Serverless Heart Rate Stream And Batch Processing Architecture

Greta Luna Ancora
dept. Computer Engeneering
University of Rome Tor Vergata
Rome, Italy
gretaluna.ancora@alumni.uniroma2.eu

Nicola Violante
dept. Computer Engeneering
University of Rome Tor Vergata
Rome, Italy
nicola.violante@alumni.uniroma2.eu

***Abstract*—In this article, we propose a serverless architecture dedicated to the collection and processing—both in streaming and batch modes—of heart rate data acquired from wearable devices. The system also includes services for managing electronic health records, medical staff, and patients. Thanks to the characteristics of the serverless paradigm, the solution ensures high levels of elasticity, scalability, and fault tolerance.**

I. Introduction

The ever-wider adoption of pervasive devices—such as health trackers, medical wearables, and smartwatches—has enabled large-scale telemedicine and health monitoring, allowing real-time detection of clinical anomalies. At the same time, ongoing improvements in communication infrastructure support the streaming transmission and processing of collected data. When connectivity is intermittent, devices can buffer information locally and defer batch processing until the network is restored.

Our work focuses on adopting a serverless architecture to develop a software application capable of managing data streams generated by IoT devices, analyzing them in real time, and making them immediately available to physicians and patients. We relied on services provided by Amazon Web Services (AWS): for the serverless core we employed AWS Lambda in combination with AWS Step Functions to orchestrate application workflows. Among the supporting services we used AWS Cognito for authentication, AWS API Gateway to expose RESTful APIs, AWS Simple Queue Service (SQS) for decoupling components and buffering messages, AWS IoT Core for device management, AWS Simple Storage Service (S3) for batch data storage, AWS DynamoDB for NoSQL persistence, AWS Simple Notification Service (SNS) to send alerts to medical staff, AWS CloudWatch for real-time monitoring and debugging and AWS CloudFormation for application deployment and migration. This serverless architecture ensures automatic scalability, high availability, and resilience, freeing developers from managing infrastructure and allowing them to focus solely on application logic.

II. Background

During the application's development, we leveraged AWS Academy Learner Lab, which provides console access to AWS services by assuming the role IAM LabRole. The backend was implemented in Python and JavaScript, while the frontend was built with HTML.

III. Solution Design

Figure 1 illustrates the overall system architecture, where users interact via a graphical interface to select the desired service. Three user profiles coexist—Admin, Doctor, and Patient—each mapped to a distinct group within an AWS Cognito user pool. To invoke the microservices associated with their role, users must authenticate through AWS Cognito; upon successful authentication they receive a token that must be included in subsequent API calls.

An Admin can register a doctor by invoking the Lambda function *addDoctor*. A Doctor, once authenticated, can add patients, read and update clinical records, and view cardiac data analysis results via the Lambda functions a*ddPatient*, *readClinicalRecords*, *writeClinicalRecords*, and *readAnalysis-Results*, respectively. A Patient can register a new device using the Lambda function *addDevice* and retrieve their own clinical records through *patientReadClinicalRecords*.

IoT devices stream heart-rate data to AWS IoT Core over MQTT by publishing to the topic *IotSimulator/bpm*. Each incoming message triggers an IoT Core rule that invokes the Lambda function *sendToSQS*, which forwards the data to the FIFO SQS queue *bpmValues.fifo*. The arrival of a message in this queue then triggers the Lambda function *sendToStep-Function*, which looks up patient information by deviceId and starts an execution of the *AnalyzeStateMachine* state machine. In case of failure, the message remains in the queue to be retried on subsequent invocations, ensuring system resilience. The *AnalyzeStateMachine* processes the BPM values, persists the results, and—if an anomaly is detected—publishes an alert to the SNS topic *AnomalyNotification*.

If a device goes offline, it buffers data locally in a JSON file and, once connectivity is restored, publishes that file via MQTT so it can be uploaded to the S3 bucket *batchdataheart-beat*. The file upload triggers the Lambda function *splitBatch-Data*, which parses the JSON into individual records, retrieves each patient's data by deviceId, and invokes *AnalyzeStateMachine* for record, thus supporting both streaming and batch processing.

