Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Gestione degli allarmi e simulazione degli spostamenti in un sistema di allerta per evacuazione

Elaborato in:
Basi di Dati

Relatore:
Prof.ssa Alessandra Lumini
Correlatore:
Prof.ssa Annalisa Franco

Presentata da:
Beatrice di Gregorio

Sessione II
Anno Accademico 2024-2025

A papà, la stella che non mi ha mai
lasciata sola nei momenti di difficoltà.

Alla mia super mamma che mi ha
insegnato a camminare passo dopo passo,
tenendomi sempre per mano.

Alle mie piccole pesti:
Stefano, Lorenzo, Adele, Mariele,
Caterina, Alma, Matilde e Ambra,
per avermi insegnato cos'è davvero la
felicità e a guardare il mondo con
gli occhi di un bambino.

Introduzione

Il problema della gestione delle emergenze, a causa dei crescenti eventi estremi e dei disastri naturali, è diventato una sfida cruciale per la sicurezza e il benessere delle popolazioni. Eventi come terremoti, inondazioni e incendi richiedono una risposta rapida e coordinata, spesso ostacolata dalla frammentazione delle informazioni e dalla mancanza di un sistema integrato. Le soluzioni esistenti, come i sistemi di allerta a livello territoriale (es. IT-Alert in Italia e IPAWS negli Stati Uniti), sono efficaci per le notifiche su larga scala ma presentano dei limiti: si concentrano sulla comunicazione passiva e non offrono una reazione dinamica e automatizzata che si adatti in tempo reale all'evolversi dell'emergenza. Questa lacuna è ancora più evidente in contesti specifici e confinati, come gli edifici, dove la necessità di una localizzazione precisa e di un ricalcolo dinamico delle vie di fuga è fondamentale per garantire l'incolumità degli occupanti.

Il presente lavoro di tesi si propone di affrontare questa sfida progettando e implementando un sistema di gestione delle emergenze innovativo. L'obiettivo principale è superare le inefficienze dei protocolli di allerta tradizionali creando un'architettura che non si limiti alla semplice notifica, ma che orchestri una risposta intelligente e automatica. L'obiettivo è quello di fornire agli utenti percorsi di evacuazione dinamici e sicuri, calcolati in tempo reale in base alla situazione di pericolo.

Per raggiungere questo scopo, il problema è stato affrontato attraverso la progettazione e l'implementazione di un'architettura a microservizi. Questa scelta metodologica ha permesso di dividere il sistema in componenti indipendenti, rendendolo modulare, scalabile e resiliente. L'architettura proposta integra componenti specializzati per la gestione degli alert, l'elaborazione e la visualizzazione dei dati spaziali e la pianificazione dei percorsi di evacuazione, realizzando un sistema modulare e coerente per il supporto alla gestione delle emergenze. Nello specifico, il mio contributo si è concentrato sullo sviluppo di tre microservizi chiave: il Gestore degli alert, responsabile dell'elaborazione delle allerte; il Simulatore delle posizioni, che ha permesso di validare il sistema in assenza di dati reali; e il Gestore delle posizioni, che valuta il rischio e gestisce le informazioni georeferenziate degli utenti.

I risultati ottenuti, validati attraverso un'analisi sperimentale condotta su un caso di

studio (il Campus universitario di Cesena), dimostrano l'efficacia e l'efficienza del sistema proposto. Le simulazioni, basate su scenari di terremoto e alluvione, hanno confermato la capacità dell'architettura di elaborare le allerte e di fornire percorsi di evacuazione aggiornati in tempo reale, fornendo così un robusto benchmark per futuri sviluppi.

Questo documento è suddiviso in sette capitoli. Il Capitolo 1 analizza il problema e lo stato dell'arte dei sistemi di allerta esistenti. Il Capitolo 2 descrive l'architettura a microservizi proposta e il flusso di gestione dell'emergenza. Il Capitolo 3 illustra le tecnologie fondamentali utilizzate nello sviluppo. I Capitoli 4, 5 e 6 entrano nel dettaglio dell'implementazione dei microservizi di mia diretta responsabilità. Il Capitolo 7 presenta e analizza i risultati sperimentali. Infine le Conclusioni riassumono il lavoro svolto e delineano le future prospettive.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Descrizione del problema affrontato

1.1 Descrizione del problema

Negli ultimi decenni, i cambiamenti climatici e le emergenze ambientali sono diventati una minaccia crescente per la sicurezza e il benessere delle popolazioni. Eventi estremi, come inondazioni, terremoti e incendi boschivi, si verificano con una frequenza sempre maggiore, richiedendo risposte tempestive e coordinate da parte delle autorità competenti. Tuttavia, la gestione di queste emergenze è spesso ostacolata da vari fattori, sia a livello di gestione territoriale, sia in scala ridotta all'interno di singole strutture ed edifici.

In primo luogo, la frammentazione delle informazioni rappresenta un problema significativo. Le diverse agenzie e organizzazioni coinvolte nella gestione delle emergenze possono utilizzare sistemi e protocolli differenti, rendendo difficile la condivisione e l'integrazione dei dati. Questa mancanza di coordinamento può portare a ritardi nelle risposte e a decisioni inefficaci. Analogamente, all'interno di un edificio, la mancanza di un sistema centralizzato per la raccolta e la diffusione delle informazioni può ostacolare una gestione efficace delle emergenze locali, come incendi o guasti strutturali.

In secondo luogo, la comunicazione con il pubblico è spesso carente. Gli avvisi e le notifiche riguardanti situazioni di emergenza non sempre raggiungono le persone in tempo utile o non sono sufficientemente chiari. Questo può portare a confusione e, in ultima analisi, a conseguenze gravi per la sicurezza delle persone. Questo aspetto è cruciale anche in un contesto edilizio, dove sistemi di allarme e comunicazione interna inefficienti possono mettere a rischio l'incolumità degli occupanti.

Infine, la rapidità con cui si evolvono le situazioni di emergenza richiede sistemi flessibili e adattabili. Le attuali soluzioni tecnologiche non sempre riescono a tenere il passo con la velocità degli eventi, rendendo difficile per le autorità rispondere in modo efficace e proattivo. Anche nella gestione delle emergenze all'interno di un edificio, la necessità di sistemi reattivi e capaci di adattarsi rapidamente all'evolversi della situazione (ad esempio,

la propagazione di un incendio o la necessità di evacuazione parziale) è fondamentale.

Queste problematiche evidenziano la necessità di sviluppare un sistema integrato e innovativo che possa migliorare la gestione delle emergenze. Un tale sistema dovrebbe facilitare la raccolta e l'analisi dei dati, migliorare la comunicazione tra le diverse entità coinvolte e garantire che le informazioni cruciali raggiungano il pubblico in modo tempestivo e chiaro. La presente tesi si concentra sulla progettazione e implementazione di un sistema di gestione delle emergenze specificamente pensato per un contesto edilizio. Tuttavia, per comprendere appieno le sfide e le potenzialità di un sistema integrato, si prenderanno in esame anche sistemi di gestione delle emergenze implementati su scala territoriale più ampia, analizzando il loro funzionamento e i protocolli utilizzati, al fine di identificare approcci e tecnologie applicabili anche al contesto specifico di un edificio.

1.2 Analisi dello stato dell'arte

L'analisi dello stato dell'arte rappresenta un passo fondamentale per comprendere il contesto attuale in cui si inserisce il progetto di gestione delle emergenze. In questo paragrafo, esamineremo le soluzioni esistenti e le tecnologie attualmente in uso, evidenziando le loro caratteristiche, i punti di forza e le limitazioni. In particolare, ci concentreremo su due sistemi di allerta pubblica: IT-alert e IPAWS, che rappresentano esempi concreti di come le tecnologie possano essere utilizzate per la comunicazione durante le emergenze.

1.2.1 Common Alerting Protocol (CAP)

Il Common Alerting Protocol (CAP) si configura come uno standard internazionale aperto, definito da OASIS[1] (OASIS Standard, 2010), con l'obiettivo di uniformare il formato dei messaggi di allerta di emergenza per diverse tipologie di pericoli e sistemi di comunicazione. La sua adozione mira a superare la frammentazione informativa, promuovendo l'interoperabilità tra piattaforme di allerta a livello globale. Un messaggio CAP è strutturato per contenere informazioni essenziali sull'emergenza, come la sua natura, la gravità, l'urgenza, l'area interessata (anche con precise coordinate geografiche) e le azioni raccomandate. [1]

In Italia, il sistema di allerta pubblica IT-alert utilizza un profilo specifico del CAP, denominato CAP-IT, adattato al contesto nazionale e alle procedure della Protezione Civile. Sebbene derivato dallo standard OASIS, il CAP-IT presenta specificità legate all'implementazione nel sistema italiano, come i requisiti per la registrazione dei mittenti. [2]

[1] OASIS - Organization for the Advancement of Structured Information Standards

Entrambe le versioni del CAP condividono l'importanza di elementi come la localizzazione geografica degli avvisi e la gestione degli aggiornamenti. Tuttavia, l'adattamento italiano sottolinea come uno standard globale possa essere specificato per rispondere alle esigenze di un determinato contesto. [2]

Per la progettazione del sistema di gestione delle emergenze all'interno dell'edificio oggetto di questa tesi, si farà riferimento al CAP come modello fondamentale per la strutturazione degli avvisi. L'obiettivo è definire un formato di messaggio che, pur essendo specifico per le dinamiche interne di un edificio (come la localizzazione per piani o aree specifiche), possa potenzialmente interoperare con sistemi basati su standard CAP, garantendo una comunicazione efficace e coerente in caso di necessità di integrazione con sistemi di allerta più ampi. Le caratteristiche di flessibilità del CAP nella gestione della localizzazione e della multicanalità saranno particolarmente rilevanti nello sviluppo del microservizio dedicato alla generazione e diffusione degli avvisi interni. [3]

La comprensione del Common Alerting Protocol e delle sue diverse implementazioni, come quella italiana adottata da IT-alert e lo standard OASIS utilizzato come base per sistemi come l'IPAWS statunitense, fornisce un contesto fondamentale per analizzare come questi sistemi gestiscono l'allerta. Nei paragrafi successivi verranno analizzati i due sistemi per evidenziare le loro caratteristiche, i punti di forza e le limitazioni, al fine di identificare le migliori pratiche applicabili al sistema proposto per la gestione delle emergenze all'interno di un edificio. [1][2][4] La Figura 1.1 mette a confronto un esempio di interfaccia di una notifica IT-alert con quella di del sistema statunitense IPWAS.

Figura 1.1: Esempio di notifica IT-Alert e IPAWS

1.2.2 IT-Alert

IT-alert è il sistema nazionale di allerta per la popolazione in Italia, progettato per garantire informazioni tempestive e precise durante situazioni di emergenza. Sviluppato dal Dipartimento della Protezione Civile, questo sistema si basa su una rete di comunicazione integrata che consente alle autorità di inviare avvisi mirati e tempestivi.

Rappresentando un significativo passo avanti nella gestione delle emergenze, IT-alert funge da strumento cruciale per la comunicazione diretta con i cittadini in momenti critici. Il sistema si attiva in risposta a eventi gravi, come incidenti nucleari, attività vulcaniche e il collasso di dighe. Utilizzando la tecnologia cell-broadcast, i messaggi possono raggiungere tutti i dispositivi collegati alle celle telefoniche attive nella zona interessata, garantendo così una diffusione rapida ed efficace, anche in situazioni di congestione della rete. Ogni avviso è caratterizzato da un suono distintivo e dall'etichetta "IT-alert", informando la popolazione su pericoli imminenti e incoraggiando comportamenti di autoprotezione. [2]

Un aspetto fondamentale del sistema è la necessità di aumentare la consapevolezza dei rischi tra gli utenti, supportata da un'educazione continua e da informazioni sui comportamenti appropriati da adottare in caso di emergenza. Inoltre, IT-alert è progettato per operare senza richiedere il download di app o la condivisione di dati personali, rispettando così la privacy degli utenti. La sua implementazione integra le modalità di comunicazione già esistenti, migliorando l'efficacia della risposta alle emergenze attraverso una rete di collaborazione tra le diverse componenti del Servizio nazionale di protezione civile. [5] [6]

Le caratteristiche principali del sistema IT-alert sono:

- Multicanalità: Utilizza diversi canali di comunicazione, come SMS, e-mail e messaggi vocali, per raggiungere un pubblico ampio.
- Localizzazione: Gli avvisi possono essere inviati a gruppi specifici in base alla loro posizione geografica, aumentando la pertinenza delle informazioni.
- Accessibilità: È progettato per essere accessibile a tutti i cittadini, indipendentemente dal tipo di dispositivo utilizzato.

I punti di forza del sistema IT-alert sono:

- Risposta rapida: La capacità di inviare avvisi in tempo reale può salvare vite in situazioni critiche.
- Integrazione con altre piattaforme: IT-alert può collaborare con altri sistemi di emergenza, migliorando la coordinazione e la risposta complessiva.

Le limitazioni dell'IT-alert sono:

- Dipendenza dalla tecnologia: La necessità di dispositivi mobili limita l'accesso per alcune fasce della popolazione, come gli anziani.

- Saturazione delle comunicazioni: Durante eventi di emergenza, l'elevato volume di notifiche può portare a una saturazione, riducendo l'attenzione degli utenti.

In sintesi, IT-alert rappresenta un passo importante verso un sistema di allerta pubblico più moderno e reattivo, contribuendo in modo significativo alla sicurezza e alla protezione dei cittadini. [2][5][6][7]

1.2.3 IPAWS

L'Integrated Public Alert and Warning System (IPAWS) è un sistema statunitense che consente la distribuzione di avvisi pubblici attraverso diverse piattaforme e canali. IPAWS integra vari sistemi di allerta, come il Wireless Emergency Alerts (WEA) e il Emergency Alert System (EAS). [4]

Le caratteristiche principali del sistema IPAWS sono:

- Multicanalità: IPAWS consente la diffusione di avvisi attraverso radio, televisioni e dispositivi mobili, aumentando la probabilità che le informazioni vengano ricevute.

- Coordinamento: Facilita la comunicazione tra diverse agenzie e livelli di governo, garantendo una risposta efficace durante le emergenze.

- Integrazione con EAS: L'EAS è un sistema di allerta pubblico nazionale che consente al Presidente di comunicare con il popolo americano entro dieci minuti durante un'emergenza nazionale. Questo sistema richiede la collaborazione di emittenti radiofoniche e televisive, operatori satellitari e sistemi via cavo.

- Test e Valutazione: La FEMA[2] è responsabile dei test nazionali di EAS, che valutano la prontezza del sistema e la capacità di allerta nazionale in assenza di connettività Internet.

I punti di forza del sistema IPWAS sono:

- Raggiungibilità: L'uso di molteplici canali aumenta la possibilità che il pubblico riceva avvisi critici.

- Standardizzazione: Fornisce un framework unificato per la creazione e la distribuzione di avvisi, migliorando l'efficacia della comunicazione.

[2]FEMA - Federal Emergency Management Agency

- Messaggi EAS: I messaggi possono interrompere la programmazione radio e televisiva per trasmettere informazioni di allerta di emergenza, coprendo una vasta area geografica.

- Avvisi WEA: Brevi messaggi di emergenza inviati da autorità di allerta pubbliche che possono raggiungere dispositivi mobili in aree specifiche, migliorando ulteriormente la sicurezza pubblica. [8]

Le limitazioni dell'IPAWS sono:

- Formazione necessaria: Le autorità locali devono essere adeguatamente formate per utilizzare IPAWS in modo efficace, il che può rappresentare una sfida.

- Problemi di copertura: Non tutte le aree hanno la stessa qualità di ricezione, il che può limitare l'efficacia degli avvisi.

- Relazioni con i Broadcast: È cruciale che le autorità che inviano messaggi EAS abbiano una relazione con le emittenti per comprendere le politiche di trasmissione, che possono variare da stazione a stazione.

- Limitazioni di geo-targeting: Anche se i WEA utilizzano il geo-targeting per inviare avvisi solo a coloro che si trovano all'interno dell'area di allerta, non tutti i dispositivi supportano attualmente questa tecnologia.

In sintesi, IPAWS è un sistema fondamentale per la gestione delle emergenze negli Stati Uniti, progettato per garantire che le informazioni vitali siano comunicate in modo tempestivo e chiaro. Grazie alla sua capacità di integrare vari canali e sistemi, IPAWS rappresenta un modello efficace per la sicurezza pubblica, sebbene ci siano ancora sfide da affrontare per ottimizzare la sua implementazione e l'efficacia degli avvisi. [4]

La Tabella 1.1 presenta un confronto tra IT-alert e IPAWS, evidenziando le loro principali caratteristiche.

Caratteristiche	IT-alert	IPAWS
Paese	Italia	Stati Uniti
Standard CAP	CAP-IT (profilo specifico italiano)	Basato su OASIS CAP 1.2
Tipologia di avvisi	Notifiche su smartphone, SMS, e-mail	Avvisi tramite radio, TV, dispositivi mobili, EAS e WEA
Localizzazione	Avvisi mirati in base alla posizione	Avvisi a livello nazionale e locale, con geo-targeting per WEA
Multicanalità	Sì	Sì
Punti di forza	Risposta rapida, integrazione con altre piattaforme	Raggiungibilità, standardizzazione, integrazione con EAS
Limitazioni	Dipendenza dalla tecnologia, saturazione delle comunicazioni	Necessità di formazione, problemi di copertura, variazione nelle politiche di trasmissione

Tabella 1.1: Confronto tra IT-alert e IPAWS

1.3 Motivazioni dello sviluppo

Come evidenziato nella descrizione del problema, la gestione efficace delle emergenze, sia a livello territoriale che all'interno di singole strutture, rappresenta una sfida complessa. Sebbene sistemi di allerta pubblica come IT-alert e IPAWS abbiano introdotto importanti miglioramenti nella comunicazione e nel coordinamento delle risposte su vasta scala, le loro caratteristiche e protocolli operativi non sono direttamente trasferibili o completamente efficaci in contesti più circoscritti come gli edifici. [4] [5]

L'analisi dei sistemi di allerta nazionali, basati su principi di multicanalità, localizzazione e integrazione, offre spunti preziosi per la progettazione di un sistema di gestione delle emergenze interno a un edificio. Tuttavia, le peculiarità di un ambiente chiuso, con le sue specifiche dinamiche di evacuazione, comunicazione interna e interazione con i sistemi di sicurezza esistenti, richiedono un approccio progettuale dedicato.

Le attuali soluzioni per la gestione delle emergenze in edificio spesso si limitano a sistemi di allarme incendio tradizionali o a protocolli di evacuazione generici, senza sfruttare appieno le potenzialità delle moderne tecnologie di comunicazione e localizzazione precisa.

Questa lacuna evidenzia la necessità di sviluppare un sistema innovativo, specificamente mirato ai contesti edilizi, che possa superare le limitazioni delle soluzioni esistenti e dei protocolli pensati per scenari più ampi.

Le motivazioni principali per lo sviluppo di un sistema di gestione delle emergenze focalizzato su un edificio possono essere così riassunte:

1. Invio di avvisi granulari e in tempo reale:
    - Esistenti: i sistemi di allerta pubblica, pur essendo rapidi nella diffusione su vasta area, non offrono la granularità necessaria per indirizzare avvisi specifici a determinate zone all'interno di un edificio in tempo reale. I sistemi tradizionali interni potrebbero non essere altrettanto dinamici o integrati con informazioni contestuali sull'emergenza.
    - Requisito: un sistema interno deve garantire l'invio immediato di avvisi specifici alle aree interessate dell'edificio, fornendo istruzioni chiare e localizzate per gli occupanti, minimizzando i tempi di risposta e massimizzando l'efficacia delle azioni di autoprotezione.
2. Localizzazione interna precisa:
    - Esistenti: i sistemi di allerta pubblica si basano sulla localizzazione geografica su vasta scala. Le soluzioni tradizionali interne agli edifici potrebbero non disporre di meccanismi di localizzazione precisa degli occupanti in tempo reale.
    - Requisito: è fondamentale integrare funzionalità di geolocalizzazione interna avanzate per inviare avvisi mirati agli occupanti effettivamente presenti nelle aree di pericolo o in quelle che necessitano di evacuazione, evitando allarmi generalizzati e aumentando la pertinenza delle informazioni ricevute.
3. Interfaccia User-Friendly:
    - Esistenti: le interfacce dei sistemi di allerta pubblica sono progettate per raggiungere un vasto pubblico con diversi livelli di familiarità tecnologica. I sistemi interni tradizionali potrebbero avere interfacce limitate o non integrate con i dispositivi personali degli utenti.
    - Requisito: il nuovo sistema deve offrire un'interfaccia semplice, intuitiva e accessibile tramite diversi dispositivi (smartphone, display dedicati, ecc.), facilitando la ricezione e la comprensione delle informazioni di emergenza da parte di tutti gli occupanti, indipendentemente dalla loro familiarità con la tecnologia.
4. Integrazione nativa con i sistemi edilizi esistenti:

- Esistenti: I sistemi di allerta pubblica operano indipendentemente dai sistemi di sicurezza e gestione degli edifici. Le soluzioni interne tradizionali potrebbero non essere completamente integrate con i sistemi antincendio, di controllo accessi o di gestione tecnica.
- Requisito: il sistema proposto deve prevedere una stretta integrazione con gli altri sistemi di sicurezza e gestione delle emergenze presenti nell'edificio (es. rilevazione incendi, controllo accessi, sistemi di ventilazione), consentendo una risposta coordinata e automatizzata agli eventi critici.

5. Protocolli di gestione delle emergenze specifici per l'edificio:

   - Esistenti: i protocolli standardizzati per la gestione delle emergenze sono spesso generici e pensati per scenari su larga scala, senza considerare le specificità di un ambiente confinato come un edificio (es. percorsi di evacuazione interni, punti di raccolta specifici, gestione di persone con mobilità ridotta all'interno della struttura).
   - Requisito: è necessario sviluppare protocolli ad hoc, basati sulle planimetrie dell'edificio, sui percorsi di evacuazione interni, sulla posizione dei dispositivi di sicurezza e sulle procedure specifiche per diverse tipologie di emergenza (incendio, allarme bomba, guasto strutturale), garantendo una gestione più efficace e mirata.

In sintesi, lo sviluppo di un sistema di gestione delle emergenze specificamente progettato per un contesto edilizio è motivato dalla necessità di superare le limitazioni delle soluzioni esistenti e dei protocolli su larga scala. Un sistema integrato, capace di fornire avvisi granulari e in tempo reale, sfruttare la localizzazione interna precisa, offrire un'interfaccia utente intuitiva e integrarsi con i sistemi edilizi esistenti, unitamente a protocolli di gestione delle emergenze specifici per l'edificio, rappresenta un passo cruciale per garantire la sicurezza e la protezione degli occupanti. L'analisi dei sistemi di allerta pubblica come IT-alert e IPAWS fornisce un quadro di riferimento utile per comprendere le potenzialità di un sistema di allerta efficace, ma sottolinea anche la necessità di un adattamento e di uno sviluppo specifico per il contesto degli edifici. [1]

Capitolo 2

Architettura del sistema

Questo capitolo descrive l'architettura a microservizi del sistema di gestione emergenze, soluzione progettata e sviluppata per affrontare le criticità di frammentazione e mancanza di coordinamento identificate nel Capitolo 1. In particolare, si approfondiscono i seguenti aspetti:

- Scelta dell'architettura a microservizi: verrà presentato un confronto tecnico tra l'approccio a microservizi e quello monolitico (Sezione 2.1), motivando la decisione in base a requisiti di scalabilità, manutenibilità e interoperabilità.

- Panoramica generale dell'architettura proposta: sarà fornita una descrizione dell'architettura proposta (Sezione 2.2), illustrando i componenti principali e il flusso dati.

- Analisi dettagliata dei microservizi sviluppati: si procederà con un'analisi approfondita dei microservizi sviluppati (Sezione 2.3), ponendo l'accento sulle funzionalità offerte e le interfacce esposte.

2.1 Scelta dell'architettura a microservizi

L'architettura a microservizi rappresenta un moderno paradigma di sviluppo software che scompone l'applicazione in una costellazione di componenti indipendenti. Ciascun microservizio, progettato per essere autosufficiente e specializzato in una precisa funzionalità, comunica con gli altri attraverso interfacce ben definite. Questa struttura modulare adotta il principio share-nothing, dove ogni servizio opera come processo stateless, garantendo così scalabilità orizzontale e facilità di manutenzione, caratteristiche fondamentali per superare le problematiche di frammentazione informativa e la difficoltà di coordinamento riscontrate nei sistemi tradizionali di gestione delle emergenze. [9]

Per contrasto, l'approccio monolitico consolida tutte le funzionalità in un'unica entità coesa, dove i diversi moduli sono strettamente interconnessi e condividono le stesse risorse di sistema. Sebbene questa architettura tradizionale offra una semplicità iniziale, presenta limitazioni evidenti in contesti che richiedono evoluzione continua e scalabilità differenziata (Figura 2.1). [10]

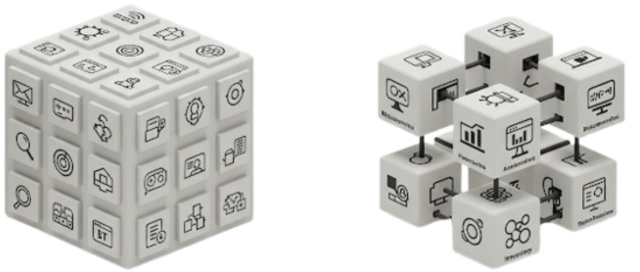

Figura 2.1: Confronto tra Architettura Monolitica e a Microservizi.

L'architettura a microservizi è stata adottata per la progettazione del sistema in esame in virtù delle sue caratteristiche di modularità, scalabilità e resilienza. Questa scelta si basa sull'analisi dei requisiti del progetto, che richiedono un'elevata flessibilità nello sviluppo, una gestione efficiente delle risorse e la capacità di evolvere rapidamente in risposta alle esigenze aziendali. [11] La scelta di questa architettura è motivata da quattro fattori chiave:

1. Modularità e indipendenza: ogni microservizio rappresenta un'unità funzionale autonoma, sviluppabile e distribuibile indipendentemente dagli altri. Questo approccio consente di aggiornare singoli componenti senza impattare l'intero sistema, riducendo i rischi legati al deployment e facilitando la manutenzione.

2. Flessibilità tecnologica: l'architettura a microservizi permette di utilizzare linguaggi di programmazione e database diversi per ciascun servizio, in base alle specifiche esigenze. Tale eterogeneità tecnologica è fondamentale per ottimizzare le prestazioni e sfruttare al meglio le competenze dei gruppi di sviluppo.

3. Scalabilità orizzontale: i microservizi possono essere scalati singolarmente in base al carico di lavoro, ottimizzando l'utilizzo delle risorse. Ad esempio, un servizio ad alta richiesta (come l'autenticazione degli utenti), può essere replicato senza dover scalare l'intera applicazione.

4. Resilienza e fault tolerance: l'isolamento dei servizi limita l'impatto dei guasti per cui un malfunzionamento in un microservizio non compromette l'intero sistema.

La motivazione per l'adozione dell'architettura a microservizi, illustrata nei quattro fattori chiave, trova un riscontro pratico nel confronto con l'architettura monolitica riportato nella Tabella 2.1. Questa tabella affianca le due architetture secondo diverse caratteristiche, offrendo una prospettiva più chiara sui benefici attesi.

Caratteristica	**Microservizi**	**Monolite**
Deployment	Indipendente per ogni servizio	Richiede il riavvio dell'intera applicazione
Scalabilità	Selettiva (solo sui servizi critici)	Globale (anche per componenti non critici)
Tecnologie	Eterogenee (multi-linguaggio, multi-database)	Omogenee (stack tecnologico unico)
Complessità	Gestione distribuita e strumenti avanzati	Centralizzata, ma rischia di diventare ingombrante
Resilienza	Isolamento dei guasti	Fallimento sistemico in caso di errore

Tabella 2.1: Confronto tra Microservizi e Architettura Monolitica

Nonostante i vantaggi, l'adozione dei microservizi introduce alcune complessità. La più rilevante riguarda la gestione della comunicazione: la comunicazione tra servizi (tramite API REST, gRPC o message broker) può introdurre latenza e problemi di sincronizzazione, oltre alla necessità di implementare meccanismi robusti per la gestione della consistenza dei dati in un ambiente distribuito.

In conclusione, la scelta dell'architettura a microservizi è giustificata dalla necessità di costruire un sistema scalabile, flessibile e resiliente, in linea con le moderne esigenze di sviluppo software. Sebbene richieda strumenti avanzati per l'orchestrazione (come Kubernetes) e il monitoring (Prometheus, Grafana), i benefici in termini di agilità e manutenibilità superano gli svantaggi, specialmente in contesti dove la rapidità di evoluzione è critica. [9][10][11][12]

2.2 Panoramica dell'architettura proposta

In questa sezione verrà presentato un diagramma dei microservizi, rappresentato come un grafo orientato, ovvero una struttura dati astratta che permette di modellare le

relazioni tra oggetti. Ogni microservizio è visualizzato come un nodo collegato agli altri componenti attraverso archi. Gli archi hanno lo scopo di mostrare i flussi di dati tra i vari microservizi. Questo diagramma (Figura 2.2) fornisce una visione chiara delle relazioni e delle interazioni tra i vari componenti dell'architettura, evidenziando come ciascun microservizio contribuisca al funzionamento complessivo del sistema di gestione delle emergenze.

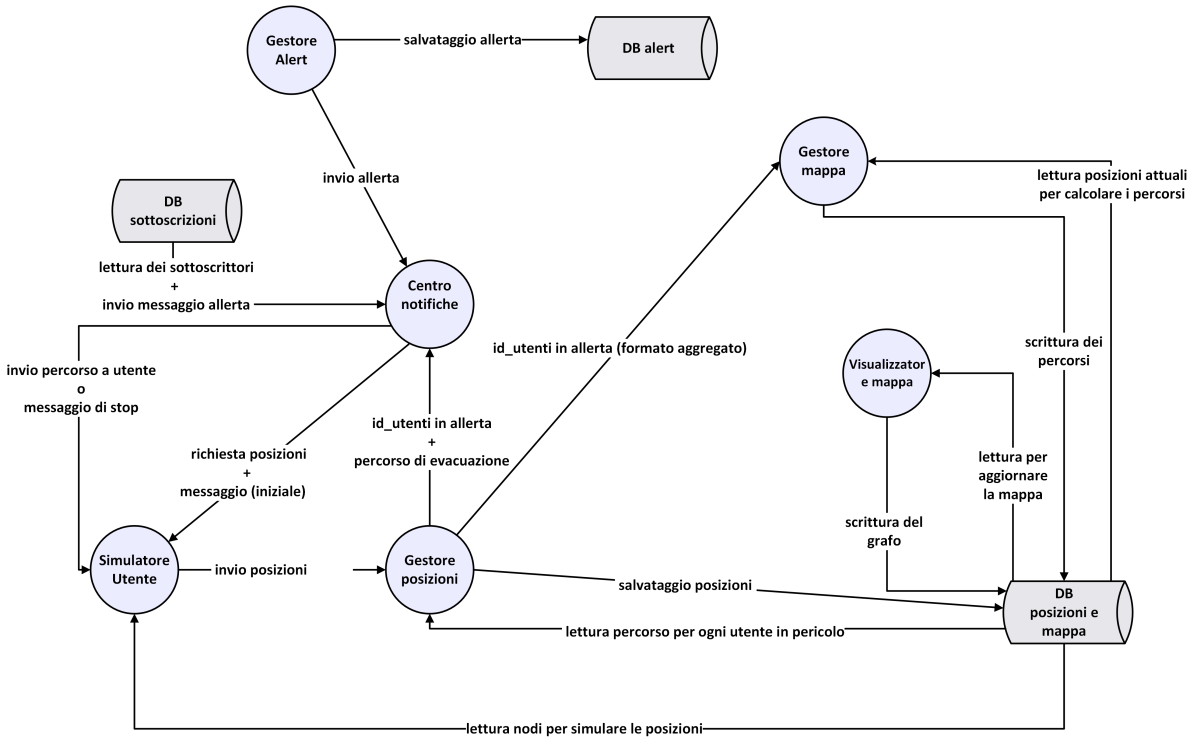

Figura 2.2: Grafo architettura del sistema a microservizi

Il sistema coinvolge diversi attori chiave, tra cui un sistema di generazione allerte, un motore per la gestione della mappa e dei percorsi di evacuazione e un sistema di rilevazione degli utenti in pericolo. Inoltre, come mostrato nel grafo, i dati fondamentali per il funzionamento del sistema sono gestiti attraverso diversi database specializzati. La struttura rappresentata evidenzia la progettazione di un sistema modulare con diversi componenti funzionali dedicati ad attività specifiche.
La rappresentazione dell'architettura sotto forma di grafo, oltre a evidenziare la modularità del sistema e la scelta dei microservizi, serve ad anticipare l'analisi più approfondita dei singoli componenti che verrà condotta nella parte successiva di questo capitolo.

2.3 Design dei microservizi

Questo paragrafo descrive il ruolo e le responsabilità di ciascun microservizio, focalizzandosi sulla loro logica di dominio – ovvero l'insieme di regole, concetti e processi che governano un'area funzionale specifica di un sistema software. Si vuole fornire una visione complessiva del sistema generale, frutto di un lavoro di squadra. Per questo motivo, verranno descritti tutti i microservizi coinvolti. Tale approccio permetterà di comprendere il contesto generale, le scelte progettuali che hanno portato alla risoluzione dei requisiti e il flusso complessivo della gestione delle emergenze. Gli aspetti implementativi, incluse le tecnologie adottate, le scelte progettuali low-level e i dettagli infrastrutturali, verranno approfonditi in seguito. Il Capitolo 3 presenterà la struttura e le scelte implementative generali, mentre i capitoli successivi scenderanno nel dettaglio per ogni microservizio dell'architettura proposta. Si sottolinea che per quanto riguarda l'implementazione, l'attenzione sarà rivolta esclusivamente ai microservizi di mia diretta responsabilità.

2.3.1 Gestore degli alert

Il microservizio Gestore degli alert (*Alert Manager*) rappresenta il componente centrale per la gestione del ciclo di vita degli allarmi all'interno dell'ecosistema software. Progettato per garantire un'elaborazione continua degli eventi critici, il servizio si occupa dell'intera catena di valore degli alert, dalla fase iniziale di generazione fino alla distribuzione finale delle notifiche.

Nell'ambito del suo dominio funzionale, l'*Alert Manager* assolve a due responsabilità fondamentali:

- Generazione strutturata ed elaborazione intelligente dei messaggi di allerta, conformemente ai protocolli standard di settore e alle specifiche tecniche del sistema.

- Valutazione contestuale della rilevanza degli eventi mediante apposite regole di business, con conseguente archiviazione selettiva nella base dati dedicata.

Il servizio non si limita alla mera gestione interna degli alert. Infatti, svolge un ruolo attivo nell'ecosistema a microservizi attraverso precise interazioni con altri componenti specializzati. In particolare, questa interazione è cruciale per la diffusione coordinata delle notifiche.

Per una trattazione esaustiva degli aspetti implementativi - tra cui le modalità di integrazione con il layer di persistenza, gli algoritmi di valutazione degli eventi e i meccanismi di comunicazione interservizio - si rimanda al Capitolo 4.

2.3.2 Centro notifiche

Il microservizio Centro notifiche (*Notification Center*) costituisce il nucleo coordinatore della comunicazione di emergenza all'interno del sistema. Garantisce la corretta distribuzione degli avvisi agli utenti e tutta la comunicazione interna del servizio.

La sua logica di dominio si concentra su:

- Gestione integrata del flusso di notifiche, dall'identificazione dei destinatari alla diffusione degli avvisi.
- Interazione con gli altri servizi per garantire una comunicazione contestuale e geolocalizzata.

Come componente centrale dell'ecosistema, il servizio assicura che ogni messaggio raggiunga i destinatari appropriati con le modalità e i contenuti rilevanti al contesto specifico, mantenendo coerenza operativa in tutto il sistema. Il suo scopo è ricevere e smistare le notifiche in modo corretto per garantire una corretta gestione dell'emergenza.

2.3.3 Simulatore delle posizioni

Nel contesto dell'attuale fase progettuale, il microservizio Simulatore delle posizioni (*User Simulator*) assume una funzione di particolare rilievo, seppur transitoria. Questo componente è stato concepito per emulare, con rigore metodologico, il comportamento e la distribuzione spaziale degli utenti all'interno degli ambienti oggetto di monitoraggio.

Va precisato che tale soluzione rappresenta un'astrazione temporanea, destinata a essere sostituita nel sistema definitivo da un meccanismo di rilevamento in tempo reale delle posizioni effettive degli utenti. Tuttavia, nella presente implementazione, il simulatore assolve a un ruolo fondamentale:

- Fornisce un modello dinamico e verosimile della presenza umana.
- Garantisce la coerenza dei flussi informatici con gli altri servizi.
- Consente la validazione delle logiche di emergenza.

La scelta di implementare questo servizio di appoggio deriva dalla necessità di:

1. Testare in ambiente controllato le dinamiche di notifica.
2. Verificare l'integrazione tra i vari componenti.
3. Sviluppare in parallelo i moduli dipendenti dalla geolocalizzazione.

Pur nella sua natura provvisoria, il simulatore si configura come un elemento cardine dell'attuale architettura, permettendo di valutare con precisione l'efficacia del sistema nelle diverse condizioni operative. Per una trattazione più approfondita delle scelte progettuali relative a questo componente, si rimanda al Capitolo 5.

2.3.4 Gestore delle posizioni

Il Microservizio Gestore delle posizioni (*Position Manager*) rappresenta un elemento architettonico cardine all'interno del sistema distribuito, assumendo la completa responsabilità dell'elaborazione e dell'interpretazione dei flussi di dati geospaziali. La sua progettazione aderisce rigorosamente ai principi di separazione delle competenze (separation of concerns) e di singola responsabilità (single responsibility). Questi principi garantiscono una netta distinzione tra le diverse funzionalità del sistema e assicurano che ciascun componente sia dedicato a un'unica funzione specifica. Tali scelte progettuali consentono una gestione precisa ed efficace della logica di dominio, con particolare riferimento all'analisi spaziale e alla valutazione del rischio. Un aspetto fondamentale di questa valutazione del rischio è la verifica continua dello stato di sicurezza degli utenti. Il Position Manager monitora costantemente i dati geospaziali per determinare se persistono condizioni di pericolo.

La logica di dominio si esprime principalmente attraverso l'analisi spaziale, ovvero l'elaborazione dei dati relativi alla posizione e al movimento, e la conseguente valutazione del rischio, che identifica e classifica le potenziali situazioni di pericolo.

Il servizio è strutturato per assolvere a due funzioni fondamentali, ciascuna delle quali contribuisce alla corretta gestione delle informazioni posizionali e alla risposta a potenziali situazioni critiche:

- Valutazione delle condizioni di rischio mediante l'applicazione sistematica di regole di dominio predefinite, utili a identificare situazioni potenzialmente pericolose e a classificare gli utenti in base alla loro esposizione a criticità spaziali. Questa valutazione include anche la verifica periodica per accertare se le condizioni si pericolo sono cessate per tutti gli utenti.

- Generazione di risposte coordinate in seguito all'identificazione di condizioni anomale, supportando le successive fasi di intervento. Inoltre, una volta accertato che non sussistono più utenti in pericolo, è responsabile della comunicazione di tale condizione, innescando la gestione della conclusione dell'emergenza.

La centralizzazione della logica spaziale all'interno di questo microservizio realizza un'astrazione coerente e univoca dei concetti di posizione e pericolo, ottimizzando l'integra-

zione con le altre componenti sistemiche e garantendo la massima consistenza semantica nell'elaborazione dei dati.

Per un'esposizione completa delle modalità di interazione con le altre componenti del sistema e delle strategie adottate per garantire l'efficienza operativa, si rimanda al Capitolo 6.

2.3.5 Visualizzatore della mappa

Il Microservizio Visualizzatore della mappa (*Map Viewer*) rappresenta la componente specializzata nell'interpretazione e rappresentazione grafica dell'ecosistema monitorato. La sua concezione architettonica incarna i principi fondamentali della domain-driven design, focalizzandosi esclusivamente sull'astrazione e visualizzazione delle informazioni georeferenziate.

Il microservizio assolve a due compiti essenziali nell'ambito dell'architettura complessiva:

- Acquisizione e trasformazione dei dati spaziali: riceve e processa i flussi informativi relativi alle posizioni, convertendoli da formati tecnici grezzi in una rappresentazione strutturata adatta alla visualizzazione.

- Generazione della rappresentazione dinamica: produce una visualizzazione interattiva che si aggiorna in tempo reale, mantenendo una perfetta corrispondenza con lo stato attuale del sistema. La mappa risultante incorpora funzionalità di esplorazione spaziale che permettono di analizzare la distribuzione delle entità a diversi livelli di dettaglio.

La soluzione proposta manifesta tre qualità essenziali che ne definiscono il valore operativo nell'ecosistema architetturale:

- Capacità rappresentativa: garantisce l'inclusione di tutte le dimensioni spaziali significative per il dominio applicativo, traducendosi in una visualizzazione esaustiva che non tralascia alcun aspetto rilevante della distribuzione geografica delle entità.

- Tempestività della rappresentazione: riflette con precisione temporale ogni variazione dello stato del sistema, assicurando una perfetta sincronizzazione tra la realtà dinamica e la sua rappresentazione grafica.

- Interazione spaziale avanzata: abilita modalità sofisticate di navigazione e interrogazione della mappa, supportando operazioni complesse di esplorazione dei dati e permettendo all'utente finale di investigare la distribuzione spaziale a diversi livelli di granularità e secondo molteplici prospettive analitiche.

