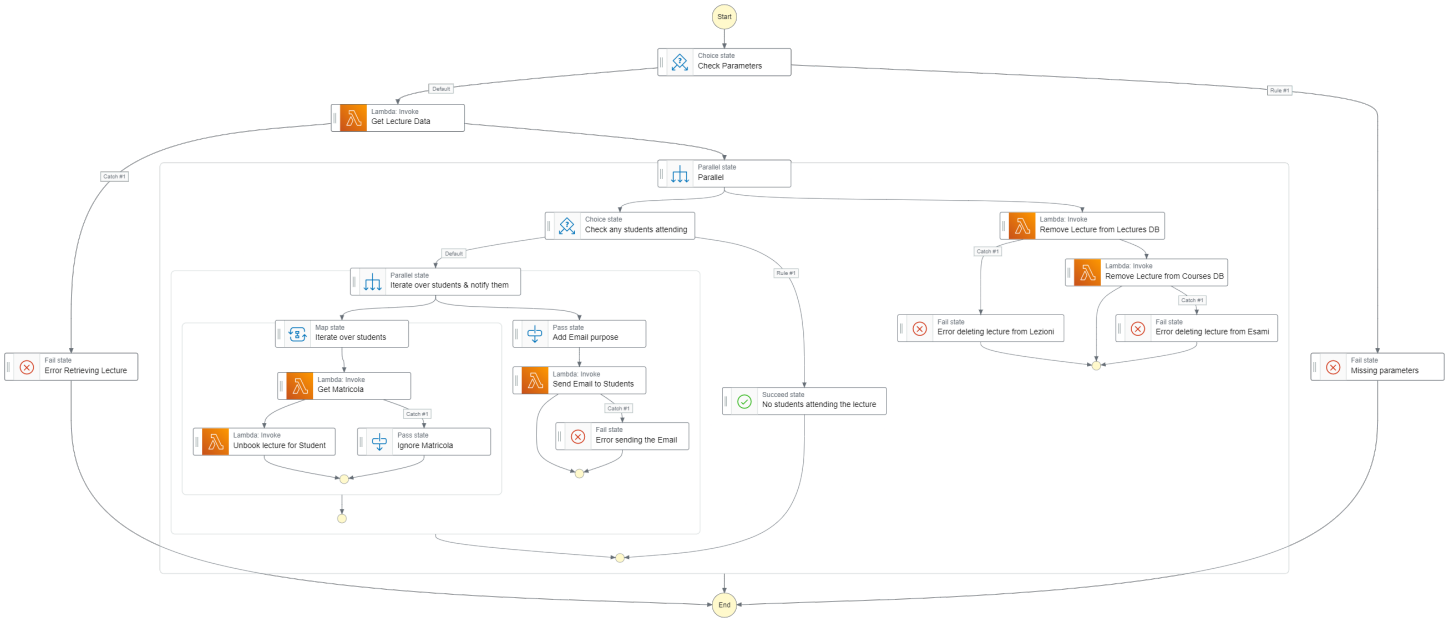

Figure 7: State Machine lets a teacher delete an existing lecture

The elimination of a lecture can be logically splitted into three different independent tasks:

- Removing the lecture from the lecture and course tables, to make it not bookable anymore
- Removing the booked lecture from the students entry to avoid to have a pointer to a non-existing lecture
- Sending an email to the interested students, notifying them of the lecture deletion

The parallelization of the tasks is mainly motivated by the fact that the first two operates with a completely different set of parameters and the email sending procedure is the slowest branch, thus a serial execution may highly impact on the user experience.