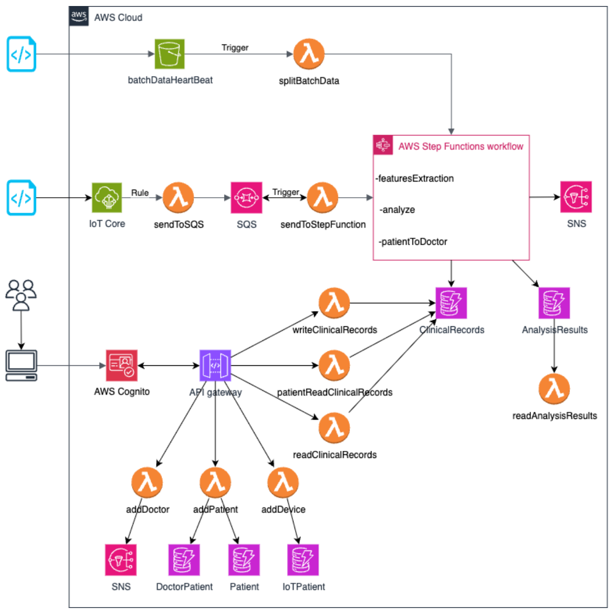

Fig. 1. System Architecture

IV. Solution Details

In this section we analyze in more details the tools used for application development.

A. AWS Lambda and AWS Step Functions

”You can use AWS Lambda to run code without provisioning or managing servers. Lambda runs your code on a high-availability compute infrastructure and manages all the computing resources, including server and operating system maintenance, capacity provisioning, automatic scaling, and logging. You organize your code into Lambda functions. The Lambda service runs your function only when needed and scales automatically.” [1]

The lambda functions that implement the application logic of our system are:

- *addDevice*: takes the device ID as input, retrieves the patient ID from the authentication token, and adds an entry to the *IoTPatient* DynamoDB table
- *addDoctor*: takes the new doctor's credentials as input, uses them to create a user in the Cognito user pool, and adds them to the Doctor group
- *addPatient*: takes the new patient's credentials and their gender and age information as input, uses the credentials to create a user in the Cognito user pool and adds them to the Patient group, inserts an entry into the *DoctorPatient* DynamoDB table containing the doctor's and patient's IDs, and inserts an entry into the *Patient* DynamoDB table containing the patient's ID along with their gender and age
- *checkAdminGroupAuth*, *checkDoctorGroupAuth*, *checkPatientGroupAuth*: takes the user's authentication token as input, verifies that it belongs to a specific group, and if so, grants the user permission to invoke the API
- *login*: server-side verification of the user's membership in the Cognito user pool and issuance of the authentication token
- *patientReadClinicalRecords*: extracts the patient ID from the authentication token and retrieves the corresponding entries from the *ClinicalRecords* DynamoDB table
- *readAnalysisResults*: takes the patient ID as input and retrieves the corresponding entries from the *AnalysisResults* DynamoDB table
- *readClinicalRecords*: takes the patient ID as input and retrieves the related entries from the *ClinicalRecords* DynamoDB table
- *sendToSQS*: takes a message from the *IoTSimulator/bpm* topic as input and publishes it to the *bpmValues.fifo* SQS queue
- *sendToStepFunction*: takes up to ten messages from the *bpmValues.fifo* SQS queue related to the device ID in the message that triggered the Lambda, retrieves the corresponding patient ID from the *IoTPatient* DynamoDB table, uses that patient ID to fetch gender and age from the *Patient* DynamoDB table, and starts the *AnalyzeStateMachine* state machine execution—passing in the data retrieved from the tables along with the SQS message payload
- *splitBatchData* takes a JSON file as input, extracts the device ID, retrieves the corresponding patient ID from the *IoTPatient* DynamoDB table, uses that patient ID to fetch gender and age information from the *Patient* DynamoDB table, and starts the *AnalyzeStateMachine* state machine execution—passing in the data retrieved from the tables and one of the records from the JSON file
- *writeClinicalRecords*: takes as input the patient's ID and a string containing the anomaly description provided by the doctor, and inserts an entry into the DynamoDB table *AnalysisResults* with those two input parameters

Furthermore, three other Lambda functions are used as tasks within the *AnalyzeSatateMachine* state machine.