2.3.6 Gestore della mappa

Il Microservizio Gestore della mappa (*Map Manager*) rappresenta il componente specializzato nell'elaborazione intelligente delle strategie di evacuazione all'interno dell'architettura distribuita. La sua progettazione incarna i principi fondamentali dei sistemi decisionali adattivi, focalizzandosi sull'analisi contestuale e sulla generazione di soluzioni ottimizzate per la gestione delle emergenze.

Il servizio esplica una funzione duale nell'ecosistema applicativo:

- Integrazione dei dati contestuali: sintetizza molteplici fonti informative, tra cui i flussi di posizionamento in tempo reale e la rappresentazione topologica dell'ambiente strutturata come grafo navigabile. Questa integrazione consente una comprensione globale dello stato corrente del sistema, considerando sia la distribuzione delle entità che le caratteristiche infrastrutturali.

- Generazione di strategie di evacuazione: applica algoritmi avanzati di analisi dei percorsi che incorporano vincoli dinamici, tra cui metriche di capacità residua e indicatori derivati da segnalazioni. L'approccio adattivo garantisce la continua rielaborazione delle soluzioni in risposta all'evoluzione delle condizioni ambientali.

La natura specializzata del componente si riflette nella sua capacità di coniugare precisione analitica e tempestività operativa. Il sistema dimostra particolare efficacia nella gestione di scenari complessi, dove la simultanea considerazione di molteplici fattori (dalla densità di occupazione alle condizioni infrastrutturali) permette di generare soluzioni sia tecnicamente rigorose che praticamente attuabili. L'integrazione di questo microservizio nell'architettura complessiva garantisce un approccio sistematico alla gestione delle emergenze, mantenendo un'astrazione completa dalle complessità implementative.

2.4 Comunicazione tra microservizi: flusso dell'emergenza

L'architettura distribuita implementa un sofisticato meccanismo di coordinamento interservizi per la gestione degli eventi emergenziali, basato su un modello a eventi e stato condiviso. Il flusso informativo si articola attraverso una serie di interazioni ben definite tra i vari componenti, mantenendo un rigoroso disaccoppiamento tra le responsabilità di ciascun microservizio.

2.4.1 Fase 1: Inizio della gestione dell'emergenza

Il processo ha inizio quando l'*Alert Manager* genera e valida una potenziale situazione critica. Dopo aver applicato i necessari filtri di rilevanza basati su regole di dominio, il servizio procede alla persistenza dell'evento nel repository dedicato e alla notifica al *Notification Center*, includendo l'identificativo univoco dell'alert e i metadati rilevanti. Questa comunicazione segnala l'attivazione del protocollo di emergenza all'intero ecosistema. Il *Notification Center*, una volta ricevuta la segnalazione, recupera dal database delle sottoscrizioni la lista aggiornata degli utenti interessati e provvede all'invio delle notifiche iniziali attraverso i vari canali configurati. Contestualmente, richiede al *User Simulator* le posizioni correnti degli utenti, avviando così la fase di analisi geospaziale della situazione. Per chiarire la sequenza degli eventi che avvengono durante la fase iniziale di gestione dell'emergenza, la Figura 2.3 presenta un diagramma di sequenza che delinea le comunicazioni tra i principali microservizi coinvolti.

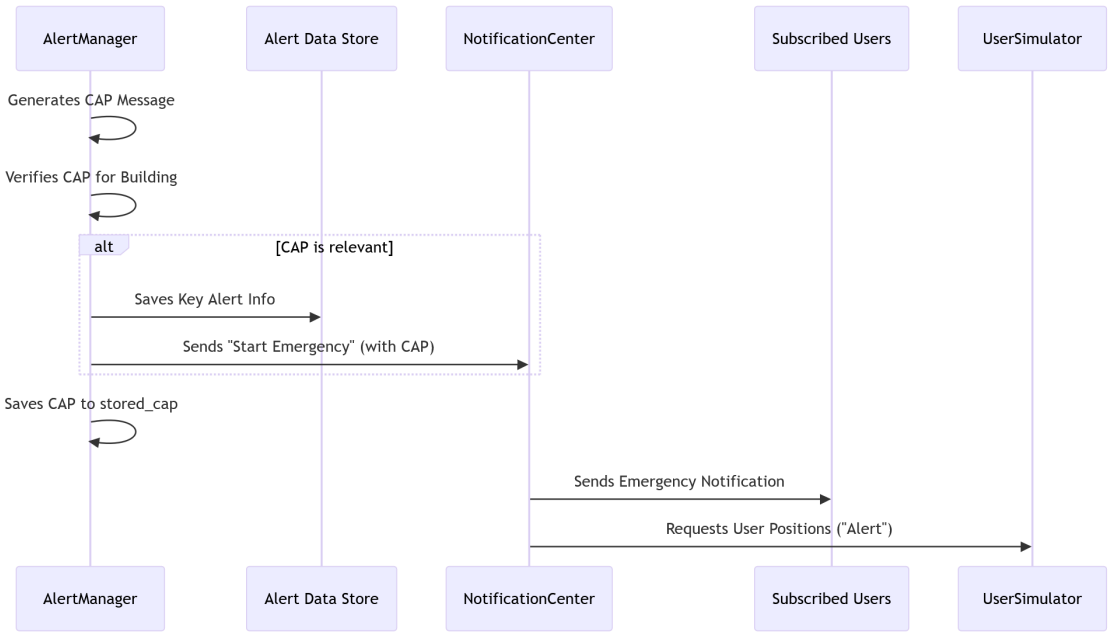

Figura 2.3: Sequence diagram sull'inizio della gestione dell'emergenza

2.4.2 Fase 2: Analisi continua del pericolo e notifiche

In questa fase lo *User Simulator* procede con la simulazione delle posizioni degli utenti all'interno dell'edificio e le comunica al *Position Manager* indicando le informazioni relative all'utente (identificato da un id univoco) e le informazioni relative al tipo di evento in corso. Il *Position Manager*, ricevuti i dati di localizzazione, esegue una validazione topologica delle coordinate e identifica gli utenti potenzialmente esposti al rischio. I risultati di questa analisi vengono comunicati parallelamente a due componenti: al *Notification*

Center, che riceve l'identificativo dell'utente in pericolo e il percorso di evacuazione relativo, e al *Map Manager*, al quale viene fornita una vista aggregata per nodi topologici. Nel caso in cui nessun utente venga identificato in pericolo, il *Position Manager* notifica il *Notification Center* inviando un messaggio di Stop. Tale notifica indica la fine della gestione dell'emergenza.
Questo flusso di interazioni è illustrato nel dettaglio nel diagramma di sequenza presentato in Figura 2.4.

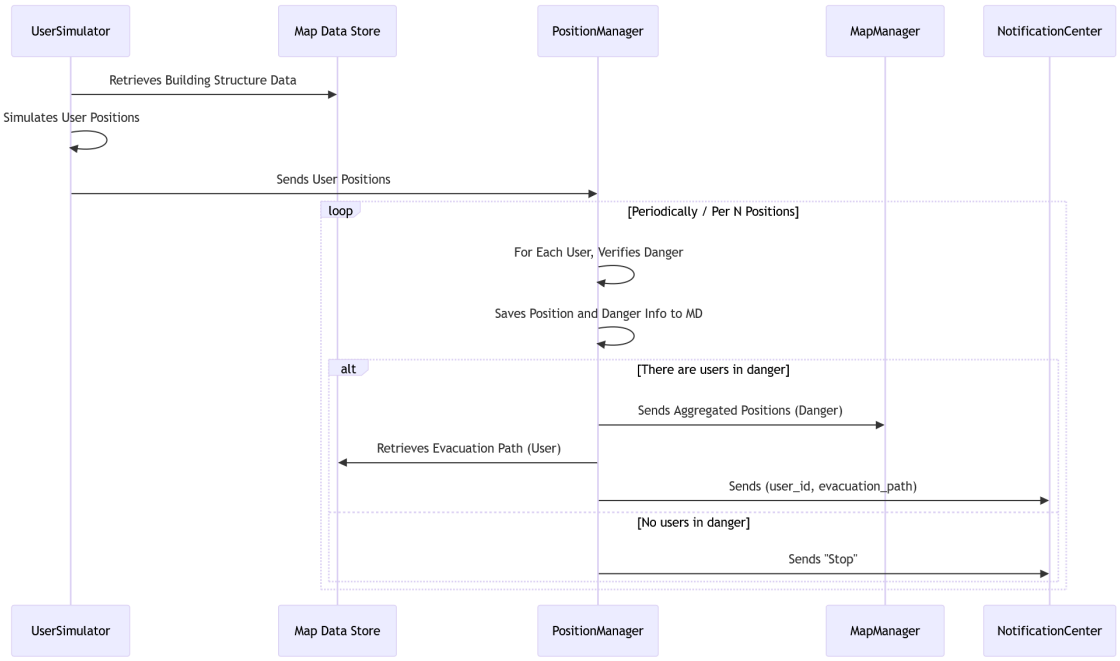

Figura 2.4: Sequence diagram sull'analisi del pericolo

2.4.3 Fase 3: Reazione alle notifiche di evacuazione e aggiornamento delle posizioni simulate

Questa fase descrive come il *Notification Center* inoltra i messaggi ricevuti dal *Position Manager* allo *User Simulator*. A seconda del messaggio (percorso di evacuazione o 'Stop'), lo *User Simulator* simula il movimento dell'utente seguendo il percorso ricevuto o interrompe la simulazione. Nel caso in cui si procede continuando a simulare lo stato di emergenza, lo *User Simulator* invia le nuove posizioni degli utenti al *Position Manager*.
Questo flusso di interazioni è illustrato nel diagramma di sequenza in Figura 2.5 nella quale è ben visibile la risposta differente dello *User Simulator* sulla base del messaggio ricevuto.

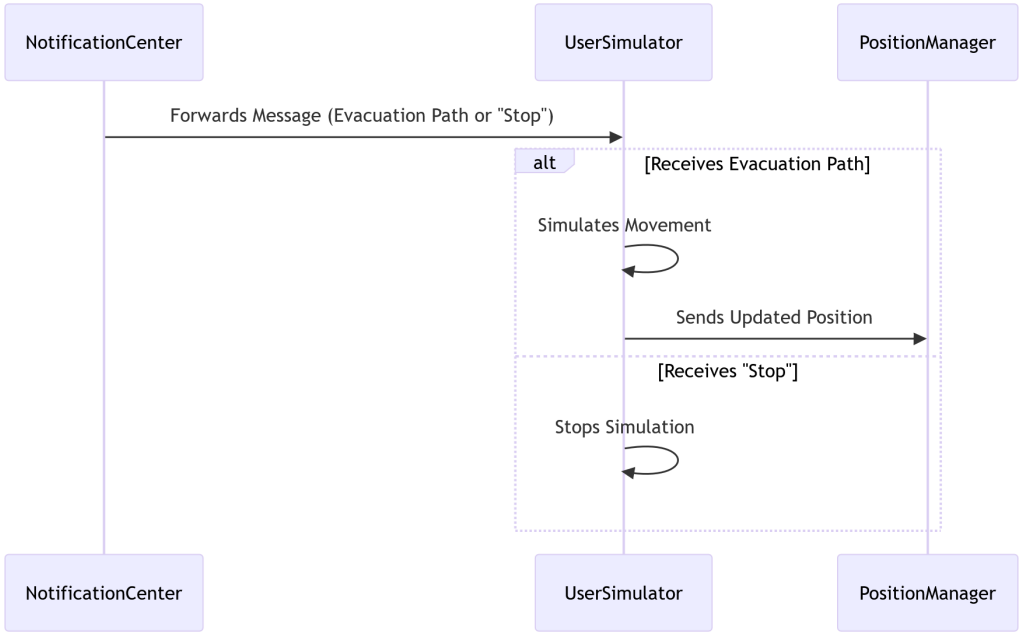

Figura 2.5: Sequence diagram sulle reazioni alle notifiche di evacuazione

2.4.4 Fase 4: Aggiornamento della mappa e ricalcolo dei percorsi

L'aggiornamento della rappresentazione visiva è gestito attraverso un meccanismo di trigger temporali, attivati dal *Position Manager* in base alla frequenza degli aggiornamenti ricevuti. Quando scattano, questi trigger inducono il *Map Viewer* a recuperare le ultime posizioni disponibili e a generare una nuova visualizzazione dello stato del sistema. Parallelamente, il *Map Manager*, ricevute le informazioni dal *PositionManager*, avvia il calcolo dei percorsi di evacuazione ottimizzati, tenendo conto di molteplici fattori, tra cui la capacità residua dei percorsi, le eventuali interruzioni rilevate e i vincoli strutturali dell'ambiente. Per ciascun nodo vengono generate diverse alternative, salvate nel repository condiviso delle posizioni e mappe.
Nella figura 2.6 viene mostrato come comunicano i vari servizi in questa fase e viene evidenziato come l'aggiornamento della visualizzazione della mappa viene ripetuto durante tutta la gestione dell'emergenza in modo da fornire una visione in tempo reale della situazione all'interno dell'edificio.

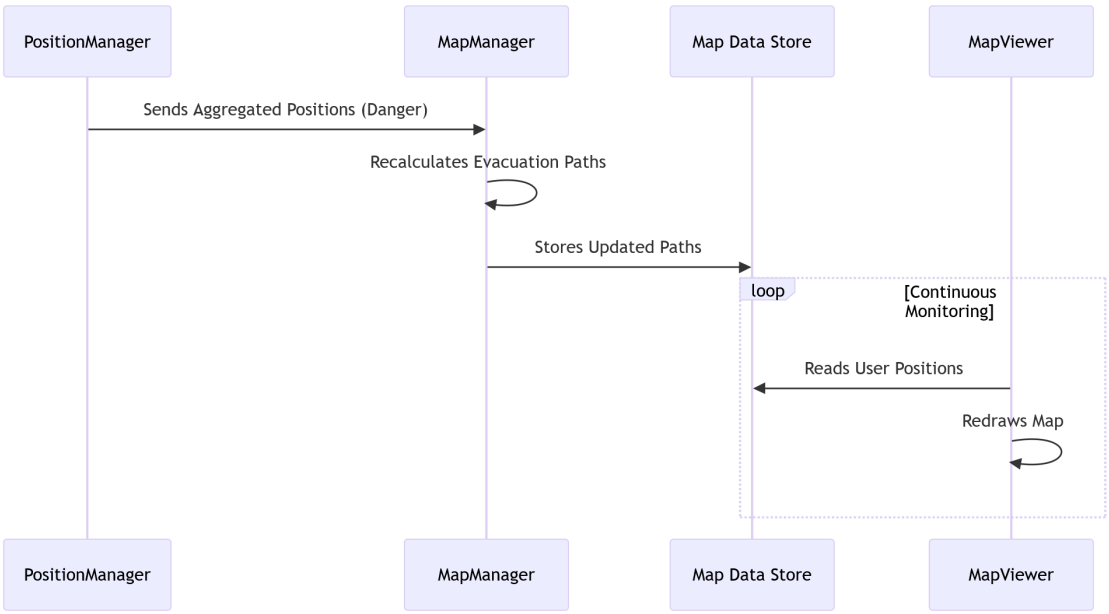

Figura 2.6: Sequence diagram sull'aggiornamento della mappa e dei percorsi

2.4.5 Fase 5: Riassegnamento delle rotte

Il *Position Manager* gioca un ruolo cruciale nel ciclo di gestione. Ogni volta che riceve la posizione di un utente, recupera il percorso di evacuazione assegnato basandosi sulla posizione ricevuta. Questi percorsi di evacuazione non sono statici, ma vengono aggiornati dinamicamente dal *Map Manager* che ha la responsabilità di ricalcolare e mantenere aggiornati i percorsi seguendo le regole di evacuazione e tenendo conto di eventuali aree non accessibili. Di conseguenza, il *Position Manager* utilizza i percorsi aggiornati per riassegnare gli utenti alle rotte. Le istruzioni risultanti vengono poi inoltrate al *Notification Center* per la diffusione finale agli utenti interessati, come descritto precedentemente. La Figura 2.7 mostra l'aggiornamento dinamico dei percorsi e il riassegnamento delle rotte.

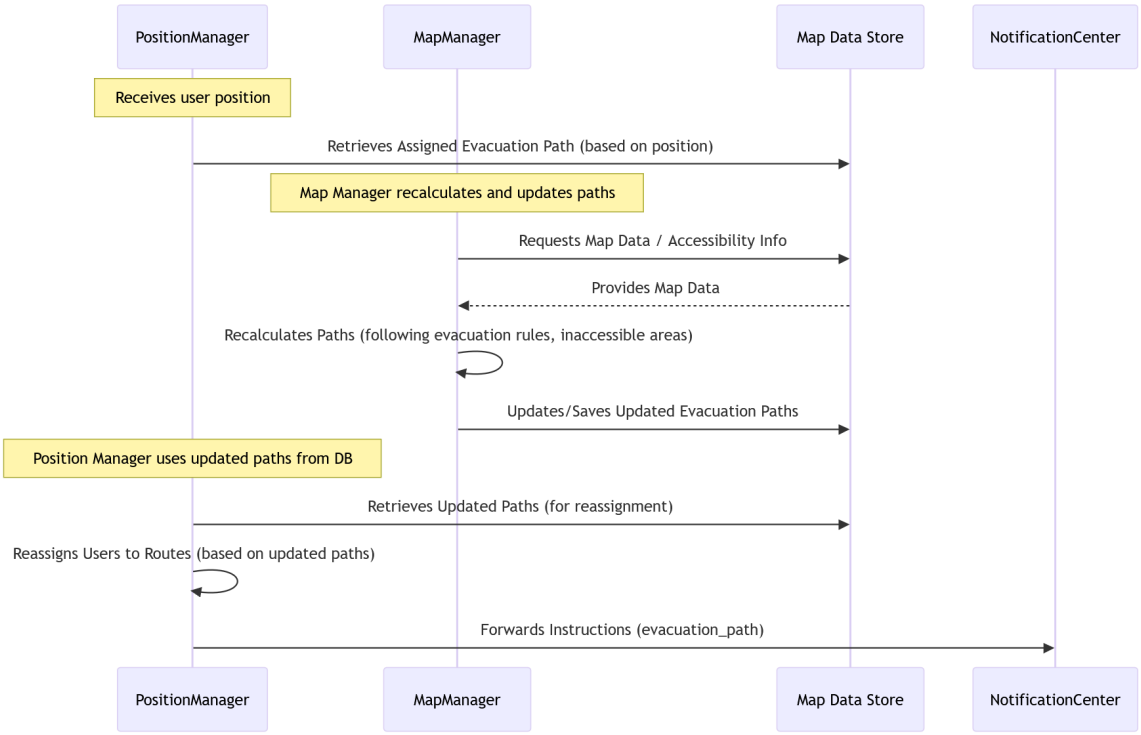

Figura 2.7: Sequence diagram sul riassegnamento delle rotte

2.4.6 Fase 0: Configurazione del sistema

Dopo aver delineato il flusso operativo del sistema di gestione delle emergenze, è fondamentale considerare anche la fase di configurazione iniziale. Questa fase, sebbene non rappresenti un'operazione continua o parte del ciclo di gestione dell'emergenza vero e proprio, è cruciale per predisporre l'ambiente e i dati necessari al corretto funzionamento di tutti i componenti. In particolare, la configurazione del sistema include l'inizializzazione delle informazioni relative alla struttura dell'edificio, che sono essenziali per la simulazione delle posizioni degli utenti e per il calcolo dei percorsi di evacuazione. Consiste nella rappresentazione della mappa dell'edificio mediante un grafo composto da nodi ed archi. Tale configurazione avviene attraverso un'interfaccia dedicata che permette di inserire i nodi, che rappresentano le stanze di un edificio e che vengono classificate mediante una tipologia, e gli archi che rappresentano i collegamenti tra le stanze. Si tratta di una fase cruciale in quanto l'intero sistema si basa poi sulla rappresentazione astratta dell'edificio. Queste informazioni, infatti, saranno utilizzate non solo dal *Map Manager* per definire i percorsi di evacuazione, ma anche dallo *User Simulator* per simulare gli utenti all'interno dell'edificio e dal *Position Manager* per inserire le posizioni degli utenti nel database e per verificare se un utente è in pericolo sulla base del tipo di emergenza che si sta gestendo. Il diagramma di sequenza presentato di seguito (Figura 2.8) illustra i passaggi chiave di

questa fase preparatoria, focalizzandosi sulle interazioni che coinvolgono il *Map Viewer* e il sistema di memorizzazione dei dati della mappa.

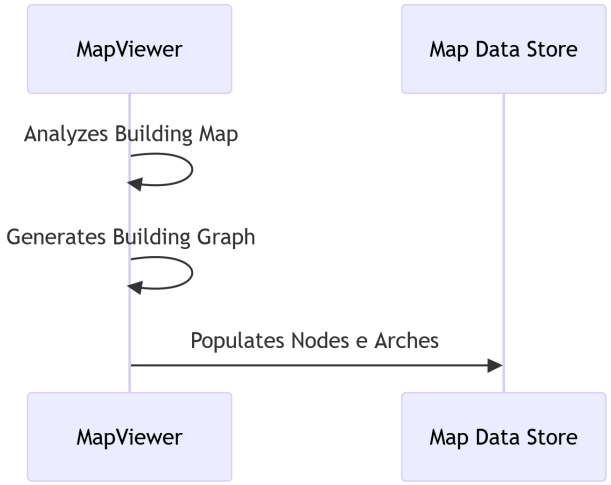

Figura 2.8: Sequence diagram sulla configurazione iniziale del sistema

Capitolo 3

Tecnologie fondamentali del sistema

Questo capitolo esplora le tecnologie fondamentali che costituiscono l'infrastruttura portante dell'intero sistema a microservizi proposto. Dopo aver discusso l'architettura generale e i suoi elementi concettuali, l'analisi si concentra ora sulle scelte concrete che definiscono il funzionamento di ogni singolo componente software. In questo capitolo, l'attenzione è rivolta alle decisioni di fondo che definiscono le capacità operative di base del sistema nella sua interezza.

Il percorso intrapreso analizzerà a fondo le motivazioni che hanno portato alla scelta di specifiche soluzioni: il linguaggio di programmazione utilizzato per lo sviluppo del codice, il sistema che orchestra la comunicazione asincrona tra i servizi, la soluzione adottata per la persistenza dei dati e l'approccio alla gestione della configurazione. Per ciascuna di queste aree, verranno ripercorse le opzioni considerate durante la fase di progettazione, mettendo in luce i pro e i contro che hanno inclinato l'ago della bilancia verso la scelta finale.

L'intento di questo capitolo è duplice:

- articolare un ragionamento solido e trasparente dietro le scelte tecnologiche del sistema, illustrando le motivazioni che hanno portato all'adozione di Python, RabbitMQ, PostgreSQL e YAML.

- costruire un quadro di riferimento tecnologico comune, essenziale per i capitoli successivi dedicati all'implementazione dei singoli microservizi. Questo permetterà di evitare ripetizioni e di focalizzarsi sulle specifiche di ogni componente.

In sintesi questo capitolo definisce le fondamenta tecnologiche su cui poggia l'intero edificio del nostro sistema. Attraverso un'analisi delle scelte e delle loro motivazioni, si mira a fornire una visione chiara e profonda delle decisioni che ne determinano l'essenza operativa a un livello fondamentale.

3.1 Linguaggio di programmazione: Python

La selezione del linguaggio di programmazione ha rappresentato una delle decisioni architetturali più ponderate per lo sviluppo del nostro sistema a microservizi. L'esigenza era quella di individuare uno strumento che combinasse potenza e flessibilità e che avesse una curva di apprendimento accessibile per il team. Secondo la nostra analisi, un linguaggio di programmazione che rispettasse tali esigenze, avrebbe consentito una gestione efficiente delle complessità intrinseche della nostra architettura. La scelta finale è stata quella di adottare Python come linguaggio principale dell'intero sistema.

Questa decisione è stata guidata da diverse considerazioni pratiche e tecniche. Innanzitutto, Python si è rivelato estremamente adatto per la gestione di diversi formati dati e configurazioni. Non solo ha semplificato notevolmente l'elaborazione dei file XML, cruciali per lo standard CAP (Common Alerting Protocol) utilizzato da microservizi come l'*Alert Manager*, ma ha anche offerto una solida capacità nella gestione dei file di configurazione in formato YAML. Questi ultimi sono ampiamente impiegati in tutti i microservizi per definire regole operative e parametri di sistema. La facilità di interazione offerta da librerie come PyYAML ha ottimizzato i processi di configurazione e manutenzione. [13]

Un altro fattore determinante è stato il supporto robusto di Python per l'interazione con i sistemi di persistenza dati. Per un sistema come il nostro, che necessita di gestire e analizzare informazioni geografiche complesse, la capacità di interfacciarsi in modo efficiente con un database relazionale dotato di funzionalità spaziali era un requisito imprescindibile. In questo contesto, Python si è dimostrato particolarmente efficace nel connettersi a PostgreSQL e, in particolare, nello sfruttare appieno l'estensione PostGIS. Quest'ultima è cruciale per la memorizzazione, l'interrogazione e la manipolazione di dati geospaziali, un elemento fondamentale per diverse componenti del sistema che elaborano posizioni, aree di interesse e percorsi. Le librerie dedicate di Python hanno garantito prestazioni affidabili e una notevole flessibilità, permettendo ai nostri microservizi di accedere e gestire queste informazioni geografiche con precisione e velocità.

Durante il processo di valutazione, abbiamo attentamente considerato anche alternative come Node.js. Si tratta di un ambiente di esecuzione che permette di utilizzare JavaScript al di fuori del browser web, tipicamente sul lato server. Tale ambiente è noto per la sua architettura basata su eventi e per la gestione "non bloccante" delle operazioni di input/output (I/O). Questo significa che è molto efficiente nel gestire un elevato numero di richieste contemporaneamente, senza che una richiesta debba attendere il completamento della precedente. Questa caratteristica lo rende particolarmente adatto per applicazioni che richiedono alta scalabilità e reattività, come le chat in tempo reale o le API ad alto traffico. [14] Nonostante questi vantaggi, la nostra analisi ha evidenziato alcuni aspetti che ci hanno orientato diversamente. Innanzitutto, per il nostro team, la curva

di apprendimento di Node.js è stata percepita come potenzialmente più ripida rispetto a Python. Ciò avrebbe potuto rallentare la fase di sviluppo iniziale e l'integrazione del contributo dei membri. In secondo luogo, pur essendo Node.js dotato di un vasto ecosistema di librerie, abbiamo riscontrato che per le nostre esigenze specifiche – in particolare per la gestione avanzata di file XML conformi allo standard CAP e per l'interazione profonda con database relazionali come PostgreSQL e la sua estensione PostGIS – l'ecosistema Python offriva soluzioni più mature e direttamente allineate alle nostre necessità. Non abbiamo identificato, per il nostro contesto, vantaggi così marcati che potessero giustificare il compromesso in termini di familiarità e integrazione specifica. [13][14]

La familiarità preesistente con Python all'interno del team, sebbene non approfondita, unita alla sua vasta comunità e alla ricchezza di librerie mature (dalla creazione di API REST con framework come Flask e FastAPI, all'integrazione con message broker come RabbitMQ tramite pika), ha offerto un percorso di sviluppo più diretto e sicuro. Questa combinazione ha ridotto la potenziale curva di apprendimento per l'intero progetto, favorendo una maggiore produttività e coesione del team.

La Tabella 3.1 riassume le motivazioni che hanno dettato la scelta di Python come linguaggio principale del sistema complessivo.

Criterio di Valutazione	**Python (Vantaggi)**	**Node.js (Motivazioni della Non-Scelta)**
Gestione Dati e Configurazione	Ottima per XML (CAP); Solida per YAML.	Ecosistema meno allineato per XML (CAP).
Interazione con Persistenza Dati	Supporto robusto PostgreSQL/PostGIS; Librerie affidabili.	Ecosistema meno allineato per PostGIS.
Familiarità e Produttività Team	Familiarità preesistente; Curva di apprendimento accessibile; Ricco ecosistema e comunità.	Curva di apprendimento percepita più ripida; Minore familiarità; Vantaggi non decisivi.
Vantaggi Comparativi Specifici del Progetto	Sviluppo diretto e sicuro; Equilibrio potenza/versatilità/facilità d'uso.	I/O asincrono non decisivo per il contesto; Nessun vantaggio significativo per esigenze specifiche.

Tabella 3.1: Sintesi delle motivazioni per la scelta di Python

Per illustrare concretamente come queste considerazioni tecniche si traducano in benefici operativi, e per fornire un esempio pratico della nostra architettura basata su Python, si prenda in esame il microservizio *Alert Manager*. Questo componente cruciale del sistema sfrutta appieno le capacità di Python per gestire flussi di dati complessi e interfacciarsi

con diverse piattaforme esterne. La figura 3.1 visualizza il funzionamento dell'*Alert Manager*, evidenziando le librerie Python specifiche e le motivazioni strategiche dietro il loro impiego in ogni fase chiave del processo. Questo esempio dimostra la versatilità e la potenza di Python nel gestire le complessità richieste dalla nostra architettura a microservizi, confermando la sua idoneità come linguaggio principale per l'intero sistema.

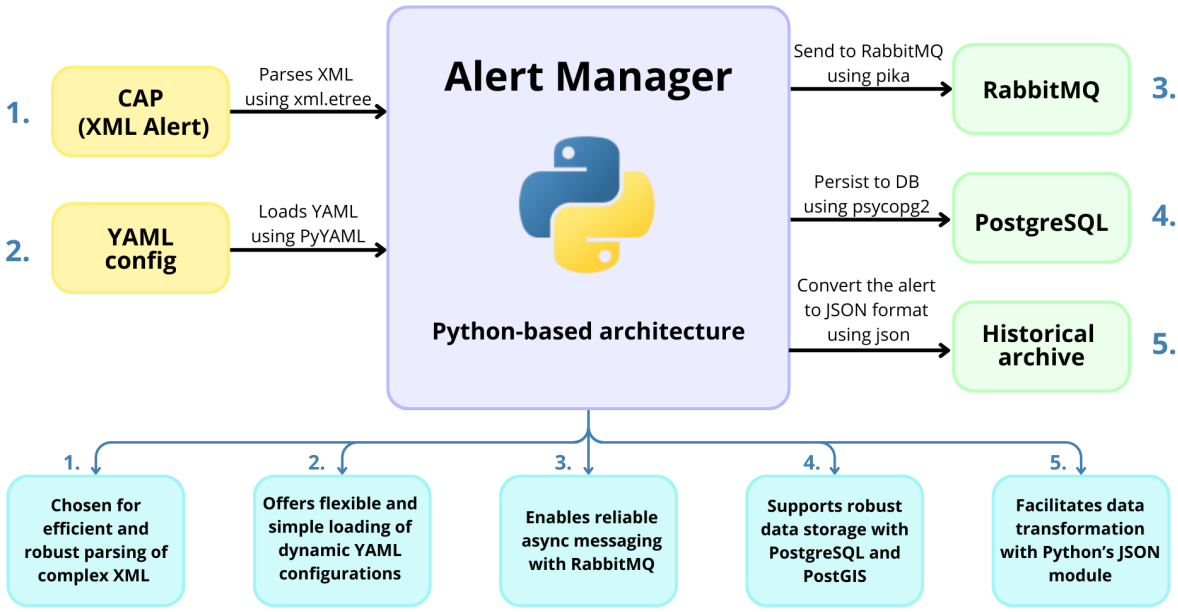

Figura 3.1: Esempio di implementazione Python-Based

In sintesi, la scelta di Python è stata dettata da un'analisi pragmatica che ha evidenziato il suo equilibrio tra potenza, versatilità e facilità d'uso. Ha fornito le fondamenta tecnologiche ideali per implementare un'architettura a microservizi complessa, consentendoci di concentrarci sulle sfide del dominio applicativo.[13]

3.2 Sistema di message queuing: RabbitMQ

Nelle architetture a microservizi, dove un sistema è composto da tanti piccoli servizi che lavorano insieme, farli comunicare in modo efficiente e affidabile è una sfida complessa. Le tecniche di comunicazione diretta, come le chiamate HTTP (Hypertext Transfer Protocol), che funzionano in modo sincrono (ovvero, un servizio invia una richiesta e si ferma ad attendere una risposta immediata), pur essendo semplici da implementare per scambi rapidi, possono introdurre vulnerabilità. Se il servizio chiamato è lento o temporaneamente non disponibile, il servizio che ha effettuato la richiesta può bloccarsi o, nel peggiore dei casi, fallire. Per mitigare questi rischi e promuovere un'architettura più resiliente e disaccoppiata, si è resa necessaria l'adozione di un modello di comunicazione asincrona.

Questo approccio si realizza tramite l'uso di code di messaggi (Message Queues), che permettono a un servizio mittente di inviare un messaggio e proseguire con le proprie attività senza attendere una conferma istantanea. Il messaggio viene depositato in una coda e sarà prelevato e processato dal servizio destinatario solo quando quest'ultimo sarà pronto. Questo meccanismo garantisce un fondamentale disaccoppiamento temporale e spaziale: i servizi non devono essere attivi e comunicare nello stesso istante, aumentando la flessibilità e la tolleranza ai guasti del sistema. [15]

La nostra analisi ha condotto alla scelta di RabbitMQ come implementazione principale per il sistema di message queuing. Questa decisione si è basata sulla sua comprovata affidabilità e sulla sua maturità operativa consolidata in numerosi contesti reali e complessi. RabbitMQ è stato specificamente progettato per offrire una garanzia di consegna dei messaggi, una caratteristica critica per il nostro progetto: essa assicura che le informazioni, una volta inviate, non vadano perse, anche in condizioni di elevato carico o in presenza di temporanee interruzioni dei servizi riceventi. Un altro aspetto determinante è la sua flessibilità nel supportare vari pattern di routing – ovvero le modalità con cui i messaggi vengono indirizzati tra chi li produce (i mittenti) e chi li consuma (i destinatari) – e diverse topologie di messaggistica. Questa versatilità ci ha permesso di modellare con precisione le esigenze differenziate di comunicazione tra i nostri microservizi, contribuendo a costruire un'infrastruttura di comunicazione robusta e adattabile. [16]

Durante il processo di valutazione delle soluzioni per la comunicazione asincrona, abbiamo esaminato anche Apache Kafka. Questa è una piattaforma di event streaming distribuita ad alte prestazioni. La sua funzione principale è gestire flussi di dati continui e ad alto volume, operando essenzialmente come un log immutabile di eventi storici. Sebbene Kafka sia estremamente potente per scenari che richiedono l'elaborazione di grandi quantità di dati in tempo reale o la riproduzione storica di flussi di eventi, per le nostre esigenze iniziali è stata ritenuta non necessaria. La nostra priorità era la garanzia di consegna individuale dei messaggi e un disaccoppiamento più tradizionale tra produttori e consumatori. Per questi obiettivi specifici, la maggiore complessità infrastrutturale e di gestione di Kafka non si giustificava. Similmente, le chiamate HTTP dirette sono state scartate poiché, essendo sincrone, avrebbero reintrodotto le dipendenze temporali che stavamo cercando di eliminare e non avrebbero fornito i meccanismi di persistenza impliciti cruciali per la resilienza nella propagazione delle informazioni critiche.

È opportuno riconoscere che, per rafforzare ulteriormente la resilienza di un'architettura a microservizi[17], esistono pattern avanzati che sono stati considerati. Tra questi, il Circuit Breaker (Interruttore di Circuito), un meccanismo che, come un fusibile, "apre il circuito" per un periodo di tempo quando un servizio mostra ripetuti fallimenti o lentezza. Questo impedisce di sovraccaricare ulteriormente un componente già in difficoltà, per-

mettendogli di recuperare.[18] Abbiamo anche valutato l'impiego di Dead Letter Queues (DLQ), code speciali dove i messaggi che non possono essere elaborati con successo (ad esempio, a causa di errori nel formato o logica applicativa incompatibile) vengono automaticamente reindirizzati. Questo evita che tali messaggi "velenosi" blocchino la coda principale e fornisce un punto centralizzato per l'analisi e la correzione degli errori.[19] Sebbene l'importanza di questi pattern sia indiscutibile, in questa fase iniziale del progetto la priorità è stata posta sulla stabilizzazione della comunicazione fondamentale e sull'implementazione di efficaci sistemi di logging e monitoraggio per l'identificazione rapida dei problemi. La loro eventuale adozione potrà essere valutata in futuro, in relazione all'evoluzione della complessità del sistema e delle specifiche esigenze operative che dovessero emergere.

Per illustrare concretamente l'implementazione della comunicazione asincrona e il ruolo di RabbitMQ in un contesto operativo reale, la Figura 3.2 presenta un diagramma semplificato delle interazioni chiave che coinvolgono il microservizio *User Simulator*. Questo diagramma evidenzia il flusso di messaggi attraverso le code di RabbitMQ, mostrando come il simulatore riceva istruzioni e invii dati in modo disaccoppiato e affidabile, ponendosi al centro di scambi informativi eterogenei tra il *Notification Center* e il *Position Manager*.

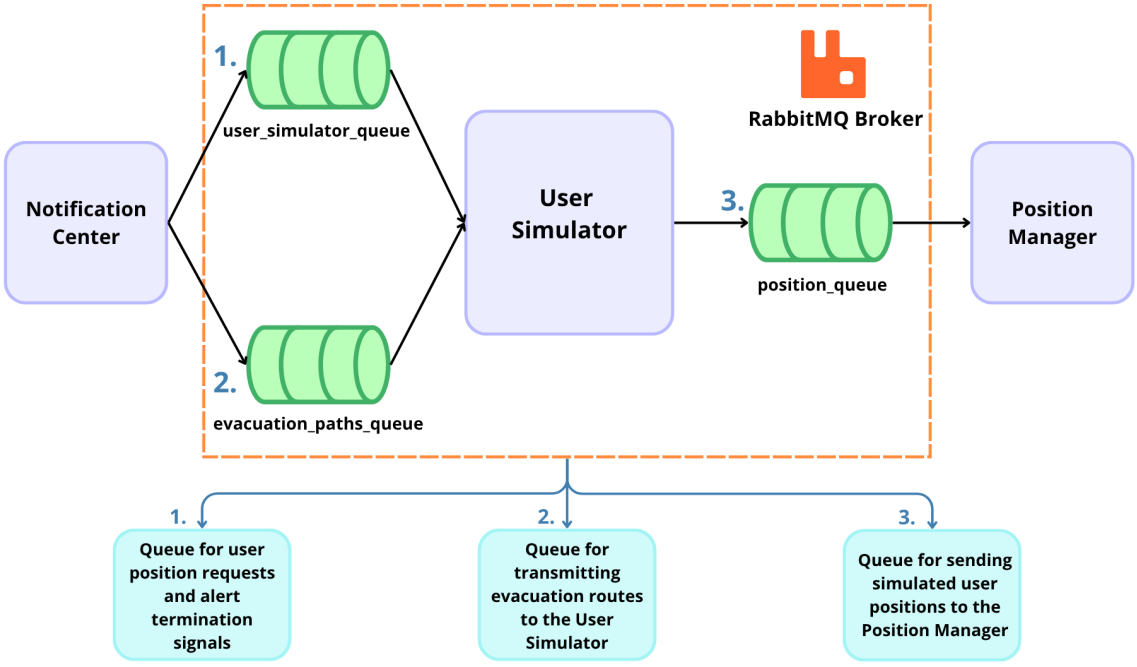

Figura 3.2: Esempio di implementazione con RabbitMQ

In sintesi, la scelta di RabbitMQ e l'adozione strategica delle code di messaggi rappresentano una decisione architetturale fondamentale. Essa ha permesso di definire un'infrastruttura di comunicazione robusta, affidabile e scalabile per l'intero sistema a mi-

croservizi, fornendo le basi necessarie per la sua futura crescita. La Tabella 3.2 illustra sinteticamente le motivazioni che hanno portato all'adozione di RabbitMQ, mettendole a confronto con le principali alternative valutate.

Criterio	**RabbitMQ (Scelta Effettuata)**	**Apache Kafka (Motivazioni della Non-Scelta Iniziale)**	**Chiamate HTTP Dirette (Motivazioni della Non-Scelta)**
Modello di Comunicazione	Comunicazione asincrona tramite code. Disaccoppiamento temporale.	Streaming di eventi. Log immutabile di eventi.	Comunicazione sincrona. Accoppiamento diretto.
Affidabilità e Garanzia Consegna	Alta affidabilità. Consegna dei messaggi garantita.	Alta resilienza. Consegna garantita con maggiore complessità.	Nessuna garanzia di consegna implicita. Fallimenti diretti.
Flessibilità di Routing	Supporto nativo per vari pattern e topologie di messaggistica.	Focalizzato su flussi di eventi e topic. Routing basato su partizioni.	Punto-punto. Nessun pattern di routing avanzato.
Complessità di Gestione	Maturità operativa. Gestione relativamente semplice per code.	Maggiore complessità infrastrutturale e di gestione iniziale.	Minima complessità infrastrutturale. Limiti funzionali.
Adeguatezza Obiettivi	Ideale per garanzia di consegna individuale e disaccoppiamento.	Orientato a big data e eventi storici. Non prioritario per esigenze attuali.	Mancanza di resilienza e persistenza asincrona.

Tabella 3.2: Sintesi delle motivazioni per la scelta di RabbitMQ nel sistema a microservizi

3.3 Database di persistenza: PostgreSQL

La definizione della strategia di persistenza dei dati ha rappresentato una scelta architetturale cardine nell'implementazione del sistema a microservizi. L'esigenza primaria era individuare una soluzione che garantisse l'affidabilità e l'integrità delle informazio-

ni, unitamente alla capacità di gestire specifici domini di dati. L'analisi ha coinvolto principalmente due categorie di database: i sistemi di gestione di database relazionali (RDBMS), che organizzano i dati in tabelle con collegamenti precisi e strutturati, esemplificati da PostgreSQL, e i database NoSQL, più flessibili e orientati a documenti o altri formati, quali MongoDB. La decisione finale di adottare PostgreSQL per le istanze di database afferenti ai microservizi è stata il risultato di una valutazione multicriterio.

