Microservice application for green smart cities: Real-Time River Level Monitoring System

1st Flavio Campobasso
dept. Computer Engineering, University of Rome Tor Vergata
Rome, Italy
flavio.campobasso@alumni.uniroma2.eu

3rd Matteo Giustini
dept. Computer Engineering, University of Rome Tor Vergata
Rome, Italy
matteo.giustini.matteo.giustini.01@alumni.uniroma2.eu

2nd Carlo Maria Fioramanti
dept. Computer Engineering, University of Rome Tor Vergata
Rocca Priora, Italy
carlomaria.fioramanti@alumni.uniroma2.eu

***Abstract*—Il cambiamento climatico sta causando eventi meteorologici estremi sempre più frequenti, tra cui inondazioni devastanti dei fiumi. Negli ultimi anni, diverse regioni del mondo hanno assistito a un aumento delle inondazioni causato da forti piogge, innalzamento del livello del mare e altri fattori climatici. Tali eventi hanno messo in evidenza la necessità di un monitoraggio in tempo reale dei livelli fluviali per prevenire danni alle comunità e migliorare la gestione delle risorse idriche. Il progetto presentato si concentra sull'implementazione di un sistema di monitoraggio in tempo reale dei livelli dei fiumi, utilizzando (indirettamente) sensori IoT e tecnologie cloud per raccogliere e analizzare i dati. Il sistema è progettato per monitorare costantemente i livelli idrometrici e fornire allarmi tempestivi in caso di rischi di inondazioni, contribuendo così alla gestione efficiente delle emergenze e alla protezione delle popolazioni vulnerabili. Questo approccio innovativo mira a migliorare la resilienza delle infrastrutture e delle comunità in un contesto di crescente instabilità climatica.**

I. INTRODUZIONE

Il monitoraggio in tempo reale dei livelli dei fiumi può migliorare notevolmente la gestione delle risorse idriche, prevenire disastri naturali e ottimizzare le politiche ambientali. Il nostro progetto si propone di implementare un sistema di monitoraggio dei livelli dei fiumi che sfrutta tecnologie IoT e cloud computing per raccogliere, analizzare e visualizzare i dati in tempo reale. Il sistema prevede l'uso di sensori distribuiti lungo i corsi d'acqua per misurare costantemente i livelli idrometrici e fornire previsioni su eventuali rischi di inondazioni.

L'Internet of Things (IoT) è un concetto che si riferisce all'interconnessione di dispositivi fisici, come i sensori, tramite internet. Questi dispositivi raccolgono e si scambiano dati tra di loro e con altre piattaforme senza la necessità dell'intervento umano. Nel contesto del nostro progetto, i sensori distribuiti lungo i fiumi inviano i dati rilevati per l'elaborazione e l'analisi. Questo consente di monitorare i livelli dei fiumi in tempo reale e di rispondere prontamente a situazioni di emergenza.

Il Cloud Computing, e in questo caso AWS, fornisce l'infrastruttura necessaria, come risorse scalabili e servizi di calcolo potenti, per elaborare, archiviare e gestire questi flussi di dati raccolti. La combinazione di IoT e cloud computing consente di creare un sistema flessibile, scalabile e altamente efficiente per il monitoraggio ambientale, che può essere facilmente adattato alle diverse esigenze e alle condizioni mutevoli, come quelle causate dal cambiamento climatico.

II. BACKGROUND

Per lo sviluppo del progetto è stato utilizzato l'ambiente AWS Academy Learner Lab, che ha permesso l'accesso temporaneo a servizi AWS come DynamoDB, utilizzato per la gestione dei dati utente e delle preferenze, e Amazon S3 per il caching. L'infrastruttura del sistema si basa su un'architettura a microservizi containerizzati con Docker e orchestrati tramite Docker Compose.

