



Università
di Catania

DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Giuseppe Pitruzzella

Healthcare: Cloud-Native Hospital Monitoring System

RELAZIONE PROGETTO FINALE

Anno Accademico 2025 - 2026

Indice

1 Introduzione e Scenario	3
1.1 L'evoluzione della Sanità Digitale	3
1.2 I limiti delle infrastrutture tradizionali	3
1.3 Un approccio Cloud-Native Serverless	4
1.4 Obiettivi del Progetto	4
2 Architettura del Sistema	6
2.1 Panoramica High-Level	6
2.2 Stack Tecnologico	6
2.2.1 Compute Layer (AWS Lambda)	6
2.2.2 Data Layer (Amazon DynamoDB)	7
2.2.3 API & Networking (Amazon API Gateway)	7
2.2.4 Frontend & Hosting (React & Amazon S3)	7
2.2.5 Sicurezza (Amazon Cognito & IAM)	7
2.2.6 Industrializzazione (DevOps)	8
2.3 Diagramma Architetturale	8
3 Gestione dei Dati e Simulazione IoT	10
3.1 Il Dataset e il Modello Clinico	10
3.2 Modellazione del Database (DynamoDB)	10
3.2.1 Tabella Patients (Registry)	10
3.2.2 Tabella VitalSigns (Time-Series Data)	11
3.3 Il Simulatore IoT (vitals-simulator)	11
3.4 Automazione via EventBridge	11
4 Elaborazione Event-Driven e Backend	12
4.1 Il Paradigma “Reactive”	12
4.2 DynamoDB Streams: Il Sistema Nervoso	12
4.3 Il “Detective”: L’Algoritmo di Analisi	12
4.4 Scalabilità	13
5 Sicurezza e Controllo Accessi	14
5.1 Sicurezza “By Design”	14
5.2 Autenticazione Utenti (Amazon Cognito)	14
5.3 Protezione delle API (Authorizers)	14
5.4 Gestione dei Permessi Infrastrutturali (AWS IAM)	14

6 Real-Time e Notifiche Multicanale	16
6.1 Oltre il REST: I limiti del Polling	16
6.2 WebSocket API	16
6.3 Amazon SNS e Alarm Fatigue	16
7 Dashboard	17
7.1 Architettura SPA	17
7.2 Integrazione Sicurezza	17
7.3 Gestione dello Stato e Resilienza	17
7.4 Deployment Serverless	17
8 Industrializzazione e Infrastructure as Code	18
8.1 Dal Manuale all'Automatizzato: La necessità di Terraform	18
8.2 Strategia Multi-Ambiente e Naming Dinamico	18
8.3 Gestione delle Dipendenze	18
8.4 Configurazione Automatizzata di CORS	19
9 Containerizzazione e Portabilità con Docker	20
9.1 Il Ruolo di Docker	20
9.2 Simulazione Locale	20
9.3 Build Multi-Stage per il Frontend	20
9.4 Orchestrazione con Docker Compose	20
10 Continuous Integration e Continuous Deployment	22
10.1 Automazione del Ciclo di Rilascio	22
10.2 Sicurezza e Workflow	22
10.3 Il Flusso di Deployment	22
11 Conclusioni e Sviluppi Futuri	23
11.1 Sintesi del Progetto	23
11.2 Risultati Raggiunti	23
11.3 Sviluppi Futuri	23

Capitolo 1

Introduzione e Scenario

1.1 L’evoluzione della Sanità Digitale

Il settore sanitario sta attraversando una fase di profonda trasformazione digitale, guidata dalla necessità di ottimizzare le risorse e migliorare la qualità dell’assistenza ai pazienti. In questo scenario, l'**Internet of Medical Things (IoMT)** gioca un ruolo cruciale: la capacità di connettere dispositivi medici (monitor multiparametrici, sensori indossabili, lettori biometrici) alla rete permette di raccogliere enormi volumi di dati clinici in tempo reale.

Tuttavia, la semplice raccolta del dato non è sufficiente. In contesti critici, come le unità di terapia intensiva o il monitoraggio post-operatorio, il valore dell’informazione è strettamente legato alla tempestività con cui viene elaborata. Un ritardo di pochi minuti nella rilevazione di un’aritmia cardiaca o di un calo della saturazione di ossigeno può avere conseguenze gravi per il paziente.