Per il nostro sistema, la garanzia di massima integrità e affidabilità dei dati era un requisito imprescindibile per entità critiche quali gli eventi di allerta, le posizioni degli utenti e la rappresentazione strutturale dell'edificio. In questo contesto, PostgreSQL si è imposto come la scelta più coerente. Essendo un RDBMS pienamente conforme agli standard ACID (Atomicità, Consistenza, Isolamento, Durabilità) - un insieme di proprietà che assicurano che le transazioni sui dati siano elaborate in modo affidabile, prevenendo corruzioni o perdite - PostgreSQL offre transazioni robuste e una integrità referenziale nativa. Quest'ultima caratteristica garantisce che i collegamenti tra i dati in diverse tabelle siano sempre validi e consistenti, aspetto fondamentale per mantenere la coerenza delle complesse relazioni intrinseche tra le diverse entità. Sebbene i database NoSQL come MongoDB offrano notevole flessibilità di schema e scalabilità orizzontale, la loro architettura non si allineava altrettanto efficacemente alle nostre esigenze di un modello dati strutturato e fortemente relazionale. La gestione di relazioni complesse e l'assicurazione dell'integrità transazionale in un contesto NoSQL avrebbe richiesto un'implementazione a livello applicativo significativamente più complessa e soggetta a potenziali incoerenze.

L'elemento definitivo che ha consolidato la preferenza per PostgreSQL è stata la disponibilità e la maturità dell'estensione PostGIS. Questa estensione dota PostgreSQL di funzionalità avanzata per la gestione di dati geografici, supportando specifici tipi di dato geometrici (come punti per posizioni, linee per percorsi o poligoni per aree) e un'ampia gamma di operazioni georeferenziali e di analisi spaziale. Ad esempio, calcolare distanza, individuare intersezioni o determinare contenimenti geografici. In un'architettura dove un'ampia varietà di informazioni - quali le zone degli alert, le coordinate delle posizioni utente, i percorsi di evacuazione e la topologia dell'edificio - sono intrinsecamente legate ad un contesto spaziale, l'integrazione nativa di PostGIS ha rappresentato un vantaggio strategico. Questa capacità di persistenza e interrogazione di dati spaziali, senza la necessità di ricorrere a soluzioni esterne o stratificazione architetturali complesse, ha ottimizzato sia la semplificazione del design complessivo, sia l'efficienza delle operazioni geospaziali. [20][21]

Nel nostro sistema, PostgreSQL con PostGIS è stato implementato attraverso due istanze principali dedicate alla persistenza dei dati sotto la nostra responsabilità. Una prima istanza è specificamente dedicata all'archiviazione selettiva degli alert rilevanti ge-

nerati, registrando i relativi metadati e le coordinate geospaziali per successive analisi storico-epidemiologiche (ovvero, studi basati sui dati storici per comprendere tendenze o impatti degli eventi). Questa istanza è primariamente gestita dal microservizio *Alert Manager*. Una seconda istanza di database, come illustrato in Figura 3.3, presenta una struttura più articolata, ospitando diverse tabelle con funzionalità distinte: include la rappresentazione dei nodi e degli archi del grafo dell'edificio, una tabella per le posizioni correnti degli utenti (cruciale per l'identificazione di situazioni di pericolo e la visualizzazione della mappa), e una tabella per lo storico delle posizioni degli utenti, orientata principalmente a finalità di analisi retrospettiva.

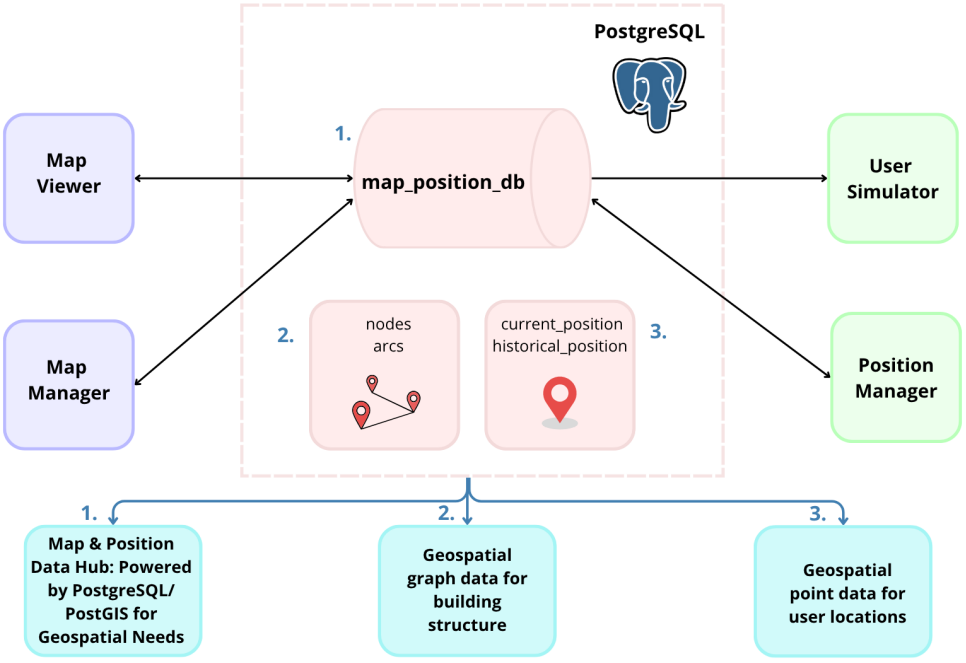

Figura 3.3: Esempio di database relazionale nel sistema complessivo

Sebbene le posizioni utente siano attualmente generate da un simulatore per scopi di testing, il sistema è progettato per integrare futuri meccanismi di rilevazione di posizioni reali e gestire coordinate geografiche effettive. Questa seconda istanza è accessibile in modalità lettura e scrittura sia dal microservizio User Simulator che dal Position Manager. L'interazione con entrambe le istanze PostgreSQL/PostGIS è gestita efficacemente tramite le librerie Python, quali psycopg2 per la connettività diretta al database e SQLAlchemy per facilitare l'Object-Relational Mapping (ORM). Questo approccio assicura una connettività robusta e performante, pienamente integrata nell'ecosistema di sviluppo selezionato.

In sintesi, l'adozione di PostgreSQL, con la sua estensione PostGIS, ha fornito una soluzione robusta, affidabile e altamente flessibile. Questa scelta ha soddisfatto pienamente

le esigenze del nostro sistema a microservizi per la gestione di dati relazionali complessi e l'elaborazione geospaziale avanzata, costituendo una base solida per l'intera architettura.

3.4 File di configurazione: YAML

Per la gestione delle configurazioni operative e dei parametri applicativi all'interno del sistema a microservizi, è stato adottato il formato YAML (YAML Ain't Markup Language). YAML si configura come un linguaggio di serializzazione di dati estremamente leggibile dall'essere umano, progettato per favorire la chiarezza e la semplicità, pur mantenendo una robusta capacità di rappresentazione di strutture dati complesse attraverso l'uso dell'indentazione per definire la gerarchia.[22]

La scelta di YAML è stata guidata da molteplici fattori strategici. La sua sintassi pulita e intuitiva contribuisce significativamente alla leggibilità e alla manutenibilità dei file di configurazione, minimizzando il rischio di errori e semplificando la collaborazione tra gli sviluppatori. La capacità di organizzare i dati in una struttura gerarchica nidificata, basata su mappe (equivalenti a dizionari o oggetti) e liste (equivalenti ad array), si è rivelata ideale per modellare parametri complessi in modo ordinato e facilmente navigabile. Questa caratteristica è particolarmente vantaggiosa in un'architettura distribuita dove ogni microservizio richiede un insieme specifico e spesso articolato di impostazioni.

Nel contesto specifico del nostro sistema di gestione delle emergenze, i file YAML svolgono un ruolo fondamentale nel disaccoppiamento tra la logica applicativa e la configurazione dell'ambiente o dello scenario. Ad esempio, vengono impiegati per:

- Definire i criteri di filtraggio degli alert rilevanti: È possibile configurare soglie, tipi di eventi o zone specifiche all'interno dell'edificio che attivano un'allerta, senza dover modificare il codice sorgente del microservizio *Alert Manager*.

- Modellare la distribuzione e il comportamento degli utenti nella simulazione: Parametri quali il numero di utenti simulati, e le probabilità di simulazione all'interno di una tipologia di nodo dell'edificio possono essere definiti esternamente, permettendo di testare il sistema in diversi scenari senza ricompilazione dell *User Simulator*.

- Stabilire le regole per la determinazione dello stato di pericolo di un utente: I file YAML configurano le condizioni che il *Position Manager* utilizza per classificare un utente come in pericolo, garantendo flessibilità nella definizione delle politiche di sicurezza.

Questo approccio basato su file di configurazione YAML incarna un principio chiave della progettazione software: la separazione delle responsabilità (separation of concerns). Tale

metodologia consente di adattare il sistema a diversi contesti ed edifici semplicemente modificando questi file, senza la necessità di intervenire sulla parte implementativa del codice. Ciò riduce drasticamente il tempo e la complessità delle operazioni di adattamento, test e deployment per nuovi scenari o strutture, elevando la flessibilità e la riusabilità del sistema a un livello superiore. In conclusione, l'adozione di YAML non solo ottimizza la gestione delle impostazioni, ma contribuisce attivamente alla modularità e all'estensibilità dell'intera architettura.

3.5 Conclusioni sulle tecnologie fondamentali

In sintesi, la progettazione architetturale del sistema si è basata su una selezione mirata di tecnologie fondamentali, ciascuna scelta per rispondere a requisiti specifici e cruciali. L'adozione di RabbitMQ come message broker ha garantito un'infrastruttura di comunicazione robusta e asincrona, essenziale per il disaccoppiamento dei microservizi e la gestione efficiente dei flussi di eventi. Parallelamente, PostgreSQL, con la sua estensione geospaziale PostGIS, ha fornito una soluzione di persistenza dati affidabile e scalabile, capace di gestire sia relazioni complesse che informazioni geografiche ad alta precisione, fondamentali per la rappresentazione dell'ambiente e il tracciamento delle posizioni utente. A completamento di queste scelte infrastrutturali, l'impiego di YAML per i file di configurazione ha assicurato la flessibilità e la manutenibilità del sistema, permettendo un adattamento rapido a scenari operativi e contesti di edificio diversi senza la necessità di modifiche al codice.

La sinergia tra questi componenti chiave è cruciale per l'efficienza e la resilienza dell'intera architettura. La comunicazione disaccoppiata via code di messaggi consente ai microservizi di operare in modo indipendente, reagendo agli eventi senza vincoli diretti di disponibilità, mentre il database relazionale garantisce l'integrità e la coerenza delle informazioni persistenti, e i file di configurazione ne rendono agevole la personalizzazione. Questa combinazione strategica crea un'architettura che non solo soddisfa i requisiti funzionali e non funzionali più stringenti, ma è anche intrinsecamente predisposta alla scalabilità, alla riusabilità e alla manutenibilità. Le scelte tecnologiche illustrate in questo capitolo costituiscono, pertanto, il pilastro su cui si erge l'intera implementazione del sistema, la cui struttura dettagliata e l'operatività verranno approfondite nei capitoli successivi.

Capitolo 4

Microservizio gestore degli alert

Questo capitolo dell'elaborato si addentra nella fase di implementazione del sistema proposto. Vuole, dunque, arrivare al punto di convergenza tra le fondamenta teoriche e le scelte architetturali di cui si è precedentemente discusso. L'analisi che seguirà si concentrerà specificamente sul contributo individuale apportato allo sviluppo, partendo dal primo microservizio la cui realizzazione ha rappresentato una responsabilità diretta all'interno del progetto complessivo.

4.1 Introduzione e funzionalità specifiche

Nel contesto dell'architettura a microservizi che abbiamo adottato, il microservizio *Alert Manager* svolge un ruolo di primo piano. La sua funzione principale è quella di coordinare l'intero percorso degli allarmi che si manifestano all'interno del nostro sistema. In sostanza, questo servizio prende in carico gli eventi critici nel momento in cui accadono, li elabora seguendo una logica precisa e si occupa di inviare le notifiche a chi ne ha bisogno. È fondamentale comprendere che una sua eventuale incapacità nell'identificare correttamente un alert che rappresenti una minaccia per l'edificio comprometterebbe seriamente la funzionalità dell'intero sistema.

Due responsabilità fondamentali gravano sull'*Alert Manager*. In primo luogo, ha il compito di ricevere e interpretare i messaggi di allerta conformi al Common Alerting Protocol, estraendone tutte le informazioni rilevanti all'evento di pericolo. Successivamente, deve valutare la rilevanza di tale evento basandosi su un set di regole di business configurabili. Queste regole sono definite all'interno di un file di configurazione e permettono di adattare il sistema alle specificità di ogni edificio, consentendo di personalizzare i criteri di filtraggio degli alert. Se l'evento viene ritenuto rilevante sulla base di queste logiche, l'*Alert Manager* provvede ad archiviarlo nel database dedicato e, infine, si occupa dell'in-

stradamento del messaggio di allerta verso i microservizi destinatari e attraverso i canali dedicati.

4.2 Analisi dello stato dell'arte e motivazione delle scelte implementative

Per comprendere meglio il panorama delle soluzioni e delle tecnologie disponibili per la gestione degli alert, e al fine di giustificare le scelte implementative che hanno guidato la progettazione dell'*Alert Manager*, questa sezione analizza lo stato dell'arte e le alternative considerate. Tale indagine si focalizzerà sia su sistemi di monitoraggio e alerting di carattere generale, impiegati nel settore dell'Information Technology, sia su soluzioni specifiche per il dominio applicativo della gestione degli alert negli edifici.

4.2.1 Sistemi di monitoraggio e allerta generici

Nel vasto panorama delle soluzioni IT dedicate al monitoraggio e all'allerting, emergono per la loro diffusione e importanza architetture come Prometheus Alertmanager, Zabbix e Nagios.[23][24][25] Nonostante la loro comune finalità di identificare e gestire situazioni anomale all'interno di un sistema, le loro architetture e le modalità operative presentano caratteristiche distintive che meritano un'analisi comparativa, specialmente in relazione ai requisiti specifici di un sistema di gestione degli alert che adotta lo standard CAP.

- Prometheus Alertmanager: componente del sistema di monitoraggio Prometheus che si specializza nell'elaborazione e nell'instradamento degli allarmi generati dal sistema, il cui funzionamento si basa sull'analisi di sequenze temporali di metriche. La sua efficacia nella deduplica, nell'aggregazione e nell'inoltro flessibile degli alert è notevole, tuttavia il suo focus primario rimane la gestione di alert derivati da indicatori di performance IT. L'integrazione con un sistema che implementa nativamente lo standard CAP richiederebbe un adattamento significativo, in particolare per quanto riguarda la strutturazione delle notifiche e la gestione di alert provenienti da sorgenti diverse dalle metriche monitorate da Prometheus, che si basano sulla semantica specifica degli eventi di emergenza, non solo su valori numerici. [23]

- Zabbix: piattaforma di monitoraggio completa per infrastrutture IT, che integra in un unico ambiente la raccolta dei dati e la gestione degli alert. Il suo modello operativo si fonda sulla sorveglianza di host, servizi e parametri specifici (item), con l'impiego di trigger per l'individuazione dei problemi e di azioni per l'ammi-

nistrazione delle notifiche. Nonostante la sua vasta gamma di canali di notifica e l'elevato grado di personalizzazione, il supporto intrinseco per lo standard CAP è assente. L'implementazione di notifiche conformi al CAP implicherebbe la creazione di script ad hoc o l'adozione di soluzioni di integrazione esterne, e la sua architettura centralizzata potrebbe non allinearsi idealmente con i principi di un'architettura a microservizi che pone l'accento su un formato standardizzato per la comunicazione degli alert.[24]

- Nagios: costituisce un sistema di monitoraggio la cui flessibilità risiede nell'utilizzo di plugin per l'esecuzione di controlli su host e servizi. Gli alert sono generati in base agli esiti di tali verifiche, e le notifiche sono gestite attraverso comandi configurabili. Analogamente a Prometheus e Zabbix, Nagios non fornisce un supporto intrinseco per lo standard CAP. L'adozione di CAP richiederebbe lo sviluppo di plugin di notifica su misura, e la sua architettura, sebbene estendibile, potrebbe necessitare di una significativa personalizzazione per integrarsi efficacemente con un sistema che fa dello standard per la comunicazione di emergenza il suo pilastro.[25]

Nessuno di questi sistemi offre un supporto nativo per lo standard CAP, che è un requisito fondamentale per il nostro progetto. La loro architettura e filosofia operativa sono orientate al monitoraggio di metriche IT o alla gestione di allarmi basati su stati di servizio, con una limitata comprensione della semantica specifica degli eventi di mergenza definita da standard come CAP. Questa lacuna, unita alla necessità di un'architettura a microservizi flessibile e interoperabile, evidenzia il bisogno di una soluzione personalizzata. [26]

4.2.2 Soluzioni specifiche per il dominio applicativo

È fondamentale, nell'analisi dello stato dell'arte, considerare come le soluzioni di gestione degli alert sono attualmente implementate nel dominio specifico dei sistemi di gestione degli edifici, al fine di identificare potenziali limitazioni e giustificare le scelte progettuali dell' *Alert Manager*.

Nel settore dell'automazione degli edifici, la gestione degli alert è spesso integrata nei Building Management Systems (BMS), che utilizzano protocolli standardizzati come BACnet e Modbus per il controllo centralizzato degli impianti. Questi sistemi si concentrano principalmente sul garantire la sicurezza, l'efficienza e la gestione centralizzata degli impianti all'interno dell'edificio. Tuttavia, le soluzioni esistenti presentano alcune limitazioni significative in termini di interoperabilità con sistemi esterni e standardizzazione del formato degli alert. In particolare:

- Interoperabilità limitata: i protocolli BACnet[1][27] e Modbus[28], sebbene ampia-

[1]BACnet - Building Automation and Control Networks

mente utilizzati per la comunicazione interna tra i dispositivi di un edificio, non forniscono un supporto nativo per la comunicazione con sistemi esterni che utilizzano standard diversi, come il CAP. Ciò rende difficile l'integrazione degli allarmi generati dai sistemi BMS con sistemi di emergenza più ampi o piattaforme di gestione degli incidenti.

- Formati di alert non standardizzati: i diversi sistemi all'interno di un edificio (ad esempio, allarmi antincendio, allarmi di sicurezza) possono utilizzare formati di alert proprietari o standard di settori specifici, rendendo complessa l'aggregazione e l'elaborazione centralizzata degli alert provenienti da fonti diverse.

L'*Alert Manager* mira a superare queste limitazioni adottando il Common Alerting Protocol. Il CAP fornisce un formato standardizzato per la comunicazione di emergenza, facilitando l'interoperabilità con una vasta gamma di sistemi esterni e consentendo una gestione più efficiente e centralizzata degli alert provenienti da diverse fonti all'interno dell'edificio. L'adozione del CAP è una scelta progettuale fondamentale che consente all'*Alert Manager* di fungere da ponte tra i sistemi BMS interni e il mondo esterno, garantendo che le informazioni di emergenza possano essere comunicate in modo tempestivo e accurato.

4.2.3 Conclusioni sull'analisi dello stato dell'arte

L'analisi dello stato dell'arte ha fornito diverse importanti indicazioni che hanno guidato le scelte implementative per il nostro microservizio *Alert Manager*.

In primo luogo, è emerso chiaramente che i sistemi di monitoraggio generici come Prometheus Alertmanager, Zabbix e Nagios, pur essendo potenti e flessibili, non offrono un supporto nativo per lo standard CAP, che rappresenta un requisito fondamentale per il nostro progetto. La loro architettura e la loro filosofia operativa sono spesso orientate al monitoraggio di metriche IT o alla gestione di allarmi basati su stati di servizio, con una limitata comprensione della semantica specifica definita da standard come CAP. Questa lacuna evidenzia la necessità di una soluzione personalizzata che possa interpretare e generare alert conformi a tale standard.

In secondo luogo, la ricerca di soluzioni specifiche per il dominio applicativo della gestione degli alert in edifici non ha rivelato l'esistenza di soluzioni che supportino nativamente lo standard CAP. I sistemi BMS e le piattaforme di gestione degli allarmi esistenti tendono a utilizzare protocolli e standard propri del settore dell'automazione edilizia. Questa mancanza di soluzioni standardizzate con supporto CAP rafforza la necessità di sviluppare un microservizio su misura che possa aderire pienamente a tale standard e integrarsi specificamente con i sistemi dell'edificio, colmando un vuoto nell'offerta attuale.

Le lacune evidenziate nell'analisi dello stato dell'arte, hanno influenzato in modo significativo le scelte implementative per il microservizio *Alert Manager*. L'obiettivo primario è stato, e rimane, quello di creare una soluzione personalizzabile, interoperabile ed estremamente efficace, per la gestione degli alert nel nostro contesto applicativo.

4.3 Sviluppo operativo del microservizio

Questa sezione rappresenta il cuore dell'implementazione del microservizio. Qui si esamina come le scelte architetturali e i requisiti funzionali si siano tradotti in una soluzione operativa concreta. Verranno descritte le sue componenti principali, il flusso dettagliato di elaborazione degli alert e gli aspetti implementativi specifici.

4.3.1 Componenti principali e flusso di lavoro degli alert

Il microservizio *Alert Manager*, sviluppato interamente in Python, è l'elemento chiave del sistema per la gestione degli alert di emergenza. La sua architettura è stata concepita per garantire robustezza e flessibilità in ogni fase di gestione di un alert, dalla sua acquisizione all'instradamento finale. La sua struttura è schematizzata in Figura 4.1, offrendo una visione d'insieme dell'organizzazione del progetto.

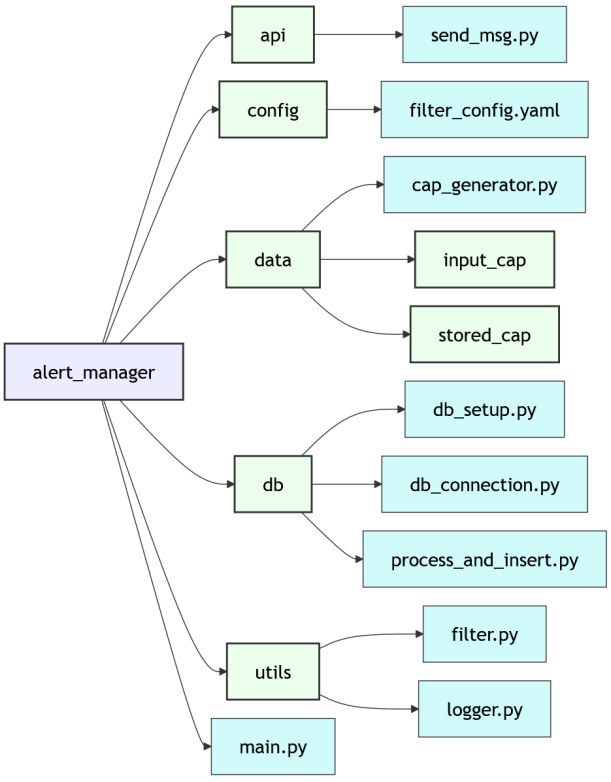

Figura 4.1: Struttura interna del microservizio *Alert Manager*

Il microservizio è logicamente scomponibile in diverse componenti principali, ognuna con una responsabilità ben definita. Tali componenti operano in sinergia per implementare il flusso di lavoro degli alert che viene orchestrato dal modulo principale del microservizio (main.py). Il flusso si articola in una pipeline ben definita, progettata per garantire che ogni alert venga sistematicamente acquisito, validato, filtrato, archiviato e infine instradato solo se ritenuto rilevante, come mostrato in Figura 4.2.

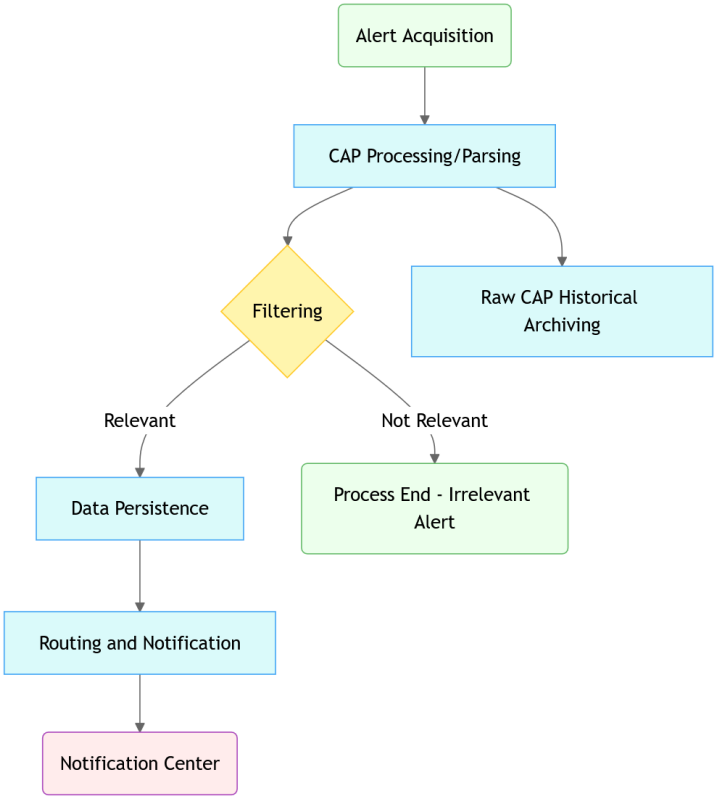

Figura 4.2: Flusso interno del microservizio *Alert Manager*

Vediamo ora in dettaglio le componenti principali che rendono possibile questo flusso:

- Modulo di acquisizione alert: questa componente riceve i messaggi di emergenza. Sebbene il sistema sia progettato per integrarsi con fonti esterne, nella fase prototipale il microservizio preleva file CAP in formato XML da una cartella locale. Questo approccio consente di testare la logica del microservizio con un set controllato di alert, simulando diversi scenari di pericolo. Il ciclo di vita di un alert inizia con la sua acquisizione e l'immediata elaborazione del file XML.

- Modulo di elaborazione e filtraggio CAP: una volta acquisito il messaggio CAP, questo modulo dà il via alla fase di elaborazione del messaggio. Attraverso l'uso della libreria xml.etree.ElementTree di Python, viene eseguito il parsing del messaggio

XML in modo da comprenderne la struttura ed estrarre le informazioni principali. Queste informazioni vengono poi trasformate in un dizionario Python, un passaggio utile per standardizzare i dati e renderli più facili da utilizzare. Durante l'elaborazione, vengono estratti i campi principali dell'alert (come identifier, sender e status), i dettagli dei blocchi info (ad esempio event, severity, description) e infine le informazioni geografiche dalla sezione area (come polygon e areaDesc), con le geometrie convertite in formato GeoJSON pronto per il database. Una volta estrapolate le informazioni essenziali dall'alert, il modulo applica una logica di filtraggio basata su regole di business configurabili, contenute all'interno di un file YAML esterno. Se un alert rientra nei criteri di rilevanza (ad esempio per tipo di evento o gravità), si procede con i passaggi successivi.

- Archiviazione storica del CAP grezzo: indipendentemente dalla sua rilevanza o dal risultato del filtraggio, ogni messaggio CAP ricevuto viene salvato in una cartella storica. Questo garantisce di avere una traccia completa di tutti gli alert passati per il sistema. Vengono conservate sia la versione XML originale, sia una versione JSON completa (ottenuta dalla conversione a dizionario), entrambe identificate con un timestamp unico per facilitare le analisi e il monitoraggio futuri.

- Modulo di persistenza dati: gli alert che superano il filtro di rilevanza vengono salvati in un database PostgreSQL. Questa componente gestisce la connessione al database e l'inserimento dei dati in tabelle normalizzate (alerts, info, areas). L'uso dell'estensione PostGIS in PostgreSQL permette di gestire e interrogare in modo efficiente le informazioni geospaziali contenute nei messaggi CAP, come le aree interessate dall'emergenza. L'intero processo di inserimento dei dati è trattato come una transazione atomica, un approccio che assicura la completa integrità e affidabilità del database.

- Modulo di instradamento e notifica: nel momento in cui l'alert supera il filtro, viene immediatamente inviato al *Notification Center*, un altro microservizio del sistema. Questo passaggio è l'ultima fase del processo e assicura una propagazione rapida delle informazioni critiche. L'invio avviene tramite un sistema di messaggistica asincrona che impiega RabbitMQ come intermediario (message broker). L'*Alert Manager* agisce da produttore (Producer), inserendo il dizionario Python che rappresenta l'alert su una coda di messaggi dedicata. Questo utilizzo di RabbitMQ è fondamentale, poiché separa l'*Alert Manager* dal *Notification Center*, migliorando la resilienza e la scalabilità dell'intero sistema.

- Modulo di supporto e logging: oltre alle sue funzioni principali, il microservizio integra un robusto sistema di logging. Questo modulo è essenziale per il monitoraggio, la risoluzione dei problemi (debugging) e la tracciabilità delle operazioni, registrando eventi importanti ed eventuali errori sia a console che su file dedicati.

Questo flusso di lavoro si ripete continuamente, permettendo all'*Alert Manager* di monitorare ed elaborare senza sosta nuovi messaggi di emergenza, assicurando che solo le informazioni critiche e rilevanti vengano propagate e archiviate in modo efficiente.

4.3.2 Elaborazione del Common Alerting Protocol

Il Common Alerting Protocol (CAP), uno standard OASIS riconosciuto a livello internazionale per la comunicazione di emergenza, rappresenta la base su cui l'*Alert Manager* costruisce la sua logica di funzionamento. Questo protocollo, come anticipato nel Capitolo 1, fornisce un formato versatile e interoperabile per la trasmissione di messaggi di allerta pubblica, indipendentemente dal mezzo di comunicazione o dall'applicazione. Per il microservizio in esame, la capacità di ricevere, interpretare e processare messaggi CAP è fondamentale per la sua missione di coordinamento degli alert.

Un messaggio CAP è un documento XML con una struttura gerarchica che incapsula tutte le informazioni relative ad un evento di emergenza. Per le finalità dell'*Alert Manager*, i blocchi principali di interesse sono:

- <alert>: rappresenta il nodo radice del messaggio. Contiene i metadati essenziali relativi all'allerta nel suo complesso, come un identificatore univoco, il mittente, il timestamp di invio, lo stato dell'allerta, il tipo di messaggio e l'ambito di distribuzione. Questi campi servono a fornire una prima contestualizzazione dell'evento.

- <info>: un messaggio CAP può contenere uno o più blocchi di questo tipo, ciascuno dei quali descrive un aspetto specifico dell'allerta, spesso in lingue diverse o per pubblico differenti. L'*Alert Manager* estrae da essi dettagli cruciali come la categoria dell'evento, il tipo di evento e le istruzioni per la risposta all'emergenza.

- <area >: all'interno di ciascun blocco <info>, possono essere presenti uno o più blocchi di questo tipo, che definiscono l'area geografica interessata dall'alert. Questo blocco include una descrizione dell'area e, in particolare, le geometrie che specificano il confine dell'evento. La gestione di queste informazioni spaziali è cruciale per localizzare l'emergenza e per successive analisi.

Per una comprensione visiva della gerarchia e dei principali elementi del messaggio CAP, si rimanda alla Figura 4.3.

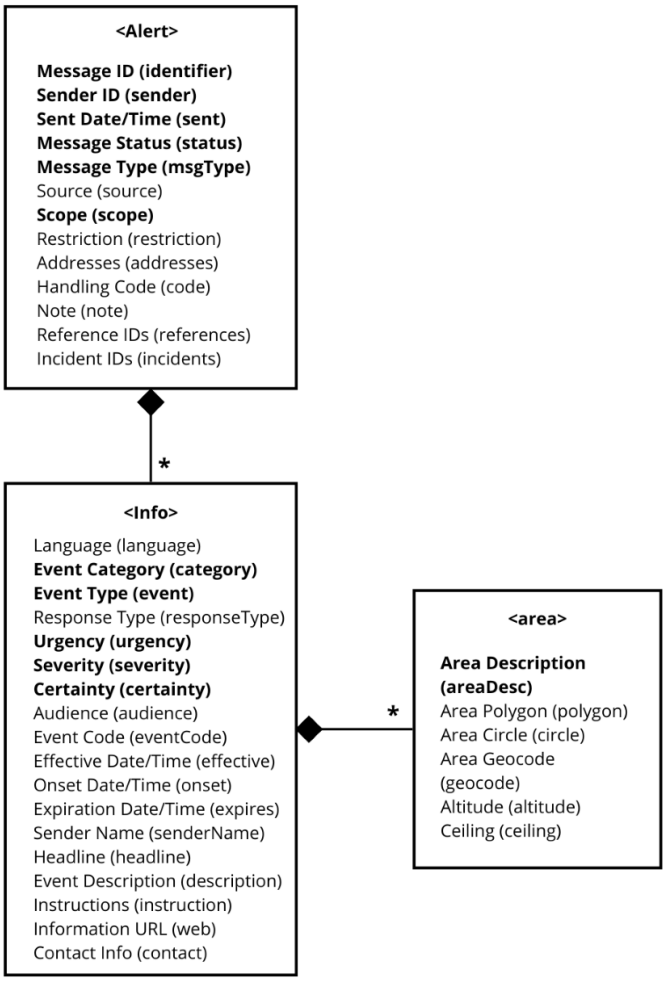

Figura 4.3: Struttura standard di un messaggio CAP

Il microservizio *Alert Manager*, in particolare attraverso il suo modulo cap_generator.py, è incaricato di effettuare il parsing dei messaggi CAP in formato XML e di convertirli in una struttura dati più gestibile per il sistema: un dizionario Python. Questo processo è fondamentale per normalizzare le informazioni e facilitarne l'accesso e la manipolazione da parte degli altri moduli. Per illustrare questa trasformazione, consideriamo un esempio schematico di messaggio CAP XML. La Figura 4.1 mostra un estratto contenente gli elementi principali che il microservizio processa, concentrandosi sui metadati di alto livello e sulla sezione <area >per le informazioni geografiche. Per il listato XML completo, si rimanda all'Allegato A, Figura 7.1.

Listing 4.1: Estratto schematico di messaggio CAP XML

```
<alert xmlns="urn:oasis:names:tc:emergency:cap:1.2">
  <identifier>Test-Alert-001</identifier>
  <sender>example@example.org</sender>
  <sent>2025-04-30T12:00:00+00:00</sent>
```

```
  <status>Actual</status>
  <msgType>Alert</msgType>
  <scope>Public</scope>
  <info>
    <area>
      <areaDesc>Building A</areaDesc>
      <polygon>45.0,9.0 45.0,9.1 45.1,9.1 45.1,9.0 45.0,9.0</
          polygon>
      <altitude>10</altitude>
    </area>
  </info>
</alert>
```

Il modulo responsabile del parsing analizza la struttura XML, estraendo i valori dei tag rilevanti e organizzandoli nel dizionario Python. Particolare attenzione è rivolta alla conversione delle coordinate geografiche dei poligoni (formato latitude, longitude) in un formato GeoJSON (longitude, latitude) compatibile con lo standard. La trasformazione dell'esempio XML mostrato sopra produce la rappresentazione parziale in dizionario Python in Figura 4.2, evidenziando la sezione geografica convertita in 'geom':

Listing 4.2: Estratto della rappresentazione in dizionario Python

```
{
  "identifier": "Test-Alert-001",
  "sender": "example@example.org",
  "sent": "2025-04-30T12:00:00+00:00",
  "status": "Actual",
  "msgType": "Alert",
  "scope": "Public",
  "info": [
    {
      "areas": [
        {
          "areaDesc": "Building A",
          "polygon": "45.0,9.0 45.0,9.1 45.1,9.1 45.1,9.0
          45.0,9.0",
          "altitude": "10",
          "geom": {
            "type": "Polygon",
```

```
            "coordinates": [
              [
                [9.0, 45.0],
                [9.1, 45.0],
                [9.1, 45.1],
                [9.0, 45.1],
                [9.0, 45.0]
              ]
            ]
          },
          "geometry_type": "Polygon"
        }
      ]
    }
  ]
}
```


Per la rappresentazione completa in dizionario Python del messaggio CAP, si rimanda all'Allegato B, Figura 7.2.

Questa trasformazione permette al microservizio di lavorare con i dati CAP in un formato che Python gestisce facilmente, semplificando le operazioni successive come il filtraggio, la validazione e l'inserimento nel database, senza dover interagire direttamente con la complessità del formato XML originale.

4.3.3 Gestione della configurazione esterna e logica di filtraggio

Il microservizio *Alert Manager* è stato progettato per essere flessibile e adattabile a diverse esigenze operative. Per questo motivo, la logica che determina la rilevanza di un alert e, di conseguenza, se debba essere processato e inoltrato, è gestita attraverso un file di configurazione esterno in formato YAML. Questa scelta architetturale offre numerosi vantaggi: permette agli operatori di modificare i criteri di filtraggio senza dover intervenire sul codice del microservizio, facilitando l'aggiornamento delle politiche di gestione degli allarmi in tempo reale e riducendo il rischio di errori.

Il file di configurazione contiene un insieme di regole predefinite che l'*Alert Manager* utilizza per valutare ogni messaggio CAP ricevuto. Queste regole sono organizzate gerarchicamente e specificano i valori che i vari campi di un alert devono assumere per essere considerati rilevanti. La figura 4.3 mostra un estratto della struttura del file di configurazione.

Listing 4.3: Esempio di configurazione del filtro alert

```
cap_filter:
  event:
    - "Fire"
    - "Earthquake"
    - "Flood"
    - "Hazardous␣Material"
    - "Severe␣Weather"
    - "Power␣Outage"
  urgency:
    - "Immediate"
    - "Expected"
    - "Future"
  area:
    - "Building␣A"
    - "Parking␣Lot"
    - "Surrounding␣Area"
```

Il file raggruppa le regole sotto la chiave principale cap_filter. Ogni sotto-chiave (come event, urgency e area) rappresenta un campo del messaggio CAP che viene utilizzato per il filtraggio. Il valore associato a ciascuna di queste chiavi è una lista di valori accettati. Ad esempio, un alert con event: "Flood" o event: "Fire" sarà considerato rilevante per il criterio "event". Per la configurazione completa del filtro, si rimanda all'Allegato C, Figura 7.3.

Il processo di filtraggio avviene dopo che il messaggio CAP è stato acquisito e convertito nel dizionario Python. L'*Alert Manager* esamina il dizionario dell'alert e lo confronta con le regole definite nel filtro seguendo una logica di tipo "AND" (congiunzione logica) e "OR" (disgiunzione logica):

- Valutazione per campo (logica OR): per ogni campo definito nel file di configurazione, il valore corrispondente nel messaggio CAP deve essere presente in almeno uno dei valori elencati nella lista di configurazione. Ad esempio, se urgency nel CAP è "immediate", il criterio è soddisfatto in quanto presente anche nella lista di configurazione.

- Valutazione complessiva (logica AND): affinchè un alert venga considerato rilevante e superi il filtro, i campi di filtraggio definiti nel file di configurazione devono soddisfare i rispettivi criteri. In altre parole, se il filtro ha regole per event e urgency, un alert sarà rilevante solo se il suo event rientra nei valori accettati e la sua urgency rientra nei valori accettati.

Per una rappresentazione visiva di questa logica di filtraggio, si veda la Figura 4.4, che illustra il flusso di decisione basato sulle regole definite.

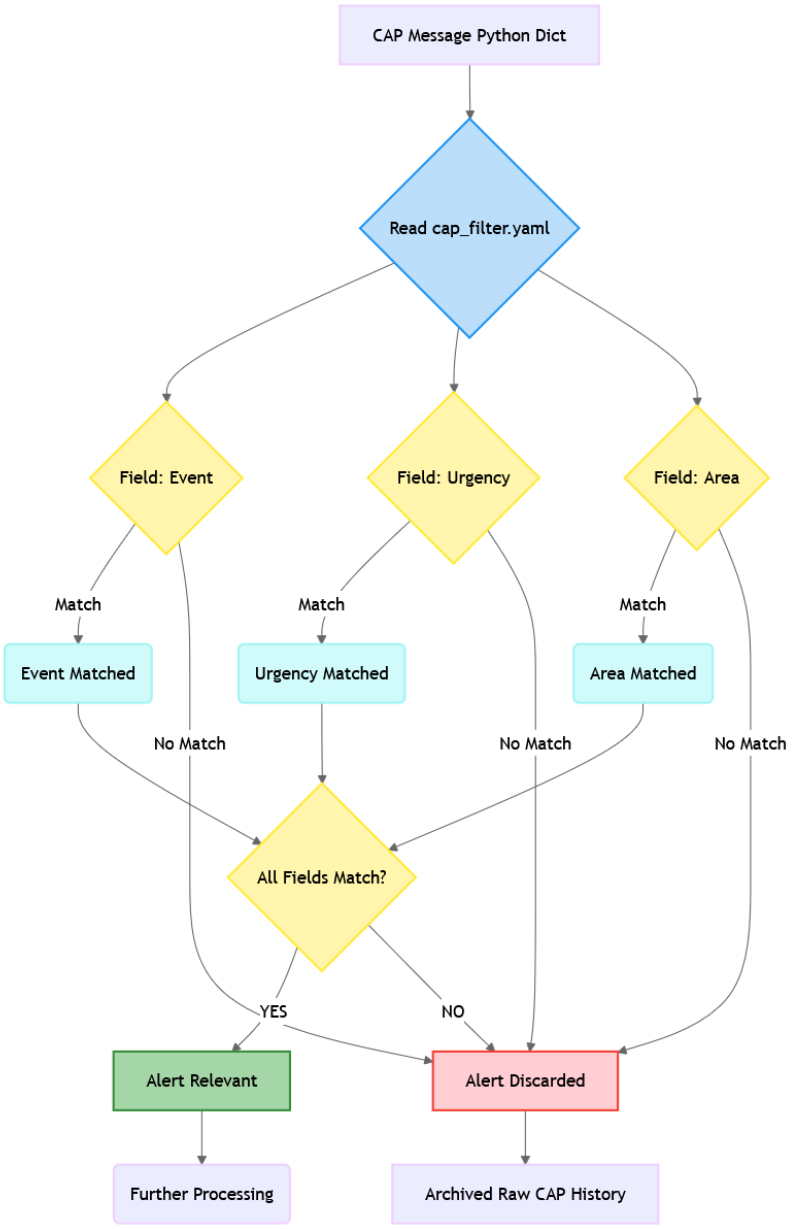

Figura 4.4: Logica di filtraggio degli Alert

Se anche un solo campo dell'alert non corrisponde a nessuno dei valori accettati nella sua lista di riferimento nel file di configurazione, l'intero alert viene scartato dal processo di inoltro e persistenza, ma viene comunque archiviato nella storia dei CAP grezzi.