3.3.4 Edit Lecture

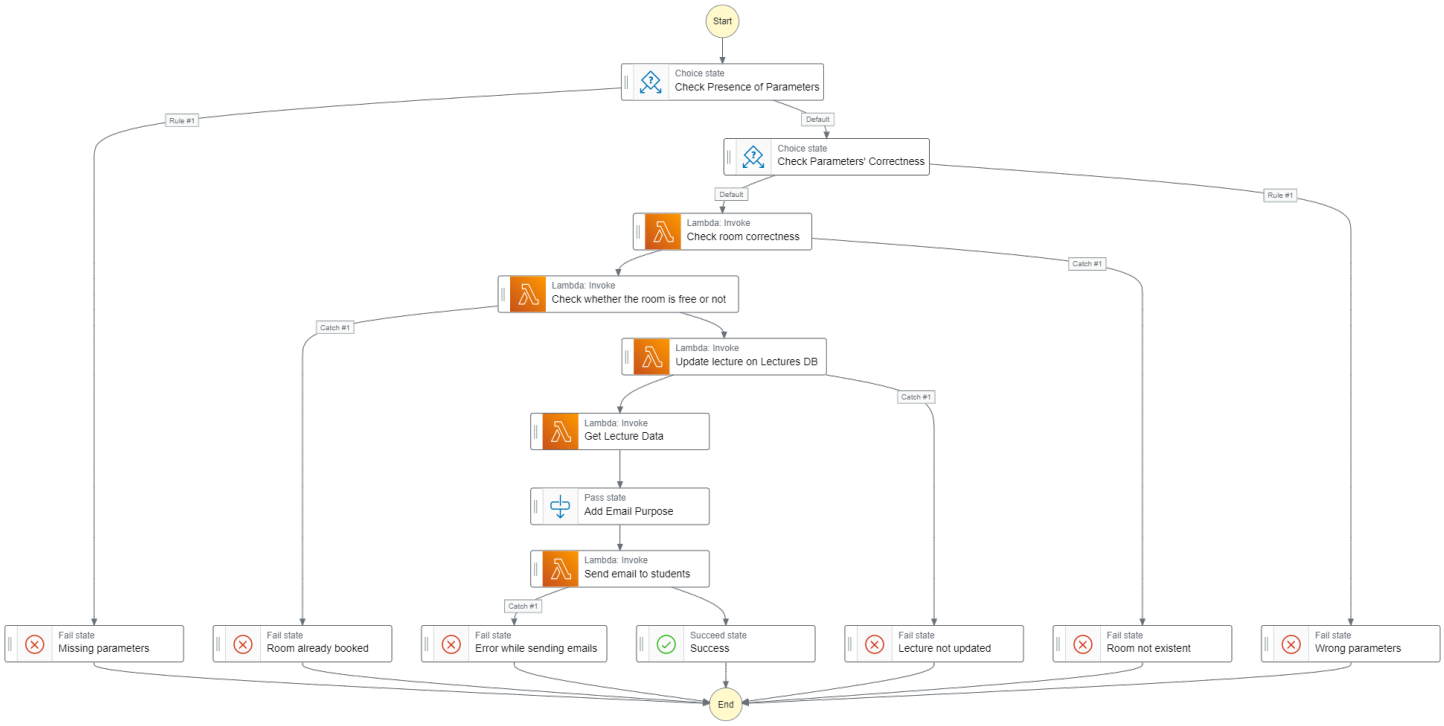

Figure 8: State Machine lets a teacher modify an existing lecture

This state machine highly resembles the add lecture state machine because it's necessary to make all the overlap checks whenever a lecture is modified. After every modification has been performed, the system notifies the students of the changing.
Differently from the delete lecture state machine, here the importance is focused more on the data consistency rather than the execution speed. A failure happening during a parallel execution may create confusion among the attending students regarding the classroom and lecture time, spreading unease toward the application.

4 Deployment of the Solution

In order to gain full control over the requests methods and parameters, Serverless Prodigit is distributed using the REST APIs provided by AWS API Gateway. The deployment is performed by creating two different API sets: one for the students and one for the teachers. Doing so makes the Gateway provide two different URLs to expose the APIs, avoiding the scenario of reaching the teachers functionalities from the students side.
Despite it is possible to make the API Gateway service to directly invoke the Step Functions, this requires the frontend developer to know the arn (Amazon Resource Name) of the Step Function itself. Such a deployment also creates a strong coupling between the API and its corresponding State Machine: the backend developer won't be able to transparently change it without distributing the new arn.
To provide a smarter solution is introduced an intermediary Lambda which is only responsible of invoking the Step Function. This removes the need of making public any used arn (neither the invoking Lambda one).
Students portal:

```
/home                => GET request to invoke "StudentLoginStateMachine"
/lectures
    /search          => POST request to invoke "SearchLectureLambda"
    /book-lecture    => POST request to invoke "BookLectureStateMachine"
    /unbook-lecture => POST request to invoke "UnbookLectureStateMachine"
```