Di conseguenza, le moderne infrastrutture ospedaliere richiedono sistemi software capaci non solo di archiviare storici clinici, ma di agire come “sistemi nervosi” digitali, in grado di reagire istantaneamente alle anomalie.

1.2 I limiti delle infrastrutture tradizionali

Molte strutture sanitarie si affidano ancora a sistemi di monitoraggio basati su architetture “On-Premise” o legacy, che presentano diverse criticità strutturali:

1. **Latenza e Polling:** I sistemi tradizionali spesso utilizzano meccanismi di Polling (il client interroga il server periodicamente: “Ci sono nuovi dati?”). Questo approccio introduce latenza e spreca risorse di rete e di calcolo quando non ci sono variazioni nello stato del paziente.
2. **Scalabilità Rigida:** Un server fisico ha limiti di capacità fissi. Durante un’emergenza sanitaria (es. una pandemia) che comporta un improvviso aumento dei pazienti monitorati, i server on-premise rischiano la saturazione e il collasso, richiedendo tempi lunghi e costi elevati per l’upgrade hardware.

3. **Costi di Manutenzione (TCO):** La gestione di server fisici richiede personale IT dedicato per aggiornamenti, patch di sicurezza e gestione dei backup, distraendo risorse dal core business sanitario.
4. **Single Point of Failure:** Se il server centrale si guasta, l'intero reparto rischia di rimanere “al buio” senza accesso ai dati vitali.

1.3 Un approccio Cloud-Native Serverless

Per superare queste limitazioni, questo progetto propone e realizza un sistema di monitoraggio ospedaliero basato interamente su architettura **Cloud-Native** utilizzando i servizi di Amazon Web Services (AWS).

La scelta architettonica ricade sul paradigma **Serverless**. In questo modello, la gestione dell’infrastruttura sottostante (server, sistema operativo, scaling) è completamente delegata al Cloud Provider. Il codice viene eseguito in risposta a eventi (Event-Driven), garantendo un utilizzo delle risorse on-demand.

I vantaggi chiave di questa soluzione includono:

- **Reattività Real-Time:** Abbandono del polling a favore di connessioni WebSocket persistenti, che permettono di inviare i dati al cruscotto medico (Dashboard) nell’istante stesso in cui vengono generati (“Push notification”).
- **Scalabilità Elastica:** Il sistema è in grado di gestire da 10 a 100.000 pazienti simultanei senza alcuna modifica manuale all’infrastruttura, grazie alla scalabilità automatica di AWS Lambda e DynamoDB.
- **Modello di Costo Pay-per-Use:** A differenza dei server sempre accesi, il sistema genera costi solo quando i dati vengono effettivamente elaborati. Se non ci sono pazienti, il costo infrastrutturale tende a zero.
- **Alta Disponibilità:** Sfruttando la ridondanza nativa dei data center AWS (Availability Zones), il sistema è resiliente ai guasti hardware.

1.4 Obiettivi del Progetto

Il progetto mira alla realizzazione di un prototipo funzionante (“Minimum Viable Product”) che soddisfi i seguenti requisiti funzionali e non funzionali:

1. **Ingestione Dati IoT:** Simulare un flusso continuo di parametri vitali (battito cardiaco, pressione, temperatura, SpO2) per centinaia di pazienti virtuali.
2. **Analisi in Tempo Reale:** Implementare algoritmi automatici per il rilevamento istantaneo di condizioni critiche (es. Tachicardia, Ipossia) direttamente sul flusso dati.

3. **Dashboard Interattiva:** Fornire al personale medico un'interfaccia web aggiornata in tempo reale senza necessità di ricaricare la pagina.
4. **Sistema di Alerting Multicanale:** Garantire che gli allarmi critici raggiungano il personale medico ovunque si trovi, tramite notifiche visive sulla dashboard ed e-mail di emergenza.
5. **Sicurezza e Controllo Accessi:** Assicurare che solo il personale autorizzato (medici autenticati) possa accedere ai dati sensibili dei pazienti, in conformità con i principi di sicurezza moderni.