Il linguaggio scelto è Python, con Flask per l'esposizione delle API REST. I microservizi comunicano tra loro via HTTP oppure tramite Apache Kafka, utilizzato come sistema di messaggistica asincrona per la gestione delle notifiche. L'interfaccia utente, realizzata come client a riga di comando, consente agli utenti di visualizzare ed utilizzare le funzionalità del sistema.

III. SOLUZIONI DI DESIGN

Il sistema è organizzato secondo un'architettura a microservizi, in cui ciascun componente svolge un ruolo specifico ed è containerizzato in maniera indipendente.

L'utente può effettuare la registrazione, il login, la gestione dei preferiti, la segnalazione manuale delle criticità dei fiumi e la visualizzazione del livello di allerta aggiornato dei sottobacini preferiti. I dati idrometrici vengono acquisiti ciclicamente da una fonte esterna esposta dalla Regione Emilia-Romagna. Una volta raccolti, i dati sono inviati al microservizio di analisi che classifica ogni sottobacino secondo soglie di rischio, assegnate da noi per semplicità: verde (se il livello del fiume è sotto i 3 m), gialla (dai 3 m ai 6 m), arancione (dai 6 m ai 9 m), rossa (dai 9 m ai 12 m).

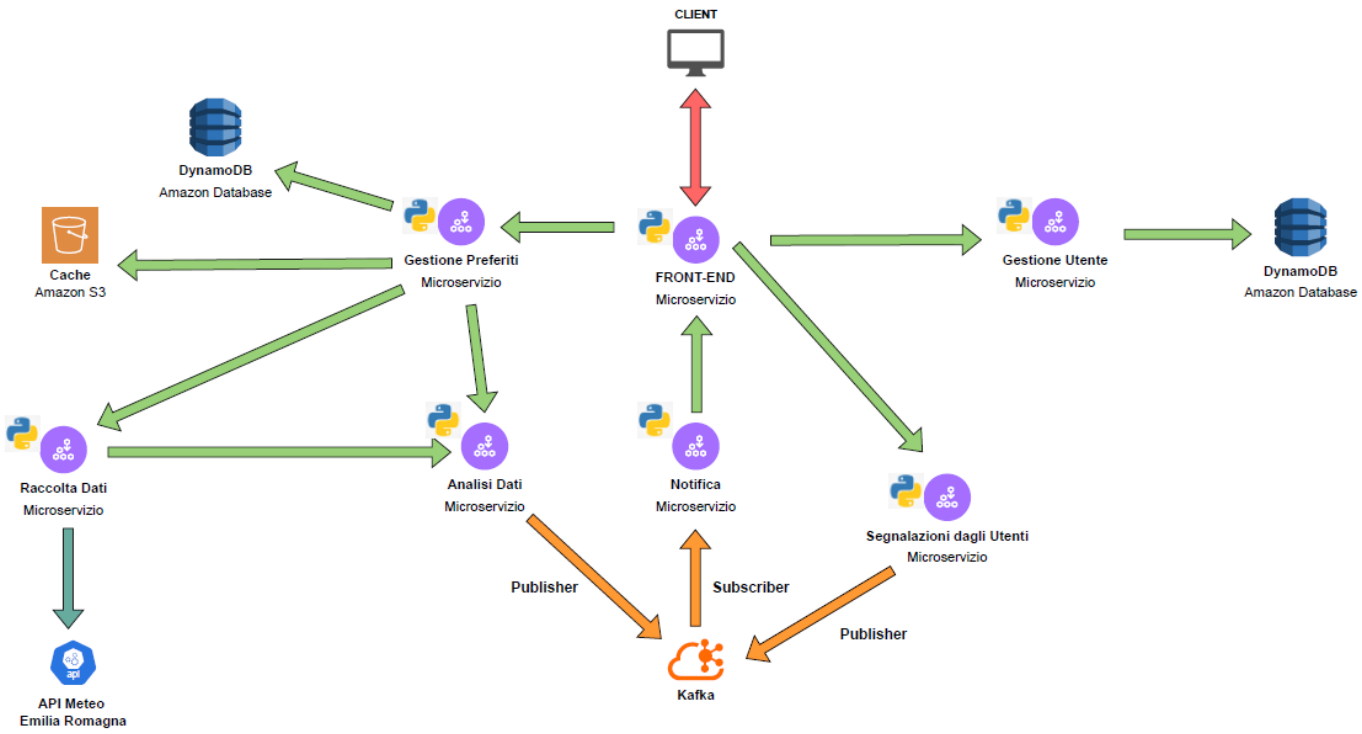

Fig. 1. *Architettura del sistema sviluppato.*

Le soglie critiche (arancione e rossa) attivano un sistema di notifica distribuita, basato su Kafka. Ogni combinazione fiume-sottobacino è associata a un topic specifico: i microservizi inviano notifiche su questi topic, e il microservizio dedicato alle notifiche consuma solo i topic relativi ai preferiti di ciascun utente. Questo approccio evita la persistenza delle notifiche e sfrutta la retention di Kafka per fornire messaggi recenti a richiesta, infatti se un utente fa una richiesta in un momento successivo, riceverà comunque l'ultima allerta rilevata finchè non viene cancellata dalla retention.

Il sistema prevede anche la possibilità per l'utente di segnalare manualmente una condizione critica per un dato sottobacino, generando a sua volta una notifica Kafka nello stesso formato di quelle automatiche. Per quanto riguarda la gestione delle credenziali e delle preferenze degli utenti viene utilizzato come supporto DynamoDB, mentre il controllo di resilienza dei servizi è garantito dall'uso del pattern Circuit Breaker.