Questa logica di filtraggio granulare, basata su una configurazione esterna, garantisce che l'*Alert Manager* processerà e inoltrerà solo le informazioni di emergenza che sono effettivamente pertinenti agli interessi e alle politiche di allerta del sistema, riducendo il rumore e concentrando l'attenzione sugli eventi critici.

4.3.4 Persistenza dei dati e archiviazione storica

Una volta che il messaggio CAP è stato elaborato e ha superato la logica di filtraggio, l'*Alert Manager* procede con la persistenza dei dati all'interno di un database relazionale. Questa fase è essenziale per garantire che le informazioni critiche sugli alert siano conservate in modo strutturato, facilitando la loro consultazione, l'analisi successiva e l'integrazione con altri componenti del sistema.

Per la persistenza degli alert filtrati, il microservizio utilizza un database PostgreSQL con l'estensione PostGIS. PostGIS è fondamentale per la gestione efficiente dei dati geospaziali, consentendo di archiviare e interrogare le geometrie delle aree interessate dagli alert (come poligoni e cerchi) in modo nativo. La Figura 4.5 mostra la struttura del database adottata per la persistenza degli alert, articolata in tre tabelle principali.

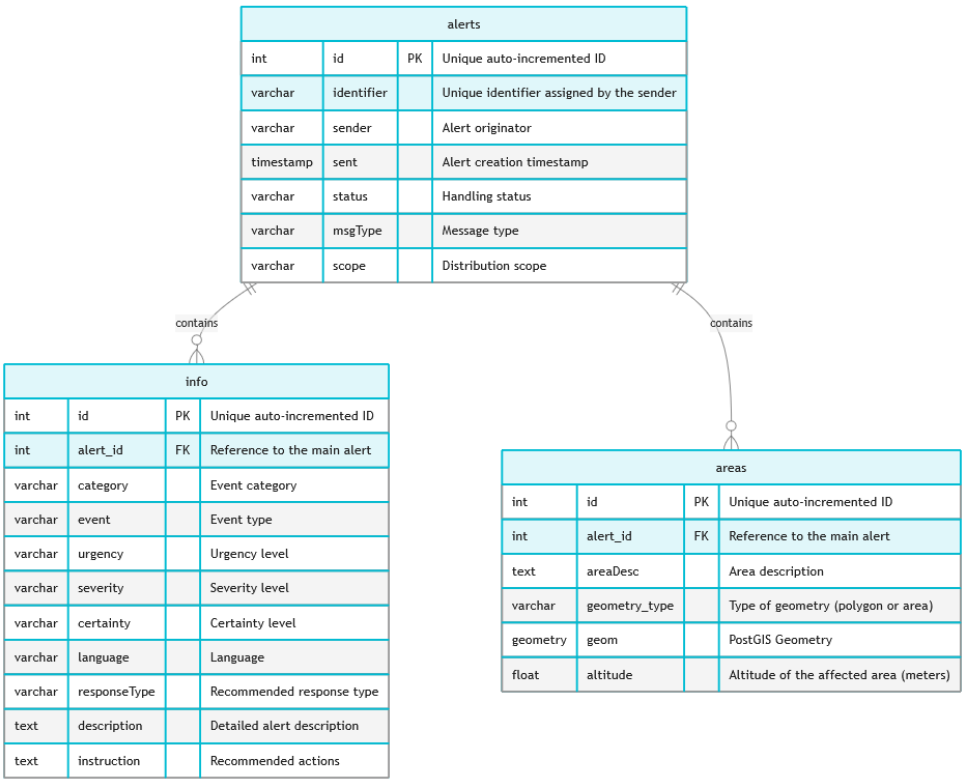

Figura 4.5: Schema semplificato del database per la persistenza degli Alert CAP

Il modello dati è progettato per riflettere la struttura gerarchica del CAP, distribuendo le informazioni principali in tre tabelle relazionate tra loro:

- Tabella alerts: memorizza i metadati principali di ogni messaggio CAP.
- Tabella info: contiene i blocchi informativi aggiuntivi associati a ciascun alert, gestendo la possibilità che un singolo alert contenga uno o più blocchi info.

- Tabella areas: dedicata alle aree geografiche coinvolte nell'alert, supportando le geometrie complesse tramite PostGIS.

L'inserimento dei dati nel database avviene in una transazione atomica. Questo significa che l'intero processo di salvataggio - che abbraccia la tabella alerts e tutte le sue dipendenze come le tabelle info e areas - viene trattata come un'unica e indivisibile unità di lavoro. Di conseguenza, il processo si conclude sempre in uno dei due modi:

- Inserimento con successo: tutti i dati vengono salvati correttamente in ogni tabella coinvolta.

- Annullamento totale (Rollback): al primo errore, non importa quanto piccolo, tutte le modifiche già apportate vengono automaticamente annullate, riportando il database allo stato precedente.

Questo approccio è fondamentale per garantire l'integrità e la coerenza dei dati. Previene scenari in cui solo una parte dell'informazione di un alert viene persistita, lasciando il database in uno stato inconsistente o incompleto.

Oltre alla persistenza dei dati strutturati nel database, l'*Alert Manager* implementa un meccanismo di archiviazione storica di tutti i messaggi CAP ricevuti, indipendentemente dall'esito del filtraggio. Ogni messaggio CAP XML originale viene salvato in un'apposita directory, insieme alla sua corrispondente rappresentazione in formato JSON. Questa archiviazione ha molteplici finalità:

- Tracciabilità completa: permette di conservare una copia esatta di ogni alert così come è stata ricevuta, garantendo una tracciabilità completa nel tempo.

- Audit e debugging: in caso di anomalie o necessità di analisi retrospettiva, è possibile risalire al messaggio originale e verificarne il contenuto esatto.

- Analisi futura: anche gli alert che non superano i criteri di filtraggio correnti possono rivelarsi utili per future analisi di trend o statistiche.

In sintesi, l'*Alert Manager* non solo filtra e struttura i dati degli alert rilevanti, ma mantiene anche un archivio completo e inalterato di tutti i messaggi CAP per finalità di robustezza operativa e analisi a lungo termine.

4.3.5 Instradamento e notifica via RabbitMQ

Dopo aver elaborato e persistito un alert rilevante, il microservizio *Alert Manager* procede con la pubblicazione del messaggio verso un sistema di message brokering basato su

RabbitMQ. Questa fase è cruciale per abilitare la comunicazione asincrona e il disaccoppiamento tra i vari componenti del sistema, garantendo che altri microservizi o applicazioni possano ricevere e reagire agli alert in tempo reale, senza dipendere direttamente dall*Alert Manager* stesso.

Il microservizio utilizza la classe interna AlertProducer per gestire l'interazione con RabbitMQ. Questa classe si connette al broker utilizzando le credenziali definite nella configurazione del sistema. Il modello di comunicazione, in questo specifico scenario di notifica, prevede che l'*Alert Manager* agisca come publisher inviando i messaggi direttamente a una coda specifica.

La Figura 4.6 illustra il flusso di un messaggio di alert attraverso RabbitMQ, evidenziando il ruolo dell'*Alert Manager* come publisher e del *Notification Center* come consumer.

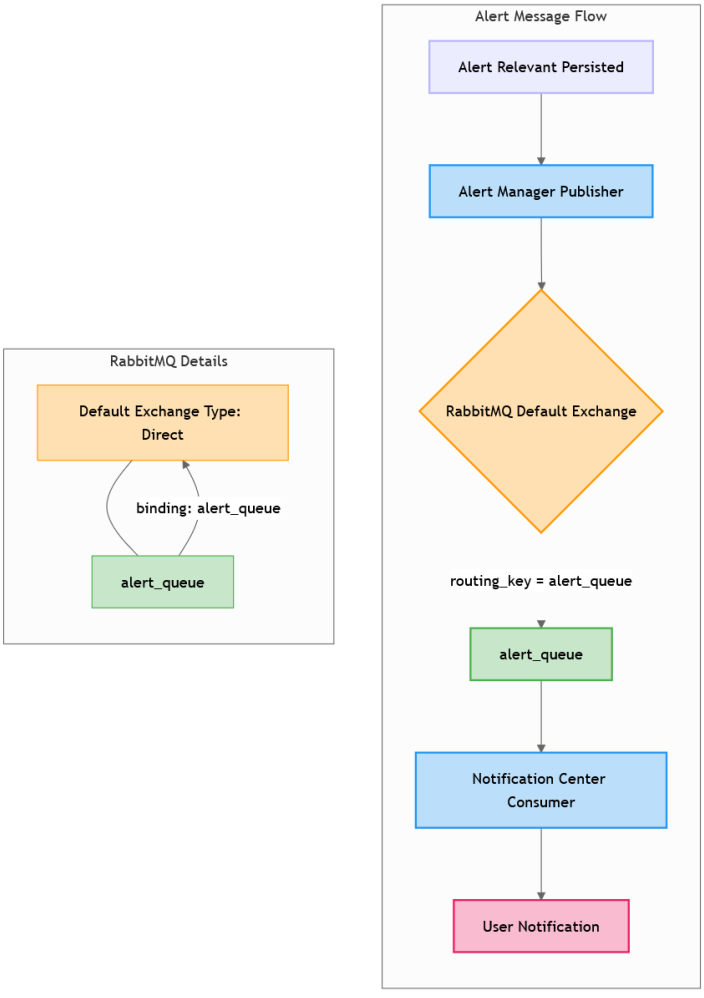

Figura 4.6: Flusso del Messaggio di Alert tramite RabbitMQ

A differenza di un modello publish/subscribe tramite un fanout exchange, l'*Alert Manager* sfrutta il Default Exchange di RabbitMQ. Questo è uno scambio implicito di tipo direct a cui tutte le code appena create sono automaticamente legate con una routing_key

pari al loro nome. Quando l'*Alert Manager* invia un messaggio, lo indirizza a una routing_key corrispondente al nome della coda di destinazione. Per le notifiche di alert, la coda predefinita è alert_queue. È importante sottolineare che la dichiarazione e la gestione di questa coda sono responsabilità del *Notification Center*, il microservizio designato a consumare questi messaggi e a gestirne la notifica finale agli utenti. Il flusso di messaggi è il seguente: una volta che un alert ha superato il filtro e i suoi dati sono stati correttamente salvati nel database, l'*Alert Manager* formatta l'alert in un messaggio e lo invia direttamente alla coda alert_queue. Per garantire la durabilità del messaggio e la sua sopravvivenza in caso di riavvii del broker RabbitMQ, i messaggi vengono inviati con l'attributo persistent=True.

Il formato del messaggio inviato su RabbitMQ è il dizionario Python dell'alert processato, convertito in formato JSON. Questo garantisce che tutte le informazioni strutturate dell'alert, così come estratte e validate, siano disponibili per i servizi consumatori in un formato standard e facilmente parsabile. L'utilizzo di JSON facilita l'interoperabilità tra servizi sviluppati in linguaggi diversi, mantenendo al contempo un'alta leggibilità dei dati.

In sintesi, l'integrazione con RabbitMQ permette all'*Alert Manager* di operare come una componente centrale di elaborazione degli alert, distribuendo le informazioni rilevanti in modo efficiente e disaccoppiato verso il *Notification Center*, che si occuperà delle fasi successive di notifica. Il tutto è supportato da un sistema di logging che traccia ogni operazione di invio.

4.3.6 Sistema di logging

Un sistema di logging robusto è un componente indispensabile per qualsiasi microservizio, specialmente in un'architettura distribuita come quella dell'*Alert Manager*. La sua funzione principale è registrare gli eventi significativi, le operazioni svolte e gli eventuali errori che si verificano durante l'esecuzione del servizio. Questo non solo facilita le attività di debugging e monitoraggio in fase di sviluppo e produzione, ma fornisce anche una traccia storica cruciale per audit, analisi delle prestazioni e conformità normativa.

Il microservizio *Alert Manager* integra un sistema di logging basato sulla libreria standard di Python, logging. La configurazione di questo sistema è centralizzata nel modulo logger.py, che gestisce la creazione e la personalizzazione dell'istanza del logger. La configurazione prevede la creazione automatica di una directory logs/ all'interno della struttura del progetto se questa non esiste. Successivamente, viene inizializzata un'istanza del logger con il nome AlertManager.

Esempi di eventi che vengono sistematicamente loggati dall'Alert Manager includono:

- Successo o fallimento delle connessioni al database: registrazioni di INFO in caso di connessione riuscita e ERROR in caso di problemi di connessione.
- Parsing e conversione dei messaggi CAP: messaggi INFO che tracciano l'estrazione dei campi e avvisi WARNING per campi obbligatori mancanti o vuoti nei messaggi CAP.
- Esito delle operazioni di filtraggio: messaggi che indicano se un alert è stato considerato rilevante o scartato.
- Stato dell'invio dei messaggi RabbitMQ: INFO per l'invio riuscito degli alert alla coda alert_queue e ERROR in caso di problemi durante la pubblicazione.

Questo approccio al logging assicura che l'*Alert Manager* fornisca una visione completa del suo funzionamento interno, facilitando la diagnosi e la risoluzione dei problemi, e garantendo la piena osservabilità delle sue operazioni.

4.4 Conclusioni sul gestore degli alert

Questo capitolo ha illustrato in dettaglio il microservizio *Alert Manager*, un componente fondamentale nell'architettura del sistema di notifica di emergenza. Il suo ruolo principale è la gestione coordinata degli allarmi, partendo dalla loro acquisizione fino all'instradamento delle notifiche. L'analisi dello stato dell'arte ha chiarito come le soluzioni esistenti per il monitoraggio IT (come Prometheus Alertmanager, Zabbix e Nagios) e quelle specifiche per la gestione di edifici (BMS) non offrano un supporto nativo al Common Alerting Protocol (CAP). Questa lacuna ha guidato la scelta progettuale di sviluppare una soluzione personalizzata, capace di interpretare e generare alert conformi a questo standard internazionale, garantendo interoperabilità e una semantica precisa per gli eventi di emergenza.

L'implementazione dell'*Alert Manager*, sviluppata in Python, è stata presentata attraverso le sue componenti principali e il flusso di lavoro degli alert. Abbiamo esaminato il processo di parsing e validazione dei messaggi CAP, che trasforma i dati XML grezzi in una struttura Python facilmente gestibile, pronta per le successive elaborazioni. Un aspetto cruciale è la logica di filtraggio, gestita tramite un file di configurazione YAML esterno. Questa configurazione permette di definire criteri granulari basati su campi specifici del CAP (come event, urgency, severity, area e altri), garantendo che solo gli alert pertinenti vengano processati ulteriormente. La combinazione di logiche "AND" e "OR" applicate a questi criteri assicura flessibilità e precisione nel determinare la rilevanza di un alert.

La persistenza dei dati e l'archiviazione storica sono state dettagliate come pilastri per la robustezza del sistema. Gli alert che superano il filtraggio vengono salvati in un database PostgreSQL con l'estensione PostGIS, fondamentale per la gestione efficiente delle informazioni geospaziali. Il modello dati, articolato nelle tabelle alerts, info e areas, riflette fedelmente la struttura gerarchica del CAP. L'utilizzo di transazioni atomiche e indici sulle chiavi esterne garantisce rispettivamente l'integrità dei dati e l'ottimizzazione delle prestazioni. Parallelamente, ogni messaggio CAP originale, indipendentemente dal suo filtraggio, viene archiviato in formato XML e JSON, fornendo una tracciabilità completa e risorse preziose per audit e future analisi.

L'instradamento e la notifica via RabbitMQ rappresentano l'ultimo passaggio nel ciclo di vita dell'alert all'interno di questo microservizio. L'*Alert Manager* agisce come publisher, inviando i messaggi (alert in formato JSON) direttamente alla coda alert_queue del Default Exchange di RabbitMQ. Questa scelta garantisce un disaccoppiamento efficace tra l'*Alert Manager* e il *Notification Center* (responsabile della gestione della coda), promuovendo scalabilità e resilienza. La persistenza dei messaggi (persistent=True) assicura inoltre la loro durabilità anche in caso di riavvii del broker.

Infine, il sistema di logging, basato sulla libreria standard logging di Python, fornisce una visione completa delle operazioni interne del microservizio. Con livelli di logging differenziati per console (INFO) e file (DEBUG), e un formato uniforme per i messaggi, facilita enormemente il debugging, il monitoraggio e la manutenza del sistema.

In sintesi, il microservizio *Alert Manager* è stato progettato e implementato come una soluzione robusta, flessibile e interoperabile per la gestione degli alert CAP. Le sue capacità di parsing, filtraggio configurabile, persistenza sicura e integrazione asincrona tramite RabbitMQ, supportate da un logging completo, lo rendono un componente critico ed efficace in un sistema di notifica di emergenza, capace di adattarsi a requisiti dinamici e di operare in scenari complessi con alta affidabilità.

Capitolo 5

Microservizio simulatore delle posizioni

Questo capitolo è dedicato all'analisi approfondita del microservizio *User Simulator*, un componente chiave ma transitorio nell'architettura del sistema di gestione delle emergenze. Sebbene la sua funzione sia quella di emulare il comportamento e la distribuzione spaziale degli utenti, il suo design e la sua implementazione sono stati concepiti per replicare fedelmente le dinamiche che un sistema di rilevamento in tempo reale dovrà un giorno gestire. Nel corso di questo capitolo, verranno esplorati in profonfità gli aspetti cruciali di questo microservizio, fornendo una visione completa della sua funzione e del suo impatto sull'architettura:

- Il ruolo e lo scopo specifico del simulatore all'interno dell'ecosistema. Si analizzerà la posizione strategica del simulatore all'interno dell'architettura a microservizi e la sua funzione primaria nel processo di sviluppo e validazione.

- La sua architettura interna e le tecnologie impiegate per la sua realizzazione. Verranno spiegate le scelte architetturali e le tecnologie, giustificandone l'adozione.

- Come vengono gestiti e configurati i dati relativi agli utenti e all'ambiente.

- Il flusso operativo completo, dalla gestione degli utenti alla gestione delle diverse fasi di movimento (normale e di emergenza).

- I meccanismi di comunicazione con gli altri microservizi. Verranno illustrati le modalità e i protocolli di interazione con gli altri microservizi, sottolineando il ruolo del simulatore come fonte di dati per gli altri microservizi.

5.1 Introduzione e funzionalità specifiche

Nel contesto dell'architettura a microservizi adottata, lo *User Simulator* ricopre un ruolo di rilevo, sebbene temporaneo. La sua funzione principale è emulare, con rigore metodologico, il comportamento e la distribuzione spaziale degli utenti all'interno degli ambienti monitorati. È fondamentale comprendere che, pur essendo una soluzione di supporto destinata a essere sostituita da un meccanismo di rilevamento in tempo reale, la sua accurata implementazione è cruciale per la validazione delle logiche di emergenza dell'intero sistema. Due responsabilità fondamentali sono assegnate allo *User Simulator*:

- Generazione e gestione dinamica delle posizioni degli utenti: il simulatore crea un numero configurabile di utenti virtuali e ne gestisce il movimento all'interno di un modello astratto dell'edificio. Questo movimento può essere casuale in condizioni normali o guidato da percorsi specifici durante un'emergenza.

- Comunicazione coerente delle posizioni: si occupa di inviare le posizioni aggiornate degli utenti al *Position Manager* in modo granulare (una posizione per utente alla volta), replicando l'interfaccia che un futuro sistema di tracciamento reale dovrebbe fornire. Riceve inoltre istruzioni dal *Notification Center* per modificare il comportamento degli utenti simulati in base allo stato dell'emergenza.

5.2 Analisi dello stato dell'arte e motivazione delle scelte implementative

La progettazione e l'implementazione del microservizio *User Simulator* sono state il risultato di un'attenta valutazione delle opzioni disponibili, ponderando le esigenze specifiche del nostro sistema a microservizi e confrontando le soluzioni esistenti nel panorama della simulazione. Sebbene l'obiettivo primario del simulatore sia quello di fungere da strumento di test e validazione in assenza di dati di posizione reali, le scelte tecnologiche e architetturali sono state dettate dalla ricerca di efficienza, accuratezza e integrazione con l'ecosistema esistente.

5.2.1 Necessità strategica di un simulatore dedicato

In un'architettura complessa, dove molteplici componenti interagiscono dinamicamente, la disponibilità di flussi di dati realistici è fondamentale per le fasi di sviluppo, test e ottimizzazione. Per il nostro sistema di gestione delle emergenze, dipendente dalle posizioni degli utenti per attivare logiche di allerta e guidare percorsi di evacuazione, la

mancanza di un sistema di tracciamento hardware in tempo reale in fase di prototipazione ha reso indispensabile l'introduzione di un simulatore. Questo componente non è una semplice soluzione provvisoria, ma si configura come un elemento chiave che permette di:

- Sbloccare lo sviluppo concorrente: permette ai microservizi dipendenti dalle posizioni - quali il *Position Manager* e il *Map Viewer* - di progredire indipendentemente dal sistema di rilevamento fisico. Questo accelera il ciclo di sviluppo complessivo e riduce le dipendenze critiche.

- Garantire scenari di test controllati e ripetibili: la capacità di generare profili di movimento utente predefiniti o casuali, e di innescare eventi di allerta in condizioni riproducibili, è essenziale per la validazione rigorosa delle logiche di business del sistema. A differenza di dati reali, che sono imprevedibili, il simulatore offre un ambiente deterministico per identificare e risolvere anomalie.

- Valutare le performance e la scalabilità: regolando il numero di utenti simulati e la frequenza di aggiornamento delle posizioni, è possibile sottoporre il sistema a carichi di lavoro variabili. Questo fornisce indicazioni preziose sulla scalabilità dei microservizi a valle (come capacità di elaborazione del *Position Manager* o del *Notification Center*) e sulla resilienza dell'infrastruttura di messaggistica. [29][30]

La realizzazione di un simulatore dinamico è stata preferita all'uso di dati statici o mock pre-registrati. Sebbene questi ultimi siano più semplici da implementare, avrebbero limitato fortemente la capacità di testare le reazioni in tempo reale del sistema a variazioni continue delle posizioni e non avrebbero permesso la simulazione di scenari complessi, come l'interazione con percorsi di evacuazione dinamici. Un simulatore attivo, pur nella sua natura transitoria, replica in modo più fedele il comportamento di una fonte di dati reale, facilitando la futura transizione e minimizzando l'impatto architetturale.

5.2.2 Confronto con framework e librerie di simulazione generiche

Nel vasto e diversificato panorama delle soluzioni software, esistono numerose categorie di strumenti e librerie dedicate alla simulazione, ognuna caratterizzata da un proprio focus e un livello di astrazione. Per la modellazione di comportamenti di entità discrete in un ambiente, in Python si distinguono in particolare le seguenti tipologie:

- Framework di Agent-Based modeling (ABM): Tra gli esempi più noti figurano Mesa[31] e NetLogo (spesso con estensioni Python)[32]. Questi strumenti sono eccellenti per costruire simulazioni dove l'attenzione è rivolta al comportamento

emergente di agenti autonomi che interagiscono tra loro e con l'ambiente. Essi offrono spesso capacità di visualizzazione complesse e strumenti di analisi statistica integrati.

- Piattaforme di simulazione integrate: strumenti come AnyLogic[33] o GAMA[1][34] rapresentano piattaforme di modellazione e simulazione complete. Queste supportano tipicamente più paradigmi (ABM, simulazione a eventi discreti, dinamica dei sistemi) e offrono interfacce grafiche avanzate, librerie predefinite per specifici domini (es. traffico, pedoni) e capacità di visualizzazione e analisi complesse. Sono soluzioni potenti per studi di ricerca o per la prototipazione di sistemi su larga scala.

- Librerie per la manipolazione e l'analisi di grafi: librerie come NetworkX[35] offrono potenti strumenti per la creazione, manipolazione e studio di strutture a grafo. Sebbene non siano framework di simulazione nel senso stretto, consentono di modellare reti complesse

- Librerie per animazioni e motori fisici: sebbene meno pertinenti per il nostro scopo, esistono librerie (es. Pygame per semplici simulazioni 2D o wrappers per motori fisici più complessi) che consentono di animare oggetti e simulare interazioni fisiche in ambienti grafici, utili per contesti con requisiti visivi.[36]

La scelta di procedere con un'implementazione custom dello *User Simulator*, piuttosto che adottare o adattare uno di questi framework generici, è stata dettata da una serie di motivazioni pragmatiche e di allineamento con gli obiettivi specifici del progetto:

- Riduzione dell'overhead funzionale e di complessità: framework come Mesa o Simpy[37], così come piattaforme integrate quali Anylogic o GAMA, pur essendo potenti e versatili, introducono un significativo overhead in termini di funzionalità non strettamente necessarie per il nostro specifico scopo. Essi sono spesso orientati alla ricerca accademica o alla modellazione di fenomeni complessi, offrendo capacità (come motori di rendering avanzati, collezionisti di dati statistici avanzati, gestione di agenti con apprendimento) che avrebbero appesantito il progetto senza un beneficio diretto e proporzionato alle nostre esigenze. L'integrazione, l'apprendimento e la configurazione di tali sistemi avrebbero incrementato la curva di apprendimento e la complessità di manutenzione del software, distogliendo risorse dal core del problema.

- Massimizzazione della flessibilità e adattabilità al dominio specifico: lo *User Simulator* è intrinsecamente legato a specifiche del nostro sistema: deve leggere la topologia dell'edificio da un database PostGIS, interpretare percorsi di evacuazione

[1]GAMA - General Agent-based Modeling Architecture

come sequenze di archi su un grafo e comunicare tramite RabbitMQ con payload di messaggi precisi. Adattare un framework generico a queste interfacce I/O e a un modello di movimento così specifico (basato su nodi e archi di un edificio reale) avrebbe richiesto uno sforzo di personalizzazione potenzialmente maggiore rispetto allo sviluppo di una soluzione mirata. L'approccio custom ha permesso un controllo granulare su ogni aspetto del ciclo di vita dell'utente simulato, dal respawn alla navigazione guidata.

- Focus sulla generazione di dati per il testing di sistema: l'obiettivo primario del simulatore non è l'analisi scientifica del comportamento di una folla o la predizione di fenomeni complessi, ma l'erogazione di un flusso di dati di posizione coerente e controllabile per testare l'efficacia e la robustezza degli altri microservizi. Una soluzione custom ha consentito di concentrare le risorse sullo sviluppo di una logica di simulazione essenziale e performante per questo scopo, senza la necessità di implementare caratteristiche che avrebbero spostato il focus dalla sua funzione di validatore di sistema.

- Integrazione nell'ecosistema a microservizi: data la predominanza di Python come linguaggio di sviluppo e l'adozione di RabbitMQ come message broker standard nell'architettura complessiva, la costruzione di un simulatore specifico ha garantito un'integrazione senza frizioni con l'architettura esistente. Questo ha facilitato la gestione delle dipendenze, la configurazione e il deployment del servizio.

5.2.3 Conclusioni sull'analisi dello stato dell'arte

L'approfondita analisi delle alternative e delle motivazioni ha chiarito come la scelta di sviluppare un microservizio *User Simulator* custom sia stata la più coerente e strategica per le esigenze del progetto. Le soluzioni di simulazione generiche, pur valide per i loro scopi, avrebbero introdotto un onere eccessivo in termini di complessità e funzionalità superflue, e non avrebbero offerto la flessibilità necessaria per integrarsi nativamente con le specifiche architetturali e di dominio del nostro sistema. L'implementazione mirata ha consentito di creare uno strumento leggero, efficiente e perfettamente allineato al suo ruolo di generatore di dati per il testing e la validazione degli altri microservizi, gettando le basi per una futura sostituzione trasparente con un sistema di rilevamento reale.

5.3 Sviluppo operativo del microservizio

Il microservizio *User Simulator* è progettato per replicare il movimento e il comportamento degli utenti all'interno di un ambiente mappato. Il suo scopo è simulare

dinamicamente le risposte degli utenti a eventi specifici e a condizioni ambientali mutevoli. L'implementazione si fonda su un'architettura modulare, caratterizzata da componenti distinti che si occupano di configurazione, interazione con il database, comunicazione asincrona e la logica core della simulazione. Questo approccio favorisce una chiara separazione delle responsabilità, promuovendo la scalabilità e la manutenibilità del sistema.

5.3.1 Componenti principali e flusso di lavoro del simulatore

Il microservizio *User Simulator*, sviluppato interamente in Python, è l'elemento chiave del sistema per la simulazione del comportamento degli utenti. La sua architettura è stata concepita per garantire robustezza e flessibilità in ogni fase della simulazione, dalla generazione iniziale degli utenti alla gestione delle loro reazioni a eventi esterni. La sua struttura è schematizzata nella Figura 5.1, offrendo una visione d'insieme dell'organizzazione del progetto.

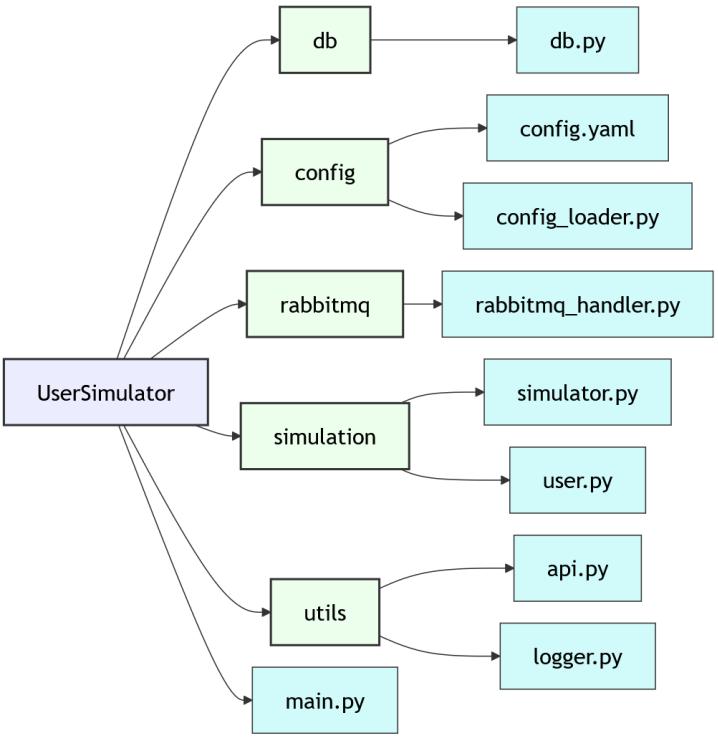

Figura 5.1: Struttura interna del microservizio *User Simulator*

Il microservizio è organizzato in diverse componenti principali, ciascuna con una responsabilità specifica. Queste componenti lavorano insieme in sinergia per realizzare il flusso operativo del simulatore, orchestrato dal modulo principale main.py. Il processo si sviluppa attraverso una pipeline ben definita, pensata per garantire che ogni utente simulato sia sistematicamente inizializzato, posizionato e aggiornato nel suo stato e mo-

vimento. Questo approccio assicura anche che le sue interazioni con l'ambiente e gli altri servizi siano gestite in modo coerente, come illustrato in Figura 5.2.

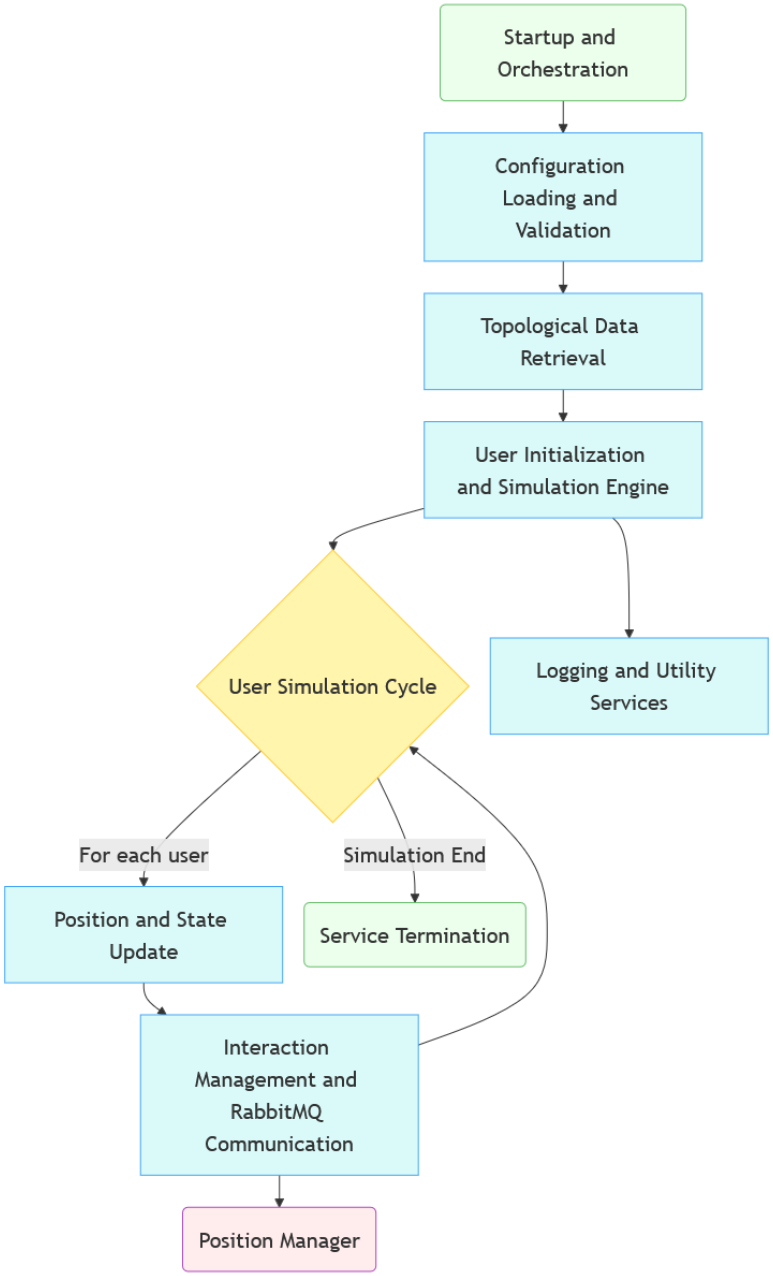

Figura 5.2: Flusso interno del microservizio *User Simulator*

Analizziamo ora in dettaglio le componenti principali che rendono possibile questo flusso:

- Modulo di avvio e orchestrazione: questa componente funge da punto di ingresso del microservizio, gestendo l'inizializzazione e l'orchestrazione delle altre parti.
- Modulo di gestione della configurazione: si occupa di caricare, validare e interpretare tutti i parametri operativi del simulatore.

- Modulo di interazione con il database: questa componente gestisce la connessione al database PostgreSQL per il recupero delle informazioni topologiche dell'ambiente.

- Modulo di comunicazione asincrona: dedicato alla gestione di tutte le interazioni con il message broker RabbitMQ.

- Modulo core di simulazione: questo è il cuore logico del simulatore, includendo le classi che modellano gli utenti e il motore che ne gestisce il comportamento.

- Modulo di logging e servizio: raccoglie funzioni di supporto generiche che non rientrano direttamente nelle logiche principali, ma sono indispensabili per il funzionamento del microservizio.

5.3.2 Modulo di avvio e orchestrazione

Il file main.py non è solo il punto di avvio del microservizio, ma è anche il suo orchestratore centrale. La sua funzione principale è quella di inizializzare l'applicazione FastAPI, un framework web leggero e performante che espone gli endpoint del simulatore per consentire l'interrogazione esterna delle posizioni simulate. Successivamente, si occupa di caricare tutti i parametri di configurazione essenziali tramite il modulo di gestione della configurazione, garantendo che il simulatore operi secondo le impostazioni desiderate.

Un ruolo cruciale di questa componente è quello di stabilire le connessioni vitali con le risorse esterne: il database PostgreSQL, dal quale vengono letti i dati topologici della mappa, e il message broker RabbitMQ, fondamentale per la comunicazione in tempo reale con gli altri microservizi del sistema. Dopo aver assicurato tutte le connessioni, main.py avvia i thread in background dedicati alla logica di simulazione vera e propria e alla gestione asincrona dei messaggi di RabbitMQ. Questo approccio basato sui thread consente al simulatore di eseguire simultaneamente il suo motore logico e di rimanere in ascolto per eventi esterni senza bloccare l'interfaccia API. In sintesi, main.py è il catalizzatore che rende l'intero simulatore funzionale, coordinando l'attivazione e l'interazione di tutti i suoi sottosistemi, come mostrato in Figura 5.3.

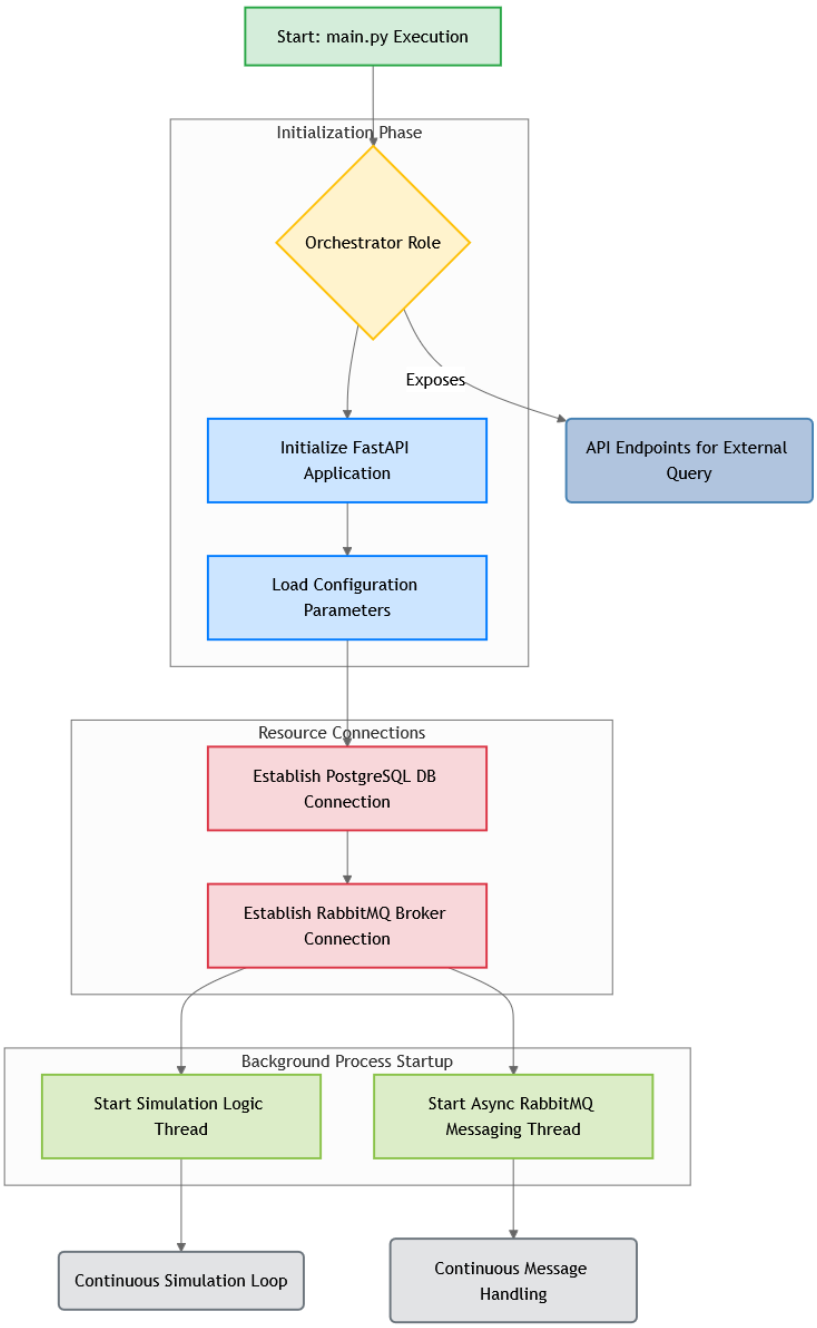

Figura 5.3: Flusso del modulo di avvio e orchestrazione

5.3.3 Modulo di gestione della configurazione

Il modulo di gestione della configurazione è fondamentale per l'adattabilità e la robustezza del microservizio. Si compone di due parti:

- Un file con estensione YAML (config.yaml): rappresenta il file di configurazione primario che contiene la definizione di tutti i parametri operativi del simulatore. La scelta del formato YAML garantisce una leggibilità elevata e una struttura chiara per la definizione delle impostazioni.

- Un modulo Python (config_loader.py): rappresenta il componente software responsabile del caricamento, della validazione e dell'interpretazione dei parametri definiti nel file di configurazione. Assicura che il simulatore operi sempre in accordo con le impostazioni previste.