Teachers portal:

```
/home                => GET request to invoke "StudentLoginStateMachine"
/lectures
    /add-lecture     => PUT request to invoke "AddLectureStateMachine"
    /delete-lecture => DELETE request to invoke "DeleteLectureStateMachine"
    /modify-lecture => POST request to invoke "EditLectureStateMachine"
```

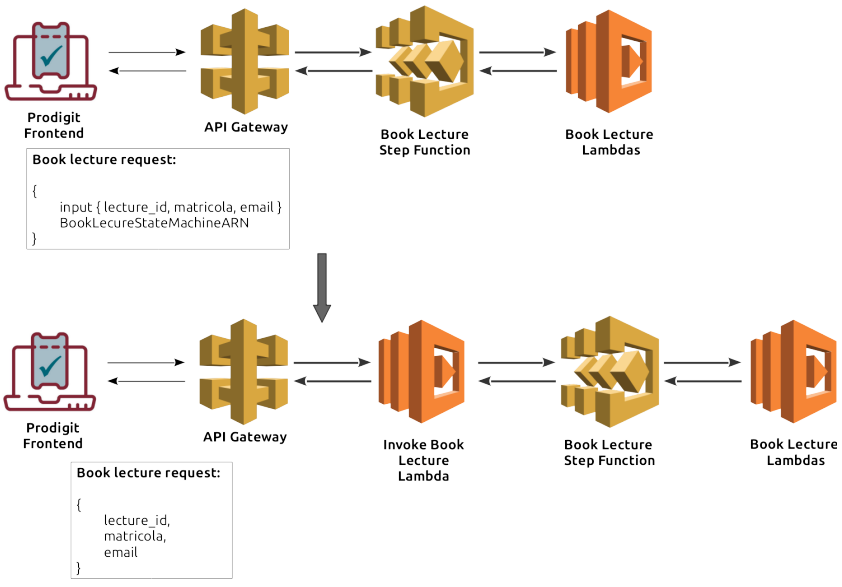

Figure 9: Difference on the API when an intermediary lambda is used to invoke the step function

5 Testing Phase

5.1 Test Design

The tests performed can be divided into three different categories: integration testing, stress testing and consistency testing.

During the integration testing phase, the execution harmony between the different Lambdas is checked, by exploring every step function execution path and verifying if the produced result is the expected one.

In the stress testing the system scalability is evaluated by providing a large number of simultaneous requests. The expected behaviour is for the backend to correctly manage the workload by scaling up to the account threshold, after that any additional request is expected to fail or experience a longer response time.

The last test category aims at breaking the consistency of the database by trying to overbook a specific lecture flooding the system with book lecture requests.
The tests will focus on the lecture booking to stress the system in the component that will be more likely to experience a "click day" type of workload.

5.2 Experimental Results

5.2.1 Integration Test

As stated before, in order to check the correct response over the different execution path, the Step Function were inkoved with purposely wrong or missing parameters.

- **Student Login:** Wrong or missing matricola
- **Book Lecture:** Wrong or missing matricola and lecture id, discording email and matricola, full lecture (0 available seats), inconsistency between lecture and student table
- **Unbook Lecture:** Wrong or missing matricola and lecture id, discording email and matricola, empty lecture (0 occupied seats), inconsistency between lecture and student table
- **Search Lecture:** Missing course id, wrong parameters types
- **Teacher Login:** Wrong or missing teacher id
- **Add Lecture:** Wrong or missing course id, classroom and building code, start and end timestamps (end greater than start), trying to add a lecture for an already occupied classroom
- **Delete Lecture:** Wrong or missing lecture id, no students attending the lecture to be deleted (do not invoke the send email lambda)

- **Edit Lecture:** Wrong or missing course id, classroom and building code, start and end timestamps (end greater than start), trying to overlap a lecture in an already occupied classroom

The system behave as expected by either answering correctly to the queries with right parameters, or by safely fail upon encountering an error.