IV. DETTAGLI IMPLEMENTATIVI

A. *Microservizi*

Il sistema è composto da più microservizi indipendenti, ciascuno con una responsabilità specifica:

- **raccolta_dati**: interroga periodicamente l'API della Regione Emilia-Romagna per acquisire dati idrometrici aggiornati.
- **analisi_dati**: elabora i dati raccolti e li classifica in fasce di rischio (verde, gialla, arancione, rossa). In caso di superamento delle soglie critiche (arancione, rossa), pubblica una notifica su Kafka.
- **segnalazione_utenti**: consente agli utenti di segnalare manualmente le condizioni critiche dei fiumi, che vengono trattate come notifiche Kafka.
- **notifica**: si iscrive dinamicamente ai topic Kafka relativi ai fiumi preferiti dell'utente e restituisce, su richiesta, le notifiche più recenti.
- **gestione_utente**: gestisce registrazione e login degli utenti, memorizzando le credenziali su DynamoDB con cifratura delle password.
- **gestione_preferiti**: consente di aggiungere, rimuovere e controllare i preferiti. Utilizza **Amazon S3** come sistema di cache per ridurre le chiamate ridondanti all'API meteo. I dati vengono riutilizzati se quelli presenti nella cache risultano essere recenti, evitando chiamate multiple in caso di accessi ravvicinati di più utenti.
- **frontend**: si occupa principalmente di gestire l'interazione dell'utente con l'applicazione. Fornisce un'interfaccia tramite cui gli utenti possono usufruire

delle funzionalità del sistema.

B. Kafka e Topic

Ogni coppia fiume–sottobacino è associata a un topic Kafka, generato dinamicamente tramite lo script `appInizializzazioneTopic.py`, basato su un file JSON dei fiumi. I messaggi Kafka contengono informazioni dettagliate tra cui: fiume, sottobacino, fascia di rischio, timestamp e tipo (sistema o utente). Questo schema consente una gestione fine delle notifiche, ottimizzando la delivery lato utente.

C. Interfaccia Utente

Il frontend è sviluppato come applicazione a riga di comando e, interfacciandosi con i servizi di backend, permette di recuperare, visualizzare e aggiornare i dati. Le richieste sono inviate ai microservizi via HTTP. Per garantire robustezza, ogni interazione è protetta tramite il meccanismo di **Circuit Breaker**, che evita sovraccarichi in caso di malfunzionamenti temporanei dei servizi.