”With AWS Step Functions, you can create workflows, also called State machines, to build distributed applications, automate processes, orchestrate microservices, and create data and machine learning pipelines. Step Functions is based on state machines and tasks. In Step Functions, state machines are called workflows, which are a series of event-driven steps. Each step in a workflow is called a state. For example, a Task state represents a unit of work that another AWS service performs, such as calling another AWS service or API. Instances of running workflows performing tasks are called executions in Step Functions.” [2]

The state machine consists of the following states:

- *featuresExtraction*: this step precedes the actual analysis; it aggregates the input data by computing the average of the BPM values
- *analyze*: this step takes the average BPM computed by the previous step and, taking into account the patient's sex and age, compares it against predefined threshold values

for the following anomaly categories: severe bradycardia, moderate bradycardia, no anomaly, moderate tachycardia, and severe tachycardia; it then returns the detected anomaly type

- *Choice*: the execution flow is directed by two rules: if any anomalies are detected, it routes to the left branch; if none are detected, it routes to the right branch
- *Parallel*: the following tasks are executed in parallel
- *patientToDocator*: retrieves, from the *DoctorPatient* DynamoDB table, the email address of the doctor associated with the patient who exhibits the anomaly.
- *AnomalyNotification*: sends a notification to the doctor subscribed to the *AnomalyDescription* topic whose email matches the one retrieved in the previous step
- *AnalysisResults*: persists the analysis details into the *AnalysisResults* DynamoDB table

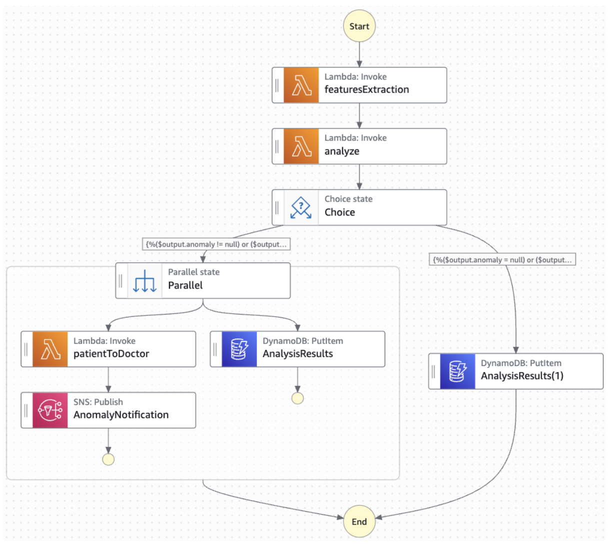

Fig. 2. AnalyzeStateMachine

B. AWS Simple Notification Service (SNS)

"Amazon Simple Notification Service (Amazon SNS) is a fully managed service that provides message delivery from publishers (producers) to subscribers (consumers). Publishers communicate asynchronously with subscribers by sending messages to a topic, which is a logical access point and communication channel." [7]

We used the SNS service to send alerts to doctors whenever an anomaly is detected. We created the *AnomalyNotification* topic, to which each doctor is subscribed at registration using their email address. For each subscription, a policy of the following form is also defined:

Listing 1. Esempio di filtro SNS

```
{
  "doctor_id": [
    "doctor@example.com"
  ]
}
```

When a message is published to the *AnomalyNotification* topic during the execution of the *AnalyzeStateMachine*, the doctor's ID is supplied among the message's attributes. As soon as a message is published to the topic, each subscription evaluates its filter policy against the message attributes. If there is a match, the corresponding doctor is notified.

Listing 2. SNS Publish Task

```
"SNS Publish": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish
      ↪ ",
    "Arguments": {
        "TopicArn": "arn:aws:sns:us-east
          ↪ -1:832059618623:
          ↪ AnomalyNotification",
        "Message": "{% 'An anomaly has been
          ↪ detected for the patient ' &
          ↪ $output2.patient_id & ': ' &
          ↪ $output.anomaly %}",
        "MessageAttributes": {
            "doctor_id": {
                "DataType": "String",
                "StringValue": "{% $output2.
                    ↪ doctor_id %}"
            }
        }
    },
    "End": true
}
```

C. AWS Cognito

"An Amazon Cognito user pool is a user directory for web and mobile app authentication and authorization. From the perspective of your app, an Amazon Cognito user pool is an OpenID Connect (OIDC) identity provider (IdP). A user pool adds layers of additional features for security, identity federation, app integration, and customization of the user experience." [3] We chose for authentication to use JSON Web Token (JWT), that is a token that contains claims about the identity of the authenticated user, such as name, email, and phone number. This token is also used to manage web API operations (this mechanism is described in the *AWS API Gateway* subsection).