Il file di configurazione è un file in cui vengono definiti tutti i parametri utili al sistema, ad esempio, il numero di utenti da generare, le velocità di movimento differenziate (una speed_normal per i comportamenti di routine e una speed_alert per le situazioni di emergenza), e l'intervallo di tempo (simulation_tick) che intercorre tra un aggiornamento e l'altro della simulazione. Un elemento distintivo e particolarmente utile è la capacità di definire time_slots (fasce orarie), ognuna delle quali è associata a una specifica distribuzione probabilistica di tipi di nodo (es. "aula", "caffetteria", "ufficio"). Questo meccanismo permette al simulatore di posizionare in modo realistico gli utenti in diverse aree della mappa a seconda dell'ora del giorno. Per esempio, è possibile simulare una maggiore presenza di persone nelle aule durante le lezioni o nelle aree comuni durante le pause, riflettendo accuratamente le dinamiche di un ambiente reale. Infine, i dettagli essenziali per la connessione al message broker RabbitMQ, come l'host, le porte e i nomi delle code, sono anch'essi configurati qui, centralizzando la gestione delle risorse esterne.

Un estratto rappresentativo del file config.yaml, che illustra la struttura e alcuni dei parametri chiave menzionati, è riportato nel Listato 5.1.

Listing 5.1: Estratto del file di configurazione (config.yaml)

```yaml
rabbitmq:
  host: "localhost"
  port: 5672
  username: "guest"
  password: "guest"
  alert_queue: "user_simulator_queue"
  evacuation_paths_queue: "evacuation_paths_queue"
  position_queue: "position_queue"

n_users: 4
speed_normal: 20.0
speed_alert: 350.0
simulation_tick: 1.0
timeout_after_stop: 15

time_slots:
  - name: "morning_class_1"
    start: "08:30"
```

```
    end: "10:30"
    distribution:
      classroom: 0.6
      corridor: 0.1
      coffee shop: 0.05
      canteen: 0.05
      office: 0.1
      bathroom: 0.05
      stairs: 0.03
      outdoor: 0.02

  - name: "morning_break"
    start: "10:30"
    end: "11:00"
    distribution:
      classroom: 0.05
      corridor: 0.15
      coffee shop: 0.3
      canteen: 0.3
      office: 0.05
      bathroom: 0.1
      stairs: 0.03
      outdoor: 0.02
```

Per una consultazione completa del file di configurazione, si rimanda all'Appendice D, Figura 7.4

Il componente software responsabile del caricamento del file di configurazione non si limita alla semplice lettura di tale file. Il suo compito include la logica necessaria per analizzare le fasce orarie e fornire al motore di simulazione le distribuzioni appropriate per la posizione iniziale degli utenti in base all'orario corrente. Questo assicura che il comportamento del simulatore si adatti dinamicamente agli scenari temporali predefiniti, garantendo che i dati di configurazione siano sempre validi e coerenti con le aspettative operative del sistema.

5.3.4 Modulo di interazione con il database

Il modulo di interazione con il database è la componente essenziale per accedere alle informazioni spaziali e topologiche dell'ambiente simulato. All'interno di questo modulo, la classe DB incapsula tutta la logica necessaria per la connessione al database PostgreSQL e per il recupero dei dati della mappa. I dati cruciali che vengono letti sono i nodi

(che rappresentano aree specifiche o punti di interesse nell'ambiente, come aule, corridoi, uscite di sicurezza) e gli archi (che definiscono le connessioni percorribili tra questi nodi, modellando percorsi pedonali, scale, ecc.). Questi dati, recuperati dal database, sono poi utilizzati dal motore di simulazione per guidare il movimento degli utenti simulati all'interno della mappa. Di seguito, la Figura 5.4 mostra la struttura delle tabelle a cui accede lo *User Simulator*.

nodes			
int	node_id	PK	Unique node identifier
int	x1		X-coordinate boundary 1
int	x2		X-coordinate boundary 2
int	y1		Y-coordinate boundary 1
int	y2		Y-coordinate boundary 2
int	z1		Z-coordinate boundary 1
int	z2		Z-coordinate boundary 2
int	floor_level		Node's floor level
int	capacity		Max occupancy capacity
varchar	node_type		Type of node (e.g., classroom)
int	current_occupancy		Current number of people
int[]	evacuation_path		Array of evacuation node IDs

arcs			
int	arc_id	PK	Unique arc identifier
int	flow		Current flow through arc
interval	traversal_time		Time to traverse arc
boolean	active		Is arc currently active?
int	x1		X-coordinate start point
int	x2		X-coordinate end point
int	y1		Y-coordinate start point
int	y2		Y-coordinate end point
int	z1		Z-coordinate start point
int	z2		Z-coordinate end point
int	capacity		Max capacity of the arc
int	initial_node	FK	Reference to starting node (nodes.node_id)
int	final_node	FK	Reference to ending node (nodes.node_id)

Figura 5.4: Tabelle nodes e arcs del database dedicato alla rappresentazione dell'edificio

È importante sottolineare che lo *User Simulator* non ha alcuna facoltà di modificare, aggiungere o eliminare dati relativi alla mappa. La responsabilità di creare, popolare e mantenere aggiornato il database con tutte le informazioni cartografiche spetta interamente a un microservizio esterno dedicato: il *Map Viewer*. Questa chiara separazione delle responsabilità garantisce l'integrità dei dati della mappa e assicura che il simulatore sia un "consumatore" passivo e affidabile delle informazioni topologiche.

5.3.5 Modulo di comunicazione asincrona

Il modulo di comunicazione asincrona è dedicato interamente alla gestione di tutte le interazioni del microservizio attraverso il message broker RabbitMQ. Questa componente è cruciale per l'integrazione del simulatore nell'architettura complessiva del sistema di gestione delle emergenze. La classe RabbitMQHandler è il fulcro di questo sistema di comunicazione. Essa gestisce l'intera lifecycle della connessione e delle operazioni con RabbitMQ: dalla connessione al server del broker, alla dichiarazione delle code necessarie sia per l'invio che per la ricezione dei messaggi, fino all'implementazione della logica di consumo e pubblicazione. Questo handler è specificamente configurato per ascoltare

messaggi provenienti da altri microservizi, quali ad esempio i messaggi di "allerta" (che innescano un cambio di stato nel simulatore e negli utenti) e i percorsi di evacuazione, essenziali per guidare il movimento degli utenti in caso di emergenza. Contemporaneamente, la RabbitMQHandler ha il compito fondamentale di pubblicare in modo continuo e regolare le posizioni aggiornate di tutti gli utenti simulati. Queste informazioni vengono inviate a code dedicate, rendendole immediatamente disponibili ad altri microservizi interessati, come il *Position Manager*, che elabora i dati di posizione per l'intero sistema. L'uso di RabbitMQ permette di disaccoppiare lo *User Simulator* dagli altri servizi, migliorando la resilienza, la scalabilità e la flessibilità dell'intera architettura, come mostrato in Figura5.5.

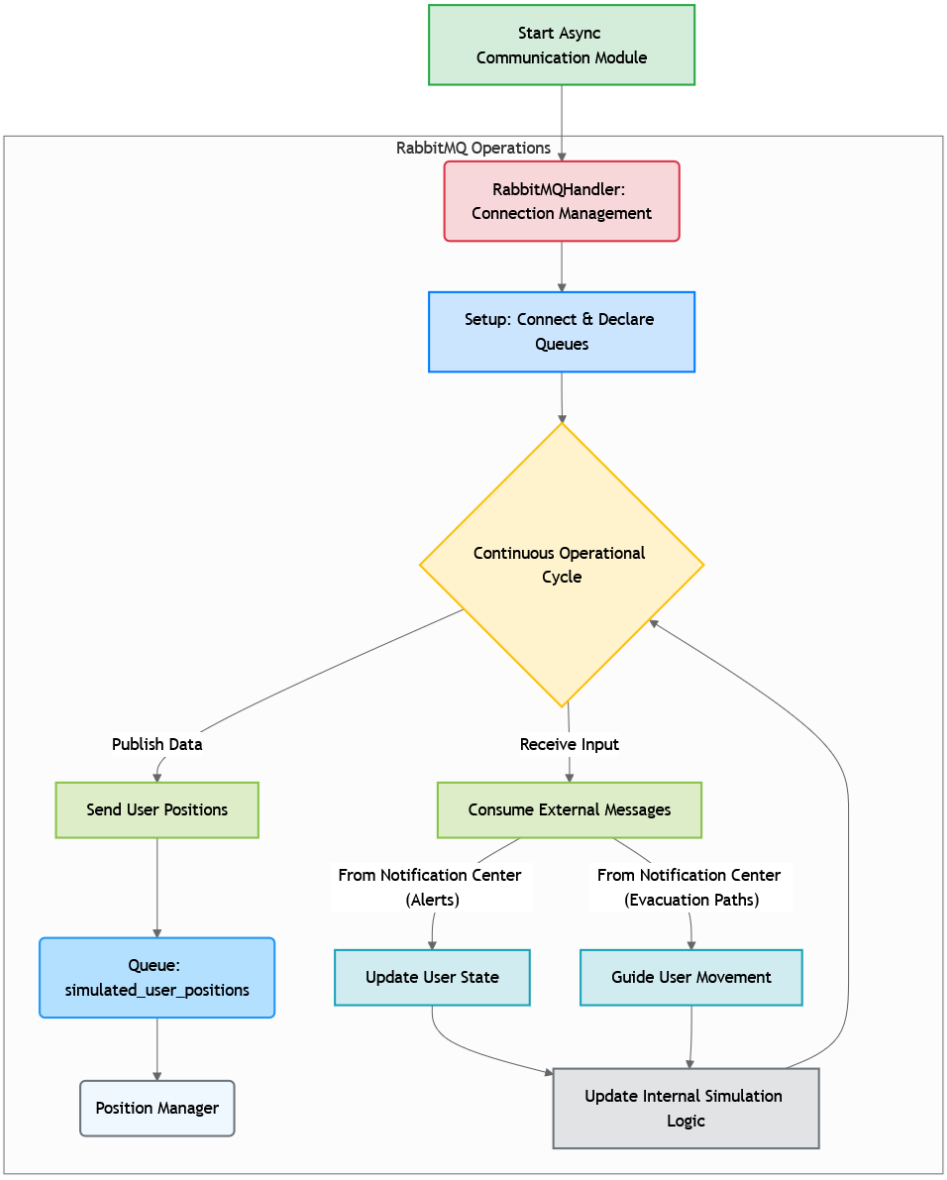

Figura 5.5: Flusso delle posizioni tramite RabbitMQ

5.3.6 Modulo core di simulazione

Il modulo di simulazione rappresenta il cuore operativo del microservizio *User Simulator*, contenendo la logica fondamentale che governa la simulazione degli utenti. È qui che avviene la generazione degli utenti, il loro movimento dinamico e la gestione dei loro stati in risposta a vari stimoli. Questo modulo integra anche il modello dati che descrive ciascun utente e le logiche che ne regolano il comportamento. Contiene due classi principali:

- La classe Simulator: è il motore principale della simulazione. È responsabile di inizializzare il pool di oggetti User, ognuno dei quali rappresenta un individuo simulato all'interno dell'ambiente. Questa classe gestisce l'avanzamento temporale della simulazione attraverso "tick" regolari, che fungono da unità di tempo discrete. Durante ogni "tick", il Simulator orchestra l'aggiornamento delle posizioni di tutti gli utenti in base alle loro logiche di movimento e allo stato corrente. È anche il punto di coordinamento per la reazione agli eventi esterni, come l'attivazione di stati di allerta (che possono modificare drasticamente il comportamento degli utenti) o la ricezione di comandi di "stop" che indicano la fine di un'emergenza o l'arresto della simulazione. Il Simulator inoltre delega al modulo di comunicazione RabbitMQ l'invio delle posizioni aggiornate degli utenti.

- La classe User: è il modello di un singolo individuo all'interno dell'ambiente simulato. Ogni istanza di User incapsula tutte le informazioni rilevanti relative al suo stato e alla sua posizione. Le proprietà chiave che descrivono un utente sono le seguenti:

  1. user_id: identificatore univoco che distingue ogni utente simulato.
  2. Posizione spaziale: l'utente mantiene le coordinate (x, y, z) della sua posizione precisa all'interno del current_node (l'ID del nodo o area in cui si trova). La posizione iniziale degli utenti al momento della loro creazione è determinata in modo casuale, ma è influenzata da distribuzioni temporali configurabili definite nel file di configurazione per replicare scenari realistici di occupazione dell'ambiente in base all'ora del giorno.
  3. Stato comportamentale: attributo fondamentale in quanto definisce il comportamento corrente dell'utente, evolvendo in base agli eventi. Può assumere diversi valori. Nello stato normale l'utente si muove liberamente e casualmente tra i nodi della mappa, simulando le attività di routine quotidiana. Nello stato di allerta, dopo aver ricevuto una notifica di emergenza, l'utente assume una priorità alta nel muoversi. La sua velocità e il suo obiettivo di movimento cambiano per riflettere la situazione. Nello stato salvo l'utente ha raggiunto il punto di sicurezza disignato o l'emergenza è stata dichiarata terminata,

cessando così il movimento attivo. Lo stato di allerta presenta anche un sottostato: in_attesa_percorso. In questo caso l'utente è momentaneamente fermo, in attesa di ricevere un percorso di evacuazione specifico da un servizio esterno.

4. Velocità di movimento: l'utente possiede due velocità predefinite: speed_normal per i movimenti di routine e speed_alert per i movimenti accelerati in condizioni di emergenza. La speed effettiva viene impostata dinamicamente in base allo stato corrente dell'utente.

5. Informazioni di evento e percorso: in caso di allerta, l'utente può avere associato un event (es. "incendio") che descrive la natura dell'emergenza, e un evacuation_path, che è una sequenza di arc_id che l'utente è istruito a seguire per raggiungere la salvezza.

6. Gestione di anomalie: vari flag e contatori, come blocked (per indicare se l'utente è momentaneamente bloccato, magari a causa di un ostacolo o in attesa di istruzioni) e stuck_ticks (un contatore che traccia per quanti "tick" consecutivi l'utente è rimasto nello stesso nodo), sono utilizzati per gestire e risolvere comportamenti inattesi o situazioni di blocco nella simulazione, permettendo al simulatore di reagire in modo robusto.

Il simulatore nel suo complesso, attraverso la classe Simulator, opera secondo diversi stati che influenzano il comportamento generale degli utenti e la logica di avanzamento:

- Stato di funzionamento normale: in questo stato, il simulatore genera utenti e li fa muovere secondo la loro logica di esplorazione casuale o basata sulle distribuzioni orarie. L'ambiente è considerato stabile e non ci sono emergenze attive.

- Stato di allerta: questo stato viene attivato dal simulatore quando riceve un messaggio di allerta da RabbitMQ. Una volta in questo stato, il simulatore inizia a gestire le reazioni degli utenti: li pone inizialmente in stato "in_attesa_percorso" e successivamente, non appena riceve percorsi di evacuazione dal servizio di pathfinding, istruisce gli utenti a seguirli, aumentando la loro velocità a speed_alert.

- Stato di Stop: questo stato si verifica quando il simulatore riceve un messaggio di "stop" (ad esempio, indicante la fine dell'emergenza). In questo caso, gli utenti che non hanno ancora completato un percorso di evacuazione passano allo stato "salvo" e smettono di muoversi. Il simulatore stesso, dopo un timeout configurabile (tempo per il reset dopo un'emergenza), può resettarsi allo stato "normale", permettendo agli utenti di riprendere il loro movimento libero se la situazione è completamente risolta.

Le transizioni tra questi stati sono governate dal motore di simulazione, che reagisce dinamicamente ai comandi esterni (allerta, stop) e alle condizioni interne (es. un utente che raggiunge la fine del suo percorso).

In sintesi, la gestione dei dati e dello stato nello *User Simulator* è un meccanismo reattivo e configurabile che consente agli utenti di modificare il proprio comportamento in base a eventi esterni e di contribuire al monitoraggio della situazione trasmettendo le proprie posizioni. L'architettura basata su configurazione esterna e la lettura della mappa da un database dedicato consentono una notevole flessibilità nella definizione degli scenari di simulazione, mantenendo al contempo una chiara separazione delle responsabilità tra i diversi microservizi del sistema.

5.3.7 Modulo di logging e servizio

Il modulo di logging e servizio raccoglie una serie di funzioni di supporto che, pur non essendo parte della logica core del simulatore, sono indispensabili per il corretto funzionamento, la manutenibilità e l'interazione del microservizio con il mondo esterno. Questo modulo è scomponibile in due parti:

- API HTTP/S: si occupa della registrazione degli endpoint API HTTP/S dell'applicazione, sfruttando il framework FastAPI. La sua funzione principale è esporre un endpoint specifico, ad esempio /positions, che consente a servizi esterni di interrogare e recuperare le posizioni attuali di tutti gli utenti simulati. Questo è cruciale per l'integrazione del simulatore con applicazioni di monitoraggio, dashboard o sistemi di visualizzazione, fornendo un flusso di dati dinamico e consultabile on-demand senza richiedere una conoscenza profonda del funzionamento interno del simulatore.

- logger: si occupa della configurazione e gestione di un sistema di logging centralizzato per l'intero microservizio. Questo assicura che tutti i messaggi generati dal sistema (come debug, informazioni operative, avvertimenti e errori critici) siano registrati in modo consistente, sia su console che su file dedicati. Il logging è fondamentale per il monitoraggio delle operazioni in tempo reale, la diagnostica di eventuali problemi (il debugging) e il tracciamento del comportamento del simulatore nel corso del tempo, sia durante le fasi di sviluppo che in un ambiente di produzione. Una buona configurazione del logger permette di avere visibilità su cosa stia accadendo all'interno del simulatore, facilitando la manutenzione e la risoluzione dei problemi.

5.4 Conclusioni sul simulatore delle posizioni

Il microservizio *User Simulator* si è dimostrato un componente indispensabile nell'architettura complessiva del sistema di gestione delle emergenze, fungendo da ponte cruciale tra la fase di prototipazione e un futuro scenario con dati di posizione reali. Sebbene la sua natura sia transitoria, la sua implementazione robusta e flessibile ha permesso di validare le logiche di sistema in assenza di un hardware di tracciamento fisico.

L'architettura modulare, con una chiara separazione delle responsabilità tra i moduli di avvio, configurazione, interazione con il database, comunicazione asincrona, simulazione e utilità, ha garantito un'elevata manutenibilità e scalabilità. La capacità di definire il comportamento degli utenti tramite configurazione esterna, inclusa la simulazione di distribuzioni spaziali basate su fasce orarie, ha permesso di creare scenari di test realistici e controllabili. La gestione dinamica dello stato degli utenti e del simulatore, in risposta a eventi come le allerte e i percorsi di evacuazione, ha validato l'efficacia delle logiche di reazione del sistema.

In conclusione, lo *User Simulator* ha raggiunto pienamente il suo scopo strategico: ha sbloccato lo sviluppo concorrente degli altri microservizi, ha fornito un ambiente di test controllato e ripetibile per scenari complessi e ha permesso di valutare le performance dell'intera pipeline di gestione delle emergenze. La sua concezione ha facilitato l'integrazione con il resto dell'ecosistema, dimostrando come un componente transitorio possa essere fondamentale per la maturazione di un sistema complesso.

Capitolo 6

Microservizio gestore delle posizioni

Nei sistemi di gestione delle emergenze in ambienti complessi, come edifici o contesti industriali, la capacità di reagire in maniera tempestiva ed efficace a eventi imprevisti è di importanza critica. Un requisito fondamentale in questo ambito è l'elaborazione in tempo reale dei dati di posizione. Storicamente, le architetture di monitoraggio si sono affidate a sistemi centralizzati in cui un server principale raccoglie dati da sensori distribuiti attraverso un modello di polling, un meccanismo in cui il server interroga regolarmente ogni sensore per verificare la disponibilità di nuovi dati. Tuttavia, tale approccio mostra limiti intrinseci di scalabilità: l'aggiunta di ogni nuovo sensore incrementa linearmente il carico sul server, portando a una latenza che può risultare inaccettabile in scenari ad alta densità di dati, come in un'evacuazione di massa. Per affrontare queste problematiche, il *Position Manager* è stato concepito come un broker intelligente. Abbandonando il paradigma del pull (in cui il server interroga attivamente i client per ricevere informazioni) in favore di un modello event-driven (push), i dati di posizione sono inviati in modo asincrono non appena disponibili. Questo design garantisce che il microservizio non resti in attesa passiva, ma si attivi solo in risposta a un evento (l'arrivo di un nuovo dato), ottimizzando l'uso delle risorse e la reattività complessiva del sistema.

6.1 Introduzione e funzionalità specifiche

Il microservizio *Position Manager* rappresenta il fulcro operativo del sistema, agendo da ponte tra i dati di posizione grezzi generati dallo *User Simulator* e le informazioni elaborate, essenziali per gli altri componenti dell'architettura. La sua funzione primaria consiste nel ricevere, analizzare e distribuire i dati di posizione degli utenti in tempo reale, distinguendo lo stato di sicurezza di ogni individuo e comunicando i dati aggregati rilevanti agli altri microservizi. Le sue responsabilità principali si possono riassumere in:

- Gestione delle posizioni: riceve e analizza le posizioni simulate. Mantiene un database con una duplice logica: una tabella per le posizioni correnti (current_position) e una tabella per lo storico degli spostamenti (user_historical_position).

- Analisi del rischio: valuta lo stato di pericolo di ciascun utente in base alla posizione e alle proprietà specifiche del nodo corrispondente all'interno della mappa.

- Aggregazione dei dati: aggrega i dati di pericolo per nodi e utenti, fornendo una visione d'insieme delle aree a rischio.

- Comunicazione: si occupa della comunicazione asincrona mirata, inviando i dati elaborati a microservizi specifici come il *Map Manager* e il *Notification Center* attraverso code RabbitMQ dedicate.

La progettazione di questo microservizio è stata guidata dalla necessità di una logica di elaborazione reattiva ed efficiente, capace di gestire un flusso continuo di dati di posizione e di reagire prontamente a ogni potenziale situazione di pericolo.

6.2 Analisi dello stato dell'arte e motivazione delle scelte implementative

Lo sviluppo del microservizio *Position Manager* è stato guidato dall'obiettivo di creare un sistema che superasse le limitazioni dei tradizionali approcci di monitoraggio, pur riconoscendo i compromessi necessari nella progettazione di una soluzione prototipale. L'analisi dello stato dell'arte nei sistemi di monitoraggio in tempo reale (RTLS[1])[38], e la successiva valutazione delle architetture per la loro gestione, ha rappresentato il fondamento teorico per le scelte progettuali del microservizio *Position Manager*. L'obiettivo primario era superare i limiti dei sistemi convenzionali, pur mantenendo un equilibrio tra complessità e funzionalità. Per sviluppare una soluzione efficace, è stato necessario valutare criticamente le architetture esistenti e i loro limiti, specialmente in un contesto di gestione delle emergenze.

6.2.1 Architetture tradizionali e limiti dei sistemi RTLS

I sistemi RTLS sono ampiamente utilizzati in vari settori, dalla logistica all'automazione industriale, sfruttando diverse tecnologie per il tracciamento di persone e risorse in

[1]RTLS - Real Time Location System

ambienti chiusi. La loro implementazione si basa su tecnologie come Bluetooth Low Energy (BLE), Wi-Fi[2] e RFID[3], ciascuna con le sue specificità. I sistemi BLE, ad esempio, usano beacon a basso consumo che emettono segnali radio, i cui dati vengono triangolati per stimare la posizione. Allo stesso modo, i sistemi Wi-Fi e RFID funzionano raccogliendo dati da punti di accesso o lettori distribuiti.

Nonostante la loro efficacia nel fornire dati di posizione, questi sistemi sono tipicamente progettati con un'architettura monolitica. Tutta la logica, dalla ricezione dei dati grezzi all'analisi e alla presentazione, risiede in un unico blocco software. Questo design porta a una serie di problemi critici, soprattutto in un contesto come la gestione di un'evacuazione:

- Scalabilità ridotta: ogni nuovo sensore o dispositivo utente aumenta linearmente il carico sull'unico server centralizzato, creando un collo di bottiglia che compromette le prestazioni all'aumentare degli utenti.

- Vulnerabilità: l'intera applicazione dipende dalla funzionalità di un singolo componente. Un guasto in una parte del sistema può causare il malfunzionamento dell'intero processo.

- Latenza elevata: l'approccio basato sul polling, dove il server interroga regolarmente ogni sensore, introduce un ritardo tra la raccolta del dato e la sua elaborazione, un'inefficienza inaccettabile durante un'emergenza.

Inoltre, un'ulteriore criticità di queste soluzioni è l'assenza di una logica di analisi del pericolo integrata. Il loro scopo è puramente di monitoraggio, fornendo una fotografia della posizione degli utenti, ma senza la capacità di valutare autonomamente il rischio e attivare risposte immediate. Per un'efficace gestione delle emergenze, è invece fondamentale che l'analisi del pericolo e la notifica siano strettamente integrate nel processo di elaborazione dei dati di posizione.

6.2.2 Modelli moderni di stream processing e la loro non applicabilità al prototipo

Le architetture per il monitoraggio su larga scala si sono evolute verso soluzioni di stream processing, progettate per gestire flussi di dati massicci con latenze minime.[39] Due delle piattaforme più influenti in questo ambito sono Apache Kafka[40] e Apache Flink[41].

[2]Wi-Fi - Wireless Fidelity
[3]RFID - Radio Frequency IDentification

- Apache Kafka: Kafka è una piattaforma di event streaming distribuita. Funziona come un message broker dove i dati vengono organizzati in topic e gestiti da un cluster di broker. I dati vengono resi persistenti su disco garantendo che gli eventi non vadano persi. Gli utenti, chiamati consumer, possono leggere i dati da questi topic in modo asincrono. Presenta una elevata scalabilità orizzontale, un'elevata resilienza e tolleranza ai guasti per cui risulta capace di gestire milioni di messaggi al secondo. Tale scalabilità tuttavia si traduce in una complessità infrastutturale molto alta in quanto richiede la gestione di un cluster, l'installazione di Zookeeper e una configurazione complessa che introduce un notevole overhead operativo.

- Apache Flink: Flink è un framework per l'elaborazione di flussi di dati in tempo reale. A differenza di Kafka, che si concentra sul trasporto dei dati, Flink si specializza nell'esecuzione di calcoli complessi, come aggregazioni e funzioni di stato, sui dati in streaming. Funziona come un motore di calcolo che riceve dati in tempo reale e produce risultati continui. Presenta una latenza estremamente bassa, una gestione dello stato dei flussi di dati e la capacità di eseguire analisi sofisticate. Tali caratteristiche comportano competenze specialistiche per essere implementate e gestite. Non è una soluzione stand-alone ma un framework che necessita di integrazione e un'infrastruttura di supporto.

L'adozione di un'architettura basata su Kafka o Flink per il presente progetto, pur offrendo una scalabilità superiore, sarebbe risultata sovradimensionata e controproducente. L'obiettivo del *Position Manager* era dimostrare la validità di un'architettura a microservizi e di un modello event-driven per la gestione di un flusso di dati controllato e simulato. L'implementazione di una piattaforma di Big Data avrebbe spostato il focus dal problema principale, introducendo una complessità e un overhead che non erano necessari per un prototipo funzionale. Il progetto si è quindi concentrato sulla creazione di un'architettura che bilanciasse in modo ottimale funzionalità, efficienza e semplicità.

6.3 Sviluppo operativo del microservizio

6.3.1 Componenti principali

Lo sviluppo del *Position Manager* ha seguito un approccio modulare e orientato alla produzione, con una chiara separazione dei compiti tra i vari componenti. La sua struttura è schematizzata nella Figura 6.1 , offrendo una visione d'insieme dell'organizzazione del microservizio.

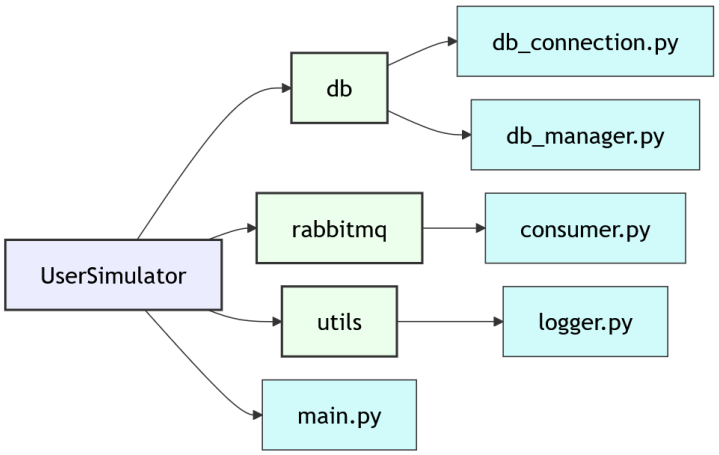

Figura 6.1: Struttura interna del microservizio *Position Manager*

Il microservizio è composto da tre moduli principali:

- Consumer: il cuore operativo del microservizio. Gestisce la connessione con RabbitMQ e contiene la logica principale per l'elaborazione dei messaggi.
- Database Manager: un'astrazione per le interazioni con il database. Questa classe incapsula tutte le query SQL, rendendo il codice del consumer indipendente dalla specifica implementazione del database.
- Logger: un modulo dedicato alla gestione del logging, essenziale per il debug e il monitoraggio del servizio in ambiente di produzione.

Infine il modulo main funge da punto di ingresso del servizio, inizializzando il consumer e avviando il processo di ascolto.

6.3.2 Flusso di lavoro delle posizioni

Il flusso di lavoro del *Position Manager* è un ciclo continuo e asincrono in più fasi:

1. Ricezione del messaggio: il microservizio si mette in ascolto sulla coda RabbitMQ position_queue per i messaggi provenienti dallo *User Simulator*. Da tale coda viene prelevato, analizzato e decodificato un messaggio JSON contenente l'identificatore dell'utente, le coordinate x,y,z e l'identificatore del nodo.
2. Analisi della sicurezza: il microservizio determina lo stato di sicurezza dell'utente in base al nodo in cui si trova. Viene interrogata la tabella del database relativa ai nodi dell'edificio in modo da ottenere informazioni sulla pericolosità del nodo stesso (l'attributo safe se impostato a true indica un nodo sicuro, altrimenti un nodo pericoloso). Tale informazione permette al microservizio di identificare un utente in

pericolo e di attribuire alla posizione corrente dell'utente la variabile danger prima che la posizione venga memorizzata nella tabella delle posizioni correnti.

3. Aggiornamento del database: la posizione, arricchita con la variabile booleana danger, viene salvata sulle due tabelle del database attraverso due operazioni distinte:

   - aggiornamento della posizione corrente dell'utente nella tabella current_position. L'operazione garantisce che per ogni utente esista un solo record aggiornato, ottimizzando l'accesso ai dati in tempo reale.
   - inserimento di un nuovo record nella tabella user_historical_position creando una cronologia degli spostamenti degli utenti. Queste informazioni sono utili per analisi a posteriori, come la ricostruzione del percorso di un utente durante un'emergenza.

4. Logica di invio dati: l'invio dei dati agli altri microservizi non avviene per ogni singola posizione, ma è regolato da una strategia ibrida che garantisce reattività ed efficienza:

   - Soglia di messaggi: dopo l'elaborazione di un numero di messaggi definito, il microservizio invia i dati aggregati agli altri microservizi.
   - Intervallo temporale: un thread separato assicura che, anche con un basso flusso di dati, un invio avvenga dopo che sia passato un intervallo definito.

5. Invio messaggi: i dati aggregati vengono inviati su due code distinte, map_manger_queue e alerted_users_queue, a seconda del destinatario e del formato del messaggio richiesto.

6. Gestione della fine emergenza: il sistema monitora costantemente lo stato di tutti gli utenti attraverso le posizioni correnti. Se tutti gli utenti sono in stato salvo, il microservizio invia un messaggio di tipo Stop al *Notification Center* per segnalare la fine dell'emergenza.

Il flusso appena descritto è rappresentato graficamente in Figura 6.2.

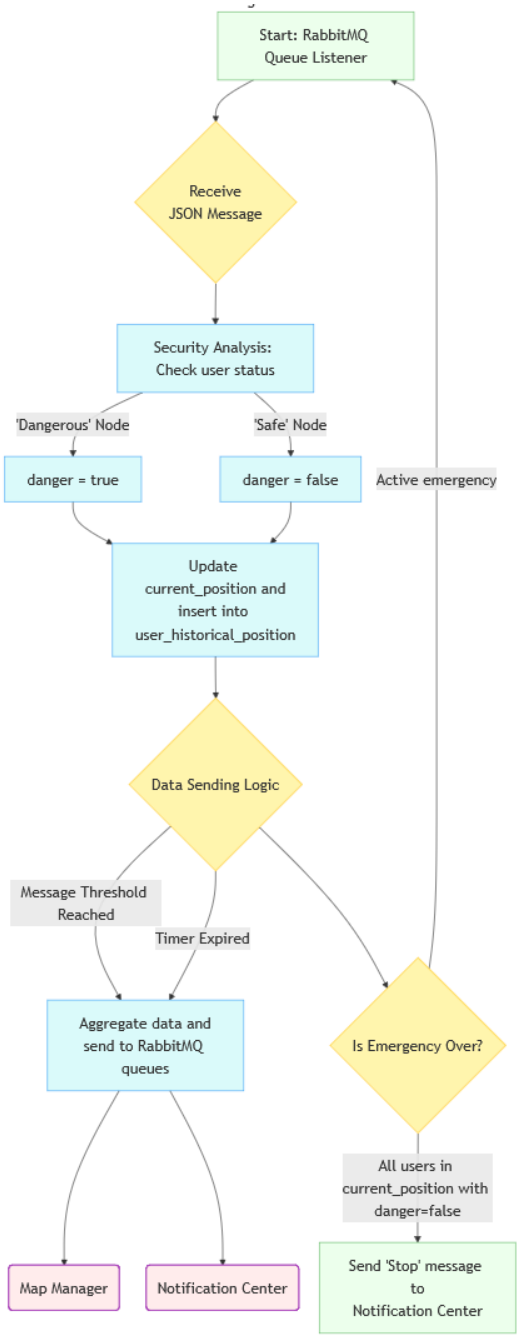

Figura 6.2: Flusso interno del microservizio *Position Manager*

6.3.3 Modulo di interazione con il database

Il modulo db_manager.py incapsula l'intera logica di interazione con il database, fungendo da strato di astrazione tra il microservizio e lo storage dei dati. Questo approccio garantisce che la logica di business rimanga indipendente dalla specifica implementazione del database. Le sue funzionalità chiave sono:

- Gestione delle posizioni: le funzioni upsert_current_position e insert_historical_position sono responsabili della persistenza dei dati. L'uso di un'operazione di UPSERT

(una combinazione di UPDATE e INSERT) sulla tabella current_position è una scelta critica di design. Questo evita la creazione di record duplicati e garantisce che la tabella rifletta sempre l'ultima posizione nota di ogni utente, ottimizzando le query per l'accesso ai dati in tempo reale. Parallelamente, ogni aggiornamento o inserimento su current_position scatta un trigger database che incrementa o decrementa automaticamente il contatore current_occupancy nella tabella nodes, mantenendo l'occupazione dei nodi sincronizzata senza richiedere calcoli aggiuntivi dal microservizio. L'utilizzo delle query SQL è rappresentato in Figura 6.3.

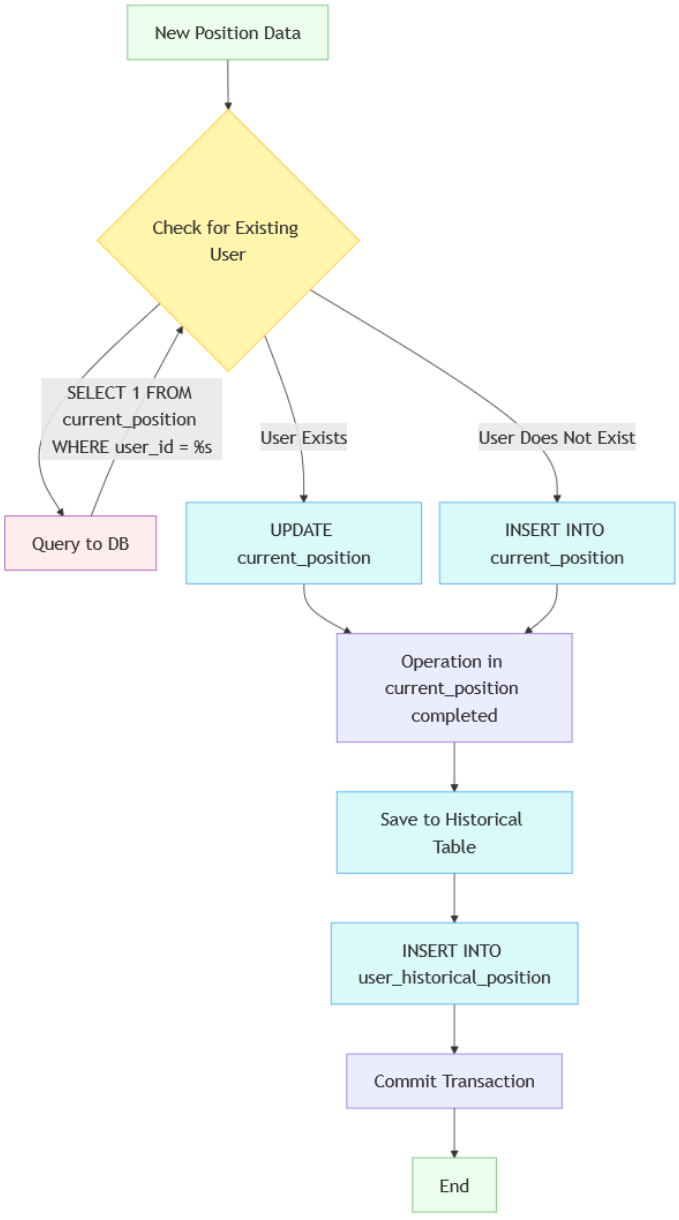

Figura 6.3: Flusso interno delle query SQL per la gestione delle posizioni

- Analisi e aggregazione: per supportare le funzionalità di analisi, il modulo fornisce query specializzate. La funzione get_dangerous_node_aggregates sfrutta le capacità

di aggregazione di PostgreSQL con la clausola GROUP BY per raggruppare gli utenti a rischio per nodo. La funzione get_aggregated_evacuation_data estende questa logica, eseguendo un'operazione di JOIN tra le tabelle current_position e nodes per recuperare, per ogni nodo in pericolo, la lista degli utenti a rischio e il corrispondente percorso di evacuazione. Il flusso interno delle query è riportato in Figura 6.4.

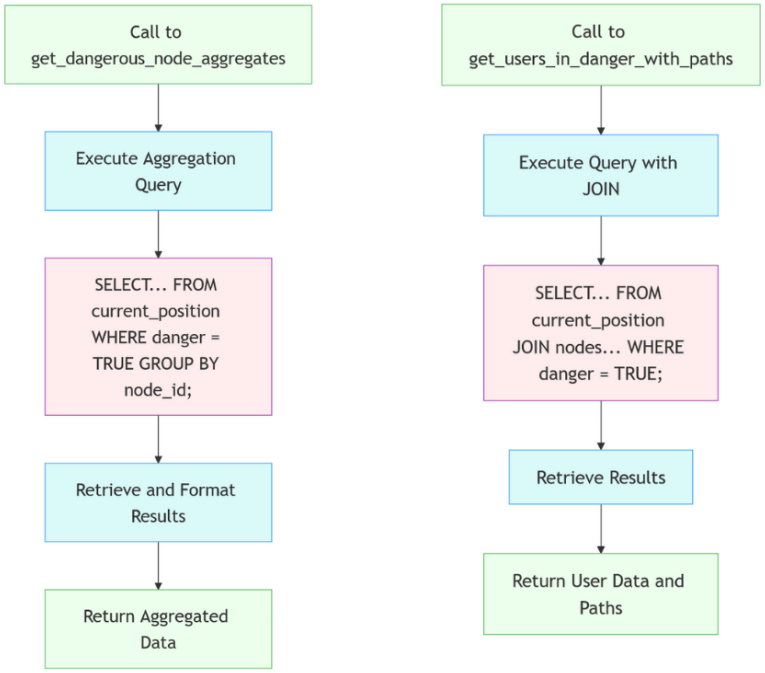

Figura 6.4: Flusso interno delle query SQL per l'analisi e l'aggregazione dei dati

- Ottimizzazione delle query: il metodo is_node_safe implementa una cache in memoria con un meccanismo di Time-To-Live (TTL) di 5 secondi. Questa strategia riduce drasticamente il numero di interrogazioni al database per lo stesso nodo, un'ottimizzazione fondamentale per un'applicazione che gestisce un flusso di dati ad alta frequenza e che deve minimizzare la latenza. Il diagramma in Figura 6.5 rappresenta la logica di caching utilizzata per ottimizzare il funzionamento del *Position Manager*.