D. Consistenza dei dati con DynamoDB

Il sistema utilizza **Amazon DynamoDB** per la persistenza dei dati. Sono gestite due tabelle principali: `Users`, che contiene le credenziali degli utenti, e `Favorites`, che memorizza le preferenze sui fiumi e sottobacini per ogni utente. Le operazioni di lettura e scrittura sono atomiche per garantire consistenza, e sono protette da *Circuit Breaker* per evitare accessi ripetuti in caso di fallimento temporaneo.

E. Sicurezza tramite cifratura

Per la protezione delle credenziali utente, il sistema utilizza l'algoritmo `bcrypt` per la cifratura delle password prima della loro memorizzazione nel database DynamoDB. Questo algoritmo è progettato per essere computazionalmente costoso, rendendo difficoltosi eventuali attacchi di forza bruta e attacchi a dizionario.

A ogni password viene associato un *salt* generato casualmente, che garantisce che hash identici non possano derivare da password uguali.

Durante la fase di login, la password inserita dall'utente viene nuovamente cifrata e confrontata con l'hash salvato in database tramite la funzione `checkpw` di `bcrypt`, che verifica la validità in modo sicuro senza esporre l'hash originale.

F. Caching con Amazon S3

Per ottimizzare l'efficienza e ridurre il numero di chiamate all'API esterna, il sistema utilizza **Amazon S3** come cache temporanea nel microservizio `gestione_preferiti`. I dati idrometrici ricevuti dalla fonte esterna vengono salvati in un oggetto S3 contenuto in un bucket e riutilizzati entro un intervallo di validità definito. Questo approccio permette di evitare richieste ridondanti in caso di accessi ravvicinati da parte di più utenti, migliorando la latenza e diminuendo il carico sui servizi esterni.

G. Pattern architetturali adottati

Durante lo sviluppo del sistema sono stati adottati alcuni pattern architetturali noti, al fine di garantire robustezza, modularità e una chiara separazione dei contesti di dominio.

a) Circuit Breaker: Per migliorare la resilienza dei microservizi, è stato implementato il pattern **Circuit Breaker**. Questo meccanismo impedisce che un servizio continui a effettuare chiamate ripetute a un'altra componente in errore temporaneo, evitando così di sovraccaricarla ulteriormente. È stato applicato utilizzando la libreria `circuitbreaker` in Python, con una soglia di errore configurata e un timeout per il ripristino automatico.

Tale approccio ha permesso di gestire in modo controllato eventuali disservizi (ad esempio, DynamoDB non disponibile o API esterne non raggiungibili), garantendo maggiore stabilità all'intero sistema.

b) Table per Service: È stato inoltre adottato il pattern **Table-per-Service** per la gestione dei dati in DynamoDB. Ogni microservizio, che ne necessita, dispone di una propria tabella dedicata:

- `gestione_utente` utilizza la tabella `Users` per la registrazione e l'autenticazione degli utenti.
- `gestione_preferiti` usa la tabella `Favorites` per memorizzare i fiumi e sottobacini selezionati dagli utenti.

Questo pattern assicura l'indipendenza tra i microservizi, minimizza il coupling tra componenti, e facilita la scalabilità e l'evoluzione indipendente di ciascun servizio.

H. DynamoDB vs Amazon S3

Amazon S3 (Simple Storage Service) e Amazon DynamoDB sono tra i servizi di tipo IaaS (Infrastructure as a Service) di archiviazione e gestione dati più popolari offerti da Amazon Web Services (AWS), ma sono progettati per scopi diversi.