Capitolo 2

Architettura del Sistema

2.1 Panoramica High-Level

L’architettura progettata per il *Cloud-Native Hospital Monitoring System* segue rigorosamente il paradigma **Event-Driven** (guidato dagli eventi). A differenza delle architetture monolitiche tradizionali, dove un singolo server gestisce tutte le funzionalità, il sistema è decomposto in Microservizi indipendenti e “Stateless” (senza stato), che comunicano tra loro in modo asincrono.

Il flusso logico dei dati attraversa quattro fasi distinte:

1. **Generazione (IoT Simulation):** Un modulo di simulazione genera flussi di dati vitali sintetici, replicando il comportamento di sensori medici connessi (ECG, saturimetri).
2. **Ingestione e Storage:** I dati vengono acquisiti e persistiti immediatamente in un database NoSQL ad alte prestazioni.
3. **Elaborazione (Processing):** L’arrivo di nuovi dati scatena automaticamente (“trigger”) l’esecuzione della logica di business per l’analisi clinica.
4. **Presentazione e Notifica:** I risultati elaborati vengono inviati in tempo reale alla Dashboard frontend e, in caso di criticità, ai canali di notifica di emergenza.

2.2 Stack Tecnologico

La selezione tecnologica è stata guidata dai requisiti di scalabilità automatica e minimizzazione dell’overhead di gestione (Serverless). Di seguito i componenti chiave:

2.2.1 Compute Layer (AWS Lambda)

Il cuore computazionale del sistema è affidato ad AWS Lambda. Sono state sviluppate tre funzioni principali in linguaggio Python:

- `vitals-simulator`: Genera i dati dei pazienti e li scrive nel database.

- **alert-detector:** Analizza lo stream di dati in entrata per identificare valori fuori soglia (es. battito cardiaco > 110 bpm).
- **connection-manager:** Gestisce il ciclo di vita delle connessioni Web-
Socket (connessione/disconnessione utenti).

2.2.2 Data Layer (Amazon DynamoDB)

Per la persistenza dei dati è stato scelto Amazon DynamoDB, un database NoSQL chiave-valore. La scelta è motivata dalla necessità di gestire elevati throughput di scrittura e lettura con latenze a singola cifra di millisecondo. Il modello dati comprende tre tabelle principali:

- **Patients:** Anagrafica statica dei pazienti.
- **VitalSigns:** Time-series dei parametri vitali (con DynamoDB Streams attivo per abilitare l'architettura reattiva).
- **Alerts:** Registro storico degli allarmi generati.

2.2.3 API & Networking (Amazon API Gateway)

Il sistema espone due tipologie di interfacce:

- **REST API:** Per le operazioni standard di richiesta dati (es. recupero lista pazienti).
- **WebSocket API:** Per mantenere un canale di comunicazione bidirezionale persistente tra il cloud e il browser, essenziale per l'aggiornamento “Live” dei grafici.

2.2.4 Frontend & Hosting (React & Amazon S3)

L'interfaccia utente è una Single Page Application (SPA) sviluppata in React.js. L'hosting è realizzato sfruttando le funzionalità di Static Website Hosting di Amazon S3, che garantisce durabilità (99.99999999%) e scalabilità globale senza necessità di web server tradizionali (come Apache o Nginx).

2.2.5 Sicurezza (Amazon Cognito & IAM)

La sicurezza è implementata su due livelli:

- **Identità Utente:** Amazon Cognito gestisce l'autenticazione dei medici (User Pool), fornendo token JWT (JSON Web Token) sicuri per l'accesso.
- **Permessi Infrastrutturali:** AWS IAM (Identity and Access Management) applica il principio del *Least Privilege*, garantendo che ogni funzione Lambda abbia accesso solo alle risorse strettamente necessarie.

2.2.6 Industrializzazione (DevOps)

Per garantire la riproducibilità e l'automazione del sistema, sono stati integrati strumenti di DevOps moderni:

- **Terraform:** Per la definizione dell'infrastruttura come codice (IaC).
- **Docker:** Per la containerizzazione dei moduli di simulazione.
- **GitHub Actions:** Per la pipeline di Continuous Integration/Continuous Deployment (CI/CD).

2.3 Diagramma Architetturale

La seguente figura illustra l'interazione tra i componenti sopra descritti. Si evidenzia come il flusso dei dati sia innescato dagli eventi generati dal simulatore e propagati attraverso DynamoDB Streams fino alla Dashboard.