5.2.2 Stress Test

To test the validity of this solution under a realistic scenario the system has been stressed by simulating the behaviour of the most common type of user: the student.

The "typical" usage of Prodigit from a student can be summarized into three different steps: logging in, searching for a lecture to book and booking one of the search results (if any).

The test has been performed with a growing number of user, from 500 to 2500. As, by using the standard account quota, the number of concurrent Lambda execution is limited to 1000 Lambda/s, we expect the system to not be able to tolerate a high amount of concurrency over a certain number of users.

Another limitation experienced upon using a free tier AWS account is the small amount of read and write operations per second of DynamoDB (25 WCU and 25 RCU). This soft cap greatly reduces the system scalability since, at just 500 concurrent users, 37 of them are already answered with an error message.
From the graph below it's possible to see that, despite the DynamoDB autoscaling, the system does not allocate enough resources to satisfy the needs of the incoming requests. In particular the graph shows that the provisioned capacity does not exceed the one given by the free tier accounts (25 RCU).

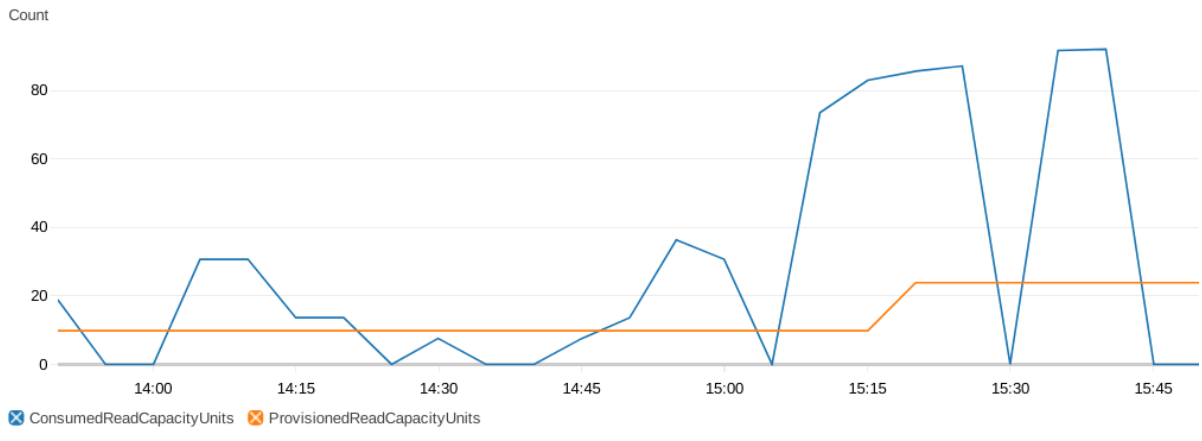

Figure 10: DynamoDB provisioned (orange) and requested (blue) RCU for 500 concurrent users

By analyzing the CloudWatch logs it was possible to identify the "Search Lecture" opera-

tion as the test bottleneck: the obtained errors were generated by a timeout signal caused by the search lectures lambda. Despite it was possible to increase the timeout, it would have provided the final user with obsolete results, especially during peak traffic times.

To increase the performances, a global secondary index was given to the lectures table: the course id.
Under this new scenario, the test of the users routine provided much better results, as shown by the graph below. In particular, by testing up to 2500 (the free tier cap) concurrent users, no unexpected response was given.