Amazon S3 è una classe di archiviazione ad alte prestazioni e a zona di disponibilità singola creata appositamente per fornire un accesso coerente ai dati in pochi millisecondi per i dati a cui si accede più frequentemente e per le applicazioni più sensibili alla latenza. S3 può migliorare la velocità di accesso ai dati di 10 volte e ridurre i costi delle richieste del 80%. Progettata per offrire una disponibilità del 99,95% con un **Contratto sul livello di servizio** per la disponibilità del 99,9%, rendendolo ideale per l'archiviazione di grandi volumi di dati non strutturati e per il recupero di oggetti in modo sicuro e rapido. È perfetto per applicazioni che richiedono un archivio permanente e non necessitano di una gestione complessa delle operazioni di lettura e scrittura.

Amazon DynamoDB, invece, è un servizio di database NoSQL completamente gestito da AWS, progettato per offrire prestazioni elevate con bassa latenza e scalabilità automatica. DynamoDB è progettato per eseguire operazioni di lettura e scrittura veloci, con un'alta disponibilità e scalabilità automatica. Questo servizio è ideale per applicazioni che richiedono

operazioni rapide su grandi quantità di dati strutturati o semi-strutturati, come sessioni di utente, carrelli della spesa e dati di cache.

Principali differenze tra Amazon S3 e DynamoDB:

- **Tipo di dati**:
  - **Amazon S3**: È progettato per archiviare oggetti di qualsiasi tipo, come file, immagini, video e backup. Nel nostro caso lo abbiamo utlizzato per archiviare i file .json ottenuti tramite l'API della regione Emilia-Romagna.
  - **Amazon DynamoDB**: Gestisce dati strutturati e semi-strutturati, come coppie chiave-valore, e permette di eseguire query rapide su questi dati. Noi lo abbiamo utilizzato per salvarci i dati degli utenti, `id user`, `username`, `password` e i fiumi preferiti per utente.
- **Scopo e utilizzo**:
  - **Amazon S3**: È più adatto per l'archiviazione di grandi volumi di dati statici, come documenti e media, e per il backup o la distribuzione di contenuti.
  - **Amazon DynamoDB**: È progettato per applicazioni che richiedono operazioni rapide e frequenti di lettura/scrittura sui dati, come giochi e applicazioni mobile..
- **Prestazioni e latenza**:
  - **Amazon S3**: Non è ottimizzato per operazioni frequenti di scrittura e lettura veloci. La latenza nelle operazioni di recupero dei dati può essere più elevata, specialmente quando si lavora con oggetti di grandi dimensioni.
  - **Amazon DynamoDB**: È ottimizzato per prestazioni con bassa latenza e alte prestazioni in lettura e scrittura, rendendolo ideale per operazioni in tempo reale.
- **Costo**:
  - **Amazon S3**: I Crediti di servizio sono calcolati come percentuale delle spese totali pagate dall'utente per la classe di storage Amazon S3 applicabile nella Regione AWS (o, per S3 Express One Zone, nella Availability Zone (AZ) di AWS) interessata per il ciclo di fatturazione in cui la percentuale di tempo di attività mensile è rientrata negli intervalli indicati nella tabella seguente.
  - Per le richieste a S3 Standard, S3 Express One Zone, S3 Glacier Flexible Retrieval, S3 Glacier Deep Archive e tutte le altre richieste non specificate nella **Tabella I**.

TABLE I
PERCENTUALE DI TEMPO DI ATTIVITÀ MENSILE E PERCENTUALE DI CREDITO DI SERVIZIO

Percentuale di tempo di attività mensile	**Credito**
Meno del 99,9% ma $\geq$ 99,0%	10%
Meno del 99,0% ma $\geq$ 95,0%	25%
Meno del 95,0%	100%

V. RISULTATI

Il sistema implementato è stato testato con successo in uno scenario realistico di acquisizione, analisi e notifica basata su dati idrometrici. I test hanno confermato il corretto funzionamento della pipeline end-to-end, dalla raccolta dei dati fino alla consegna personalizzata delle notifiche agli utenti.