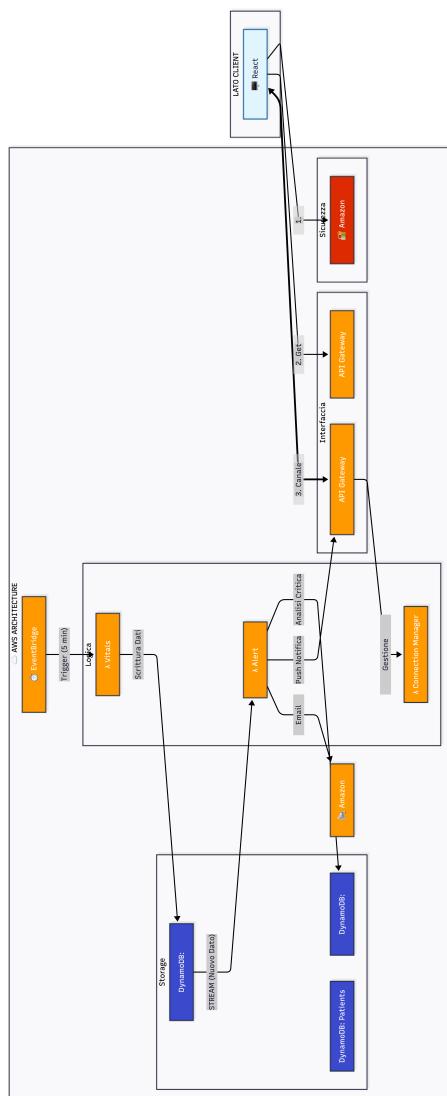


Figura 2.1: Diagramma Architetturale del Sistema

Capitolo 3

Gestione dei Dati e Simulazione IoT

3.1 Il Dataset e il Modello Clinico

La validità di un sistema di monitoraggio dipende dalla qualità dei dati che elabora. Per questo progetto, non potendo collegare sensori fisici a pazienti reali per motivi di privacy e logistica, è stato necessario generare un dataset sintetico verosimile.

Il modello dei dati si ispira ai parametri clinici standard presenti nel dataset pubblico MIMIC-III (Medical Information Mart for Intensive Care). Per ogni paziente vengono tracciati quattro parametri vitali fondamentali:

1. **Frequenza Cardiaca (HR):** Battiti per minuto (bpm).
2. **Pressione Sanguigna (BP):** Sistolica e Diastolica (mmHg).
3. **Saturazione Ossigeno (SpO₂):** Percentuale (%).
4. **Temperatura Corporea:** Gradi Celsius (°C).

3.2 Modellazione del Database (DynamoDB)

A differenza dei database relazionali (SQL), la progettazione in Amazon DynamoDB (NoSQL) è guidata dagli Access Patterns. Sono state progettate due tabelle principali:

3.2.1 Tabella Patients (Registry)

Contiene i dati anagrafici statici.

- **Partition Key (PK):** patient_id (Stringa, es. "PT-1024").
- **Attributi:** Nome, Età, Genere, Reparto.
- **Scopo:** Permettere il recupero rapido delle informazioni del paziente per arricchire la dashboard.

3.2.2 Tabella VitalSigns (Time-Series Data)

Questa tabella è progettata per gestire serie temporali ad alta frequenza di scrittura.

- **Partition Key (PK):** patient_id (Raggruppa tutti i dati di un singolo paziente).
- **Sort Key (SK):** timestamp (Ordina i dati cronologicamente ISO-8601).
- **Scopo:** Questa struttura permette di eseguire query efficienti come: “Dammi tutti i parametri vitali del paziente X nell’ultima ora”, senza dover scansionare l’intero database.

3.3 Il Simulatore IoT (vitals-simulator)

Il componente `vitals-simulator` è una funzione AWS Lambda che agisce come un “Paziente Virtuale”. Per evitare la generazione di dati casuali irrealistici (es. temperatura che salta da 36°C a 41°C in un minuto), è stato implementato un algoritmo di “Random Walk” (Camminata Casuale) con vincoli fisiologici.