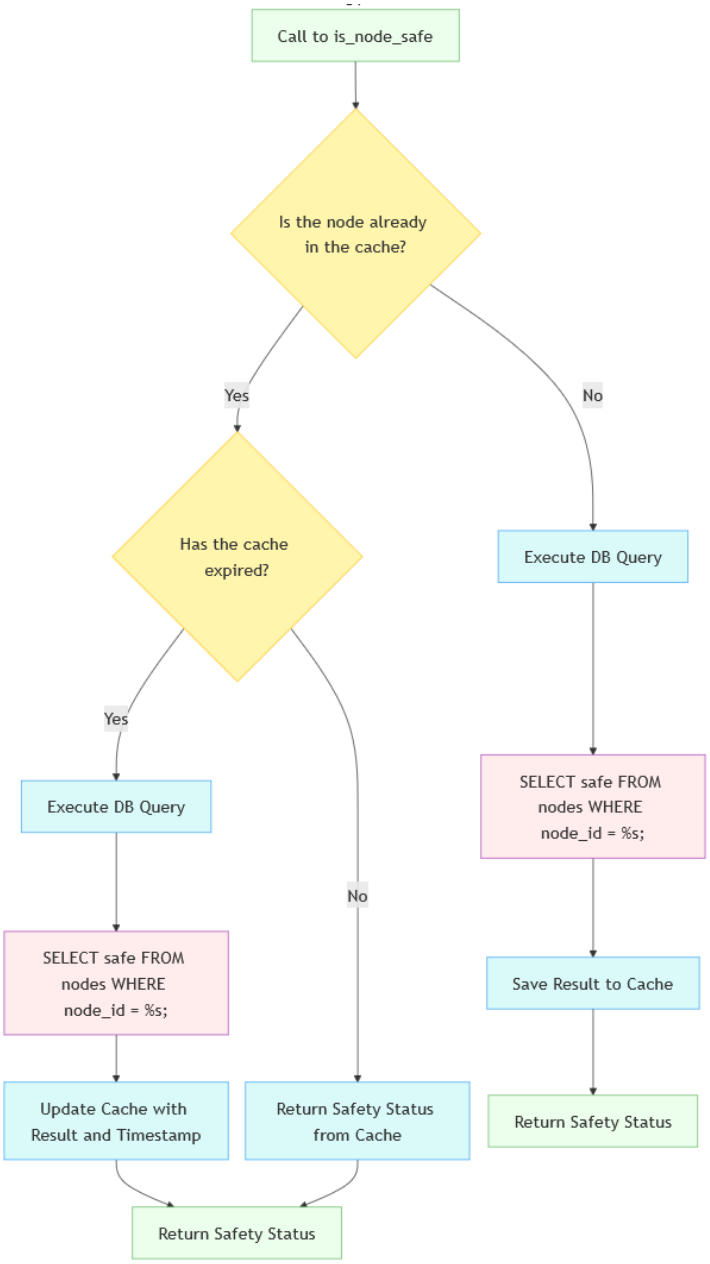

Figura 6.5: Rappresentazione del meccanismo TTL per l'ottimizzazione delle query

6.3.4 Modulo di comunicazione asincrona

Il modulo consumer.py rappresenta il cuore reattivo del microservizio. È responsabile dell'ascolto continuo sulla coda di input e della gestione efficiente dei messaggi in arrivo. La sua architettura interna è progettata per bilanciare reattività e stabilità, utilizzando un modello di concorrenza basato su thread.

- Concorrenza: oltre al thread principale che consuma i messaggi dalla position_queue, un thread separato viene utilizzato per garantire che i messaggi aggregati vengano inviati a intervalli regolari. Questa logica previene che i dati rimangano bloccati nel

microservizio in situazioni di traffico ridotto, assicurando che gli altri componenti del sistema ricevano aggiornamenti tempestivi.

- Gestione delle code: il consumer gestisce diverse connessioni e canali RabbitMQ per inviare messaggi a destinatari specifici. Vengono utilizzate due code di output principali: map_manager_queue, per inviare dati aggregati al microservizio di visualizzazione, e alerted_users_queue, per inviare dati più dettagliati, inclusi i percorsi di evacuazione, al microservizio di notifica.

- Logica di invio: la strategia di invio dei messaggi è ibrida. I dati aggregati vengono inviati sia al raggiungimento di una soglia di messaggi elaborati, sia quando un intervallo di tempo scade. Questo approccio garantisce che il sistema risponda in modo rapido agli eventi improvvisi e frequenti, mantenendo al contempo un flusso costante di aggiornamenti.

6.3.5 Modulo di logging

Il modulo di logging (logger.py) è un componente cruciale per la manutenibilità e il debug del microservizio. Nonostante la sua apparente semplicità, è stato configurato per fornire output su due destinazioni distinte, ciascuna con un livello di dettaglio differente.

- Console: l'handler della console è configurato per mostrare messaggi con un livello di gravità INFO o superiore. Questo fornisce una visione chiara e sintetica dell'attività del microservizio in tempo reale, utile per il monitoraggio operativo.

- File: l'handler del file di log cattura tutti i messaggi con un livello di gravità DEBUG o superiore. Questo livello di dettaglio è fondamentale per l'analisi post-mortem e il debug, consentendo di tracciare ogni singola operazione, inclusi i valori delle variabili, i risultati delle query e lo stato interno del consumer.

L'uso di un logger strutturato e configurabile è essenziale in un'architettura a microservizi, dove il debug su sistemi distribuiti può essere particolarmente complesso.

6.3.6 Conclusioni sul gestore delle posizioni

Il microservizio *Position Manager* ha dimostrato con successo la validità di un'architettura event-driven e a microservizi per la gestione di dati di posizione in tempo reale. Sebbene la sua implementazione non sia ottimizzata per una scalabilità a livello di Big Data, le scelte di design adottate, come l'uso di Python, PostgreSQL e RabbitMQ, hanno

fornito una piattaforma robusta e affidabile per la validazione del sistema. Le funzionalità di gestione dei dati, di analisi e di comunicazione sono state implementate con un'attenzione alla performance e alla resilienza.

Questo microservizio rappresenta un prototipo funzionale che convalida i principi fondamentali del progetto: la separazione delle responsabilità, la resilienza del sistema attraverso la messaggistica asincrona e l'efficienza nell'elaborazione dei dati. Le lezioni apprese e i compromessi di design evidenziati in questo capitolo saranno fondamentali per un'eventuale futura evoluzione del sistema, orientata a gestire volumi di dati di ordini di grandezza superiori, magari con l'integrazione di tecnologie di stream processing specializzate.

Capitolo 7

Risultati sperimentali

Questa sezione dell'elaborato è dedicata alla presentazione e all'analisi dei risultati ottenuti dalle simulazioni condotte per validare l'architettura a microservizi proposta. L'obiettivo principale è duplice: dimostrare il funzionamento del sistema nella gestione di scenari di emergenza e misurarne le performance opertative in termini di latenza e efficacia. I dati raccolti offrono un'analisi quantitativa che supporta le motivazioni progettuali esposte nei capitoli precedenti, confermando come la modularità e il disaccoppiamento dei microservizi contribuiscano a un'elevata reattività e resilienza del sistema. Per garantire una valutazione completa e robusta, l'approccio adottato è stato a due livelli: una valutazione qualitativa basata su simulazioni complete di singoli scenari, e una valutazione quantitativa su larga scala, variando il numero di utenti.

7.1 Analisi del caso di studio: Campus universitario di Cesena

Le simulazioni sono state eseguite per valutare la capacità del sistema di gestione delle emergenze di reagire in modo tempestivo e coerente a eventi critici in un ambiente complesso. Il caso di studio scelto è il Campus universitario di Cesena, un ambiente che, per le sue dimensioni e la sua struttura, rappresenta uno scenario ideale per testare l'architettura distribuita.

L'edificio è stato modellato come un grafo orientato, dove i nodi rappresentano le aule, i corridoi e le uscite, e gli archi denotano i collegamenti tra di essi. Questa astrazione topologica consente al sistema di operare su una rappresentazione precisa dell'ambiente, essenziale per il calcolo dei percorsi di evacuazione. La configurazione specifica dell'edificio è stata definita nel *Map Viewer* come segue: l'edificio si sviluppa su tre piani, ciascuno con funzionalità distinte. Il terzo piano è adibito a uffici e spazi di lavoro, mentre il primo e il

secondo piano ospitano principalmente aule didattiche, laboratori, zone ristoro e servizi igienici. La configurazione dell'edificio è rappresentata dalle seguenti Figure 7.1, 7.2, 7.3.

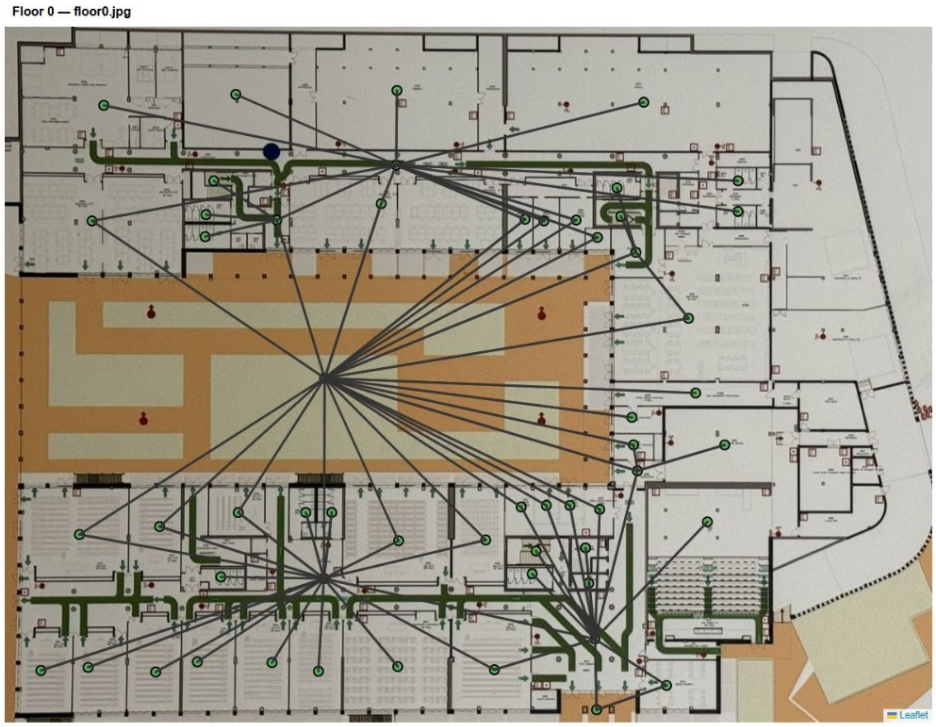

Figura 7.1: Piano 0 del Campus di Cesena

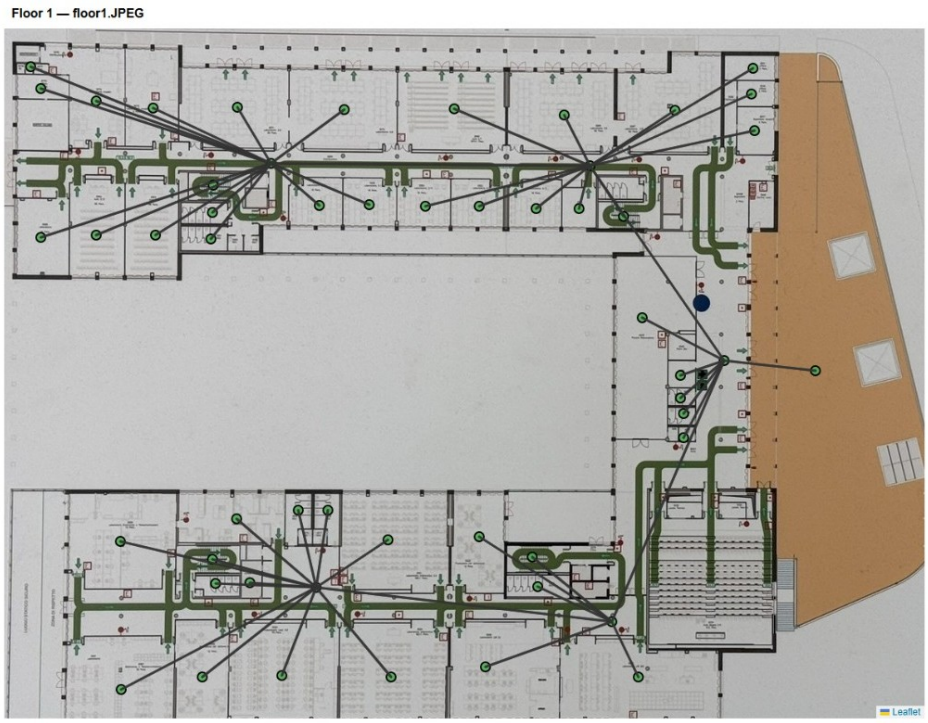

Figura 7.2: Piano 1 del Campus di Cesena

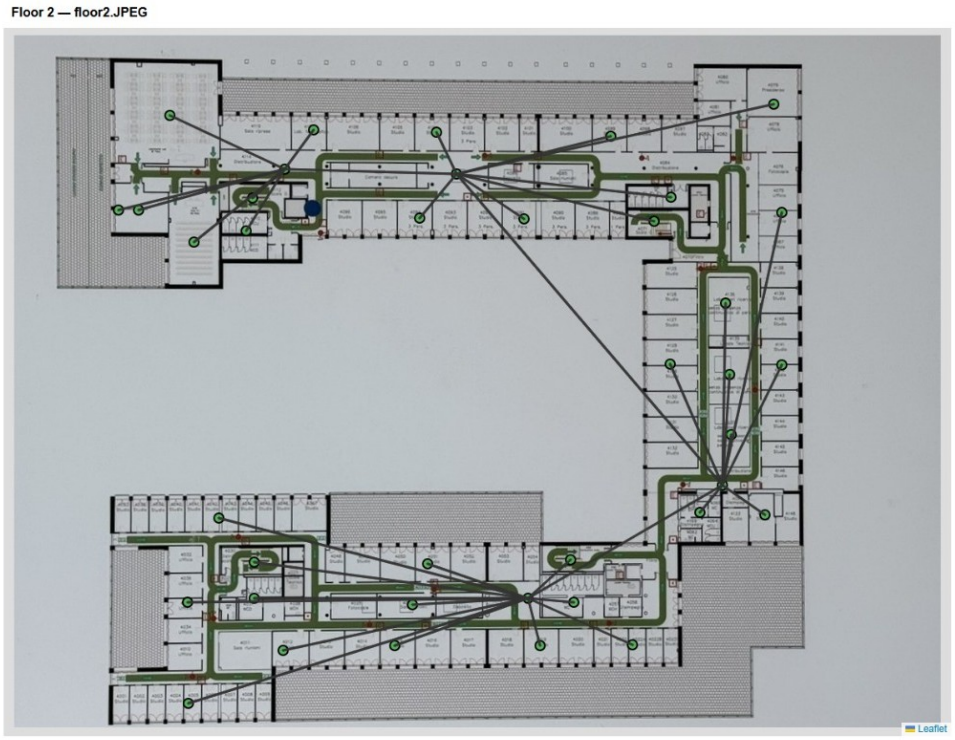

Figura 7.3: Piano 2 del Campus di Cesena

L'edificio è rappresentato attraverso l'insieme di 148 nodi differenti, collegati tra loro mediante 343 archi. La configurazione iniziale della distribuzione degli utenti, che avviene all'avvio del sistema in stato normale, è un aspetto cruciale delle simulazioni. Questo permette di posizionare un numero variabile di utenti in modo strategico, simulando scenari realistici come la concentrazione di persone nelle aule durante le lezioni o negli uffici durante l'orario di lavoro. Una volta che l'*Alert Manager* genera un'allerta, la simulazione del movimento degli utenti è gestita in modo dinamico, basandosi sui percorsi di evacuazione calcolati in tempo reale dal sistema.

Per dimostrare la versatilità del sistema, sono stati simulati due tipi di emergenza. La scelta di questi scenari non è casuale, in quanto rappresentano due macro-categorie di eventi che richiedono logiche di evacuazione differenti, testando la capacità del sistema di adattarsi a vincoli diversi.

- Allagamento (Flood): Questo scenario simula un'emergenza localizzata che rende alcune aree della mappa inaccessibili, costringendo il sistema a ricalcolare i percorsi escludendo dinamicamente le zone a rischio. Nel caso di studio, è stato ipotizzato che l'allagamento renda inaccessibile l'intero Piano 0. Di conseguenza, tutti gli utenti presenti in quel piano, indipendentemente dalla loro posizione, devono essere direzionati verso i nodi che consentono il passaggio ai livelli superiori, ovvero i nodi

di tipo stairs. Il sistema ha il compito di calcolare i percorsi più efficienti per raggiungere le scale, evitando che nessun utente rimanga intrappolato al Piano 0. Questo scenario testa in modo specifico la logica di esclusione di aree e la navigazione tra i piani.

- Terremoto (Earthquake): Questo scenario, che si applica anche ad altre tipologie di allerta come attacco terroristico, incendio e crollo strutturale, richiede una logica di evacuazione basata sulla distribuzione degli utenti verso l'esterno dell'edificio. La risposta del sistema in questo caso deve essere il calcolo di percorsi che portino gli utenti verso le uscite di sicurezza e punti di raccolta esterni. A differenza dell'allagamento, dove l'obiettivo è raggiungere un piano superiore, qui la priorità è lasciare l'edificio il più rapidamente possibile.

Questa distinzione è fondamentale: l'allagamento si concentra sull'inaccessibilità di alcune aree e sulla navigazione tra i piani, mentre il terremoto (e gli altri eventi correlati) richiede una logica di evacuazione completa verso l'esterno, testando la capacità del sistema di gestire un'emergenza su larga scala. Le analisi successive si baseranno su questi scenari per fornire una valutazione completa delle performance e della robustezza del sistema.

7.2 Simulazioni qualitative

Per un'analisi dettagliata del flusso operativo, sono state eseguite due simulazioni complete, ciascuna con 25 utenti, replicando i due scenari di emergenza predefiniti: Allagamento (Flood) e Terremoto (Earthquake). L'obiettivo primario di queste simulazioni non era la misurazione delle performance, bensì la validazione funzionale del sistema. Di seguito verrà mostrato l'intero processo di emergenza eseguito dal sistema complessivo per entrambe le tipologie di allerta, soffermandosi sull'evacuazione completa di 5 utenti. Viene mostrata una visuale limitata in modo da rendere più comprensibile il processo di evacuazione fornito dal sistema.

7.2.1 Allerta di tipo Earthquake

L'analisi qualitativa del primo caso di studio riguarda la tipologia di allerta Earthquake, che ha permesso di verificare, passo dopo passo, la corretta interazione tra i microservizi, la reattività del sistema e l'efficacia del calcolo dei percorsi di evacuazione nel caso in cui è necessaria un'evacuazione totale dell'edificio.

La simulazione ha inizio con il sistema in stato normale. Gli utenti sono distribuiti in diverse aree dell'edificio, in conformità con la configurazione iniziale definita dallo *User*

Simulator. Le Figure7.4, 7.5, 7.6 mostrano le posizioni di partenza degli utenti. Questo stato iniziale serve come punto di riferimento per l'intero processo di evacuazione.

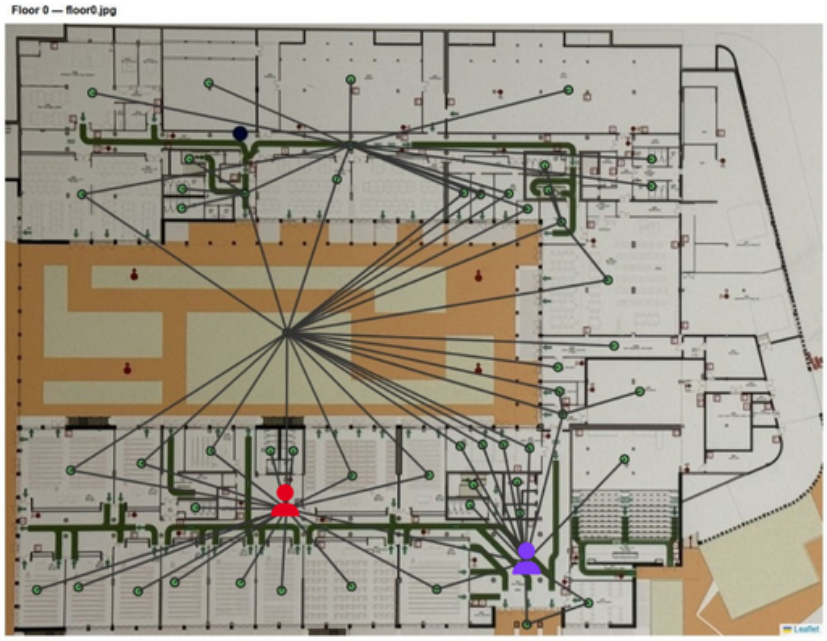

Figura 7.4: Posizioni iniziali Piano 0 prima dell'allerta Terremoto

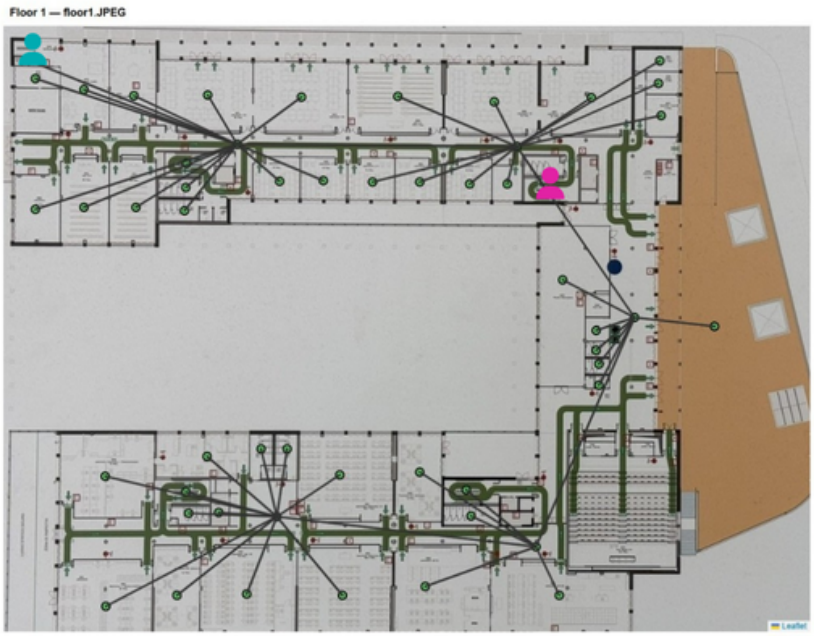

Figura 7.5: Posizioni iniziali Piano 1 prima dell'allerta Terremoto

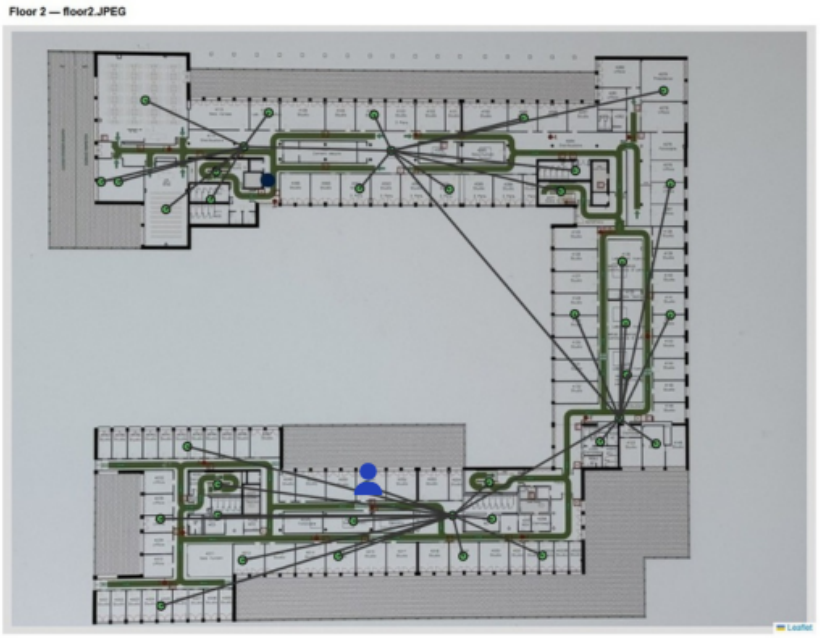

Figura 7.6: Posizioni iniziali Piano 2 prima dell'allerta Terremoto

A seguito della ricezione di un'allerta di tipo Earthquake, il sistema entra in funzione, calcolando per ogni nodo in cui si trova un utente in pericolo il percorso di evacuazione ottimale per evacuare l'utente all'esterno dell'edificio. La Figura 7.7 mostra un esempio dei percorsi generati. In questa immagine, vengono riportati i nodi iniziali degli utenti, il piano in cui si trovano, la tipologia di nodo e la sequenza ordinata di archi che compongono il percorso di evacuazione.

	node_id [PK] integer	floor_level integer[]	node_type character varying (50)	evacuation_path integer[]
1	21	{0}	corridor	{26}
2	22	{0}	corridor	{222}
3	59	{1}	bathroom	{381,375,100,106}
4	75	{1}	stair	{305,100,106}
5	168	{2}	office	{155,149,360,344,377}

Figura 7.7: Nodi iniziali degli utenti e percorsi di evacuazione associati per l'allerta Earthquake

Dopo aver osservato i percorsi, il sistema inizia a simulare il movimento degli utenti seguendo gli archi indicati. Le Figure 7.8, 7.9, 7.10 rappresentano il percorso eseguito dagli utenti, fornendo una prova visiva che il movimento simulato segue fedelmente i percorsi calcolati.

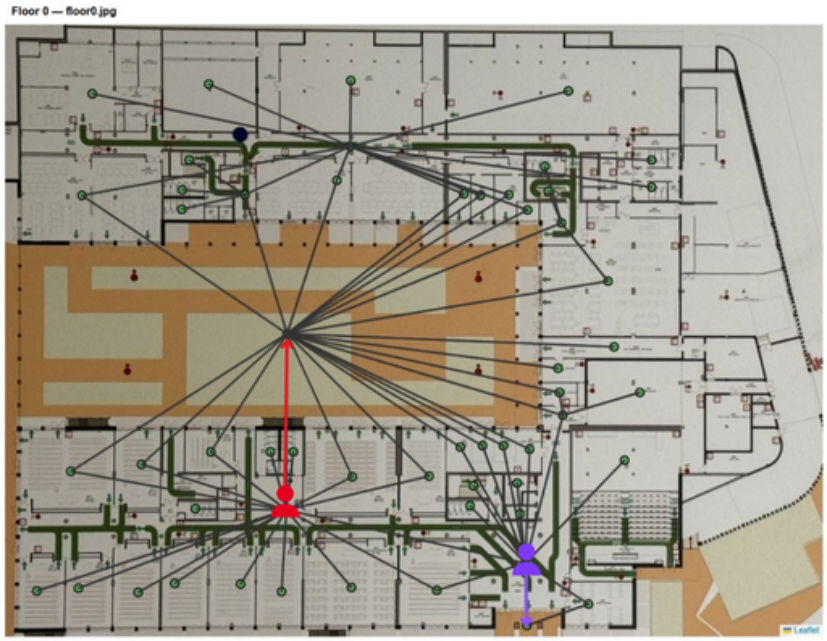

Figura 7.8: Percorso degli utenti del Piano 0

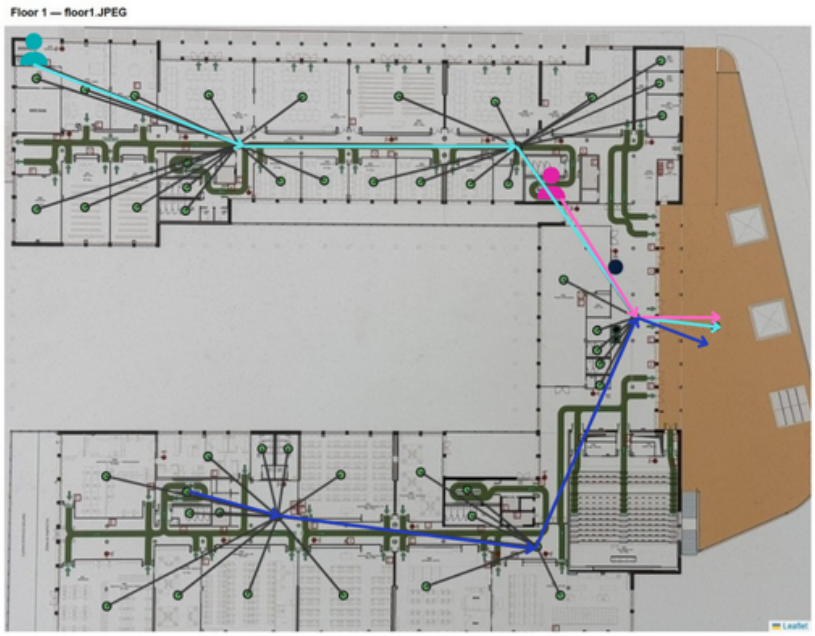

Figura 7.9: Percorso degli utenti del Piano 1

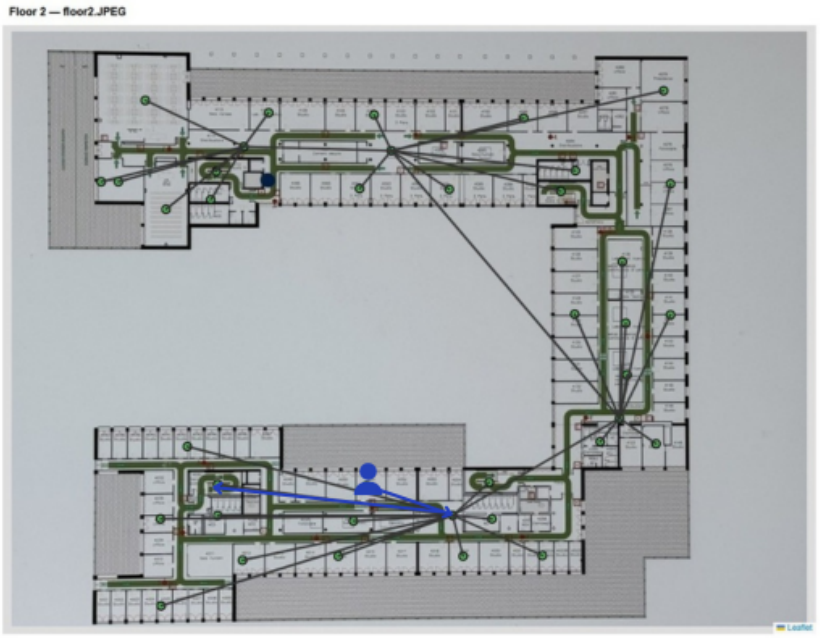

Figura 7.10: Percorso degli utenti del Piano 2

L'analisi qualitativa effettuata, supportata da rappresentazioni visive, valida in modo esaustivo la logica e l'affidabilità del sistema di gestione delle emergenze nel caso in cui è necessaria un'evacuazione totale dell'edificio.

7.2.2 Allerta di tipo Flood

La valutazione qualitativa del secondo caso di studio, l'allerta Flood, ha permesso di verificare la corretta reazione del sistema a un'evacuazione parziale e mirata, basata su una configurazione predefinita. In questo scenario, l'allerta Flood prevede l'evacuazione di tutti gli utenti che si trovano nel Piano 0, considerato inagibile.

La simulazione inizia con il sistema in stato normale e gli utenti distribuiti strategicamente nei tre piani dell'edificio. Come mostrato nelle Figure 7.11, 7.12, e 7.13, il Piano 0 è l'unica area di pericolo, mentre il resto dell'edificio rimane sicuro. Pertanto, il protocollo di evacuazione si attiva solo per gli utenti del Piano 0, senza coinvolgere quelli dei piani superiori.

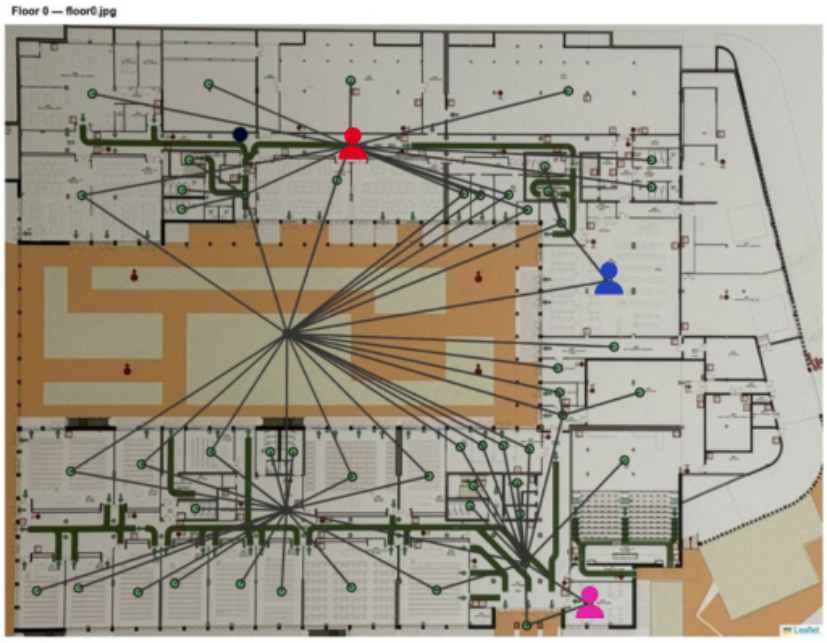

Figura 7.11: Posizioni iniziali Piano 0 prima dell'allerta Alluvione

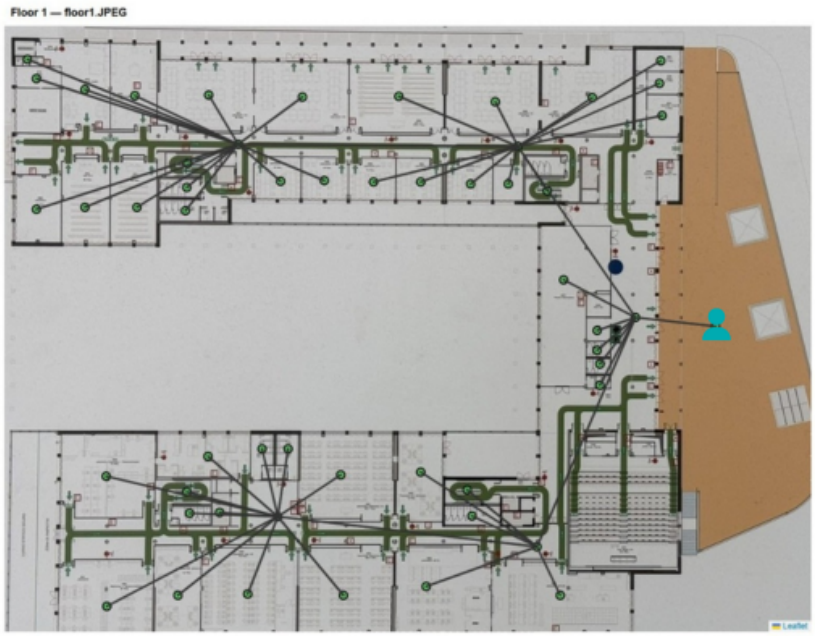

Figura 7.12: Posizioni iniziali Piano 1 prima dell'allerta Alluvione

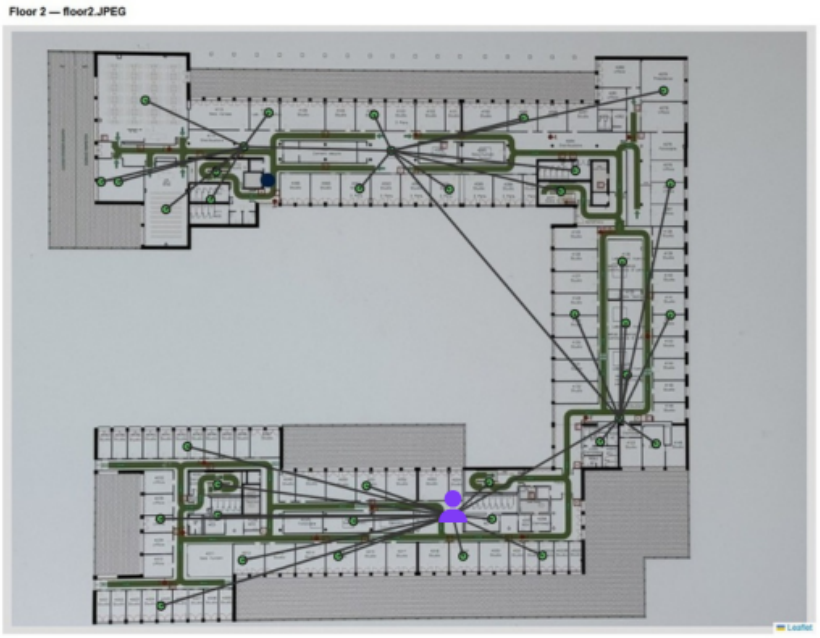

Figura 7.13: Posizioni iniziali Piano 2 prima dell'allerta Alluvione

A seguito dell'allerta Flood, il sistema richiede al *Map Manager* di calcolare i percorsi di evacuazione. La logica applicata è che gli utenti in pericolo devono essere direzionati verso i nodi di tipo stairs che li portino ai piani superiori, evitando completamente l'area inagibile. La Figura 7.14 mostra un esempio dei percorsi calcolati, dove si nota che tutti i percorsi degli utenti del Piano 0 convergono verso il nodo che rappresenta le scale. Questa convergenza è la prova visiva del corretto funzionamento dell'algoritmo in uno scenario di evacuazione parziale.

	node_id [PK] integer	floor_level integer[]	node_type character varying	evacuation_path integer[]
1	24	{0}	classroom	{11,227}
2	53	{0}	canteen	{263,70}
3	57	{0}	corridor	{272,256}

Figura 7.14: Nodi iniziali degli utenti e percorsi di evacuazione associati per l'allerta Flood

Similmente allo scenario Earthquake, lo *User Simulator* inizia a muovere gli utenti in pericolo lungo i percorsi calcolati. Il *Position Manager* registra le nuove posizioni a intervalli regolari. Le Figure 7.15, 7.16, 7.17 mostrano il movimento degli utenti sulle rispettive mappe, rendendo visibile il percorso compiuto dai soli utenti del Piano 0.

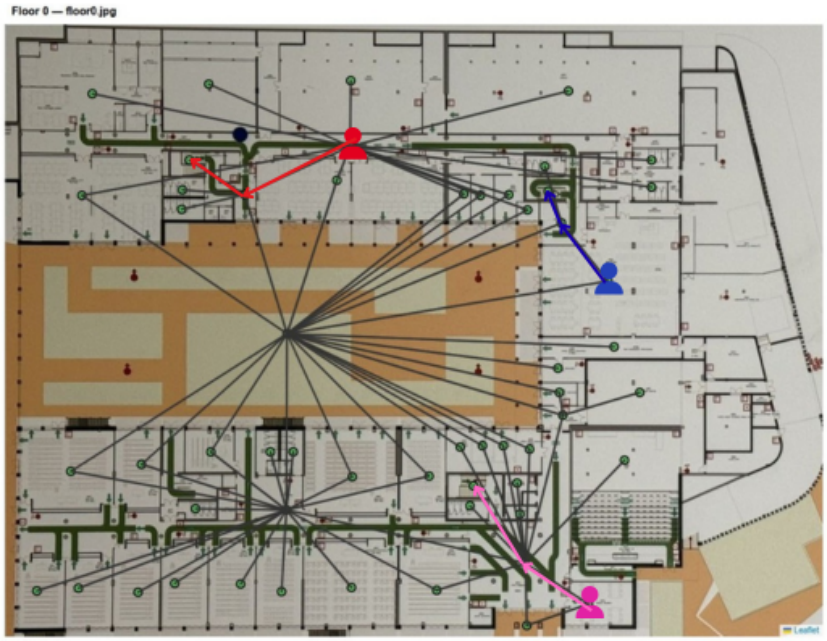

Figura 7.15: Percorso degli utenti nel Piano 0

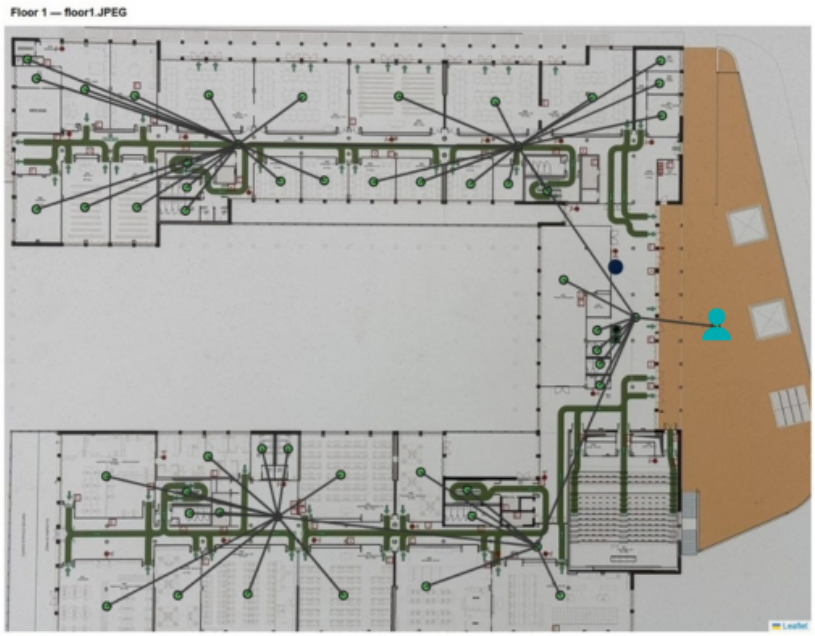

Figura 7.16: Posizioni degli utenti nel Piano 1

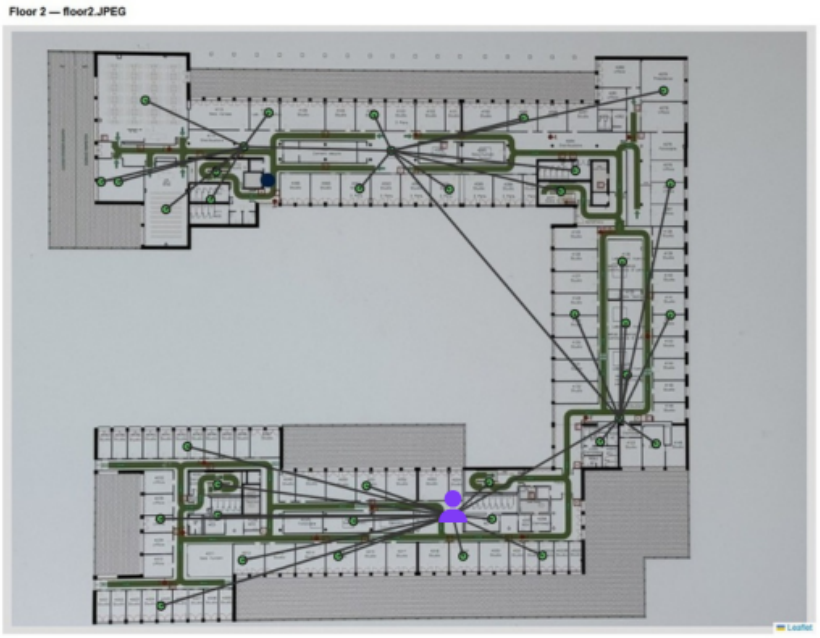

Figura 7.17: Posizioni degli utenti nel Piano 2

L'analisi qualitativa effettuata, supportata dalle rappresentazioni visive, valida in modo esaustivo la logica e l'affidabilità del sistema di gestione delle emergenze nel caso in cui è necessaria un'evacuazione parziale e selettiva dell'edificio.