Durante le simulazioni, l'utilizzo di **Kafka** si è rivelato efficace nel garantire la consegna asincrona e affidabile dei messaggi. L'approccio basato su topic specifici per ogni coppia fiume–sottobacino ha consentito una *filtrazione precisa* delle notifiche, evitando la necessità di gestire uno storico centralizzato. Il sistema invia notifiche solo agli utenti che hanno espresso un interesse per un determinato bacino, riducendo il traffico superfluo.

L'introduzione di **Amazon S3 come meccanismo di cache** per i dati idrometrici ha ridotto il numero di chiamate all'API esterna, dimostrandosi utile in scenari con accessi multipli ravvicinati da parte di più utenti. Questo ha permesso di alleggerire il carico sul microservizio di analisi e di migliorare la latenza complessiva del sistema.

Il sistema si è mostrato resiliente grazie all'uso del *Circuit Breaker*, che ha impedito la propagazione degli errori in caso di malfunzionamenti temporanei dei microservizi. Anche in presenza di errori simulati, i servizi hanno sospeso le chiamate e si sono riattivati correttamente una volta ripristinata la connessione.

Infine, l'interfaccia a riga di comando ha fornito un'esperienza utente efficace, consentendo un'interazione semplice per la gestione dei preferiti e il recupero delle notifiche, dimostrando l'usabilità del sistema anche in assenza di un'interfaccia grafica.

VI. DISCUSSIONE

Il sistema sviluppato ha evidenziato buone capacità di scalabilità e modularità, grazie all'adozione dell'architettura a microservizi e della messaggistica asincrona basata su Kafka. Tuttavia, lo sviluppo ha messo in luce alcune limitazioni e possibili aree di miglioramento.

L'uso dell'ambiente **AWS Academy Learner Lab** ha imposto restrizioni, come la disponibilità limitata dei servizi e delle regioni, ostacolando la distribuzione geografica dei microservizi. In un contesto produttivo, sarebbe preferibile un deployment su più regioni per aumentare resilienza e disponibilità.

L'uso di **Kafka** ha eliminato la necessità di una base dati per la persistenza delle notifiche, sfruttando la retention per fornire solo i messaggi più recenti. Tuttavia, questa scelta impedisce la conservazione di uno storico delle allerte, che potrebbe essere utile per analisi retrospettive o audit.

L'introduzione di **Amazon S3** come cache nel microservizio di controllo preferiti ha migliorato l'efficienza, riducendo le chiamate ridondanti all'API esterna in presenza di richieste ravvicinate. Attualmente, la gestione della validità dei dati è manuale; un'evoluzione possibile è l'integrazione con un sistema di cache distribuita dotato di TTL automatico, come Redis.

La resilienza del sistema è stata aumentata con l'impiego del *Circuit Breaker*, ma una futura estensione potrebbe prevedere l'uso di *service mesh*, retry intelligenti o logging distribuito per migliorare il monitoraggio e la gestione dei fault.

Infine, la *CLI* si è rivelata utile nella fase di sviluppo e testing. Tuttavia, per un utilizzo su larga scala, sarebbe opportuno affiancare un'interfaccia grafica accessibile via web, più adatta a utenti finali non tecnici.

VII. Deployment e DevOps

L'intero sistema è stato progettato per essere eseguito in locale tramite **Docker Compose**, sfruttando la containerizzazione dei microservizi per garantire isolamento, riproducibilità e semplicità di avvio. Il file `docker-compose.yml` definisce la configurazione completa dell'architettura, inclusi:

- Microservizi (es. `raccolta_dati`, `analisi_dati`, `notifica`, `frontend`, ecc.)
- Servizi esterni come `Kafka`, `Zookeeper`, `DynamoDB local`
- Variabili d'ambiente condivise tramite file `.env`
- Comandi di avvio automatizzati tramite lo script `start.py`, che esegue la sequenza `build` → `up` → `exec`.

Tale configurazione consente a chiunque di avviare il sistema con una sola istruzione, senza dover installare manualmente le dipendenze Python o configurare i servizi esterni.