Logica dell’Algoritmo:

1. **Fetch dello Stato:** La funzione legge la lista dei pazienti e i loro valori “baseline”.
2. **Variazione Delta:** Il simulatore calcola una variazione rispetto al valore ideale.
3. **Simulazione Anomalie:** Con una probabilità configurabile (es. 5%), il sistema introduce una “deviazione critica” per simulare l’insorgere di una patologia.
4. **Clamping:** I valori finali vengono forzati entro limiti biologici possibili per evitare errori di dato “sporco”.

3.4 Automazione via EventBridge

Poiché le funzioni Lambda sono “effimere”, è stato configurato Amazon EventBridge Scheduler con una regola cronologica (`rate(5 minutes)`). Questo componente agisce come il “metronomo” del sistema: ogni 5 minuti invoca la Lambda, scatenando la scrittura massiva di nuovi record su DynamoDB.

Capitolo 4

Elaborazione Event-Driven e Backend

4.1 Il Paradigma “Reactive”

Nelle architetture tradizionali, il server deve interrogare il database ripetutamente (Polling) per sapere se ci sono novità. Questo approccio è inefficiente. In questo progetto è stato adottato un approccio **Event-Driven**: il sistema è “dormiente” e non consuma risorse finché non si verifica un evento significativo (la scrittura di un nuovo dato nella tabella `VitalsSigns`).

4.2 DynamoDB Streams: Il Sistema Nervoso

Per realizzare questo meccanismo reattivo, è stata abilitata la funzionalità **DynamoDB Streams**.

- **Configurazione:** Modalità `NEW_IMAGE` (trasporta l’intera fotografia del record).
- **Trigger:** Collegato direttamente alla funzione Lambda `lambda alert-detector`.

4.3 Il “Detective”: L’Algoritmo di Analisi

La funzione `lambda alert-detector` agisce come un medico virtuale. Il flusso logico dell’algoritmo implementato in Python è il seguente:

1. **Decodifica:** Deserializza il batch di eventi dallo Stream.
2. **Valutazione Clinica:** Confronta ogni record con le Soglie di Allarme (es. Tachicardia se $HR > 110$ bpm, Ipoossia se $SpO2 < 90\%$).
3. **Biforcazione:**
 - **Caso A (Critico):** Genera un oggetto Alert, lo persiste su DB, invia notifica WebSocket e attiva SNS (Email).

- **Caso B (Stabile):** Invia comunque un payload WebSocket per aggiornare il grafico in tempo reale.

Esempio di Logica (Pseudocodice):

```
1 def analyze_patient(vitals):
2     violations = []
3
4     # Regola Tachicardia
5     if vitals.heart_rate > 110:
6         violations.append("TACHICARDIA DETECTED")
7
8     # Regola Iporessia
9     if vitals.spo2 < 90:
10        violations.append("IPRESSIA CRITICA")
11
12    if len(violations) > 0:
13        trigger_code_red(violations) # Allarme + Email
14    else:
15        broadcast_live_update(vitals) # Solo aggiornamento grafico
```

4.4 Scalabilità

Grazie al partizionamento di DynamoDB e allo “Scale Out” di AWS Lambda, il sistema può processare migliaia di eventi al secondo mantenendo una latenza costante.

Capitolo 5

Sicurezza e Controllo Accessi

5.1 Sicurezza “By Design”

Trattandosi di dati sanitari (PHI), la sicurezza è stata integrata in ogni livello. L’obiettivo è garantire che ogni transazione sia autenticata e che ogni componente operi con il minimo livello di privilegi.

5.2 Autenticazione Utenti (Amazon Cognito)

La gestione delle identità è affidata ad **Amazon Cognito User Pool**. Il flusso segue lo standard OIDC:

1. Il medico inserisce le credenziali nella Dashboard.
2. Cognito rilascia tre token JWT (ID, Access, Refresh).
3. I token hanno scadenza breve per sicurezza.

5.3 Protezione delle API (Authorizers)

È stato configurato un **Cognito Authorizer** sulle rotte REST e WebSocket.

- **Meccanismo:** Il frontend invia l’Access Token nell’header `Authorization`.
- **Validazione Zero-Trust:** API Gateway verifica la firma del token prima di invocare la Lambda. Se il token è invalido, restituisce `401 Unauthorized`.