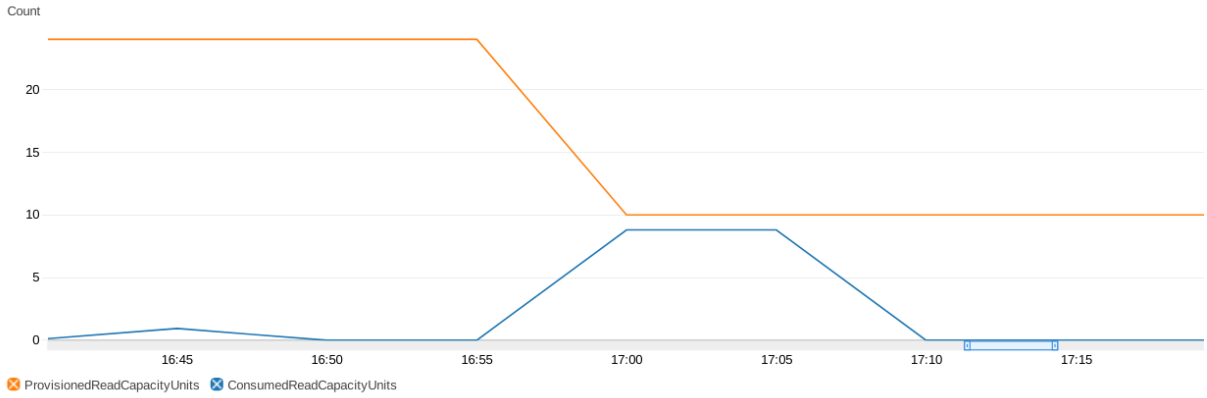

Figure 11: DynamoDB provisioned (orange) and requested (blue) RCU for 500 concurrent users by using the course id as secondary index

5.2.3 Consistency Test

In order to try to break the database consistency we tried to overbook a lecture by sending a number of simultaneous requests that exceeded the number of available seats in the classroom.

The test has been performed on a "realistic" classroom of 100 available seats, the number of attempted booking keeps increasing from 110% of the room capacity to 500%. The final consistency tests purposely lies outside a realistic scenario, anyway it was performed to ensure the system correctness also under stressing scenarios.

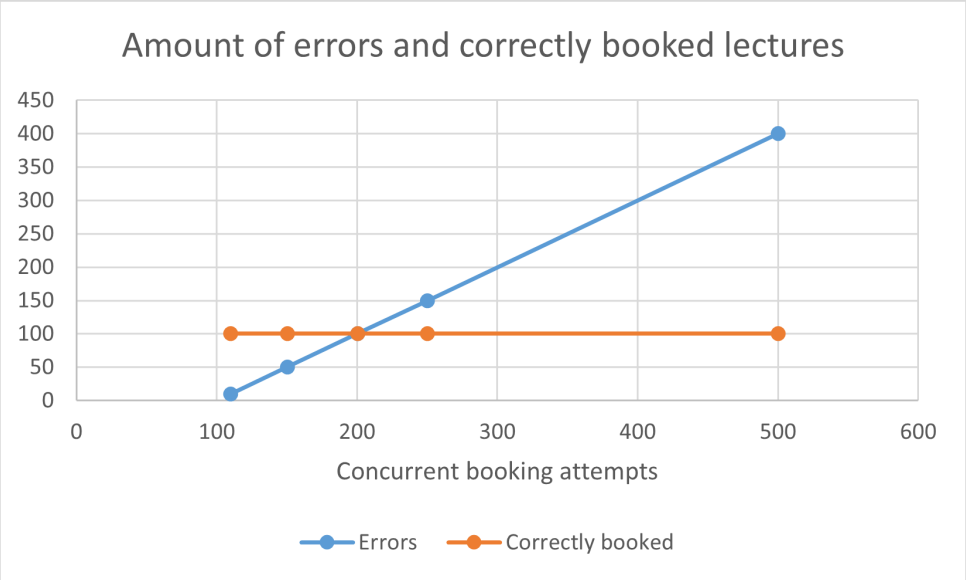

Figure 12: Difference on the API when an intermediary lambda is used to invoke the step function

As expected the number of correctly booked lecture is stable at 100 despite the increasing number of requests, which shows the absence of overbooking. Moreover the number of errors, due to lack of available seats, grows linearly as predicted.

6 Further developments

The tests has shown the system resilience upon managing an everyday workload, however the account limitations didn't allow us to stress the system reactivity to check the response over huge traffic peaks (tens of thousand of concurrent users).
Other future developments for Serverless Prodigit may include:

- The integration of Amazon Cognito to provide a capillary resource access management, which should be merged with the SPID authentication protocol
- The introduction of periodic consistency checker that scans the tables for inconsistencies
- The automatization of adding and removing a batch of lectures