D. AWS API Gateway

"Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. As an API Gateway API developer, you can create APIs for use in your own client applications. Or you can make your APIs available to third-party app developers. " [4]

We used the API Gateway service to create and publish a REST API for our application in order to expose the services it implements. In particular, during its creation we chose to enable the following integrations:

- *lambda proxy integration*: allows the developer to integrate an API method – or an entire API – with a Lambda

function; the developer can change the Lambda function implementation at any time without needing to redeploy the API
- *CORS integration*: CORS defines a way for client web applications that are loaded in one domain to interact with resources in a different domain; CORS allows the client browser to check with the third-party servers if the request is authorized before any data transfers

With regard to access control, we decided to associate a Lambda Authorizer with each resource in the REST API. When a client makes a request to one of your API's methods, API Gateway invokes the corresponding Lambda authorizer. The authorizer takes the caller's identity as input and returns an IAM policy as output. This allowed us to implement a custom authorization scheme using a bearer-token authentication strategy. The authorizers we used are the Lambda functions *CheckAdminGroupAuth*, *CheckDoctorGroupAuth* and *CheckPatientGroupAuth*, as described in the previous section *AWS Lambda and AWS Step Functions*.

E. AWS IoT Core

"AWS IoT provides the cloud services that connect your IoT devices to other devices and AWS cloud services. AWS IoT provides device software that can help you integrate your IoT devices into AWS IoT-based solutions. If your devices can connect to AWS IoT, AWS IoT can connect them to the cloud services that AWS provides." [11]

To implement our solution, we first created a "Thing" within AWS IoT Core to represent the device simulated by our Python scripts. We then attached an X.509 certificate to this Thing, which is essential for AWS IoT Core to securely authenticate each connection. Authorization is managed via an AWS IoT policy, in which we specified the permissions required to connect to the MQTT message broker and to publish and subscribe to the *IoTSimulator/bpm* topic.

Next, we defined an IoT Core routing rule that captures all messages arriving on the IoTSimulator/bpm topic. Because IoT Core rules do not natively support sending messages to FIFO SQS queues, each time the rule is triggered, it invokes a Lambda function named *sendToSQS*. This Lambda function takes the payload from the device and publishes it to our FIFO SQS queue *bpmValues.fifo*, ensuring both message order and atomicity.

F. AWS Simple Queue Service (SQS)

"Amazon Simple Queue Service (Amazon SQS) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components. Amazon SQS offers common constructs such as dead-letter queues and cost allocation tags. It provides a generic web services API that you can access using any programming language that the AWS SDK supports." [6]

For ingesting data streamed from the IoT devices, we use an Amazon SQS queue named *bmpValues.fifo*. This choice allows us to leverage the main benefits of a message-queue architecture:

- interoperability: publishers and consumers can be implemented in different languages, interacting with the queue via heterogeneous APIs
- fully managed service: Amazon SQS automatically handles replication, load balancing, and scalability, making the underlying infrastructure transparent to developers
- pull model: consumers poll the queue to retrieve messages, with a timeout-based delivery semantics that governs reliability

The choice of a FIFO queue guarantees chronological message delivery—i.e., the consumer receives messages in the exact order they were enqueued by the publisher—and duplicate management via the *MessageDeduplicationId*. This ID is used exclusively in Amazon SQS FIFO queues to prevent duplicate message delivery: within a 5-minute deduplication window, only one instance of a message bearing the same deduplication ID is processed and delivered. If Amazon SQS has already accepted a message with a given deduplication ID, any subsequent messages with that same ID will be acknowledged but not delivered to consumers. The drawback of a FIFO queue is its lower throughput compared to a standard queue, which is why we opted to enable the high-throughput FIFO feature. Moreover, when a Lambda function encounters an error while processing a batch, all messages in that batch become visible in the queue again by default, including messages that Lambda processed successfully. As a result, the function can end up processing the same message several times. To avoid reprocessing successfully processed messages in a failed batch, we configured the lambda function *sendToStepFunction*'s SQS trigger to make only the failed messages visible again, enabling the option *ReportBatchItemFailures*. When it is activated, Lambda doesn't scale down message polling when function invocations fail, so the failed messages don't impact the message processing rate.

G. AWS DynamoDB

"Amazon DynamoDB is a serverless, NoSQL, fully managed database with single-digit millisecond performance at any scale. DynamoDB addresses your needs to overcome scaling and operational complexities of relational databases. DynamoDB is purpose-built and optimized for operational workloads that require consistent performance at any scale." [9]

We chose DynamoDB because it allowed us to focus entirely on our application's logic without having to worry about servers, patches, or maintenance: in just a few clicks the database was up and ready to handle the load. Its NoSQL nature—with both key-value and document models—gave us the flexibility to denormalize data and serve queries with single-digit millisecond latency, even as our user base grew exponentially. Thanks to on-demand mode, we pay only for the operations we actually perform and can scale from zero to millions of requests per second without any manual intervention, thus ensuring a fully serverless, resilient application that's ready to tackle any traffic spike.