5.4 Gestione dei Permessi Infrastrutturali (AWS IAM)

Mentre Cognito gestisce “chi sono gli umani”, **AWS IAM** gestisce “cosa possono fare le macchine”. È stato applicato il principio del *Least Privilege*:

- **Vitals Simulator:** Ha permesso di scrivere su VitalSigns, ma non di leggere i dati o accedere ad altre tabelle.
- **Alert Detector:** Può leggere lo Stream e pubblicare su SNS, ma non modificare l'anagrafica.
- **Frontend (S3):** Accessibile solo in lettura.

Capitolo 6

Real-Time e Notifiche Multicanale

6.1 Oltre il REST: I limiti del Polling

Il polling tradizionale (chiedere “ci sono novità?” ogni secondo) introduce latenza e spreca risorse. Per superare questi limiti, il progetto implementa un’architettura **Push** basata su WebSocket.

6.2 WebSocket API

Il protocollo WebSocket permette di stabilire un canale bidirezionale persistente. In ambiente Serverless, questo è gestito da API Gateway WebSocket APIs.

1. **Handshake:** Il browser stabilisce la connessione inviando il Token.
2. **Persistenza:** La Lambda connection-manager salva l’ID di connessione su DynamoDB.
3. **Broadcasting:** La Lambda alert-detector invia i dati a tutti i client connessi attivi.

6.3 Amazon SNS e Alarm Fatigue

Per le notifiche di emergenza (quando il medico non è al PC), viene utilizzato **Amazon SNS** per inviare email. Per evitare l’*Alarm Fatigue* (assuefazione agli allarmi), il sistema distingue tra:

- **Vital Update (Bassa Severità):** Solo aggiornamento grafico WebSocket.
- **Critical Alert (Alta Severità):** WebSocket con allarme visivo + Email urgente tramite SNS.

Capitolo 7

Dashboard

7.1 Architettura SPA

L’interfaccia è una **Single Page Application (SPA)** in React.js. Questo evita il ricaricamento della pagina, garantendo fluidità nella visualizzazione dei grafici in movimento.

7.2 Integrazione Sicurezza

Utilizzando **AWS Amplify UI**, il componente `Authenticator` gestisce il ciclo di vita della sessione e inietta automaticamente il Token JWT nelle chiamate API.

7.3 Gestione dello Stato e Resilienza

L’applicazione utilizza React Hooks (`useState`, `useEffect`) per sincronizzare la vista con i dati WebSocket. È stata implementata una logica di **Auto-Reconnection**: se la connessione cade, il frontend tenta autonomamente di ristabilirla dopo un intervallo di backoff, rendendo la dashboard “Self-Healing”.

7.4 Deployment Serverless

I file statici sono ospitati su **Amazon S3**, garantendo scalabilità infinita e costi ridotti.

Capitolo 8

Industrializzazione e Infrastructure as Code

8.1 Dal Manuale all’Automatizzato: La necessità di Terraform

Nelle fasi iniziali di prototipazione, la configurazione manuale delle risorse tramite la console AWS è un approccio intuitivo, ma presenta criticità insormontabili per un ambiente di produzione professionale. La creazione manuale è intrinsecamente non riproducibile e soggetta a errore umano. Per superare questi limiti, l’intera infrastruttura del progetto è stata codificata utilizzando **Terraform**. Questo approccio (*Infrastructure as Code*) ci permette di definire lo stato desiderato del sistema in file di configurazione dichiarativi, garantendo che l’infrastruttura sia immutabile, versionabile e replicabile.

8.2 Strategia Multi-Ambiente e Naming Dinamico

Nel file `main.tf`, abbiamo implementato una strategia di naming dinamico. Invece di assegnare nomi statici, ogni risorsa viene creata con un suffisso dipendente dall’ambiente (es. “`PatientsTable-demo`” o “`PatientsTable-prod`”). Questa logica garantisce un isolamento completo: è possibile distruggere e ricreare l’ambiente di demo infinite volte senza rischiare di impattare la stabilità della produzione.

8.3 Gestione delle Dipendenze

Terraform risolve automaticamente il grafo delle dipendenze tra le risorse (es. crea la tabella prima della Lambda che la utilizza). Particolare attenzione è stata posta alla sicurezza: il codice genera policy IAM specifiche che applicano il principio del *Least Privilege*.