7.3 Validazione quantitativa

Questa sezione è dedicata alla validazione quantitativa della piattaforma, un processo che si concentra sull'impiego di metodi numerici e misurazioni precise per dimostrare le prestazioni, la scalabilità e la robustezza del sistema al variare delle condizioni operative. A differenza della validazione qualitativa, che si basa su osservazioni e descrizioni, questo approccio si fonda su dati e statistiche misurabili. In particolare, l'analisi si concentra sui microservizi di simulazione del movimento (*User Simulator*) e di gestione dello stato degli utenti (*Position Manager*), componenti cruciali per la capacità del sistema di modellare il comportamento della popolazione e di tracciarne la condizione di sicurezza in tempo reale.

Per condurre l'analisi, è stato adottato un approccio di analisi di sensitività univariata. Questo metodo prevede l'esecuzione di una serie di esperimenti, partendo da uno scenario di base e modificando un singolo fattore sperimentale alla volta, mantenendo gli altri costanti. Questa tecnica ha permesso di isolare e quantificare con precisione l'impatto di ciascuna variabile sugli Indicatori Chiave di Prestazione (KPI). Un KPI è una metrica utilizzata per misurare e valutare il successo di un sistema; nel nostro caso, i KPI traducono obiettivi astratti come la sicurezza in valori numerici e misurabili. Per ogni scenario

di test, sono state eseguite cinque simulazioni distinte. Questo approccio ha permesso di minimizzare l'impatto della variabilità stocastica intrinseca al modello, garantendo la significatività statistica e la riproducibilità dei risultati.
La valutazione delle prestazioni si basa su due KPI primari: efficacia ed efficienza.

- L'efficacia del sistema: quantificata dal numero di utenti salvati, che rappresenta la percentuale di occupanti che completano con successo il percorso designato, misurando la capacità del sistema di garantire la sicurezza della totalità della popolazione coinvolta.

- L'efficienza del sistema: misurata dal tempo totale di evacuazione, definito come l'intervallo temporale che intercorre tra l'istante di emissione dell'allerta e il momento in cui l'ultimo utente raggiunge un punto di raccolta sicuro.

Per ogni scenario, l'evento di allerta definisce $t_0 = 0$; tutti i tempi sono riportati come Δt rispetto a t_0. Le misure elementari raccolte in ogni esecuzione sono:

1. Ricezione primo path Δt_{First}: istante di consegna del primo percorso allo *User Simulator*.

2. Ricezione ultimo path Δt_{Last}: istante di consegna dell'ultimo percorso allo *User Simulator*.

3. Ricezione Stop Δt_{Stop}: istante della notifica di termine emergenza, che attesta che tutti gli utenti simulati a rischio sono in sicurezza, rappresenta il tempo totale di evacuazione.

Per l'interpretazione dei risultati, sono state impiegate metriche statistiche standard, calcolate sui dati aggregati delle cinque esecuzioni per ogni scenario. Questo approccio è stato scelto per superare le limitazioni di una singola misurazione, che potrebbe essere influenzata da fattori casuali. Analizzando un campione di dati, è possibile ottenere una stima più robusta e rappresentativa delle performance. Per ogni set di dati, sono stati calcolati la media (μ) e la deviazione standard (σ). La media è utilizzata come indicatore di tendenza centrale, mentre la deviazione standard quantifica la dispersione dei dati rispetto alla media. Un valore basso di σ indica una maggiore stabilità e affidabilità del sistema.

Le formule utilizzate per il calcolo sono: La media campionaria definita come:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{7.1}$$

La deviazione standard campionaria definita come:

$$s = \sqrt{\frac{1}{,n-1,}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2} \tag{7.2}$$

dove n rappresenta il numero di osservazioni (in questo caso $n = 5$), x_i è la i-esima osservazione e $\bar{x}$ è la media. I risultati sono presentati nella notazione compatta $\mu \pm \sigma$.

Le prestazioni sono state inoltre caratterizzate da due metriche chiave: il throughput e il latency gap. Il throughput è definito come il numero di utenti evacuati con successo per unità di tempo e misura l'efficienza del flusso di persone. Concettualmente, è calcolato come il rapporto tra il numero totale di utenti salvati e il tempo totale di evacuazione. La sua formula è:

$$\text{Throughput} = \frac{\text{Utenti salvati}}{\text{Tempo totale di evacuazione}} \tag{7.3}$$

Il latency gap, che quantifica l'intervallo temporale in cui il sistema è in una fase attiva, è definito come la differenza tra il momento in cui viene inviato l'ultimo percorso di evacuazione e il momento in cui viene inviato il primo. La sua formula è:

$$\text{Latency Gap} = T_{\text{Ultimo percorso inviato}} - T_{\text{Primo percorso inviato}} \tag{7.4}$$

Entrambe le misurazioni sono calcolate come intervalli di tempo, con l'istante di ricezione allerta considerato come il tempo zero del processo.

L'analisi è stata strutturata modulando quattro fattori sperimentali che riflettono le diverse dimensioni del problema, con particolare attenzione a come influenzano il movimento degli utenti e la gestione dei loro dati:

- Carico del sistema: sono stati definiti cinque livelli di carico (100, 300, 500, 750 e 1000 utenti) per testare la scalabilità dei microservizi di simulazione e di gestione del database sotto stress crescente.

- Contesto temporale e spaziale: sono state modellate quattro fasce orarie rappresentative per simulare profili di occupazione eterogenei e valutare come la distribuzione iniziale degli utenti influenzi le metriche di evacuazione e la capacità del sistema di identificare gli utenti in pericolo.

- Tipologia di allerta: sono state investigate due tipologie di emergenza, Terremoto e Alluvione, per verificare come il sistema si adatta a strategie di evacuazione che variano da totali a parziali.

- Capacità del grafo topologico: sono state confrontate due configurazioni, a capacità infinita e a capacità reale, per valutare come i fenomeni di congestione influiscano sulla fluidità del movimento degli utenti gestito dal modulo di simulazione.

I risultati di queste analisi, presentati nei sottoparagrafi successivi, permetteranno di ottenere una comprensione dettagliata e oggettiva delle performance della piattaforma, fornendo una base solida per le conclusioni generali del lavoro.

7.3.1 Impatto del carico del sistema

In questa sezione viene analizzato l'impatto del carico del sistema sulle metriche di performance della piattaforma. L'esperimento è stato condotto variando il numero di utenti simulati (da 100 a 1000) e mantenendo costanti le altre condizioni: uno scenario di allerta Terremoto con capacità di archi e nodi limitata, simulato alle ore 10. Questa analisi è particolarmente rilevante per valutare la scalabilità dei microservizi di simulazione del movimento e di gestione dello stato degli utenti, responsabili di processare e tracciare un numero crescente di agenti.

La Tabella 7.1 e la Figura 7.18 presentano i tempi medi e la deviazione standard per i principali eventi di notifica. I tempi di ricezione del primo percorso mostrano una notevole stabilità fino a 750 utenti, attestandosi tra i 16 e i 25 secondi. Questo risultato evidenzia la tempestiva capacità del sistema di avviare il processo di evacuazione anche con un numero elevato di richieste simultanee. Solo al carico massimo di 1000 utenti si osserva un aumento del tempo medio (38.6 s) accompagnato da una maggiore variabilità ($\sigma = 45.2$), indicando che in condizioni di carico estremo, il microservizio di simulazione gestisce le richieste in modo meno uniforme.

Terremoto con capacità archi e nodi limitata simulata alle ore 10			
Utenti simulati	**Ricezione primo path (Δt)**	**Ricezione ultimo path (Δt)**	**Ricezione Stop (Δt)**
100	25.0 ± 7.4	33.2 ± 7.1	37.8 ± 6.7
300	18.0 ± 24.8	33.8 ± 17.6	35.2 ± 17.4
500	16.8 ± 4.6	23.6 ± 14.7	44.0 ± 10.2
750	16.0 ± 2.9	61.4 ± 14.4	91.8 ± 13.0
1000	38.6 ± 45.2	154.0 ± 38.7	165.0 ± 43.1

Tabella 7.1: Dati raccolti per allerta Terremoto con capacità di archi e nodi limitata, simulata alle ore 10

Parallelamente, i tempi di ricezione dell'ultimo percorso e del segnale di stop aumentano in modo progressivo con l'incremento degli utenti, con un'accelerazione notevole oltre i 750 utenti. Questo incremento non rappresenta un fallimento, ma un comportamento atteso in sistemi complessi: l'aumento del numero di agenti da tracciare e coordinare porta inevitabilmente a un allungamento dei tempi di completamento del processo. Questo

trend dimostra che i microservizi mantengono la loro funzionalità anche a fronte di un carico massiccio, completando l'evacuazione di tutti gli utenti, seppur con tempistiche più estese.

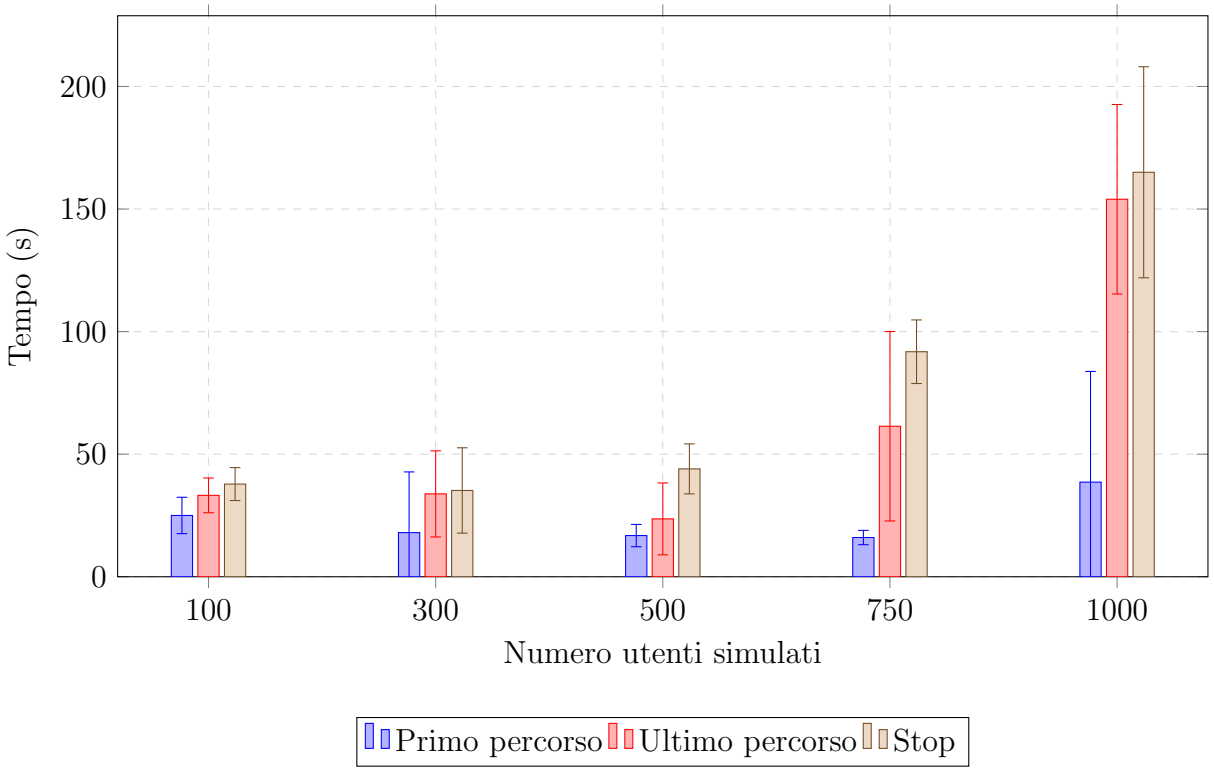

Figura 7.18: Confronto variazione tempi di ricezione al variare del numero di utenti simulati

La Figura 7.19 illustra le curve di utenti salvati nel tempo per i diversi livelli di carico, offrendo una rappresentazione visiva diretta dell'andamento del processo di evacuazione.

- Per carichi inferiori o uguali a 500 utenti, le curve mostrano una crescita estremamente rapida e quasi verticale, raggiungendo il 100% degli utenti salvati in un intervallo di tempo molto breve. Questo andamento a "S" schiacciata è tipico di sistemi ad alta efficienza dove i microservizi di simulazione gestiscono il movimento degli agenti con un'efficienza ottimale, senza che si verifichino colli di bottiglia o ritardi significativi. La ripidità della curva evidenzia la capacità del sistema di processare rapidamente un elevato numero di agenti e di aggiornare in tempo reale il loro stato di sicurezza.

- Con l'aumento del carico a 750 e 1000 utenti, le curve di evacuazione diventano progressivamente meno ripide. L'evacuazione non è più istantanea, ma si distribuisce su un arco di tempo più lungo. Questo cambiamento nella pendenza non indica un fallimento, ma piuttosto un comportamento di saturazione previsto in scenari ad

alta densità. I microservizi di simulazione e di gestione del database lavorano sotto stress crescente, gestendo un numero maggiore di collisioni tra gli agenti e un flusso di dati più intenso. La pendenza meno accentuata è la diretta conseguenza di questa gestione adattiva che, pur rallentando il processo, assicura che il 100% degli utenti raggiunga la salvezza.

In sintesi, l'analisi delle curve dimostra la resilienza del sistema: pur in condizioni estreme, dove il numero di utenti supera la soglia ottimale di performance, la piattaforma continua a garantire il successo dell'evacuazione per tutta la popolazione, sacrificando la rapidità a favore della robustezza.

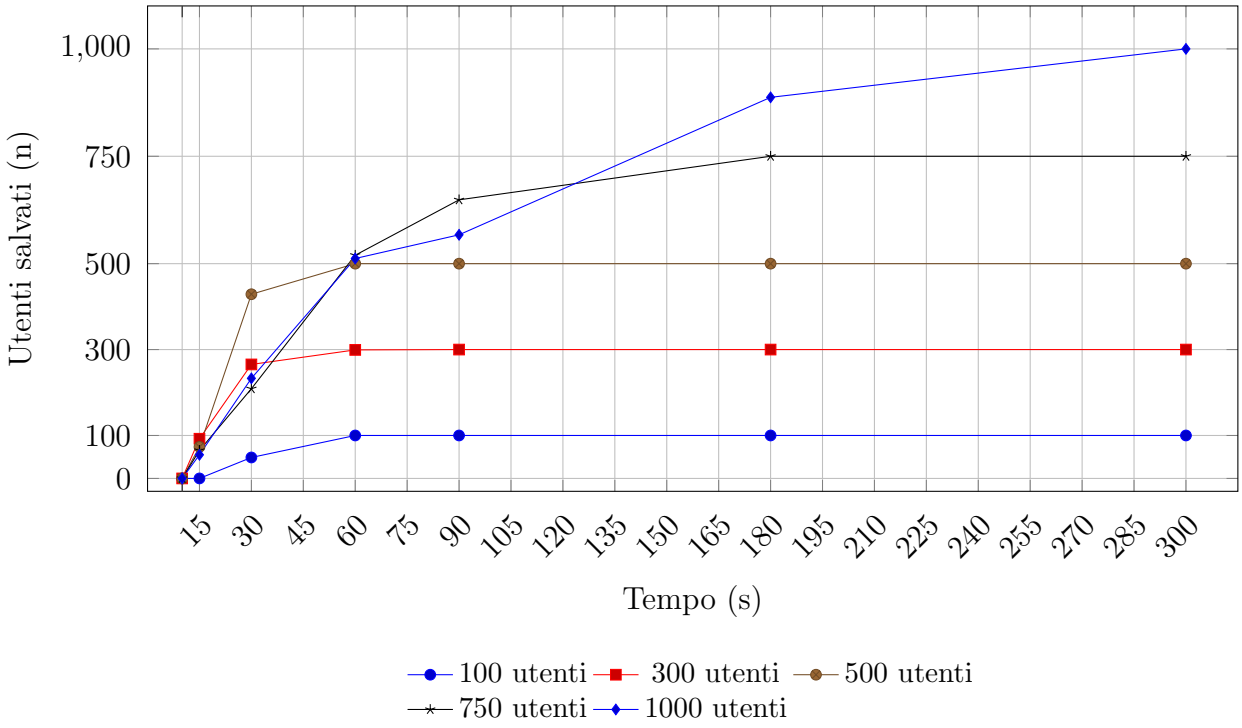

Figura 7.19: Utenti salvati nel tempo — Terremoto, capacità limitata, ore 10.

La Tabella 7.2 e la Figura 7.20 offrono una visione quantitativa di questo fenomeno, concentrandosi sul throughput e sul latency gap.

- Il throughput, che misura l'efficienza del flusso di persone, mostra un comportamento interessante. Raggiunge il suo massimo con 500 utenti (11.1 utenti/s), superando notevolmente i carichi inferiori. Questo suggerisce un'ottimizzazione dei microservizi a carichi intermedi, probabilmente dovuta a una migliore gestione delle risorse. Oltre questa soglia, il throughput si riduce progressivamente (8.2 utenti/s a 750 utenti, 6.1 a 1000 utenti), ma rimane su valori elevati. Questa riduzione non è un segno di inefficienza, ma riflette l'aumento delle collisioni e dei ritardi inevitabili in un ambiente sovraffollato, che i microservizi gestiscono in modo adattivo per garantire che nessun agente venga lasciato indietro.

- Il latency gap, che riflette l'intervallo di tempo durante il quale il sistema continua a inviare percorsi di evacuazione, mostra un trend di crescita non lineare, con un balzo notevole in corrispondenza del passaggio da 500 a 750 utenti (da 6.8 s a 45.4 s) e da 750 a 1000 utenti (da 45.4 s a 115.6 s). Questo aumento è una diretta conseguenza del lavoro aggiuntivo che i microservizi devono svolgere: l'aumento del carico estende il tempo necessario per calcolare e inviare tutti i percorsi, ma il sistema mantiene la sua funzionalità, dimostrando che la capacità di risposta è resiliente anche in condizioni di forte stress.

Utenti simulati	Evacuazione totale (s)	Throughput (utenti/s)	Latency Gap (s)	Successo (%)
100	37.8	2.6	8.2	100
300	35.2	8.5	15.8	100
500	44.0	11.1	6.8	100
750	142.8	8.2	45.4	100
1000	73.8	6.1	115.6	100

Tabella 7.2: Riepilogo delle metriche di performance nello scenario Terremoto con capacità limitata e simulazione alle ore 10

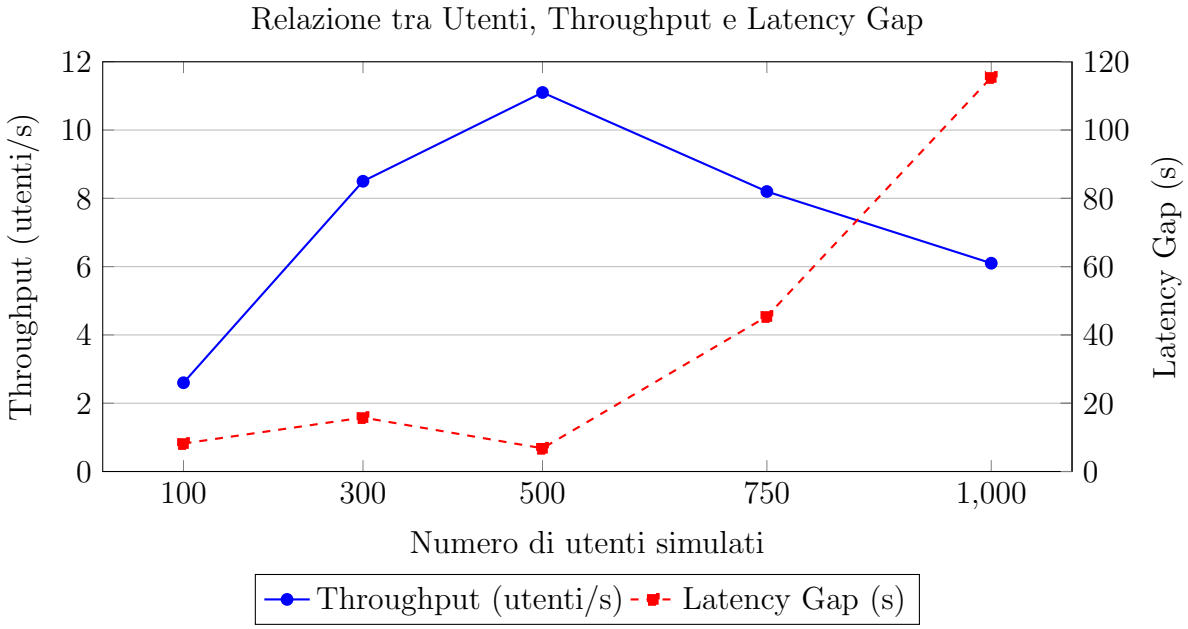

Figura 7.20: Confronto tra Throughput e Latency Gap al variare del numero di utenti a rischio nello scenario Terremoto con capacità archi e nodi limitata e simulazione alle ore 10.

L'analisi dei dati in questo scenario dimostra che la piattaforma possiede un'ottima scalabilità fino a 500 utenti, mostrando anzi un'efficienza ottimale. Oltre questa soglia,

il sistema dimostra una notevole robustezza, affrontando l'aumento del carico non con un collasso, ma con un comportamento adattivo che privilegia la completa evacuazione di tutti gli utenti. Il calo del throughput e l'allungamento dei tempi in scenari estremi non sono un fallimento, ma una dimostrazione di come i microservizi di simulazione e di gestione degli utenti siano stati progettati per gestire e superare le sfide del carico computazionale, garantendo sempre l'obiettivo principale del progetto. Questo comportamento resiliente è un dato cruciale per la futura calibrazione e ottimizzazione del sistema in contesti di elevata densità di popolazione.

7.3.2 Impatto del contesto spaziale e temporale

Questa sezione esplora come la variazione della distribuzione iniziale degli utenti all'interno dell'edificio influenzi le prestazioni della piattaforma. Le fasce orarie simulate, descritte in dettaglio nel Capitolo 5, riflettono scenari realistici di occupazione, in cui il simulatore distribuisce gli utenti in aree specifiche in base a percentuali predefinite (es. aule, laboratori, uffici). L'analisi si basa su uno scenario ad alto carico (1000 utenti) con allerta Terremoto e capacità del grafo limitata, per valutare la robustezza dei microservizi di simulazione e gestione degli utenti in contesti di evacuazione complessi.

La Tabella 7.3 e il relativo istogramma (Figura 7.21) presentano i tempi medi e la deviazione standard per i principali eventi di notifica. Si nota una notevole variazione nelle performance tra le diverse fasce orarie. La fascia 15:15-16:00, che rappresenta la pausa tra le lezioni, mostra i tempi più brevi per la ricezione dell'ultimo percorso (79.8 s) e del segnale di stop (88.8 s). In questo scenario, gli utenti sono più distribuiti o meno concentrati in zone a rischio, il che permette ai microservizi di simulazione di calcolare i percorsi e di tracciare gli agenti in modo più efficiente. Al contrario, le fasce 13:00-14:00 (Pausa Pranzo) e 16:00-18:00 (Lezione Pomeriggio) mostrano i tempi di evacuazione più lunghi e, in particolare, una deviazione standard molto elevata ($\sigma > 100 s¿$). Questo suggerisce che la distribuzione degli utenti in queste fasce orarie crea una maggiore variabilità nel processo di evacuazione, probabilmente a causa di una concentrazione in aree con capacità di evacuazione limitata. Questo comportamento è un'indicazione diretta delle sfide che i microservizi di simulazione affrontano nel gestire la complessità spaziale e dinamica del sistema.

Terremoto con capacità archi e nodi limitata simulata per 1000 utenti			
Fascia oraria	**Ricezione primo path (Δt)**	**Ricezione ultimo path (Δt)**	**Ricezione Stop (Δt)**
8:30-10:30	38.6 ± 45.2	154.0 ± 38.7	165.0 ± 43.1
13:00-14:00	17.4 ± 6.2	173.2 ± 131.4	182.0 ± 132.1
15:15-16:00	22.6 ± 23.0	79.8 ± 18.3	88.8 ± 14.2
16:00-18:00	36.8 ± 44.3	181.6 ± 110.3	189.6 ± 108.1

Tabella 7.3: Dati raccolti per allerta Terremoto con capacità di archi e nodi limitata, simulata per 1000 utenti

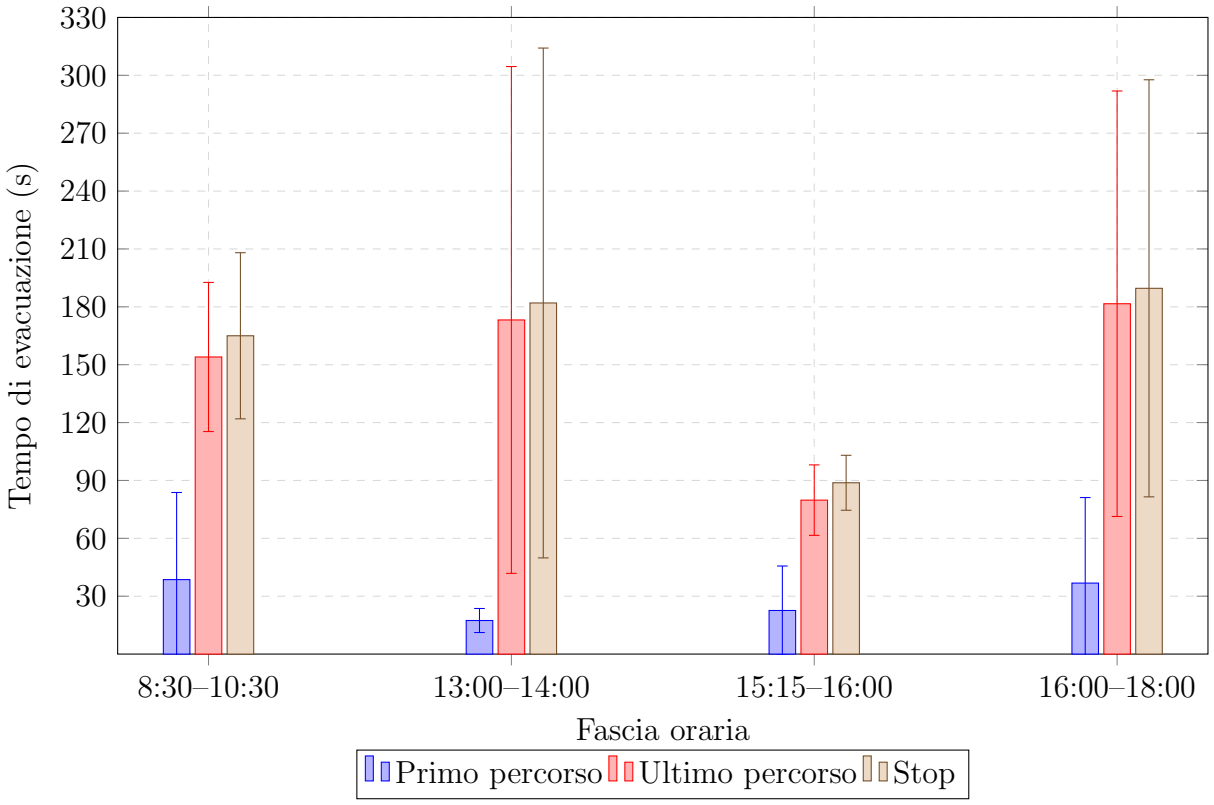

Figura 7.21: Confronto variazione tempi di ricezione al variare della fascia oraria di simulazione

La Figura 7.22 illustra le curve di utenti salvati nel tempo, evidenziando chiaramente le differenze di efficienza in base alla fascia oraria. Tutte le curve, come atteso, raggiungono il successo del 100% dell'evacuazione, dimostrando la resilienza del sistema. Tuttavia, la pendenza delle curve varia in modo significativo. La fascia 15:15-16:00 mostra una pendenza molto più ripida, raggiungendo rapidamente il completamento dell'evacuazione, a conferma della sua maggiore efficienza in termini di tempo. Le altre fasce mostrano pendenze più graduali, indicando che la gestione degli utenti è più complessa e richiede più tempo.

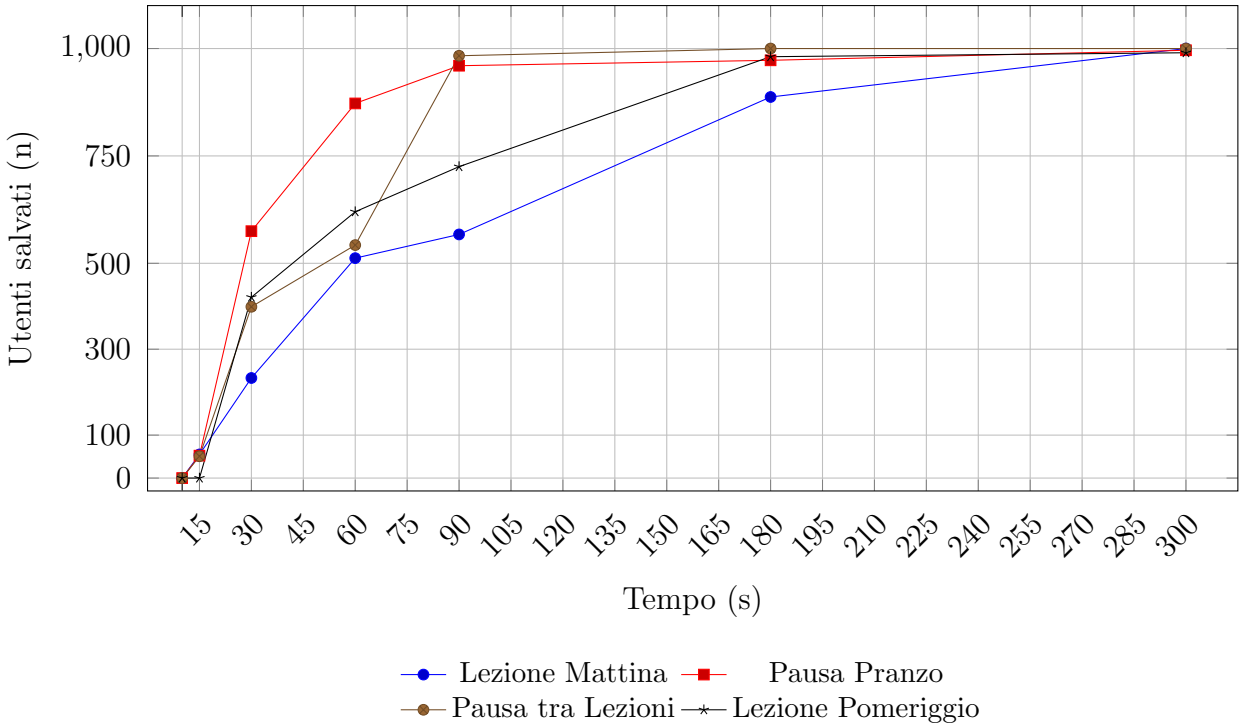

Figura 7.22: Utenti salvati nel tempo — Terremoto, capacità limitata, 1000 utenti; confronto tra quattro fasce orarie.

La Tabella 7.4 e la Figura 7.23 offrono una visione quantitativa di questo fenomeno, concentrandosi sul throughput e sul latency gap.

- Il throughput raggiunge il suo massimo nella fascia 15:15-16:00 (11.3 utenti/s), confermando che la distribuzione degli utenti in questo intervallo di tempo ottimizza l'efficienza del flusso di evacuazione. Al contrario, il throughput è inferiore nelle altre fasce, specialmente nella fascia 16:00-18:00 (5.3 utenti/s), dove la concentrazione degli utenti probabilmente crea colli di bottiglia che i microservizi devono gestire in modo adattivo.

- Il latency gap mostra una correlazione inversa con il throughput. La fascia con il throughput più alto (15:15-16:00) ha il latency gap più basso (57.2 s), indicando che il sistema completa la fase di invio dei percorsi di evacuazione in un tempo più breve. Le fasce con un throughput più basso, come 13:00-14:00 (155.8 s) e 16:00-18:00 (144.8 s), presentano un latency gap significativamente più elevato. Questo suggerisce che una distribuzione degli utenti in aree congestionate o complesse impegna i microservizi per un periodo di tempo più lungo nel calcolo e nell'invio dei percorsi, pur mantenendo la loro funzionalità.

Fascia oraria	Evacuazione totale (s)	Throughput (utenti/s)	Latency Gap (s)	Successo (%)
8:30-10:30	165.0	6.1	115.4	100
13:00-14:00	182.0	5.5	155.8	100
15:15-16:00	88.8	11.3	57.2	100
16:00-18:00	189.6	5.3	144.8	100

Tabella 7.4: Riepilogo delle metriche di performance nello scenario Terremoto con capacità limitata e numero utenti a 1000

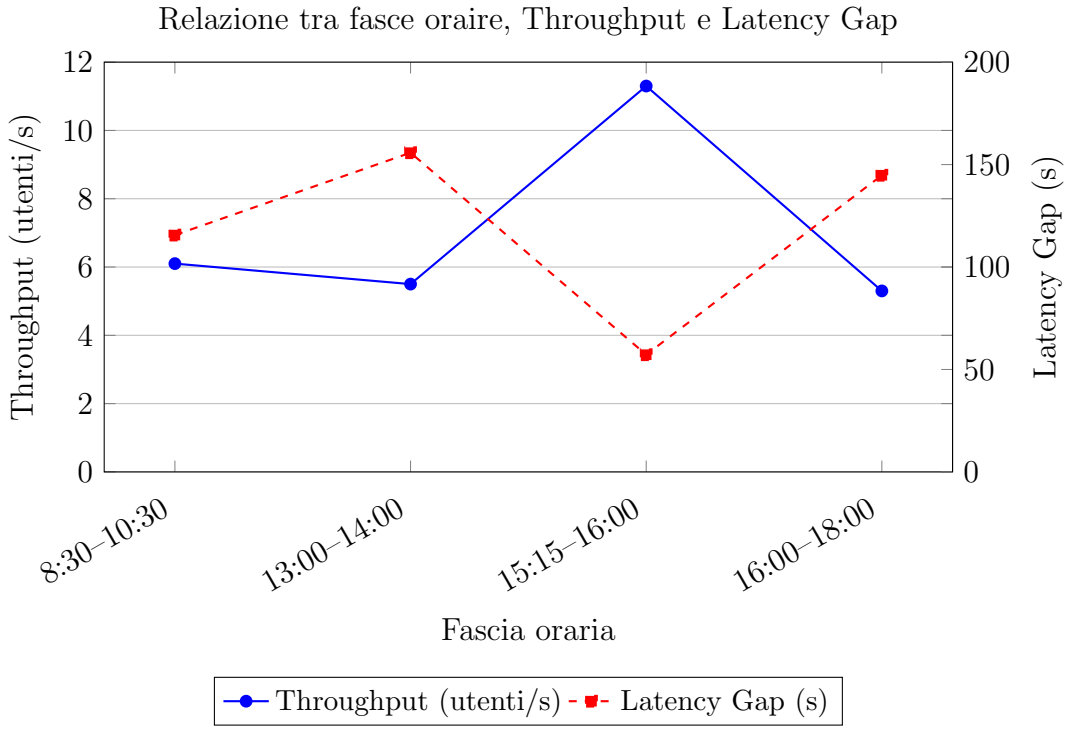

Figura 7.23: Confronto tra Throughput e Latency Gap nelle diverse fasce orarie (scenario Terremoto, capacità limitata, 1000 utenti).

L'analisi dimostra che il contesto spaziale e temporale dell'allerta ha un impatto diretto sulle performance della piattaforma. La distribuzione degli utenti che caratterizza la fascia 15:15-16:00 risulta essere la più efficiente per il processo di evacuazione, grazie a una gestione ottimizzata dei flussi da parte dei microservizi. Al contrario, le distribuzioni delle altre fasce orarie, pur rendendo il processo più lungo, non compromettono la funzionalità del sistema. I risultati evidenziano la robustezza dei microservizi di simulazione e gestione degli utenti che, anche in scenari complessi con una distribuzione non ottimale, garantiscono l'evacuazione del 100% degli agenti.

7.3.3 Impatto della tipologia allerta

Questa sezione analizza l'influenza della tipologia di allerta sulle prestazioni della piattaforma, confrontando uno scenario di evacuazione totale (Terremoto) con uno di evacuazione parziale (Alluvione). In entrambi i casi, la simulazione mantiene le condizioni di alto carico (1000 utenti), capacità del grafo limitata e fascia oraria 10:00, permettendo di isolare l'impatto della natura dell'allerta. Questa comparazione è fondamentale per valutare la versatilità dei microservizi di simulazione e gestione degli utenti nel gestire processi di evacuazione con obiettivi diversi.

La Tabella 7.5 e il relativo istogramma (Figura 7.24) mostrano i tempi medi e la deviazione standard per i principali eventi di notifica. I tempi di ricezione del primo percorso per l'allerta Alluvione sono significativamente più rapidi e stabili ($21.2 \pm 7.5s$) rispetto a quelli per il Terremoto ($38.6 \pm 45.2s$). Questa differenza è un indicatore diretto dell'efficienza del microservizio di gestione degli utenti, che nel caso dell'alluvione deve identificare e filtrare solo la popolazione a rischio, escludendo quella già in zone sicure. Tale operazione, seppur complessa, porta a un avvio del processo di evacuazione più snello.

Al contrario, i tempi di ricezione dell'ultimo percorso e del segnale di stop risultano più lunghi per lo scenario Alluvione (191.0 s e 202.8 s, rispettivamente) rispetto a quelli del Terremoto (154.0 s e 165.0 s). Questo andamento, sebbene controintuitivo, è spiegabile. L'evacuazione parziale comporta che gli utenti a rischio si trovino in aree specifiche dell'edificio, spesso caratterizzate da una maggiore complessità del percorso o da un accesso limitato ai punti di evacuazione. I microservizi di simulazione si trovano quindi a gestire una congestione più localizzata e intensa, che rallenta il processo di completamento dell'evacuazione, nonostante il minor numero totale di utenti da gestire.

Simulazione ore 10 con capacità archi e nodi limitata e 1000 utenti			
Tipologia allerta	**Ricezione primo path (Δt)**	**Ricezione ultimo path (Δt)**	**Ricezione Stop (Δt)**
Terremoto	38.6 ± 45.2	154.0 ± 38.7	165.0 ± 43.1
Alluvione	21.2 ± 7.5	191.0 ± 61.5	202.8 ± 60.2

Tabella 7.5: Dati raccolti per simulazioni alle ore 10 con capacità di archi e nodi limitata e 1000 utenti simulati

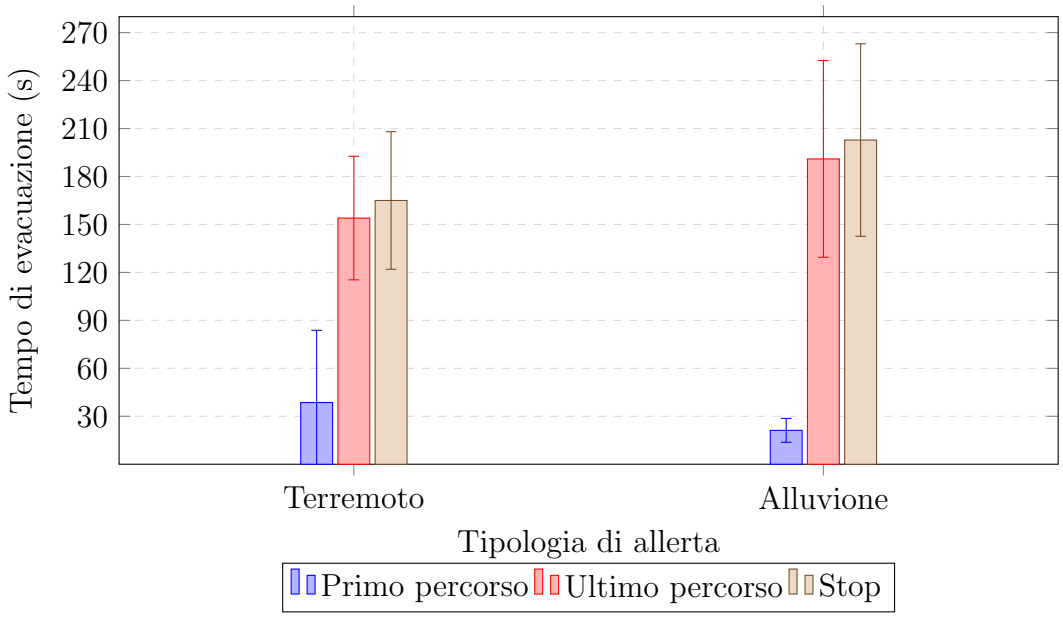

Figura 7.24: Confronto variazione tempi di ricezione al variare della tipologia di allerta

La Figura 7.25 illustra le curve di utenti salvati nel tempo. Come atteso, entrambe le simulazioni portano a un successo del 100% nell'evacuazione, dimostrando la resilienza del sistema in entrambi gli scenari. Tuttavia, la curva dell'Alluvione mostra una pendenza complessiva più graduale e un tempo di completamento più lungo, confermando che, nonostante il minor numero di utenti, la loro specifica distribuzione spaziale rende l'evacuazione più complessa.