In particular, our application includes five different tables:

- *AnalysisResults*: used to store the analyzed heartbeat values in a given time interval, along with their average and any detected anomaly. The partition key is the patientID, while the sort key is the timestamp, thus organizing the results for each patient chronologically
- *ClinicalRecords*: used to store the descriptions of detected anomalies, once analyzed by the physicians, along with the IDs of the physicians who documented them. The partition key is the patient's name, while the sort key is the timestamp, thus maintaining a chronologically ordered history of clinical annotations for each patient
- *DoctorPatient*: used to track the assignment of patients to their respective doctors. The partition key is the patientId, while the sort key is the doctorId, thus ensuring rapid access to all doctors associated with each patient and vice versa
- *IoTPatient*: used to track the devices associated with each patient. The partition key is the deviceID, allowing rapid identification of which patient a given IoT device belongs to
- Patient: used to store the patient's demographic information, specifically age and gender. The partition key is, naturally, the patientId, which uniquely identifies each patient

H. AWS Simple Storage Service (S3)

"Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use Amazon S3 to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides management features so that you can optimize, organize, and configure access to your data to meet your specific business, organizational, and compliance requirements." [10]

We used the S3 service to create the following buckets:

- *batchdataheartbeat*: used by IoT devices to upload data for batch processing
- *frontendheartbeat*: contains the HTML files used as the application's frontend

For the *frontendheartbeat* bucket, we enabled the static website hosting option so that objects in the bucket can be accessed via HTTP requests.

I. AWS CloudFormation

"AWS CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Amazon EC2 instances or Amazon RDS DB instances), and CloudFormation takes care of provisioning and configuring those resources for you." [8]

Once all the components of our AWS-based system were completed, we used AWS CloudFormation to export the configurations and settings into Infrastructure as Code (IaC) templates. We produced four different templates in YAML format:

- *CognitoTemplate*: automatically defines and configures the three user groups in our Cognito User Pool—Doctor, Admin, and Patient—that will be used to consistently manage permissions and access policies within the application
- *RestAPITemplate*: defines our API in API Gateway by directly importing the Swagger file stored in S3, configuring routes, methods, models, and validations without any manual rewriting. When exporting the API, we chose Swagger as the specification, JSON as the format, and enabled API Gateway extensions
- *BatchBucketTemplate*: describes an S3 bucket dedicated to batch heart rate data
- *HeartBeatTemplate*: automates the provisioning of all remaining resources for our *HeartBeat* application, including Lambda functions, SQS queues, SNS topics, IoT rules, S3 buckets, DynamoDB tables, and Step Functions

We divided the infrastructure into four distinct templates to avoid circular dependencies and ensure that each resource always references components that already exist.

J. AWS CloudWatch

"Amazon CloudWatch monitors your Amazon Web Services (AWS) resources and the applications you run on AWS in real time, and offers many tools to give you system-wide observability of your application performance, operational health, and resource utilization." [5]

We used CloudWatch primarily to quickly pinpoint where errors occurred and to examine past inputs and outputs: by centralizing logs in our services' log groups, we accessed entries containing exceptions, payloads, and results directly. This way, a single view of the logs was all we needed to understand what happened and immediately fix the issue.

V. Discussion

The IAM role LabRole constrained our capabilities, preventing us from fully leveraging AWS's advanced services and features. Therefore, we could consider extending and enhancing the application's functionality by using additional services not supported by the IAM role LabRole. One improvement would be to replace the current Lambda functions dedicated to IoT data analysis with more advanced machine learning models implemented via AWS SageMaker AI [12]: this would provide us with a managed environment for building, training, and deploying algorithms capable of extracting deeper, real-time insights. Similarly, to make infrastructure provisioning more repeatable and controlled, we could adopt Terraform as our Infrastructure as Code tool, defining all AWS resources in configuration files versioned alongside the application code. Finally, instead of relying on a simple S3 bucket for static front-end hosting, we could leverage AWS Amplify Hosting

[13]: thanks to its Git-based workflow and global CDN distribution, every code change would be automatically propagated and delivered with optimized load times.

References

[1] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[2] https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html
[3] https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools.html
[4] https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
[5] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html
[6] https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html
[7] https://docs.aws.amazon.com/sns/latest/dg/welcome.html
[8] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html
[9] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
[10] https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
[11] https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html
[12] https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html
[13] https://docs.aws.amazon.com/amplify/latest/userguide/welcome.html