8.4 Configurazione Automatizzata di CORS

La configurazione CORS su API Gateway è stata completamente automatizzata. Abbiamo definito risorse specifiche che istruiscono il Gateway a rispondere alle chiamate OPTIONS con gli header appropriati. Inoltre, Terraform calcola dinamicamente l'URL del bucket S3 e lo inietta come *Callback URL* autorizzato in Cognito, eliminando interventi manuali post-deployment.

Capitolo 9

Containerizzazione e Portabilità con Docker

9.1 Il Ruolo di Docker

Sebbene l’architettura di produzione sia Serverless, l’utilizzo di Docker è fondamentale per migliorare la *Developer Experience*. L’obiettivo è la creazione di un ambiente di sviluppo locale robusto e indipendente dal sistema operativo dell’host.

9.2 Simulazione Locale

Abbiamo sviluppato un container per il **Simulatore IoT Locale**. Questo container esegue lo stesso codice Python della Lambda, avvolto da uno script che simula l’invocazione di EventBridge. Le credenziali AWS vengono iniettate come variabili d’ambiente, permettendo al codice locale di scrivere sulle tabelle DynamoDB di sviluppo nel cloud.

9.3 Build Multi-Stage per il Frontend

Per il frontend React, abbiamo adottato una strategia di build Docker **Multi-Stage**:

1. **Builder:** Utilizza un’immagine Node.js per compilare il codice sorgente.
2. **Runner:** Copia solo i file statici compilati in un’immagine Nginx Alpine leggerissima.

Il risultato è un container finale di dimensioni minime, privo di codice sorgente o strumenti di build.

9.4 Orchestrazione con Docker Compose

Tramite `docker-compose`, l’avvio dell’intero stack di sviluppo richiede un singolo comando. Viene creata una rete interna isolata che permette ai contai-

CAPITOLO 9. CONTAINERIZZAZIONE E PORTABILITÀ CON DOCKER21

ner di comunicare in sicurezza, mentre i volumi mappano il codice sorgente per consentire modifiche in tempo reale.

Capitolo 10

Continuous Integration e Continuous Deployment

10.1 Automazione del Ciclo di Rilascio

Per industrializzare il processo di rilascio, abbiamo implementato una pipeline CI/CD con **GitHub Actions**. L'architettura della pipeline è definita come codice (*Pipeline as Code*).

10.2 Sicurezza e Workflow

La pipeline è strutturata in workflow modulari. Un aspetto centrale è la sicurezza: le credenziali AWS non sono nel codice, ma iniettate tramite **GitHub Repository Secrets** nell'ambiente volatile del runner.

10.3 Il Flusso di Deployment

Il processo automatizzato segue una sequenza rigorosa:

1. **Infrastructure:** Provisioning tramite Terraform.
2. **Backend:** Pacchettizzazione e aggiornamento delle funzioni Lambda.
3. **Frontend:** Build dell'applicazione React e sincronizzazione con il bucket S3.

Questo automatismo riduce il tempo di rilascio a pochi secondi ed elimina l'errore umano.

Capitolo 11

Conclusioni e Sviluppi Futuri

11.1 Sintesi del Progetto

Il progetto *Cloud-Native Hospital Monitoring System* ha dimostrato con successo la fattibilità di un'architettura **Serverless** in contesti sanitari critici. La transizione al paradigma **Event-Driven** ha permesso di abbattere la latenza, garantendo una visione “live” dello stato dei pazienti.

11.2 Risultati Raggiunti

Il prototipo ha soddisfatto i requisiti:

- **Reattività:** Aggiornamenti della dashboard in pochi millisecondi.
- **Sicurezza:** Modello Zero-Trust con Cognito e IAM.
- **Industrializzazione:** Infrastruttura replicabile con Terraform e Docker.
- **Costi:** Modello Pay-per-Use efficiente.

11.3 Sviluppi Futuri

L'architettura modulare predispone a evoluzioni future:

1. **AI e Machine Learning:** Integrazione con Amazon SageMaker per la medicina predittiva.
2. **Hardware IoT Reale:** Integrazione di dispositivi fisici tramite AWS IoT Core.
3. **App Mobile:** Sviluppo di un'app nativa con notifiche push.
4. **Disaster Recovery:** Estensione Multi-Regione con DynamoDB Global Tables.

In conclusione, il progetto dimostra come le tecnologie Cloud-Native siano un abilitatore fondamentale per sistemi sanitari più resilienti e sicuri.