Evoluzione con Kubernetes

In un contesto di produzione o di maggiore scalabilità, il sistema potrebbe essere migrato su una piattaforma di orchestrazione come **Kubernetes**, beneficiando di:

- **Auto-scaling** dei pod in base al carico (es. per il microservizio di notifica).
- **Gestione centralizzata della configurazione** con ConfigMap e Secret.
- **Monitoraggio e logging distribuito** tramite Prometheus e Grafana.
- **Deploy cloud-native** su piattaforme come Amazon EKS, Google GKE o Minikube in ambienti locali.

L'architettura a microservizi è compatibile con questo tipo di evoluzione, e l'adozione di Kubernetes rappresenterebbe il naturale passo successivo per un deployment affidabile e scalabile in ambienti reali.

VIII. Conclusioni

Il progetto ha permesso la realizzazione di un sistema distribuito per l'allerta idrometrica, strutturato secondo i principi dell'architettura a microservizi e basato su comunicazione asincrona tramite Apache Kafka. La containerizzazione con Docker e l'uso di servizi come DynamoDB e Amazon S3 hanno garantito modularità, portabilità e scalabilità.

Attraverso l'interfaccia a riga di comando, l'utente può interagire con il sistema in modo semplice, ricevendo notifiche personalizzate sui fiumi e sottobacini di interesse, in base a soglie di rischio predefinite o segnalazioni dirette. L'adozione del pattern *Circuit Breaker* ha migliorato la resilienza del sistema, mentre l'uso di Amazon S3 come cache ha ottimizzato il numero di chiamate verso l'API esterna, riducendo la latenza e il carico.

L'architettura è stata pensata per essere estendibile e pronta per un possibile deployment su piattaforme cloud-native come Kubernetes. In prospettiva futura, il sistema potrebbe essere arricchito con una dashboard web, supporto multi-regione e un'analisi storica delle allerte, ampliando il valore applicativo in ambito di Protezione Civile o monitoraggio ambientale.

Il lavoro svolto rappresenta un caso concreto di applicazione dei concetti di sistemi distribuiti, stream processing e cloud computing, con particolare attenzione all'efficienza e alla gestione delle risorse.

Tuttavia il valore del sistema va oltre l'aspetto tecnico: in un'epoca segnata da eventi climatici estremi e sempre più frequenti, uno strumento capace di anticipare i rischi idrologici può fare la differenza nella tutela di vite umane e territori. Questo lavoro non è solo un esercizio accademico: rappresenta un contributo concreto verso città più intelligenti, sicure e resilienti.

Appendix

TABLE II
PRINCIPALI LIBRERIE UTILIZZATE NEL PROGETTO

Libreria	Descrizione
`flask`	Framework per la creazione di API REST in Python.
`confluent-kafka`	Client Kafka per produrre e consumare messaggi.
`requests`	Invio di richieste HTTP verso API esterne e interne.
`boto3`	SDK AWS per interagire con DynamoDB e S3.
`bcrypt`	Cifratura sicura delle password utente.
`dotenv`	Caricamento delle variabili d'ambiente da file `.env`.
`circuitbreaker`	Implementazione del pattern Circuit Breaker nei microservizi.

References

[1] Amazon Web Services, *AWS Documentation*, https://docs.aws.amazon.com/
[2] Apache Kafka, *Kafka Documentation*, https://kafka.apache.org/documentation/
[3] Flask, *Flask Web Framework*, https://flask.palletsprojects.com/
[4] bcrypt, *Password Hashing for Python*, https://pypi.org/project/bcrypt/
[5] Amazon DynamoDB, *NoSQL Database*, https://aws.amazon.com/dynamodb/
[6] Microsoft Azure, *Circuit Breaker Pattern*, https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker
[7] Amazon S3 , *Amazon S3* , https://docs.aws.amazon.com/s3/
[8] Emilia Romagna , *API Emilia Romagna* , https://allertameteo.regione.emilia-romagna.it/sviluppatori