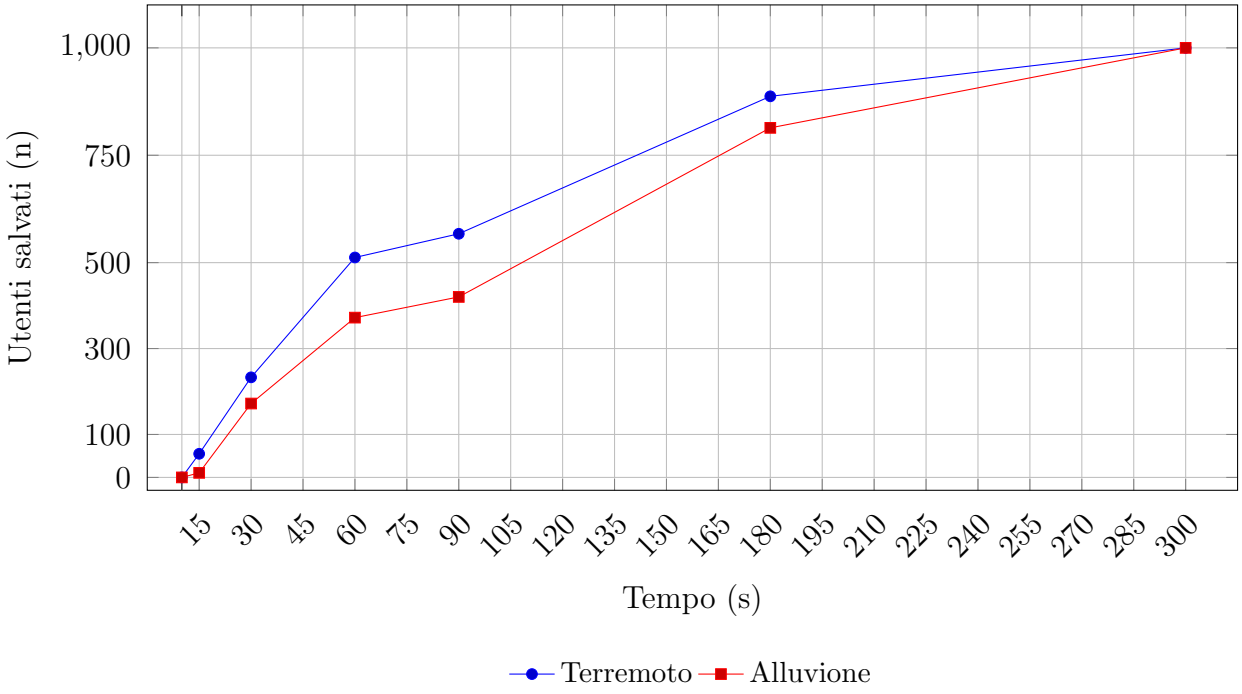

Figura 7.25: Utenti salvati nel tempo — Ore 10, capacità limitata, 1000 utenti; confronto tra allerta terremoto e alluvione.

La Tabella 7.6 e la Figura 7.26 offrono una visione quantitativa di questo fenomeno, concentrandosi sul throughput e sul latency gap.

- Il throughput nello scenario Alluvione (4.9 utenti/s) è inferiore a quello del Terremoto (6.1 utenti/s). Questo dato conferma l'ipotesi che la congestione localizzata, tipica di un'evacuazione parziale, riduca la velocità complessiva del flusso. I microservizi di simulazione si trovano a dover gestire un numero elevato di utenti in un'area ristretta, portando a un rallentamento del processo di evacuazione rispetto a un'evacuazione totale, dove il carico è distribuito su un'area più vasta.
- Il latency gap per l'allerta Alluvione (169.8 s) è notevolmente superiore a quello del Terremoto (115.4 s). Questo risultato è particolarmente rilevante: il sistema impiega un tempo significativamente più lungo per completare l'invio dei percorsi di evacuazione, pur gestendo un minor numero di utenti. Questo comportamento è una diretta conseguenza della complessità del calcolo dei percorsi in un ambiente parzialmente inagibile, che richiede ai microservizi una maggiore elaborazione e un tempo esteso per garantire la sicurezza di ogni singolo agente a rischio.

Tipologia allerta	Evacuazione totale (s)	Throughput (utenti/s)	Latency Gap (s)	Successo (%)
Terremoto	165.0	6.1	115.4	100
Alluvione	202.8	4.9	169.8	100

Tabella 7.6: Riepilogo delle metriche di performance con capacità archi e nodi limitata, simulazione di 1000 utenti alle ore 10

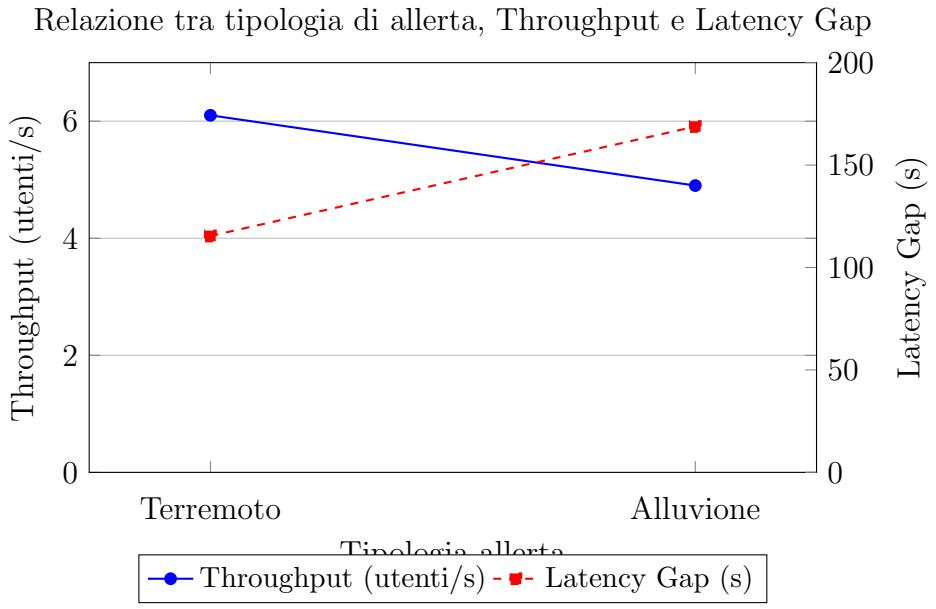

Figura 7.26: Confronto tra Throughput e Latency Gap negli scenari Terremoto e Alluvione con capacità archi e nodi limitata e 1000 utenti.

L'analisi comparativa tra i due tipi di allerta rivela che, sebbene un'evacuazione parziale (Alluvione) possa sembrare più semplice, la sua implementazione pratica presenta sfide specifiche legate alla distribuzione spaziale degli utenti a rischio. La piattaforma dimostra una notevole robustezza in entrambi gli scenari, garantendo sempre il successo dell'evacuazione. Tuttavia, le performance in termini di tempi e throughput evidenziano che la tipologia di allerta e la conseguente distribuzione degli utenti hanno un impatto diretto sulla complessità della simulazione. I microservizi gestiscono in modo efficace queste sfide, ma il processo per un'allerta parziale risulta più lungo e meno efficiente a causa della congestione localizzata.

7.3.4 Impatto della capacità del grafo (archi e nodi)

Questa sezione confronta le prestazioni della piattaforma in due scenari distinti che si differenziano unicamente per la capacità degli archi e dei nodi del grafo che rappresenta la mappa dell'edificio. In un caso, la capacità è limitata e calcolata in base alle dimensioni reali delle aree dell'edificio, mentre nel secondo caso è considerata infinita, eliminando i vincoli di congestione. L'esperimento mantiene costanti le altre condizioni (allerta Terremoto, 1000 utenti simulati, fascia oraria 10:00). L'analisi è fondamentale per isolare l'impatto della congestione sul processo di evacuazione e per dimostrare come i microservizi di simulazione del movimento e di gestione dello stato affrontino tali sfide.

La Tabella 7.7 e il relativo istogramma (Figura 7.27) mostrano una netta superiorità delle performance nello scenario con capacità illimitata. I tempi di ricezione dell'ultimo percorso e del segnale di stop sono drasticamente ridotti: il tempo di evacuazione totale scende da 165.0 s a 77.0 s. Questa riduzione di oltre il 50% evidenzia l'effetto critico dei colli di bottiglia causati dalla capacità limitata. I microservizi di simulazione, dovendo gestire la congestione, rallentano il movimento degli agenti per evitare che si sovrappongano, portando a tempi di evacuazione più lunghi.

Nello scenario a capacità illimitata, la deviazione standard per tutti i tempi di ricezione è significativamente più bassa, attestandosi tra i 5 e i 6 secondi, a differenza dello scenario limitato dove raggiunge valori superiori a 40 secondi. Questo dato dimostra che un ambiente non soggetto a congestione permette ai microservizi di operare in modo molto più stabile e prevedibile, con una variabilità minima tra le diverse simulazioni.

Terremoto con 1000 utenti simulata alle ore 10			
Capacità archi e nodi	**Ricezione primo path (Δt)**	**Ricezione ultimo path (Δt)**	**Ricezione Stop (Δt)**
Limitata	38.6 ± 45.2	154.0 ± 38.7	165.0 ± 43.1
Infinita	21.6 ± 5.7	72.0 ± 5.9	77.0 ± 6.1

Tabella 7.7: Dati raccolti per allerta Terremoto, simulazioni alle ore 10 con 1000 utenti simulati

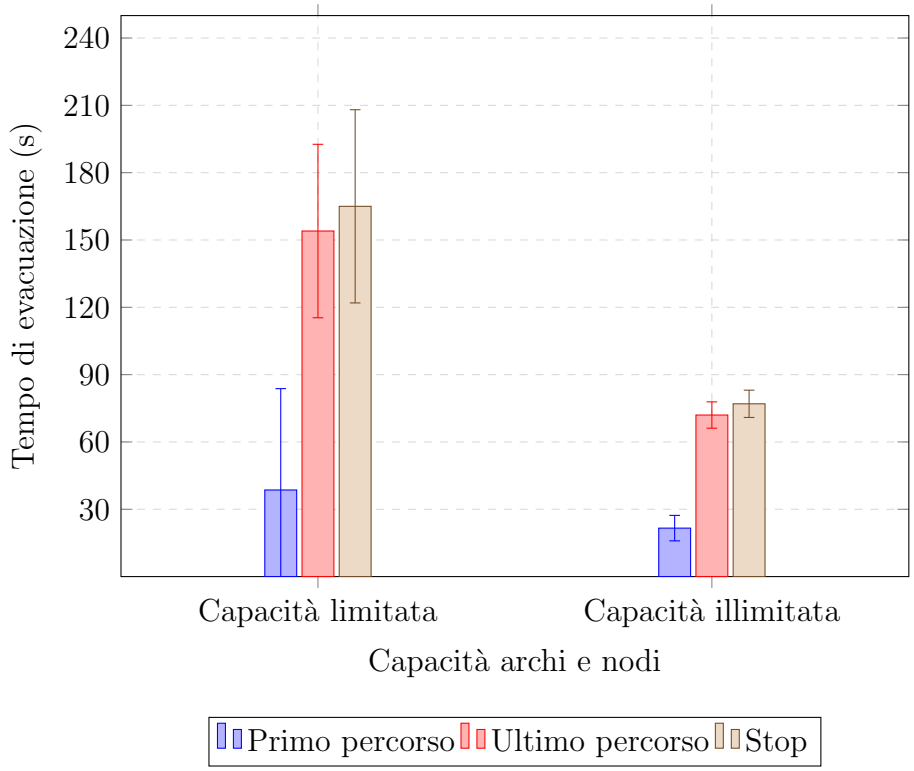

Figura 7.27: Confronto variazione tempi di ricezione al variare della capacità degli archi e dei nodi

La Figura 7.28 illustra le curve di utenti salvati nel tempo. Nello scenario a capacità illimitata, la curva mostra una crescita quasi istantanea, raggiungendo il 100% degli utenti salvati in un tempo molto più breve rispetto allo scenario a capacità limitata. Questo andamento evidenzia che, senza i vincoli fisici, il processo di evacuazione è notevolmente più rapido ed efficiente, un risultato atteso e che conferma il modello simulativo.

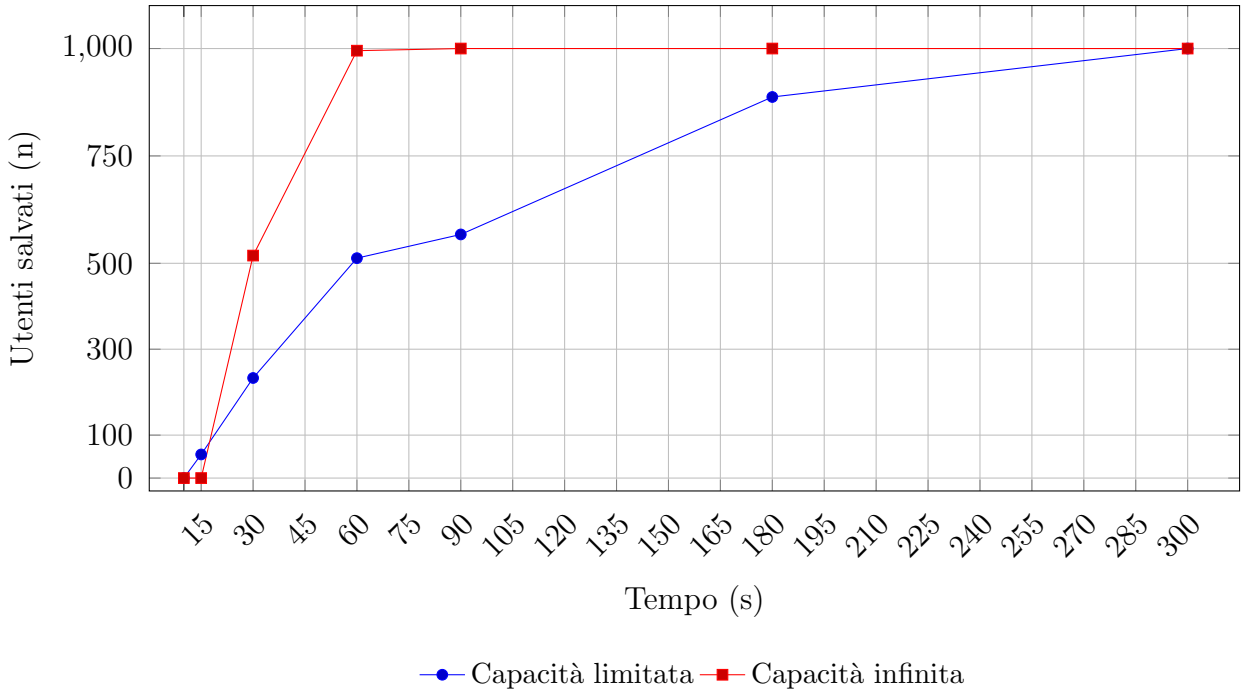

Figura 7.28: Utenti salvati nel tempo — Terremoto, ore 10, 1000 utenti; confronto tra capacità limitata e infinita.

La Tabella 7.8 e la Figura 7.29 offrono una visione quantitativa di questo fenomeno, concentrandosi sul throughput e sul latency gap.

- Il throughput nello scenario a capacità illimitata (13.0 utenti/s) è più che doppio rispetto allo scenario a capacità limitata (6.1 utenti/s). Questo dimostra in modo inequivocabile come i vincoli fisici del grafo abbiano un impatto diretto sulla velocità del flusso di evacuazione. Il throughput maggiore nello scenario illimitato riflette la capacità del simulatore di muovere gli agenti senza dover considerare le collisioni o la saturazione dei percorsi.

- Il latency gap nello scenario a capacità illimitata (50.4 s) è significativamente più basso rispetto allo scenario a capacità limitata (115.4 s). Questo dato indica che il microservizio di simulazione completa l'invio dei percorsi di evacuazione in un tempo notevolmente inferiore quando non deve gestire la complessità derivante dalla capacità limitata. Il sistema termina il suo lavoro in modo più rapido ed efficiente.

Capacità archi e nodi	Evacuazione totale (s)	Throughput (utenti/s)	Latency Gap (s)	Successo (%)
Limitata	165.0	6.1	115.4	100
Illimitata	77.0	13.0	50.4	100

Tabella 7.8: Riepilogo delle metriche di performance per allerta di tipo Terremoto, simulazione di 1000 utenti alle ore 10

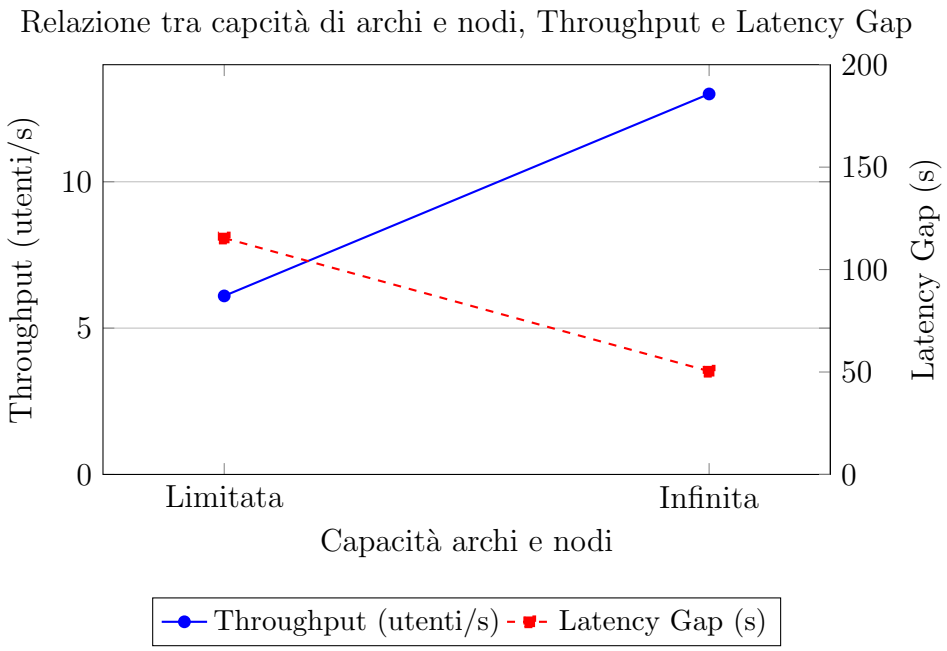

Figura 7.29: Confronto tra Throughput e Latency Gap per allerta Terremoto e simulazione di 1000 utenti alle ore 10

L'analisi comparativa tra i due scenari dimostra che la capacità degli archi e dei nodi è un fattore critico che influenza direttamente le performance del sistema. La presenza di vincoli fisici, sebbene rallenti il processo di evacuazione e ne aumenti la variabilità, non compromette l'obiettivo finale di salvare tutti gli utenti. I microservizi di simulazione e gestione degli utenti dimostrano la loro robustezza e la capacità di operare in condizioni di congestione, gestendo in modo efficace il flusso di persone e garantendo il completamento del processo. Lo scenario a capacità illimitata serve come benchmark teorico, dimostrando il potenziale massimo di efficienza del sistema in assenza di vincoli spaziali.

7.4 Riepilogo della validazione quantitativa

L'analisi quantitativa è stata condotta per validare la robustezza e l'affidabilità dell'intera piattaforma in scenari dinamici e complessi. I risultati complessivi hanno confermato che il sistema è in grado di garantire l'obiettivo primario del progetto: l'evacuazione del 100% degli utenti a rischio in ogni condizione simulata.

I test condotti si sono concentrati in particolare sulla performance dei microservizi di simulazione del movimento e di gestione dello stato degli utenti, per i quali la validazione ha evidenziato tre risultati principali:

- Scalabilità e adattabilità: i microservizi dimostrano una scalabilità ottimale fino a 500 utenti, mantenendo un'elevata efficienza. Oltre questa soglia, la loro capacità

di adattamento permette di gestire efficacemente l'aumento del carico, garantendo il completamento dell'evacuazione nonostante il naturale incremento dei tempi di processo.

- Gestione del contesto spaziale: la loro efficienza varia in base alla distribuzione iniziale degli utenti. Sono particolarmente performanti in scenari con flussi di persone più uniformi, mentre in contesti di congestione localizzata, come quelli dovuti a un'evacuazione parziale, gestiscono in modo efficace i colli di bottiglia, sebbene ciò si traduca in un aumento del latency gap e del tempo totale di evacuazione.

- Robustezza in presenza di vincoli: l'analisi ha isolato il ruolo fondamentale dei vincoli fisici del grafo. I microservizi di simulazione riescono a operare con successo anche in presenza di capacità limitate, dimostrando la loro robustezza nel gestire le sfide del mondo reale. Lo scenario a capacità infinita ha fornito un benchmark teorico che ha evidenziato come l'implementazione del simulatore sia in grado di sfruttare al massimo le risorse disponibili.

In sintesi, i dati raccolti confermano che la parte del sistema dedicata alla simulazione e alla gestione degli utenti è non solo efficiente, ma anche resiliente, fornendo una base solida e affidabile per la pianificazione di emergenze in contesti reali.

Conclusioni

L'obiettivo principale di questo lavoro di tesi è stato quello di affrontare le sfide e le inefficienze intrinseche dei sistemi di gestione delle emergenze, sia a livello territoriale che, in modo più specifico, all'interno di contesti edilizi complessi. L'analisi dello stato dell'arte ha evidenziato come le soluzioni esistenti, come IT-Alert e IPAWS, pur rappresentando un notevole passo avanti nella comunicazione su vasta scala, si concentrino principalmente sulla notifica passiva del pericolo, trascurando la necessità di una risposta dinamica e automatizzata che si adatti in tempo reale all'evolversi dell'evento.

La lacuna principale individuata è stata l'assenza di un sistema in grado di orchestrare una reazione attiva a un'emergenza. I protocolli attuali non sono sufficientemente granulari per contesti specifici, non sfruttano la localizzazione precisa all'interno degli edifici e non offrono un ricalcolo dinamico delle vie di fuga in base all'evoluzione del pericolo. Di fronte a queste limitazioni, l'obiettivo di questa ricerca è stato duplice: da un lato, progettare e implementare un'architettura di sistema che superasse queste criticità; dall'altro, dimostrare la validità di un approccio che integra la notifica di allerta con una risposta coordinata e intelligente, mirata a guidare gli occupanti verso la sicurezza nel modo più efficiente possibile.

Per superare le criticità identificate, il presente lavoro ha adottato un'architettura a microservizi, un paradigma moderno che ha permesso di costruire un sistema intrinsecamente modulare, scalabile e resiliente. Tale approccio si è rivelato fondamentale per gestire la complessità di un sistema che deve reagire in modo dinamico a eventi critici in tempo reale. Nello specifico, il contributo principale di questa tesi si è concentrato sulla progettazione e sull'implementazione dei microservizi di simulazione delle posizioni (*User Simulator*) e di gestione delle posizioni (*Position Manager*), componenti cruciali per la validazione e il funzionamento del sistema. Il primo ha rappresentato una soluzione metodologica innovativa, permettendo di emulare il comportamento degli occupanti in assenza di dati reali, mentre il secondo ha garantito la logica di valutazione del rischio e l'aggiornamento dinamico delle posizioni.

L'efficacia dell'architettura proposta è stata validata attraverso un'analisi sperimentale basata su un caso di studio concreto: il Campus universitario di Cesena. Le simulazioni

qualitative, che hanno riprodotto scenari di terremoto e alluvione, hanno dimostrato la capacità del sistema di processare un'allerta in tempo reale e di orchestrare una risposta dinamica, generando percorsi di evacuazione aggiornati in base all'evoluzione del pericolo. Questa fase ha confermato la validità concettuale del nostro approccio e la sua capacità di adattarsi a eventi di natura diversa.

La validazione quantitativa ha rappresentato una fase cruciale, fornendo un benchmark oggettivo delle performance del sistema. I dati hanno dimostrato che il sistema è non solo scalabile fino a 500 utenti, ma anche resiliente in condizioni di stress computazionale, garantendo sempre l'evacuazione del 100% degli utenti. Le analisi hanno inoltre rivelato che il contesto spaziale e la tipologia di allerta hanno un impatto diretto sull'efficienza: la gestione di un'evacuazione parziale, sebbene coinvolga un numero inferiore di utenti, può essere più complessa a causa della congestione localizzata, che i microservizi gestiscono in modo efficace. Il confronto con uno scenario a capacità illimitata ha infine confermato che i vincoli fisici dell'edificio sono il fattore principale che incide sui tempi di evacuazione, pur senza compromettere la funzionalità del sistema.

I risultati raggiunti in questo lavoro dimostrano la validità e l'affidabilità del sistema proposto. Il prototipo ha mostrato la sua capacità di gestire scenari di emergenza complessi, e i dati di validazione hanno permesso di identificare con precisione le aree di miglioramento che aprono a promettenti sviluppi futuri.

Un primo e fondamentale sviluppo riguarda la transizione da un ambiente simulato a un sistema operativo. Il microservizio *User Simulator* dovrà essere integrato con tecnologie di tracciamento in tempo reale, come il Wi-Fi Positioning System (WPS) o l'Ultra-Wideband (UWB), per acquisire la posizione esatta degli utenti. Parallelamente, l'*Alert Manager* dovrà evolvere per connettersi a fonti di allerta ufficiali tramite protocolli standard come il CAP.

Ulteriori ottimizzazioni si concentreranno sul miglioramento dell'interazione con l'utente finale, attraverso l'integrazione di un sistema di notifiche push per l'invio istantaneo di avvisi e percorsi. Infine, l'analisi dei dati ha suggerito la presenza di possibili inefficienze nella comunicazione asincrona, che rallentano la gestione della congestione. I futuri sviluppi si focalizzeranno sull'ottimizzazione del flusso di dati tra i microservizi per migliorare il throughput e la reattività del sistema, con la possibilità di integrare tecniche di analisi predittiva per mitigare i colli di bottiglia in anticipo.

In un'ottica più ampia, l'architettura a microservizi qui proposta è intrinsecamente flessibile e può essere estesa per supportare una vasta gamma di contesti, dalla gestione di eventi di massa alla sicurezza in ambienti industriali complessi, offrendo una base solida per lo sviluppo di sistemi di gestione delle emergenze di nuova generazione.

Appendici

Appendice A:

Listing 7.1: Esempio schematico di messaggio CAP XML

```
<alert xmlns="urn:oasis:names:tc:emergency:cap:1.2">
  <identifier>Test-Alert-001</identifier>
  <sender>example@example.org</sender>
  <sent>2025-04-30T12:00:00+00:00</sent>
  <status>Actual</status>
  <msgType>Alert</msgType>
  <scope>Public</scope>

  <info>
    <category>Env</category>
    <event>Flood</event>
    <urgency>Immediate</urgency>
    <severity>Severe</severity>
    <certainty>Likely</certainty>
    <language>en-US</language>
    <responseType>Shelter</responseType>
    <description>Heavy flooding expected in low-lying areas.</
       description>
    <instruction>Move to higher ground.</instruction>

    <area>
      <areaDesc>Building A</areaDesc>
      <polygon>45.0,9.0 45.0,9.1 45.1,9.1 45.1,9.0 45.0,9.0</
         polygon>
      <altitude>10</altitude>
    </area>
```

```
  </info>

</alert>
```

.

Appendice B:

Listing 7.2: Rappresentazione in dizionario Python del messaggio CAP

```
{
  "identifier": "Test-Alert-001",
  "sender": "example@example.org",
  "sent": "2025-04-30T12:00:00+00:00",
  "status": "Actual",
  "msgType": "Alert",
  "scope": "Public",
  "info": [
    {
      "category": "Env",
      "event": "Flood",
      "urgency": "Immediate",
      "severity": "Severe",
      "certainty": "Likely",
      "language": "en-US",
      "responseType": "Shelter",
      "description": "Heavy flooding expected in low-lying areas
      .",
      "instruction": "Move to higher ground.",
      "areas": [
        {
          "areaDesc": "Building A",
          "polygon": "45.0,9.0 45.0,9.1 45.1,9.1 45.1,9.0
          45.0,9.0",
          "altitude": "10",
          "geom": {
            "type": "Polygon",
            "coordinates": [
```

```
                [
                  [9.0, 45.0],
                  [9.1, 45.0],
                  [9.1, 45.1],
                  [9.0, 45.1],
                  [9.0, 45.0]
                ]
              ]
            },
            "geometry_type": "Polygon"
          }
        ]
      }
    ]
}
```

Appendice C:

Listing 7.3: Esempio di configurazione del filtro alert

```
cap_filter:
  event:
    - "Fire"
    - "Earthquake"
    - "Flood"
    - "Hazardous Material"
    - "Severe Weather"
    - "Power Outage"
  urgency:
    - "Immediate"
    - "Expected"
    - "Future"
  severity:
    - "Extreme"
    - "Severe"
    - "Moderate"
    - "Minor"
  certainty:
```

```
    - "Observed"
    - "Likely"
  area:
    - "Building␣A"
    - "Parking␣Lot"
    - "Surrounding␣Area"
  responseType:
    - "Shelter"
    - "Evacuate"
    - "Prepare"
    - "Monitor"
    - "AllClear"
  status:
    - "Actual"
    - "Exercise"
    - "System"
    - "Test"
  msgType:
    - "Alert"
    - "Update"
    - "Cancel"
  scope:
    - "Public"
    - "Restricted"
    - "Private"
```

Appendice D:

Listing 7.4: Esempio di configurazione del simulatore utenti

```
rabbitmq:
  host: "localhost"
  port: 5672
  username: "guest"
  password: "guest"
  alert_queue: "user_simulator_queue"
  evacuation_paths_queue: "evacuation_paths_queue"
  position_queue: "position_queue"
```

```
simulation_mode: "from_scratch"      # "from_scratch" | "from_file
    "
user_file: "UserSimulator/config/current_position.csv"
alert_event_type: "Earthquake"

n_users: 750   message
speed_normal: 20.0
speed_alert: 350.0
simulation_tick: 2.0
timeout_after_stop: 60


time_slots:

  - name: "morning_class_1"
    start: "08:30"
    end: "10:30"
    distribution:                                  across locations
      classroom: 0.6
      corridor: 0.1
      coffee shop: 0.05
      canteen: 0.05
      office: 0.1
      bathroom: 0.05
      stairs: 0.03
      outdoor: 0.02

  - name: "morning_break"
    start: "10:30"
    end: "11:00"
    distribution:
      classroom: 0.05
      corridor: 0.15
      coffee shop: 0.3
      canteen: 0.3
      office: 0.05
      bathroom: 0.1
      stairs: 0.03
      outdoor: 0.02
```

```
- name: "morning_class_2"
  start: "11:00"
  end: "13:00"
  distribution:
    classroom: 0.6
    corridor: 0.15
    coffee shop: 0.05
    canteen: 0.05
    office: 0.1
    bathroom: 0.03
    stairs: 0.01
    outdoor: 0.01

- name: "lunch_break"
  start: "13:00"
  end: "14:00"
  distribution:
    classroom: 0.02
    corridor: 0.1
    coffee shop: 0.3
    canteen: 0.4
    office: 0.05
    bathroom: 0.08
    stairs: 0.03
    outdoor: 0.02

- name: "afternoon_class_1"
  start: "14:00"
  end: "15:15"
  distribution:
    classroom: 0.6
    corridor: 0.15
    coffee shop: 0.05
    canteen: 0.05
    office: 0.1
    bathroom: 0.03
    stairs: 0.01
    outdoor: 0.01
```

```
  - name: "afternoon_break"
    start: "15:15"
    end: "16:00"
    distribution:
      classroom: 0.05
      corridor: 0.15
      coffee shop: 0.3
      canteen: 0.3
      office: 0.05
      bathroom: 0.1
      stairs: 0.03
      outdoor: 0.02

  - name: "afternoon_class_2"
    start: "16:00"
    end: "18:00"
    distribution:
      classroom: 0.4
      corridor: 0.1
      coffee shop: 0
      canteen: 0
      office: 0.1
      bathroom: 0.1
      stairs: 0
      outdoor: 0.3
```

Bibliografia

[1] OASIS Standard. Common alerting protocol version 1.2, July 2010. URL http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.pdf. Informazioni utilizzate nel capitolo 1.2.1.

[2] Dipartimento della Protezione Civile. It-alert: il sistema nazionale di allarme pubblico, . URL https://www.protezionecivile.gov.it/it/approfondimento/it-alert-il-sistema-nazionale-di-allarme-pubblico/. Informazioni utilizzate nel capitolo 1.2.2.

[3] UNDRR. Studio del protocollo cap (common alert protocol). URL https://www.undrr.org/early-warnings-for-all/common-alerting-protocol. Informazioni utilizzate nel capitolo 1.2.1.

[4] Federal Emergency Management Agency. Ipaws overview. URL https://www.fema.gov/emergency-managers/practitioners/integrated-public-alert-warning-system. Informazioni utilizzate nel capitolo 1.2.3.

[5] IT-alert. Sito ufficiale di it-alert. URL https://www.it-alert.it/it/. Informazioni utilizzate nel capitolo 1.2.2.

[6] Dipartimento della Protezione Civile. Il sistema di allarme pubblico it-alert in italia, . URL https://www.protezionecivile.gov.it/static/79a4f5e9bb9d804a043f55e29dc4e0a6/all6-it-alert-io-capit.pdf. Informazioni utilizzate nel capitolo 1.2.1 e 1.2.2.

[7] Dipartimento della Protezione Civile. Rappresentazione tabellare del cap italiano e standard italiano, . URL https://www.protezionecivile.gov.it/static/60f148a4a63756446dfb15acab736dbe/all7-dpc-edxl-cap-it-10.pdf. Informazioni utilizzate nel capitolo 1.2.1.

[8] National Oceanic and Atmospheric Administration. Wireless emergency alerts (wea). URL https://www.weather.gov/wrn/wea. Informazioni utilizzate nel capitolo 1.2.3.

[9] Martin Fowler. Microservices. 2014. URL https://martinfowler.com/articles/microservices.html. Definizione e principi dei microservizi, utilizzati nel capitolo 2.1.

[10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. Riferimento utilizzato per la descrizione dell'architettura monolitica nel Capitolo 2.1.

[11] Amazon Web Services. What is microservices architecture? URL https://aws.amazon.com/microservices/. Informazioni utilizzate per il confronto tra architetture nel Capitolo 2.1.

[12] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. Informazioni utilizzate nel capitolo sul confronto tra architetture 2.1.

[13] Python Software Foundation. Python 3.13.0 documentation. URL https://docs.python.org/3/. Documentazione ufficiale di Python utilizzata nel capitolo 3.1.

[14] Node.js. Node.js v20.13.1 documentation. URL https://nodejs.org/en/docs/. Documentazione ufficiale di Node.js (se pertinente) utilizzata nel capitolo 3.1.

[15] Amazon Web Services (AWS). What is a message queue?, . URL https://aws.amazon.com/message-queue/. Informazioni sul pattern Message Queues utilizzate nel capitolo 3.2.

[16] RabbitMQ Team. Rabbitmq documentation. URL https://www.rabbitmq.com/documentation.html. Documentazione ufficiale di RabbitMQ utilizzata nel capitolo 3.2.

[17] Microsoft. Event-driven architecture style - azure architecture center, . URL https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven. Informazioni sul pattern Event-Driven Architecture utilizzate nel capitolo 3.2.

[18] Microsoft. Circuit breaker pattern, . URL https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker. Informazioni sul pattern Circuit Breaker utilizzate nel capitolo 3.2.

[19] Amazon Web Services (AWS). What is dlq? - dead-letter queue explained - aws, . URL https://aws.amazon.com/what-is/dead-letter-queue/. Informazioni sul pattern Dead Letter Queue utilizzate nel capitolo 3.2.

[20] PostgreSQL Global Development Group. Postgresql documentation. URL `https://www.postgresql.org/docs/current/`. Documentazione ufficiale di PostgreSQL utilizzata nella sezione 3.3.

[21] PostGIS Development Team. Postgis documentation. URL `https://postgis.net/documentation/`. Documentazione ufficiale di PostGIS utilizzata nella sezione 3.3.

[22] YAML Ain't Markup Language (YAML) Community. Yaml 1.2 (5th edition) specification. URL `https://yaml.org/spec/1.2/spec.html`. Specifiche ufficiali di YAML utilizzate nel capitolo 3.4.

[23] Prometheus. Prometheus - monitoring system time series database. URL `https://prometheus.io/`. Informazioni generali su Prometheus e Alertmanager utilizzate nel capitolo 4.2.1.

[24] Zabbix. Zabbix - the enterprise-class open source monitoring solution. URL `https://www.zabbix.com/`. Informazioni generali su Zabbix utilizzate nel capitolo 4.2.1.

[25] Nagios Enterprises, LLC. Nagios - the industry standard in it infrastructure monitoring. URL `https://www.nagios.org/`. Informazioni generali su Nagios utilizzate nel capitolo 4.2.1.

[26] Amir Mahdavi, Alireza Khosravi, Ahmed Al-Ani, and Fawaz Al-Saif. A review of interoperability challenges and solutions in smart building systems. *Journal of Building Engineering*, 34:101887, 2021. URL `https://doi.org/10.1016/j.jobe.2020.101887`. Articolo di rassegna sull'interoperabilità nei sistemi di building automation.

[27] ASHRAE. Bacnet - the building automation and control protocol. URL `https://www.bacnet.org/`. Sito ufficiale e fonte della documentazione per il protocollo BACnet.

[28] Modbus Organization. Modbus - the universal industrial protocol. URL `https://modbus.org/`. Sito ufficiale e fonte della documentazione per il protocollo Modbus.

[29] Michael Wooldridge. An introduction to multiagent systems. *John Wiley Sons*, 2009. Testo di riferimento sui sistemi multi-agente e l'Agent-Based Modeling, utile per il contesto teorico.

[30] Sang-Hee Lee, Jun-Jae Lee, and Gye-Yong Cho. A review of evacuation simulation models and their applications. *Journal of the Korean Society of Civil Engineers*, 32 (3A):213–222, 2012. Articolo di rassegna sui modelli di simulazione di evacuazione e le loro applicazioni.

[31] M. Masad and A. Kazil. Mesa: Agent-based modeling in python. *Journal of Open Source Software*, 4(37):1655, 2019. URL `https://joss.theoj.org/papers/10.21105/joss.01655`. Framework di Agent-Based Modeling utilizzato per il confronto nel capitolo 5.2.

[32] Uri Wilensky. Netlogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999. URL `https://ccl.northwestern.edu/netlogo/`. Framework di simulazione agent-based citato per contesto nel capitolo 5.2.

[33] AnyLogic Company. Anylogic simulation software. AnyLogic Company, 2024. URL `https://www.anylogic.com/`. Piattaforma di simulazione integrata citata per contesto nel capitolo 5.2.

[34] Patrick Taillandier, Duong Vo, Antoine Grignard, and Barbara Sgreggia. Gama: A platform for agent-based modeling and simulation. Open-Source Project, 2023. URL `http://gama-platform.org/`. Piattaforma di simulazione integrata citata per contesto nel capitolo 5.2.

[35] NetworkX Developers. Networkx: Documentation. URL `https://networkx.org/documentation/stable/`. Documentazione ufficiale della libreria NetworkX, utilizzata per la gestione e l'analisi dei grafi dell'edificio.

[36] Pygame Community. Welcome to pygame. URL `https://www.pygame.org/docs/`. Libreria per animazione e simulazioni 2D di base considerata nel confronto del capitolo 5.2.

[37] SimPy Developers. Simpy 4.0.1 documentation. URL `https://simpy.readthedocs.io/en/latest/`. Libreria di simulazione di processi discreti utilizzata per il confronto nel capitolo 5.2.

[38] Juan L. Carpio and Alberto J. Arribas. Real-time location systems (rtls) in industry 4.0: A literature review. *Sensors*, 19(17), 2019.

[39] J. Kreps, N. Narkhede, and J. Rao. The log: What every software engineer should know about real-time data's unifying abstraction. *ACM Queue*, 11(5), 2013.

[40] Apache Software Foundation. Apache kafka documentation, . URL `https://kafka.apache.org/documentation/`. Documentazione ufficiale utilizzata per l'analisi dei sistemi di stream processing.

[41] Apache Software Foundation. Apache flink documentation, . URL `https://flink.apache.org/docs/`. Documentazione ufficiale utilizzata per l'analisi dei sistemi di stream processing.

Ringraziamenti

A colei che ha saputo essere roccia e faro, la cui melodia non è sempre stata facile da cogliere, ma che a distanza si è fatta più limpida. A lei che mi ha insegnato che a rimettermi al mio posto è solo la mano di chi mi ama.

A lui, la mia stella nel cielo, la cui assenza ha lasciato un vuoto che si è fatto spazio, ma la cui ombra mi ha protetto, insegnandomi che a volte la strada più spensierata è quella che non ti aspetti. A lui che mi ha mostrato che seconda stella a destra, questo è il cammino.

Ai miei nipoti, rifugio di risate e innocenza. Con loro, ho imparato che anche dopo le giornate più lunghe, la serenità si trova in un abbraccio improvviso e in occhi che vedono la magia in ogni angolo.

Infine, a me stessa, per aver creduto in questo viaggio, anche quando la strada si è fatta ripida. Per aver imparato che il futuro non è scritto, non è scritto da nessuna parte e lo devi scrivere tu.

array, multirow, makecell, tabularx